1886

MASTERING THE 68000™ MICROPROCESSOR

An in-depth analyses of the 16/32 bit
CPU chips used in the Macintosh™, Commodore Amiga,
ATARI® ST series, Apollo® and Imagatex® CAD/CAM machines,
Sinclair® QL, IBM® 9000, and other state-of-the-art micro and mini computers!

PHILLIP R. ROBINSON

MASTERING THE 68000 MICROPROCESSOR

Also by the Author from TAB BOOKS Inc.

1656 *The Programming Guide to the Z80™ Chip*

MASTERING THE 68000 MICROPROCESSOR

PHILLIP R. ROBINSON

TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

FIRST EDITION

FIRST PRINTING

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Robinson, Phillip R.
Mastering the 68000 microprocessor.

On t.p. the registered trademark symbol "TM" is superscript following "6800" in the title.

Includes index.
1. Motorola 68000 (Microprocessor) I. Title.
QA76.8.M6895R63 1985 001.64 85-4666
ISBN 0-8306-0886-9
ISBN 0-8306-1886-4 (pbk.)

Contents

Preface

Most introductions to microprocessors center on 8-bit chips. Because 16-bit chips such as the 68000 microprocessor are powerful, they are often considered too complicated for beginners. I want this book to fight that image and that custom.

The 68000 is far more powerful than the 8-bit chips that were the basis of the first personal computers, but it is also, in many ways, easier to use. This pair of characteristics—power and ease of use—haven't washed away all of the reasons to learn about 8-bit chips. The best chips of the 8-bit generation are still used and programmed in personal computers, controllers, and instrumentation. But for those who want to keep up-to-date, or to catch up on technology, 16-bit chips are the natural choice. They are built into most new microprocessor-based systems including virtually all new personal computers.

The 68000 is no more difficult to program than any of the 8-bit chips. It just has more depth and more capability than those chips. Sure, to get the full use of the 68000 you must understand advanced concepts such as frame pointers, supervisor mode, and memory management, but you don't have to use them.

The 68000 can be used for simple programs just as an 8-bit chip can. But if you have a complicated data structure or routine to program, the 68000 will make your life easier because it provides you with more tools for implementing such things. An 8-bit chip makes you do all the work with long sequences of simple instructions: the 68000 lets you use just a few, more powerful instructions. Many functions that took planning and programming on an 8-bit chip are reduced to a single, automatic operation on the 68000. This process appears in many technologies and is particularly strong in microelectronics. While the chips become more powerful, using them doesn't get more difficult (which is nice because we aren't getting smarter). The elemental functions of the chips just keep advancing. Multiplying 16-bit numbers or setting up a portion of the stack for a subroutine was a major programming task on a 4-bit microprocessor and required a

carefully written subroutine on an 8-bit chip. Doing the same jobs on a 16-bit chip such as the 68000 only requires a single instruction.

Keep in mind that you don't have to use all or even most of the 68000's power. By learning a few instructions and addressing modes, you can begin to write fast, practical, and clean routines and programs.

If you're an old hand in the microprocessor business, you'll know that the 68000 is one of the most widely used 16-bit chips. If you're new to microprocessors, there is a no reason why you shouldn't start with the 68000: it is one of the two most popular 16-bit chips, and you can apply the 68000 concepts you learn to 8-, 16-, or 32-chips.

In fact, the 68000 is more than a 16-bit microprocessor. Many of its features handle 32-bits at a time. The 68000 family—which is also described in this book—is a set of chips that includes the 68008 (found in inexpensive home computers) and the 68020 (a full 32-bit chip that is found in super-minicomputers). Learn about the 68000 and you will know most of the details of these chips, too.

Acknowledgments

I would like to thank Cindy Martin, my wife, who helps me continue my real life while I'm caught up in the pages of a book. I also thank the people at TAB BOOKS Inc. (especially Kimberly Tabor, Raymond Collins, Sandy Shatzer, and Leslie Wenger) who were patient and helpful all the way from sending me foundation materials for this book to accepting my pleas for more time.

The people in the computer industry, from chipmakers to computer crafters, who sent me information and photographs of their products helped me greatly. These include the following individuals:

Stan Victor of Mostek Corporation.
David Anderson, James Lovegrove, Susan Dunn, and Lothar Stern of Motorola Inc.
Erica Vogler of Apple Computer Corporation.
Jill Palmquist of Corvus Systems.
Barbara Henry of Micro Craft Corporation.
Donna Caruso of Charles River Data Corporation.
Mike Radisich and Dennis Steiner of Hewlett-Packard.
Rick Bennett of Synapse Computer Corporation.
Tracy Adams of Convergent Technologies.
Leslie Baba of Callan Data Systems.
Sinclair Research, Ltd.

MASTERING THE 68000 MICROPROCESSOR

Introduction

This book has two purposes. First, it introduces you to microprocessors using the 68000 chip as a guide. If you are unfamiliar with microcomputing technology, the 68000 is an excellent choice as a first microprocessor. Despite its deserved reputation as a very powerful 16-bit microprocessor, the 68000 can be both understood and used by beginners. The fundamental programming elements of the 68000—from instructions to addressing modes—are no more complicated to learn than those of far less powerful 8-bit chips. But because those elements contain more punch and because the 68000 family includes chips with advanced instructions and memory addressing schemes, a 68000 program gets more done than a comparable program for an 8-bit chip. For that reason, if you are familiar with microcomputing, you'll probably have heard of the power and influence of the 68000 microprocessor.

The second purpose of this book is to present the fundamental information you need to understand and write 68000 assembly language programs. From the architecture to the individual instructions, this book covers each vital part of the chip. There are some advanced details of the 68000 that this book doesn't attempt to cover, including such things as the exact timing of instructions, the interfacing of peripherals, and the algorithms for assembly language subroutines. Those subjects are better left to the original literature from the chipmaker or a book dedicated to this subject.

My intention is to present the software side of the 68000 in a simple and painless way. If you want to learn about the hardware side of the 68000, you still need to know what is in this book. After you have read it, then you'll be ready to study the manufacturers' hardware details and specifications.

Chapter 1 explains the genesis and the importance of microprocessors and the 68000. This is not a detailed history—plenty of those have been written elsewhere. Instead, Chapter 1 is just a refresher on the reasons why any of us bother with these tiny and complicated chips along with some suggestions on how to use this book.

Chapter 2 describes the inside of the 68000. As

with the rest of the book, this chapter assumes no prior knowledge of microprocessors. Before describing a feature of the 68000, the meaning and use of that feature will be covered in general. Chapter 2 begins by explaining the difference between 8-, 16-, and 32-bit microprocessors and then details the registers, memory addressing space, flags, instructions, addressing modes, and interrupts that the 68000 has. This chapter compares other 16-bit chips to the 68000. Chapters 3 through 7 explore in more detail the chip elements first listed in Chapter 2.

Chapter 3 details the registers and compares them to those on other microprocessors. The structure, function, and use of each register is explained.

Chapter 4 begins by explaining the reasons for different modes of address and then takes each 68000 addressing mode in turn and shows its use, flexibility, and restrictions.

Chapter 5 explains what an instruction is and what it can do. The orthogonality of the 68000 instruction set is explained. The instructions are grouped by function and their uses and idiosyncrasies are discussed.

Chapter 6 is the longest chapter in the book. It describes each instruction individually with definitions, condition code effects, allowable addressing modes, and a quick bit-breakdown of the object code.

Chapter 7 explains exception processing. This subject is called *Interrupts* on many other microprocessors, but the 68000 adds new categories of exceptional processing that help the programmer handle everything from meaningless mathematics to unauthorized memory access.

Chapter 8 lists the members of the 68000 chip family, including CPUs and peripherals. All of the CPUs (68008, 68010, 68020, 68200) and some of the more important peripherals are discussed.

Chapter 9 begins with a quick introduction to computer languages. Since this is a manual and not a programming exercise book, the coverage is brief. The chapter then launches into a discussion of the advantages of assembly language and the basics of assembly language technique.

Chapter 10 pictures and describes sample products that use the 68000. These range from personal computers that cost $500 to minicomputers that cost half-a-million dollars. The last category is chosen to show the range and power of the 68000 family.

The experienced reader of technical material will be familiar with the style of a microelectronics manual such as this one. Novices, however, may be awed and confused by the repetition of words and phrases. A common complaint from those who overhear the language of computerese is that the lingo is impenetrable and that it was probably designed to obscure meaning and cloak the subject in mystery. The language may be impenetrable for those who haven't opened the technical dictionary, and it can and is used as obfuscation.

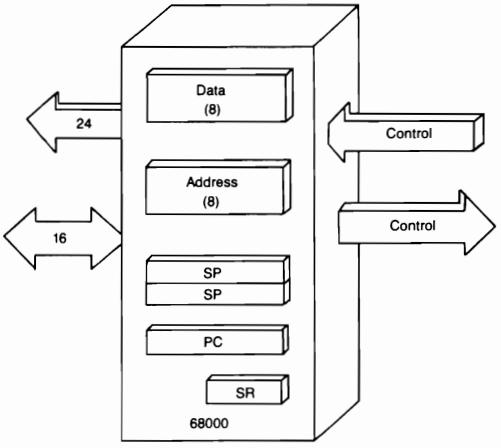
In a book such as this, however, use of technical terms is vital and is the only practical way to convey the information in less than an encyclopedic length. Each term will be explained, but once explained the terms will be used. When you are familiar with the terms, you'll be glad to avoid the endless repetition of definitions. When you begin to learn a foreign language it is hard to even know the subject of a conversation. But once you are gaining fluency, you wouldn't want to have every word followed by a snippet from a dictionary. The same is true of technical terminology. Technical disciplines are built on technical lingo just as surely as mathematics is built on numbers and letters. Without using these symbols, the science would be crippled.

Microelectronics is built on an edifice of understood terms. Their combinations and relations make their meaning; adopting a more mellifluous set of names and terms would make for easier reading and less understanding. In studying a subject such as this one, just remember this: what looks complicated and needlessly obscure now will be too simple to bother with later, except when you explain it to the next novice.

Most authors place the request for debugging in the acknowledgments, but because I know that most people, myself included, don't read acknowledgments unless they have reason to believe their own names are included, I'll mention debugging here. Please let me know about any errors in

this book so that I can correct them. By errors, I mean factual mistakes, typos, and even omissions of material. Obviously, I and my editors have tried to catch these problems, but there is no more vigilant or observant proofreader than a user who is cursing the transposition of a 1 and a 0.

1



Why Bother About Microprocessors?

MOST ELECTRONIC CHIPS ARE NAMED WITH a number. The 68000 chip is a microprocessor chip. By microprocessor, I mean that the 68000 is a single, small, square piece of silicon that has been impregnated with all the electrical circuits of the heart of a computer. It can be put to use in everything from a household appliance to a million-dollar computer.

This chapter is written for those who aren't sure why microprocessors have captured so much news attention. Many other books are available that describe in detail the people and events that led to the invention and improvement of microprocessors. This chapter only sketches the role of microprocessors and a few of the concepts vital to understanding them.

Microprocessors are the pivot of the information revolution. They are the brains of the electronic-chip world and so are vital to computers and thousands of other products, from sewing machines to railroad cars, advanced X-ray equipment to pacemakers.

POWER AND POPULARITY

The 68000 has both facets necessary to make it a leading microprocessor: it is powerful and popular. Without power, a microprocessor will soon be left behind in the technological race. An unpopular chip, no matter how state-of-the-art, will not have enough software to be quickly useful in future situations.

The question of a microprocessor's popularity is an interesting one. The first chip that fulfills the system designers' needs will often be the most popular chip. But if the previous generation of microprocessors can still handle most designer demands, the first chip marketed may be left behind by a later, more powerful chip.

In addition, there is a snowballing effect to microprocessor popularity. Once a chip begins to be popular, programs for it start to appear. Those programs make the chip even more popular and result in more systems being designed with that chip. More systems means even more software will appear for it. And around and around it goes. In the final analysis, the size and reputation of the com-

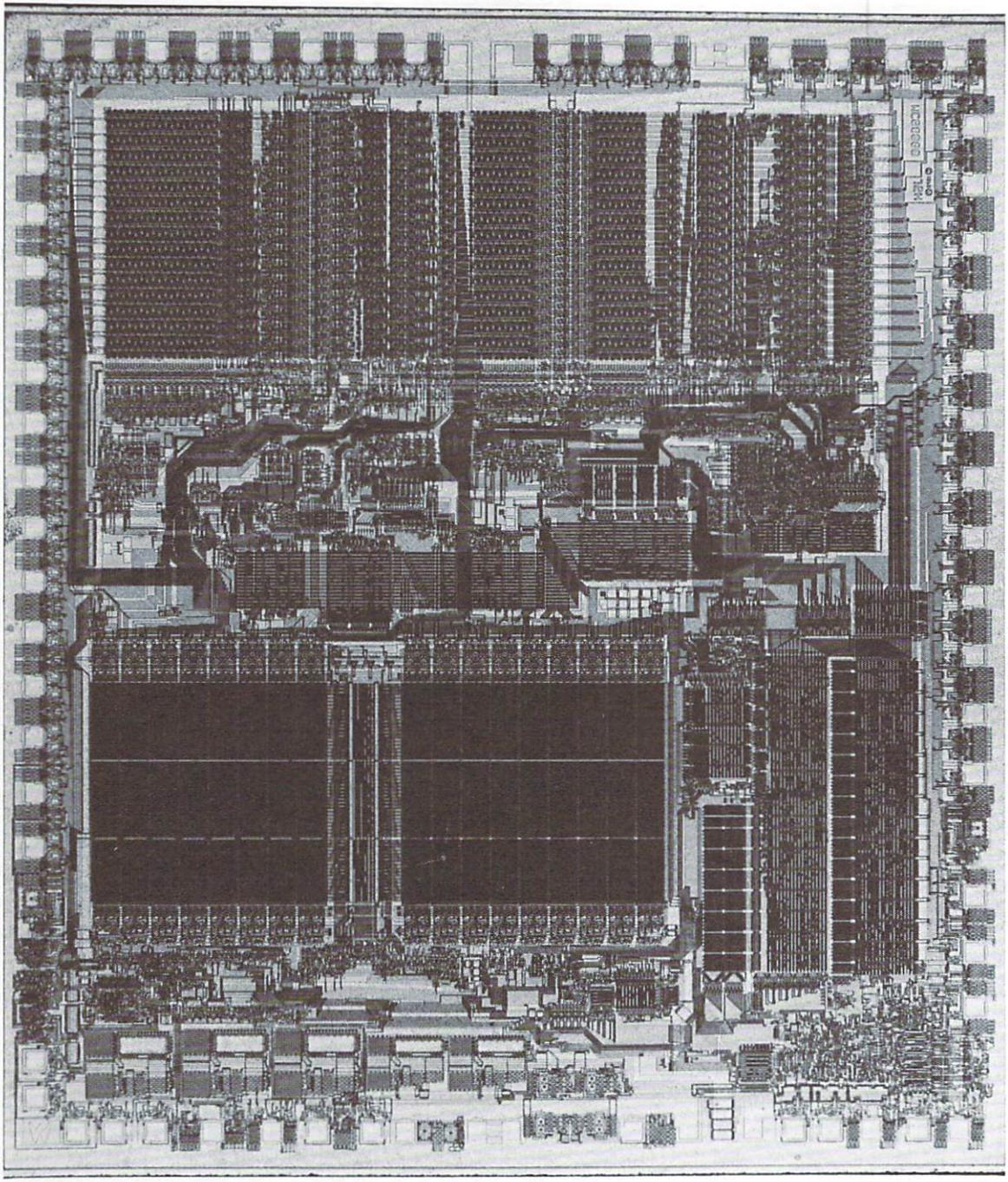


Fig. 1-1. The 68000 chip (courtesy of Motorola).

panies that use the chip in their systems are also vital to its popularity.

The 68000 is not the preeminent microprocessor today. Some older 8-bit chips, such as the Z-80 and 6502, are sold in larger numbers. The 8086, a chip from the same 16-bit generation as the 68000, can be found in more personal computer designs than the 68000, but it is less powerful. In many ways, the selection of the 8088 (a close relative of the 8086) by IBM for its personal computers was the largest single boost to that chip.

68000s are found in more engineering and instrumentation systems than the 8086 and are beginning to appear in personal computers such as the Apple Macintosh, Apple LISA, and the Sinclair QL. Together, the 8086 and the 68000 have captured most of the 16-bit microprocessor market. Figure 1-1 shows a microphotograph of the 68000.

The next generation of microprocessors will work with 32-bits at a time. These super microprocessors will have as much power as many full-size computers have now. The 68000 opens the door to this world in two ways. First, the 68000 uses 32-bits in internal processing, so in some ways it is a hybrid 16- and 32-bit microprocessor. Second, the 68000 family already includes a full 32-bit processor called the 68020 that can run all the programs written for the 68000. The 8086 doesn't have this advantage. Its 32-bit relative, the 80386, was not yet available when the 68020 started to appear in systems.

STANDARDIZATION VERSUS SPECIALIZATION

The entire field of microelectronics is rooted in both World War II and the space race. Missiles and aircraft need control systems that are lightweight and reliable, yet very complex in design. The technology that provides all these benefits is solid-state electronics.

Several breakthroughs took place during the late 1950s and throughout the 1960s that allowed designers to put more and more electronic circuit elements on the surface of a single crystalline sheet. The silicon chips of these processes have since been celebrated in books, magazines, songs, and lent their name to the Santa Clara Valley in California, where much of the groundbreaking work was done.

Simultaneously with the miniaturization of flight electronics, the science and technology of computers exploded. Computer designers quickly adopted solid-state electronics and integrated circuits because of their high reliability, low power demands, and fast operation. Microelectronics was used to build the central processors and memories of the huge computers of the 1960s. By 1970, however, a roadblock had appeared in the way of further development of integrated circuits.

COST AND YIELD

The complex and subtle processes used to implant microscopic transistors, resistors, capacitors, and other electronic components on a block of extremely pure crystal are difficult to master and require very expensive equipment and highly trained personnel. The more complex a circuit, the harder it is to fabricate correctly. In fact, with increasing complexity the number of working circuits drops precipitously. This relationship is shown in Fig. 1-2.

At the beginning of the process of creating circuits, each sheet of silicon crystal is laid with the foundations for several hundred chips. By the end of two dozen different steps, only a fraction of those chips will have been formed perfectly. Almost any flaw, no matter how small, destroys the chip. The percentage of good chips from a wafer or sheet of silicon is called the yield. Cost is directly related to yield.

If the engineers cannot get a high enough yield out of a wafer, the resulting microcircuits will be too expensive. These costs were looming like a wall in the face of progress in the late 1960s. Since microcircuits are difficult to design and even harder to produce, they can only be economical if they are sold in large volume. A chip that has only a few uses will be very expensive and so will only appear in special costly systems.

Except for memory chips, chip designers were having a hard time coming up with new chips that would sell in volumes large enough to pay for design and process development.

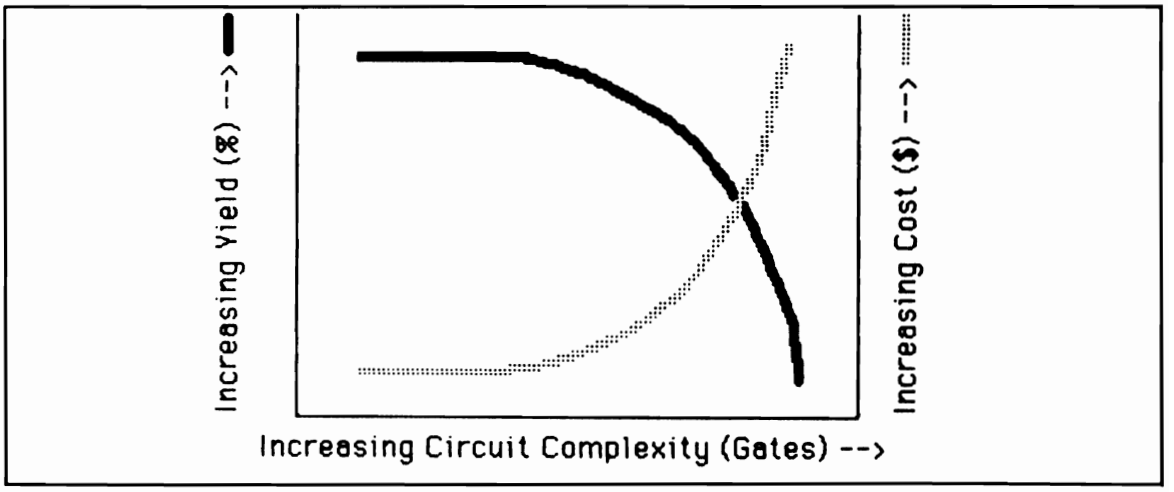


Fig. 1-2. The learning curve (Idealized).

INTEL'S BREAKTHROUGH

The breakthrough came in the guise of something known as programmability. Computers are programmable. Usually, they are not dedicated to any particular use: instead, the computer does whatever it is programmed to do. A programmer, or computer engineer, writes a series of instructions that tells the computer exactly what steps to take to solve a problem. Solving a problem doesn't mean just working out math equations. From cataloging fingerprints to keeping the books, from playing games to controlling traffic lights, computers perform a huge variety of tasks. In fact, the magic of computers is that they are malleable machines. If you think of a new job that needs doing, all you have to do is write a program. You don't have to design and build an entirely new machine. The microcircuit engineers were slow to realize that this fact could provide the answer to their design problems.

A few IC (integrated circuit) manufacturers were employed by calculator companies to fashion the central parts of a calculator on a single chip. When chips are economical, they are very economical, even downright cheap. Typically the costs of an integrated circuit will halve every two or three years. That's because the yields rise with processing experience: a phenomenon known as the learning curve. An idealized version of the learning curve is shown in Fig. 1-3. Other advantages of ICs over conventional circuits on fiberglass boards include lower power consumption and higher reliability. So the calculator companies were hoping that the primitive computing functions of their machines could be incorporated into ICs and thus become cheaper to produce and more reliable to work with.

Intel was the first manufacturer to get all the circuits on a single IC, but the chip wasn't fast enough (it took too long to make its calculations). So the calculator companies went back to their old methods of making circuits and Intel tried, without a lot of hope, to sell an already paid for, slow-calculating chip.

That chip, the Intel 4004, was the first microprocessor. It began to sell, and sell, and sell. It sold so well that Intel quickly put out an improved version, the 4040 (like the 4004, a 4-bit chip) and then an even more powerful 8-bit chip, the 8008. The 8008, in turn, gave way to the 8080. Other manufacturers developed competing chips, such as the Motorola 6800, and the microprocessor revolution was on its way. Who was buying all those chips? A lot of people.

Though they were slow compared to the refined circuits of the calculator manufacturers, these chips had the advantage of programmability. In other words, different people could use them for different purposes. The breakthrough in chip design

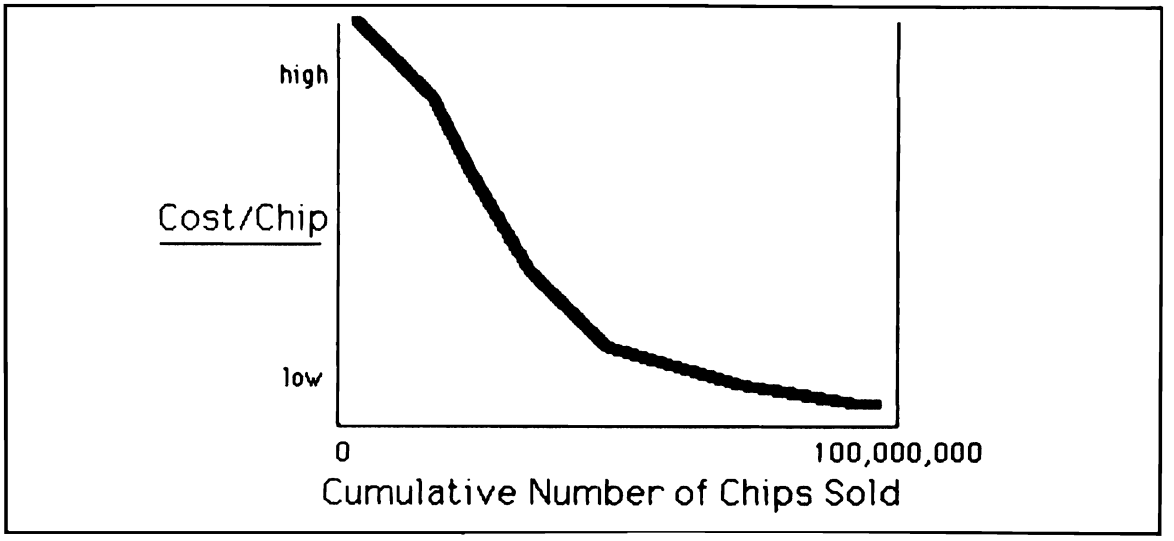


Fig. 1-3. Sales of microprocessor chips vs. cost per chip.

was the development of a general-purpose chip that anyone could use. The same chip would work for many applications; only the program, or software, needed changing. Industrial engineers designed microprocessors into machine controllers; automobile manufacturers designed them into engines; hobbyists designed them into just about everything.

The ability of a microprocessor to gather information, to manipulate that information, to test it, and then to make a decision based upon the programmer's earlier decision, and finally to initiate the action of some other machinery makes it into a tiny thinking device. The unreliable and limited mechanical controllers in traffic lights, microwave ovens, airplanes, and vending machines can all be replaced with microprocessors. In addition, systems that didn't use controllers before, could now be made more efficient by the judicious use of microprocessors. Applications appeared where no one had even thought to look. IC manufacturers found that they could concentrate on making better processors and support chips rather than worrying about specialized chips that no one could afford. Better yet, they discovered that designers who bought microprocessors would then order lots of memory chips (which is where the IC companies made their profits).

The breakthrough came in hardware design. But now, because of that breakthrough, designers needed to become software experts just as much as hardware experts.

MICROPROCESSOR EVOLUTION

The first microprocessors were 4-bit devices. Like most computers, they were based on digital electronics and the binary number system (which represents all information by strings of 1s and 0s). Digital systems can be made more reliable and more precise than analog systems.

Four-bit microprocessors deal with groups of four binary digits, or bits, at a time. Although these were adequate for simple applications, many users soon needed 8-bit devices. The manufacturers were quick to respond. Figure 1-4 depicts the family tree of the most popular microprocessors. There are many other microprocessors that are not shown or discussed here. Some are used solely for military applications, others never appeared in many systems, and still others have just recently appeared and are not yet well known.

There are several important trends in the microprocessor field. The first is the development of chips that can handle more bits at a time. The first chips were founded on the use of 4 bits; with

double length registers (locations on the chip for storing bits), 8 bits could be used for some operations. In some circumstances, this capability of chips to work with twice their fundamental bit number helps to confuse the labeling of chips. You'll hear of 8-bit, 16-bit, and 32-bit chips, but you'll also probably run into mention of 8/16-bit chips and 32-bit chips with 8-bit buses. I'll explain some of those mysteries in Chapter 2, but for now, remember that 16-bit chips are generally more powerful than 8-bit chips, and 32-bit chips are the most powerful now available.

The second new branch in the tree of microprocessors is the single-chip microcomputer. Most microprocessors need support chips such as clock oscillators, bus multiplexors, and I/O (Input/Output) controllers. Single-chip microcomputers have all of these functions integrated on a single chip. The 68200 is an example of such a chip (see Chapter 8 for more information on the 68200).

MICROPROCESSORS AND MICROCOMPUTERS

Almost as soon as microprocessors made the vital central processing unit (CPU) available in a small, affordable package, hobbyists started building them into tiny computers. By attaching a power supply, I/O devices, and some memory, a microcomputer was created (as mentioned above, the term *microcomputer* is sometimes also used for a microprocessor that has additional functions packaged onto a single chip). The early microcomputer models quickly gave way to sophisticated packages with keyboards, monitors, large memories, peripherals, and even systems software.

The first hobby computers were built around the 8-bit chips such as the 8080 and the Z-80. But as professionals, business people, and engineers began to use hobby computers—soon renamed personal computers or PCs—the computer designers knew they needed more power than the 8-bit chips

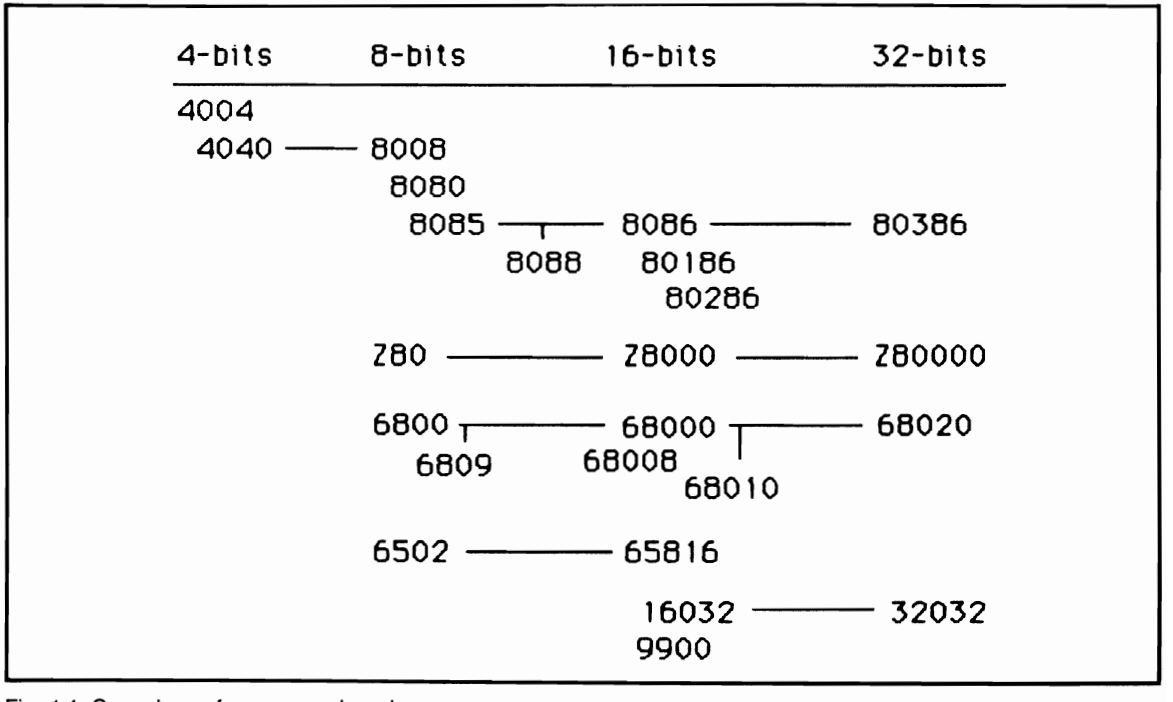


Fig. 1-4. Genealogy of some popular microprocessors.

could provide. At that point, 16-bit chips like the 68000 and the 8086 began to appear in PCs. Microcomputers soon became a faster growing segment of the computer market than the traditional mainframes and minicomputers and can now be found in homes, schools, businesses, and laboratories.

SOFTWARE VERSUS HARDWARE

As the information revolution proceeds, fewer people design hardware and more use commercial hardware as a tool to design software. Way back in the early 1970s, if you wanted a microcomputer, you had to build it from a kit. You had to understand the electrical aspects of microprocessors just to be able to use one. That isn't true any longer. In fact, only dedicated hobbyists or inventors need to worry about enable signals, line buffers, race conditions and all the other esoteric aspects of hardware. Before, you had to build a computer to own one; today it is much cheaper to buy a standard system and program it to do what you want. This book is aimed at software because that is where the action is.

Very few people today will interface a 68000 chip to other chips. Many, many people will be programming 68000-based systems in machine, assembly, and high-level languages. This book sketches some of the hardware aspects of the 68000 so that you will know what hardware people are talking about when they discuss 68000 systems. But by far the majority of the book leans on the things you must know to program a 68000 microprocessor or to understand how a 68000 program works.

To understand and create software for 68000 CPUs, this book should be sufficient. You won't have to get the manufacturers data. However, it is always a good idea to have manufacturers literature in hand. In some cases, learning to read the original documentation is as important a skill as actually programming a chip.

If you want to incorporate a 68000 chip into some hardware, this is not the only book you'll need. You'll need the software knowledge in this book and the hardware facts in the chip manufacturers' latest manuals. In fact, a single book couldn't hold all the necessary software and hardware facts and explanations and still be easy to carry.

COMPATIBILITY AND CHIP FAMILIES

Besides a software emphasis, I also try to be diligent in explaining the concepts of compatibility and chip families. The 68000 isn't really a single chip. Instead, it is a family of CPU chips and peripheral chips.

Other CPU chips include an 8-bit chip (the 68008), a 16-bit virtual memory chip (the 68010), a single chip controller (the 68200), and a super-powerful 32-bit CPU (the 68020). The peripheral chips are dedicated to relieving the CPU of particular tasks including memory management, I/O (input/output), floating point arithmetic, and peripheral device control. Many of these chips are discussed in detail in Chapter 8. Throughout the bulk of this book, almost all explanations refer to the 68000 and the 68008.

This family of chips presents the programmer with two types of compatibility. First is the compatibility of learning. The CPU chips all have very similar structures and operation (with the 68200 varying more than the rest). That means you only have to learn one chip to have command of 8-bit, 16-bit, and 32-bit microprocessors. Also, you need only learn one set of peripheral chips because the CPUs in the 68000 family are specifically designed to work with many of the same peripheral chips.

The second type of compatibility has a stricter meaning. 68000 CPU chips are designed to run software written for other 68000 family CPUs. Not only do you only have to learn one chip, but many programs you write will run without change on other 68000 CPUs. The software written for a small 8-bit personal computer can be run on a 16-bit engineering workstation or a 32-bit superminicomputer. With programmers and programming representing the major slice of most computer budgets, that time and effort savings is substantial and important.

The rules of this compatibility are fairly simple. First, the 68200 doesn't figure into the pattern. It is only a similar chip, like a distant cousin, instead

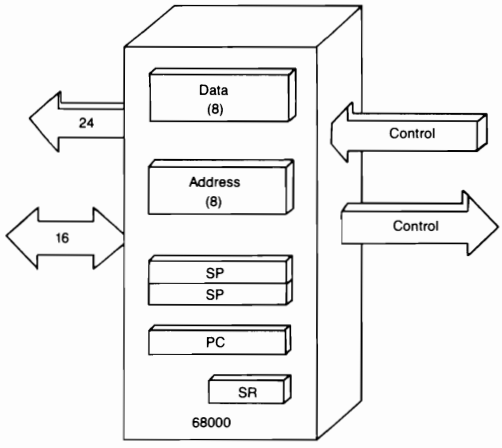
of a sibling. Other than that special case, programs written for a less powerful member of the family will run on a more powerful member but programs written on the more powerful chip will not necessarily (but often will) run on the less powerful chip.

For instance, the 68008 is, at 8-bits, the minor member of the 68000 family. Programs that run on the 68008 will run without modification on the 68000. However, programs that work on the 68000 may use some of the additional power—in the form of instructions and interrupt priorities—that the 68008 doesn't provide. Those programs won't run properly on the 68008. Similarly, programs for the 68010 will run on the 68020, but a program that runs on and uses the advanced addressing and instructions of the 68020 will not run on the 68010.

Motorola, the designer of the 68000, claims that future members of the 68000 family, while more powerful, will be software compatible with the present chips. While hardware designers will have to look up the particular facts for the particular 68000 chip they use, be it 68000, 68008, 68010, 68020, etc., software designers can learn one chip and use any of the others with only a little extra study.

Remember that Motorola reserves the right, as do most semiconductor manufacturers, to change any of its chips to improve their design, function, or reliability. You can bet, though, they'll do what they can to keep the chips compatible. That's something that's easier for you as a software designer than for your hardware compatriots. Chips are often changed at the hardware level. Speeds, chip size, and other details can change with evolutionary improvements in chip production and design. Those changes are made though with the thought uppermost that the software must run undisturbed.

2



Architecture

THE ARCHITECTURE OF A MICROPROCESSOR is its internal structure: it includes such elements as registers, interrupt signals, instructions, and addressing modes. This chapter will explain what those things are and why they are important. Chapters 3, 4, 5, 6, and 7 each treat one aspect of the 68000's architecture in much more detail. The organization of these chapters follows the outline in this chapter.

HISTORY AND DESIGN PHILOSOPHY

The 68000 design effort started at Motorola in the middle 1970s. The Motorola 6800 family of 8-bit microprocessors was very popular, but users were asking for more power. It was soon clear that only a 16-bit chip could have all that users were looking for. To design that chip, a project called MACSS (Motorola's Advanced Computer System on Silicon) began to investigate quite a number of possible architectures and design strategies. Finally, the project team settled on one design that held the most promise. They used that design to start the 68000 family.

68000 History

Motorola formally introduced the 68000 family's first chip, the 68000 itself, in late 1979. The 68000 has a 16-bit data bus and a 23-bit address bus. While Motorola designed the 68000 and still manufacturers it, other companies also manufacture it (and are designing some of their own peripheral chips to go with it). Those other companies are known as second sources and include International, Signetics/Phillips, Mostek, Hitachi, and EFCIS (Thomson-CSF). Motorola also makes peripheral chips, support systems, and development systems for the 68000.

By the way, Motorola refers to the 68000 series as the MC68000 series. I have left off the prefixes in this book to avoid confusion. Other manufacturers who second source the 68000 or make peripherals for it also attach prefixes to their chips. Mostek, for instance, calls its main CPU the MK68000. It is the same chip as the MC68000 and operates in exactly the same way.

Motorola intended to create an entire family of chips with a standardized architecture. The 68008

was the next chip in the family and has an 8-bit data bus and a 20-bit address bus. The 68008 was clearly aimed at allowing even those who wanted 8-bit systems to use the 68000 family. The 68010, introduced next, is the first chip in the series that has the virtual memory capabilities. The 68020, with 32-bit data and address buses, was announced in 1984.

When the 68000 (sometimes called the 68K) first came out, it was used mainly in expensive computers, because of its very high performance and fairly high price. In 1984, however, because its price had fallen from the original $450 (for a single chip) to approximately $50, the 68000 started showing up in personal and even home computers.

Power Versus Compatibility

Chip designers soon discover that you can't have everything when you design a new chip. Maximizing some aspect of the chip's performance will often have a direct negative impact on some other aspect. Also, you can't just cram every innovation onto a single chip. If you try to design the perfect chip, chances are it will be very difficult to manufacture. And if a chip can't be made cheaply, it probably won't be widely used and will just end up as a curiousity in magazine comparisons.

The rapidly changing state of semiconductor process technology means that you can't just decide what can be made at present. You have to make a guess at what the state-of-the-art will be when you have finished designing the chip; you have to try to judge what the manufacturing engineers and their equipment will be able to make when you finish designing your chip. Guess too conservatively and the competition's more advanced chips will leave you in the silicon dust. Jump too far out ahead, and you may have the devil of a time getting a working prototype.

There are many other design considerations in any computer design—not just in microprocessors or microcomputers. Software compatibility is a vital factor in any design. Software is a huge part of the expense of computers. Users may well say, "So what?" if you offer them a newer, faster superchip for which they will have to write all new software. It pays to make a new processor that can run the software written for the previous chip.

The 68000 designers decided not to shoot for direct software compatibility with the earlier 8-bit chips because those chips were designed without the future in mind. Eight-bit processors appeared within a very short time after the very first microprocessors. Their designs were often just quick copies of minicomputers or commonly available circuit boards. Many manufacturers thought of them mainly as ways to sell more memory chips. In particular, the 8-bit chips were very hardware oriented: there wasn't much consideration given for easy adaptation of the chip to high-level language software. Future expansion of systems was, for the most part, ignored.

To design an exciting new 16-bit chip so that it would run the programs written for a workhorse 8-bit chip would handicap the newcomer. Motorola's designers decided against it. They wanted a powerful, flexible microprocessor that would be designed from scratch to be easy to use and would simplify the job of writing high-level language systems software. High-level languages are the most efficient medium for writing large programs. They are explained in more detail in Chapter 9.

There was another sort of compatibility that Motorola found important: peripheral chip compatibility. Chips are rarely the monolithic entities that beginners see. Instead they are families. Having a marvelous CPU won't mean much (that is, it won't be designed into many systems) unless there are Input/Output controllers, memory managers, CRT controllers, floppy disk controllers, interrupt handlers, timers, and a horde of other chips that can be directly and easily hooked up (or interfaced) to the CPU. The MC6800 had such a family. In fact, because many I/O operations don't require more than 8 bits of transfer at a time, 16-bit I/O chips are rarely necessary: 8-bit chips can handle the jobs.

The 68000 was designed to interface directly to the 8-bit 6800 peripheral chips. Therefore the day the 68000 was introduced there was a whole family waiting to greet it.

8, 16, OR 32 BITS

Why is a 16-bit chip more powerful than an 8-bit chip? For a number of reasons. But before getting into those reasons, take a look at those two numbers. Is a 16-chip going to offer exactly twice the power of its 8-bit cousin? No. Although 16 is twice 8, the comparison of programming punch is much more complicated. For instance, the number of possible values that can be held by 16 bits is 256 times as many as can be held in 8 bits. Where the 8-bit chip could only have 256 separate and distinguishable values in a register, such as for instructions, the 16-bit chip can have 65536 (64K as shown in Fig. 2-1). Even that's not an endless number. Those bits get eaten up in a hurry.

Registers are small memory spaces on the microprocessor chip itself. Because they are directly on the chip they can manipulate information far faster than if that information had to be taken from memory and then put back. But if you have 16 main registers (as the 68000 does), specifying which one of those registers will be used in any given operation requires at least four bits of information (this is shown in Fig. 2-2). That single use—specifying a register—would swallow half of the bits of the instruction byte in an 8-bit CPU.

Addressing modes are another important component of microprocessing. By having a variety of modes, you can build data structures and program interrupt mechanisms that make for quick and efficient programs. But 8 modes will require 3 more bits of specification. Along with 16 registers, 8 addressing modes would almost exhaust the instruction byte of an 8-bit chip. There are ways around the problem. The specification bits can be sent in several sequential bytes. That approach, however, just makes for other problems. More bits are eaten up just to tell the CPU that another instruction byte is coming. Waiting for those bytes slows execution; and speed is supposed to be what computers are all about.

A larger number of bits also allows for many more instructions. That in turn allows more complex instructions to be written into the repertoire of the chip. Division and multiplication that take programmer time and slow down execution in 8-bit chips are often implemented directly as single instructions on 16-bit chips.

# of Bits	Effective Multiplication	Addressable Memory
1	2	2 bytes
2	2*2	4 bytes
3	2*2*2	8 bytes
4	2*2*2*2	16 bytes
5	2*2*2*2*2	32 bytes
6	2*2*2*2*2*2	64 bytes
7	2*2*2*2*2*2*2	128 bytes
8	2*2*2*2*2*2*2*2	256 bytes
16	2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2	64 kilobytes
24	2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2* 2*2*2*2*2*2*2*2	16 megabytes
32	2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2* 2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2	4 gigabytes

Fig. 2-1. Addressing bits and addressable memory.

Values of 4 specifying Bits	Registers 1 - 16
0000	1
0001	2
0010	3
0011	4
0100	5
0101	6
0110	7
0111	8
1000	9
1001	10
1010	11
1011	12
1100	13
1101	14
1110	15
1111	16

Fig. 2-2. Bits required for register specification.

Another good reason to have more bits in the CPU is to address larger memory spaces. eight bits can address 64K. That may sound like a lot, and it is quite impressive compared to the memory spaces of some of the early computers. But for modern applications, single programs can easily require 100K of memory. Implement I/O devices, screen memory, and multiple character sets, and 64K is soon gone. Sixteen bits, on the other hand, can address 256 times more memory. Add a single bit to a CPU address bus and you double the addressable memory space. That's some sort of binary magic.

The width of the data bus is so important that it is often the sole criterion for deciding whether or not a chip is 16-bit. A bus that is twice as wide can move information twice as fast. In programs that require lots of writing to or reading from memory, a 16-bit chip with the same clock frequency as an 8-bit chip would be almost twice as fast for that single reason alone.

Is the 68000 a 16-bit chip? Apple (which uses it in the Macintosh) likes to call it a 32-bit chip. Sixteen-bit chips in general are not easy to classify as 8-bit chips. In fact, a close examination of many of the newer microprocessors shows that few are completely in the 8-bit, 16-bit, or 32-bit camp. Manufacturers love to claim as much width as possible for their chips, but users often discover limitations on data bus, address bus, internal buses, operation units, and other specifications. The data-bus width is generally used as the main indicator of chip type, but unless a chip has an op code that can reach x-bits, it probably shouldn't be called an x-bit CPU. Figure 2-3 lists a number of popular chips.

The MC68000 family has 32 bit registers but a 16-bit ALU and data path. The 68000 was the first 16-bit microprocessor that was actually, internally, a 32-bit microprocessor. It was also the first with 16-megabytes of unsegmented directly addressable memory.

I conclude that the 68000 is basically a 16-bit chip with many internal 32-bit features. For instance, its main registers (described in Chapter 3) are 32-bits wide. Also, its program counter (that specifies the location of the present instruction) is 32-bits wide. But the 68000 is clearly a 16-bit chip. It fetches data 16-bits at a time (a word at a time). It can work with bits, nibbles, bytes, words, or long-words (1, 4, 8, 16, or 32 bits).

Because its instructions are coded into words (see Chapter 6 for more detail on individual word

8-bits	8/16-bits	16-bits
8080	8088	8086
8085		80186
Z80		68000
6502		Z8000
6800		

Fig. 2-3. Some popular microprocessors.

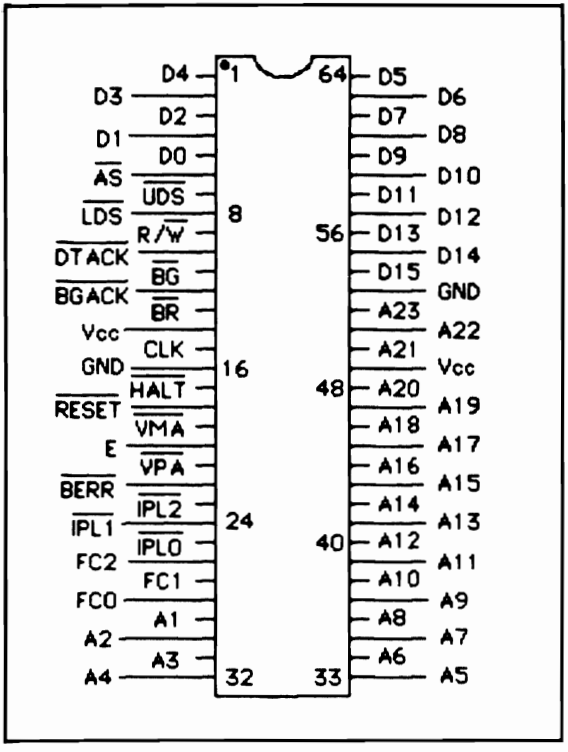


Fig. 2-4. 68000 pinout (assignments).

coding), it frequently requires only a single word to fully describe an instruction. Some instructions do require 16-bit extension words. These follow the op code in the assembly coding. The extensions add addressing information and can increase the length of an instruction to as much as 5 words.

BUSES

The standard 16-bit, NMOS 68000 comes in a 64-pin package. That is, it has 64 separate wires or lines that connect the chip to the outside world. Figure 2-4 shows the pins of a 68000 and Fig. 2-5 is a logical layout of the pin functions. A group of similar signal lines is called a bus. The 68000 buses are shown in Fig. 2-6.

Data Bus

The data bus (shown in Fig. 2-7) is 16-bits wide. It has 16 separate lines. The lines are bidirectional; information can move on them out from the 68000 CPU or in to it, but not both directions at once. This is the bus that handles the actual bits of information. The width of the data bus cannot be used as the sole criterion for the bits of a microprocessor.

For instance, the 68008, described in more detail in Chapter 9, has all of the attributes of the 68000 except that its data bus is limited to 8 bits. Does that make it an 8-bit microprocessor? No. With 32-bit registers and the ability to multiply and divide 16-bit operands, it certainly should not be classified with other 8-bit chips such as the Z80.

Address Bus

The 68000 address bus (shown in Fig. 2-8) is 23-bits wide and is unidirectional. It is mainly used to send addresses from the 68000 to memory. It also carries interrupt information. The 23-bit width of the address bus may be surprising, in view of the 32-bit width of the program counter (which holds addresses to be sent out on this bus). But the 68000 designers decided that 68000 users didn't need a full 32-bit address; that a 24-bit address would encompass enough memory. Using the full address would have pushed the required number of

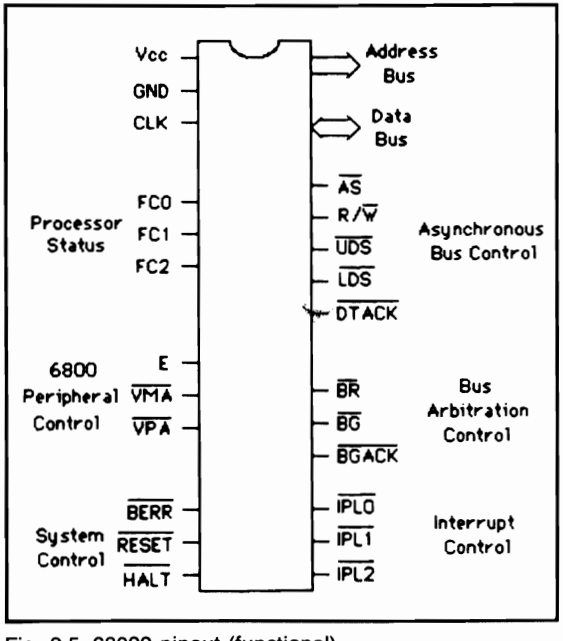


Fig. 2-5. 68000 pinout (functional).

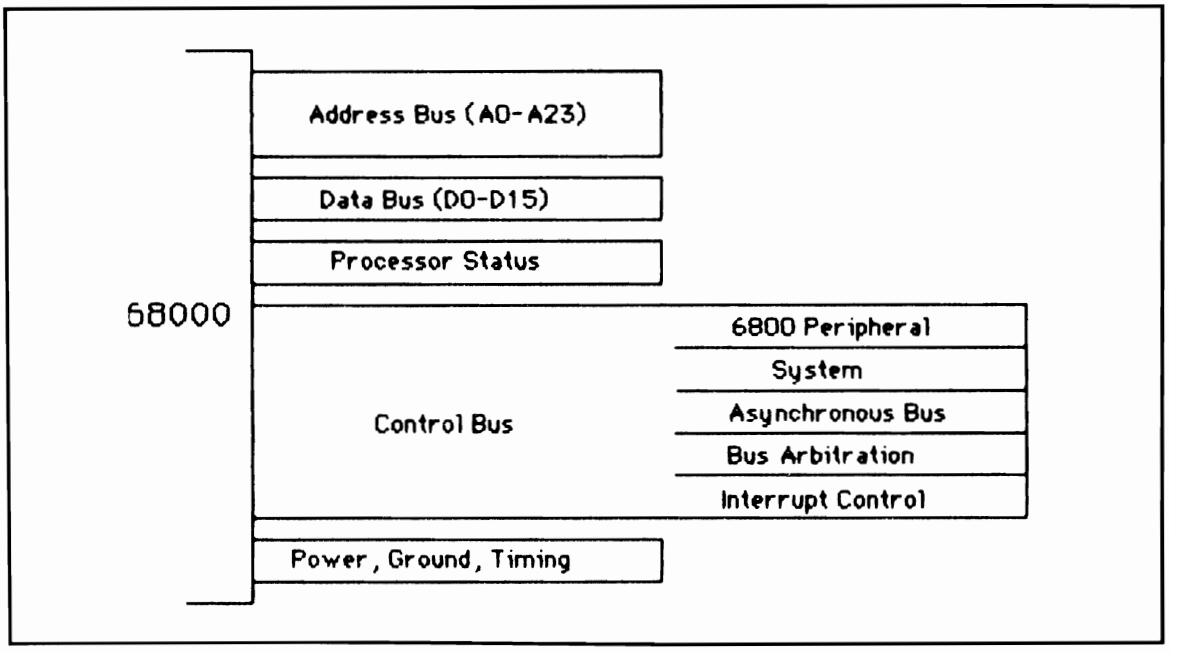


Fig. 2-6. 68000 buses.

pins beyond the already huge 64-pin package. On advanced chips such as the 68020 (described in Chapter 9), a full 32-bit address bit is used, but these chips are much more expensive than the 68000 and must be packaged in a pin-grid array type of package that has more pins that the DIP used for the 68000.

The 68000 depends on direct linear addressing: some other systems get by with a narrower address bus by complicated schemes of paging and segmenting.

68000 I/O (Input/Output) is memory mapped. That means the same instructions are used to move data to peripherals as within memory itself. Some microprocessors have a different set of instructions that refer specifically to I/O devices.

Memory management (MM) is a technique used in many computers. Memory is divided into blocks and the area any programmer can use or see can be limited by a system supervisor. That isn't done just to protect secrets. It's also to keep a wild program from dicing up everyone else's memory. Sometimes a special chip called an MMU (Memory Management Unit) does this job. Other times, it is

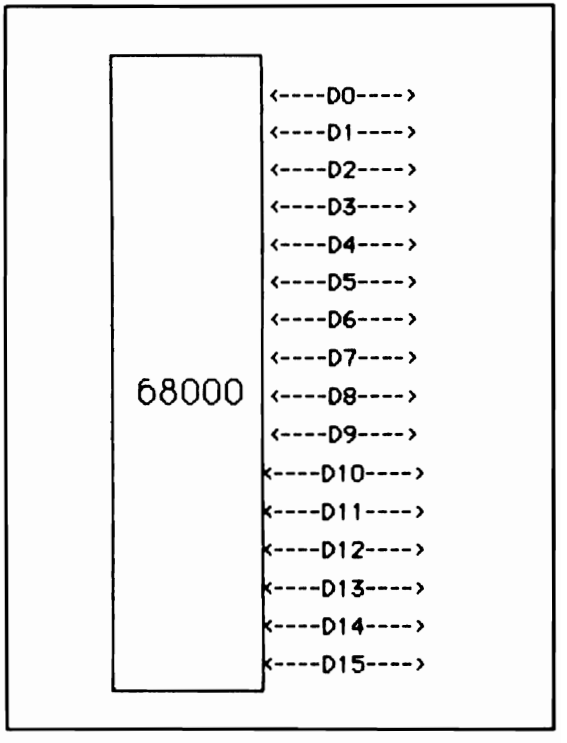


Fig. 2-7. Data bus.

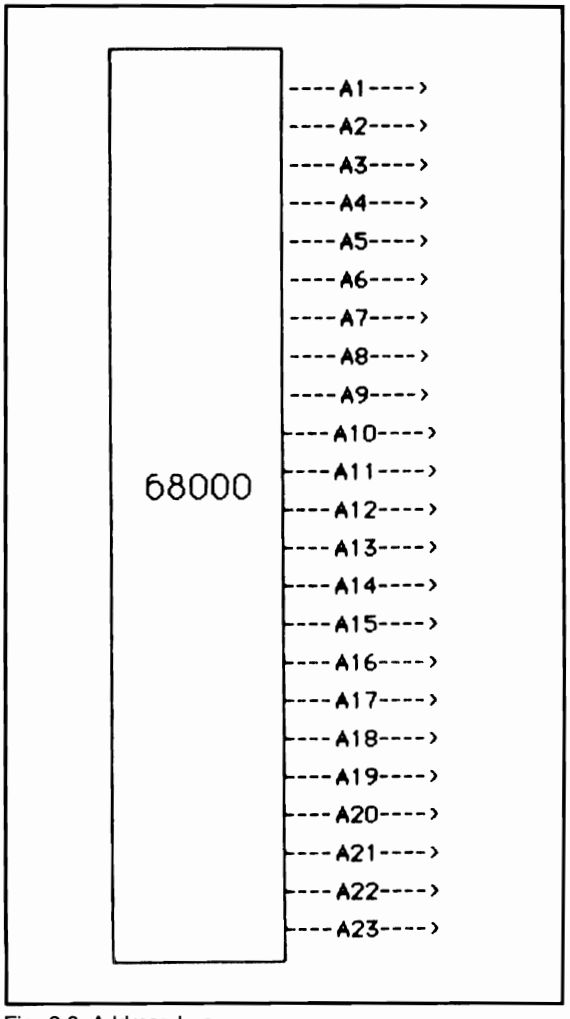


Fig. 2-8. Address bus.

a combination of software, operating system, and chips. As an application programmer, you wouldn't know it was there. You just write programs and the system takes care of MM. Chapter 8 describes the 68000 family Memory Management Chips. The 68000 does distinguish between two types of memory references: data and program. All operand writes are to data space.

Control Bus

The control bus (shown in Fig. 2-9) contains a more diverse group of signals than the data or address buses. These are the signals that provide communications between other chips and the CPU. There are both asynchronous control lines (for 68000 peripherals devices) and synchronous control lines (for 6800 peripherals and other slower 8-bit peripherals).

REGISTERS

Registers are memory storage places on the CPU chip itself. Because they are easily addressed and

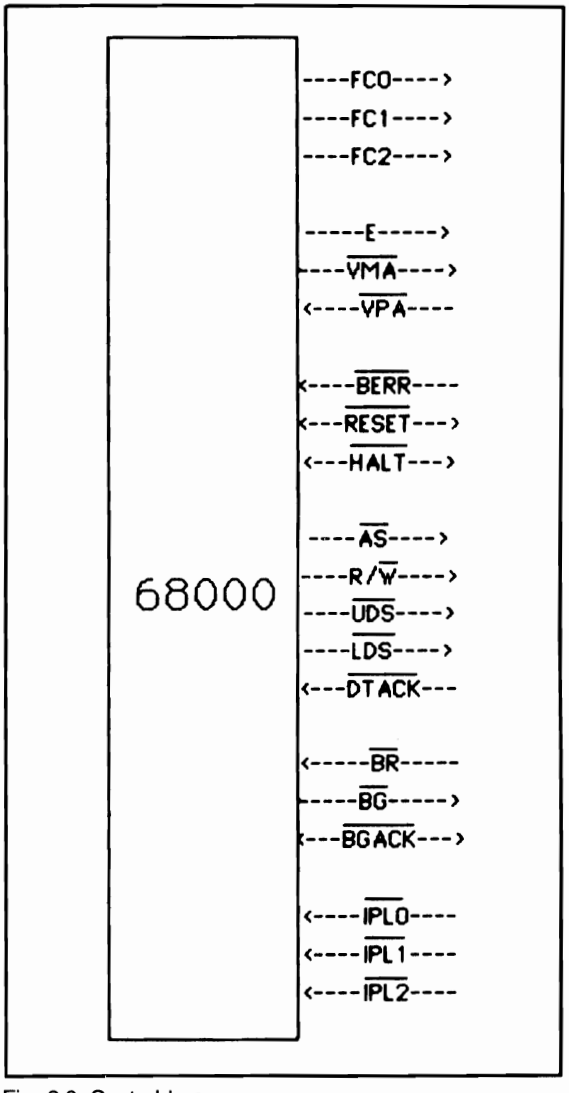


Fig. 2-9. Control bus.

are built into the microprocessor, they are very quickly read or written to. So instructions that refer to registers can execute faster than instructions that refer to external memory locations. Advanced microprocessors often have more registers than simple microprocessors.

Registers can be dedicated to a special purpose or they can be so-called general purpose. Special-purpose registers often include stack pointers and flag registers and cannot be used for other tasks. General-purpose registers may still have commonly-assigned tasks that their design slants them toward, but you can use general-purpose registers for a variety of tasks. For example, any general purpose register can be used as an index register.

Figure 2-10 shows the 68000 registers.

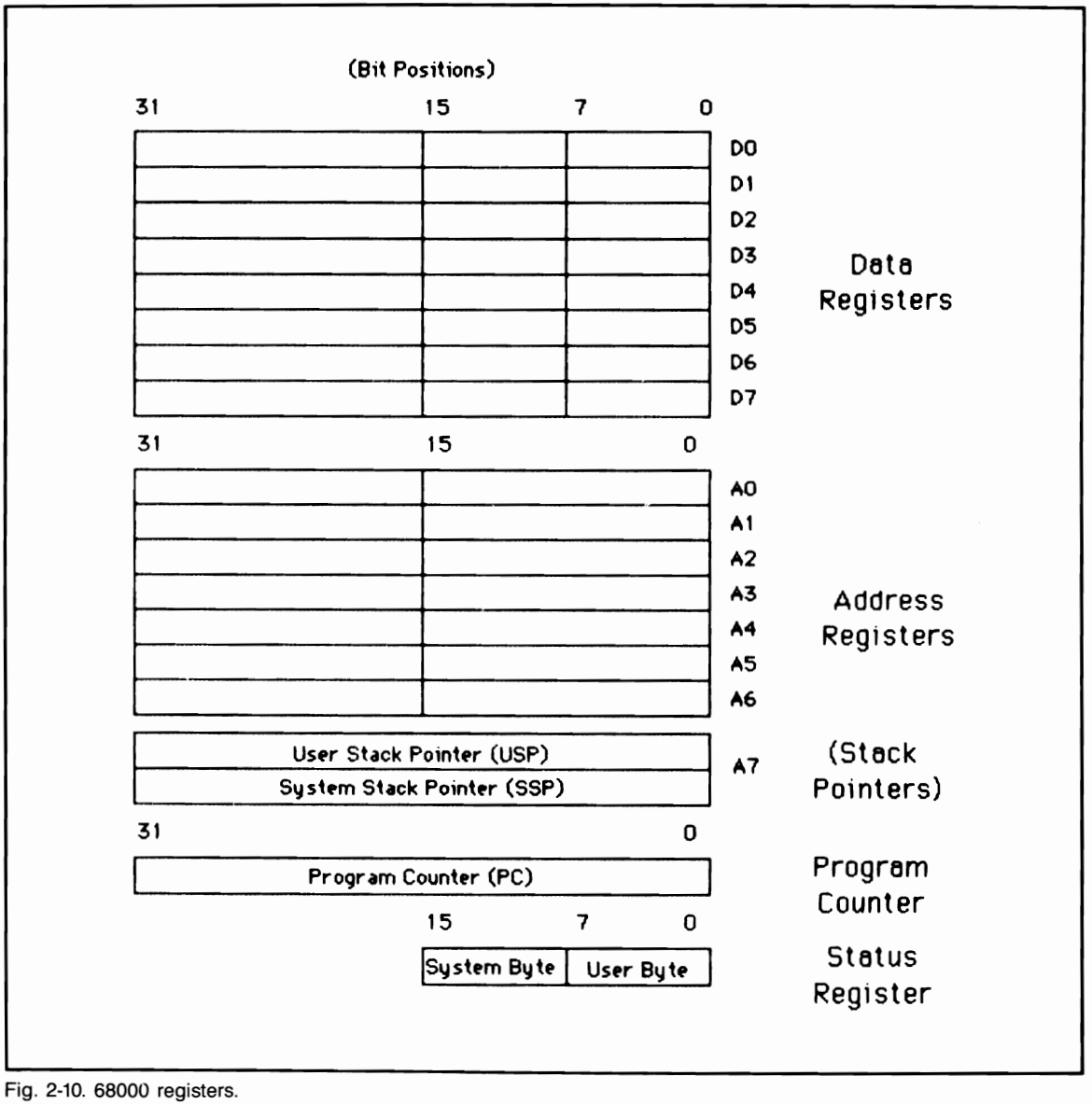


Fig. 2-10. 68000 registers.

General Purpose

The 68000 has 16 32-bit, general-purpose registers divided into 8 data registers and 8 address registers. The data registers can work with byte, word, or long-word operands while the address registers can only work with word or long-word operands. The data registers commonly perform the function that the accumulator performs on many 8-bit microprocessors: acting as a central point for logical and arithmetical operations.

The address registers replace the base address registers and index registers found on many 8-bit CPUs. They can also operate as stack pointers.

Special Purpose

Register A7 is in many ways a special-purpose register. The 68000 has two stack pointers, the User stack pointer (USP) and the Supervisor stack pointer (SSP). These are 32-bit registers that contain the address of the top of the stack (as explained in more depth in Chapter 3). Only one of the two stack pointers is active at any given time. The other still exists physically on the chip but is not normally available to the programmer. When the 68000 is in User state—when the S bit in the condition codes holds a 0—register A7 is the User stack pointer. When the 68000 is in Supervisor state—when the S bit holds a 1—register A7 is the Supervisor stack pointer.

Other special-purpose registers include the program counter (PC) and the status register (SR). The PC holds the address of the current instruction and is 32 bits wide. The 68000 only uses the lower 24 bits to specify memory addresses (23 of them go out over the address bus). By coincidence, that is the same direct addressing space as the IBM 370 mainframe computer. Most of those memory locations are free; not many have a dedicated purpose. The lowest 8 bytes hold the reset vector. Other addresses in the bottom 1024 bytes are used for interrupt vectors, error vectors, and exception vectors in general (explained in Chapter 7).

The status register is 16 bits wide. The low byte of the SR is called the User byte or condition codes register (CCR). On many other microprocessors this is the flags register. Five bits of this byte hold information about the last operation performed by the microprocessor.

The high byte is called the System byte. Five bits of this byte contain information about the status of the microprocessor such as what priority of interrupt to acknowledge, whether the CPU is in user state, and whether the trace mode is on.

ARITHMETIC LOGIC UNIT

The part of the microprocessor that does the actual computing as most people use the word (meaning calculating and figuring) is the ALU. But when you look at it objectively, the ALU is no more the heart of the chip than is the data bus, the register, or the decorder. They all need each other.

Data from memory or the registers is routed through the ALU, where the mathematical and logical operations are performed. Then the processed data is sent on to the final destination.

The 68000 has three ALUs: one for data and two for addresses. There is a 16-bit ALU that makes all the data calculations and single pass evaluation of the 16-bit data. Thirty-two-bit data operations are done in two passes, first at the lower word and then at the upper word.

Two other internal ALUs are each 16 bits wide and are used together to calculate addresses (that is, to find operand effective addresses: see Chapter 4 for more detailed explanations). The effective address (EA) is the final result that is generated from the instruction data and the addressing mode. It is necessary to use two such ALUs because the addresses are 32 bits wide. With powerful addressing modes, this kind of calculation can take up a lot of time; having dedicated ALUs speeds it up.

The address calculation and the 16-bit data calculation can take place at the same time. This approach is a sort of parallel execution.

DECODER

The decoder is the microprocessor part that interprets instructions. It breaks up the patterns of 1s and 0s that make up machine language and tells the rest of the microprocessor what to do. In some ways,

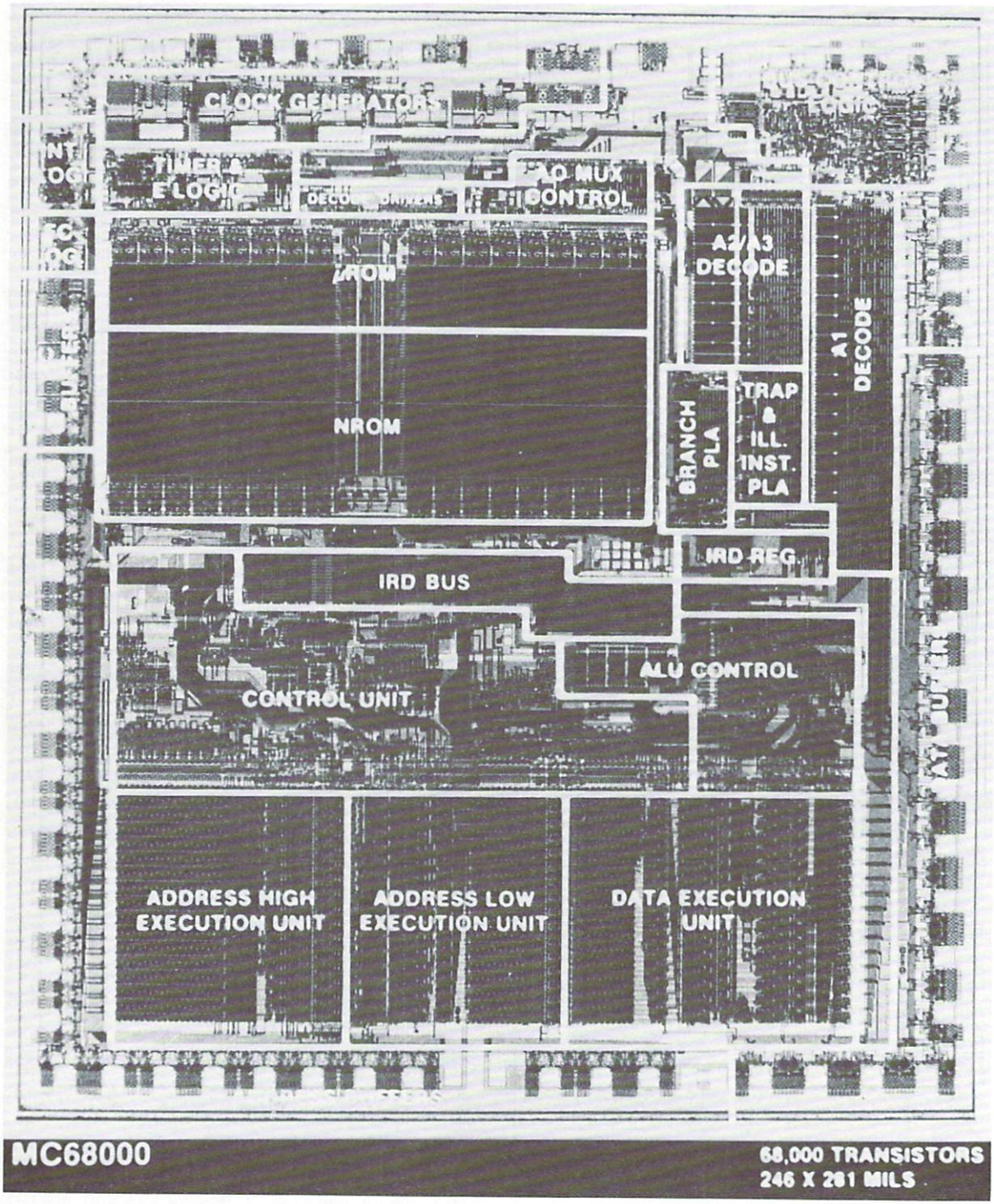


Fig. 2-11. 68000 floor plan showing function of the chip regions.

it is an even smaller microprocessor within the 68000 microprocessor. Studying the design of the decoder quickly brings to light the worlds within worlds of a microprocessor.

One of the first decisions microprocessor designers have to make is between hard-wired random logic and microprogramming. Early microprocessors were generally random logic. The designers simply decided what they wanted the chip to do and then found a way to string the microscopic wires together to do it.

More advanced microprocessors like the 68000 are often microcoded. They have, in effect, a smaller microprocessor that runs the rest of the larger microprocessor. There is a ROM memory that is programmed with tiny instructions to tell the microprocessor what to do (look at Fig. 2-11 to see what these ROMs look like). This simplifies the actual design of very complicated chips; software takes the place of very tangled hardware. Designing, testing, and fixing the chip all become easier. The entire microprocessor chip doesn't have to work the first time.

The tiny microprocessor is called a microsequencer. Its instructions are simpler than machine language. They involve elemental actions (called microwords) such as sending a signal to a certain gate or unlatching a particular bit of the status register. Microwords are built into microroutines that become the instructions of assembly language. Microroutines can have branches and conditions just like assembly language. You do not have to ever concern yourself with microprogramming.

Another major advantage of microcoding is that upgrading the chip will be much easier because the designers only have to change the final processing step for the onchip ROM. A new CPU chip, with different or more powerful instructions and addressing modes, can be designed simply by changing the microcode.

Microcoding on chips takes up as much as 20% more space than hardwiring because there will be circuits and gates that aren't used efficiently. That disadvantage is far outweighed by the ease of making and changing the chip in the first place.

There are two types of microprogramming: horizontal and vertical. Horizontal is more direct; a single bit of the microword may enable a register. Horizontal microwords are long and require wide microbuses and storage facilities. All of this extra real estate, as chip designers refer to it, adds to the cost of the chip. Nor is horizontal microprogramming a particularly efficient scheme. For example, a full microword with just a bit in the 2 position might be required just to enable register 2.

Vertical microwords encode the information for 16 registers in 4 bits. This scheme is slower, because the microword itself has to be decoded, but it does take up less chip area.

Believe it or not, both forms of microcoding are used in the 68000. That dual use means that the 68000 has *nanocode*. The microcode information points to the microsubroutines in nanocode which actually do the routing, selecting, and directing. Because of the two levels, the microcode routines can share subroutines of nanocode instead of having to keep them in several different places.

PREFETCH QUEUE

One special design feature of the 68000 is an instruction prefetch queue. While one instruction is being executed, another can be decoded and another fetched. While this won't always speed execution of a program (a branch or jump may eliminate the utility of the instructions grabbed) it can still yield significant improvement in most circumstances. The conditional aspect of a jump may not allow knowledge beforehand of what instructions must be used.

The queue is fairly intelligent. It will try to stay just as full as is useful. When a conditional jump is detected, it will grab the instruction after the jump and the one that may be jumped to. The unneeded operation is ignored and the useful one performed. There are special attractions to the prefetch queue. The Move Multiple Registers instruction uses it to speed the data transfers, fetching one while decoding another, so each takes only the time necessary to fetch the next code. The prefetch queue keeps the bus busy about 90% of the time: far better than that of chips without such a queue.

ADDRESSING MODES

Every microprocessor has certain ways of addressing operands. Those ways are called *addressing modes* and range from simply including data in the instruction to complicated, calculated addresses that are built of original values, displacements, and indexes. These modes allow the programmers to find what they need within the 16 linear megabytes of the 68000's addressing range.

The 68000 has a set of 14 addressing modes. These are explained in detail in Chapter 4 and are listed in Fig. 2-12.

DATA TYPES

The 68000 can work with bits, nibbles, bytes, words, and long-words. These 5 data types work with many of the instructions and provide quite a bit of flexibility for the programmer. Chapter 6 provides information on which data types can be used for each instruction.

INSTRUCTIONS

The 68000 has a large and orthogonal set of instructions. They are listed in Fig. 2-13 and are detailed in Chapters 5 and 6. These are the basic operations that the 68000 can perform and range from no operation at all (the NOP instruction just marks time), to moving data (the particularly flexible MOVE instruction), to multiplying two numbers together (which 8-bit chips cannot do with a single instruction).

The flexibility of the MOVE instruction is typical of the entire instruction set. The instructions are simple, with the programming variety and power coming from the addressing modes, choice of operand size, and wealth of registers. The aim of making the operation of different functions similar is taken seriously. This is called *orthogonality*. For instance, different types of addition instructions have the same addressing modes.

OPERATING MODES

After you realize you're working with a 16/32-bit chip (meaning a chip from a family that works with either 16 or 32 bits of information at a time), the next thing to realize is that the 68000 chips have two basic modes: User and Supervisor. This is one of the ways in which the 68000 more closely resembles a minicomputer than it does classic

1. Data Register Direct
2. Address Register Direct
3. Address Register Indirect
4. Address Register Indirect with Postincrement
5. Address Register Indirect with Predecrement
6. Address Register Indirect with Displacement
7. Address Register Indirect with Index
8. Absolute Short Address
9. Absolute Long Address
10. Program Counter with Displacement
11. Program Counter with Index
12. Immediate
13. Quick Immediate
14. Implicit

Fig. 2-12. Addressing modes.

1. ABCD	18. DIVU	35. NOT	52. TST
2. ADD	19. EOR	36. OR	53. UNLK
3. AND	20. EXG	37. PEA	
4. ASL	21. EXT	38. RESET	
5. ASR	22. ILLEGAL	39. ROL	
6. Bcc	23. JMP	40. ROR	
7. BCHG	24. JSR	41. RTE	
8. BCLR	25. LEA	42. RTR	
9. BRA	26. LINK	43. RTS	
10. BSET	27. LSL	44. SBCD	
11. BSR	28. LSR	45. Scc	
12. BTST	29. MOVE	46. STOP	
13. CHK	30. MULS	47. SUB	
14. CLR	31. MULU	48. SWAP	
15. CMP	32. NBCD	49. TAS	
16. DBcc	33. NEG	50. TRAP	
17. DIVS	34. NOP	51. TRAPV	

Fig. 2-13. 68000 Instruction set.

microprocessors. User mode is the most commonly used mode, particularly for application programs.

The difference occasioned by Supervisor mode is simple; Supervisor mode allows more freedom, more access to memory, and more executable instructions. Operating systems and systems software, in general, use the supervisor mode (which is explained in more depth in Chapters 3 and 7). The User mode of all 68000 chips is kept very similar, to make software compatibility between CPUs as complete as possible.

SPEED

The 68000 can run at 4, 6, 8, and 10 MHz (depending on the chip you buy). These choices are listed in Fig. 2-14. The speed is coded as a number

CPU Number	Speed Frequency	Clock Period
68000L4	4 MHz	250 microseconds
68000L6	6 MHz	167 microseconds
68000L8	8 MHz	125 microseconds
68000L10	10 MHz	100 microseconds

Fig. 2-14. 68000 speeds.

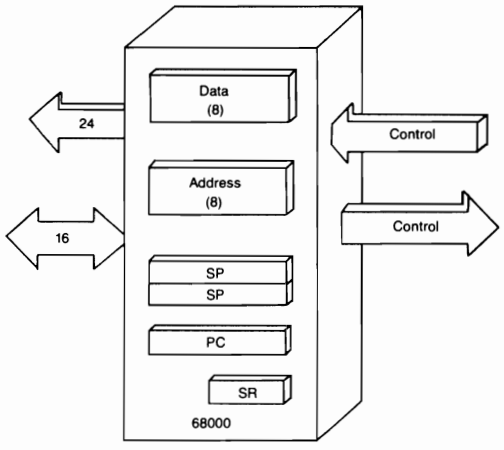
after the 68000 code. for instance, the code MC68000L4 runs at 4 MHz and the MC68000L10 runs at 10 MHz.

The clock periods for those chips are 250, 167, 125, and 100 ns. The shortest instruction, copying one register into another, takes four clock cycles. The longest, 32 by 16 signed division, takes up to 170 clock cycles.

INTERRUPTS AND EXCEPTIONS

The 68000 has both hardware and software interrupts. There is also a trace mode for debugging (and a trace bit in the status register). On the 68000, however, all special cases are lumped into a large class called *exceptions* that includes everything from illegal instructions to external interrupts. Exceptions are explained in Chapter 7. When an exception is generated, the regular processing ends, and the exception service routine is processed. The address for that routine can be found in a number of ways, depending on what sort of exception occurred.

3



Registers

A MICROPROCESSOR MOVES AND STORES BITS of information. There are three basic places a particular bit can be stored: mass storage, memory chips, and registers. Figure 3-1 lists the three and some characteristics of each.

The first form of storage, mass storage, is actually a large category of devices including magnetic tape, magnetic disks, and optical disks. These devices can store huge amounts of data but are comparatively slow. After the microprocessor asks for a particular bit of information it must wait some time before receiving the information.

The second storage form, memory chips, is much faster and more expensive than mass storage. Whenever someone tells you that a computer has so many K of memory, they are referring to the memory chips that are built into the computer itself. These chips can be either ROM (which have permanent information and therefore can only be read from, not written to) or RAM (that can be read from, written to, or erased).

The third storage form is based on the same technology as memory chips. The difference is that registers are on the microprocessor chip itself: they are in essence a small memory chip built into the CPU.

REGISTER ADVANTAGES

Registers are the fastest form of storage for two reasons. First, they are closer to the microprocessor elements that manipulate data and so the signals don't have to run out to a distant chip and back. Second, because the number of registers is far smaller than the number of memory addresses, it will take fewer bits of instruction to specify a register. That, in turn, means the instructions used with registers can be shorter than those used with memory chip addresses. Smaller instructions execute faster than long instructions and so programs that work with registers execute faster than those that depend on outside memory.

Programmers eagerly consume registers. The 6800 had two registers, A and B, for data work and one index register for addressing. By designing enough—but not too many—registers onto a microprocessor, you can improve its performance:

Type	Devices	Speed	Expense
Mass Storage	Disk Drives, Tape Drives	Slow	Low
Memory Chips	RAM, ROM, Bubble Memories	High	High
Registers	RAM-on-chip	Very High	Very High

Fig. 3-1. The three types of memory.

too few and the chip will be forced to use only slow memory instructions; too many and the advantages of registers will disappear.

REGISTER TYPES

Registers can be dedicated to a particular task, or they can be flexible, which gives them the ability to handle many different tasks. These two types of registers are called special-purpose and general-purpose. Special-purpose registers (also known as dedicated registers) can only work in a certain way with certain instructions for a certain purpose.

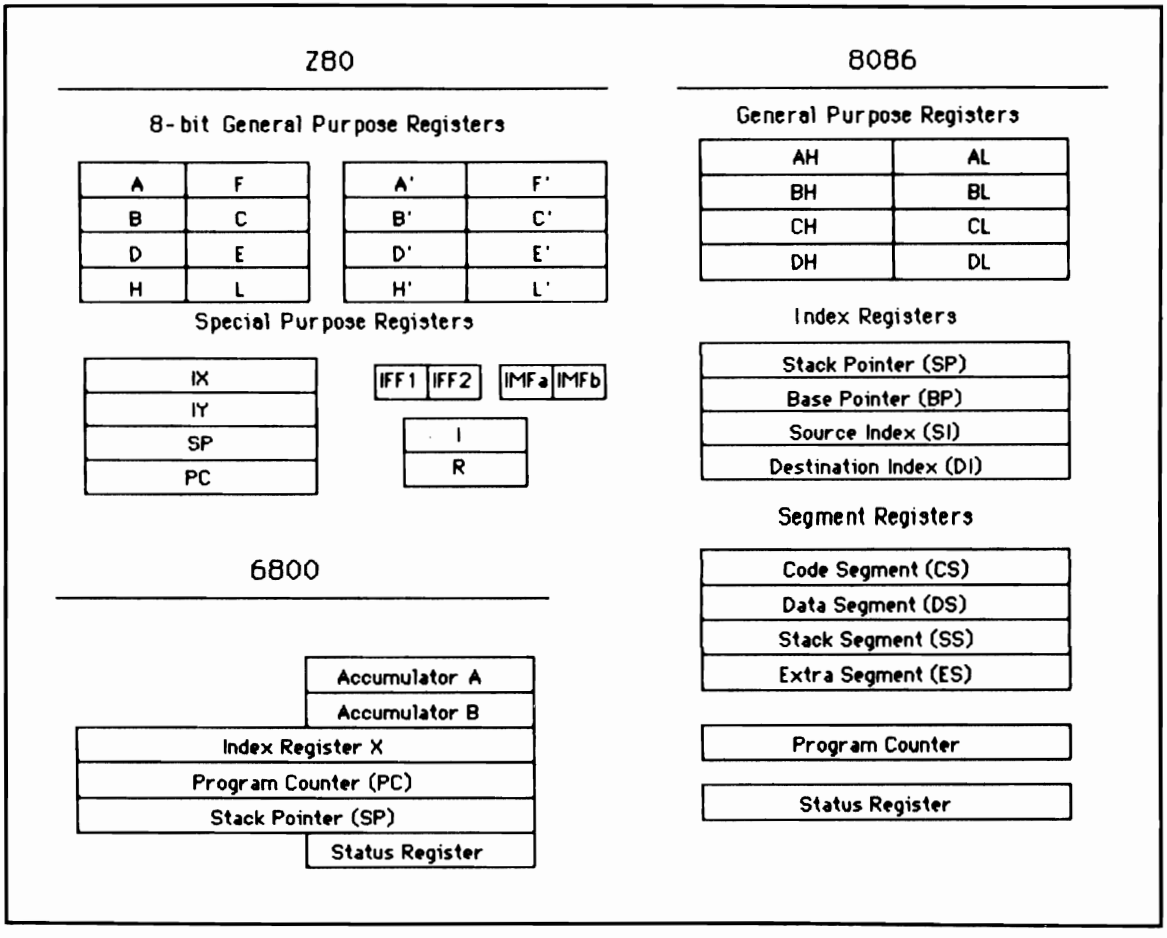


Fig. 3-2. Registers of some popular microprocessors.

Figure 3-2 shows the register sets of some popular microprocessors. Figure 3-3 shows the 68000 register set. Most other 16-bit microprocessor register sets are either smaller or less flexible than the 68000's set. Eight-bit microprocessors typically have much smaller register sets.

The 68000 leans heavily on general-purpose registers. They are harder to design into a chip, but they make that chip easier to program.

68000 GENERAL-PURPOSE REGISTERS

In many ways, a 68000 is a 32-bit microprocessor. The general-purpose registers (shown in Fig. 3-3) are a prime example of that: they are 32 bits wide. The general-purpose registers are divided into eight data registers, seven address registers, and two stack pointers. (The special-purpose program counter (PC) is also 32 bits wide, even though only

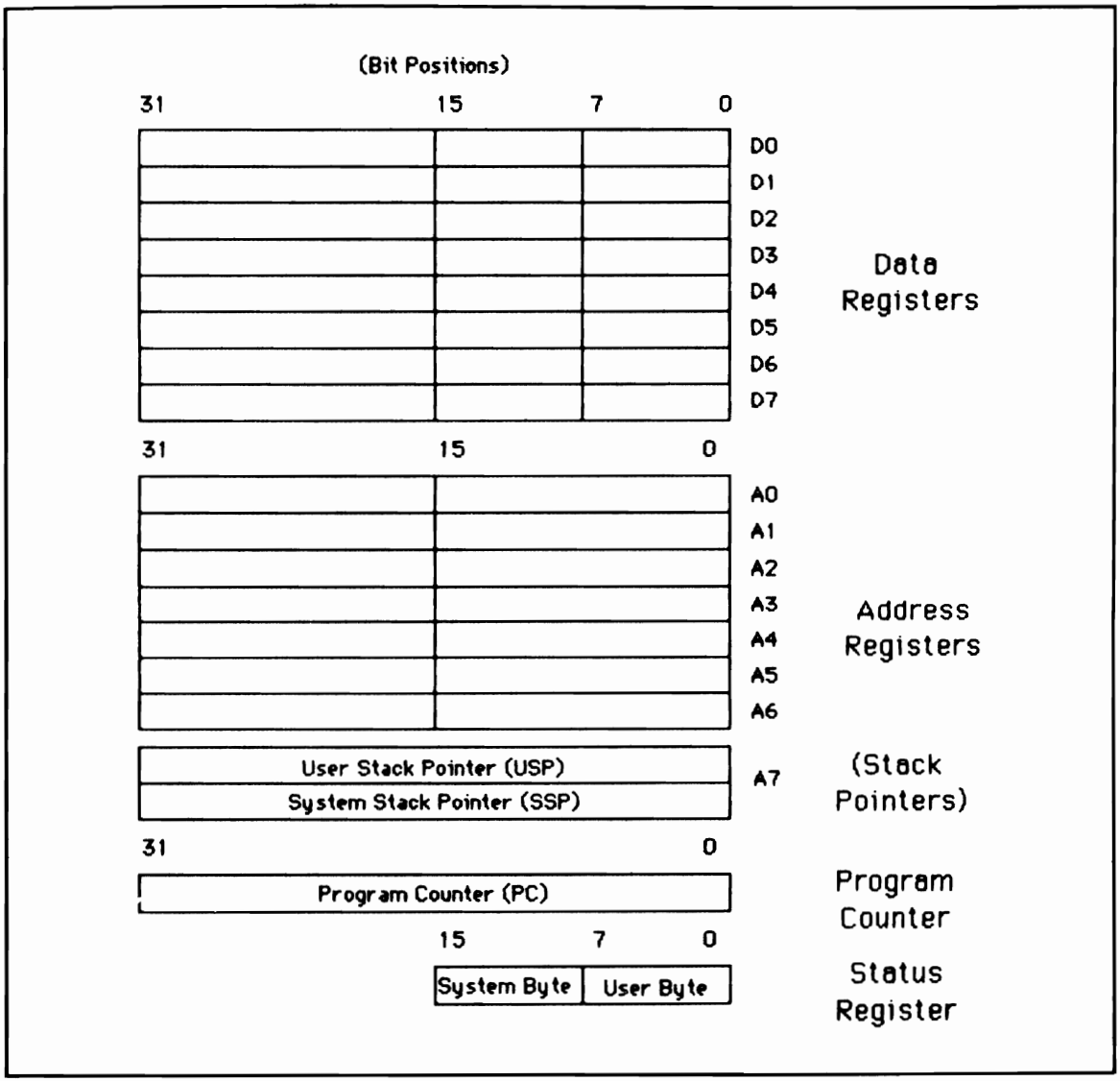


Fig. 3-3. 68000 register set.

the lower 24 bits are sent out to become the 68000 address bus.)

Data Registers

There are eight 32-bit data registers as shown in Fig. 3-3. They are labeled D0 through D7. These registers can be used with bytes, words, or long words. The data always sits as low as it can in the register: bytes run from bit position 0 to bit position 7, words from bit position 0 to bit position 15, and long-words from it position 0 to bit position 31. The data registers perform the work that more specialized registers such as index registers and accumulators handle on many other microprocessors. Because the 68000 lets programmers decide how to use the general-purpose registers, they can have as many as 7 accumulators (which handle arithmetic) or none at all, whichever is more useful at a particular point within a program. The task of a data register can be changed instantly and whenever desired during the execution of a program.

Both data and address registers are general purpose. They can be used for many computing purposes, but the data registers are more flexible for data storage and the address registers are well-adapted to storing addresses. Bytes or words in the data registers are only sign extended in a few exceptional cases. Words loaded into an address register are automatically sign extended. When a data register has an operand written into it or read from it, only the operand is affected. All other bit positions are left unaffected.

All of the data registers can work as accumulators for arithmetic. They can also work as index registers or counters. This flexibility makes them more powerful than even a similar number of registers would be on an 8-bit microprocessor.

Address Registers

The eight 32-bit wide address registers (shown in Fig. 3-3) are labeled A0 through A7. They are also general purpose registers. While they cannot handle byte-size data, they are otherwise quite similar to the data registers. Word operands sit in the low-order word of address registers. Long-words occupy the entire address register.

There are some differences between data and address registers. For instance, the address registers sign-extend words automatically while the data registers do not. Another difference appears between data register D7 and address register A7. Data register D7 is just like the other data registers, but address register A7 is quite different from the rest of the address registers. A7 has the special function of stack pointer. That function is explained in detail in the special-Purpose Registers discussion that follows.

Finally, when a word is written to an address register, the entire register is affected (the operand is size-extended to fill the register). Within a data register, only the operand is affected.

68000 SPECIAL-PURPOSE REGISTERS

The 68000 has several special-purpose registers (shown in Fig. 3-3) that are used for program control and support. Most of these registers are quite similar to standard registers found on other microprocessors. The existence of two stack points and of an extended status register adds significantly to the advantage the 68000 has over 8-bit chips.

The 68000 has two major modes of operating: Supervisor and User. A bit-position in the special-purpose condition codes register (also known as a flag) controls which mode it is in. The Supervisor mode is also known as System mode and the User mode is known as Normal mode. If you want to be able to use any instruction, keep the microprocessor in Supervisor mode. The User mode is more restricted.

The basic reason for dividing the 68000 operation into two modes is to let the system software, the operating system, have complete control of the computer (in Supervisor mode) while the applications software, the particular software that handles jobs, have only partial control (User mode). That keeps application software from controlling the system.

Stack Pointers

There are two stacks, both controlled by the

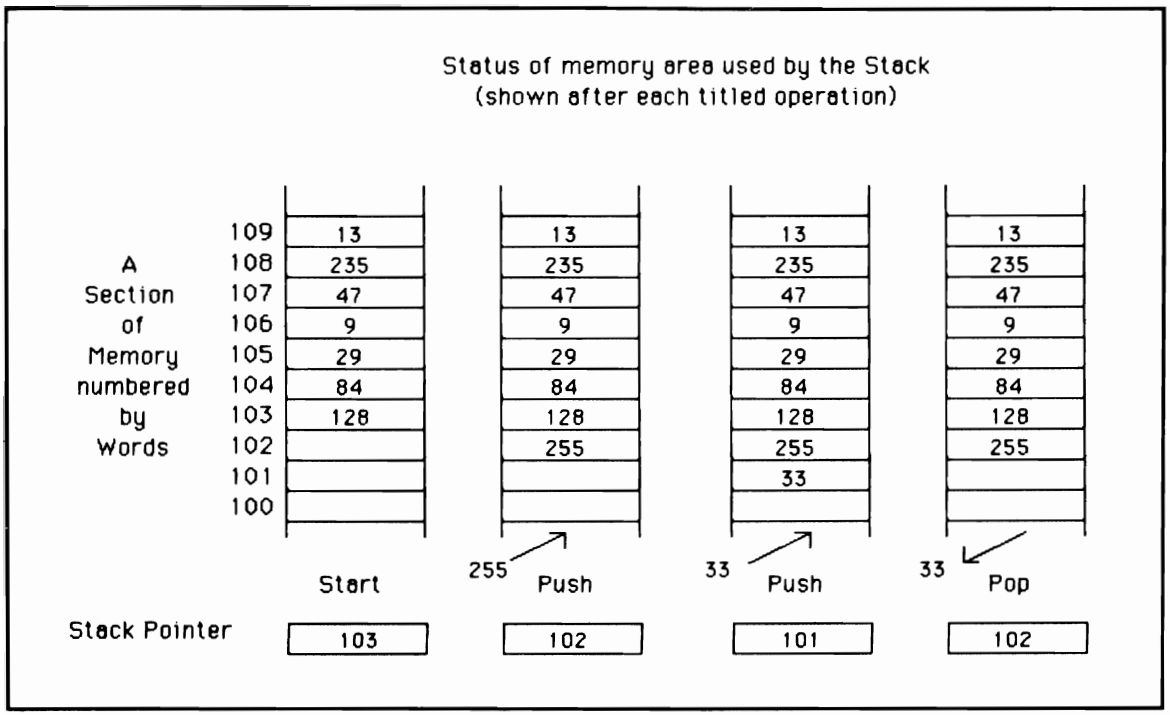


Fig. 3-4. An example of stack manipulation.

stack pointers at A7 in the register bank: the Supervisory stack and the User stack. The stack pointer that you will find when you look at A7 is the active stack pointer for the mode the CPU is in.

A stack pointer is a special-purpose register that, like the program counter, is a pointer to memory. A stack is a data structure that is useful for many purposes such as interrupt processing. Figure 3-4 shows how items are moved into and out of a stack.

LIFO (Last In First Out) is an acronym used to describe stacks. Each item entered into a stack is *pushed onto* the stack. The item entered before the most recent item is now covered up. A newer item is always pushed on top of the previous top item. An item removed from the stack is *popped off* of the stack. Only one word at a time can be popped from a 68000 stack, and that word is always the most recent addition to the stack. The last item in is the first item out. Although the typical real-life example of a stack given by computer science texts is the stack of plates in a restaurant, you should realize that stacks can grow up or down in memory.

As mentioned, stacks are particularly handy for servicing interrupts or subroutines. When the CPU operation is interrupted, the programmer will want to remember what the present CPU state is so that he can return to it after the interrupt is taken care of. By executing push instructions that deposit vital register contents on the stack, the programmer can be sure that the CPU will be able to remember the important aspects of its current state. When the interrupt is over, the important data (including PC and certain register values) can be recovered from the stack. In fact, some of the 68000 instructions are designed to do this automatically (see the RTR instruction for an example). If such data were only saved in a particular special register, what would the CPU do if a second interrupt broke into the first interrupt routine? It couldn't just save the new status in the special registers because the data necessary to eventually recover from the first interrupt would be lost.

Even a hardware stack could be overwhelmed

by a relatively small number of such nested interrupts or routines. So most microprocessors, including the 68000, now use a software stack. All they need is a special-purpose register that points to an address in memory along with the CPU logic that will automatically decrement the register value when data is pushed onto the stack (the stack grows downward in memory toward smaller addresses) and increments the register value when data is popped off of the stack.

This special register, called a stack pointer contains the address of the top of the stack. Unfortunately, because the 68000 stack rows downward in memory, the top of the stack has the lowest address of any stack location. When you push a value onto the stack, the values within the stack don't actually get pushed any deeper into memory, they just get pushed further away from the changing stack pointer value. That is a tough concept to grasp at first; but it will soon seem like second nature.

The stack is just an area of memory: it isn't delimited by any boundary. If you popped enough data off of the stack you would go beyond all of the pushed on values, but you would still get data coming out. Such data would just be the values at the addresses the stack pointer pointed to. And because the stack is implemented in memory, it can grow to a huge size (theoretically as large as the full memory space). Because of that flexibility, a software stack can handle almost any number of nested routines.

The 68000 actually has two stack pointers: User stack pointer (USP) and Supervisor stack pointer (SSP). They are shown in Fig. 3-3 as address register A7. Whenever you; try to write or read that register, you will actually get either the USP or the SSP. What decides which one you get? The active stack pointer will be in A7 and the mode of the 68000 controls which stack pointer is active. The following paragraph explains that control.

To isolate system software from application software, the 68000 has an advanced facility called *modes.* There is a bit in the status register (explained next) called the Supervisor or S flag. If that bit is set, the CPU is in Supervisor mode; if it is cleared, the CPU is the User mode. In Supervisor mode, the A7 register will contain the Supervisor stack pinter. In User mode, A7 will contain the User stack pointer. The active stack pointer is the one used (unless an instruction specifies USP or SSP) and the other stack pointer is not directly available to inspection or change (except through the privileged USP instruction). In most cases, the programmer doesn't have to address a particular stack pointer: the CPU decides that automatically. But the programmer must be aware of which is being used because they can and most often do point to different parts of memory and so will point to different values.

The MOVE USP instruction is built into the instruction set so that programs in the Supervisor mode can see and change what is in the User stack pointer. The LINK and UNLK instructions are very powerful, complex routine controllers that depend on stack manipulations and *frames.* Look these instructions up in Chapter 6 for detailed explanations.

Status Register

All computers must have some way of knowing what has happened in the recent past. The normal way of doing this is to have a status register. This register is a collection of a number of small registers (many only a single bit wide). Single bit positions of the status register are commonly called flags.

Flags are the essence of computers. The ability to make decisions without human intervention is the vital factor that makes computers the powerful machines they are and computers make those decisions by testing the flags. Even the simplest processor has a few flags because of their critical role in branching and condition testing. The versatility of a computer program lies in the decision capability designed into it; that versatility and capability rest, in turn, on proper use of flags.

The 68000 status register (shown in Fig. 3-5) is 16-bits wide and is divided into two different parts: the system byte and the User byte.

User Byte. The lower (less significant) byte of the status register is also known as the condition codes register. This byte contains 5 active flags (in

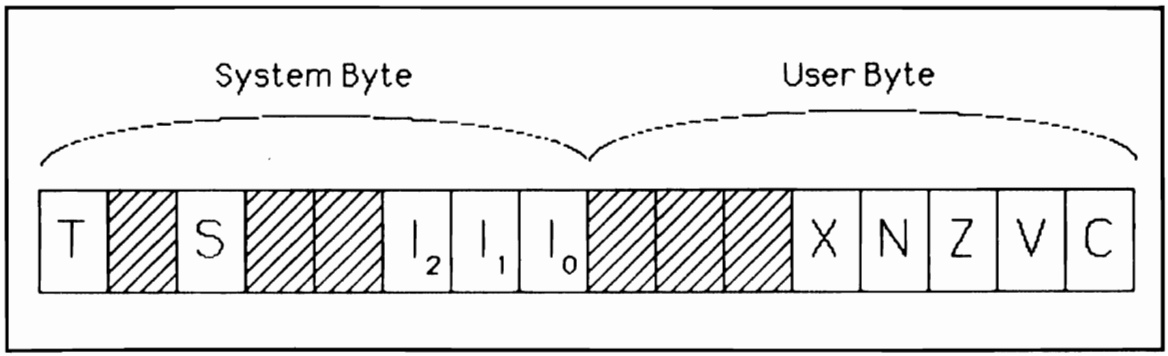


Fig. 3-5. Status register.

the least significant 5-bit positions) and 3 bits that aren't used. The condition codes register corresponds directly to the flags register on most other microprocessors.

Each flag is affected by some instructions but not by others. Also, not all instructions affect a given flag the same way. In fact, the effects are sometimes quite different. See the individual instructions descriptions in Chapter 6 for a detailed description of the flag effects of each instruction.

Some terms that you will hear repeatedly in microprocessing and particularly in reference to flags are defined next.

Set. A bit position that has a 1 value put into it has been set. Another word for a 1 is True. Unfortunately, the word set is also used as a more general verb such as set to the value shifted out of the register in which case it doesn't refer only to 1s.

Cleared. This is the opposite of set. A bit position or flag has been cleared if a 0 has been put into it. Unfortunately (again) the terms set and cleared can even be mixed as in "A cleared bit has been set to 0." Try not to worry about it. The words false and reset are also sometimes used to mean clear.

Undefined. A flag that is undefined can be either 0 or 1.

Not Affected. This means the same as none under the Chapter 6 descriptions of individual instructions. This means the flag retains whatever value it had before the instruction was fetched for execution.

The 5 active flags have the following names, positions, and uses:

Bit Position 0 is the *C flag* (for Carry). There are three ways this flag is used. First, and simplest, is that any carry out of the most significant bit of an operation will be represented here. After the instruction, a 1 in this flag means an operation resulted in a carry; a 0 means no carry occurred. Figure 3-6 shows such an operation.

Borrows are also shown in this flag. As explained in the Chapter 6 descriptions of subtraction and compare operations, an instruction that could cause a borrow (and doesn't deal with carries) will interpret the C flag as the borrow flag. After the instruction, a 1 in the flag means the operation caused a borrow; a 0 means no borrow occurred. Figure 3-7 shows a subtraction with a borrow.

Finally, the rotate and the shift instructions frequently deposit the bit values that fall off the end of the operand in the C flag. Figure 3-8 shows both a shift and a rotate that each result in a new C value.

Bit Position 1 is the *V flag* (for oVerflow). The V flag seems simple to understand but it is more complex than it looks. It tells when an operation overflowed the register. When an arithmetic operation result is larger than a register can hold, the V flag is set to one. This is done by setting the V flag equal to the exclusive-OR value of the carries into and out of the most significant bit of the register. That is, when the carry out is different from the carry in, the V flag is set. Otherwise, the V flag is cleared. This operation is shown in Fig. 3-9.

Many other operations also affect the V flag. Moves, rotates, and multiplies, for instance, clear the V flag (put a zero into it).

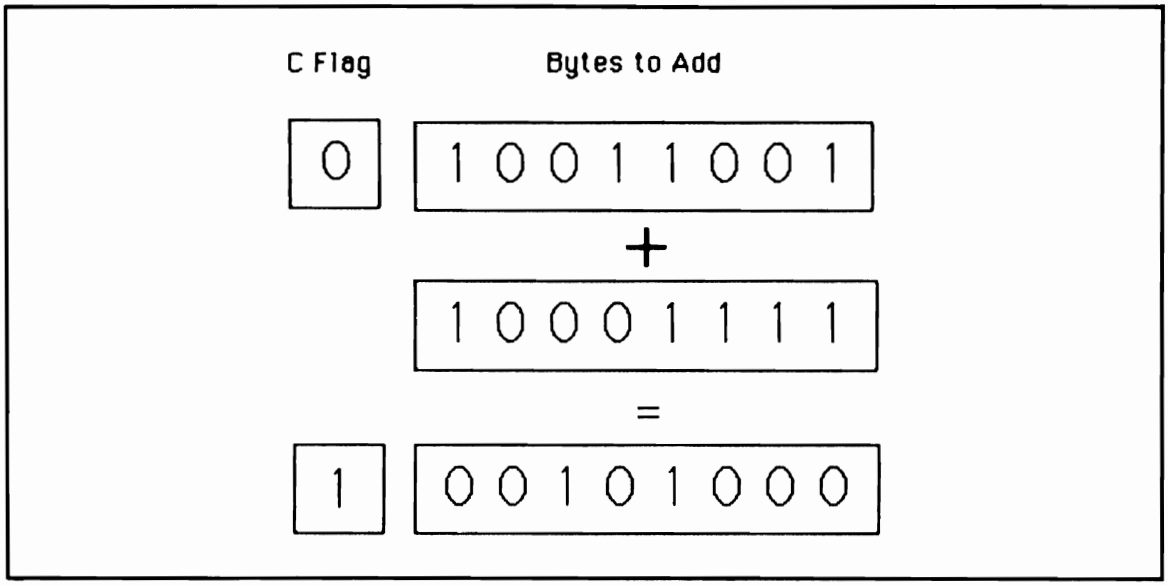


Fig. 3-6. Example of a carry.

Bit Position 2 is the *Z flag* (for Zero). The Z flag is set to one if the result of an operation is zero, otherwise the flag is cleared (set equal to zero).

Bit Position 3 is the *N flag* (for Negative). This flag is set to one if a signed arithmetic operation or arithmetic shift produces a negative result. Otherwise, the N flag is cleared (to zero). N follows the most significant bit of the operand whether that operand is 8, 16, or 32 bits long. Figure 3-10 shows how the most significant bit gets transferred to this flag.

Bit Position 4 is the *X flag* (for eXtend). This

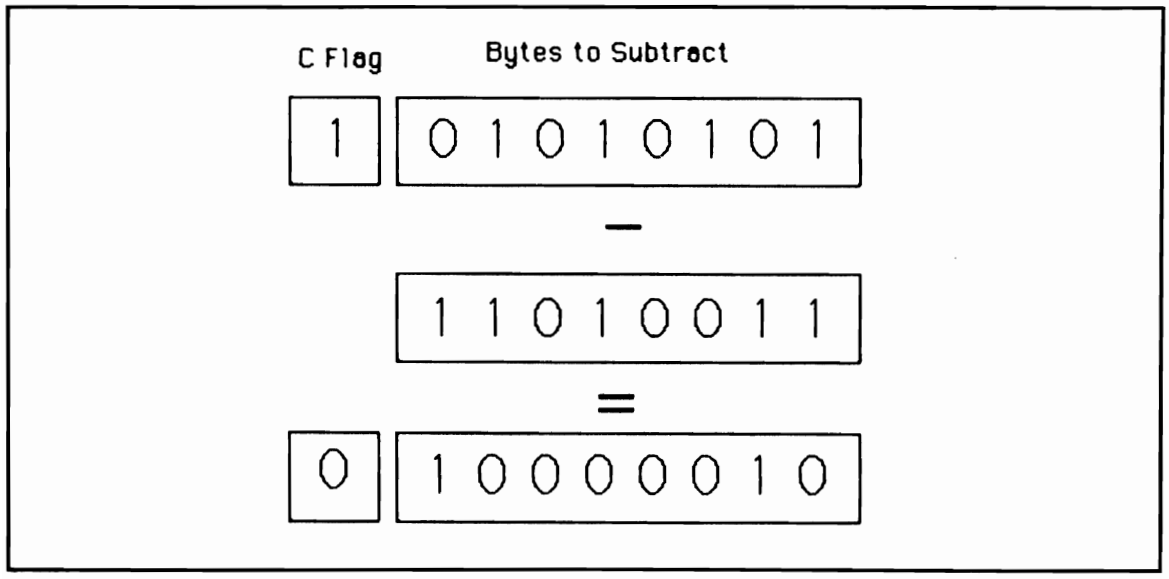


Fig. 3-7. Example of a borrow.

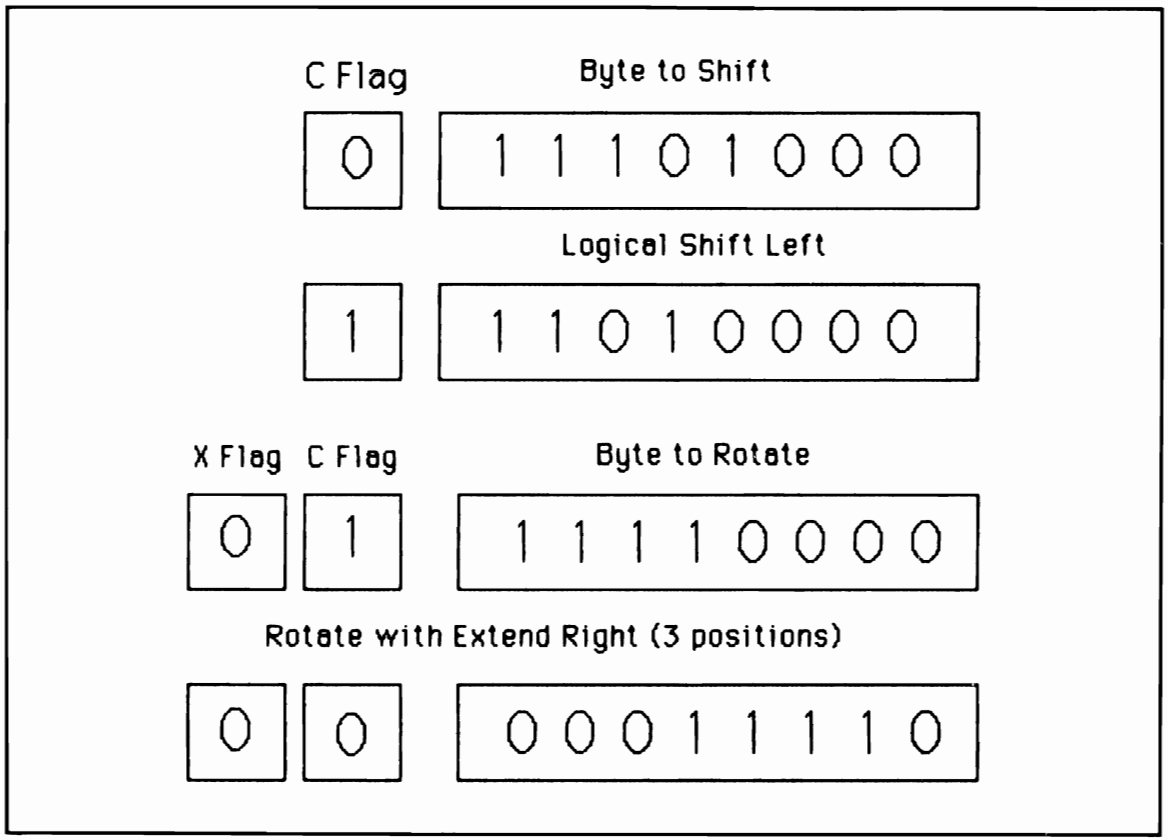


Fig. 3-8. Examples of a shift carry and a rotate carry.

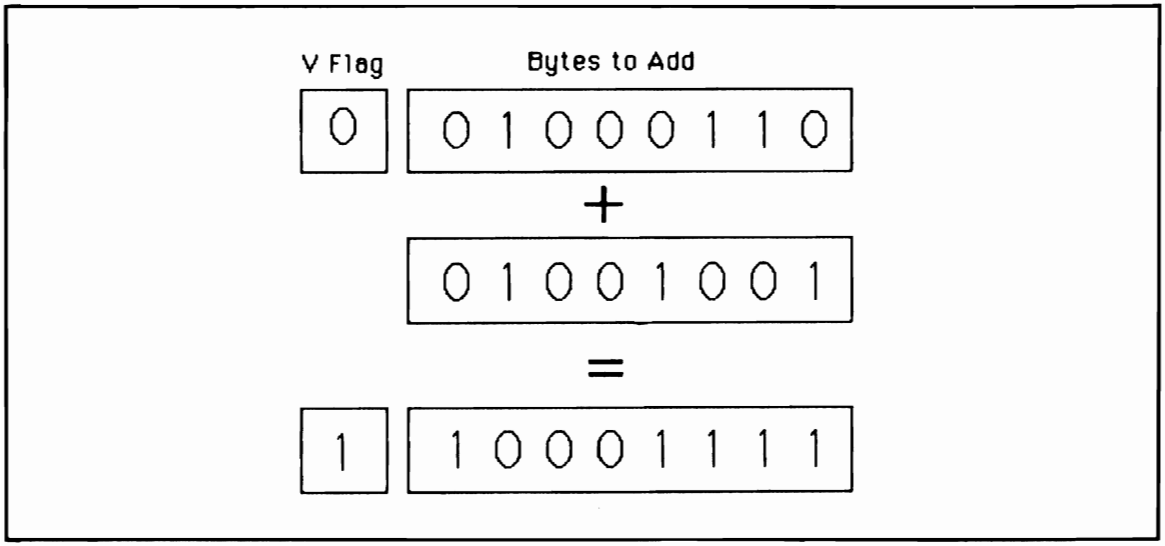


Fig. 3-9. Example of an overflow.

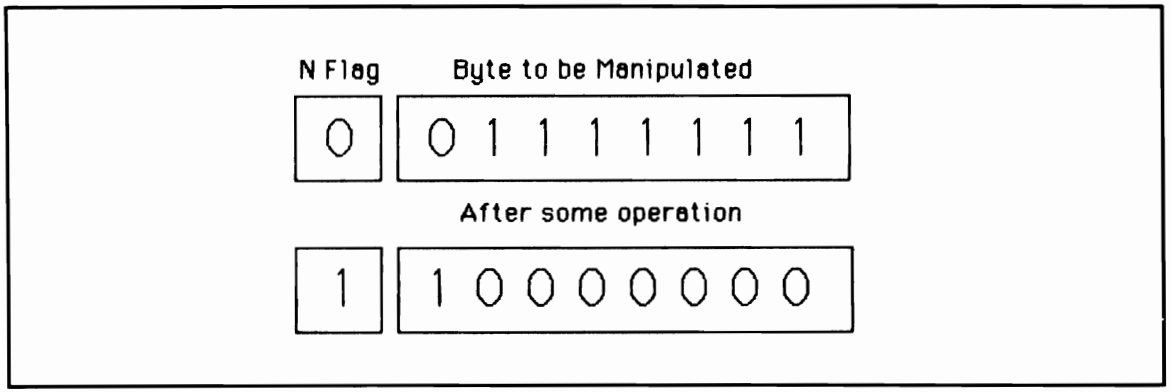


Fig. 3-10. Example of an N flag change.

flag is a special sort of carry flag and is typically used for multiple-precision arithmetic operations. It is affected by special "with-extend" add, subtract, negate, and shift instructions. These instructions set X in the same way as they set C. The relevant instruction descriptions in Chapter 6 explain the use of the X flag.

System Byte. The more significant byte of the status register is called the system byte. It contains an interrupt mask (that is 3 bits wide), a Supervisor/User state flag, and a Trace mode flag.

The *interrupt mask* comprises bit positions 8, 9, and 10 of the status register. These three bits can hold any value from 0 to 7. When outside devices want to interrupt what the 68000 microprocessor is doing, they send signals on three priority pins. Those signals can also represent any number from 0 to 7. The interrupt will only be recognized—that is, will only be effective—if the interrupt request signal number is equal to or less than the interrupt mask number.

By manipulating the interrupt mask bits, the programmer can control which devices can interrupt the 68000. Chapter 7 explains interrupts and all exceptions in more detail.

The *T flag* is in bit position 15. If it is set to 1, the 68000 is in Trace mode. This special environment generates an exception after every instruction. That lets the programmer force the CPU through the program one instruction at a time. Such control is vital to program debugging.

The *S flag* (which stands for Supervisor) determines whether the CPU is in Supervisor or User mode. This dichotomy was explained in the description of Supervisor and User stack pointers earlier in this chapter.

Basically, User and Supervisor mode differ in two ways:

1. Active Stack Pointer. Address register 7 is always the stack counter. But in User mode that register will have the User stack pointer value while in Supervisor mode the Supervisor stack pointer will be in that position. Whichever mode you are in, any reference to address register 7 can only elicit the active stack pointer. Figure 3-3 (earlier in this chapter) shows this arrangement.

2. Instruction Set. The Supervisor mode has several instructions that the User mode does not. These instructions—called privileged—cannot be executed in User mode and will only generate an exception. The individual descriptions in Chapter 6 will tell you which instructions are privileged and which are not.

Program Counter

A good case can be built that the program counter (known as the PC and shown in Fig. 3-11) is the most important of the special-purpose registers. Virtually every microprocessor has a PC. This 32-bit register monitors and controls the position of the microprocessor within the program. The data it contains is the address of the next instruction to be executed.

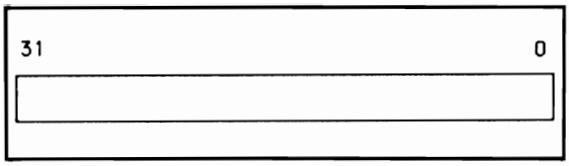


Fig. 3-11. Program counter.

At the beginning of each instruction, the PC value is sent to memory to fetch the next byte from the stored program. In the course of executing the instruction, the CPU automatically increments the PC. It is increased by as many bytes as the instruction is long. That variable increase ensures that the PC points at the next instruction in the sequence. This sequential execution is a foundation of almost all computer architectures. Even the NOP (No Operation) instruction increments the PC: that is so elemental that it is considered no operation.

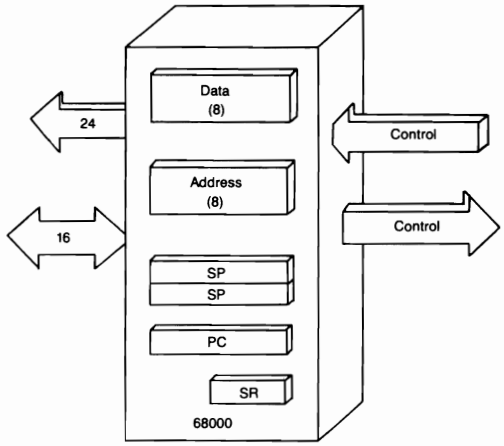
After any instruction, then, the PC holds the address of the next instruction. The exception to this rule is caused by jump, branch, or reset operations. These cause a new value to be directly fed into the PC: the new value will move the processing to a different point in the program. Returning from subroutines, similarly, requires the reloading of the original PC value.

Even though the 68000 PC is 32-bits wide, only 24 bits of it are wired through to the pins on the chip. The result is that only 24 bits can be used to address memory space. By keeping to only 24 lines, the 68000 designers were able to fit the chip into a 68-pin DIP. This is still a much larger package than most other microprocessors use. Still, the 24-bit address means the 68000 can address 16 megabytes of memory. That space in hexadecimal addresses ranges from 000000 to FFFFFF. It is 256 times larger than the 64K (65536) that most 8-bit microprocessors can address. The 68008 has a smaller memory space and the 68010 and 68020 have much larger memory spaces (see Chapter 8 for more information).

As is explained in Chapter 4, the memory is organized in a particular way. Words and long-words are found at even numbered addresses; bytes are located on the odd numbers or even numbers. Bits 1 through 23 of the PC become address lines A1 through A23. Bit 0 of the PC is manipulated internally with the operand length specified with in the instruction to make two *data strobe* signals.

For the hardware aficionados, those strobe signals (pulses on particular pins) are UDS bar (Upper Data Strobe), and LDS bar (Lower Data Strobe). Both strobes assert to move a word; to move a byte only one strobe is asserted (an even numbered byte strobes UDS and an odd-numbered byte strobes LDS).

4



Addressing

A COMPUTER, OR MICROPROCESSOR, PROCESSes data. It must be told both how to process the data and where to find the data. The instruction set of a microprocessor is a list of what the computer can do (the 68000 instruction set is detailed in Chapters 5 and 6). Some operations are self-contained; once the instruction has been decoded, it can be immediately executed. But others require a specified location that contains or will contain necessary data. These locations give the instruction something to add, somewhere to put the result, or something to move. The methods provided for finding the data are called addressing modes. Each mode offers the programmer a different way of directing the CPU in its reach for necessary bits, bytes, or blocks of information.

Early computers had only a few addressing possibilities. As the art of computer design has progressed, many designers have come to believe that having many flexible addressing modes is more important than having many instructions. A deft programmer can use different modes to create a huge number of different instructions from a basic instruction set.

Programmers quickly discover that, while instructions are important, understanding addressing is very important. In fact, it is a little silly to talk about instructions and addressing modes as though they are completely different things. Instructions are only as good as the addressing modes that they use to find and place operands. Addressing modes, if not accompanied by a clear and complete set of instructions, can add little more than complexity to the programming environment.

The 68000 has a large family of addressing modes. This chapter describes these modes after a couple of quick detours into operand sizes and the shape of memory.

OPERAND SIZES

Another subject that is directly related to addressing is operand size. A microprocessor can only work with certain selected chunks of data. The 68000 is

Name	Number of Bits
Bit	1
Nibble	4
Byte	8
Word	16
Long-word	32

Fig. 4-1. 68000 operand sizes.

quite flexible: it can work with bits, nibbles, bytes, words, long-words, or multiple long-words (1, 4, 8, 16, 32, or more bits at a time). Not all instructions can work with all operand sizes, however. Explicit instructions work with bytes, words, or long-words: Implicit instructions don't all use all three data sizes. Figure 4-1 shows the various data sizes.

Many single instructions can work with bytes, words, or long-words by simply changing the letter extension on the mnemonic. For instance, MOVE.B will move a byte, MOVE.W will move a word, and MOVE.L will move a long-word. Other microprocessors that have the ability to work with bytes, words, and long-words sometimes use different instructions for the various sizes.

Some special instructions on the 68000 allow it to handle special data sizes such as multiple registers or nibbles. The BCD instructions (listed in the Decimal Group in Chapter 5) work with groups of 4 bits (4 bits is a nibble). MOVEM can work with words or long-words and can send or receive the full register set.

THE SHAPE OF MEMORY

Memory can be shaped, or organized, in many different ways. The 68000's memory organization is detailed in the following descriptions and figures.

Registers

As has already been described, a small amount of the memory in a 68000 system is organized into registers. These registers are on the 68000 chip and are used for either specific, dedicated tasks or for fast, general tasks.

The eight data registers work with bits, bytes, words, or long-words.The eight address registers work with words or long-words. See Chapter 3 for the details of register organization.

The PC (program counter) works with long-words. The status register works with bits, bytes, or words. These registers are also described in detail in Chapter 3.

Memory

The simplest way to look at memory is as a series of bytes. This scheme is shown in Fig. 4-2. Each byte has an address that is a single bit larger than the previous byte. This is a common method of working with memory.

The 68000, however, can work with 16 bits at a time (its data bus is a full word wide). Memory is therefore also organized in words as shown in Fig. 4-3. The bytes are put side by side, with the lower addressed byte in the high-order position of the word. That may surprise you. What it accomplishes is that words are all addressed by even numbers (with the address also referring to the high-order byte of the word). Long-words appear with the high-order word first (lower in memory) and then the low-order word. This is shown in Fig. 4-4.

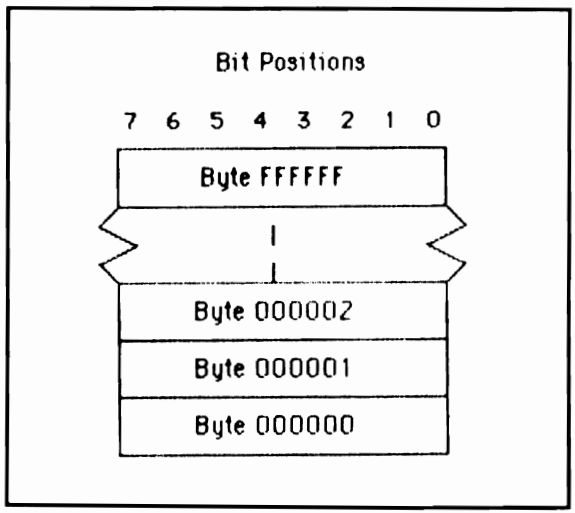


Fig. 4-2. Memory addressing (bytes).

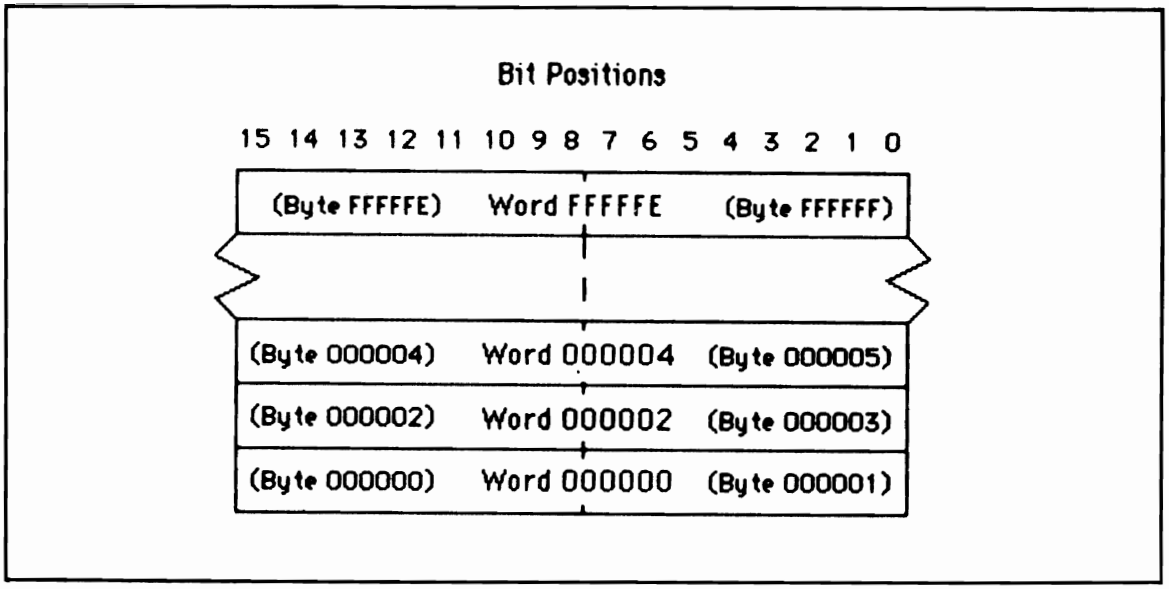


Fig. 4-3. Memory addressing (words).

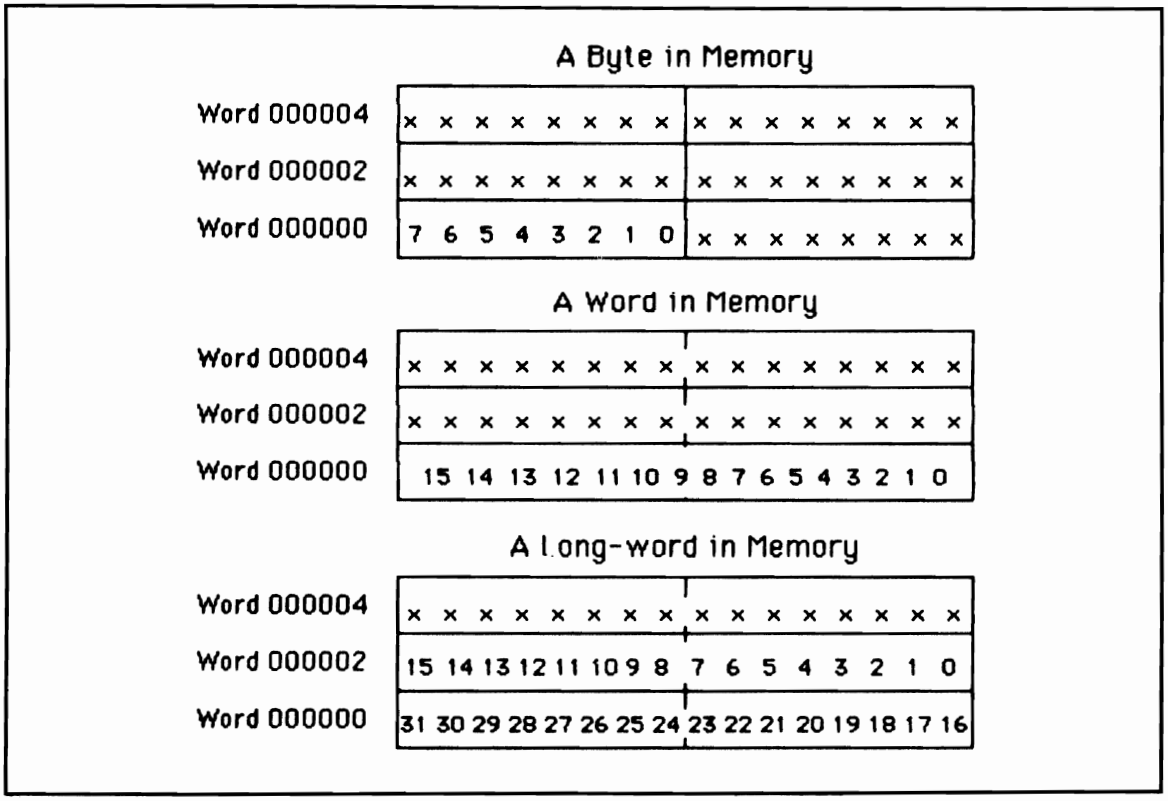


Fig. 4-4. Memory organization (bytes, words, and long-words).

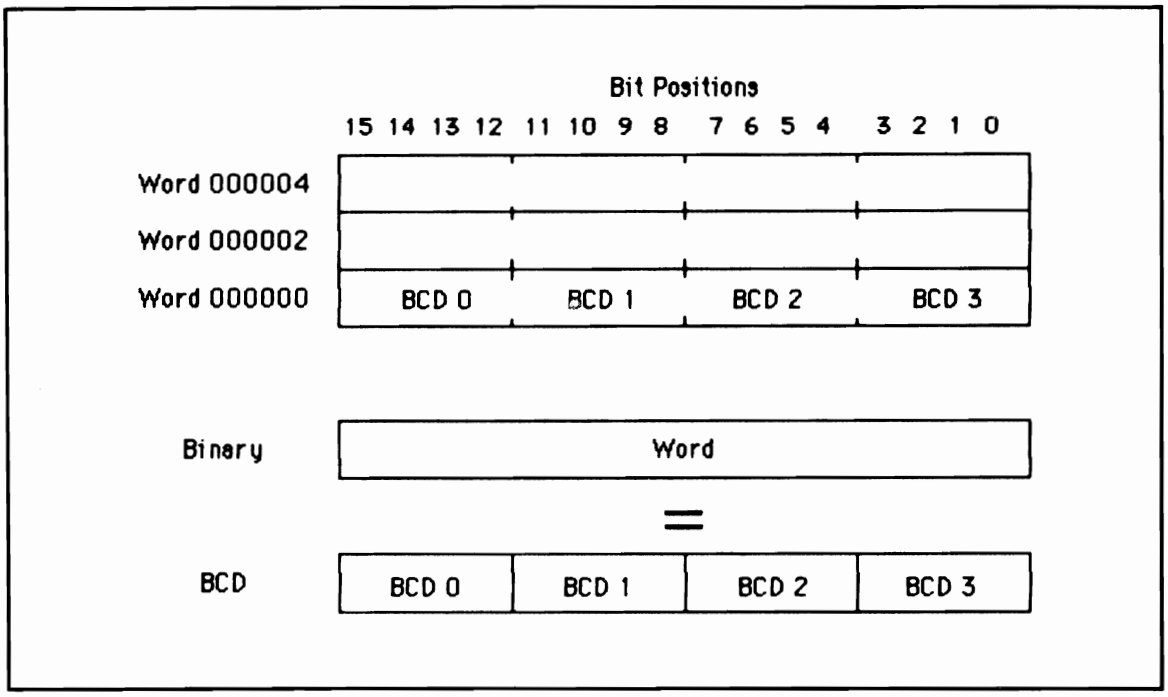


Fig. 4-5. Memory organization (nibbles: BCD).

BCD (Binary Coded Decimal) data is stored as shown in Fig. 4-5. The nibbles (4-bit chunks) are addressed in the opposite order of the bit positions.

ADDRESSING MODES

Addressing modes are the ways a computer uses to determine data locations. The 68000 has three major types of modes: register specification, effective address, and implicit reference. Register specification uses a part of the object code to tell which register to use. Effective address uses any of the addressing modes listed in Fig. 4-6 except Implicit. Implicit reference is the name given to the instructions that imply a particular use of registers.

As a momentary aside, you should be aware of the mess computer designers have made of naming addressing modes. Don't just look at the manufacturer's name for a mode; look at how it works. In a number of cases, the same name on two microprocessors refers to quite different machinations. For example, in the 8-bit microprocessor world, the 6502 has an indexed addressing mode that is not at all the same as the indexed addressing on the 6800.

Figure 4-6 lists the 68000 addressing modes. Using these, the 68000 assembly language programmer can expand the fundamental instruction set of the 68000 many times. But don't let the number of modes scare you. As with instructions, you don't have to use them all. Knowledge of just a few will permit you to write programs.

Register Specification

Instructions that work with *register specification* use a field within the instruction to signify whether the register used is a data register or an address register. Another field contains the number of the register to use.

Effective Address

Motorola uses the term *effective address* to refer to the address determined by two fields within of the 68000's addressing modes: a register number field and a mode field. These are typically coded

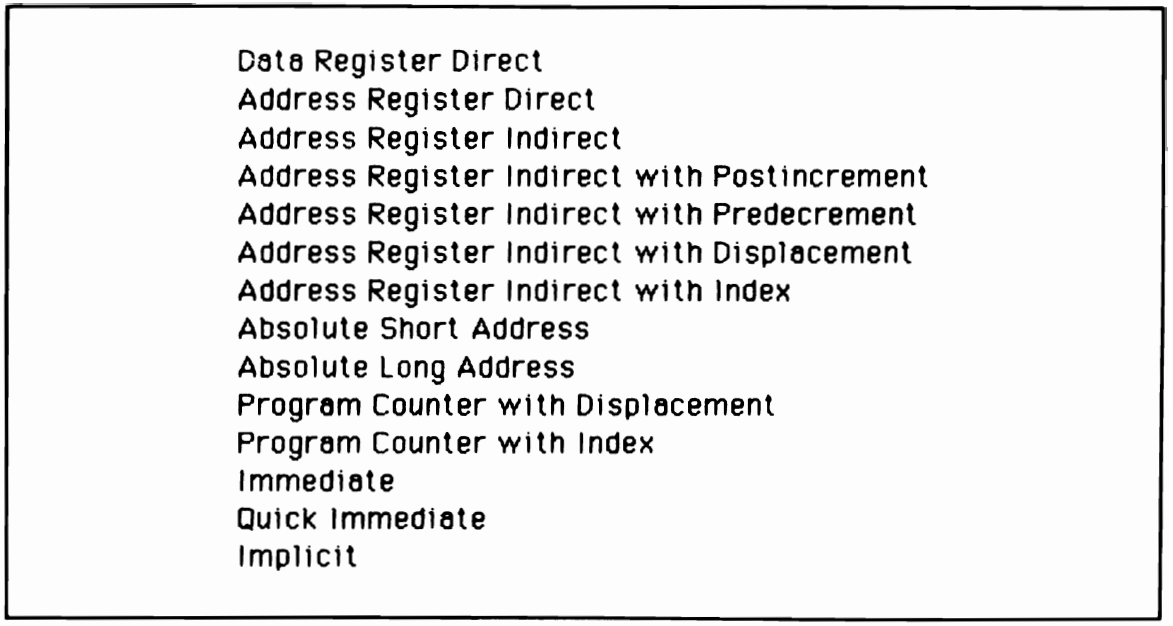


Fig. 4-6. 68000 addressing modes.

as the least significant six bits of the object code as shown in Fig. 4-7. Three bits are used to represent the mode and three more to represent the register number that mode is to use.

If two effective addresses are needed, two sets of six bits may appear in the object code. This may occur because a source is addressed in one way and a destination in another. Further information can be contained in extension instruction words.

Syntax

The *syntax* of an addressing mode is the way that the symbols are written in assembly language to represent the addressing mode. Each of the modes below will show both the proper syntax and an example of the mode's use.

There are several syntax rules that you should be aware of before you read through this section.

1. Instructions are written with the op code first, then the operands. For example,

ADD D1,D2

adds the contents of data register 1 to the contents of data register 2. D1 is the source and D2 is the destination.

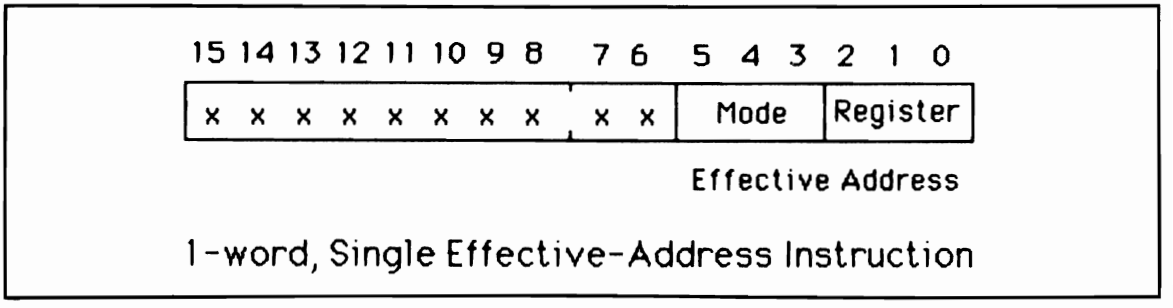


Fig. 4-7. Effective address coding.

ADD D2,D1

would add the same two values but the result would end up in data register 1 instead of data register 2. When an instruction needs only a single operand, that operand follows the op code and there is no comma in the instruction.

2. The operand size is specified by an extension letter to the op code. For example,

ADD.B D1,D2

performs the same operation as in the example above except that only the low-order byte of each register is used. The three extensions are as follows:

.B for bytes
.W for words
.L for long-words

If no specification is made and the instruction may work with more than one operand size, the default size (word) is used.

3. Most assemblers use the $ symbol to mean *hexadecimal.* A number without this symbol will normally be interpreted as a decimal value. I say normally because there are two other systems used—octal and binary—which also have symbols (B or % for binary; Q or C for octal). Those systems, however, are rarely seen and are explained in Chapter 9. Different assemblers may use different symbols: read your assembler instruction manual before jumping into programming.

4. Parentheses are used to indicate indirection. The symbol A4 means the instruction will work with the value in address register 4 as an address. The value found at that address in memory is the value the instruction will work with. This indirection is a two stage process. If you think that's complicated, you should know that many microprocessors have what is termed *true indirection* where the instruction finds an address, takes the contents of that address in memory and interprets it as a new address, takes the contents of that new address and uses it as the operand.

5. As mentioned before, many instructions need two operands. Each operand may have its own addressing mode, but not every mode will work for every instruction. See the individual instruction descriptions in Chapter 6 for a listing of the available modes.

Register Direct Modes

This mode keeps the data itself in a data register or address register. The register number is specified in the instruction.

Data Register Direct. This mode simply puts the name of a data register into the instruction's operand field. For example;

NEGX D6

negates (with the extend flag) the value in data register 6. D6 is found by Data Register Direct addressing.

Address Register Direct. This mode puts the name of an address register into the instruction's operand field. For example;

EXG A3,A2

exchanges the values inside address registers 2 and 3. What was in A3 is now in A2 and what was in A2 is now in A3. Both A2 and A3 in this instruction are examples of Address Register Direct addressing mode.

Memory Address Modes

These modes specify a location in memory. They all use indirection. The first value found is the address of the value the program will work with. The examples below will show this process clear.

The advantage of using indirect addressing is that you don't have to repeat a full address over and over. Instead of having that address appear in the code, you only need to deal with a register. The value in that register can be quickly and efficiently changed, and the object code needs only a few bits to specify a register.

Address Register Indirect. This mode doesn't directly point to the desired data as Address Register Direct does. Instead, it specifies an address register that holds the address of the data. The data

is in memory. Address Register Direct specifies a register that holds the data itself. For example;

CPM (A5),D3

compares the A source value to a destination value. The destination can only be found using Data Register Direct mode, as D3 does. The source can be found using any address mode.

In this case, the parentheses around A5 signifies indirection. It is not the value in A5 that will be compared. Rather, the value in A5 will be interpreted to be an address in memory. The value at that address will be compared to the value of data register 3.

Address Register Indirect with Postincrement. This mode is a lot simpler than it sounds. Together with the next mode, Address Register Indirect with Predecrement, this mode allows 68000 programmers to implement data structures such as stacks and queues using the general-purpose address registers.

The first part of Address Register Indirect with Postincrement mode works just as Address Register Indirect mode does. An address register is specified and the value within that register is interpreted as the address in memory of the operand. Then, however, the address register is incremented (by 1 if the instruction specifies byte, by 2 if the instruction specifies word, and by 4 if the instruction specifies long-word). For example;

TST.W (A1)+

will use the contents of address register 1 as an address in memory. The value of the word at that address will be tested and the flags will be set according to the results. The + symbol indicates that address register 1 will be incremented (by 2 because of the .W suffix) at the end of this instruction. The stack pointer will always be decremented by 2 (to keep it on a word boundary) if it is the specified address register.

Address Register Indirect with Predecrement. This mode, too, is simpler than it sounds. This instruction begins by decrementing (by 1 if the instruction specifies byte, by 2 if the instruction specifies word, and by 4 if the instruction specifies long-word) the value of the specified address register. From that point on, Address Register Indirect with Predecrement mode works just as Address Register Indirect mode does. The (now decremented) value of a specified address register is interpreted as the address in memory of the operand. For example;

TST.W (A1)–

will decrement (by 2 because of the .W suffix) the contents of address register 1. The symbol – indicates that this is predecrement mode. The new value in address register 1 is interpreted as an address in memory. The value of the word at that address will be tested and the flags will be set according to the result. The stack pointer will always be decremented by 2 (to keep it on a word boundary) if it is the specified address register.

Postincrement mode and predecrement mode can be used to transform address registers into stack pointers. One mode is used to push data onto the stack and the other mode is used to pull data off of the stack. Similar word sizes must be used so that the stack doesn't get misaligned. Such a stack can be made to grow up or down in memory.

Queues (waiting lines with FIFO—First in First Out—activity) can be implemented using postincrement or predecrement modes. One address register is the Put pointer and another is the Get pointer (corresponding to opposite ends of the queue).

Address Register Indirect with Displacement. This mode is also called Register Indirect with Displacement but only address registers may be used. While this mode resembles the Address Register Indirect mode described above, it adds one more step to the final address calculation.

The address in the address register is added to a signed 16-bit displacement value (which is the second word—the first extension word—of the instruction). That sum is interpreted as the address in memory of the desired operand. For example;

ROR 11(A2)

will take the value of address register 2 and add 11 (decimal) to it. The 11 is the displacement value and is always shown outside the parentheses as in this example. That sum will be interpreted as a memory address. The contents of that address will be rotated to the right. The default operand size (word) will be used.

This addressing mode lets you create many different data structures. The science and use of data structures is beyond the coverage of this book, but I will briefly explain one possibility here and one after the description of the Register Indirect with Index and Displacement mode.

The displacement addressing mode can be used to find specific fields within a record. Data storage systems frequently divide large sets of information into files, records, and fields. A file contains a number of records. A record contains information about a particular subject and is made up of several fields. A field contains a finite number of characters.

For example, a company might have a file of all employees salaries. That file would have a separate record for each employee. Each record would have fields like the following:

Name
Social Security Number
Date of Hire
Current Salary
Number of exemptions

These fields would contain characters that gave information relating to the field.

To find several facts about a particular employee, the computer would first have to load the employee's record into memory. The address of the beginning of the record would be loaded into an address register. At this point, each fact about an employee could be found simply by varying a displacement value. The address register value points to the beginning of the record and the displacement values add to the address just enough to reach name, social security number, or whatever you want to know.

Address Register Indirect with Displacement and Index. This mode is also known simply as Register Indirect with Displacement and Index or Address Register Indirect with Index. It is a special offshoot of Address Register Indirect and resembles Address Register Indirect with Displacement but is more complicated than either of those modes.

The instruction that uses this mode needs at least one extension word. The low-order byte of that word will be an 8-bit displacement value. The high-order byte will specify a register that holds an index value. Both the displacement and the index values are added to the address register value and the final sum is interpreted as an address in memory. The value contained at that address is the operand for the instruction. For example;

EOR.B D2,11(A6,D1.B)

Now this is getting complicated. But if you unravel it step by step it isn't so bad. First, EOR.B is the op code. This exclusive-OR operation will be performed on the low-order byte (.B) of the source and destination operands. The source, D1, is found by Data Register Direct addressing mode. The destination, 11(A6,D1.B), is found by Address Register Indirect with Index and Displacement addressing mode. The 11 is the displacement (and will be found in the low-order byte of the second instruction word). Address register A6 is the indirection register. Data register D1 is the index register. The low-order byte (.B) of the value in that register is the indexing value. In this instruction, therefore, two different operand sizes are specified.

The actual destination operand value is found in three steps.

1. Adding the index value, the displacement value, and the address register value.
2. Using that sum as an address.
3. Using the contents of that address as the operand.

This addressing mode also lets you create and work with a variety of data structures. An example is given next.

The displacement and index mode can be used to find specific fields within a number of records. As explained earlier under the Address Register Indirect with Displacement mode, storage systems work with files, records, and fields. The same example of a company's salary files can be used again here.

Addressing with Displacements and Indexing helps you find out the same facts about several different employees. The computer first loads the salary file into memory. The address of the beginning of the file is loaded into an address register and each record has the same known length. Also, each field within a record has the same length. At this point, if you wanted to load into the microprocessor all of the employees' salaries (to find an average, for example) you would use the length of records as an index value and the length of the fields as a displacement. By increasing the index values by regular steps, you can walk through the file, hitting each record. Each time you land on a record, the displacement value would put you into the salary field.

Special Addressing Modes

These effective addressing modes use the EA field to specify a special mode instead of a particular register.

Immediate. After Implicit addressing, Immediate addressing is probably the simplest mode to understand. Immediate addressing uses data that is immediately available. Such data is contained within the instruction words themselves. The effective address is the value in the program counter after the operation code has been fetched.

There are typically three operand sizes available for immediate addressing: byte, word, and long-word. The first instruction word contains the code that specifies operand size and addressing mode. If the immediate data is supposed to be one byte long, the low byte of the second instruction word is the data. A word of immediate data is the entire second instruction word. A long-word is the second and third instruction words, with the second instruction word being the high word and the third instruction word the low word. If that is all too confusing, look at Fig. 4-8 to see how this works.

The second and third instruction words are sometimes called *extension words* which can confuse things even more because the second instruction word is the first extension word, and so on. For example;

```
DIVU 165,D1
```

divides the value within data register 1 by 165 (decimal) and puts the result into data register 1. The 165 value is an immediate value. (The use of D1 is an example of Data Register Direct mode addressing for the destination.)

Quick Immediate. The *quick* instructions—such as ADDQ and SUBQ—are actually immediate instructions that don't need even a single extension word. These instructions make room within the first word of op code for some immediate data. Because there isn't much space, the data is limited to the range from 1 to 8. These instructions can execute especially quickly because they are so short (and so the mode is termed Quick Immediate). MOVEQ is another such instruction, see Chapter 6 for a detailed description of its operation.

Absolute Addressing

These two modes are related to immediate addressing. The difference is that, instead of providing the operand within the instruction code, they provide the operand's address within the instruction code.

Absolute Short Addressing. The second word of the instruction (the first extension word) is a 16-bit address. That 16-bit value is sign-extended (the value of bit position 15 is copied to bit positions 16 through 31) to a full 32 bits. That long-word is the address in memory of the operand. Because of the extension of the sign-bit, this mode can work with addresses in the ranges $0000 through $7FFF (the lowest part of memory) and $FFFF8000 through $FFFFFF (the highest part of memory).

For example;

```
PEA 1000
```

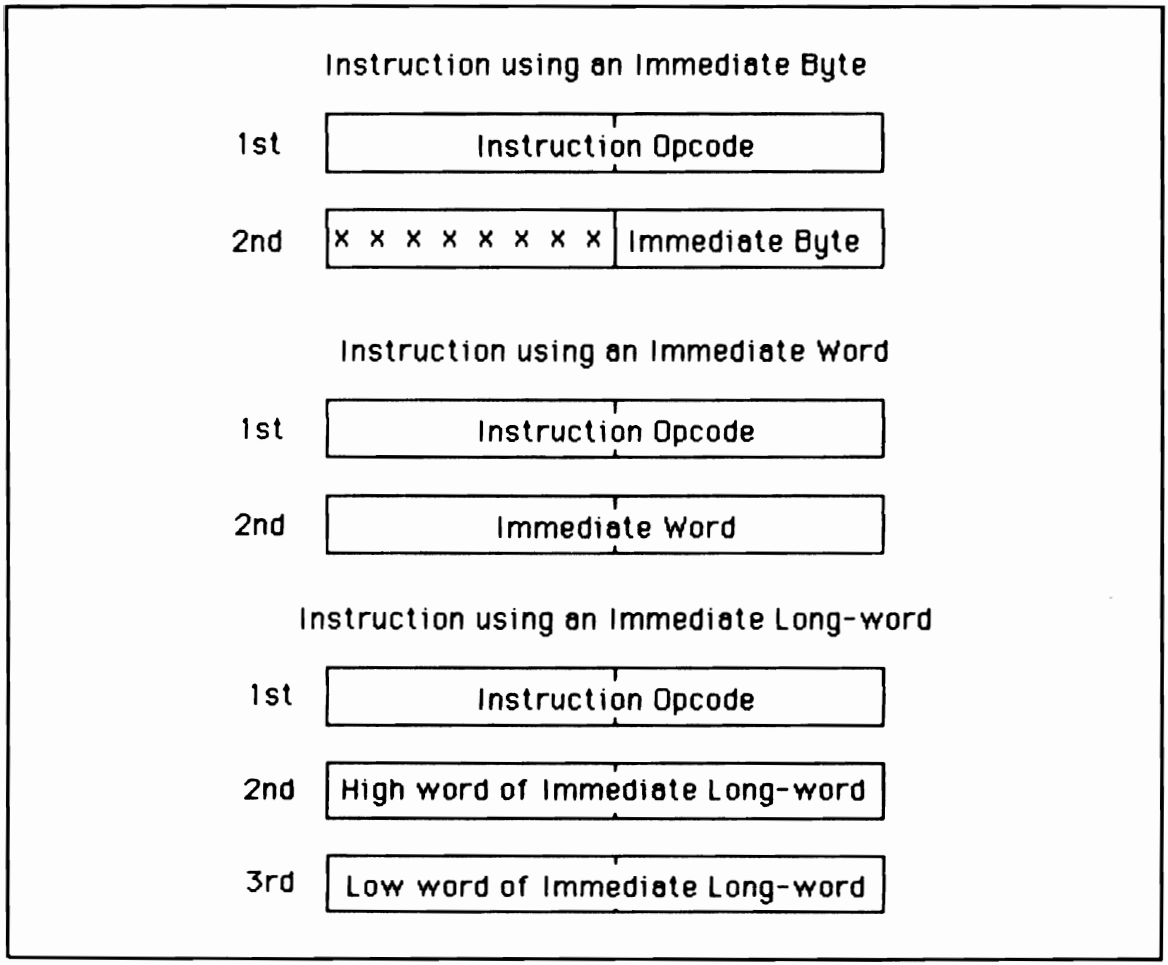


Fig. 4-8. Format of immediate data.

forms an effective address using the decimal value 1000. After sign-extension, that value is pushed onto the stack. If the number 1000 has a # symbol in front of it, it would be interpreted as immediate data. Without that symbol, it is absolute data. Because this instruction is only two words long and doesn't require calculation other than sign-extension, it can execute quickly.

Absolute Long Addressing. The second and third words of the instruction (the first and second extension words) are a 32 bit address. The first extension word is the high-order word of the address and the second extension word is the low-order word of the address. That long-word address is the location in memory of the operand.

For example:

SUBI.L 43,$A0000

subtracts the immediate data 43 (decimal) from the contents of the long-word starting at address A0000 (hexadecimal). If the $A0000 had a # symbol in front of it, it would be interpreted as immediate data. Without that symbol, it is absolute data.

Relative Addressing

This type of special addressing finds the operand by working with the current PC (Program

Counter) value. This is typically used for instructions that change the program's direction such as jumps and branches. By using relative values instead of absolute addresses, programmers can make their code more *relocatable.* In other words, the same program will work at any point in memory instead of just in the exact spot it was written for. It is normally better, though, to let the assembler calculate the distance for jumps and branches by giving it symbolic names instead of numbers.

Program Counter Relative with Displacement. This addressing mode works in much the same way that Address Register Indirect with Displacement mode works. In fact, this mode can be seen as a special case of Register Indirect mode. The 16-bit displacement value (found in the instruction extension word) is sign-extended and the PC value is incremented (as it is by any instruction). Then the two values are added together.

For example;

```
DIVS.W 250(PC),D1
```

adds the decimal value 250 to the incremented PC value. (Actually, most assemblers will add 248 to the PC value because they assume that the programmer wanted to use the point 250 bytes beyond the beginning of the DIVS instruction. Assemblers often let you use the symbol * to mean the present PC value.) That sum will be interpreted as an address. The low word of data register 1 will be divided by the contents of that address.

Program Counter Relative with Index and Displacement. For an introduction to this mode, you should read the description of Address Register Indirect with Index and Displacement because the two modes are quite similar. The PC instead of an address register is used as the foundation register.

The format for this instruction is tricky; check your assembler's instructions to see how to write it. For example;

```
CHK $AF(PC,A1),D4
```

checks the contents of the low-order word of D4 against the source value. The source value is calculated by adding both the hexadecimal value AF (the displacement) and the contents of register A1 (the index) to the program counter value.

Instruction	Implied Registers
Bcc, BRA	PC
BSR	PC, SP
CHK	SSP, SR
DBcc	PC
DIVS	SSP, SR
DIVU	SSP, SR
JMP	PC
JSR	PC,SP
LINK	PC,SP
MOVE to CCR, MOVE from CCR	SR
MOVE to SR, MOVE from SR	SR
MOVE USP	USP
PEA	SP
RTE	PC, SP, SR
RTR	PC. SP. SR
RTS	PC, SP
TRAP	SSP, SR
TRAPV	SSP, SR
UNLK	SP
ANDI to CCR, EORI to CCR, ORI to CCR, ANDI to SR, EORI to SR, ORI to SR	SR

Fig. 4-9. Instructions that depend entirely upon Implicit Addressing.

Implicit Reference Addressing

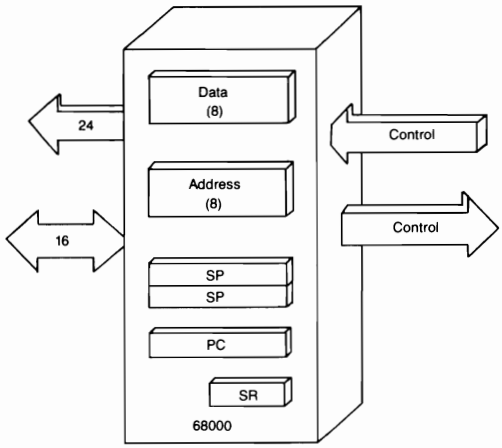
Some instructions do not offer you any addressing choice. These instructions, listed in Fig. 4-9, by their very nature specify exactly what addresses they will work with. Implicit addressing, called *implied* addressing on some other chips, is often

associated with fast instructions because they can be shorter and do not have to calculate an effective address.

IMPORTANCE OF ADDRESSING MODES

Let me add a postscript about addressing modes: Don't ignore them. Beginning programmers are often thrilled to learn instructions and only learn the bare minimum about addressing. Such behavior is the same as high-level language programmers learning only instructions and avoiding data structures. Both addressing modes and data structures are vital to efficient, maintainable (meaning someone can fix it) programs. If you play with addressing modes, learning when to use which, you will soon write more efficient and more structured programs than those who stick to immediate and implict addressing.

5



Instructions Groups

A MICROPROCESSOR WITHOUT A PROGRAM IS just a lifeless lump of silicon, metal, and plastic. To write programs in machine or assembly language you have to understand the registers, addressing modes, and the instruction set of the microprocessor.

The fundamental operations that a microprocessor can perform are represented by instructions. They are the words in the microprocessor language. The full instruction set of the 68000 is shown in Fig. 5-1. Learn a few and you can make a little sense. Combine the few with a knowledge of the punctuation of flags and the grammar of addressing modes and you can write simple programs. If you learn most of the instructions available on a microprocessor, you will have a full vocabulary. (Not even professional programmers use all of the instructions. Some are just not that practical.) True fluency requires the vocabulary plus understanding of data structures and control sequences along with lots of practice.

This book will provide you with the first ingredients: the vocabulary of instructions, the grammar of flags, and addressing modes. For an understanding of programming concepts, you should refer to a general book on computer science or programming. For practice, find a way to use a 68000 based system with an assembler program, and go to it. Eventually you'll want a library of subroutines to study, imitate, and use, but at first all you'll need is this book, the system, time, and some patience.

By the way, because most microprocessors are designed to perform similar tasks, they have similar instruction sets. While the actual codes used in assembly language to represent the instructions differ from chip to chip, the functions of the instructions are in many cases identical. In other words, once you learn the language of one chip, learning the next chip language (especially the fundamental instructions) will be quick and easy.

Advanced instructions do tend to diverge more from chip to chip. Different designers have a variety of ideas on what functions users would like to see.

The number of instructions on a microprocessor isn't necessarily related to its power. A small number of instructions with a variety of flexible ad-

ABCD	CLR	LSR	ORI to SR	TRAP
ADD	CMP	MOVE	PEA	TRAPV
ADDA	CMPA	MOVE to CCR	RESET	TST
ADDE	CMPI	MOVE to SR	ROL	UNLK
ADDQ	CMPM	MOVE from SR	ROR	
ADDX	DBcc	MOVE USP	ROXL	
AND	DIVS	MOVEA	ROXR	
ANDI	DIVU	MOVEM	RTE	
ANDI to CCR	EOR	MOVEP	RTR	
ANDI to SR	EORI	MOVEQ	RTS	
ASL	EORI to CCR	MULS	SBCD	
ASR	EORI to SR	MULU	Scc	
Bcc	EXG	NBCD	STOP	
BCHG	EXT	NEG	SUB	
BCLR	ILLEGAL	NEGX	SUBA	
BRA	JMP	NOP	SUBI	
BSET	JSR	NOT	SUBQ	
BSR	LEA	OR	SUBX	
BTST	LINK	ORI	SWAP	
CHK	LSL	ORI to CCR	TAS	

Fig. 5-1. 68000 instruction set.

dressing modes can often accomplish more than a large but rigid set of instructions with few addressing modes. The 68000 depends on flexibility and addressing and has an *orthogonal* instruction set. That means that each instruction is made to work much as the other instructions work with different data sizes (byte, word, or long-word) and with as many addressing modes as possible (see Chapter 4 for a description of these).

YOU DON'T HAVE TO LEARN THEM ALL

Not all of the microprocessor's instructions are equally important. In fact, you can write virtually any program using just a small portion of the instruction set. Certainly beginners should learn the simple instructions and use them with the various addressing modes before worrying about learning the more advanced and unusual instructions.

The particular selection of 68000 instructions shown in Fig. 5-2 and Fig. 5-3 are not set in concrete: use them as suggestions. The three and four letter codes are called the *mnemonics* for the instructions. As with most computer languages, the programmers don't want to have to write out the entire instruction each time they use it. Using "BSR" instead of "Branch to Subroutine" saves typing time, printing space, and memory space. It won't take you long to learn what the mnemonics stand for. Some, like STOP, are even obvious.

The 68010 and 68020 chips brought some new instructions to the 68000 family, and changed the performance of some old instructions. These instructions are listed in Fig. 5-4 and Fig 5-5 and are detailed in Chapter 9.

ADD	DIVU	MULU
AND	EOR	NEG
ASL	EXG	NOP
ASR	EXT	NOT
Bcc	JMP	OR
BRA	JSR	ROL
BSR	LSL	ROR
CLR	LSR	RTS
CMP	MOVE	STOP
DIVS	MULS	SUB

Fig. 5-2. Beginning instructions.

INSTRUCTION GROUPS

The fundamental instructions of any microprocessor can be divided into functional groups. Those groups are quite similar from chip to chip. The 68000 chip family groups defined in this book shown in Fig. 5-6.

While all of the instructions are described individually in Chapter 6, this chapter will discuss the uses of the various groups of instructions. Instructions can't be easily separated from the addressing methods they use. This chapter will, however, try to concentrate on the essence of each instruction.

ABCD	NBCD	TRAPV
BCHG	PEA	TST
BCLR	RESET	UNLK
BSET	RTE	
BTST	RTR	
CHK	SBCD	
DBcc	Scc	
ILLEGAL	SWAP	
LEA	TAS	
LINK	TRAP	

Fig. 5-3. Advanced instructions.

New Instructions	Modified Instructions
MOVE from CCR	MOVE from SR
MOVEC	RTE
MOVES	
RTD	

Fig. 5-4. New or changed 68010 instructions.

Data Movement

This group, shown in Fig. 5-7, is often described first because it includes some of the very first instructions any programmer uses. Many people think of computers as mathematics machines, when in fact their major use is as symbol storage and manipulation machines. Any program has to move the various bits and bytes around from input to output (at the very least), to memory and back, and within the CPU.

On the 68000, the bulk of this group is made up of a single instruction.

MOVE

When combined with the addressing modes and the byte, word, and long-word options, this single

CMP2	BFTST	cpDBcc
DIVSL	PACK	cpGEN
DIVUL	UNPK	cpRESTORE
EXTB	CALLM	cpSAVE
BFCHG	RTM	cpScc
BFCLR	BKPT	cpTRAPcc
BFEXTS	CHK2	
BFEXTU	TRAPcc	
BFFFO	CAS	
BFINS	CAS2	
BFSET	cpBcc	

Fig. 5-5. New 68020 instructions.

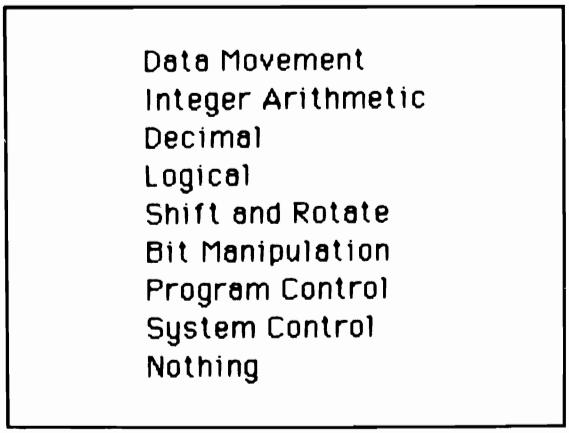


Fig. 5-6. 68000 instruction groups.

instruction is capable of moving almost any piece of information anywhere the microprocessor bus lines go. It can move information from register to register, register to memory, or memory to memory. In contrast to many other microprocessors, the 68000's limited number of instructions and their orthogonality (explained above) makes it easy for you to memorize the instructions.

MOVE and most of the other instructions in this group don't really move the information. They make a copy of the information and put that copy in the destination. The source retains its data. Some instructions, though, do alter the source contents. EXG, for example, moves the destination contents to the source and so completely erases the original source value.

While other microprocessors often have I/O (Input/Output) instructions that differ from the internal data movement instructions, the 68000 uses memory-mapped I/O. That means that all you have to do with the 68000 is wire up any I/O device to a memory address and use the same MOVE instruction you would use if the CPU was working with a memory chip instead of an I/O device.

MOVE can move either addresses or data. Data moves can work with byte, word, or long-word information. Address moves work with word or long-word operands.

There are also instructions that are special cases of the MOVE instruction. These lead the list in Fig. 5-7.

MOVE to CCR (condition codes register), *MOVE to SR* (status register), and *MOVE from SR* are actually just different ways of addressing the MOVE instruction. They are separated here because they deal with the status register. Any such dealings can completely change the 68000's status,and possibly disrupt a program. In fact, MOVE to SR and MOVE from SR are specifically protected as privileged instructions. Because they directly affect the S flag, which determines whether the 68000 is in supervisor or user state, both of these instructions have to be shielded from user programs. Figure 5-8 shows the structure of the status register for both the System byte and User byte also known as the condition codes register). See Chapters 3 for a more extensive explanation of what Supervisor and User mode mean.

MOVE USP (User stack pointer) is another privileged instruction. There is no reason to use this instruction when the CPU is in User mode; the User stack pointer is then simply addressed as address register A7. When the S flag is set and the CPU is in the Supervisor mode, however, there is no other way to reach the User stack pointer. In Supervisor mode, address register A7 is the System stack pointer. Figure 5-9 shows the arrangement of two stack pointers.

MOVEA, which stands for MOVE Address, is

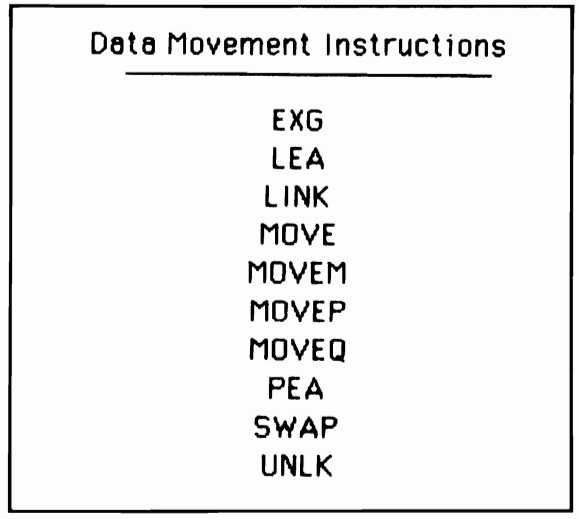


Fig. 5-7. Data movement instructions.

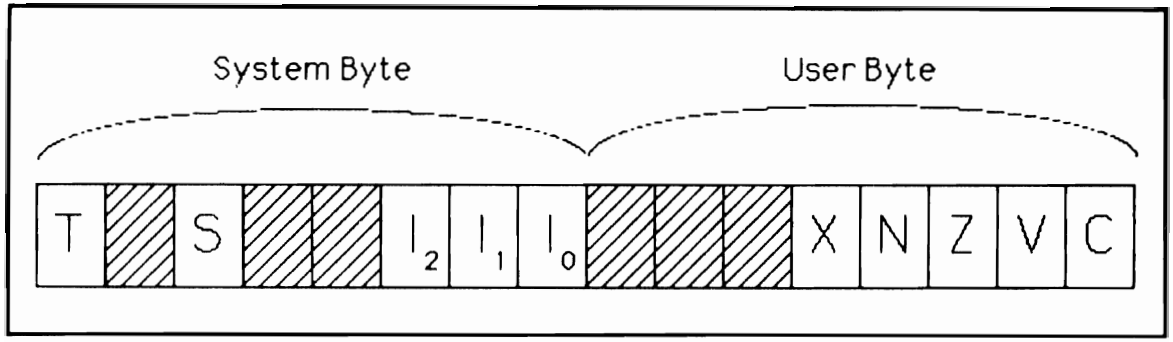


Fig. 5-8. Status register.

simply the instruction used if the MOVE instruction specifies the destination directly as an address register. Besides that addressing change, the only other difference is that MOVEA doesn't affect any of the flags, while MOVE does. Thus, the programmer can use MOVEA to set up for an operation by moving the necessary address information into registers, without upsetting the state of the flags.

MOVEM, which stands for MOVE Multiple registers, is a complex instruction that beginners will rarely use. It moves the data in a group of registers to or from memory. This saves programming work, and is very helpful to high-level languages that have to save and restore the information in the CPU. For example, if a program requires that processing shift to another point for a while, and then come back, the program should save the information in the data registers, address registers, and flags. MOVEM does this with a single instruction, putting the information into a sequence of memory addresses. Figure 5-10 portrays the pattern MOVEM uses to relate register data to memory data.

MOVEP, or MOVE Peripheral data, is a special MOVE command designed to make it easy for the CPU to send information to or receive information from peripheral devices. Such chips frequently read or write data in 8-bit chunks (bytes): MOVEP does also. With the automatic incrementing of this instruction (similar to that of MOVEM), the data coming from or going to the several byte-wide addresses of a peripheral can be handled with a single instruc-

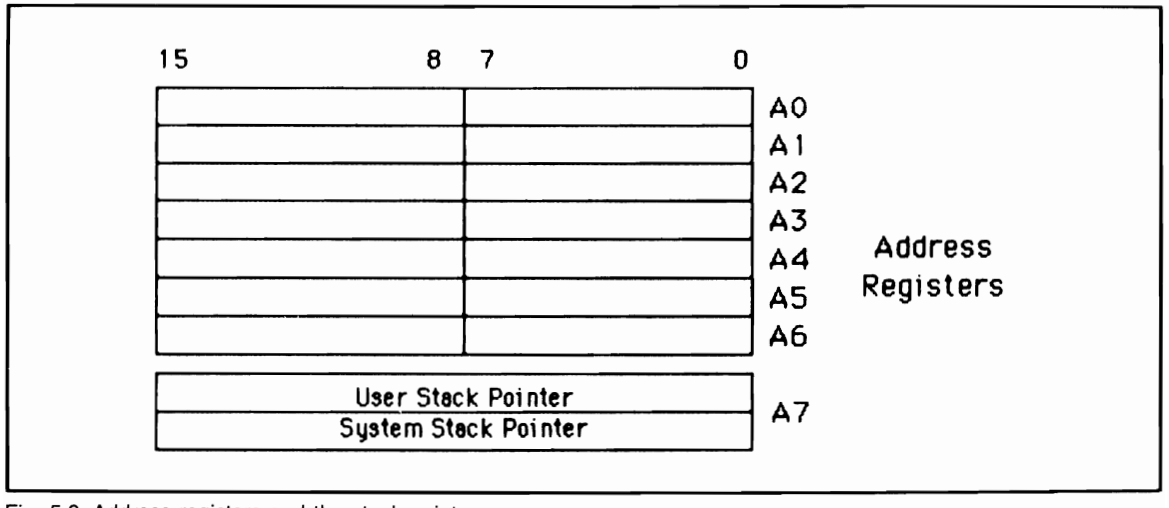


Fig. 5-9. Address registers and the stack pointers.

Effective Address in the Control Mode

(Data can move either direction)

Memory: Start at the specified address and climb through higher addresses

Registers: Start with the least significant bit of the mask (D0) and climb through more significant bits of the mask (D1 through D7 and A0 through A7). Only move the registers whose mask bits are set.

MASK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Effective Address in the Postincrement Mode

(Data can only move from memory to registers)

Memory: Start at the specified address and climb through higher addresses

Registers: Start with the least significant bit of the mask (D0) and climb through more significant bits of the mask (D1 through D7 and A0 through A7). Only move the registers whose mask bits are set.

MASK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Effective Address in the Predecrement Mode

(Data can only move from registers to memory)

Memory: Start at the specified address and load the register values into progressively lower addresses.

Registrs: Start with the register represented by the least significant bit of the mask (A7) and proceed through the more significant bits of the mask (A6 through A0 and then D7 through D0). Only transfers registers whose mask bit is set.

MASK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

Fig. 5-10. MOVEM addressing.

tion. With MOVEP's organization of data, the bytes from a peripheral can be automatically sandwiched into the long-word size of 68000 registers. Figure 5-11 shows the way MOVEP organizes data.

MOVEQ is dedicated to fast execution (MOVE Quick). Several other instructions also have this sort of variant. While MOVEQ is fast, it is limited. It can only move a byte of immediate data (data that is contained within the instruction word) to a data register. That byte is sign-extended to a full 32 bits. Because the addressing is implicit and the instruction is only a single word long, MOVEQ can save time over MOVE. If used within a heavily-worked part of the program, such as a counting loop, the incremental time saved can add up to quite a bit.

EXG performs the work of three MOVE instructions: it exchanges the contents of two registers. If you had to program that with MOVE,

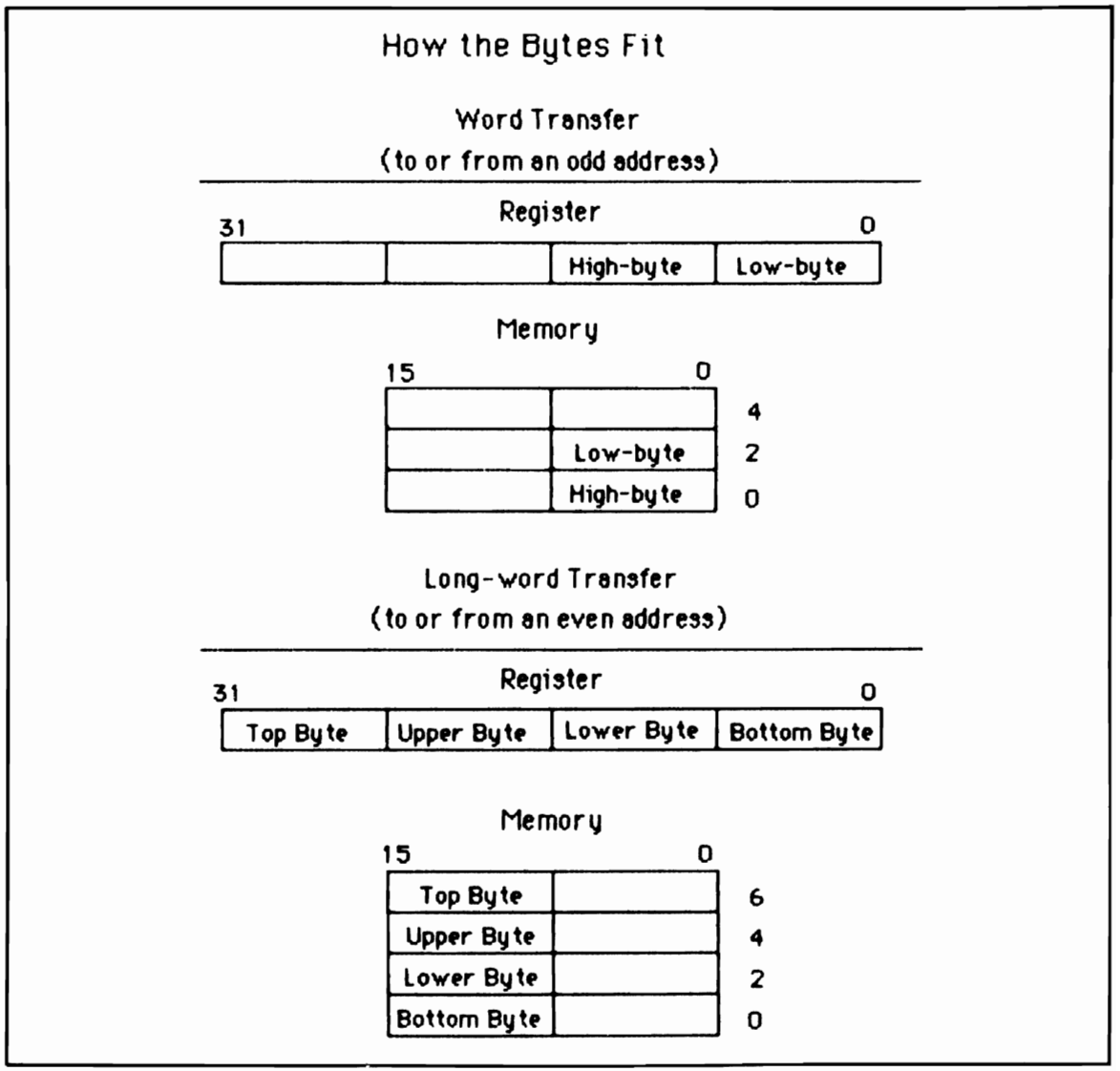


Fig. 5-11. MOVEP addressing.

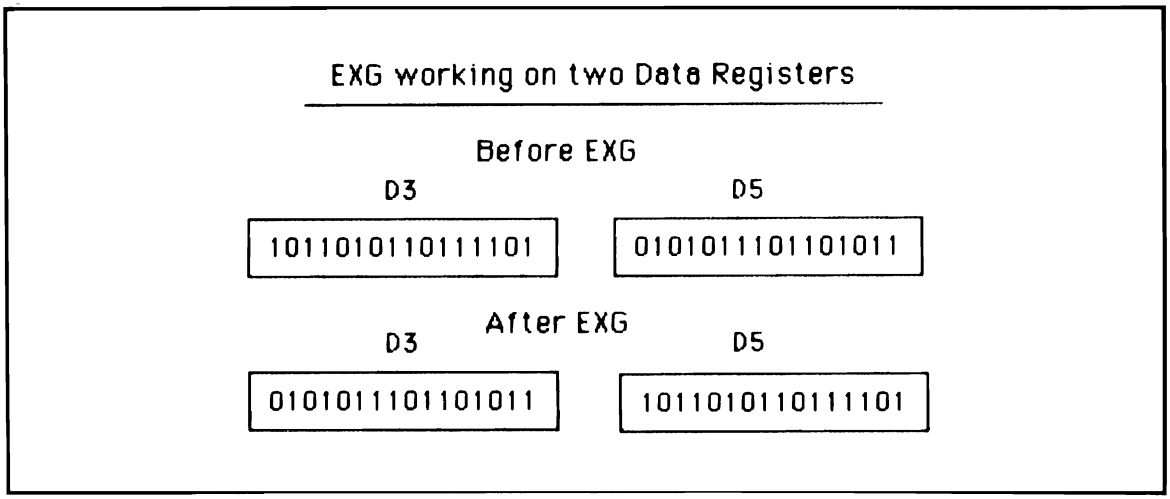


Fig. 5-12. EXG in action.

you'd need to move the contents of the first register into a third, then move the contents of the second register into the first, and finally move the contents of the third register into the second register. Figure 5-12 shows the action of EXG.

SWAP moves words, exchanging the low and high words of a data register. Figure 5-13 shows how SWAP switches the contents of the two halves of a single data registers.

LEA means Load Effective Address. *Load* is the term used on many microprocessors for data movement instructions. The Effective Address is the address information calculated from addressing mode and specified data. LEA does the calculating and then puts the EA in an address register. This can be useful for setting up to access a table in memory.

PEA pushes an Effective Address (explained in the previous paragraph) onto the stack. The stack was reached by address register 7, the active stack pointer: User stack pointer if the 68000 is in User mode; System stack pointer if the 68000 is in Supervisor mode. *Push* and *Pop* are verbs that mean, respectively, to put onto or take off of a stack (stacks are explained in more detail in Chapter 3). A stack is a LIFO (Last-in, First-out) data structure that is implemented in memory by the CPU control of a 16-bit stack pointer register.

The 68000 stack grows down in memory, so the stack pointer contains the address of the lowest address of the stack, which, unfortunately, is known as the top of the stack. PEA decrements the stack pointer by two (there are two bytes in a word) and puts the low word of the EA on the stack. Then the stack pointer is again decremented by two and the high word of the EA is put, or pushed, onto the stack.

LINK (Link and Allocate) and *UNLK* (Unlink) are advanced instructions that you won't find on simpler, 8-bit microprocessors. What LINK does, UNLK undoes. Together, they let you manipulate the stack to organize different stack memory areas for subroutines,functions, and program modules.

If a program section needs to use the stack, but you don't want to disturb the main program information on the stack, you can employ LINK to put

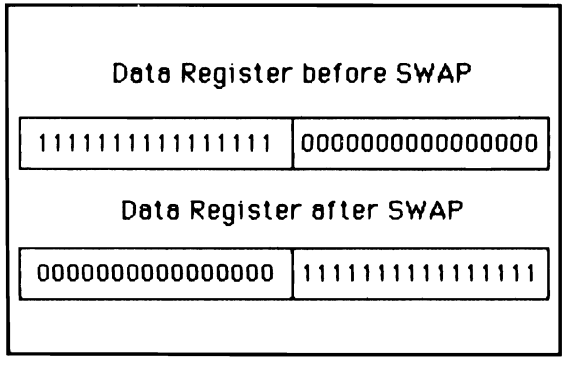


Fig. 5-13. SWAP in action.

a new value into the stack pointer. A section of stack is called a *frame*, and an address register that holds the value to be put onto the stack is called a *frame pointer*. The present contents of the frame pointer address register is put onto the stack. Then the new stack pointer value (the stack pointer was automatically decremented when a new value was put onto the stack) is saved in the frame pointer address register. A negative displacement value (negative because the stack grows downward) which is essentially the size of the frame, is then added to the stack pointer to open up as large a frame as the subroutine needs.

Integer Arithmetic

The Integer Arithmetic instructions are shown in Fig. 5-14. While the 68000 can perform the same add and subtract operations that all microprocessors must, it also has multiply and divide instructions. That will come as a relief to 8-bit programmers who had to use routines of simpler instructions to implement those operations.

ADD means to add using binary arithmetic. This differs from the other addition instruction, ABCD, which employs BCD (Binary Coded Decimal) arithmetic. (ABCD is explained later in this chapter in the Decimal Arithmetic Group section.)

ADD sums the source and destination operands.

Integer Arithmetic

ADD	MULU
ADDX	NEG
CLR	NEGX
CMP	SUB
DIVS	SUBX
DIVU	TAS
EXT	TST
MULS	

Fig. 5-14. Integer arithmetic instructions.

The operands may be reached by any of a large number of addressing modes. ADD demonstrates another common feature of 68000 instructions in its placement of the result. The result is stored in the destination, erasing the previous destination contents. ADD is also representative of 68000 instructions in that addressing is split into two cases. The first cases uses a data register as the destination and can use any addressing mode for the source. The second case uses a data register as the source and employs any of nine different addressing modes to reach the destination (Program Counter Relative and Immediate modes cannot be used).

Binary arithmetic is the foundation of all digital computing. If you don't know how to add 1s and 0s, you can still program computers, but not in assembly language. Any elementary programming or computer science book can explain the methods of adding, subtracting, and complementing the bits (binary digits) of bytes, words, and long-words.

A quick summary of binary arithmetic is shown in Fig. 5-15. The terms *unsigned* and *two's complement* should also be referred to another text. Simply put, unsigned binary numbers interpret the entire string of 1s and 0s as a single positive number. Two's complement binary numbers use the most significant bit as a sign bit. A 0 means the entire number is positive; a 1 means negative. To make things more complicated, a negative two's complement number is written in a special form with most of the bits inverted (0s become 1s and 1s become 0s). It sounds crazy, I know, but it does make arithmetic a lot easier, and you'll appreciate it once you learn the system and work through some problems.

ADD affects all of the condition code flags. C (carry) and X (extend) are set if the addition results in a carry. Z (zero) is set if the result is zero. N (negative) is set if the result is negative. Finally, V (overflow) is set if the addition causes an overflow. These results are all straightforward, and apply to most other arithmetic instructions, except that subtraction operations work with a borrow instead of a carry for the C flag.

ADDA (Add Address) is a special form of ADD. It adds a source operand, specified by any of the addressing modes, to an address register destina-

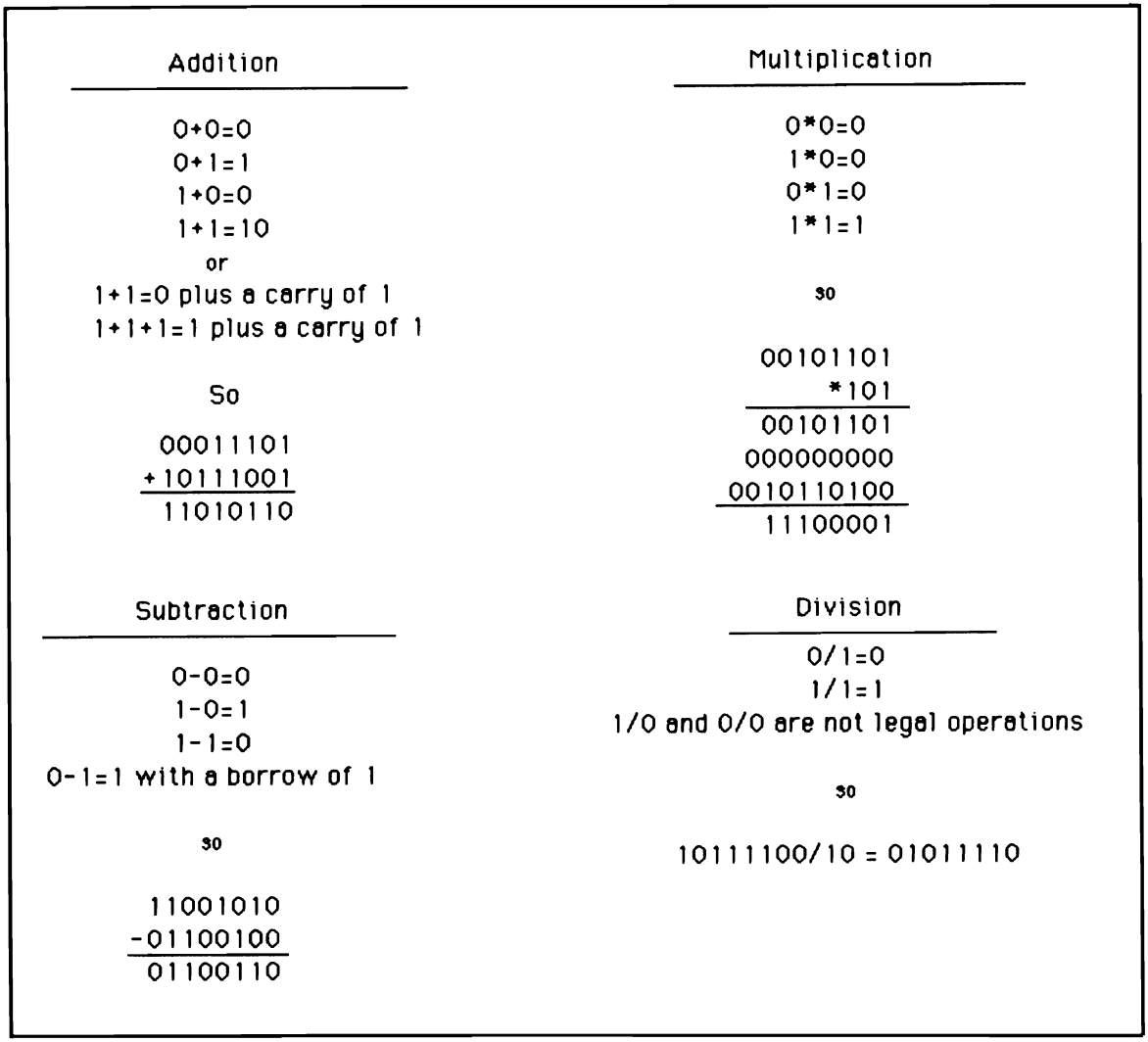


Fig. 5-15. Rules of binary arithmetic.

tion. What's the big difference between ADD and ADDA? ADDA doesn't affect any of the condition code flags. That means you can manipulate addresses and set up the address registers with values you need for the addressing modes of succeeding instructions without altering the status of the chip. Remember that address registers cannot work with byte-size data, only words and long-words. This is explained in Chapter 3.

ADDI means Add Immediate. This instruction adds the data contained within the instruction (a byte, a word, or a long-word) to a destination operand. It differs from the ADD instruction because when ADD has an immediate source, it can only use Data Register Direct mode to find the destination. ADDI can use any of eight different addressing modes.

ADDQ (Add Quick), like MOVEQ, is dedicated to quick execution. Its addressing range is narrower than ADD's; only Quick Immediate mode can be used for the source. Its immediate data size is more limited than that of ADDI; only a single byte can

be added. These two restrictions allow the entire instruction to fit into a single instruction word and to execute very quickly. Again, as with MOVEQ, using ADDQ in sensitive positions where the addition may be repeated many times can save a lot of processing time. ADDQ is a good demonstration of the orthogonality of the 68000 instruction set. Many instructions have the same special cases and forms: MOVEA and ADDA; MOVEQ and ADDQ.

ADDX, which stands for Add Extended, mainly differs from ADD in its effect on the Z flag. ADD sets (1) the Z flag if the sum equals zero and clears it (0) otherwise. ADDX leaves the Z flag unchanged if the result equals zero and clears it if the result is nonzero (anything but zero). This special effect adapts the ADDX instruction to multiple-precision (also known as extended) addition. Multiple-precision arithmetic works with numbers that are too large to fit into bytes, words, or longwords. Figure 5-16 shows a simple example of multiple-precision arithmetic.

Multiple-precision addition often begins by using the MOVE to CCR instruction to set the Z flag (putting a 1 in it). Then the addition is performed. Any nonzero result will clear the flag. Thus the program is alerted to the data that may affect other parts of the addition.

There are two forms of ADDX: register to register and memory to memory. Memory to memory uses predecrement mode so that the multiple operands for extended arithmetic may be easily addressed in order.

SUB is the binary subtraction instruction. It operates very much as the ADD instruction except that the source operand is subtracted from the destination operand and the C and X flags are set if the operation results in a borrow instead of a carry. SUBA is Subtract Address, SUBI is Subtract Immediate, SUBQ is Subtract Quick, and SUBX is Subtract Extended. See the analogous addition instruction descriptions above (ADDA, ADDI, ADDQ, ADDX) for the purposes of these instructions.

MULU and *MULS* are the multiplication instructions. One of the advantages of 16-bit chips such as the 68000 is that they have multiplication and division instructions: 8-bit chips have to perform those operations with routines of move, shift, and compare instructions.

MULU stands for Multiply Unsigned; MULS stands for Multiply Signed. As in our previous discussion of binary numbers,the most significant bit can be used to indicate the sign of the number or the number can be assumed to be positive. MULU multiplies two unsigned 16-bit operands to yield a 32-bit unsigned result. MULS multiplies two signed 16-bit operands to yield a 32-bit signed result. both operations take register operands from the low word of a register and ignore the high word.

DIVU and *DIVS* are the Division Unsigned and Division Signed instructions respectively. DIVU divides a 32-bit destination operand by a 16-bit source operand using unsigned binary numbers. The 32-bit result is stored in the destination (which must be a data register). The low word of the result is the quotient and the high word is the remainder. DIVS does the same thing as DIVU but uses

Multiple-precision Addition

Byte 8	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1
00110010	10101100	11110000	10111100	10110011	01010101	10111111	00000001
01000010	11101010	11010111	01011010	11111111	00001101	01011010	10111100
01110101	10010111	11001000	00010111	10110010	01100011	00011001	10111101

Fig. 5-16. Multiple-precision binary addition.

signed arithmetic. The sign of the remainder is always the same as that of the divided unless the remainder equals zero.

Division carries with it two special circumstances. Any attempt to divide by zero will automatically be trapped and exception processing will take over from the main program. See Chapter 7 for more details on exception processing and the divide by zero trap.

The second special circumstance involves overflow. If the division operation causes an overflow which is detected before the instruction is completely executed, the overflow condition will be signaled by the flags and the original division operands will be left as they were. The more advanced 68020 doesn't have to worry about this circumstance and is fully capable of dividing 32 bits by 32 bits as well as multiplying 32 bits by 32 bits.

Computers also need to compare, clear, and negate the values within registers or memory addresses. Because the 68000 can work with several different data sizes, it also needs a sign-extension instruction to allow byte and word data to operate correctly in 32-bit registers. The instructions shown in Fig. 5-14 handle these chores.

CLR (which stands for Clear) is probably the simplest to understand of this lot: it puts zero into the destination. The destination can be a data register or a memory location and is filled with as many zeros as the size specification requires (either byte, word, or long-word). CLR also effects most of the condition code flags; it sets Z and clears N, V, and C.

CMP, for Compare, offers a basic instruction and a number of special cases, just as ADD, SUB, and MOVE do. CMP compares a source and a destination operand (by subtracting and source contents from the destination contents), sets the condition code flags, and leaves the source and destination unchanged. Because the CMP operation is a subtraction, the C flag is treated as a borrow flag (see the SUB description earlier in this chapter for details of this treatment).

CMPA (Compare Address) is a special case of CMP that uses an address register as the destination. Also, CMPA extends the source value to 32-bits before subtracting it from the destination value.

CMPI (Compare Immediate) can only have immediate data as the source. Otherwise, it operates as does CMP.

CMPM (Compare Memory) is a complicated special case of CMP that compares the contents of two memory locations. Both source and destination are addressed by the specified address registers by the postincrement register indirect addressing mode. In other words, the instruction specifies two address registers. The contents of one of those registers specifies the source location. The contents of the other is the address of the destination. After the source contents are subtracted from the destination contents and the flags are set, the values of the two address registers are incremented (by 1 if a byte operation, by 2 if a word operation, by 4 if a long-word operation).

The use of an automatically incrementing addressing mode eases the design of loops. For instance, CMPM is used in loops that search for a particular value in memory. If the automatic incrementing weren't available, the programmer would have to write incrementing instructions into the loop so that a region of memory could be sequentially searched (compared against a known value).

TST and *TAS* are special comparison operations. TST subtracts zero from the destination contents and manipulates the condition code flags according to the results. In other words, the flags are changed according to the value in memory. If that value is negative, the N flag is set. If that value is zero, the Z flag is set. Flags V and C are always cleared. The X flag is not affected. The effects of TST on the flags is considerably different than the effects of CMP on those flags.

TAS is a more complex instruction that TST. TAS can only work with a single byte, though the operand containing that byte can be addressed by any of eight different modes. The byte is examined and the N and Z flags are set according to its value. Then the high-order bit of the byte is set (equal to 1). A vital aspect of this instruction is that it uses a read-modify-write memory cycle. This way of

reading memory makes TAS *indivisible*. In other words, TAS cannot be interrupted: no other device, CPU or peripheral, can get at that particular memory location while TAS is working on it. External interrupt or bus requests are ignored while TAS is executing. The point of this is to allow you to set flags in memory that multiple devices can use to communicate with each other.

If TAS could be interrupted, the following sequence of events could occur and disrupt processing. The first processor checks the flag and finds it cleared (meaning the other processor hadn't touched it yet). The second processor interrupts the first, checks the flag, and then sets it. After the interruption, the first processor goes back to what it is doing convinced the second processor hasn't checked the flag location. TAS, therefore, allows synchronization of independent processes.

NEG means Negate. The destination value (there is no source) is subtracted from zero using two's complement binary arithmetic. The result is stored in the destination, replacing the former contents. Because this is a subtraction operation, the flags are affected in the same way as with the SUB instruction. Although negation is a necessary tool in binary arithmetic, you will probably not use it until you have a firm grasp on binary algorithms. Figure 5-17 shows an example of negation.

NEGX means Negate Extended and is a special form of NEG that affects the X flag. The operand and the X flag value are subtracted from zero. NEGX is intended to simplify multiple-precision arithmetic work, as are ADDX and SUBX. Figure 5-16 expands on multiple-precision binary arithmetic.

EXT extends the sign bit of a binary number. Its action is shown in Fig. 5-18. A binary number can be interpreted as signed or unsigned. When interpreted as signed, the most significant bit is read as the sign of the number: a 0 means positive and a 1 means negative. The most-significant bit of a byte is in the Bit 7 position. For a word, it is in the bit 15 position, and for a long-word it is in the bit 31 position. Because the 68000 can work on bytes, words, or long-words, the sign of a value could be forgotten or mistakenly manipulated. For instance, a data register that holds a negative number will have a 1 in the bit 31 position. If you then moved a positive word into the data register, the 1 would still be in a bit position 31 and so the data register value would still be interpreted as negative even though the bit 15 position (the sign bit of the moved-in word) held a 0 (meaning positive). The solution is to extend the meaningful sign bit. In this case, EXT would be used to write the 0 sign bit in bit position 15 into all the bit positions up to and including 31. If a byte is used, EXT extends the bit in position 7 up to and including bit 15. Some other instructions automatically sign-extend values to keep from corrupting results.

Scc (for Set According to Condition) is put into the Program Control group of instructions by some books. Like some other Program Control instructions (Bcc and DBcc), Scc is a conditional instruction. It tests the flags for a programmer specified condition and only performs its set (1) task if the

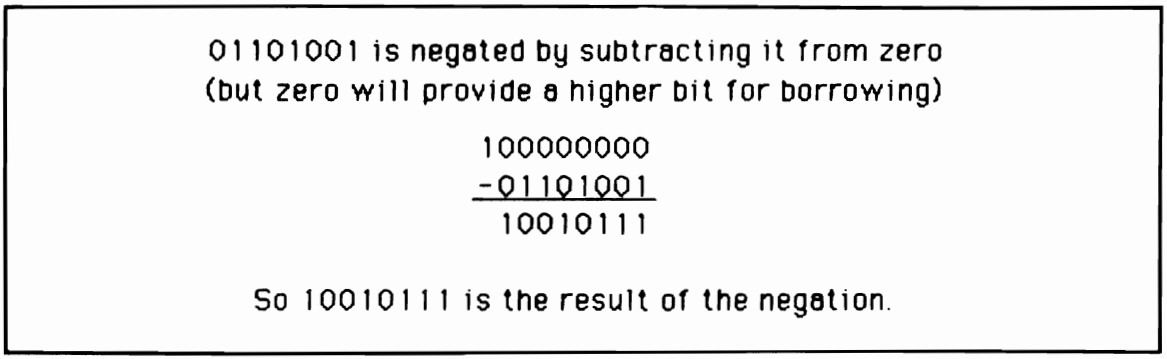


Fig. 5-17. Negation—NEG.

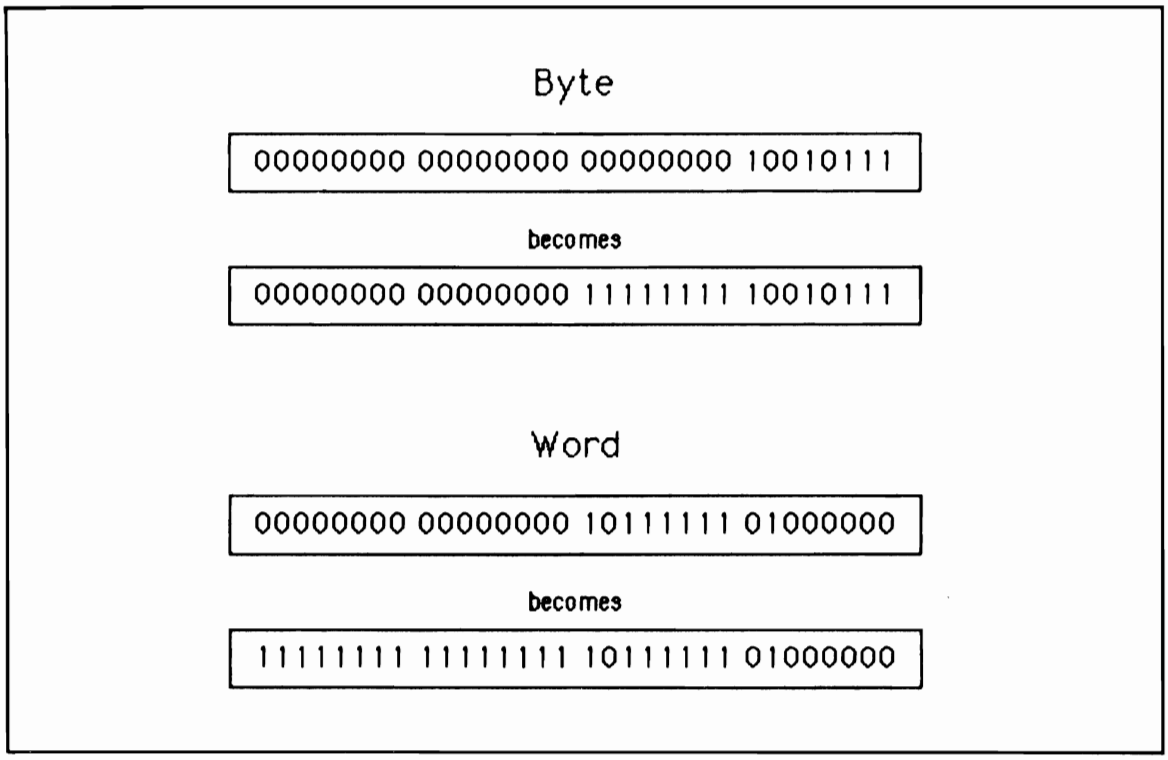


Fig. 5-18. Sign extension—EXT.

condition is met. The conditions the programmer can choose from are listed in Fig. 5-19. If the condition is true, the addressed byte is set to all 1s. If the condition is false, the addressed byte is cleared to all 0s.

Decimal

Although digital computers are built around binary 1s and 0s, not all arithmetic is done using the rules shown in Fig. 5-15. The binary digits can be interpreted by a variety of codes. One of the popular codes is called BCD (for Binary Coded Decimal). BCD, illustrated in Fig. 5-20, uses groups of four bits to represent the numbers from 0 through 9. This is a more direct translation of the decimal numbers people use when not thinking of computers. Because much of the data we put into and take out of computers is in decimal number form already, using BCD saves some translation time. However, BCD numbers cannot work with the same

CC	Carry Clear
CS	Carry Set
EQ	Equal
F	False
GE	Greater or Equal
GT	Greater Than
HI	High
LE	Less or Equal
LS	Low or Same
LT	Less Than
MI	Minus
NE	Not Equal
PL	Plus
T	True
VC	Overflow Clear
VS	Overflow Set

Fig. 5-19. Scc conditions available.

```
0000 = 0                    0001 0000 = 10
0001 = 1                    0001 0001 = 11
0010 = 2                    0001 0010 = 12
0011 = 3                    0001 0011 = 13
0100 = 4                    0001 0100 = 14
0101 = 5
0110 = 6                    0010 0111 = 27
0111 = 7
1000 = 8                    0100 1000 = 48
1001 = 9
1010 = illegal code         1001 1001 0111 = 997
1011 = illegal code
1100 = illegal code
1101 = illegal code
1110 = illegal code
1111 = illegal code
```

Fig. 5-20. BCD numbers.

Addition

1. Add bits within nibbles by binary rules.
2. If the sum within a nibble is more than 9, add 6 to the sum and move the appropriate carry values to the next higher nibble. (Adding 6 carries the value past the illegal codes).

Subtraction

1. Subtract bits within nibbles by binary rules.
2. If the result within a nibble requires a borrow, remember to subtract six to move beyond the illegal codes.

Fig. 5-21. BCD addition and subtraction rules.

arithmetic rules as standard binary numbers. The rules of BCD arithmetic ar shown in Fig. 5-21.

The 68000 has three instructions for calculating with BCD data. These are shown in Fig. 5-22.

ABCD means Add using BCD with extend. ABCD is quite similar to ADDX, which also adds two operands together and affects the flags (including X). It is called "with extend" because it does affect the X flag and is therefore useful for multiple-precision arithmetic. BCD multiple-precision work is done much the same as the binary multiple-precision examples shown in Fig. 5-16 (using BCD rules, however). The Z flag is normally set before the addition. ABCD will interpret the operands it works with as BCD data, even if they are not.

ABCD can add two operands that are in data registers or two operands that are in memory. The memory case is accomplished with the predecrement addressing mode.

SBCD subtracts BCD data and operates in the same way ABCD does, though of course subtraction replaces addition and the C flag is affected by borrows instead of carries. Decimal borrows are different from binary borrows. If you are going to use BCD arithmetic much, you should understand it well first, and be able to work some problems by hand.

NBCD is Negate BCD with extend. Again, use NEG and ABCD as comparisons for this. The operand is negated by subtracting it and the X flag value from zero. The outcome of this operation is the *ten's complement* if the X flag equals 0 and the *nine's complement* if the X flag equals 1. NBCD can only work with bytes.

Logical

All digital computers make heavy use of logical operations. The 68000 provides the logical operations shown in Fig. 5-23. If you do not know the AND, OR, EOR, and NOT functions, you'll need to learn them before doing any serious programming. Any introductory computer science text will introduce you to them. As a quick introduction or review, Fig. 5-24 through Fig. 5-27 show each of those functions applied to two bytes. Logical operations are used to clear, set, and test specified bit positions of a byte, word, or long-word.

Decimal
ABCD
NBCD
SBCD

Fig. 5-22. Decimal instructions.

Logical
AND
OR
EOR
NOT

Fig. 5-23. Logical instructions.

AND compares a source and a destination operand bit position by bit position. Whatever bit positions are set (equal to 1) in both operands are also set in the result. The other positions are cleared (equal to 0) in the result. This operation is shown in Fig. 5-24. The result is stored in the destination

AND

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

SO

```
  10110110
AND
  00001111
----------
  00000110
```

Fig. 5-24. AND operation.

and the condition code flags are set according to that result.

AND is typically used to *mask off* bits. By setting one of the operands with a desired sequence of 1s and 0s, you can clear any pattern of bit positions. Wherever a 0 appears in your constructed operand, that bit position in the result will be a 0 regardless of the value in that bit position of the other operand. Try this for yourself on paper and see.

ANDI merely uses an immediate source operand and performs the same operation as AND. ANDI had two special forms.

ANDI to CCR
ANDI to SR

They perform the same AND operation, but they use some part of the status register as the destination.

ANDI to CCR works with a single immediate byte and the low-byte of the status register: the condition codes register (CCR). This instruction helps you control the flag values.

ANDI to SR works with an immediate word and the full status register (SR). Because the System byte (the high byte) of the SR contains the flag that controls Supervisor and User mode, this instruction is privileged. The processor must be in Supervisor mode to execute it. If the CPU is only in User mode, ANDI to SR will not execute, and a Trap will be generated instead.

OR compares a source and a destination operand bit position by bit position. Whatever bit positions are set (equal to 1) in either operand are also set in the result. The other bit positions (those that have 0s in both operands) are cleared in the result. This operation is shown in Fig. 5-25. The result is stored in the destination and the condition code flags are set according to that result.

OR is typically used to *mask in* bits. By setting one of the operands with a desired sequence of 1s and 0s, you can set any pattern of bit positions. Wherever a 1 appears in your constructed operand, that bit position in the result will be a 1, no matter what value that bit position of the other operand has. Again, try this for yourself on paper and see. *Masking* is something you should develop a facility for. It is about the only part of assembly language where you'll actually care about individual bit values (except for flags contents). Masking is particularly important in I/O operations.

ORI uses an immediate source operand and performs the same operation as OR. ORI has two special forms also.

ORI to CCR
ORI to SR

They perform the same OR operation, but they use some part of the status register as the destination.

ORI to CCR works with a single immediate byte and the low-byte of the status register: the condition codes register (CCR). This instruction helps you control the flag values.

ORI to SR works with an immediate word and the full status register (SR). Because the System byte (the high byte) of the SR contains the flag that controls Supervisor and User mode, this instruction is privileged. The processor must be in Supervisor mode to execute it. If the CPU is only in User mode, ORI to SR will not execute, and a Trap will be generated instead.

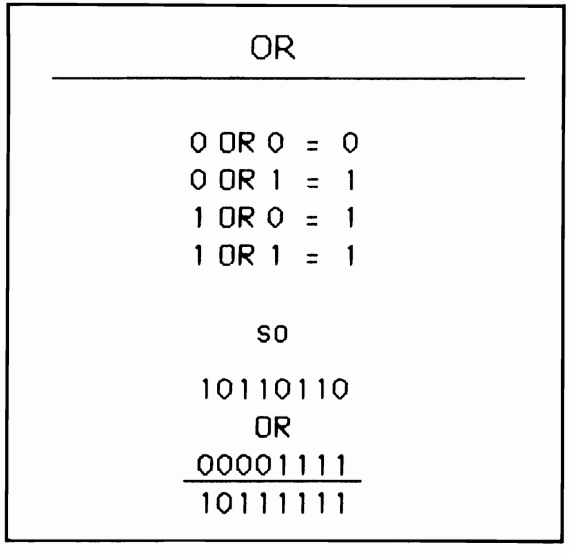


Fig. 5-25. OR operation.

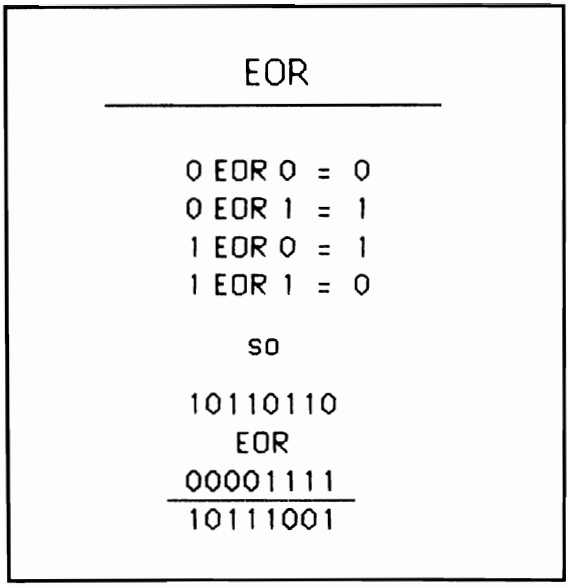


Fig. 5-26. EOR operation.

EOR compares a source and a destination operand bit position by bit position. Whatever bit positions are set (equal to 1) in one or the other operand, but not in both, are also set in the result. The other bit positions (those that have 0s in both operands) are cleared in the result. The operation is shown in Fig. 5-26. The result is stored in the destination and the condition code flags are set according to that result.

EOR is used to *invert* bits. By setting one of the operands with a desired sequence of 1s and 0s, you can invert chosen bit positions of the result. Wherever a 1 appears in your constructed operand, or mask, the value of that bit position in result will be the opposite of the value of that bit position in the second operand. Check this out by doing it, bit by bit, on paper.

EORI uses an immediate source operand and performs the same operation as EOR. EORI has two special forms.

EORI to CCR
EORI to SR

They perform the same EOR operation, but *EORI to CCR* works with a single immediate byte and the low-byte of the status register (the condition codes register (CCR). This instruction helps you control the flag values.

EORI to SR works with an immediate word and the full status register (SR). Because the System byte (the high byte) of the SR contains the flag that controls Supervisor and User mode, this instruction is privileged. The processor must be in Supervisor mode to execute it. If the CPU is only in User mode, EORI to SR will not execute, and a Trap will be generated instead.

NOT complements the value of the destination and then stores that new value in the destination. As shown in Fig. 5-27, the *one's complement* (which is the type used in this instruction) of a value changes all the 1s to 0s and the 0s to 1s.

Shift and Rotate

This group of functions doesn't correspond to high-level language functions as do the groups previously described. Still, the ability to move or change the contents of an operand within that operand's location is useful. As you will see, shift and rotate instructions perform similar functions. Figure 5-28 lists the 68000's shift and rotate instructions.

The bits of any location, register or memory, are numbered. Figure 5-29 shows the appropriate

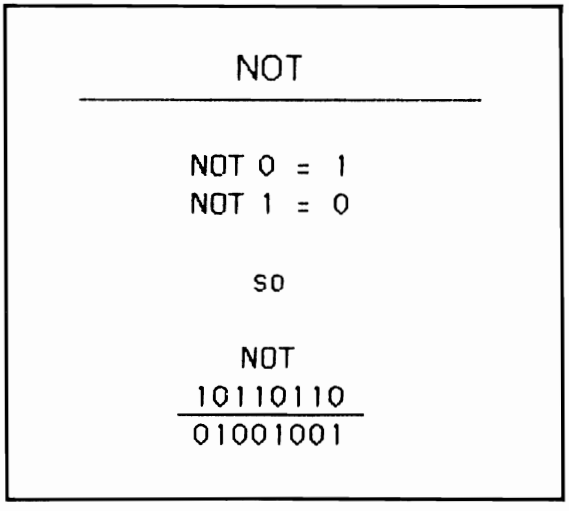


Fig. 5-27. NOT operation.

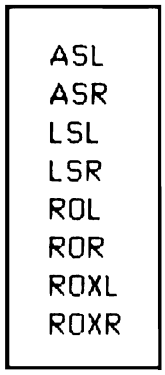


Fig. 5-28. Shift and rotate instructions.

scheme for both registers and memory locations on the 68000. The least significant bit is typically shown on the right, just as it true of binary numbers themselves. The bit positions traditionally start at 0 because bit string that are interpreted directly as binary numbers give the symbol in that position the value of 2 to the 0 power. In turn, each bit position to the left adds one to the power, as shown in Fig. 5-30. The result of this is that the most significant bit of an 8-bit byte is bit position 7; and of a 32-bit long-word is bit position 31.

The bit positions are storage cells that can hold either a 1 or a 0. Shift and rotate instructions move the values sideways from one bit position to the next through a register or memory location instead of transferring complete sets from one location to another.

Shift Instructions. *LSL* is called Logical Shift Left and is shown in Fig. 5-31. The term *logical* is used to differentiate this from *arithmetic* shifting, which will be described later in this section. LSL moves the bit values to the left, or toward the more significant bit position.

If you are using LSL on a register operand, you can choose the number of bit positions in the shift. A shift of one bit will move whatever was in the bit 0 position to the bit 1 position. the former bit 1 position value will be in the bit 2 position. Either immediate (for a shift of from 1 to 63 bits) data can specify the size of the shift.

The differences in shift operations normally appear in their treatment of the end bits. In other words, what happens to the bit that is shifted off the end of the operand and what appears in the bit position that is at the other end of the operand?

LSL feeds zeros into the least significant bit position. That is, the position that has a bit shifted out of it and no bit below it to shift in will be cleared. In fact for each bit position of shift, a zero will be put into the shifting string of values. A 3 bit position shift will put zeros in bit positions 0, 1, and 2. the arithmetic left shift, ASL, also feeds zeros into the least significant bit position.

LSL puts copies of the last value to be shifted out (of the most significant bit position) into both the C and X flags. the most significant bit position

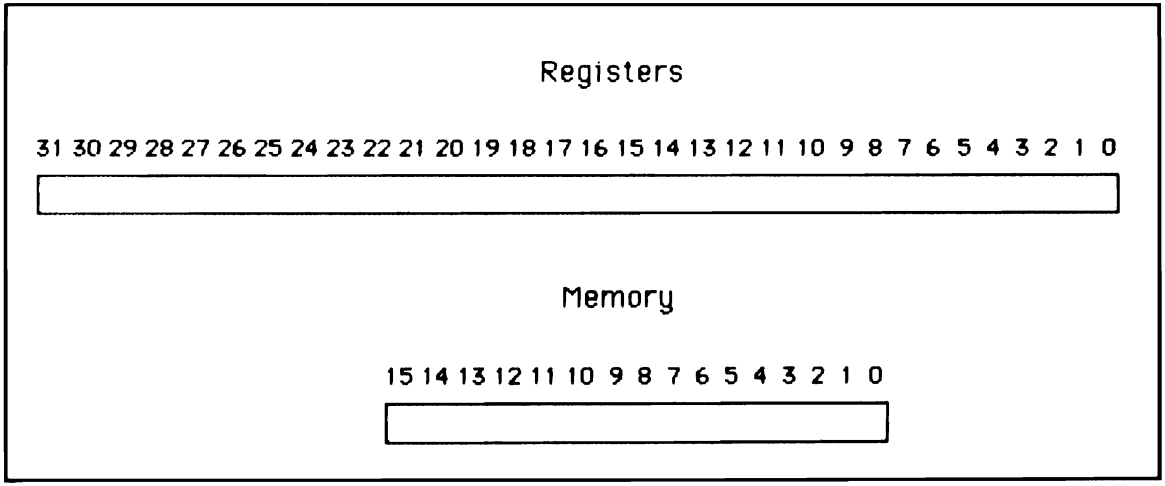


Fig. 5-29. Bit positions of registers and memory.

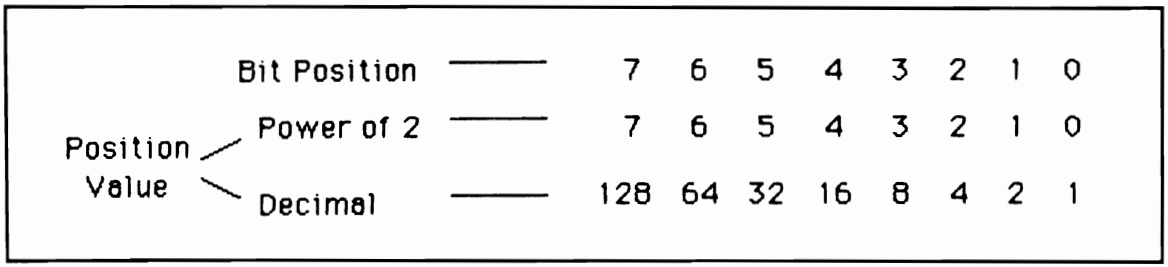


Fig. 5-30. Numeric values of bit positions.

doesn't have to be the bit 31 of a register. When shifting register contents, you can specify byte, word, or long-word shifts. The most significant bit positions of those various operand sizes differs. If you choose to shift a byte one bit position to the left, the former bit position 7 value will be copied into both the C and X flags. If you shifted that byte two bit positions instead of one, the former bit position 6 value would appear in the C and X flags, and the original bit position 7 value would just disappear.

LSR, Logical Shift Right, is similar to LSL. The main difference, naturally, is that LSR moves the bit values to the right, or toward the more significant bit position as shown in Fig. 5-32. As with LSL, there is an arithmetic right shift relative to LSR, called ASR, that will be discussed further on in this section.

If LSR is working on a register operand, you can choose the number of bit positions in the shift. A shift of one bit will move whatever was in the bit 1 position to the bit 0 position. The former bit 2 position value will be in bit 1 position. Figure 5-33 shows the LSR shift. Either immediate (for a shift or from 1 to 8 bit positions) or register (for a shift of from 1 to 63 bits) data can specify the size of the shift. As mentioned above,the orthogonality of the instruction set makes the sizing rules for LSL and LSR virtually the same.

LSR feeds zeros into the most significant bit position. That position has a bit shifted out of it and no bit above it to shift in. It is cleared for each bit position of shift. A 3 bit position shift of a long-word will put zeros in bit positions 29, 30, and 31.

The most significant bit position doesn't have to be the bit position 31 of a 32-bit register. When shifting register contents, you can specify byte, word, or long-word shifts. The most significant bit position of those various operands sizes differs. If you choose to shift a word one bit position to the right, the former bit 0 value will be copied into both the C and X flags. If you shifted that byte two positions instead of one, the former bit 1 value would appear in the C and X flags, and the original bit 0 value would just disappear.

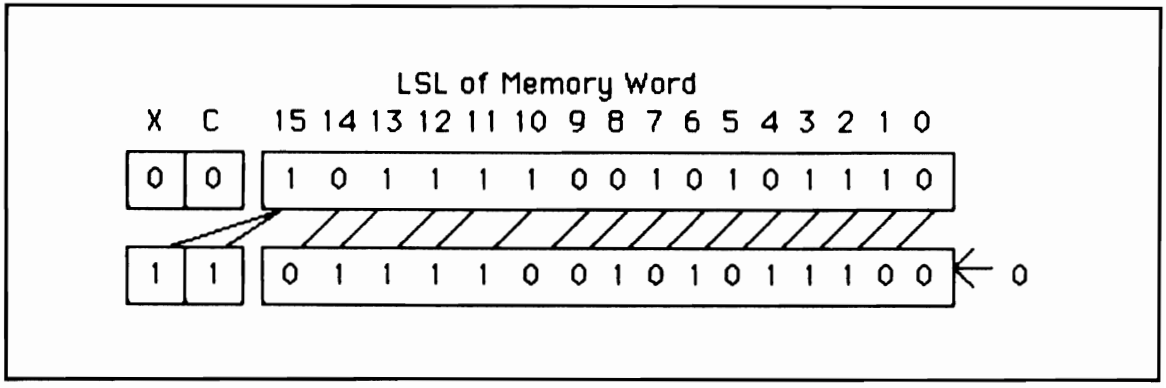


Fig. 5-31. LSL operation.

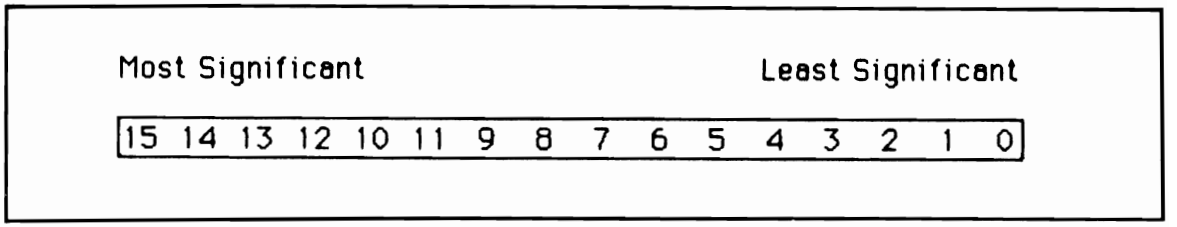


Fig. 5-32. Most significant and least significant bit positions.

ASL is called Arithmetic Shift Left. The term *arithmetic* is used to differentiate this from the *logical* shifts described earlier. ASL moves the bit values to the left, or toward the more significant bit position just as LSL does. The difference between the two instructions is in what happens at the ends of the operand.

If ASL is working on a register operand, you can choose the number of bit positions in the shift. A shift of one will move whatever was in the bit 0 position to the bit 1 position. The former bit 1 position value will be in bit 2 position. Figure 5-34 shows the ASL shift. Either immediate (for a shift of from 1 to 8 bit positions) or register (for a shift of from 1 to 63 bits) data can specify the size of the shift.

The differences in shift operations on any microprocessor normally appear in their treatment of the end bits. In other words, what happens to the bit that is shifted off the end of the operand and what appears in the bit position that is at the other end of the operand?

LSL feeds zeros into the least significant bit position. That is, the position that has a bit shifted out of it and no bit below it to shift in will be cleared for each bit position of shift. A 3-bit position shift will put zeros in bit positions 0, 1, and 2. The arithmetic left shift, ASL, also feeds zeros into the least significant bit position.

ASL puts copies of the last value to be shifted out of the most significant bit position into both the C and X flags. The most significant bit position doesn't have to be the bit 31 of a register. When shifting register contents, you can specify byte, word, or long-word shifts. The most significant bit positions of those various operand sizes differs. If you choose to shift a byte one bit position to the left, the former bit 7 value will be copied into both the C and X flags. If you shifted that byte two bit positions instead of one, the former bit 6 value would appear in the C and X flags, and the original bit 7 value would just disappear.

ASL puts a zero in the least significant bit position. That is why ASL is called an arithmetic shift. Shifting an operand one bit position to the left is the same thing as multiplying it by two. This fact is often used in complicated mathematical algorithms as long as no new digits are added to the operand. Shifting two bit positions to the left is the equivalent of multiplying by four, and so on. If some value other than a 0 were put in the least

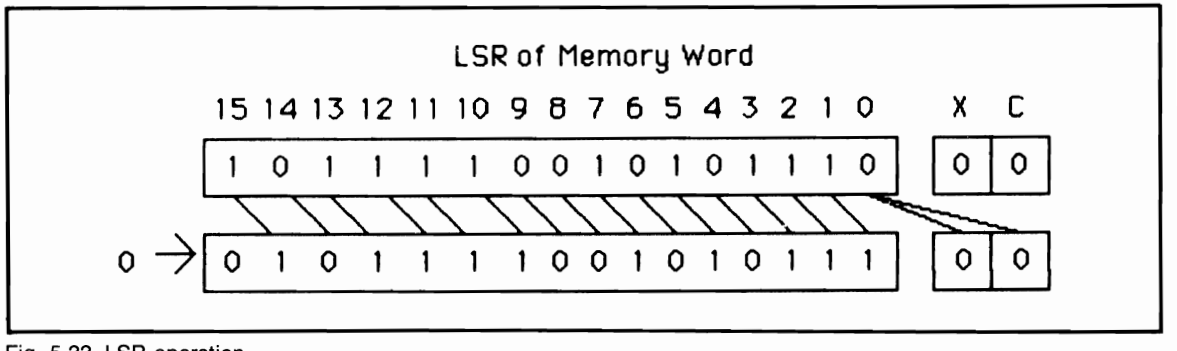


Fig. 5-33. LSR operation.

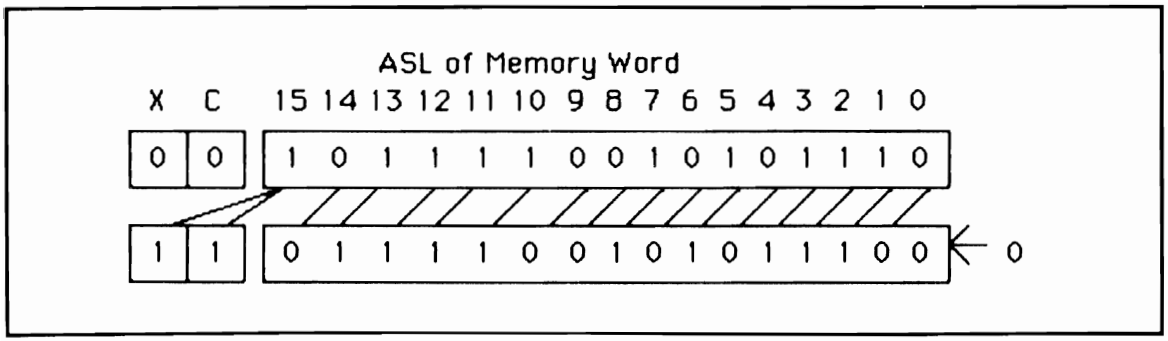


Fig. 5-34. ASL operation.

significant bit position (bit 0) this relationship wouldn't be true.

ASR is called Arithmetic Shift Right. The term *arithmetic* differentiates this from the *logical* shifts described earlier. ASR moves the bit values to the right, or toward the less significant bit position just as LSR does. The difference between the two instructions is in what happens at the ends of the operand.

If ASR is working on a register operand, you can choose the number of bit positions in the shift. A shift of one bit will move whatever was in the bit 1 position to the bit 0 position. the former bit 2 position value will be in bit 1 position. Figure 5-35 shows the ASR shift. Either immediate (for a shift of from 1 to 8 bit positions) or register (for a shift of from 1 to 63 bits) data can specify the size of the shift.

Unlike LSR, which feeds zeros into the most significant bit position, ASR puts a copy of the original most significant bit value into that position. In other words, the most significant bit value won't change, no matter how many bit positions are shifted. This means the sign value won't change. Furthermore, the two's complement binary arithmetic used requires that the top bit be repeated for the digits of the operand not to be altered. The outcome is that ASR works as a division operation. Each shift of a bit position to the right equals a division by two.

A point to keep in mind is that when shifting register contents, you can specify byte, word, or long-word shifts. The most significant bit positions of those various operand sizes differs. The bit that is repeated may be bit 7 (of a byte), bit 15 (of a word), or bit 31 (of a long-word).

ASR puts copies of the last value to be shifted out of the least significant bit position into both the C and X flags. If you choose to shift a word one bit position to the right, the former bit 0 value will be copied into both the C and X flags. If you shifted

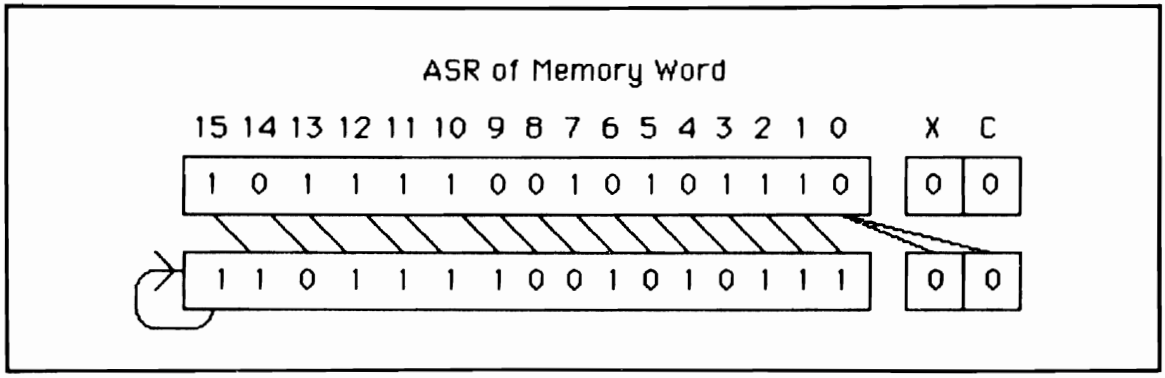


Fig. 5-35. ASR operation.

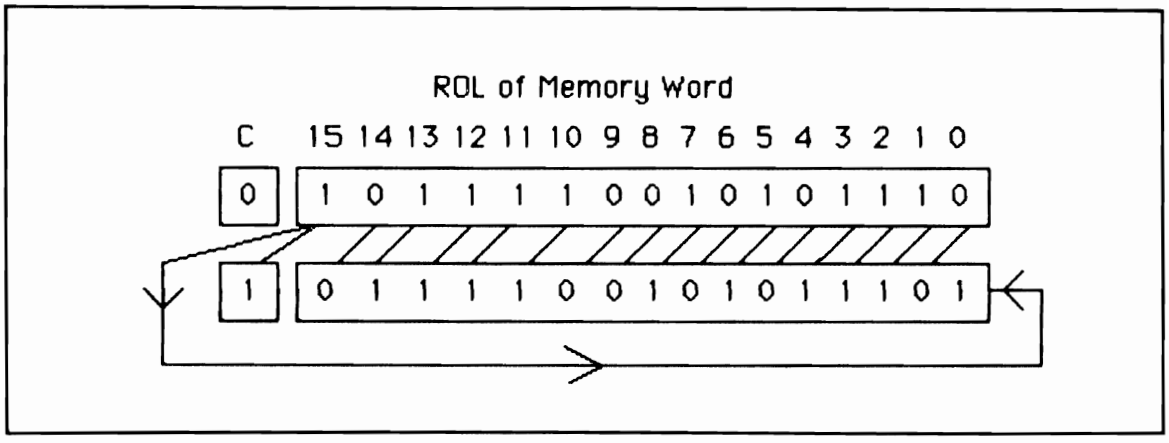


Fig. 5-36. ROL operation.

that word two bit positions instead of one, the former bit 1 value would appear in the C and X flags, and the original bit 0 value would just disappear.

Rotate Instruction. The shift instructions move the bit that falls off the end of the register or memory location into the C and X flags. Rotations move the bit around and put it back into the opposite end of the register or location.

ROL stands for Rotate Left. The action of this instruction is shown in Fig. 5-36. The bit values are moved to the left, more-significant end of the register or memory location. The bit value that moves out of the most significant bit position is moved into the C flag and the least significant bit position (bit 0).

Remember, as mentioned in the discussion of the Shift instructions, the most significant bit position will not necessarily be the highest bit of the register. Although memory locations can only work with word operands, registers can work with byte, word, or long-word operands. That means the most significant bit position in your operation on a register operand can be bit 7 (for a byte), bit 15 (for a word), or bit 31 (for a long-word).

You can choose the number of bit positions in the rotation. Immediate data can specify a rotation of from 1 to 8 bit positions. You may notice that a rotation of as many as 63 bits will be to loop the values through the register or location more than once. That is ok. Modulus arithmetic sometimes requires that sort of manipulation.

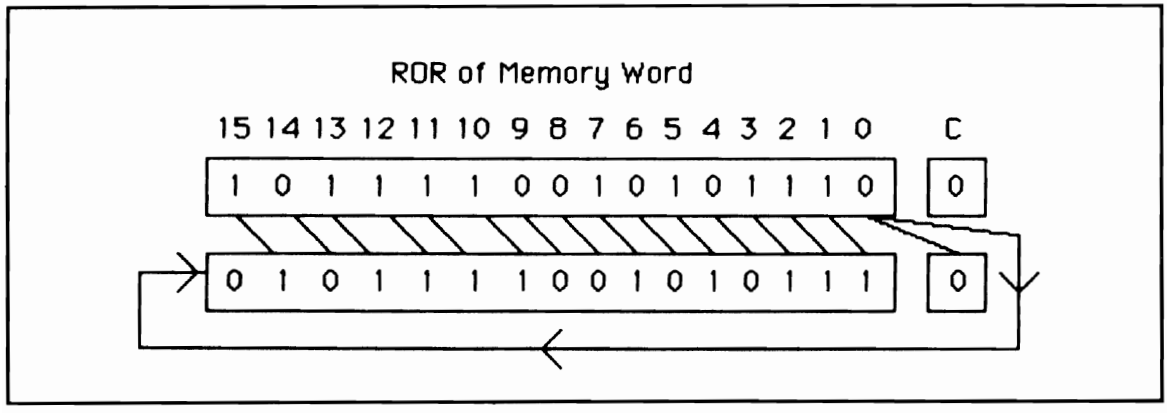


Fig. 5-37. ROR operation.

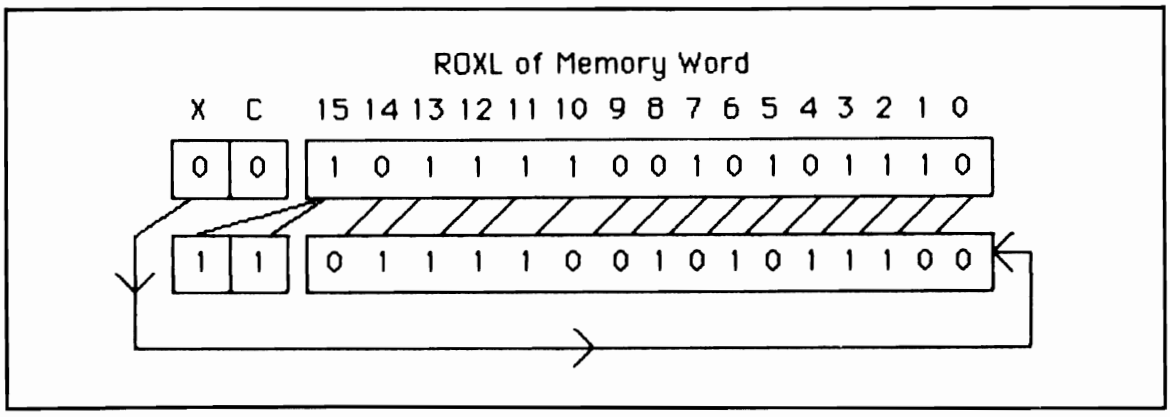


Fig. 5-38. ROXL operation.

ROR is a rotation right. This instruction uses the same rules except that the rotation direction is reversed from ROL. Figure 5-37 shows ROR's performance.

Because ROR moves the bits to the right, the bit that falls off and is copied into the C flag comes from the least-significant bit position.

ROXL is Rotate Left with Extend. That *with Extend* simply means that the X flag is put into the rotation loop (this is shown in Fig. 5-38). While ROL moves the last, most significant drop-off bit into the C flag, that flag is not actually in the loop. The drop-off bit also is rotated directly into the least significant bit position. ROXL puts the X flag in between the most significant and least significant bit positions, so that bits move from the most significant bit, to the X flag, and then to the least significant bit. The first bit value to be shifted into the least significant bit position comes from the X flag. The C flag still receives a copy of the same bit that will also go into the X flag.

ROXL offers the same choices of operand size and number of rotation positions as ROL. Registers can be shifted from 1 to 63 bit positions and can operate on bytes, words, or long-words. Memory locations can only work with words.

ROXR is Rotate Right with Extend. Figure 5-39 shows its action. ROXR works just like ROXL except that the rotation is to the right. The X flag is put between the least significant and the most significant bit positions.

Bit Manipulation

These instructions can work on individual bits,

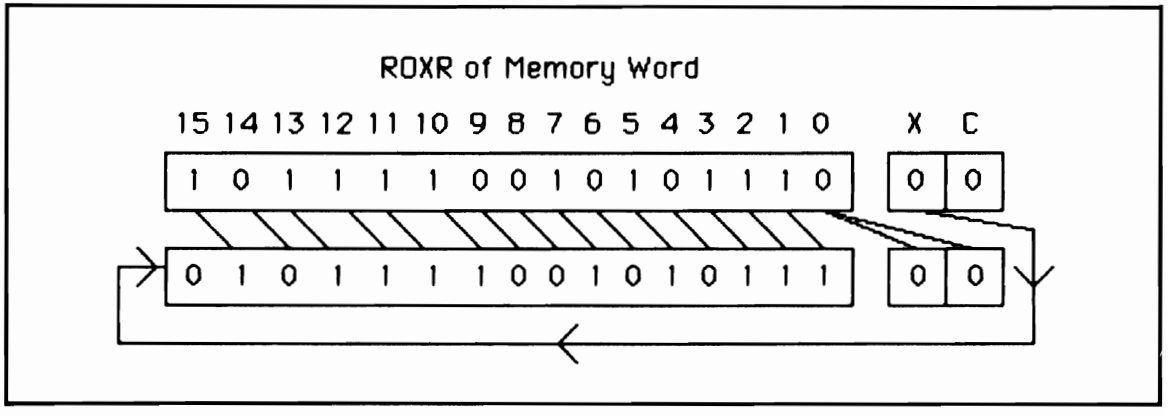


Fig. 5-39. ROXR operation.

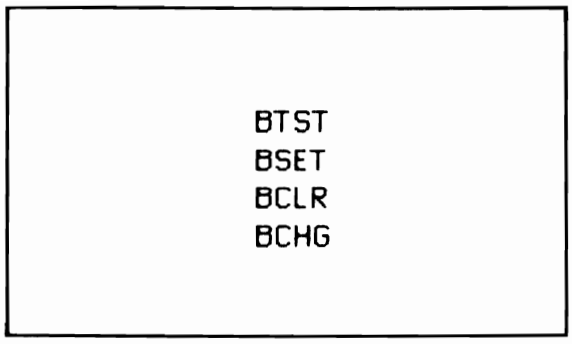


Fig. 5-40. Bit manipulation instructions.

instead of bytes, words, or long-words. Figure 5-40 lists the four 68000 bit manipulation instructions.

BTST is the Test a Bit instruction. BTST simply tests the value of a particular bit location and then uses the Z flag to communicate that value to you. If the Z flag is set (equal to 1), the tested bit was zero. If the Z flag is cleared (equal to 0), the tested bit was equal to 1. That may sound contradictory; it is. But what choice did the designers have? If they had reflected the bits value directly in the Z flag, the same logic of "setting if the condition is met" that is used for other flags couldn't have been used for the Z flag. That, in turn, would have seemed contradictory.

BSET is called Test a Bit and Set. After testing the particular bit, BTST only works on the Z flag. BSET tests the particular bit, works on the Z flag, and then sets (equal to 1) the tested bit. The Z flag is used in the same way as described above for BTST.

BCLR is called Test a Bit and Clear. Like BSET and BTST, BCLR tests a single bit and then uses the Z flag to report the value. BCLR, however, then clears the tested bit.

BCHG is Test a Bit and Change. As you might guess from the previous descriptions, BCHG tests a particular bit, reports the value in the Z flag, and then changes the value of the tested bit to the opposite of whatever it originally was (a 1 in that position would become a 0 and a 0 would become a 1).

Program Control Operations

The Program Control instructions, listed in Fig. 5-41, are used in loops, jumps, and subroutines. There are three subgroups within this main group.

Unconditional Branches and Jumps. These are the easiest program control instructions to understand. Any of these instructions forces the program to continue processing at a new position within the program. Program instructions are normally written in a sequence in memory. The address of the instruction that is to be executed next is kept in the PC (Program Counter) register. Unconditional control instructions put a new value into the PC so that processing will continue at a new address.

BRA means Branch Always. *Branching* is a computerese term that means changing to another part of the program. BRA uses Program Counter with displacement addressing mode to find the new PC value. This mode adds an 8- or 16-bit, two's complement number to the old PC value. This means that a new PC value between 32766 bytes behind

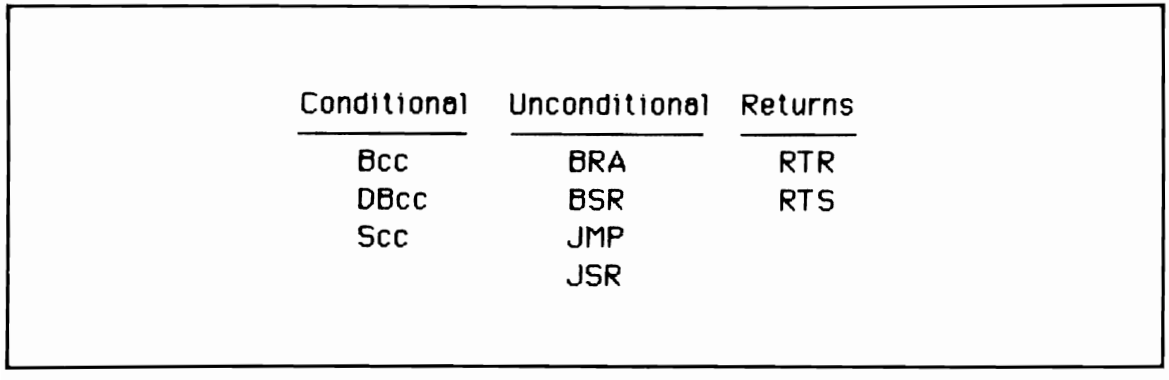


Fig. 5-41. Program control instructions.

or 32769 bytes ahead of the old value can be used. 8-bit microprocessors are normally limited to 8-bit displacement values, which only allows jumps to −126 to +129 bytes. The forward and backward values are not equal for two reasons.

First, two's complement representation allows a larger negative number than positive number with a limited number of bits. More importantly, the displacement is from the PC value at the end of the BRA instruction. That value has been incremented by two (because that is the length of the BRA instruction) from the beginning of BRA. When you use machine code you have to keep this rule in mind. When you use assembly language, however, you only need calculate from the beginning of the BRA instruction; the assembler will automatically subtract two from your suggested branch displacement.

BSR means Branch to Subroutine. This instruction is almost identical to BRA. The difference is that BSR saves the address of the next instruction on the stack before adding the displacement value to the PC. The advantage of BSR and the reason that *subroutine* is in the name is that processing can return to the saved address after the subroutine is complete. If BRA were used, the CPU wouldn't know where to return to. RTS returns processing after a subroutine and is explained below.

It is a good programming habit to use labels instead of absolute displacement values for branching addresses. Labels keep the program code flexible: slight changes will not ruin labeled code but they can totally disrupt absolute code. Chapter 9 explains labels in more detail.

JMP (which stands for Jump) forces processing to continue at a new address. The major difference between JMP and BRA is that while BRA can only use the Program Counter Relative with Displacement addressing mode, JMP can use any of seven different addressing modes to find the new PC value. That flexibility means that JMP can force processing to any point in the 68000's address space. JMP is an unconditional program control statement.

JSR (Jump to Subroutine), like BSR, saves a value on the stack so that processing can return to its present position after the subroutine is processed. The value saved is the address of the instruction right after JSR. JSR is an unconditional jump and can use any of the seven addressing modes JMP uses. RTS, explained below, is the instruction used to return processing after the subroutine.

Conditional Branches and Jumps. *Conditional* means the branch or jump isn't always made. If the state of the flags isn't right, the conditional instruction will perform no function at all except to increment the PC (so the next instruction will be addressed).

Bcc stands for Branch Conditionally. Like BRA, Bcc changes the PC value using Program Counter Relative with Displacement mode to execute a different part of the program. Bcc, however, first checks to see if a certain condition is met. The conditions the programmer can choose from are listed in Fig. 5-42. Once one of these conditions is added to the root Bcc instruction, the mnemonic changes. For instance, the instruction to "branch if the overflow is set" is coded as follows:

BVS

The second and third letters signifying the condition. Some of the conditions are straightforward. For example, Carry Set and Carry Clear simply

```
CC  Carry Clear
CS  Carry Set
EQ  Equal
GE  Greater or Equal
GT  Greater Than
HI  High
LE  Less of Equal
LS  Low or Same
LT  Less Than
MI  Minus
NE  Not Equal
PL  Plus
VC  Overflow Clear
VS  Overflow Set
```

Fig. 5-42. Bcc conditions available.

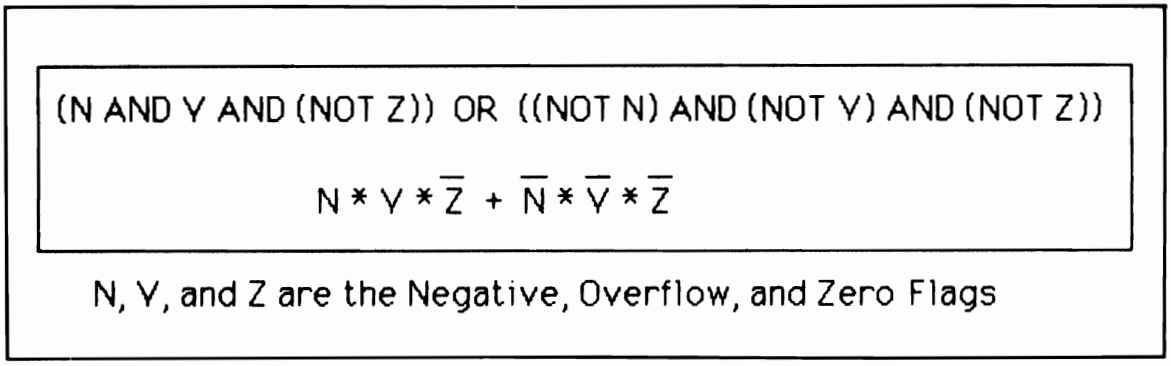


Fig. 5-43. Boolean equation for the "greater than" condition.

refer to the value of the C (carry) flag. Other conditions are more complex. "Greater than" can be represented by a Boolean equation as shown in Fig. 5-43. Boolean Algebra is a special mathematics that is often used in computer science. The logical instructions described earlier in this chapter (AND, OR, EOR, NOT) are the basic ingredients of Boolean Algebra. Those ingredients are then combined into more complicated, compound statements. Don't worry if you don't understand them when you begin programming; you won't need to use them. The same warning about using labels instead of absolute displacements and the discussion of the distance you can branch that accompanied the BRA instruction above, also apply to Bcc.

If the condition is met, the branch is made. If the condition is not met, the branch isn't made, and processing continues with the instruction following Bcc.

DBcc is called Test Condition, Decrement, and Branch (though the mnemonic looks more like Decrement and Branch on Condition). Many microprocessors have an instruction such as this one that helps the programmer set up loops. Loops, where processing repeats itself more than once, are very useful structures in programs. Programs typically use a counter to know how many times to process through a loop. This is a value stored in a particular place that specifies how many times to loop.

DBcc first tests a condition, just as Bcc does. (DBcc does have two more condition choices than Bcc has: the full DBcc set is shown in Fig. 5-44.) As with Bcc, the DBcc instruction mnemonic takes on the two letters of the condition. Test for Equal, Decrement, and Branch is coded as follows:

DBEQ

You will never see DBcc in a program. Two condition letters must take the place of the cc.

If the chosen condition is met, no operation is performed and processing continues with the in-

```
CC  Carry Clear
CS  Carry Set
EQ  Equal
F   False
GE  Greater or Equal
GT  Greater Than
HI  High
LE  Less or Equal
LS  Low or Same
LT  Less Than
MI  Minus
NE  Not Equal
PL  Plus
T   True
VC  Overflow Clear
VS  Overflow Set
```

Fig. 5-44. DBcc conditions available.

struction after DBcc. If the condition is not met, the next stage of DBcc takes place. The value of the low word of a specified data register, called the counter, is decremented by 1. If the counter then equals −1, the counter is *exhausted.* That is, the counter has counted all the way down. In that case, processing continues with the instruction after DBcc. If the counter is not yet equal to −1, a branch is made. A displacement value is added to the PC just as BRA would make. Using labels is also a good idea with DBcc.

The actions of DBcc can be summarized as follows:

1. The condition is tested. A true condition sends processing to the next instruction. This is the opposite of what is normally implemented. Most instrucitons would perform the branch if the condition were true. In fact, Bcc is the opposite of this: Bcc makes the branch if the condition is met (is true).
2. If the condition was false, the counter is decremented. Once the counter reaches −1, the processing continues with the next instruction. If the counter hasn't reached −1 yet, the branch is made.

The branch will typically send processing back to repeat instructions just executed. In the way, DBcc is the last instruction in a loop which will repeat until the condition changes or the counter reaches −1. By setting the counter properly, the programmer can control the number of times the program runs through the loop.

Return Instructions. These instructions bring the program back, or return it, after a subroutine has been processed. They differ in what values they retrieve from the stack.

RTS is the basic return instruction. Return from Subroutine pulls a value from the stack and puts it into the program counter. Processing then continues at the new PC location. If the PC value was saved before a subroutine was processed, and the subroutine didn't alter the stack, processing will have returned to the instruction after the branch or jump to subroutine instruction.

RTR (Return and Restore Condition Codes) does just a bit more work than RTS. RTR first pulls the condition codes off of the stack, then it pulls the PC value off. Just as with RTS, processing will return to the point after the subroutine branch or jump, but RTR will have restored more of the CPU's status because the flags will have their previous value.

RTE (Return from Exception) is a privileged instruction. If the CPU is in supervisor state, RTE will put the complete SR (shift register) value and then the PC value off of the stack. (The Supervisor stack is used because the 68000 is in Supervisor mode.) If the CPU isn't in Supervisor mode, RTE will generate a Trap.

System Control

The 68000 System Control instructions that are listed in Fig. 5-45 deal with the status register, the condition codes, or traps and are divided into three groups: privileged, Trap generating, and condition code register.

Privileged. These instructions are privileged to restrict User programs from changing the system status. Most can alter the status register contents. Privileged instructions will only execute when the CPU is in Supervisor mode. If the CPU is in User mode, a Trap will be generated (Chapter 7 describes Exceptions and Traps).

ANDI to SR, EORI to SR, ORI to SR, MOVE to SR, MOVE USP, and *RTE* have all been described earlier in this chapter. ANDI to SR, EORI to SR, and ORI to SR in the section on Logical Instructions; MOVE to SR and MOVE USP in the section on Data Movement instructions; and RTE in the Program Control section.

RESET asserts the Reset line (signal wire) which resets all external devices. Processing then continues with the next instruction.

STOP loads the immediate word (the second word of the STOP instruction) into the status register (SR). The PC is incremented as it would be with any instruction and then the CPU stops. While stopped, no instructions are fetched or executed. The CPU will remain stopped until a trace, interrupt, privilege, or reset exception occurs.

Privileged	Trap Generating	Condition Code Register
ANDI to SR	CHK	ANDI to CCR
EORI to SR	TRAP	EORI to CCR
MOVE EA to SR	TRAPV	MOVE CCR to EA
MOVE SR to EA		ORI to CCR
MOVE USP		
ORI to SR		
RESET		
RTE		
STOP		

Fig. 5-45. System control instructions.

1. If the T (trace) bit of the status register is set (meaning the CPU is the trace state) a trace exception will occur when STOP is executed.

2. Any interrupt request which has priority higher than the current processor priority (which is stored as the interrupt priority mask in the status register) will cause an interrupt exception. A lower priority interrupt request will have no effect.

3. If the value loaded into the status register clears the S bit (the Supervisor mode flag), a privilege violation exception will occur immediately.

4. An external reset (a low pulse on the reset line) will always generate a reset exception.

Condition Code Register. These instructions manipulate the low byte of the status register. That byte contains the X, N, Z, V and C flags. *ANDI to CCR, EORI to CCR, ORI to CCR,* and *MOVE to CCR* have all been described earlier in this chapter: ANDI to CCR, EORI to CCR, and ORI to CCR in the Logical section; MOVE to CCR in the Data Movement section.

Trap Generating. Traps are a particular sort of internal interrupt that the 68000 uses to control processing. They are described in detail in Chapter 7.

CHK can Check a value against set limits. If the value is outside those limits, an exception is generated and regular processing ceases while exception processing takes over.

CHK examines the low word of a data register and compares it to a specified *upper bound* (a two's complement integer that can be addressed with any of 11 addressing modes). If the examined data register word is less than zero or greater than the upper bound, a Trap is generated. Normal processing ceases and exception processing begins with the CHK exception vector. CHK is very useful for a quick and easy check to see that a number, such as an array size, hasn't been exceeded.

TRAP generates a Trap and initiates exception processing. Any of 16 different Trap vectors can be specified by immediate data. The vector specifies at what address the exception processing should begin.

TRAPV, called Trap on Overflow, is a conditional form of the TRAP instruction. TRAPV examines the V (overflow) flag and generates a trap if that flag is set (equal to 1). The exception processing takes place at the TRAPV vector address. If the V flag is not set, processing continues with the instruction after TRAPV.

Nothing Instructions

This is not a joke category. Most microprocessors have such instructions. On the 68000 they are called NOP and ILLEGAL (shown in Fig. 5-46).

Fig. 5-46. Nothing instructions.

NOP stands for No Operation. All this instruction does is increment the PC (Program Counter) so the next instruction is ready to execute. Why have it at all? A finite length instruction can help in building timing loops. Having a harmless filler can also give you extra space in a program while you are still debugging that program.

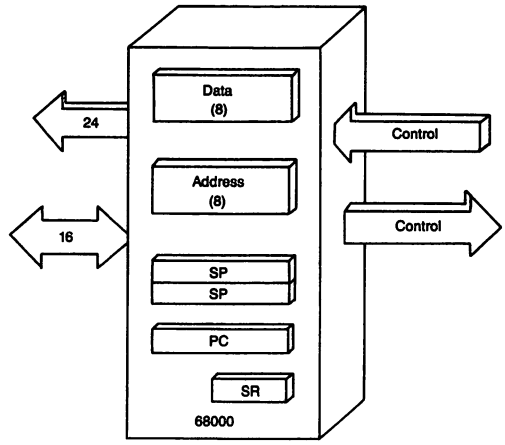
ILLEGAL is a very interesting instruction. Using it will cause an illegal instruction exception. In fact, that is what will happen when the CPU tries to execute any object code that doesn't represent one of the instructions listed in this chapter. The difference with the particular illegal instruction code of this instruction and all other illegal instruction codes is that Motorola promises that this code will remain illegal. While future, improved chips (such as the 68020) will use some of the object codes that are illegal on the 68000 for their own legal instructions, the ILLEGAL object code will remain illegal through all future chips.

Why have this instruction at all? So that the programmer can work with the Illegal instruction exception and yet write upwardly compatible code. By using ILLEGAL instead of some random object code, the programmer is ensuring that the program will run the same way on future 6800 family chips.

SUMMARY

Those are all of the 68000 instructions. While this chapter attempted to describe them briefly and show their relation to other 68000 instructions. Chapter 6 shows the actual code, condition code effects, and addressing modes for each and every instruction.

6



Instruction Set

THE PREVIOUS CHAPTER DESCRIBES WHAT AN instruction is and what general categories of instructions the 68000 provides to the programmer. This chapter describes all of the 68000 instructions individually. It does not cover the instructions that are specific to the other 68000 CPUs (such as the 68020).

The instructions are listed alphabetically by their mnemonics (remember that the mnemonics are the abbreviated instruction names used in assembly language). The information vital to programmers appears at the top of the listing: mnemonic, definition (the full name of the instruction), description (what the instruction does), addressing (which addressing modes you can use), operand size (byte, word, and long-word), flag effects (which condition codes are affected and how), and notes (special aspects of the instruction or pitfalls to watch out for). Following these elements is a breakdown of the binary object code for the instruction (this bit by bit description of the instruction is important to a complete understanding of how microprocessors are designed and built, but is not necessary for programming). Assembly language syntax (the way the instruction is written) depends on the addressing mode used. Read Chapter 4 to learn how the different modes are written.

If you don't know the mnemonic for a particular instruction, check in the instruction groups in Chapter 5. For instance, if you want to do some BCD arithmetic, check in the Decimal group. Then, when you find a particular instruction (such as ABCD for BCD addition) come back to the individual descriptions in this chapter.

Browsing through these instructions is a great way to glimpse the power of the 68000. Once you know some instructions, a few addressing modes, and the assembly language format, you'll be ready to experiment with assembly language.

ABCD

Definition: add decimal (with extend).

Description: ABCD adds the least significant byte (the bottom byte) of source operand, and the value of the extend flag, to the least significant byte of the destination. It then stores the result in the destination. The term *decimal* means that this addition is done with BCD (Binary Coded Decimal) arithmetic. ABCD has two major cases, register-to-register and memory-to-memory.

1. Register-to-register uses data registers for both source and destination.
2. Memory-to-memory uses memory locations for both source and destination. The memory address of the source operand is stored in an instruction-specified address register, and the destination address is stored in another instruction-specified address register. Before the operation, each address register is decremented (predecrement addressing mode).

Addressing:

1. Register-to-register. Data Register Direct mode is used for both source and destination. (The values are stored in registers and are specified directly by the instruction).
2. Memory-to-memory. Address Register Indirect mode is used for both source and destination (the instruction specifies the address registers that hold the memory addresses of both source and destination). Before the values in the address registers are used, they are decremented by 1. This helps in multibyte BCD addition.

Operand size: byte.

Instruction length: 1 word.

Condition code effects:

C Set if a decimal carry results from the operation; otherwise cleared.
X Set if a decimal carry results from the operation; otherwise cleared.
Z Cleared if the result is not equal to zero; unchanged if the result equals zero.
N Undefined (could be set or cleared).
V Undefined (could be set or cleared).

Object code:

1. Register-to-register: 1100aaa100000bbb

Breakdown

1100: ABCD instruction.
aaa: Destination data register.
100000: Specifies register-to-register case.
bbb: Source data register.

2. Memory-to-memory: 1100aaa1000001bbb

Breakdown

1100: ABCD instruction.
aaa: Destination data register.

100001: Specifies memory-to-memory case.
bbb: Source data register.

Notes: Programmers often set the Z flag before using this instruction for multiple-precision arithmetic. The set flag makes it easy to check for a zero result.

ADD

Definition: add (binary).
Description: ADD adds the contents of the source to the contents of the destination and stores the result in the destination. There are two forms of this instruction that differ only in addressing (as described in the next paragraph).
Addressing: ADD can be used with any of a large number of addressing modes. The two forms of this instruction offer different addressing choices.

1. Data Register Direct destination. The destination must be addressed by Data Register Direct mode. Any addressing mode can be used for the source.

2. Data Register Direct source. The source must be addressed by Data Register Direct mode. Almost any addressing mode (except Program Counter Relative with Displacement, Program Counter Relative, or Immediate) can be used for the destination including:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word. As noted below, bytes cannot be used with Address Register Direct mode.
Instruction length: 1 word.
Condition code effects:

C Set if a carry occurs; otherwise cleared.
X Set if a carry occurs; otherwise cleared.
Z Set if the result is zero; otherwise cleared.
N Set if the result is negative; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.

Object code: 1101aaabccdddeee

Breakdown

1101: ADD instruction.
aaa: Data register number (for either source or destination Data Register Direct addressing).

b: Operating mode.
0 means the data register is the destination.
1 means the data register is the source.
cc: Size specification.
00 means byte.
01 means word.
10 means long-word.
ddd: Addressing mode.
eee: Addressing register number.

Notes: If Address Register Direct addressing is used, the operand size cannot be specified as byte because address registers cannot work with bytes (only with words and long-words).

ADDA

Definition: add address.
Description: ADDA is a special case of the ADD instruction. ADDA adds the source operand to the specified address register and stores the result in that address register.
Addressing: The destination is only reached by Address Register Direct mode. Any mode can be used for the source operand.

Data Register Direct
Address Register Direct
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: words or long-words. The full destination address register is used no matter which operand size is chosen. A word source-operand will be size-extended to a long-word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 1101aaab11cccddd

Breakdown

1101: ADDA instruction.
aaa: Address register (destination).
b: Size specification.
0 means word.

1 means long-word.
11: ADDA instruction continued.
ccc: Source addressing mode.
ddd: Source addressing register.

ADDI

Definition: add immediate.
Description: ADDI adds immediate data (which is contained in the next byte or bytes of the instruction) to the specified destination operand. The result is stored in the destination.
Addressing The source is addressed by Immediate mode. The destination is reached any of:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 2 words (if the immediate data is a byte or a word, the first word is the instruction and the second contains the data); 3 words (if the immediate data is a long-word, the first word is the instruction and the next two are the long-word data).
Condition code effects:

C	Set if a carry occurs; otherwise cleared.
X	Set if a carry occurs; otherwise cleared.
Z	Set if the result is zero; otherwise cleared.
N	Set if the result is negative; otherwise cleared.
V	Set if an overflow occurs; otherwise cleared.

Object code: First word (00000110aabbbccc)
Second word (immediate data)
Third word (immediate data)

Breakdown

00000110: ADDI instruction.
aa: Size specification.
00 means byte.
01 means word.
10 means long-word. If a byte is specified, the low-order byte of the next instruction is used by the assembler.
bbb: Destination addressing mode.

ccc: Destination addressing register.
immediate data: Byte data is held in the low-order byte of the second word. Word data is the second word. Long-word data requires a three word instruction with the second and third words representing the data.

ADDQ

Definition: add quick.
Description: ADDQ adds immediate data (contained within the instruction word itself) to the specified destination operand. The result is stored in the destination. As the definition implies, ADDQ is used for quick execution.
Addressing: For the source operand, you can use only Immediate mode. For the destination operand, you can use any of these modes:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:
C Set if a carry occurs; otherwise cleared.
X Set if a carry occurs; otherwise cleared.
Z Set if the result is zero; otherwise cleared.
N Set if the result is negative; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.

Object code: 0101aaa0bbcccddd

Breakdown

0101: ADDQ instruction.
aaa: Data field (holding three bits of immediate data with 000 representing 8 and 001 through 111 representing 1 through 7).
0: ADDQ instruction (cont.).
bb: Operand size specification.
00 means byte.
01 means word.
10 means long-word. If a byte is specified, the low-order byte of the next instruction is automatically used by the assembler.
ccc: Destination addressing mode.
ddd: Destination addressing register number.

Note: If Address Register Direct addressing is used, the operand size cannot be specified as byte because address registers cannot work with bytes (only with words and long-words).

ADDX

Definition: add extended.
Description: ADDX adds the source contents and the extend flag to the destination contents. Stores the result in the destination. ADDX has two major cases, register-to-register and memory-to-memory.

1. Register-to-register uses data registers for both source and destination.
2. Memory-to-memory uses memory locations for both source and destination. The memory address of the source operand is stored in an instruction-specified address register and the destination address is stored in another instruction-specified address register. Before the operation, each address register is decremented (predecrement mode).

Addressing: Register-to-register uses Data Register Direct mode; Memory-to-memory uses Address Register Indirect mode.
Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

C Set if a carry occurs; otherwise cleared.
X Set if a carry occurs; otherwise cleared.
Z Cleared if the result is nonzero; otherwise unchanged.
N Set if the result is zero; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.

Object code:

1. Register-to-register: 1101aaa1bb000ccc

Breakdown

1101: ADDX instruction.
aaa: Destination data register number.
1: ADDX instruction (cont.).
bb: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
000: ADDX instruction (cont.) Register-to-register case.
ccc: Source addressing data register number.

2. Memory-to-memory: 1101aaa1bb001ccc

Breakdown

1101: ADDX instruction.
aaa: Destination address register number.
1: ADDX instruction (cont.).

bb: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
001: ADDX instruction (cont.) Memory-to-memory case.
ccc: Source address register number.

AND

Definition: AND logical.
Description: AND performs a logical AND operation on the contents of a specified destination operand. Stores the result in the destination. There are two forms of this instruction, that differ only in addressing, as described in the next paragraph.
Addressing: AND can be used with any of a large number of addressing modes. The two forms of this instruction offer different addressing choices.

1. Data Register Direct destination. The destination must be addressed by Data Register Direct mode. Any addressing mode except Address Register Direct can be used for the source including:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

2. Data Register Direct source. The source must be addressed by Data Register Direct mode. Almost any addressing mode (except Program Counter Relative with Displacement, Program Counter Relative, and Immediate) can be used for the destination including:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word. As noted just above, bytes cannot be used

with Address Register Direct mode.
Instruction length: 1 word.
Condition code effects:

N Set if the MSB (Most Significant Bit) of the result is one; otherwise cleared.
Z Set if the result is zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 1100aaabccdddeee

Breakdown

1100: AND instruction.
aaa: Data register number (for either source or destination, depending on case).
b: Determines addressing case.
0 means data register is destination.
1 means data register is source.
cc: Operand size.
00 means byte.
01 means word.
11 means long-word.
ddd: Addressing mode.
eee: Addressing register number.

Notes: If Address Register Direct addressing is used, the operand size cannot be specified as byte because address registers cannot work with bytes (only with words and long-words).

ANDI

Definition: AND logical immediate.
Description: ANDI performs a logical AND operation on the contents of a specified destination operand and an immediate value (contained in the next program words). ANDI then stores the result in the destination.
Addressing: The source is addressed by Immediate mode. The destination is reached any of:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 2 words (if the immediate data is a byte or a word, the first word is the instruction and the second contains the data); 3 words (if the immediate data is a long-word, the first word is the instruction and the next two words are the long-word data).
Condition code effects:

N Set if the MSB (Most Significant Bit) of the result is one; otherwise cleared.
Z Set if the result is zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: First word (00000010aabbbccc)
Second word (immediate data)
Third word (immediate data)

Breakdown

00000010: ANDI instruction.
aa: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
bbb: Addressing mode.
ccc: Addressing register number.
immediate data: Byte data is held in the low-order byte of the second word. Word data is the second word. Long-word data requires a three word instruction with the second and third words representing the data.

ANDI to CCR

Definition: AND logical immediate to condition codes register.
Description: This is a special form of the ANDI instruction. ANDI to CCR performs a logical AND operation of the contents of the condition code register (CCR: the low-order byte of the status register) and the immediate value (contained in the low-order byte of the next program word). The result is then stored in the CCR.
Addressing: The source is addressed by Immediate mode. The destination is the condition code register.
Operand size: byte.
Instruction length: 2 words (the first is the instruction word and the second contains the data as the low-order byte. The high order byte of the second word holds all zeros.)
Condition code effects: (These are set directly from the operation's results, which are stored in the flag register.)

N Cleared if bit 3 of the immediate operand is zero; otherwise unchanged.
Z Cleared if bit 2 of the immediate operand is zero; otherwise unchanged.

V Cleared if bit 1 of the immediate operand is zero; otherwise unchanged.
C Cleared if bit 0 of the immediate operand is zero; otherwise unchanged.
X Cleared if bit 4 of the immediate operand is zero; otherwise unchanged.

Object code: First word (0000001000111100)
Second word (immediate data)

Breakdown

0000001000111100: ANDI to CCR instruction.
immediate data: The high-order byte is filled with zeros; the low-order byte holds the immediate data.

ANDI to SR

Definition: AND logical immediate to status register (privileged).
Description: This is a special form of the ANDI instruction and is a privileged instruction. The CPU must be in the supervisor state for this instruction to execute. ANDI to SR performs a logical AND operation on the contents of the full status register and the immediate value contained in the next program word. The result is stored in the status register. Clearly, this instruction must be privileged because it can change the entire state of the CPU.
Addressing: The source is addressed by Immediate mode. The destination is the status register.
Operand size: word.
Instruction length: 2 words (the first is the instruction word and the second contains the immediate value).
Condition code effects: (These are set directly from the operation's results, which are stored in the status register).
N Cleared if bit 3 of the immediate operand is zero; otherwise unchanged.
Z Cleared if bit 2 of the immediate operand is zero; otherwise unchanged.
V Cleared if bit 1 of the immediate operand is zero; otherwise unchanged.
C Cleared if bit 0 of the immediate operand is zero; otherwise unchanged.
X Cleared if bit 4 of the immediate operand is zero; otherwise unchanged.

Object code: First word (0000001001111100)
Second word (immediate data)

Breakdown

0000001001111100: ANDI to SR instruction.
immediate data: The full 16 bits of this word hold the immediate data to be added to the status register.

ASL

Definition: arithmetic shift left.
Description: Shifts the bits of the specified location to the left. The last bit shifted out of the most significant bit position will be copied into both the C and X flags. The least significant bit position will be filled with a 0. A register or a memory location can be shifted.

1. Register shifts can range from 1 to 63 bit positions. The number of positions shifted (the shift count) is specified in one of two ways. A. The shift count can be held in the least significant six bits of a data register. That can specify a shift of from 0 to 63 bits. B. Immediate data can specify a shift number that can range from 1 to 8 bits. (A value of 000 means a shift of 8 positions.)

2. Memory shifts can only be a single bit position.

Addressing:

1. Register. Uses Register Direct mode for the register to be shifted (the destination).

2. Memory. Uses any of the following modes for the memory location to be shifted:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
2. Memory: word.

Instruction length: 1 word.

Condition code effects:

N Set if the most significant bit of the result is 1; otherwise cleared.

Z Set if the result equals zero; otherwise cleared.

V Set if the most significant bit changes at any point during the shift; otherwise cleared.

C Set by the last bit shifted out of the operand (which is copied into the C flag). C is cleared if a shift count of zero is performed.

X St by the last bit shifted out of the operand. That bit is copied into the X flag. X, unlike C, is unaffected by a shift count of zero.

Object code:

1. Register: 1110aaa1bbc00ddd

Breakdown

1110: ASL instruction.

aaa: Shift count or number of specified register(which it is depends on bit 5, "c", as described below).

1: ASL instruction.

bb: Operand size specification.
00 means byte.
01 means word.
11 means long-word.

c: Specifies where the shift count is held. 0 means the shift count is held in bits 9, 10, and 11 ("aaa"). 1 means the shift count is held in a data register specified by "aaa."

1 means the shift count is held in a data register specified by "aaa".

00: ASL instruction.

ddd: Specifies the data register to be shifted.

2. Memory: 1110000111aaabbb

Breakdown

1110000111: ASL instruction, memory.

aaa: Addressing mode.

bbb: Addressing register number.

ASR

Definition: arithmetic shift right.

Description: Shifts the bits of the specified location to the right. The last bit shifted out of the least significant position will be copied into both the C and X flags. The most significant bit position will retain its value no matter how many bit positions are shifted. Either a register or a memory location can be shifted.

1. Register shifts can range from 1 to 63 bit positions. The number of positions shifted (the shift count) is specified in one of two ways. A. The shift count can be held in the least significant six bits of a data register. That can specify a shift of from 0 to 63 bits. B. Immediate data can specify the shift number and that can range from 1 to 8 bits. (A value of 000 means a shift of 8 positions.)

2. Memory shifts can only be a single bit position.

Addressing:

1. Register. Uses Register Direct mode for the register to be shifted (the destination).

2. Memory. Uses any of the following modes for the memory location to be shifted:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
2. Memory: 1 word.

Instruction length: 1 word.

Condition code effects:

N Set if the most significant bit of the result is 1; otherwise cleared.

Z Set if the result equals zero; otherwise cleared.

V Unaffected.

C Set by the last bit shifted out of the operand. That bit is copied into the C flag. C is cleared if a shift count of zero is performed.

X Set by the last bit shifted out of the operand. That bit is copied into the X flag. X, unlike C, is unaffected by a shift count of zero.

Object code:

1. Register: 1110aaa0bbc00ddd

Breakdown

1110: ASR instruction.
aaa: Shift count or number of specified register (which it is depends on bit 5, "c," as described below).
0: ASR instruction.
bb: Operand size specification.
00 means byte.
01 means word.
11 means long-word.
c: Specifies where the shift count is held.
0 means the shift count is held n bits 9, 10, and 11 ("aaa").
1 means the shift count is held in a data register specified by "aaa."
00: ASR instruction.
ddd: Specifies the data register to be shifted.

2. Memory: 1110000111aaabbb

Breakdown

1110000111: ASR instruction, memory case.
aaa: Addressing mode.
bbb: Addressing register number.

Bcc

Definition: branch (conditionally).
Description: Bcc tests the state of a particular flag (condition code) in the status register. If the condition (which is specified in the instruction by the programmer) is met, the PC (program counter) is set to a new value. Otherwise, if the condition is not met, execution continues with the next instruction. You can choose from any of the following conditions.

Symbol	Title	Operation
HI	High	~C AND ~Z
LS	Low or same	C OR Z
CC	Carry clear	~C
CS	Carry set	C
NE	Not equal	~Z
EQ	Equal	Z
VC	Overflow clear	~V

Symbol	Title	Operation
VS	Overflow set	V
PL	Plus	~N
MI	Minus	N
GE	Greater or equal	(N and V) OR (~N AND ~V)
LT	Less than	(N AND ~V) OR (~N AND V)
GT	Greater than	(N AND V AND ~Z) OR (~N AND V AND ~Z)
LE	Less or equal	Z OR (N AND ~V) OR (~N AND V)

The Bcc instruction is written with the specified condition symbol replacing the "cc." For example, BNE means "branch if not equal."

Addressing: Program Counter Relative is the only mode used for this instruction. That means the displacement from the present instruction's location can only be from −126 to +129 or from −32766 to +32769 bytes (depending on whether you choose the 8-bit or the 16-bit displacement). The displacement is added to the program counter value (after the program counter value has been incremented by two). The displacement value, either 8 bits or 16 bits, is interpreted as a two's complement integer specifying a distance in bytes.

Operand size: byte or word.

Instruction length: 1 or 2 words. 1 word if the displacement is 8 bits long (with the low-order byte being the displacement); 2 words if the displacement is 16 bits long (with the low-order byte of the first word equal to zero and the second word being the displacement).

Condition code effects: none.

Object code: First word (0110aaaabbbbbbbb)
Second word (displacement)

Breakdown

0110: Bcc instruction

aaaa: Specifies the condition to be tested. (See the table above).

bbbbbbbb: Displacement value. After incrementing the PC by two (because of the current Bcc instruction) add the sign-extended value of the 8-bit displacement to the PC to get the new processing address. If the 8-bit displacement is equal to zero, the displacement value to use is the sign-extended value of the next instruction word.

displacement: As mentioned just above, if the 8-bit displacement value is zero, this word represents the displacement value.

Note: If you try to program a branch to the next word, you have to use the 2 word instruction (with the 16-bit displacement). A value of zero in the 8-bit displacement (which would be necessary for such a branch) tells the CPU to consider the next word as the displacement value.

BCHG

Definition: bit test and change.

Description: BCHG tests a specified bit of a specified operand (found in either data

register or memory location) and indicates the result in the Z flag (which is set if the tested bit is zero and is cleared otherwise). After it is tested, the bit is complemented (made the opposite of its present value).
Addressing: An operand must be addressed and then a particular bit within that operand must be specified.

1. The operand can be specified by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

2. The bit to test can be specified in either of two ways. A. Data Register Direct. The contents of a data register specify the bit to test. Even at this level, there are two possibilities: a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the specifying data register determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least significant 3 bits of the specifying data register determine which of the 8 bits to test. B. Immediate. Immediate data in the word following the instruction word specifies which bit to test. As in section a. above, there are two possibilities here: a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the immediate data determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least significant 3 bits of the immediate data determine which of the 8 bits to test.
Operand size: byte or long-word.
Instruction length: 1 word (if the bit to test is specified by a data register) or 2 words (if the bit to test is specified by immediate data).
Condition code effects:

N Not affected.
Z Set if the bit tested equals zero; otherwise cleared.
V Not affected.
C Not affected.
X Not affected.

Object code:

1. Data Register specification of bit to test (this is called bit number dynamic); 0000aaa101bbbccc

Breakdown

0000: BCHG instruction.
aaa: Specifies the data register that holds the bit to test.
101: BCHG instruction.

bbb: Addressing mode for the operand holding the bit to test.
ccc: Addressing register number for the operand holding the bit to test.

2. Immediate specification of bit to test (called bit number static):
First word (0000100001aaabbb)
Second word (bitnumber)

Breakdown

0000100001: BCHG instruction.
aaa: Addressing mode for the operand holding the bit to test.
bbb: Addressing register number for the operand holding the bit to test.
bitnumber: This word specifies the number of the bit to test. The upper 8 bits are all zeroes, the lower 8 bits are the bit number. If the bit to test is in a data register, the least significant 6 bits determine its bit position; if the bit to test is in a memory location, the least significant 3 bits determine its bit position.

BCLR

Definition: bit test and clear.
Description: BCLR tests a specified bit in a specified location (either data register or memory location) and indicates the result in the Z flag (which is set if the tested bit is zero and is cleared otherwise). After it is tested, the bit is cleared (set equal to zero). BCLR is very similar to BCHG. The only difference is that this instruction clears the tested bit instead of complementing it.
Addressing: An operand must be addressed and then a particular bit within that operand must be specified.

1. The operand can be specified by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

2. The bit to test can be specified in either of two ways. A. Data Register Direct. The contents of a data register specify the bit to test. Even at this level, there are two possibilities: a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the specifying data register determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least significant 3 bits of the specifying data register determine which of the 8 bits to test. B. Immediate. Immediate data in the word following the instruction word specifies which bit to test. As in section a. above, there are two possibilities here:

a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the immediate data determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least significant 3 bits of the immediate data determine which of the 8 bits to test.
Operand size: byte or long-word.
Instruction length: 1 word (if the bit to test is specified by a data register) or 2 words (if the bit to test is specified by immediate data).
Condition code effects:
N Not affected.
Z Set if the bit tested equals zero; otherwise cleared.
V Not affected.
C Not affected.
X Not affected.

Object code:
1. Data register specification of bit to test (this is called bit number dynamic): 0000aaa110bbbccc

Breakdown

0000: BCLR instruction.
aaa: Specifies the data register that holds the bit to test.
110: BCLR instruction.
bbb: Addressing mode for the operand holding the bit to test.
ccc: Addressing register number for the operand holding the bit to test.

2. Immediate specification of bit to test (called bit number static): First word (0000100010aaabbb)
Second word (bitnumber)

Breakdown

0000100010: BCLR instruction.
aaa: Addressing mode for the operand holding the bit to test.
bbb: Addressing register number for the operand holding the bit to test.
bitnumber: This word specifies the number of the bit to test. The upper 8 bits are all zeroes, the lower 8 bits are the bit number. If the bit to test is in a data register, the least significant 6 bits determine its bit position if the bit to test is in a memory location, the least significant 3 bits determine its bit position.

BRA

Definition: branch (unconditionally).
Description: BRA forces the CPU to continue processing at a new point in the program. It accomplishes this by putting a new value into the program counter register (PC). The specified address for new processing is found by adding the current PC value (after it has been incremented by two), and a displacement value. The

displacement (which is sign-extended) is a two's complement number found in the lease significant byte of the instruction word or (when that byte is equal to zero) in the next word in the instruction sequence.
Addressing: Only the Immediate mode can be used to produce the new PC value.
Operand size: byte or word.
Instruction length: 1 or 2 words (1 word if the displacement is 8 bits; 2 words if the displacement is 16 bits).
Condition code effects: none.
Object code: first word (01100000aaaaaaaa)
second word (displacement)

Breakdown

01100000: BRA instruction.
aaaaaaaa: Byte displacement value. If equal to zero, the next instruction word is the displacement value.
displacement: The immediate displacement value is found here if it is 16 bits long.

Note: As with Bcc, you cannot branch to the next instruction with a short (8-bit) displacement. Such a branch would require a zero value for the 8 bits, which would tell the CPU that the next word was the 16-bit displacement.

BSET

Definition: bit test and set.
Description: BSET tests a specified bit in a specified operand (either data register or memory location) and indicates the result in the Z flag (which is set if the tested bit is zero and is cleared otherwise). After it is tested, the bit is set to 1.
Addressing: An operand must be addressed and then a particular bit within that operand must be specified.

1. The operand can be specified by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

2. The bit to test can be specified in either of two ways. A. Data Register Direct. The contents of a data register specify the bit to test. Even at this level, there are two possibilities: a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the specifying data register determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least significant 3 bits of the specifying data register determine which of the 8 bits to test. B. Immediate. Immediate data in the word following the instruction word

specifies which bit to test. As in section A above, there are two possibilities here: a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the immediate data determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least significant 3 bits of the immediate data determine which of the 8 bits to test.
Operand size: byte or long-word.
Instruction length: 1 word (if the bit to test is specified by a data register) or 2 words (if the bit to test is specified by immediate data).
Condition code effects:
N Not affected.
Z Set if the bit tested equals zero; otherwise cleared.
V Not affected.
C Not affected.
X Not affected.

Object code:

1. Data register specification of bit to test (this is called bit number dynamic): 0000aaa111bbbccc

Breakdown

0000: BSET instruction.
aaa: Specifies the data register that holds the bit to test.
111: BSET instruction.
bbb: Addressing mode for the operand holding the bit to test.
ccc: Addressing register number for the operand holding the bit to test.

2. Immediate specification of bit to test (called bit number static): First word (0000100011aaabbb)

Second word (bitnumber)

Breakdown

0000100011: BSET instruction.
aaa: Addressing mode for the operand holding the bit to test.
bitnumber: This word specifies the number of the bit to test. The upper 8 bits are all zeros, the lower 8 bits are the bit number. If the bit to test is in a data register, the least significant 6 bits determine its bit position; if the bit to test is in a memory location, the least significant 3 bits determine its bit position.

BSR

Definition: branch to subroutine.
Description: BSR forces the CPU to continue processing at a new point in the program. It accomplishes this by putting a new value into the program counter register (PC). This is called a branch to a subroutine, because unlike BRA (which is just a ranch) the old PC value is saved (on the system stack). That means you can return to the same point in the program later, for example, after executing a subroutine.

The specified address for new processing is found by adding the current PC value (after it has been incremented by two), and a displacement value. The displacement (which is sign-extended) is a two's complement number found in the least significant byte of the instruction word or (when that byte is equal to zero) in the next word int he instruction sequence.
Addressing: Immediate.
Operand size: byte or word.
Instruction length: 1 or 2 words (1 word if the displacement is 8 bits; 2 words if the displacement is 16 bits).
Condition code effects: none.
Object code: First word (01100001aaaaaaaa)
Second word (displacement)

Breakdown

01100001: BSR instruction.
aaaaaaaa: Byte displacement value. If equal to zero, the next instruction word is the displacement value.
displacement: The immediate displacement value is found here if it is 16 bits long.

Note: As with Bcc, you cannot branch to the next instruction with a short (8-bit) displacement. Such a branch would require a zero value for the 8 bits, which would tell the CPU that the next word was the 16-bit displacement.

BTST

Definition: bit test.
Description: BTST test a specified bit in a specified location (either data register or memory location) and indicates the result in the Z flag (which is set if the tested bit is zero and is cleared otherwise).
Addressing: An operand must be addressed and then a particular bit within that operand must be specified.

1. The operand can be specified by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

2. The bit to test can be specified in either of two ways. A. Data Register Direct. The contents of a data register specify the bit to test. Even at this level, there are two possibilities: a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the specifying data register determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least

significant 3 bits of the specifying data register determine which of the 8 bits to test. B. Immediate. Immediate data in the word following the instruction word specifies which bit to test. As in section A. above, there are two possibilities here: a. Any of the 32 bits in a data register may be tested. The least significant 6 bits of the immediate data determine which of the 32 bits to test. b. Any of the 8 bits of a memory location can be tested. The least significant 3 bits of the immediate data determine which of the 8 bits to test.
Operand size: byte of long-word.
Instruction length: 1 word (if the bit to test is specified by a data register) or 2 words (if the bit to test is specified by immediate data).
Condition code effects:
N Not affected.
Z Set if the bit tested equals zero; otherwise cleared.
V Not affected.
C Not affected.
X Not affected.

Object code:

1. Data Register specification of bit to test (this is called bit number dynamic): 0000aaa100bbbccc

Breakdown

0000: BTST instruction.
aaa: Specifies the data register that holds the bit to test.
100: BTST instruction.
bbb: Addressing mode for the operand holding the bit to test.
ccc: Addressing register number for the operand holding the bit to test.

2. Immediate specification of bit to test (called bit number static): First word (0000100000aaabbb)

Second word (bitnumber)

Breakdown

0000100000: BTST instruction.
aaa: Addressing mode for the operand holding the bit to test.
bbb: Addressing register number for the operand holding the bit to test.
bitnumber: This word specifies the number of the bit to test. The upper 8 bits are all zeros, the lower 8 bits are the bit number. If the bit to test is in a data register, the least significant 6 bits determine its bit position; if the bit to test is in a memory location, the least significant 3 bits determine its bit position.

CHK

Definition: check register against boundaries.
Description: CHK compares the contents of a data register to a specifies source value (either in a data register or in memory) and to zero. If the data register con-

tents are less than zero, or greater than the source contents, CHK generates a TRAP and begins exception processing (see both the TRAP instruction and Chapter 7, Exceptions). Exception processing will use the CHK vector stored in the exception vector table (which begins at 018H in memory). Only the low-order word of the specified source data register is compared. The comparison is made using two's complement integers.

Addressing: The destination (the data register to be checked) is only addressed by Data Register Direct mode. The source (the operand to check against) can be addressed by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: word.
Instruction length: 1 word.
Condtion code effects:

N Set if the destination contents are negative; cleared if they are greater than the upper bound (the source contents).
Z Affected but undefined (value could be 1 or 0).
V Affected but undefined.
C Affected but undefined.
X Not affected.

Object code: 0100aaa110bbbccc

Breakdown

0100: CHK instruction.
aaa: Number of the source data register.
110: CHK instruction (cont.).
bbb: Destination addressing mode.
ccc: Destination addressing register number.

Note: CHK can be useful for ensuring that array references don't run beyond the array's dimensions.

CLR

Definition: clear an operand.

Description: Clears a specified location (fills the location with zeros).
Addressing: These modes can specify the location (destination) to clear:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:
Z Always set.
N Always cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 01000010aabbbccc

Breakdown

01000010: CLR instruction.
aa: Specifies operand size.
00 means byte.
01 means word.
10 means long-word.
bbb: Destination addressing mode.
ccc: Destination addressing register number.

CMP

Definition: compare.
Description: CMP subtracts the contents of a specified location (the source) from the contents of a data register (the destination). The flags in the status register are set according to the result. Neither the source contents nor the data register contents are changed.
Addressing: The destination is only addressed by Data Register Direct mode. The source can be addressed by any addressing mode:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect

Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

N Set if the result is negative; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.
C Because this is a subtraction operation, C operates as a borrow flag. This is, if a borrow occurs, this flag is set. Otherwise, it is cleared.
X Not affected.

Object code: 1011aaa0bbcccddd

Breakdown

1011: CMP instruction.
aaa: Specifies the destination data register.
0: CMP instruction (cont.).
bb: Specifies the operand size.
00 means byte.
01 means word.
10 means long-word.
ccc: Source addressing mode.
ddd: Source addressing register number.

Note: Remember that the carry flag (C) represents a borrow for this instruction and not a carry.

CMPA

Definition: compare address.
Description: CMPA subtracts the contents of a specified location (the source) from the contents of an address register (the destination). The condition codes (flags) in the status register are set according to the result. Neither the source nor the destination contents are changed. CMPA is a special case of the CMP instruction.
Addressing: The destination is only addressed by Address Register Direct mode. The source can be addressed by any addressing mode:

Data Register Direct
Address Register Direct
Address Register Indirect

Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: word or long-word.
Instruction length: 1 word.
Condition code effects:

N Set if the result if negative; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.
C Because this is subtraction operation, C operates as a borrow flag. That is, if a borrow occurs, this flag is set. Otherwise, it is cleared.
X Not affected.

Object code: 1011aaab11cccddd

Breakdown

1011: CMPA instruction.
aaa: Specifies the destination address register.
b: Specifies the operand size.
0 means word.
1 means long-word.
11: CMPA instruction (cont.).
ccc: Source addressing mode.
ddd: Source addressing register number.

Note: Remember that the carry flag (C) represents a borrow for this instruction and not a carry. This instruction is nearly identical to CMP except that, because it uses an address register, it cannot work with bytes (only with words or long-words).

CMPI

Definition: compare immediate.
Description: CMPI subtracts the immediate data (found in the next program instruction word) from the contents of the specified destination and then sets the flags according to the result. The destination contents are not changed.
Addressing: The source (the data to subtract) is addressed by the Immediate mode. The destination can be addressed by any of these modes:

Data Register Direct

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 2 or 3 words (2 words if the immediate data is 8 or 16 bits long, 3 words if it is 32 bits long).
Condition code effects:

N Set if the result is negative; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.
C Because this is a subtraction operation, C operates as a borrow flag. That is, if a borrow occurs, this flag is set. Otherwise, it is cleared.
X Not affected.

Object code: First word (00001100aabbbccc)
Second word (immediate)
Third word (immediate)

Breakdown

00001100: CMPI instruction.
aa: Specifies the operand size.
00 means byte.
01 means word.
10 means long-word.
bbb: Destination addressing mode.
ccc: Destination addressing register number.
immediate: If the immediate data is 8 bits (as specified by aa above) it is held in the low-order byte of the second instruction word. If the immediate data is 16 bits, it is held in the full second instruction word. And if the immediate data is a full 32 bits long, it is held in both the second and third instruction words.

Note: Remember that the carry flag (C) represents a borrow for this instruction and not a carry.

CMPM

Definition: compare memory.
Description: CMPM subtracts the contents of the specified source memory location from that of the specified destination memory location. The status flags are set according to the result. Neither the contents of the destination nor of the source are changed by this instruction.

Addressing: Both source and destination are always addressed by Postincrement Register Indirect mode. After the two specified address registers have been used to get the two operands, and the operands have been compared, each of the specified address registers is incremented (by 1 if a byte was specified, by 2 if a word was specified, and by 4 if a long-word was specified).
Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

N Set if the result is negative; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.
C Because this is a subtraction operation, C operates as a borrow flag. That is, if a borrow occurs, this flag is set. Otherwise, it is cleared.
X Not affected.

Object code: 1011aaa1bb001ccc

Breakdown

1011: CMPM instruction.
aaa: Specifies the address register that holds the destination address.
1: CMPM instruction (cont.).
bb: Specifies the operand size.
00 means byte.
01 means word.
10 means long-word.
001: CMPM instruction (cont.).
ccc: Specifies the address register that holds the source address.

Note: Remember that the carry flag (C) represents a borrow for this instruction and not a carry.

DBcc

Definition: test condition, decrement and branch.
Description: DBcc is a simple looping instruction. It first tests whether the specified condition (the choices are listed below) has been met. If it has, the processing goes on to the next instruction in the program. If the condition has not been met, the low-order word the specified data register (often called the counter) is decremented by 1. If the contents of that data register are then equal to −1, the processing goes on to the next program instruction. If the contents are not equal to −1, the program branches to a new location. That location is found by adding a 16-bit (sign-extended) displacement to the PC. You can choose any of the following conditions:

Symbol	Title	Operation
T	True	1
F	False	0

Symbol	Title	Operation
HI	High	~C AND ~Z
LS	Less or same	C OR Z
CC	Carry clear	~C
CS	Carry set	C
NE	Not equal	~Z
EQ	Equal	Z
VC	Overflow clear	~V
VS	Overflow set	V
PL	Plus	~N
MI	Minus	N
GE	Greater or equal	(N AND V) OR (~N AND ~V)
LT	Less than	(N AND ~V) OR (~N AND V)
GT	Greater than	(N AND V AND ~Z) OR (~N AND V AND ~Z)
LE	Less or equal	Z or (N AND ~V) OR (~N AND V)

The instruction is written with the condition symbol replacing the "cc." For example, DBNE means "Test, decrement, and branch until not equal."
Addressing: The counter data register is only reached by Data Register Direct mode. The new program execution address is found by Program Counter Relative with Displacement mode.
Operand size: word.
Instruction length: 2 words
Condition code effects: none.
Object code: First word (0101aaaa11001bbb)
Second word (displacement)

Breakdown

0101: DBcc instruction.
aaaa: Specifies the condition to be tested.
11001: DBcc instruction (cont.).
bbb: Specifies the data register to be decremented.
displacement: This is the 16-bit value added to the PC to get the new execution address (the displacement value is counted as a two's complement number of bytes).

DIVS

Definition: signed divide.
Description: DIVS divides (using two's complement arithmetic) the long-word in the destination data register by the word source operand. It then stores the result in the destination data register (with the 16-bit quotient in the lower word and the 16-bit remainder in the upper word). The sign of the remainder is the same as that of the dividend (unless the remainder equals zero). There are two special circumstances involved in using this division instruction:

1. Division by zero is not possible and causes a TRAP. Exception process-

ing begins automatically. The vector is 014H (also known as vector #5). See the TRAP instruction description for more details.

2. An overflow may be detected before the instruction is complete. The flag will be set and the operands won't be affected.

Addressing: The destination can only be addressed by Data Register Direct mode. The source can be addressed by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: word.
Instruction length: 1 word.
Condition code effects:

N Set if the result is negative; otherwise cleared (except that it is undefined if there is an overflow).
Z Set if the result is zero; cleared otherwise (except that it is undefined if there is an overflow).
V Set if there is an overflow. This will happen if the source contents are larger than the destination contents. The V flag will be set before division is performed, and the operands will not be changed. In other words, the division will not be carried out.
C Always cleared.
X Not affected.

Object code: 1000aaa111bbbccc

Breakdown

1000: DIVS instruction.
aaa: Specifies the destination data register.
111: DIVS instruction (cont.).
bbb: Source addressing mode.
ccc: Source addressing register number.

DIVU

Definition: Unsigned divide.
Description: DIVU divides (using unsigned binary arithmetic) the long-word in the destination data register by the word source operand. It then stores the result

in the destination data register (with the 16-bit quotient in the lower word and the 16-bit remainder in the upper word). There are two special circumstances involved in using this division instruction:

1. Division by zero is not possible and causes a TRAP. Exception processing begins automatically. The vector is 014H (also known as vector #5). See the TRAP instruction description for more details.

2. An overflow may be detected before the instruction is complete. The overflow flag will be set, the instruction won't be executed, and the operands won't be affected.

Addressing: The destination can only be addressed by Data Register Direct mode. The source can be addressed by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: word.
Instruction length: 1 word.
Condition code effects:

N Set if the most significant bit of the quotient is set; cleared otherwise (except that it is undefined if there is an overflow).
Z Set if the result is zero; cleared otherwise (except that it is undefined if there is an overflow).
V Set if there is an overflow. This will happen if the source contents are larger than the destination contents. The V flag will be set before division is performed, and the operands will not be changed. In other words, the division will not be carried out.
C Always cleared.
X Not affected.

Object code: 1000aaa011bbbccc

Breakdown

1000: DIVU instruction.
aaa: Specifies the destination data register.
011: DIVU instruction (cont.).
bbb: Source addressing mode.
ccc: Source addressing register number.

Note: If the quotient is larger than 16 unsigned bits, an overflow will occur.

EOR

Definition: exclusive logical OR.
Description: EOR performs a bit-by-bit exclusive-OR operation on the contents of a specified data register and a specified destination operand. The result is stored in the destination data register. Exclusive-OR sets each bit where either one or the other, but not both, of the bit positions of the source and destination are set. All other bit positions of the result are cleared.
Addressing modes: The source can only be addressed by Data Register Direct mode. The destination can be addressed by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:
N Set if the most significant bit of the result is set; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 1011aaa1bbcccddd

Breakdown

1011: EOR instruction.
aaa: Specifies the source data register.
1: EOR instruction (cont.).
bb: Specifies the operand size.
00 means byte.
01 means word.
10 means long-word.
ccc: Destination addressing mode.
ddd: Destination addressing register number.

EORI

Definition: exclusive-OR immediate.
Description: EORI performs a bit-by-bit exclusive-OR operation on the immediate

data (found in the next program instruction words) and the contents of a specified destination operand. The result is stored in the destination. Exclusive-OR sets each bit where either one or the other, but not both, of the bit positions of the source and destination are set. All other bit positions of the result are cleared.
Addressing modes: The source (data to be compared) is reached only by the Immediate mode. The destination can be reached by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 2 or 3 words (2 if the immediate data is 8 bits or 16 bits long, 3 words if the immediate data is 32 bits long).
Condition code effects:

N Set if the most significant bit of the result is set; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: First word (00001010aabbbccc)
Second word (immediate)
Third word (immediate)

Breakdown

00001010: EORI instruction
aa: Specifies the operand size.
00 means byte.
01 means word.
10 means long-word.
bbb: Destination addressing mode.
ccc: Destination addressing register number.
immediate: If the immediate data is specified as 8 bits (by aa above), the lower byte of the second word contains that data. If the immediate data is 16 bits, the full second word is that data. If the immediate data is a full 32 bits, both the second and third instruction words contain the data.

EORI to CCR

Definition: exclusive-OR immediate to condition codes register.
Description: This is a special form of the EORI instruction. EOR performs a bit-

by-bit exclusive-OR operation on the contents of the condition codes register (CCR: the low-order byte of the status register) and the immediate value contained in the low-order byte of the next program word. The result is stored in the CCR. Exclusive-OR sets each bit where either one or the other, but not both, of the bit positions of the source and destination are set. All other bit positions of the result are cleared.

Addressing: The source is addressed by Immediate mode. The destination is the condition code register.

Operand size: byte.

Instruction length: 2 words. (The first is the instruction word and the second contains the data as the low-order byte. The high order byte of the second word holds all zeros.)

Condition code effects: All of these are changed directly by the result of the operation (which is stored in this register).

N Changed if bit 3 of the immediate operand is one; otherwise unchanged.
Z Changed if bit 2 of the immediate operand is one; otherwise unchanged.
V Changed if bit 1 of the immediate operand is one; otherwise unchanged.
C Changed if bit 0 of the immediate operand is one; otherwise unchanged.
X Changed if bit 4 of the immediate operand is one; otherwise unchanged.

Object code: First word (0000101000111100)
Second word (immediate)

Breakdown

0000101000111100: EORI to CCR instruction.

immediate data: The high-order byte is filled with zeros; the low-order byte holds the immediate data.

EORI to SR

Definition: exclusive-OR immediate to status register (privileged).

Description: This is a special form of EORI and is a privileged instruction: it can only be performed if the CPU is in supervisor mode. EOR performs a bit-by-bit exclusive-OR operation on the contents of the status register and the immediate value contained in the next program word. The result is stored in the CCR. Exclusive-OR sets each bit where either one or the other, but not both, of the bit positions of the source and destination are set. All other bit positions of the result are cleared.

Addressing: The source is addressed by Immediate. The destination is the status register.

Operand size: word.

Instruction length: 2 words. (The first is the instruction word and the second contains the data as the low-order byte. The high order byte of the second word holds all zeros.)

Condition code effects: All of these are changed directly by the result of the operation (which is stored in this register).

N Changed if bit 3 of the immediate operand is one; otherwise unchanged.

Z Changed if bit 2 of the immediate operand is one; otherwise unchanged.
V Changed if bit 1 of the immediate operand is one; otherwise unchanged.
C Changed if bit 0 of the immediate operand is one; otherwise unchanged.
X Changed if bit 4 of the immediate operand is one; otherwise unchanged.

Object code: First word (0000101001111100)
Second word (immediate)

Breakdown

0000101001111100: EORI to SR instruction.
immediate data: The second instruction word is the immediate data.

Note: Remember that when the entire status register is used as a destination for EOR (of the other logical instructions), EORI becomes a privileged instruction, so it can only be used from the Supervisor mode.

EXG

Definition: exchange registers.
Description: Exchanges the contents of one 32-bit register with that of another 32-bit register. This is the equivalent of several Move operations. The entire value in one register is replaced with that of the second register and the second register is filled with the previous contents of the first register. The exchange can be made between Data Register and Data Register, or Data Register and Address Register, or Address Register and Address Register.
Addressing Modes: The source and destination are addressed by either Data Register Direct mode or Address Register Direct mode.
Operand size: long-word.
Instruction length: 1 word.
Conditon code effects: none.
Object code: 1100aaa1bbbbbccc

Breakdown

1100: EXG instruction.
aaa: Specifies a register. If the exchange is between a data and an address register, ccc specifies the data register.
1: EXG instruction (cont.).
bbbbb: Specifies the mode of the operation.
01000 means both source and destination are data registers.
01001 means both are address registers.
10001 means a data and an address register are to be exchanged.
ccc: Specifies a register. If the exchange is between a data and an address register, ccc specifies the address register.

EXT

Definition: sign extend.
Description: EXT is used only with a data register. It extends the sign-bit of a

datum to the most significant bit of the register space. In other words, for a byte, the sign-bit (bit 7) is extended through bit 8 to bit 15 of the register. For a word, the sign-bit (bit 15) is extended through bit 16 to bit 32 of the register. This instruction helps keep arithmetic correct when you are working with bytes or words—data that doesn't fill the data register. If the sign bit weren't extended, the sign of the number wouldn't be read. Since the register is 32 bits long, the bit 32 is assumed to be the sign bit. EXT instruction extends the true sign bit to the usable position.
Addressing mode: The data register to be manipulated is found by Data Register Direct mode.
Operand size: word or long-word.
Instruction length: 1 word.
Condition code effects:
N Set if the result of the operation is negative; otherwise cleared.
Z Set if the result of the operation is zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 01001000aa000bbb

Breakdown

01001000: EXT instruction.
aa: Size specification.
10 means to extend a low-order byte to a word.
11 means to extend a low-order word to a long-word.
000: EXT instruction (cont.).
bbb: Specifies the data register.

ILLEGAL

Definition: illegal instruction.
Description: ILLEGAL forces the illegal instruction exception to start processing. (This instruction will be retained in the newer 68000 family CPUs.) Other illegal instruction object codes may be used in other CPUs for new instructions. The PC value and the SR value will be saved on the stack and then the Illegal Instruction Vector will be put into the PC.
Addressing: none. As noted above, the PC (program counter) is loaded with the Illegal Instruction Vector.
Operand size: none.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100101011111100

Breakdown

0100101011111100: ILLEGAL instruction.

JMP

Definition: jump.
Description: JMP forces an unconditional jump to a different place in the program. In other words, the program counter value is replace with a new value (specified by the instruction). Processing will continue with the instruction found at that new address. Because memory is organized in bytes, the new program counter value will be found in the specified address and the specified address + 2.
Addressing: The new value to put into the PC can be found by any of these addressing modes:

Address Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index

Operand size: no operand.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100111011aaabbb

Breakdown

0100111011: JMP instruction.
aaa: Addressing mode.
bbb: Addressing register number.

JSR

Definition: jump to subroutine.
Description: JSR, like JMP, unconditionally forces processing to a different part of the program. The PC (program counter) value will be replaced by the value specified by the instruction. Unlike JMP, JSR saves the old PC value (the long-word address of the instruction following JSR) on the system stack. That is done so that—once the subroutine has been completed—processing of the main program can resume where it left off. An RTS (Return from Subroutine) instruction will handle the return duties as long as the stack values (both stack pointer and stack) haven't been changed while processing the subroutine.
Addressing: The new value to put into the PC can be found by any of these addressing modes:

Address Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Program Counter Relative with Displacement
Program Counter Relative with Index

Operand size: no operand.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100111010aaabbb

Breakdown

0100111010: JSR instruction.
aaa: Addressing mode.
bbb: Addressing register number.

LEA

Definition: load effective address.
Description: LEA forms an effective address using one of the addressing modes listed below, and loads the address into an address register. This instruction is unusual because it loads the address—not the contents found at that address—into the destination register.
Addressing: The destination—the register to load the address into—is only reached by Address Register Direct mode. The source can be reached by any of these addressing modes:

Address Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index

Operand size: long-word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100aaa111bbbccc

Breakdown

0100: LEA Instruction.
aaa: Destination address register number.
111: LEA instruction (cont.).
bbb: Source addressing mode.
ccc: Source addressing register number.

LINK

Definition: link and allocate.
Description: LINK pushes the contents of a specified address register onto the

system stack. That address register is called a frame pointer. The stack pointer is then loaded into that address register. Then a sign-extended two's complement displacement value—from the next word of the instruction—is added to the stack pointer. All of this manipulating lets you use a certain space of the stack—called frame—for the variables in a subroutine. The displacement moves the active stack pointer to the new frame. An UNLK instruction reverses this process.
Addressing: immediate.
Operand size: no operand.
Instruction length: 2 words.
Condition code effects: none.
Object code: First word (0100111001010aaa)
Second word (displacement)

Breakdown
0100111001010: LINK instruction.
aaa: Address register number.
displacement: This 16-bit value is interpreted as a two's complement number and is sign-extended to a full 32-bits.

Note: This is a tricky instruction for beginners to understand and use. For instance, if you specify a positive displacement, the new frame contents may overlay—and thus obliterate—previous stack contents that you didn't want to lose.

LSL

Definition: logical shift left.
Description: LSL shifts the contents of the specified location to the left. The last bit shifted out of the most significant bit position will be copied into both the C and X flags. The least-significant bit position will be filled with a 0. LSL performs just as ASL does. The shift can be made on a register or on a memory location.

1. Register shifts can range from 1 to 63 bit positions. The number of positions shifted is specified in one of two ways. A. The shift count can be held in the least significant six bits of a data register. That can specify a shift of from 0 to 63 bits. B. Immediate data can specify the shift number and that can range from 1 to 8 bits. (A value of 000 means a shift of 8 positions.)
2. Memory shifts can only be a single bit position.

Addressing:

1. Register. Uses Register Direct mode for the register to be shifted (the destination).
2. Memory. Uses any of the following modes for the memory location to be shifted:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index

Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
1. Memory: word.

Instruction length: 1 word.

Condition code effects:

N Set if the most significant bit of the result is 1; otherwise cleared.

Z Set if the result equals zero; otherwise cleared.

V Set if the most significant bit changes at any point during the shift; otherwise cleared.

C Set by the last bit shifted out of the operand. That bit is copied into the C flag. C is cleared if a shift count of zero is performed.

X Set by the last bit shifted out of the operand. That bit is copied into the X flag. X, unlike C, is unaffected by a shift count of zero.

Object code:

1. Register: 1110aaa1bbc01ddd

Breakdown

1110: LSL instruction.

aaa: Shift count or number of specified register (which it is depends on bit 5, "c", as described below).

1: LSL instruction.

bb: Operand size specification.
00 means byte.
01 means word.
11 means long-word.

c: Specifies where the shift count is held.
0 means the shift count is held in bits 9, 10, and 11 ("aaa").
1 means the shift count is held in a data register specified by "aaa".

01: LSL instruction.

ddd: Specifies the data register to be shifted.

2. Memory: 1110001111aaabbb

Breakdown

1110001111: LSL instruction, memory.

aaa: Addressing mode.

bbb: Addressing register number.

LSR

Definition: logical shift right.

Description: LSR logically shifts the contents of the specified operand to the right. The last bit shifted out of the least significant bit position will be copied into both

the C and X flags. The most significant bit position will be filled with a zero, no matter how many bit positions are shifted. LSR is the same as the ASR instruction except the LSR puts a zero in the most significant position and ASR repeats the previous value in that position. The shift can be made on a register or on a memory location.

1. Register shifts can range from 1 to 63 bit positions. The number of positions shifted is specified in one of two ways. A. The shift count can be held in the least significant six bits of a data register. That can specify a shift of from 0 to 63 bits. B. Immediate data can specify the shift number and that can range from 1 to 8 bits. (A value of 000 means a shift of 8 positions.)

2. Memory shifts can only be a single bit position.

Addressing:

1. Register. Uses Register Direct mode for the register to be shifted (the destination).

2. Memory. Uses any of the following modes for the memory location to be shifted:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
2. Memory: word.

Instruction length: 1 word.

Condition code effects:

N Always cleared.
Z Set if the result equals zero; otherwise cleared.
V Unaffected.
C Set by the last bit shifted out of the operand. That bit is copied into the C flag. C is cleared if a shift count of zero is performed.
X Set by the last bit shifted out of the operand. That bit is copied into the X flag. X, unlike C, is unaffected by a shift count of zero.

Object code:

1. Register: 1110aaa0bbc01ddd

Breakdown

1110: LSR instruction.
aaa: Shift count or number of specified register (which it is depends on bit 5, "c", as described below).
0: LSR instruction (cont.).

bb: Operand size specification.
00 means byte.
01 means word.
11 means long-word.
c: Specifies where the shift count is held.
0 means the shift count is held in bits 9, 10, and 11 ("aaa").
1 means the shift count is held in a data register specified by "aaa".
01: LSR instruction.
ddd: Specifies the data register to be shifted.

2. Memory: 1110001011aaabbb

Breakdown

1110001011: LSR instruction, memory.
aaa: Addressing mode.
bbb: Addressing register number.

MOVE

Definition: move data (from source to destination).
Description: A very flexible instruction that can move data from register to register, register to memory, or memory to register. Other microprocessors often use a variety of instructions—such as Store, Load, Input—in the place of MOVE.
Addressing: MOVE is flexible because the programmer only needs to learn a single instruction, and can use most of the addressing modes available to all other instructions. Any of the addressing modes can be used to reach the source:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

These addressing modes can be used to reach the destination:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect

Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:
N Set according to the moved data value.
Z Set according to the moved data value.
V Always cleared.
C Always cleared.
X Not affected.

Object Code: 00aabbbcccdddeee

Breakdown

00: MOVE instruction.
aa: Size specification.
01 means byte.
11 means word.
10 means long-word.
bbb: Destination addressing register number.
ccc: Destination addressing mode.
ddd: Source addressing mode.
eee: Source addressing register number.

Note: MOVEQ is a quicker version of MOVE but cannot use all the addressing modes that MOVE can. If you want to move a value directly to a register, use MOVEA.

MOVE to CCR

Definition: move to condition code register.
Description: MOVE to CCR is a special case of the MOVE instruction. It moves the contents from the specified source to the low byte (the condition code register) of the status register.
Addressing: The destination is always the CCR. The source can be reached by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short

Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: word.
Instruction length: 1 word.
Condition code effects: All of the condition codes (flags) are changed directly by the result of the operation because that result is stored in this register.
N Set to the same value as bit 3 of the source contents.
Z Set to the same value as bit 2 of the source contents.
V Set to the same value as bit 1 of the source contents.
C Set to the same value as bit 0 of the source contents.
X Set to the same value as bit 4 of the source contents.

Object Code: 0100011011aaabbb

Breakdown
0100001011: MOVE to CCR instruction.
aaa: Source addressing mode.
bbb: Source addressing register number.

MOVE from SR

Definition: move from the status register.
Description: MOVE from SR is a special case of the MOVE instruction. It moves the contents of the status register to the specified destination.
Addressing: The source is always the status register. The destination can be reached by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: word.
Instruction length: 1 word.
Condition code effects: none.
Object Code: 0100000011aaabbb

Breakdown
0100000011: MOVE from SR instruction.
aaa: Destination addressing mode.
bbb: Destination addressing register number.

MOVE to SR

Definition: move to the status register (privileged).
Description: MOVE to SR is a special case of the MOVE instruction and is a privileged instruction. It moves the contents of the specified destination to the status register.
Address: The destination is always the status register. The source can be reached by almost any addressing mode (except by Address register direct):

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: word.
Instruction length: 1 word.
Condition code effects: All of these are changed directly by the result of the operation (which is stored in this register).

N Set to the same value as bit 3 of the source contents.
Z Set to the same value as bit 2 of the source contents.
V Set to the same value as bit 1 of the source contents.
C Set to the same value as bit 0 of the source contents.
X Set to the same value as bit 4 of the source contents.

Object Code: 0100011011aaabbb

Breakdown

0100011011: MOVE to SR instruction.
aaa: Source addressing mode.
bbb: Source addressing register number.

MOVE USP

Definition: move user stack pointer.
Description: Moves the contents of the User stack pointer (also known as address register 7 while the CPU is in User mode) to or from a specified address register. This is a privileged instruction and can only be executed from Supervisor mode.
Addressing: The address register is always specified by Address Register Direct mode, whether data is transferred from the address register to the USP or from the USP to the address register (in other words, whether the address register is source or destination).

Operand size: long-word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 010011100110abbb

Breakdown

010011100110: MOVE USP instruction
a: Direction of the transfer.
0 means to move data from an address register to the stack pointer.
1 means to move data from the stack pointer to the address register.
bbb: Specifies the address register.

Note: MOVE USP is a privileged instruction and so can only be executed while the CPU is in Supervisor mode. While in that mode, the User stack pointer contents are not visible because any reference to address register 7 will bring up the value of the System stack pointer. MOVE USP, therefore, makes it possible for a Supervisor mode program to see what is in the User stack pointer.

MOVEA

Definition: move address.
Description: MOVEA is a special case of the MOVE instruction. It moves the contents of a specified source to a specified address register.
Addressing: The destination is always reached by Address Register Direct mode. The source can be specified by any addressing mode:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand size: word or long-word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 00aabbb001cccddd

Breakdown

00: MOVEA instruction.
aa: Specifies the operand size.

11 means a word (If a word is used, it is sign-extended before it is moved).
10 means a long-word.
bbb: Destination address register.
001: MOVEA instruction (cont.).
ccc: Source addressing mode.
ddd: Source addressing register number.

MOVEM

Definition: move multiple registers.
Description: MOVEM moves the contents of multiple registers to or from consecutive memory locations. The registers to move are specified by setting the appropriate bits in a mask that is the second word of the instruction (the mask is detailed below). Either words or long-words can be transferred. If words are selected, the low-order word of a register is transferred. Also, word values are sign-extended to a full 32 bits.
Addressing modes: There is one set of permissible addressing modes for register-to-memory transfers and another for memory-to-registers transfers.

1. Move Multiple Registers from Memory. The source—the beginning of the consecutive locations in memory whose contents will be copied to the registers—can be addressed by:

Address register indirect
Postincrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index

The registers that accept that data are specified by the mask that is the second word of the instruction (this is explained in the Object code breakdown below).

2. Move Multiple Registers to Memory. The destination—the beginning of the consecutive locations in memory that the register contents will be copied to—can be addressed by:

Address Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index

The registers are specified by the mask that is the second word of the instruc-

tion (this is explained in the Object code breakdown below).

Addressing is not simple with MOVEM. There are also three different interpretations of the consecutive memory addressing, which depend on the addressing mode.

1. Postincrement mode (which can only be used for memory-to-register movement) loads the registers with the contents starting at the specified location and climbing up through higher addresses. The order of the registers is from D0 through D7 and then A0 through A7. The incremented address register will finally contain the address of the last word loaded plus two.

2. Predecrement mode (which can only be used for register-to-memory movement) loads memory from the registers starting at the specified address minus two and then moves down through lower addresses. The order of the registers is from A7 through A0 and then D7 through D0. The decremented address register will finally contain the address of the last word stored.

3. All other permissible addressing modes listed about (which can be used to move data in either direction) copy data starting with the specified address and then climb up through higher addresses. The order of the registers is from D0 through D7 and then A0 through A7 (just as in Postincrement mode).

Operand size: word or long-word.
Instruction length: 2 words.
Condition code effects: none.
Object code: First word (01001a001bcccddd)
Second word (mask)

Breakdown

01001: MOVEM instruction.
a: Specifies transfer direction.
0 means register to memory.
1 means memory to register.
001: MOVEM instruction (cont.).
b: Specifies operand size.
0 means word.
1 means long-word.
ccc: Source addressing mode.
ddd: Source addressing register number.
mask: The second word of this instruction is a mask that tells which registers to move. For all addressing modes except Predecrement, the mask is set up like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

For Predecrement mode, the mask is set up like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

If a register's bit position is set to 1 in the mask, it will be transferred; otherwise it will not.

Note: This is a great instruction for quick moves of processor status to memory. It is also, however, a difficult instruction to use. Be sure you understand it before incorporating it into your programs. There are extra factors to consider such as an extra read bus cycle that occurs for memory operands. Also, the assembler syntax for this instruction includes special notations. D2/D3 to means to load registers D2 and D3 and D2-D5 means to load registers D2 through D5.

MOVEP

Definition: move peripheral data.
Description: MOVEP moves two or four bytes between a specified data register and alternate byte locations in memory. This is called a peripheral data move because it simplifies the transfer of data from the CPU to peripheral devices that require 8 bits of data at a time.
Addressing modes: The data register that the data will be transferred to or from is reached only by Data Register Direct mode. Address Register Indirect with Displacement mode is used to find the memory location. If the memory address is even, the data is transferred on the most significant half of the data bus. If the memory address is odd, the data is transferred on the least significant half of the data bus. The organization of the bytes that are transferred is described below:

1. Words. The two bytes in a data register correspond to the two bytes in memory as shown in Fig. 6-1.

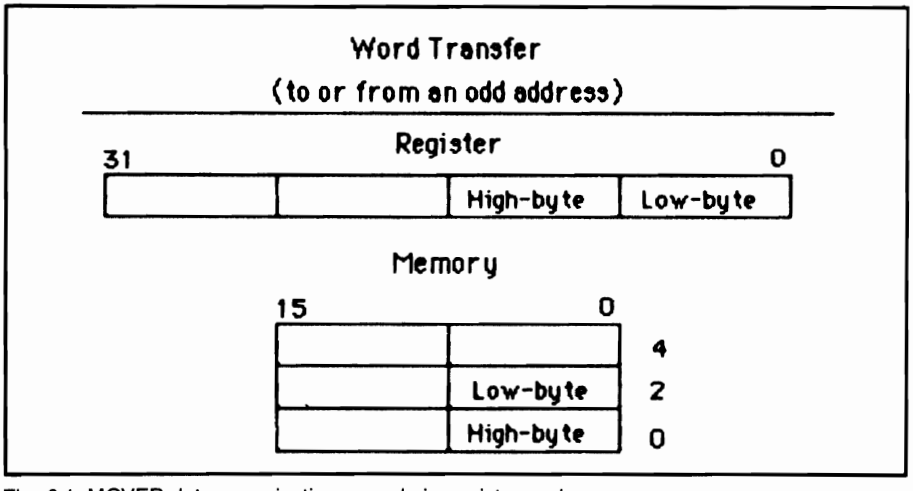


Fig. 6-1. MOVEP data organization—words in register and memory.

2. Long-words. The four bytes in a data register correspond to the four bytes in memory as shown in Fig. 6-2.
Operand size: word or long-word.
Instruction length: 2 words.

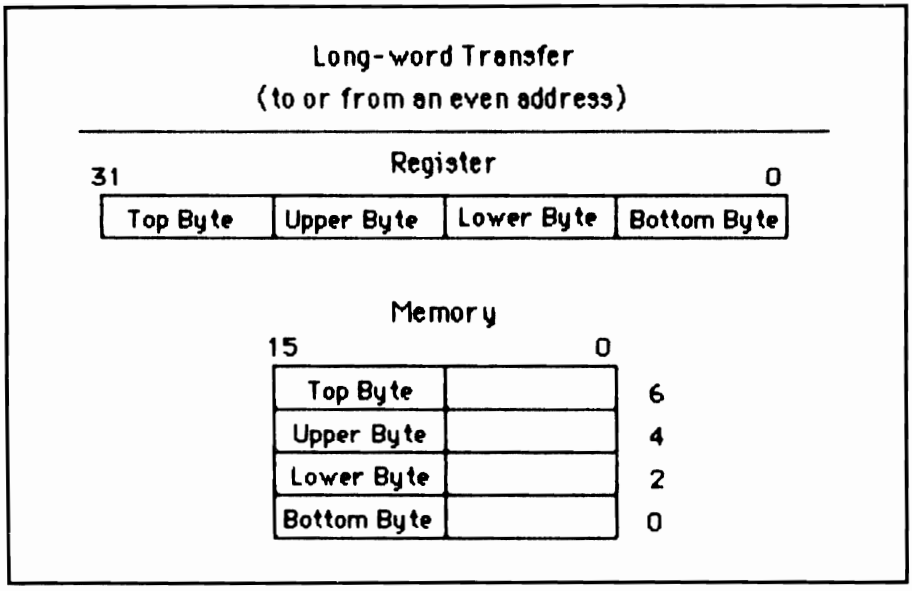


Fig. 6-2. MOVEP data organization—long-words in register and memory.

Condition code effects: none.
Object code: First word (0000aaa1bc001ddd)
Second word (displacement)

Breakdown

0000: MOVEP instruction.
aaa: Specifies the data register that the data will be transferred to or from.
1: MOVEP instruction (cont.).
b: Specifies the direction of information transfer.
1 means from memory to register.
0 means from register to memory.
c: Specifies the operand size.
0 means word.
1 means long-word.
001: MOVEP instruction (cont.).
ddd: Specifies the address register that holds the memory address.
displacement: A 16-bit value that is added to the address register specified by ddd to find the memory location for data transfer.

MOVEQ

Definition: move quick.
Description: MOVEQ moves the immediate data (found in the least-significant byte of the instruction code) to a specified data register. The eight bits of immediate data are sign-extended to 32 bits and the full 32 bits is then put in the destination data register.
Addressing modes: The source data is found only by Immediate mode. The destina-

tion data register is found only by Data Register Direct mode.
Operand size: long-word.
Instruction length: 1 word.
Conditon code effects:

N Set if the immediate data is negative; otherwise cleared.
Z Set if the immediate data is zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 0111aaa0bbbbbbbb

Breakdown

0111: MOVEQ instruction.
aaa: Destination data register number.
0: MOVEQ instruction (cont.).
bbbbbbbb: Immediate data.

Note: Because this instruction has a limited addressing ability and the immediate data is included within the single word, it executes faster than the standard MOVE instruction.

MULS

Definition: multiply signed.
Description: MULS multiplies (using two's complement arithmetic) two signed, 16-bit operands. The signed 32-bit result is stored in the destination (which is always a data register).
Addressing: The destination is always a data register and is addressed only by Data Register Direct mode. The source can be addressed by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

A source or destination operand taken from a register is always taken from the low word of that register. The upper word of the source is not disturbed: the upper word of the destination is overwritten by the result.
Operand size: word.

Instruction length: 1 word.
Condition code effects:
N Set if the product is negative; otherwise cleared.
Z Set if the product equals zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 1100aaa111bbbccc

Breakdown

1100: MULS instruction.
aaa: Destination data register number.
111: MULS instruction (cont.).
bbb: Source addressing mode.
ccc: Source addressing register number.

MULU

Definition: multiply unsigned.
Description: MULU multiplies (using unsigned binary arithmetic) two unsigned, 16-bit operands. The unsigned 32-bit result is stored in the destination data register.
Addressing: The destination is always a data register and is only specified by Data register direct mode. The source can be specified by any of the following modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

A source of destination operand taken from a register is always taken from the low word of that register. The upper word of the source is not disturbed: the upper word of the destination is overwritten by the result.
Operand size: word.
Instruction length: 1 word.
Condition code effects:
N Set if the most significant bit of the result is set; otherwise cleared.
Z Set if the product is zero; otherwise cleared.
V Always cleared.

C Always cleared.
X Not affected.

Object code: 1100aaa011bbbccc

Breakdown

11000: MULU instruction.
aaa: Destination data register number.
011: MULU instruction (cont.).
bbb: Source addressing mode.
ccc: Source addressing register number.

NBCD

Definition: negate decimal (BCD) with extend.
Description: NBCD subtracts the destination contents and the Extend flag contents from zero. The result is stored in the destination. NBCD can only work with one data byte. The term decimal means that this addition is done with BCD (Binary Coded Decimal) arithmetic. NBCD produces the ten's complement of the destination value if the extend flag equals 0, the nine's complement is the extend flag equals 1.
Addressing: Only the destination needs to be specified. It can be reached by any of these addressing modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte.
Instruction length: 1 word.
Conditon code effects:
N Undefined.
Z Cleared if the result is not zero; unchanged if the result is zero.
V Undefined.
C Set if a borrow occurs; otherwise cleared.
X Set if a borrow occurs; otherwise cleared.

Object code: 0100100000aaabbb

Breakdown

0100100000: NBCD instruction.
aaa: Destination addressing mode.

bbb: Destination addressing register number.

Notes: Programmers often set the Z flag before using this instruction for multiple-precision arithmetic. The set flag makes it easy to check for a zero result.

NEG

Definition: negate.
Description: NEG subtracts the destination contents from zero and stores the result in the destination. The computation is done using two's complement binary arithmetic.
Addressing: Only the destination needs to be specified. It can be reached by any of these addressing modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

N	Set if the result is negative; otherwise cleared.
Z	Set if the operation result is negative; otherwise cleared.
V	Set if an overflow occurs; otherwise cleared.
C	Set if a borrow occurs; otherwise cleared.
X	Set if a borrow occurs; otherwise cleared.

Object code: 01000100aaabbbccc

Breakdown

01000100: NEG instruction.
aa: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
bbb: Destination addressing mode.
ccc: Destination addressing register number.

NEGX

Definition: negate with extend.
Description: NEGX subtracts the destination contents and the extend flag contents from zero. It then stores the result in the destination. The operation is per-

formed using two's complement binary arithmetic. Because the extend flag is included, this is the binary negation instruction to use for multiple precision arithmetic.
Addressing: Only the destination needs to be specified. It can be reached by any of these addressing modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:
N Set if the result is negative; otherwise cleared.
Z Cleared if the result is non-zero; otherwise unchanged.
V Set if an overflow occurs; otherwise cleared.
C Set if a borrow occurs; otherwise cleared.
X Set if a borrow occurs; otherwise cleared.

Object code: 01000000aabbbccc

Breakdown

01000000: NEGX instruction.
aa: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
bbb: Destination addressing mode.
ccc: Destination addressing register number.

Notes: Programmers often set the Z flag before using this instruction for multiple-precision arithmetic. The set flag makes it easy to check for a zero result.

NOP

Definition: no operation.
Description: Just as the name implies, NOP doesn't do anything. The PC (program counter register) is incremented to point to the next instruction, but other than that this is just a way of wasting time. It is, however, not a wasted instruction. Though NOP is rarely used in final programs, there are many cases where NOP is used while writing or debugging a program to leave a space for a label, to precisely time some operation or loop, or to replace unwanted instructions.

Addressing: none.
Operand size: none.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100111001110001

Breakdown

0100111001110001: NOP instruction.

NOT

Definition: logical NOT (complement).
Description: NOT performs the logical NOT operation on the contents of the destination operand and stores the result in the destination. In other words, each bit of the source is examined, 1s are changed to 0s, 0s are changed to 1s, and the final complemented result is put into the destination.
Addressing: Only the destination needs to be specified. It can be reached by any of these addressing modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

N Set if the result is negative; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 01000110aabbbccc

Breakdown

01000110: NOT instruction.
aa: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
bbb: Destination addressing mode.
ccc: Destination addressing register number.

OR

Definition: OR logical.

Description: OR performs the logical inclusive-OR operation on the contents of the specified source and the specified destination. The result is stored in the destination. There are two forms of this instruction, that differ only in addressing, as described in the next paragraph.

Addressing: OR can be used with any of a large number of addressing modes. The two forms of this instruction offer different addressing choices.

1. Data Register Direct destination. The destination must be addressed by Data Register Direct; any addressing mode except Address Register Direct can be used for the source including:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

2. Data Register Direct source. The source must be addressed by Data Register Direct. Almost all the addressing modes (except Program Counter Relative with Displacement, Program Counter Relative, and Immediate) can be used for the destination including:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

If you want to directly address a data register, use the first case described above, not Data Register Direct in this mode.

Operand size: byte, word, or long-word.

Instruction length: 1 word.

Condition code effects:

N Set if the MSB of the result is one; otherwise cleared.

Z Set if the result equals zero; otherwise cleared.

V Always cleared.
C Always cleared.
X Not affected.

Object code: 1000aaabccdddeee

Breakdown

1000: OR instruction.
aaa: Specifies data register number (four source or destination, depending on first or second case).
b: Specifies operation mode (first or second case).
0 means first case (data register is destination).
1 means second case (data register is source).
cc: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
ddd: Effective addressing mode.
eee: Effective addressing register number.

Notes: Address register contents cannot be used as an operand.

ORI

Definition: OR logical immediate.
Description: ORI performs the logical inclusive-OR operation on the contents of a specified destination and an immediate value contained in the next program words. The result is stored in the destination.
Addressing: The source is addressed by Immediate. The destination is reached by any of these addressing modes.

Data Register Direct
Address Register Indirect.
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 2 words (if the immediate data is a byte or a word, the first word is the instruction and the second contains the data); 3 words (if the immediate data is a long-word, the first word is the instruction and the next two are the long-word data).
Condition code effects:
N Set if the result is negative; otherwise cleared.

Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: First word (00000000aabbbccc)
Second word (immediate data)
Third word (immediate data)

Breakdown

00000000: ORI instruction.
aa: Operand size specification (if byte is specified, the low-order byte of the immediate instruction word will be used).
00 means byte.
01 mean word.
10 means long-word.
bbb: Destination addressing mode.
ccc: Destination addressing register number.
immediate data: Byte data is held in the low-order byte of the second word. Word data is the second word. Long-word data requires a three word instruction with the second and third words representing the data.

ORI to CCR

Definition: OR logical immediate to condition code register.
Description: This is a special form of the ORI instruction. ORI to CCR performs a logical inclusive-OR operation on the contents of the condition codes register (CCR: the low-order byte of the status register) and the immediate value contained in the low-order byte of the next program word. The result is stored in the CCR.
Addressing: The source is addressed by Immediate. The destination is the condition code register.
Operand size: byte.
Instruction length: 2 words (the first in the instruction word and the second contains the data as the low-order byte. The high order byte of the second word holds all zeros.)
Condition code effects: (these are set directly from the operation's results, which are stored in the flag register)

N Set if bit 3 of the immediate data equals 1; otherwise unchanged.
Z Set if bit 2 of the immediate data equals 1; otherwise unchanged.
V Set if bit 1 of the immediate data equals 1; otherwise unchanged.
C Set if bit 0 of the immediate data equals 1; otherwise unchanged.
X Set if bit 4 of the immediate data equals 1; otherwise unchanged.

Object code: First word (0000000000111100)
Second word (immediate data)

Breakdown

0000000000111100: ORI to CCR instruction

immediate data: The high-order byte is filled with zeros; the low-order byte holds the immediate data.

ORI to SR

Definition: OR logical immediate to statues register (privileged).

Description: This is a special form of the ORI instruction and is a privileged instruction: the CPU must be in the Supervisor state for this instruction to execute. ORI to SR performs a logical AND operation on the contents of the full status register and the immediate value contained in the next program word. The result is stored in the status register. This instruction is privileged—meaning it can only be executed if the processor is in the Supervisor state—because it can change the entire state of the CPU.

Addressing: The source is addressed by Immediate mode. The destination is the status register.

Operand size: word.

Instruction length: 2 words (the first is the instruction word and the second contains the immediate value).

Condition code effects: (these are set directly from the operation's results, which are stored in the status register)

N Set if bit 3 of the immediate data equals 1; otherwise unchanged.
Z Set if bit 2 of the immediate data equals 1; otherwise unchanged.
V Set if bit 1 of the immediate data equals 1; otherwise unchanged.
C Set if bit 0 of the immediate data equals 1; otherwise unchanged.
X Set if bit 4 of the immediate data equals 1; otherwise unchanged.

Object code: First word (0000000001111100)
Second word (immediate data)

Breakdown

0000000001111100: ORI to SR instruction.

immediate data: The full 16 bits of the word hold the immediate date to be added to the status register.

PEA

Definition: push effective address.

Description: PEA puts together an effective address and stores that long-word address on the stack. Chapter 4 explains what an effective address is.

Addressing: The address to be put on the stack can be calculated using any of these addressing modes:

Address Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short

Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index

Operand size: long-word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100100001aaabbb

Breakdown

0100100001: PEA instruction.
aaa: Effective addressing mode.
bbb: Effective addressing register number.

RESET

Definition: reset external devices (privileged).
Description: RESET sends a pulse out on the RESET pin. Such pulses reset external devices (peripherals that are hooked to the 68000). RESET is a privileged instruction and so cannot be executed unless the CPU is in Supervisor state. The internal state of the 68000 CPU doesn't change, except that the PC is incremented by two to move to the next instruction.
Addressing: none. The RESET pin is always the target on the output pulse.
Operand size: none.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100111001110000

Breakdown

0100111001110000: RESET instruction.

ROL

Definition: rotate left (without extend).
Description: ROL rotates the bits of the specified location to the left. In other words, the contents of each bit position is moved to a more significant bit position. The bit content of the most significant bit is moved into both the least significant bit position and into the carry flag (replacing whatever was in the flag). The rotation can be made on a data register or on a memory location.

1. Data register rotations can range from 1 to 63 bit positions. The number of positions rotated is specified in one of two ways. A. The rotation count can be held in the least significant six bits of a data register. That can specify a rotation of from 0 to 63 bits. B. Immediate data can specify the rotation number and that can range from 1 to 8 bits. (A value of 000 in the immediate data means a rotation of 8 positions.)
2. Memory location rotations can only be a single bit position.

Addressing:

1. Register. Uses Data Register Direct mode for the register to be rotated (the destination).

2. Memory. Uses any of the following modes for the memory location to be rotated:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
1. Memory: word.

Instruction length: 1 word.

Condition code effects:

N Set if the most significant bit of the result is 1; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C The last bit shifted out of the operand is copied into the C flag. C is cleared if a shift count of zero is performed.
X Not affected.

Object code:

1. Register: 1110aaa1bbc11ddd

Breakdown

1110: ROL instruction.
aaa: Rotation count or number of specified.
Register (which it is depends on bit 5, "c," as described below).
1: ROL instruction (cont.).
bb: Operand size specification.
00 means byte.
01 means word.
11 means long-word.
c: Specifies where the rotation count is held.
0 means the rotation count is held in bits 9, 10, and 11 ("aaa"). 1 means the rotation count is held in a data register specified by "aaa."
11: ROR instruction (cont.).
ddd: Specifies the data register to be rotated.

2. Memory: 1110011111aaabbb

Breakdown

1110011111: ROL instruction, memory.

aaa: Destination addressing mode.
bbb: Destination addressing register number.

ROR

Definition: rotate right (without extend).
Description: ROR rotates the bits of the specified location to the right. In other words, the contents of each bit position are moved to a less significant bit position. The bit content of the least significant bit is moved into both the most significant bit position and into the carry flag (replacing whatever was in the flag). The rotation can be made on a data register or on a memory location.

1. Data register rotations can range from 1 to 63 bit positions. The number of positions rotated is specified in one of two ways. A. The rotation count can be held in the least significant six bits of a data register. That can specify a rotation of from 0 to 63 bits. B. Immediate data can specify the rotation number and that can range from 1 to 8 bits. (A value of 000 in the immediate data means a rotation of 8 positions).
2. Memory location rotations can only be a single bit position.

Addressing:

1. Register. Uses Data Register Direct mode for the register to be rotated (the destination).
2. Memory. Uses any of the following modes for the memory location to be rotated:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
2. Memory: word.

Instruction length: 1 word.
Condition code effects:

N Set if the most significant bit of the result is 1; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C The last bit rotated out of the operand is copied into the C flag. C is cleared in a rotation count of zero is performed.
X Not affected.

Object code:

1. Register: 1110aaa0bbc11ddd

Breakdown

1110: ROR instruction.
aaa: Rotation count or number of specified register (which it is depends on bit 5, "c," as described below).
0: ROR instruction (cont.).
bb: Operand size specification.
00 means byte.
01 means word,
11 means long-word.
c: Specifies where the rotation count is held.
0 means the rotation count is held in bits 9, 10, and 11 ("aaa"). 1 means the rotation count is held in a data register specified by "aaa."
11: ROR instruction (cont.).
ddd: Specifies the data register to be rotated.

2. Memory: 1110011011aaabbb

Breakdown

1110011011: ROR instruction, memory.
aaa: Destination addressing mode.
bbb: Destination addressing register number.

ROXL

Definition: rotate left (with extend).
Description: ROXL rotates the bits of the specified location to the left. In other words, the contents of each bit position is moved to a more significant bit position. The contents of the most significant bit is moved into the carry flag and the extend flag (replacing whatever was in the flags). The previous extend flag value is moved into the least significant bit position of the specified destination. By including the extend flag in the rotation, this instruction is useful for multiple precision arithmetic. The rotation can be made on a data register or on a memory location.

1. Data register rotations can change from 1 to 63 bit positions. The number of positions rotated is specified in one of two ways. A. The rotation count can be held in the least significant six bits of a data register. That can specify a rotation of from 0 to 63 bits. B. Immediate data can specify the rotation number and that can range from 1 to 8 bits. (A value of 000 in the immediate data means a rotation of 8 positions.)

2. Memory location rotations can only be a single bit position.

Addressing:

1. Register. Uses Data Register Direct mode for the register to be rotated (the destination).

2. Memory. Uses any of the following modes for the memory location to be rotated:

Address Register Indirect

Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
2. Memory: word.

Instruction length: 1 word.

Condition code effects:

N Set if the most significant bit of the result is 1; otherwise cleared.

Z Set if the result equals zero; otherwise cleared.

V Always cleared.

C The last bit rotated out of the operand is copied into the C flag. C is cleared if a rotation count of zero is performed.

X The last bit rotated out of the operand is copied into the X flag. A rotation count of zero leaves the X flag unaffected.

Object code:

1. Register: 1110aaa1bbc10ddd

Breakdown

1110: ROXL instruction.

aaa: Rotation count or number of specified register (which it is depends on bits 5, "c", as described below).

1: ROXL instruction (cont.).

bb: Operand size specification.
00 means byte.
01 means word.
11 means long-word.

c: Specifies where the rotation count is held.
0 means the rotation count is held in bits 9, 10, and 11 ("aaa"). 1 means the rotation count is held in a data register specified by "aaa".

10: ROXL instruction (cont.).

ddd: Specifies the data register to be rotated.

2. Memory: 1110010111aaabbb

Breakdown

1110010111: ROXL instruction, memory.

aaa: Destination addressing mode.

bbb: Destination addressing register number.

ROXR

Definition: rotate right (with extend).

Description: ROXR rotates the bits of the specified location to the right. In other words, the contents of each bit position is moved to a less significant bit position. The contents of the least significant bit is moved into the carry flag and the extend flag (replacing whatever was in the flags). The previous extend flag value is moved into the most significant bit position of the specified destination. By including the extend flag in the rotation, this instruction is useful for multiple precision arithmetic.

The rotation can be made on a data register or on a memory location.

1. Data register rotations can range from 1 to 63 bit positions. The number of positions rotated is specified in one of two ways. A. The rotation count can be held in the least significant six bits of a data register. That can specify a rotation of from 0 to 63 bits. B. Immediate data can specify the rotation number and that can range from 1 to 8 bits. (A value of 000 in the immediate data means a rotation of 8 positions.)
2. Memory location rotations can only be a single bit position.

Addressing:

1. Register. Uses Data Register Direct mode for the register to be rotated (the destination).
2. Memory. Uses any of the following modes for the memory location to be rotated:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size:

1. Registers: byte, word, or long-word.
2. Memory: word.

Instruction length: 1 word.

Conditon code effects:

N Set if the most significant bit of the result is 1; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C The last bit rotated out of the operand is copied into the C flag. C is cleared if a rotation count of zero is performed.
X The last bit rotated out of the operand is copied into the X flag. A rotation count of zero leaves the X flag unaffected.

Object code:

1. Register: 1110aaa0bbc10ddd

Breakdown

1110: ROXR instruction.

aaa: Rotation count or number of specified register (which it is depends on bit 5, "c," as described below).
0: ROXR instruction (cont.).
bb: Operand size specification.
00means byte.
01 means word.
11 means long-word.
c: Specifies where the rotation count is held.
0 means the rotation count is held in bits 9, 10, and 11 ("aaa"). 1 means the rotation count is held in a data register specified by "aaa."
10: ROXR instruction (cont.).
ddd: Specifies the data register to be rotated.

2. Memory: 1110010011aaabbb

Breakdown

1110010011: ROXR instruction, memory.
aaa: Destination addressing mode.
bbb: Destination addressing register number.

RTE

Definition: return from exception (privileged).
Description: RTE loads the SR (status register) and then the PC (program counter) from the system stack. The first word pulled off the Supervisor stack is put into the SR; the second and third words pulled are put into the PC (the second becomes the high word and the third and low word of the PC). The previous SR and PC values are lost. This is typically the last instruction executed in an exception processing service routine. RTE is a privileged instruction and so will only execute when the CPU is in supervisor state. The new state of the CPU will depend on the values put into the status register. The bits of the status register that have not yet been assigned values will always retain a 0.
Operand size: none.
Instruction length: 1 word.
Condition code effects: Set directly from the value pulled off the stack and stored in the status register.
Object code: 0100111001110011

Breakdown

0100111001110011: RTE instruction.

RTR

Definition: return and restore condition codes.
Description: RTR pulls a word off the active stack and puts the five least significant bits of that word into the condition codes in the status register. Then the PC is filled from the top two words of the stack. The stack pointer is incremented to keep up with these manipulations. This allows a return from subroutine—with

replacement of the user flags—without affecting the system byte of the status register.
Addressing: none.
Operand size: none.
Instruction length: 1 word.
Conditon code effects: Set directly from the value pulled off the stack and stored in the status register.
Object code: 0100111001110111

Breakdown

0100111001110111: RTR instruction.

Note:

1. Some microprocessors automatically save the condition codes (or flags) when they jump to a subroutine: the 68000 does not. You have to add a MOVE from SR instruction to your program to save the flags if you want to recall them later with this RTR instruction.

2. The only difference between RTR and RTE is that RTR doesn't affect the new value of the high word of the SR.

3. Remember that RTR doesn't only restore the condition codes, it also restores the PC.

RTS

Definition: return from subroutine.
Description: RTS pulls the top two words off of the active stack and puts them into the PC. The first word pulled becomes the high word of the PC and the second word pulled becomes the low word. The stack pointer is incremented by 2 after each word is pulled off. RTS is used to return the program control to the point it was at before a subroutine.
Addressing: none.
Operand size: none.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100111001110101

Breakdown

0100111001110101: RTS instruction.

SBCD

Definition: subtract decimal (with extend).
Description: SBCD subtracts the least significant byte of the source, and the extend flag, from the least significant byte of the destination. It then stores the result in the destination. The term decimal means that this addition is done with BCD (Binary Coded Decimal) arithmetic. SBCD has two major cases, register-to-register and memory-to-memory.

1. Register-to-register uses data registers for both source and destination.

2. Memory-to-memory uses memory locations for both source and destination. The memory address of the source operand is stored in an instruction-specified address register, and the destination address is stored in another instruction-specified address register. Before the operation, each address register is decremented (predecrement addressing mode).

Addressing:

1. Register-to-register. Data Register Direct mode is used for both source and destination. (The values are stored in registers and the registers are specified directly by the instruction).

2. Memory-to-memory. Address Register Indirect mode is used for both source and destination (the instruction specifies the address registers that hold the memory addresses of source and destination. Before the values in the address registers are used, they are decremented by 1. This helps in multibyte BCD subtraction).

Operand size: byte.

Instruction length: 1 word.

Condition code effects:

N Undefined.

Z Cleared if the result is not equal to zero; unchanged if the result equals zero.

V Undefined.

C Set if a borrow occurs; otherwise cleared.

X Set if a borrow occurs; otherwise cleared. (Set in the same way as the carry (C) flag.)

Object Code:

1. Register-to-register: 1000aaa100000bbb

Breakdown

1000: SBCD instruction
aaa: Specifies destination data register.
100000: Specifies register-to-register case.
bbb: Specifies the source data register.

2. Memory-to-memory: 1000aaa100001bbb

Breakdown

1000: SBCD instruction
aaa: Specifies destination data register.
100001: Specifies memory-to-memory case.
bbb: Specifies the source data register.

Note: Programmers often set the Z flag before using this instruction for multiple-precision arithmetic. The set flag makes it easy to check for a zero result.

Scc

Definition: set (conditionally).
Description: Scc tests the state of the flags in the status register. If the condition (which is specified within the instruction) is met, all of the bits of the specified byte are set (equal to 1; this is called TRUE). Otherwise, if the condition is not met, the bits are all cleared (equal to 0; called FALSE). You can choose from any of the following conditions:

Symbol	Title	Operation
T	True	1
F	False	0
HI	High	~C AND ~Z
LS	Low or same	C OR Z
CC	Carry clear	~C
CS	Carry set	C
NE	Not equal	~Z
EQ	Equal	Z
VC	Overflow clear	~V
VS	Overflow set	V
PL	Plus	~N
MI	Minus	N
GE	Greater or equal	(N AND V) OR (~N AND ~V)
LT	Less than	(N AND ~V) OR (~N AND V)
GT	Greater than	(N AND V AND ~Z) OR (~N AND V AND ~Z)
LE	Less or equal	Z or (N AND ~V) OR (~N AND V)

The instruction is written with the condition symbol replacing the "cc." For example, SNE means "set if not equal."
Addressing: The byte to set or clear is addressed by any of these modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0101aaaa11bbbccc

Breakdown

0101: Scc instruction.
aaaa: Specifies the condition to be tested.
11: Scc instruction (cont.).
bbb: Effective addressing mode.
ccc: Effective addressing register number.

STOP

Definition: load status register and stop (privileged).
Description: STOP loads an immediate word into the status register and then stops fetching and executing instructions. Execution will resume when a trace, interrupt, or reset exception occurs. The PC is incremented by four to point to the next instruction: the word following STOP is the immediate data.

A trace occurs immediately if the T flag is set when STOP is executed.

Exception processing from interrupt will occur if an interrupt request of high enough priority is detected. If the external reset signal goes low, the reset exception will begin.

STOP is privileged; it can only be executed from the Supervisor mode. Attempting to execute it in User mode will violate privilege and begin exception processing.
Addressing: The value to load into the status register is found by Immediate mode and is the second word of the instruction.
Operand size: none.
Instruction length: 2 words.
Condition code effects: All of the flags are changed by the word that is loaded into the status register. The old values are lost.
Object code: First word (0100111001110010)
Second word (immediate)

Breakdown

0100111001110010: STOP instruction.
immediate: This is the value that is put into the SR.

SUB

Definition: subtract (binary).
Description: SUB subtracts the source operand from the destination operand and stores the result in the destination. There are two forms of this instruction that differ only in addressing, as described in Addressing.
Addressing: SUB can be used with any of a large number of addressing modes. The two forms of this instruction offer different addressing choices.

1. Data Register Direct destination. The destination must be addressed by Data Register Direct mode; any addressing mode can be used for the source including:

Data Register Direct
Address Register Direct

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

2. Data Register Direct source. The source must be addressed by Data Register Direct mode. Almost all the addressing modes (except Program Counter Relative with Displacement, Program Counter Relative, and Immediate) can be used for the destination including:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word. As noted below, bytes cannot be used with Address Register Direct mode.
Instruction length: 1 word.
Condition code effects:
N Set if the result in negative; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.
X Set if a borrow occurs; otherwise cleared.
C Set if a borrow occurs; otherwise cleared.

Object code: 1001aaabccdddeee

Breakdown

11010: SUB instruction.
aaa: Data register number (for either source or destination Data Register Direct addressing).
b: Operating mode.
0 means the data register is the destination.
1 means the data register is the source.
cc: Size specification.

00 means byte.
01 means word.
10 means long-word.
ddd: Addressing mode.
eee: Addressing register number.

Notes:

1. If Address Register Direct addressing is used, the operand size cannot be specified as byte because address registers cannot work with bytes (only with words and long-words).

2. To use a data register as a destination you must use the Data Register Direct mode.

SUBA

Definition: subtract address.
Description SUBA is a special case of the SUB instruction. SUBA subtracts the source operand from the destination address register contents and stores the result in that address register.
Addressing: The destination is only reached by Address Register Direct. Any mode can be used for the source operand including:

Data Register Direct
Address Register Direct
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long
Program Counter Relative with Displacement
Program Counter Relative with Index
Immediate

Operand Size: words or long-words. The full destination address register is used no matter which operand size is chosen. A word source-operand will be size-extended to a long-word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 1001aaab11cccddd

Breakdown

1001: SUBA instruction.
aaa: Destination address register.
b: Size specification.
0 means word.
1 means long-word.

11: SUBA instruction (cont.).
ccc: Source addressing mode.
ddd: Source addressing register.

SUBI

Definition: Subtract immediate.
Description: SUBI subtracts immediate data (which is contained in the next instruction byte or bytes) from the specified destination operand. The result is stored in the destination.
Addressing: The source is addressed by Immediate mode. The destination is reached by any of the following modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 2 or 3 words. 2 words if the immediate data is a byte or a word (the first word is the instruction and the second contains the data). 3 words if the immediate data is a long-word (the first word is the instruction and the next two are the long-word data).
Condition code effects:
N Set if the result is negative; otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Set if an overflow occurs; otherwise cleared.
C Set if a borrow occurs; otherwise cleared.
X Set if a borrow occurs; otherwise cleared.

Object code: First word (00000100aabbbccc)
Second word (immediate data)
Third word (immediate data)

Breakdown

00000100: SUBI instruction.
aa: Size specification.
00 means byte.
01 means word.
10 means long-word.
If a byte is specified, the low-order byte of the next instruction is used by the assembler.
bbb: Destination addressing mode.

ccc: Destination address register.
immediate data: Byte data is held in the low-order byte of the second word. Word data is the second word. Long-word data requires a three word instruction with the second and third words representing the data.

SUBQ

Definition: subtract quick.
Description: SUBQ subtracts immediate data (contained within the instruction word itself) from the specified destination operand. The result is stored in the destination. As the definition implies, SUBQ is used for quick execution.
Addressing: For the source operand you can use only Immediate mode. For the destination operand you can use any of the following modes:

Data Register Direct
Address Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

N	Set if the result is negative; otherwise cleared.
Z	Set if the result equals zero; otherwise cleared.
V	Set if an overflow occurs; otherwise cleared.
C	Set if a borrow occurs; otherwise cleared.
X	Set if borrow occurs; otherwise cleared.

Object code: 0101aaa1bbcccddd

Breakdown

0101: SUBQ instruction.
aaa: Data field (holding three bits of immediate data with 000 representing 8 and 001 through 111 representing 1 through 7).
1: SUBQ instruction (cont.).
bb: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
IF a byte is specified, the low-order byte of the next instruction is automatically used by the assembler.

ccc: Destination addressing mode.
ddd: Destination register number.

Note: If Address Register Direct addressing is used, the operand size cannot be specified as byte because address registers cannot work with bytes (only with words and long-words).

SUBX

Definition: subtract with extend.
Description: SUBX subtracts the source contents, and the extend flag, from the destination contents. The result is stored in the destination. SUBX has two major cases, register-to-register and memory-to-memory.

1. Register-to-register uses data registers for both source and destination.
2. Memory-to-memory uses memory locations for both source and destination. The memory address of the source operand is stored in an instruction-specified address register, and the destination address is stored in another instruction-specified address register. Before the operation, each address register is decremented (predecrement mode).

Addressing: Register-to-register uses Data Register Direct mode for both source and destination. Memory-to-memory uses Address Register Indirect mode.

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

N	Set if the result is negative; otherwise cleared.
Z	Cleared if the result is not equal to zero; otherwise unchanged.
V	Set if an overflow occurs; otherwise cleared.
C	Set if a carry occurs; otherwise cleared.
X	Set if a carry occurs; otherwise cleared.

Object code:

1. Register-to-register: 1001aaa1bb000ccc

Breakdown

1001: SUBX instruction.
aaa: Destination data register number.
1: SUBX instruction (cont.).
bb: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
000: SUBX instruction (cont.).
ccc: Source address data register number.

2. Memory-to-memory: 1001aaa1bb001ccc

Breakdown

1001: SUBX instruction.
aaa: Destination address register number.
1: SBUX instruction (cont.).
bb: Operand size specification.
00 means byte.
01 means word.
10 means long-word.
001: SUBX instruction (cont.). Memory to memory case.
ccc: Source address register number.

SWAP

Definition: swap register halves.
Description: SWAP exchanges the contents of the low word and high word of a specified data register.
Addressing: The only mode used is Data Register Direct.
Operand size: word.
Instruction length: 1 word.
Condition code effects:

N Set if the result is negative (if the most significant bit of the 32-bit result is set); otherwise cleared.
Z Set if the result equals zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 0100100001000aaa

Breakdown

0100100001000: SWAP instruction.
aaa: Number of the data register to be swapped.

TAS

Definition: test and set an operand.
Description: TAS tests a specified byte, sets the N and Z flags according to the contents of that byte, and sets the high-order bit of the byte (equal to 1). This is called an indivisible instruction because the CPU uses a read-modify-write memory cycle that cannot be interrupted (which means that no other device can get that operand while this instruction is being executed). This allows separate processors to synchronize their activities.
Addressing: Any of these modes can be used to find the byte:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect

Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte.
Instruction length: 1 word.
Condition code effects:

N Set if the most significant bit of the operand is set; otherwise cleared. The bit is tested at the beginning of the instruction, because after the instruction the MSB will be set.
Z Set if the operand contents equal zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 0100101011aaabbb

Breakdown

0100101011: TAS instruction.
aaa: Destination addressing mode.
bbb: Destination addressing register number.

TRAP

Definition: trap.
Description: TRAP initiates exception processing (see Chapter 7 for an explanation of exceptions). The PC value is incremented (as it would be to get the next instruction) and then is pushed onto the system stack (using the Supervisor stack pointer: SSP). The status register word is pushed onto the stack next. Two words from the exception vector table—specified by the 4-bit vector of the instruction word—are put into the PC. The sixteen possible vectors allow different processing for different types of exceptions. The T (trace) flag is set to zero and the S (Supervisor) flag is set to one. All of this activity saves the old status of the CPU before moving to the new status. Processing continues at the new PC value.
Addressing: none.
Operand size: none.
Instruction length: 1 word.
Condition code effects: none.
Object code: 010011100100aaaa

Breakdown

010011100100: TRAP instruction.
aaaa: Vector number. This value specifies the address is the exception vector table from which the new PC value will be taken. This table is shown in Chapter 7. aaaa can specify sixteen different addresses. These are addresses 32 through 47 in the 255 vector table.

TRAPV

Definition: trap on overflow.
Description: TRAPV checks the V (overflow) flag and initiates exception processing if that flag is set. See Chapter 7 for details on exception processing. Exception processing increments the PC value (as it would to get the next instruction) and then pushes it onto the system stack (using the Supervisor stack pointer: SSP). The status register word is pushed onto the stack next. Two words from the exception number 7 of the vector table (beginning at 01CH) are put into the PC. The T (trace) flag is set to zero and the S (Supervisor) flag is set to one. All of this activity saves the old status of the CPU before moving to the new status. Processing continues at the new PC value.
Addressing: none.
Operand size: none.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100111001110110

Breakdown

0100111001110110: TRAPV instruction.

TST

Definition: test and operand.
Description: TST tests the contents of a specified operand and sets the N and Z flags according to the result. This instruction doesn't change anything in the CPU or memory except the flags.
Addressing: The operand to test can be found by any of the following addressing modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: byte, word, or long-word.
Instruction length: 1 word.
Condition code effects:

N Set if the tested operand is negative; otherwise cleared.
Z Set i the tested operand equals zero; otherwise cleared.
V Always cleared.
C Always cleared.
X Not affected.

Object code: 01001010aabbbccc

Breakdown

01001010: TST instruction.
aa: Operand size.
00 means a byte.
01 means a word.
10 means a long-word.
bbb: Operand addressing mode.
ccc: Operand addressing register number.

Note: This can be a very handy instruction to change flag values without affecting anything else in the CPU (except the PC, of course).

UNLK

Definition: unlink.
Description: UNLK loads the contents of a specified address register into the system stack pointer. That address register, called a frame pointer, is then loaded with a long word from the stack. This, in effect, restores the frame pointer and the system stack pointer to what their state was before a LINK instruction was executed. See the description of the LINK instruction to understand the overall action of this command.
Addressing: none.
Operand size: none.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100111001011aaa

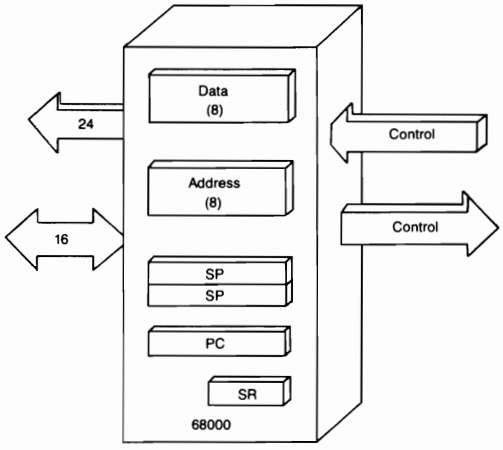
Breakdown

0100111001011: UNLK instruction.
aaa: Specifies the frame pointer address register.

Addressing Modes:

	Mode	Reg #
Dn	000	ø-7
An	001	ø-7
(An)	010	ø-7
(An)+	011	ø-7
-(An)	100	ø-7
d(An)	101	ø-7
d(An, Xi)	110	ø-7
Abs. W	111	ØØØ
Abs. L	111	øø1
d(PC)	111	010
d(PC, Xi)	111	011
Immed.	111	100

7



Exceptions

EXCEPTIONS ARE SPECIAL OCCURRENCES that require processing outside of normal instruction execution. If an exception occurs, normal processing will cease and exception processing will commence. The high-order byte of the status register controls many aspects of exception processing.

There are two main reasons to define exceptions. The first is to allow the microprocessor to act quickly when some special situation occurs (such as someone pressing a key on the keyboard of a computer). The second is to allow the microprocessor to report and deal with errors. Division by zero or the execution of a nonexistent instruction are just two situations that would require exception processing.

POLLING, INTERRUPTS, AND EXCEPTIONS

Microprocessors have to respond to many outside events. These may range from alarm signals to disk information input. There are two basic ways of watching for such inputs: polling and interrupts.

When a microprocessor asks an I/O device if that device has any information, the microprocessor is *polling* that device. Polling schemes are sometimes used because they are simple to implement and ensure that every I/O device is monitored. The problem with polling is in the timing. If there is a large polling loop and a device needs to signal the microprocessor just after having been polled, that device will have to wait for its next turn in the polling loop. Even though the information from the device might be urgent (such as an alarm condition) the microprocessor wouldn't have any way of reacting immediately.

Interrupts avoid the timing problems associated with polling. While interrupt hardware and software is more complex than polling schemes, it ensures timely response to events. Interrupt schemes also allow the programmer to dynamically assign priorities to the various I/O devices.

How are interrupts implemented? When an I/O device has some information for the CPU, it sends an interrupt request signal and a priority signal.

When the microprocessor is free to answer—typically after the current instruction is done executing—it will compare the signalled priority to its stored priority level. If the interrupt request has a high enough priority, the microprocessor will jump to an interrupt handling routine. When that routine is finished, the microprocessor will return processing to the point it had reached before the interrupt.

Most interrupt handling schemes allow multiple, simultaneous interrupts. When a second interrupt (which must be of greater priority than the first) breaks in on the first, the microprocessor sends processing to the second interrupt handling routine. Then, when the second interrupt has been completely taken care of, processing will take up in the first interrupt handling routine where it left off. The 68000 has seven levels of interrupt priority; the current level is stored in the mask in the Status register.

The addressing of interrupt handling routines can be simple or complex. Some systems have a single routine, others have a long table full of routines that are called for different types of interrupts. Other systems allow the programmer to modify addresses from within the program.

8-bit microprocessors commonly have interrupt handling capability. As explained above, interrupts are requests by devices outside the microprocessor for special processing. The program routine that is used to handle the interrupt may come from any of a variety of addresses, depending on the microprocessor's *vectoring* scheme. The vector is the value that points to the interrupt service routine. Different microprocessors have different methods to calculate vectors.

Exception is a broader term than interrupt. The 68000 also has interrupt handling abilities, but they are classified as one type of exception.

68000 PROCESSING STATES

There are three processing states that a 68000 can be in: normal, halted, or exception. In the normal state, the microprocessor is fetching and executing instructions. There is also the special normal state case of the STOP instruction which stops the referencing of memory.

The halted state is different from the stopped condition of the Normal state. Only a catastrophic hardware error can send the CPU into the halted state. The only way out of the halted state is an external reset (having the right pulse of electricity sent to the correct pin of the microprocessor). Programmers don't have to worry about halted state.

The exception state may be generated internally (by instructions) or externally (by an interrupt, a bus error, or a reset). The exception state is used to work with interrupts, traps, and tracing.

68000 PRIVILEGE MODES

There are two privilege modes that the 68000 can be in: User mode and Supervisor mode. The status register Supervisor/User bit (flag) controls which mode the microprocessor is in and therefore controls the following:

1. Which instructions are legal.
2. How external memory management reaches memory.
3. Which stack pointer, Supervisor or User, is active.

These two Privilege modes are intended as a basic security structure for 68000 systems. In other words, general computation is done in the User mode and system modification is done in Supervisor mode.

User Mode

The microprocessor is in User mode if the S flag contains a 0. Certain *privileged instructions* will not execute while the CPU is in this mode. This restriction protects programs by not allowing applications to work with system software. Nonprivileged instructions execute the same way in both User and Supervisor modes.

STOP and RESET are both privileged. Also, instructions that can modify the entire status register are privileged because they could be used to get into Supervisor mode. Figure 7-1 lists the 68000's privileged instructions.

The active stack pointer in User mode is the User stack pointer, naturally. Any references to the

ANDI to SR
EORI to SR
MOVE to SR
MOVE USP
ORI to SR
RESET
RTE
STOP

Fig. 7-1. Privileged instructions.

stack pointer or to address register 7 will encounter the USP.

When the CPU is in user mode, and working through instructions, only an exception can move it into supervisor mode. Exception processing always begins by asserting the S bit, thereby putting the CPU into Supervisor mode.

The four instructions that allow the user to move from Supervisor to User mode are among those listed in Fig. 7-1. RTE gets the new SR and PC values from the Supervisor stack. MOVE, ANDI, EORI change the SR and therefore the S bit and are able to change mode.

Supervisor Mode

If the Supervisor/User bit in the status register has a 1 value, the CPU is in Supervisor mode. In the Supervisor mode, all instructions function and all of memory is available. Supervisor mode, therefore, is the more powerful mode. Address register 7 (the stack pointer) is the Supervisor stack pointer (SSP) in this mode.

Exceptions are always processed in the Supervisor mode. If the Supervisor/User bit isn't 1 when the exception starts, it will be changed to execute the exception.

REFERENCE CLASSIFICATION

Whenever the 68000 CPU refers to memory, the reference is classified as shown in Fig. 7-2. These references show up as various voltage levels on the FC0, FC1, FC2 pins so that external devices can understand what the 68000 is doing. This allows external address translation and memory protection.

EXCEPTION PROCESSING

When an exception is processed by the 68000, it automatically saves the PC and SR values, and then it puts an exception vector address in the PC for further processing. The exception vectors are stored in low memory. Because the PC and SR values were saved, processing can resume at the same point later. The exception handling routines are, in

Function Codes (put out on signal lines)			Type of Reference
FC2	FC1	FC0	
0	0	0	Unassigned
0	0	1	User Data
0	1	0	User Program
0	1	1	Unassigned
1	0	0	Unassigned
1	0	1	Supervisor Data
1	1	0	Supervisor Program
1	1	1	Interrupt Acknowledge

Fig. 7-2. Types of memory reference.

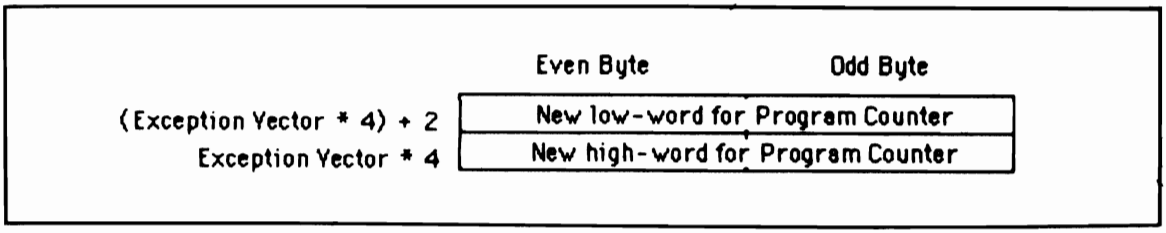


Fig. 7-3. Exception vector format.

essence, subroutines from which processing will return.

Exception processing takes place in four steps.

1. The SR is copied and saved and then filled with a new SR value for the exception. The S flag is set and the T flag is cleared. Reset and interrupt exceptions also bring the interrupt mask up to data with a new level.
2. The vector number is found. Interrupts get the vector number from a processor fetch (which is known as an interrupt acknowledge). All other exceptions get the vector number internally from the type of exception. The vector number is used to get a vector address (this is explained shortly).
3. The current CPU information is saved. The Reset exception is the only one that doesn't save CPU status information. The PC and the SR values are put on the Supervisor stack.
4. A new context (CPU information including PC value) is put in place and processing starts at the new address.

The vectors tell the CPU where to go to handle a particular exception. All vectors are two words long, except RESET which is four words long. All vectors are in Supervisor data space except RESET which is in Supervisor program space. The vectors are numbered by byte numbers. These bytes, multiplied by four, give the offset of the exception vector from 0. The exception vector format is shown in Fig. 7-3.

The numbers can be generated internally or externally. For interrupts, some outside device provides a byte vector number on lines D0 through D7. The format of this vector is shown in Fig. 7-4.

The CPU left shifts the vector number two bit positions and puts zeros into the most significant bits. This generates a 32-bit long-word vector offset. For the 68000 and the 68008, this is the actual address (absolute) to find the vector. The address is then truncated to fit the address bus available on the particular CPU. For the 68010, the offset is added to a 32-bit vector base register (VBR) to get the absolute address of the vector. The VBR is shown in the Chapter 8 description of the 68010. For more details, et the 68010 documentation.

When an exception occurs, the CPU needs a routine to handle the exception. The 68000, unlike some CPUs that use one routine for all exceptions (the routine then has to determine what happened) uses different routines for the different exceptions.

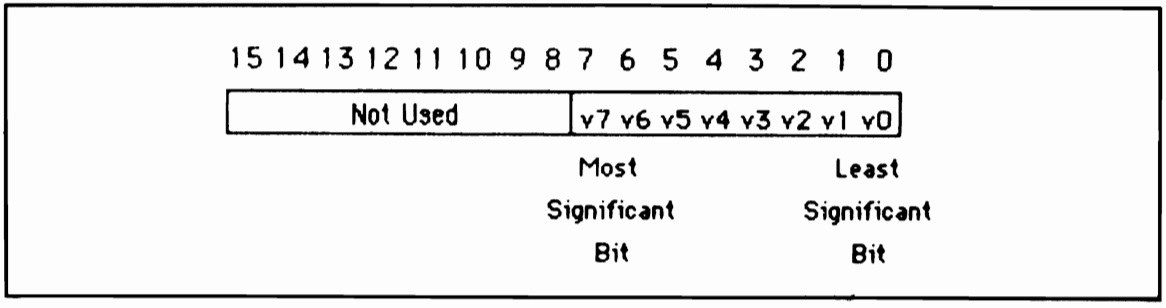


Fig. 7-4. Format of external device vector.

Vector Number	Address (DEC)	Address (HEX)	Type of Exception or interrupt
0	0	000	Reset: Initial SSP
1	4	004	Reset: Intitial PC
2	8	008	Bus error
3	12	00C	Address error
4	16	010	Illegal instruction
5	20	014	Divide by zero
6	24	018	CHK instruction
7	28	01C	TRAPV instruction
8	32	020	Privilege violation
9	36	024	Trace
10	40	028	Line 1010 emulator
11	44	02C	Line 1111 emulator
12	48	030	(Unassigned, reserved)
13	52	034	(Unassigned, reserved)
14	56	038	Format Error (MC68010 only, reserved unassigned on 68000 and 68008)
15	60	03C	Uninitialized Interrupt Vector
16 to 23	64 to 95	040 to 05F	(Unassigned, reserved)
24	96	060	Spurious interrupt (when there is a bus error during interrupt processing)
25	100	064	Level 1 interrupt autovector
26	104	068	Level 2 interrupt autovector
27	108	06C	Level 3 interrupt autovector
28	112	070	Level 4 interrupt autovector
29	116	074	Level 5 interrupt autovector
30	120	078	Level 6 interrupt autovector
31	124	07C	Level 7 interrupt autovector
32 to 47	128 to 191	080 to 0BF	TRAP instruction vectors (#0 through #15, TRAP *n uses vector number 32 + n)
48 to 63	192 to 255	0C0 to 0FF	(Unassigned, but reserved by Motorola for future expansion)
64 to 255	256 to 1023	100 to 3FF	User interrupt vectors

Addresses that are unassigend, but reserved, should not be used for peripherals assignemensts, etc. Motorola may use them in a future version of the chip, and then the peripherals would have to be reassigned to work properly.

Fig. 7-5. Complete table of 68000 exception vectors.

The bottom 1K of memory, 1024 bytes, are reserved specifically for the addresses of all the routines. Each address has a 4-byte chunk of this 1024 bytes, each chunk is called an exception vector. The vectors are numbered (each number being the address divided by four).

Internal traps have implicit vectors, the user cannot choose where they will head. External interrupts that use auto-vectoring are the same. The designed-in circuitry of the CPU and the system decides the vector number.

The full table of exception vectors is shown in Fig. 7-5. This table takes up 1024 bytes of memory and starts at address 0. There are 255 unique vectors, though many are reserved for TRAPS and system functions. 192 vectors are reserved for User interrupt vectors.

Types of Exceptions

There are two main types of exceptions: internal and external. All of these are shown in Fig. 7-6.

Internal exceptions, or traps, come from instructions, address errors, or tracing. Some instructions generate exceptions automatically (ILLEGAL, illegal instructions, TRAP) and some may generate an exception in special circumstances (DIVS or DIVU by zero, CHK, TRAPV). Word fetches from odd addresses and privilege violations on instructions also generate exceptions. Tracing is a high priority internal interrupt after each instruction.

External exceptions, or interrupts, indicate that some outside device wants the CPU's time and attention. Bus errors and reset inputs are also classified as external exceptions.

Exception Priorities

Figure 7-7 shows the priorities of exceptions. Group 0 is the highest priority and Group 2 is the lowest. Group 0 exceptions cause the current instruction to abort; Group 1 exceptions let the current instruction finish before changing to exception processing. Group 2 exceptions take place as part of regular instruction processing. Within Group 0, Reset is highest, bus error next, and address error lowest in priority. Within Group 1 the order of priority from highest to lowest is trace, external interrupts, and then illegal instructions and privilege violations. Group 2 doesn't have to worry about priorities because only one instruction at a time can execute.

If multiple exceptions occur simultaneously, these priority levels determine which is processed first. Just as with interrupt priorities, when a higher priority exception breaks into the processing of a lower priority exception, the higher priority is processed first, and then attention is returned to the lower priority exception.

SUMMARY

There's a lot more to know about exceptions, but you need some programing experience before you can use this 68000 feature. Just as a reminder, though, a few of the specific reasons for exceptions are explained here.

External	Internal
Interrupts	Illegal Instructions
Bus Error	TRAP, TRAPV, CHK instructions
External Reset	Privileged Violations
	Addressing Error
	Tracing
	Division by Zero

Fig. 7-6. Exception types.

Priority Level	Priority Group Number	Exception Types	Timing of the Exception Processing
Highest	0	Reset Address Error Bus Error	Begins within 2 clock cycles
Middle	1	Trace Interrupt Illegal Instruction Privilege Violation	Begins before next instruction
Lowest	2	TRAP,TRAPV,CHK Divide by Zero	Started by normal instruction execution

Fig. 7-7. Exception priorities.

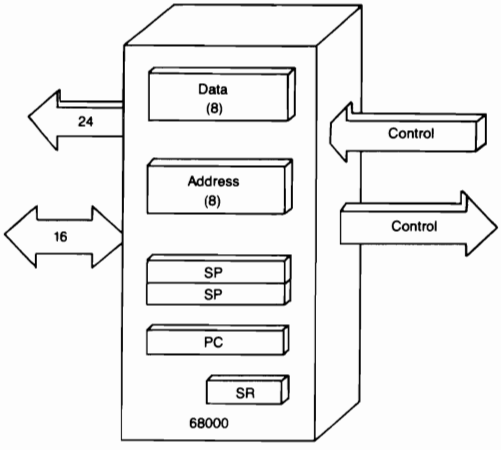
1. RESET lets the processor be started from scratch. This is used on every microprocessor to escape from such problems as endless loops.

2. Illegal instruction exceptions protect program execution from trying to execute something that is not an instruction. All of the object codes that haven't been implemented (and the ILLEGAL instruction itself) are in this category. Because of the vector address scheme, new instructions can be added in software emulation. That is, if you want a new instruction, just specify an unused object code for it and put the routine that will be the instruction in the right part of memory (where the vector will find it).

3. Tracing allows the programmer to slow execution and see what happens after every instruction. This is vital to debugging.

4. CHK lets the programmer keep instruction work within certain bounds. This helps implement the data type and size protection available in some high-level languages.

8



The 68000 Family

THE 68000 IS A FAMILY OF CHIPS, NOT A LONE microprocessor. This family includes several different microprocessors and a large number of peripheral chips. If you want to use the true power of the CPU, you have to unbundle many system tasks and assign them to other specialized chips.

The 68000 family of chips breaks into three natural divisions. The first category is the CPUs themselves. Choosing the right chip from this set can save money and design time. The standard 68000 is not ideal for every system. The next category is the 6800 support chips. These 8-bit chips can be used with the 6800 microprocessor or with the 68000 (any of them will interface directly with the 68000). The last group is the 68000 support chips. These are dedicated to supporting the CPU, although some of them are as complicated, or even more complicated, than the CPU itself. Figure 8-1 lists the 68000 family.

Although Motorola invented the 68000, there are other companies that make the chip. These companies, licensed by Motorola, are called *second sources*. Designers don't like to work with chips that only come from a single supplier; they are afraid of what will happen if that supplier goes out of business or has some production problems. Second sources don't necessarily make all of Motorola's 68000 peripheral chips of their own.

The 68000 second sources are as follows:

Hitachi Ltd.	Rockwell International
Mostek Corp.	Signetics
Philips	Thompson EFCIS

These are true partnerships with Motorola with exchange of masks and joint product development. (The masks are the actual patterns used to put the transistors on a chip.)

Figure 8-2 shows a complete 68000-based system. Although you will never see a real system that includes all of these peripheral devices, this illustration depicts where they would attach to the buses.

This chapter will describe the following:

1. All of the 68000 CPUs. There are only four

Chip	Acronym	Description
68000	CPU	16-bit Microprocessor
68008	CPU	Reduced-bus (8-bit) Microprocessor
68010	CPU	16-bit Virtual-memory Microprocessor
68020	CPU	32-bit Virtual-memory Microprocessor with Cache
68120	IPC	Intelligent Peripheral Controller
68121	IPC-NR	Intelligent Peripheral Controller with No ROM
68122	CTC	Cluster Terminal Controller
68153	BIM	Bus Interrupt Module
68172	E-BUSCON	VME Bus Controller
68173	S-BUSCON	VME Bus Controller
68174	E-BAM	VME Bus Arbitration Module
68200	MCU	Micro-computer Unit
68230	PI/T	Parallel Interface/Timer
68340	DPR	Dual Port RAM
68341	IEEE FP	IEEE Floating Point (Software Package - M68KFPS)
68342	RTE	Real Time Executive (Software Package)
68343	FFP	Fast Floating Point (Software Package - M68KFFP)
68345	FIFO	First-in/First-out
68430	DMAI	Direct Memory Access Interface
68440	DDMA	Dual Direct Memory Access
68450	DMAC	Direct Memory Access Controller
68451	MMU	Memory Management Unit
68452	BAM	Bus Arbitration Module
68454	IMDC	Intelligent Multiple Disk Controller
68459	DPLL	Disk Phase-Locked-Loop
68465	FDC	Floppy Disk Controller
68485	RMC	Raster Memory Controller
68486	RMI	Raster Memory Interface
68561	MPCC-II	Multi-protocol Communication Controller II
68562	DUSCC	Dual Universal Serial Communications Controller
68564	SIO	Serial Input/Output
68590	LANCE	Local Area Network Controller
68605	SDMA	Serial Direct Memory Access
68652	MPCC	Multi-protocol Communications Controller
68653	PGC	Polynomial Generator Checker
68661	EPCI	Enhanced Programmable Communications Interface
68681	DUART	Dual Universal Asynchronous Receiver/Transmitter
68802	LAN-802.3	Local Area Network (IEEE 802.3 Standard)
68851	PMMU	Paged Memory Management Unit
68881	FPCP	Floating Point Co-processor
68901	MFP	Multifunction Peripheral
68920	MAC	Memory Access Controller

Fig. 8-1. Chips in the 68000 family.

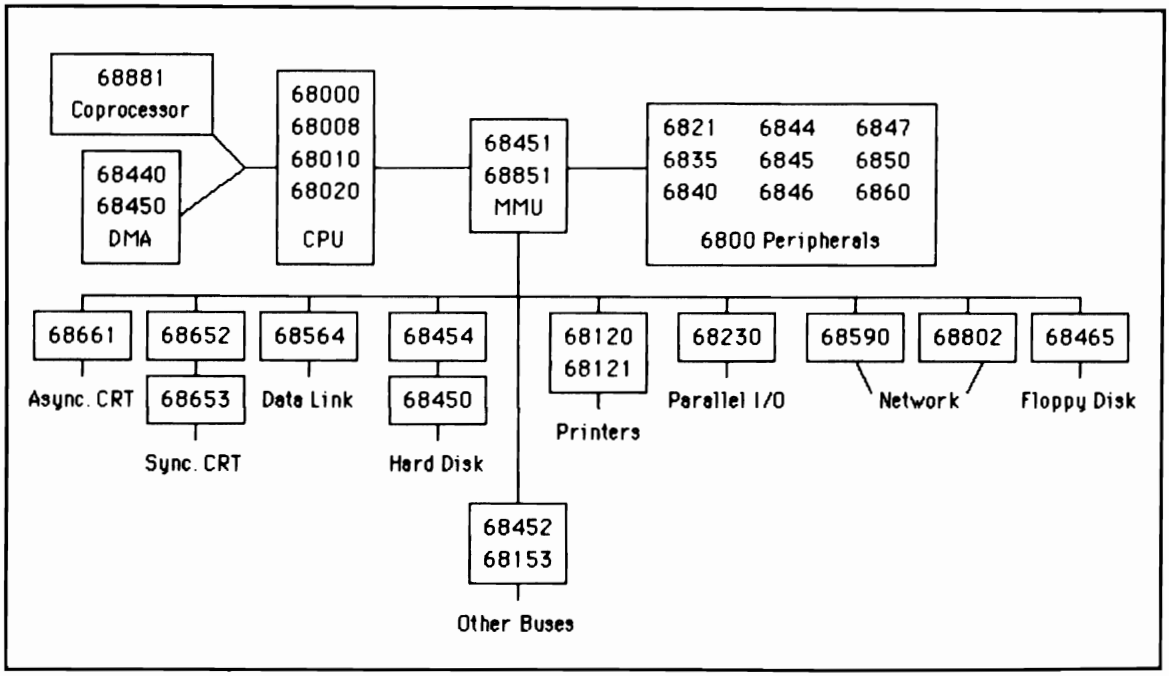


Fig. 8-2. Block diagram of a system built around the 68000 family.

major members of this group: 68008, 68000, 68010, and 68020. The 68200 is a special case. It is related to the 68000, but it is not directly compatible.

2. Some of the 68000 peripheral chips. There are many of these chips; describing them all would require another book. Also, because new chips are introduced all the time from every 68000 manufacturer, only a few chips are described here. Those should be enough to give you a feel for the uses and applications of peripheral chips.

3. None of the 6800 peripheral chips. There are also quite a few of these chips, and they have all been described before in many books about the 6800 microprocessor.

CPU CHIPS

The CPUs of the 68000 family are shown in Fig. 8-3. They differ in a number of respects yet they are all built around the same architecture and have a great deal of compatibility. Which one is used in a system will depend on how much power the system needs, and how much it can afford.

68000

The standard of the family is the 68000. It was the first chip of the family, introduced in 1979. As demonstrated by Fig. 8-4, you have choices to make even if you opt for the 68000. Besides having to decide on a package material (which is true when you buy many chips) you have to choose a speed.

The "L" of the chip number indicates a ceramic package; different letters are used for the other packages. G or Y stands for a plastic DIP package (with 64 pins), ZB stands for Type B Leadless Chip Carrier, and ZC or Z stands for a Hi-rel Type C Leadless Chip Carrier. Leadless Chip Carriers are square and take up much less circuit board space than traditional DIP packages.

The digits at the end of the chip number indicates the speed of the chip. Increased clock frequency means a shorter clock period (time for the clock to run through one cycle). That also means less time spent on each instruction, and thus, faster program execution.

Chips with a faster clock may work through programs faster, but they will also cost more than

68000	16-bit Microprocessor
68008	Reduced-bus (8-bit) Microprocessor
68010	16-bit Virtual Memory Microprocessor
68020	32-bit Virtual Memory Microprocessor
68200	Micro-computer Unit

Fig. 8-3. 68000 CPUs.

slower chips. In addition, for faster chips to be able to use their speed, other components may also have to be chosen for speed (and will therefore be more expensive). For instance, in many systems a change to a fast microprocessor may require a change to faster memory chips (chips that can be read or written to in less time) for system speed to actually increase. If you specify faster chips for all the

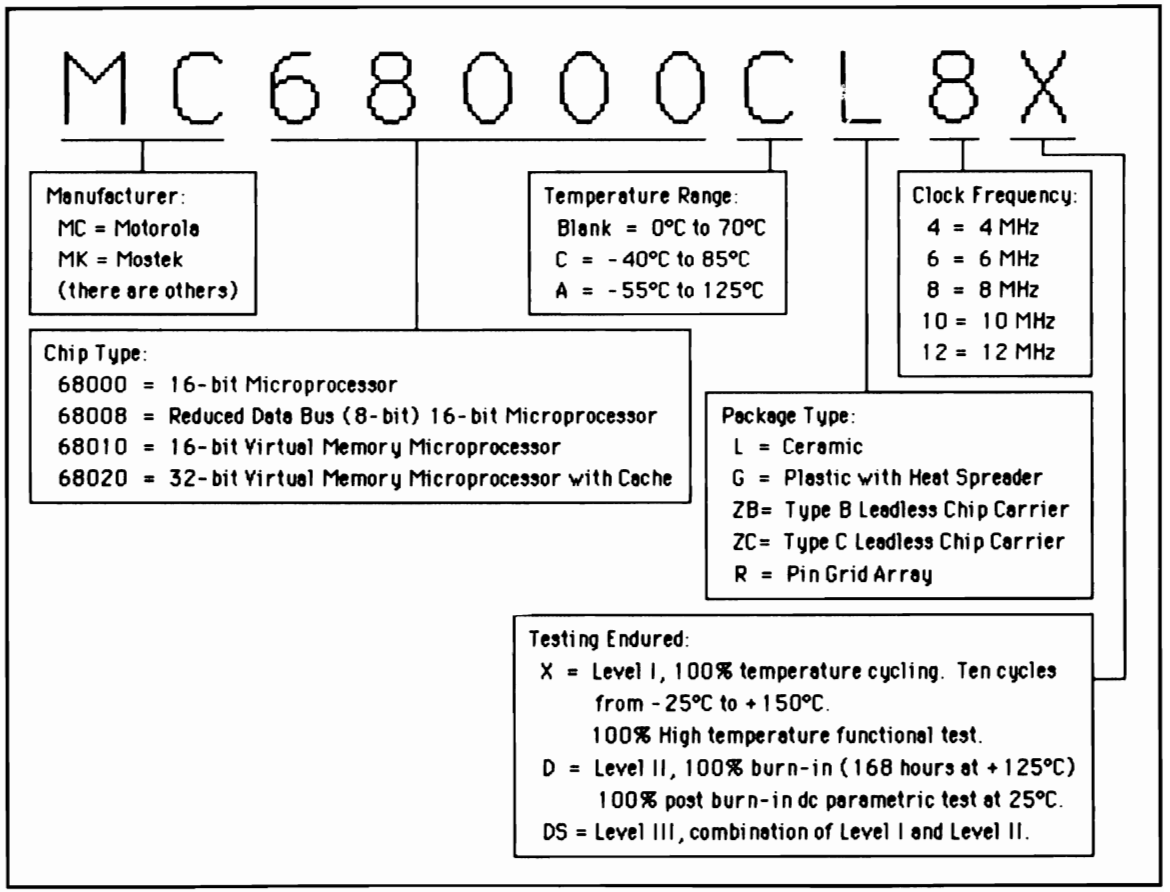


Fig. 8-4. CPU specification codes—package and speed.

positions, system design may become more difficult because problems of line isolation and spurious signals may worsen.

So, in most cases you cannot just plug a faster 68000 in and watch a computer fly. Nevertheless, you do have a choice of 68000 speeds ranging from the 68000L4 which runs at 4 MHz to the 68000L12 which flies at 12.5 MHz.

Theoretically, a system with a 68000L12 (and all the support chips necessary) will run a program more than three times faster than a system with a 68000L4. However, since many programs are I/O bound—that is, most of the program time is spent waiting for the human operator to enter or read data—the increased speed may not even appear.

The 68000 will not be covered in any more detail here: the rest of this book takes care of that task.

68008

The 68008 is completely code compatible with the 68000. Programs written for either one will generally run on the other. The programmer only needs to understand a few minor software differences—such as the limited interrupt priorities and changed memory organization of the 68008—to quickly adapt programs written on the 68000 for use on the 68008. Programs written for the 68008 will run without any modification at all on the 68000.

The 68008 has a different memory organization, as shown in Fig. 8-5. The register and instruction set of the 68008 are the same as on the 68000. Because the two chips are so familiar, and the rest of this book is devoted to the 68000, only the functional differences between the chips are described in this chapter.

68008 vs. 8-Bit CPUs. The main difference between the 68008 and the 68000 is that the 68008 has a data-bus that is only 8 bits wide (which is why it has the "8" on the end of the number). The standard 68000 has a 16-bit data bus. To provide the 16-bit chip (with a 32-bit internal structure) advantages for 8-bit system designers, the 68008 was invented. The narrow data-bus allows simpler printed circuits than 16-bit bus allows. 8-bit systems are typically simpler and therefore less expensive than 16-bit systems. The 68008 comes packaged in a 48-pin DIP instead of the 64-pin DIP of the 68000.

Why is an 8-bit wide data bus cheaper to work with than a 16-bit wide data bus? Because systems costs are reduced. Byte-wide memories and peripheral components (chips that address a full byte of memory at a time instead of a single bit can

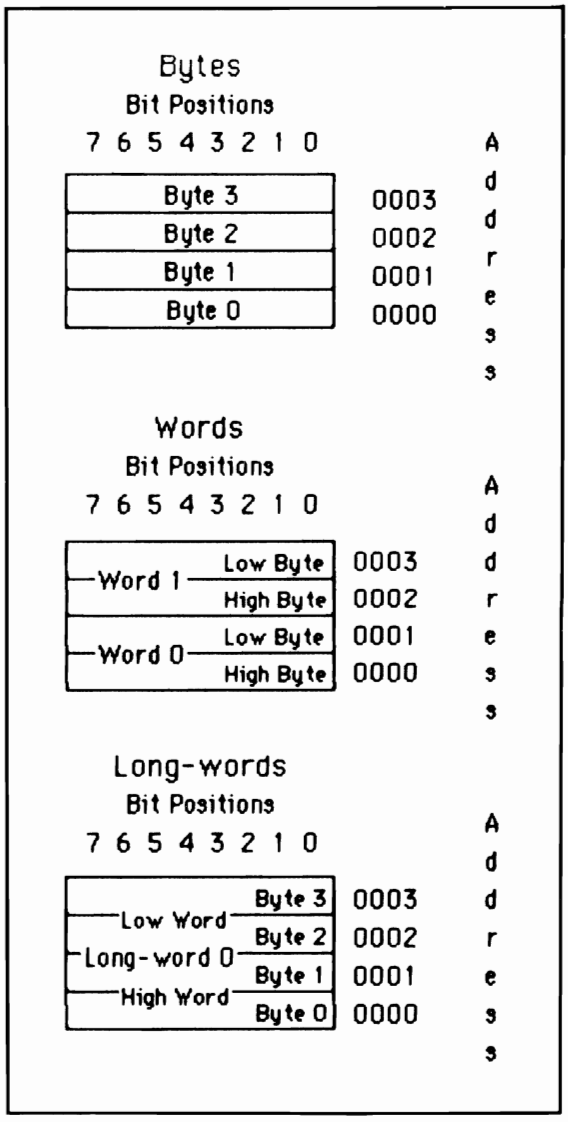


Fig. 8-5. 68008 memory organization.

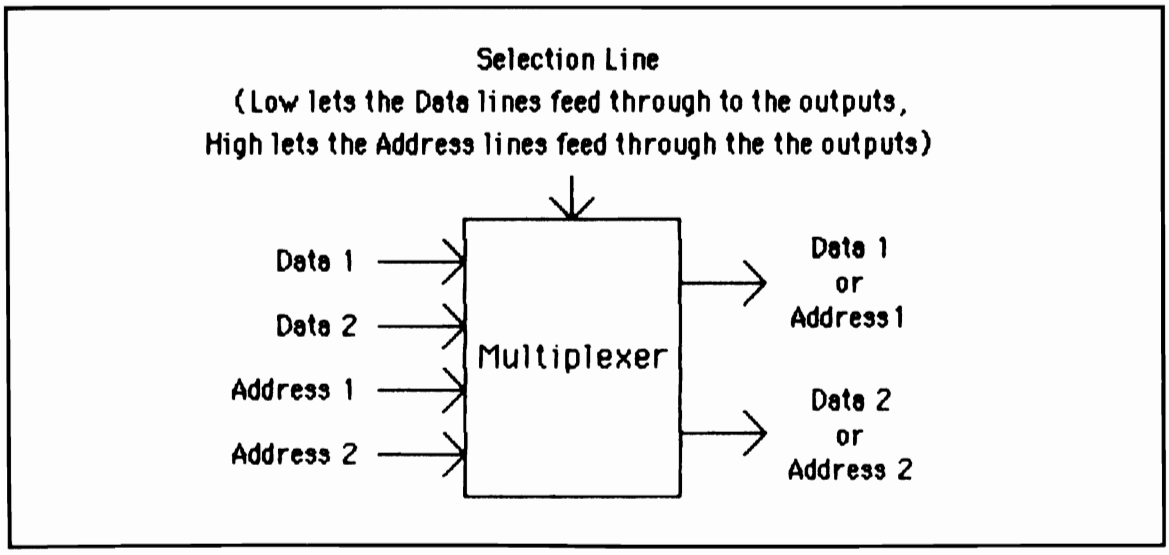


Fig. 8-6. An example of multiplexing.

be used. This decreases the chip cost associated with the memory portion of the system.

Why use the 68008 instead of some 8-bit processor? Because the 68008 retains all of the power of the 68000 instructions, addressing modes, and internal architecture. This, in effect, lets you use a 32-bit architecture in what was formerly 8-bit territory. Only the external data-bus is narrowed to 8 bits.

Some microprocessors use multiplexed buses to save pins on the chip. Multiplexing allows more signals to use a limited number of lines, as shown in Fig. 8-6. The disadvantage of multiplexing is that it requires extra chips on the circuit board to demultiplex the signal lines. The 68008 8-bit data-bus is not multiplexed, nor is the address bus.

68008-68000 Compatibility. Because the 68008 object code is directly compatible with that of the 68000, any software written for the 68000 will execute on the 68008 (and vice-versa). Having the 68008 (an 8-bit data-bus CPU), the 68000 and 68010, (16-bit data buses), and the 68020 (32-bit data bus), allows the programmer to learn a single assembly language which can be used for systems ranging from simple controllers to super-minicomputers.

Memory Addressing and Data Organization. The 68008 can address 1 megabyte of nonsegmented linear address space. That means it can directly reach for information in up to 1 megabyte of memory. Segmenting is a technique of adding extra address bits to determine a chunk of memory within which the actual address will be used to find the data. Many microprocessors that claim to be able to access a large memory space (such as 1 megabyte) must use segmenting to do so. You can segment if you want to with the 68008, and the segmenting can be quite flexible. You can let each program use segments of different lengths, whatever is most efficient for that program. Other microprocessors force you (the programmer) to use a set segmenting scheme.

Data organization is a bit different with the 68008 than with the 68000 because the 68008 can only move a single byte at a time on its 8-bit data bus. The 68000 individually addresses bytes with the high-order bytes having an even address (the same as the addressed word). This gives the low-order byte an odd address one count higher than the word address. Another facet of 68000 addressing is that multibyte data is only addressed on word boundaries (the 68020, on the other hand, is not limited to this).

The 68008 fetches a pair of bytes, or a word,

at a time to ensure compatibility with the 68000 which fetches words. The 68000 addresses data as shown in Fig. 8-5. Instructions always start on a word boundary to keep compatibility with the 68000. Function codes are used to indicate the address space being accessed during a bus cycle. Bits are specified from bit 0 to bit 7 (the high bit) within a byte. Bytes are addressed in order in memory. Words are addressed with the most significant byte at the lower address and the least significant byte one position higher in memory. Long-words are addressed with the high-order word first (lower in memory) and the low-order word last (at a higher point in memory) and the division within the words within the long word remains that same as just described. This may sound complicated, but once you inspect the illustration and use the 68008 a few times, it will be second nature.

The Chip and Its Pinout. Figure 8-7 shows the signals for the 68008 chip. These are, of course, somewhat different from the 68000 because of the change of bus sizes. The following list describes the similarities and differences between 68000 and 68008 signals. This information is only presented for the interest of those who will have personal computers built around the 68008; it is not vital to programming.

The physical pinout of the 68008 is shown in Fig. 8-8. The 68008 comes in a standard DIP package but has only 48 pins compared to the 64 of the 68000. The 68008 comes in versions that run at 8, 10, or 12.5 megahertz.

Address Bus. The address bus (A0-A19) is only 20 lines wide instead of the 23 (A1 through A23) on the 68000. As with the 68000, this bus provides the address for bus operations during all cycles except interrupt acknowledge cycles. Then, lines A1, A2, and A3 provide the level of the interrupt (just

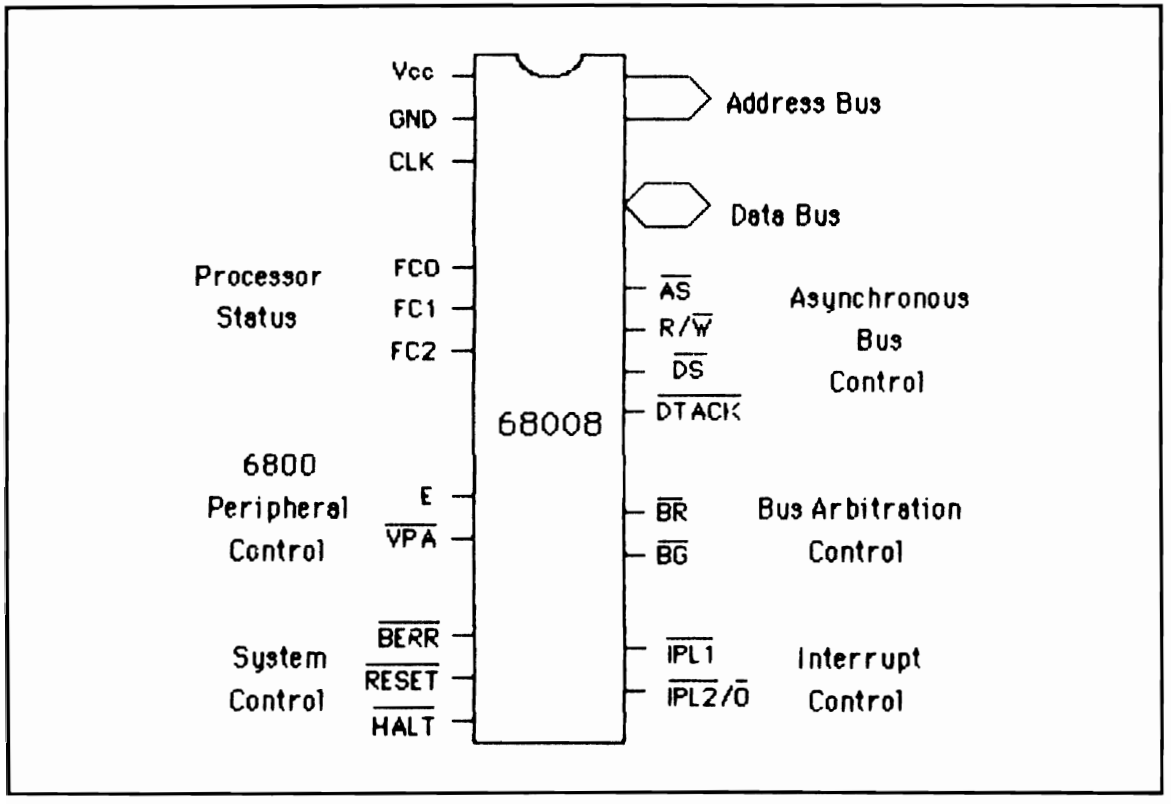


Fig. 8-7. 68008 pinout (functional).

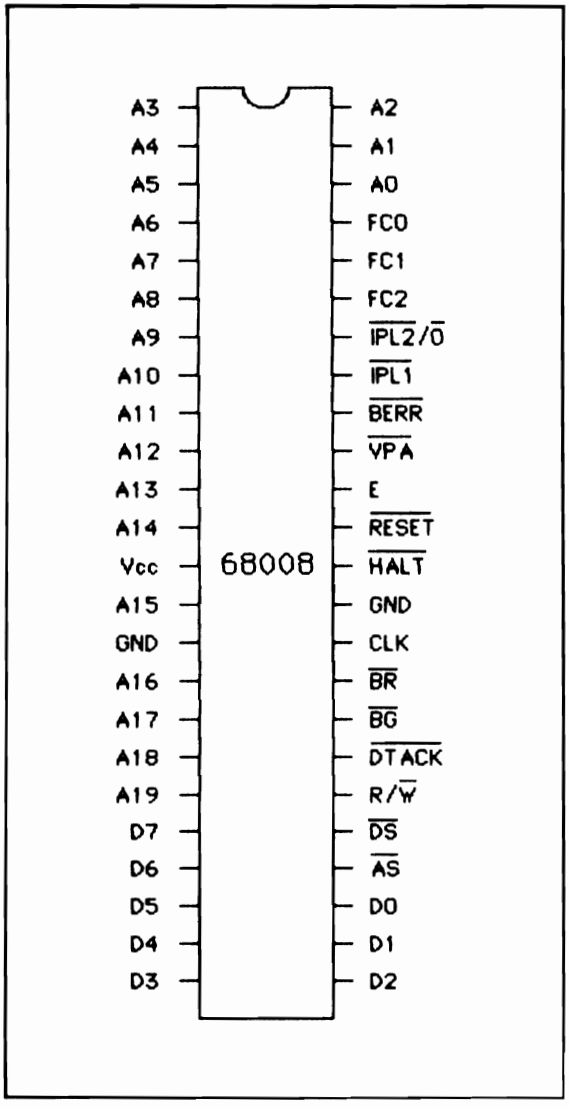


Fig. 8-8. 68008 pinout (assignments).

as on the 68000) and lines A0, and A4 through A19 are all set high (on the 68000 lines A4 through A23 are all set high).

Data Bus. The data bus (D0-D7) is only 8 bits wide instead of the 16-bits (D0-D15) of the 68000. As with the 68000, during interrupt acknowledgements, the interrupt vector is supplied to the CPU by the interrupter on lines D0-D7.

Asynchronous Bus Control. Asynchronous bus control changes slightly from the 68000 to the 68008. The 68000 has address strobe, read/write, upper data strobe, lower data strobe, and data transfer acknowledge. The 68008's difference is that it has only a single data strobe signal instead of the upper and lower data strobes.

Bus Arbitration Control. Bus arbitration control uses only two signals instead of the three (Bus Request, Bus Grant, and Bus Grant Acknowledge) on the 68000. The 68008 uses Bus Request and Bus Grant. These handle all daisy-chained networks, priority encoded networks, or combinations. The BR is wire ORed with all other devices that could control the bus. Those devices may ask for the bus at any time. BG indicates to all other bus controllers that the CPU will release control of the bus at the end of the current bus cycle. The only time BG cannot be issued is during the two clock interval between the transition of AS from inactive to active. When a 68008 is put into a 68000 system the BR and BGACK signals should be ANDed and then connected to BR.

Interrupt Control. A device requesting an interrupt uses these pins to indicate to the CPU the priority of the interrupt. The 6800 uses three pins to accept these signals, (IPL0bar, IPL1bar, and IPL2bar); the 68008 uses only two, ($\overline{\text{IPL0}}$/$\overline{\text{IPL2}}$ and $\overline{\text{IPL1}}$). The 68000 thus can handle seven levels of priority (a zero on the three pins indicates no interrupt with level seven not being maskable.) For the interrupt request to be acknowledged, the priority must be greater than the contents of the processor status register interrupt level. The 68008 attaches the $\overline{\text{IPL0}}$/$\overline{\text{IPL2}}$ pin to both $\overline{\text{IPL0}}$ and $\overline{\text{IPL2}}$ internally and therefore can only fit values of 0, 2, 5, and 7. Level seven, as in the 68000, is a nonmaskable edge-triggered interrupt. IPL0bar is the least significant bit and $\overline{\text{IPL2}}$ is the most significant bit. The level must be less than or equal to the processor status register level for two successive clocks before triggering an internal interrupt request. Interrupt acknowledgement is made by all of the function code lines (FC0-FC2) going high.

System Control. System control is accomplished in the same way as on the 68000. BERR is used for Bus Errors, RESET is used to reset the processor from an external signal, and HALT is us-

ed to stop the processor after the current bus cycle.

M6800 Peripheral Control. Peripheral control differs a little from the 68000 method. This is the interfacing of synchronous peripherals to the asynchronous MC68000. The 68000 supplies a valid memory address ($\overline{\text{VMA}}$) signal, but the 68008 does not. This signal (on the 68000) tells the peripherals that a valid address is on the bus and that the CPU is synchronized to the enable clock. When using the 68008, this signal can be generated externally (outside the CPU chip).

Processor Status. The processor status (FC0, RC1, FC2) signals are the same on both the 68000 and the 68008.

There are other differences between the chips that involve things already mentioned. For instance, the 68000 must use an internal A0 signal to determine which byte to grab from memory when the instruction specifies a byte operation, read or write. The 68008 has an external A0 for that job.

That is the extent of the signal differences between the 68008 and the 68000. Most of what you just read, except the Interrupt priority discussion, is hardware stuff that a programmer doesn't need to remember.

Exception Processing. The 68000 and the 68008 do differ slightly in exception processing because, among other things, the 68008 cannot recognize the full seven levels of interrupt priority. Because of the limited pins (as explained above in the "Interrupt Control" section) the 68008 can only work levels 0, 2, 5, and 7 of the 68000's interrupt priorities. So 68000 programs that depend on full ordering of interrupt priorities will have to make some accommodations to run on the 68008.

Summary. Basically, having the 68008 to work with means you can use the powerful 68000 architecture and assembly language even in smaller and cheaper systems. You can apply the same knowledge, and even the same programs, to a wider variety of problems.

68010

The 68010 is an improved and more powerful 68000 CPU. It is completely program compatible with the 68000. The 68010 uses the same addressing modes, registers, buses, package, and instructions as the 68000. What it adds is new exception processing power and Virtual Memory (in fact, it is called the 16-bit Virtual Memory Microprocessor).

Figure 8-9 shows the 68010 register set. The new Vector Base Register and the Alternate Function Code Registers, along with the new instructions, protect Supervisor mode from User mode and allow the 68010 chip to understand when virtual memory operations are necessary.

Virtual memory is a common technique in mainframe and minicomputers because it allows the computer to access a much larger address area than is built into the memory chips. The virtual memory information is kept in a larger storage medium, such as a disk. The microprocessor keeps track of what information is loaded into the memory chips. When an address that is outside the presently stored area is called for by an instruction, a page fault has occurred. That page fault stops the execution, loads the necessary information from the disk into the memory chips, and then continues the instruction execution—all without letting the programmer know that such a special memory operation was necessary. The 68010 uses hardware—including new registers and stack controls—to run virtual memory.

You can program a 68010 system just as if it had a 68000 (as long as you realize that a few instructions have been modified—see the descriptions below). If you want to understand virtual memory operations, look for a general computer science textbook. To understand the action of the new 68010 registers, which are outside the territory of this book, read the manufacturer's original manuals such as, *M68000: 16/32-bit Microprocessor Programmers Reference Manual,* from Motorola.

New 68010 Instructions. The following pages describe the new 68010 instructions in the same manner that Chapter 6 of this book describes the 68000 instructions. The 68010 uses the entire set of 68000 instructions (though a few are modified) and adds these instructions. You'll need to know differences if you work on a 68010 machine.

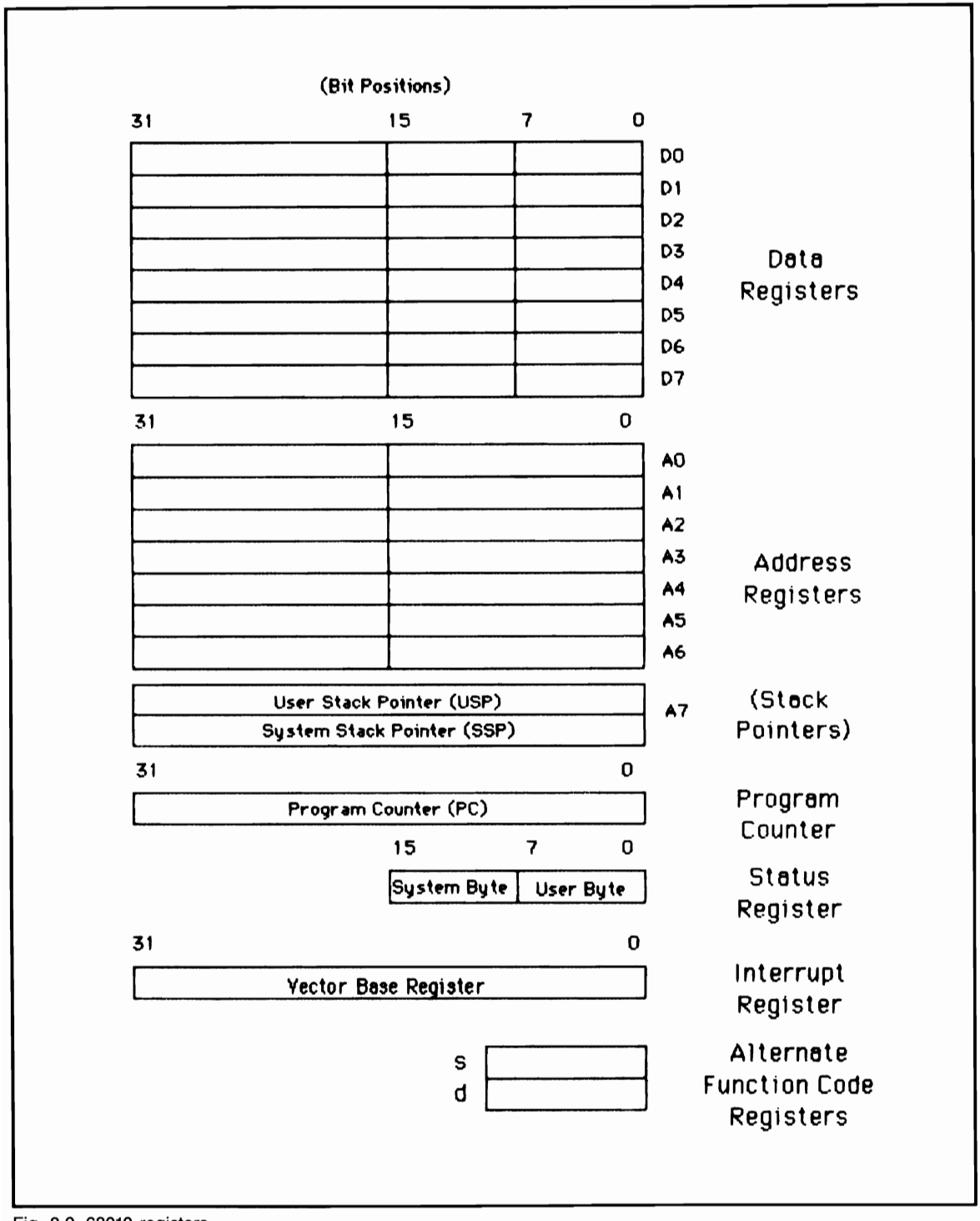


Fig. 8-9. 68010 registers.

MOVE from CCR

Definition: move from condition code register (68010).
Description: MOVE from CCR is a special case of the MOVE instruction and is a 68010 instruction. It moves the contents from the low byte (the condition code register) of the status register to the specified destination.
Addressing: The source is always the CCR. The destination can be reached by any of these following modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100001011aaabbb
Breakdown
0100001011: MOVE from CCR instruction.
aaa: Destination addressing mode.
bbb: Destination addressing register number.

MOVE from SR

Definition: move from the status register (68010).
Description: MOVE from SR is a special case of the MOVE instruction. It moves the contents of the status register to the specified destination. The 68010 version of MOVE from SR is the same as the 68000 version except that it is a privileged instruction. If the CPU tries to execute it while in User state, a TRAP will be generated.
Addressing: The source is always the status register. The destination can be reached by any of the following modes:

Data Register Direct
Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

Operand size: word.
Instruction length: 1 word.
Condition code effects: none.
Object code: 0100000011aaabbb
Breakdown
0100000011: MOVE from SR instruction.
aaa: Destination addressing mode.
bbb: Destination addressing register number.

MOVEC

Definition: move control register (68010).
Description: MOVEC is a special case of the MOVE instruction that is only implemented on the 68010, not the 68000. It moves data either from a specified control register to a specified general register, or from a specified general register to the specified control register.

The data is copied, so the source contents aren't changed by this instruction. Even if the control register doesn't use a full 32 bits, the transfer is always of a long-word. The unused bits are read as zeros.

Addressing: The general register (which functions as source or destination) is specified by either Data Register Direct mode (if it is an address register) or Address Register Direct mode (if it is an address register).
Operand size: long-word.
Instruction length: 2 words.
Condition code effects: none
Object code: First word
(010011100111101a)
Second word (bcccddddddddddddd)
Breakdown
010011100111101: MOVEC instruction.
a: SPECIFIES the transfer direction.
0 means control to general.
1 means general to control.
b: Specifies the type of the general register.

0 means a data register.
1 means an address register.
ccc: Specifies the general register number.
ddddddddddddd: Specifies the control register.
000000000000 means the source function code (SFC) register.
000000000001 means the destination function code (DFC) register.
100000000000 means the User stack pointer.
100000000001 means the vector base register for the exception vector table.

Note: Any code other than those shown for the control register specification will force an illegal instruction exception.

MOVES

Definition: move address space (68010).

Description: MOVES is a privileged instruction that is found on the 68010 and not the 68000. It moves a byte, word, or long-word between a general register and a memory location.

Addressing: The memory location is addressed by the SFC (source function code) register—if it is the source—or the DFC (destination function code) register—if it is the destination. The general register is specified by any of the following addressing modes:

Address Register Indirect
Postincrement Register Indirect
Predecrement Register Indirect
Register Indirect with Displacement
Register Indirect with Index
Absolute Short
Absolute Long

If the destination general register is a data register, the operand replaces the low-order bits (and doesn't affect the higher bits). If the destination general register is an address register, the operand is sign-extended to 32-bits and then is put into the register.

Operand size: byte, word. or long-word.

Instruction length: 2 word.

Condition code effects: none.

Object code: First word (00001110aabbbbbb)
Second word (cdddde00000000000)
Breakdown
00001110: MOVES instruction.
aa: Specifies the operand size.
00 means byte
01 means word
10 means long-word
bbbbbb: Specifies the effective address of the memory location.
c: Specifies the type of general register used.
0 means a data register.
1 means an address register.
ddd: Register number.
e: Specifies the transfer direction.
0 means from memory to the general register.
1 means from the general register to memory.

RTD

Definition: return and deallocate parameters (68010).

Description: A long-word is pulled off of the stack and put into the PC (program counter). The previous PC value is lost. A word displacement value—which is the second word of the instruction—is then sign-extended to a full 32 bits and added to the stack pointer. This instruction doesn't work on the 68000; it is only a 68010 instruction.

Addressing: none.

Operand size: not applicable.

Instruction length: 2 words.

Condition code effects: none.

Object code: First word (0100111001110100)
Second word (displacement)
Breakdown
0100111001110100: RDE instruction
displacement: This two's complement value is sign extended to 32 bits and then added to the stack pointer.

RTE

Definition: return from exception (privileged) (68010).

Descrition: RTE loads the SR (status register) and then the PC (program counter) from the system stack. The first word pulled off the Supervisor stack is put into the SR; the second and third words

pulled are put into the PC (the second becomes the high word and the third the low word). The previous SR and PC values are lost. This is typically the last instruction executed in an exception processing service routine. RTE is a privileged instruction and so will only execute when the CPU is in Supervisor state. The new state of the CPU will depend on the values put into the Status Register. The bits of the status register that have not yet been assigned values will always retain a 0. RTE is slightly different on the 682010 than on the 68000. The 68010 RTE also pulls the vector offset from the stack and then examines the format field to see how much data is to be restored (see the note below for details).
Operand size: none.
Instruction length: 1 word.
Condition code effects: Set directly from the value pulled off the stack and stored in the status register.
Object Code: 0100111001110011

Breakdown

0100111001110011: RTE instruction.

The vector offset word has 10 bits of vector offset (bits 0 through 9), 2 bits that don't change (both bit 10 and bit 11 are always 0), and 4 bits of format (this is the field: bits 12 through 15). The format field specifies the amount of data to restore (to pull from the top of the stack):

0000 means to short restore. 4 words are restored.
1000 means to long restore. 29 words are restored.

Any other pattern means the CPU will take the format error exception.

68020

The 68020 was formally announced by Motorola on June 28, 1984. It took 60 person-years to design it and it is supposed to offer four times the power of the 68010. The only full 32-bit microprocessor to precede it onto the commercial market was the 32032 from National Semiconductor. Motorola disputes this, claiming its chip is the first true 32-bit chip around. Other 32-bit firms, though, including Hewlett-Packard with its HP Focus CPU and Western Electric with the Bellmac 32 had been producing 32-bit processors for their own machines (but not for sale to other manufacturers)

The 68020, then, was the first commercial, 32-bit, upwardly-compatible chip from a major microprocessor maker. Intel had yet to weigh in with its 80386 entry (although the iAPX 432 had been announced several years before, it had an unusual design and had affected future microprocessor designers more than it had gained commercial acceptance).

Having a particular microprocessor included in the design of a new product is called a *design win.* A *big win* is a product that will lead the market either in technological complexity or in volume of sales. The 68020, by virtue of its compatibility and its position as an early entrant, will clearly have a large number of design wins. In fact, Motorola circulated most of the specifications of the 68020 for at least a year before the chip was available. Between that, and working directly with important customers to let them know of the chip's development progress, Motorola was able to use the feedback to make a better chip and to announce a chip that was already in several computers that were about to hit the market.

Why use the 68020, and why is it important? All you really need to know, as a software designer, is that the 68020 is a superset of the 68010. Once you learn 68000 code, you can write programs for the 68020, a chip that will be used well into the 1990s. That means your 68000 knowledge is guaranteed to have a future. Second, the 68020 is more powerful than the 68000 or 68010 with more instructions, more addressing modes, and greater speed.

The 68020 is Motorola's most advanced microprocessor. It is completely 32-bit, that is, the internal and external data and address paths (or buses) are 32-bits and are not multiplexed—a different wire is dedicated to each signal. That means no time is lost decoding whether a line is supposed to carry addresses or data. The registers are all 32-bit wide, the ALUs are 32-bits wide, and the program counters and stack pointers are also 32-bits wide.

HCMOS Technology. Motorola makes the

68020 using a 2-micron HCMOS process. 2-micron is a way of referring to the design rules for the smallest features on the chip. A micron is a millionth of a meter (a micrometer). That isn't the smallest design rule, some chips are now made using 1 micron design rules. Smaller design rules mean more transistors can be fitted into a smaller space. Theoretically, ignoring power buses and other real-world factors, a 1-micron design rule chip could hold 4 times as many transistors as a 2-micron design rule chip. Conversely, a 2-micron design rule chip will be easier to make than a 1-micron design rule chip. Larger features means higher yields from the wafers and lower prices for the final chip.

With any chip as complex as the 68020, density (getting the maximum number of transistors onto the chip), chip size (keeping the actual chip as small as possible so that the number of working devices per processed wafer are high), speed (keep it as high as possible by putting everything on a single wafer and yet use a technology to make the transistors that can work at a high frequency), power consumption (as low as possible so heat won't be a problem and the chip can be used in systems with smaller, cheaper power supplies), and manufacturability (make it easy to make by using transistors as large as possible, and techniques that are proven) are vital factors.

There are many ways to make the tiny transistors on a silicon wafer. PMOS was one of the first used (it stands for P-channel Metal-Oxide-Semiconductor). Then NMOS (N-channel MOS) became more prevalent because it allowed more transistors in the same area. Finally in the 1980s, CMOS, a technology of making transistors that had been invented years before at RCA, became the most popular.

CMOS, which stands for Complementary MOS, uses very little power compared to PMOS and NMOS. It has the disadvantages that it takes up more space and the circuits it makes are traditionally slower than NMOS. However, when very small design rules are used, it was discovered that CMOS can work pretty quickly. Because it uses little power, it doesn't heat up the chip as much as the other techniques. Heat becomes a major barrier to larger, more complex chips, because all those transistors are giving off heat and when the temperature of a chip rises the chip becomes more likely to make mistakes and, eventually, stop working.

HCMOS is an advanced type of CMOS process (High-performance and High-density CMOS). This mixture of NMOS and CMOS (90% CMOS with NMOS for critical circuits) combines the advantages of low-power operation with high speed.

Chip Geography. The 68020 has 200,000 transistors on a chip of silicon that is 375 by 350 mils (thousandths of an inch), or about 3/8 of an inch square. The 68000 has 70,000 transistors. The 68020 transistors are divided up into the functional regions you see in Fig. 8-10. It uses the 68010 as a core (subset) but has a 32-bit barrel shifter which by itself has more transistors than the entire 6800 microprocessor. The barrel shifter speeds execution of shifts, multiplications, divisions, and other instructions.

The Execution Unit is made up of three parts: the program counter section, the address section, and the data section. The program counter section calculates instruction addresses and maintains instruction stream pointers. The address section calculates operand addresses and stores the User-visible address register set. The data section performs all the data operations and also contains the User-visible data register set, a barrel shifter, and elements of the instruction pipe.

The μROM (microROM) and nROM (nanoROM) are a modified 2-level control store. That is, the μROM is a permanent memory that holds the information needed for decoding the 68000 instructions. The nROM is a permanent memory that holds the information needed to decode the μROM instructions.

At this point, no further ROMs are necessary; the instructions in the nROM are just directly implemented in hardware. If the nROM and the μROM were given different values to permanently store, the instruction set of the 68000 would be different. This approach to microprocessor design allows the designer to fix bugs, add features, and improve the chip without completely redesigning it. In other words, the μROM controls the sequence of actions

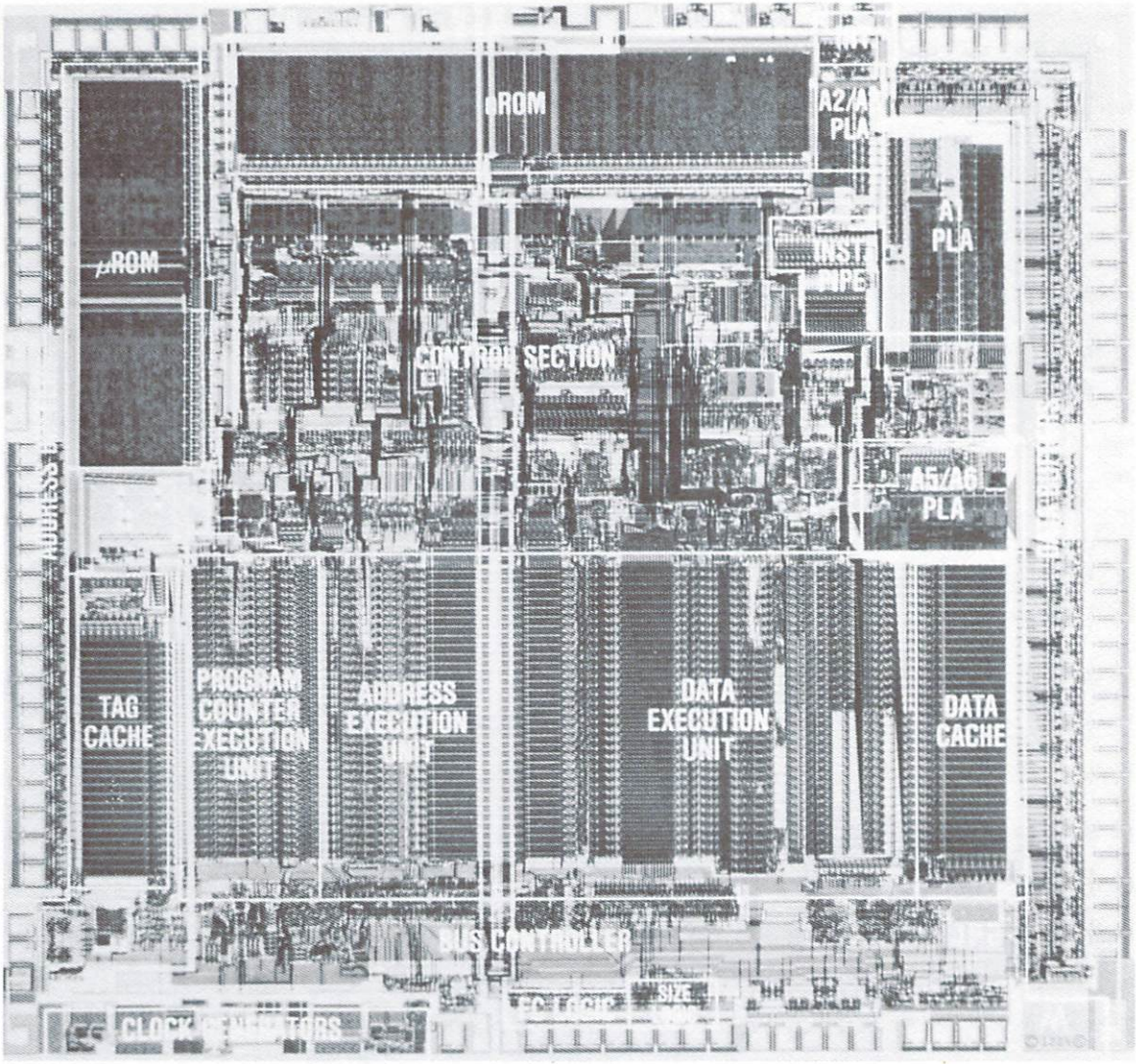


Fig. 8-10. 68020 floor plan (courtesy of Motorola).

the bus controller and the micromachine (of the microprocessor) make to carry out machine language instructions. The nROM controls the operation of the micromachine.

The Instruction Decode Unit decodes the instructions. Within this unit, the A1 PLA makes the initial decoding. This section determines if the instruction is legal and provides the initial microaddress. The A2/A3 PLA generates the rest of the microaddresses necessary for instruction decoding. The A5/A6 PLA decodes the coprocessor operations. PLA, by the way, stands for Programmable Logic Array. PLAs are arrays of gates that can be customized to particular uses by the layout of the final metal layer in processing. So, as with the ROMs, PLAs can be changed easily without completely changing chip design.

The Instruction Cache has a Tag Cache and a Data Cache. The Tag Cache contains instruction tag information (including the address and a validity

bit). The Data Cache doesn't actually contain data. It does hold the instruction stream.

The Bus Controller manages memory access (including access of the cache).

The Control Unit controls the parts of the micromachine. It interprets nROM information and combines it with secondary decoding of the instruction pipe to finally control the micromachine.

Other parts shown in Fig. 8-10 include the Instruction Pipe, the Clock Generators, the FC Logic, the Size Logic, the Address Buffers, and the Data Buffers.

Packaging. The VLSI 68020 data and address buses aren't multiplexed, and because they are 32-bits wide, the normal DIP package can't hold the chip. Instead, a 114-lead pin-grid array package is used (shown in Fig. 8-11). This is a square package with the chip in the center and with many pins sticking out of the bottom like a bed of nails. The pin grid array package also offers a small size (small footprint compared to a DIP) and high reliability (for better heat dissipation as ceramic packages have). The pins are in a 13 × 13 square with 114 pins total. Not all of the pins are used: the 114-pin package is a standard size.

Speed. The 68020 comes in different versions that run at different speeds. Not all of these will be available immediately. The first samples ran at 12.5 MHz. 1985 will see the emergence of 16.65 MHz samples that have a clock with a 60 nanosecond period and dissipate less than 1.5 watts (which is less than the original 68000 or the 68008 dissipate). A common measure of large computer speed is the MIPS. One MIPS means that a computer can execute one Million Instructions Per Second. That is an averaged figure, and depends on the type of instructions. The 68020 can cruise at 2 to 3 MIPS (say 2.5) for interger processing and can run in short bursts (with the right sort of instructions) at up to 8 MIPS.

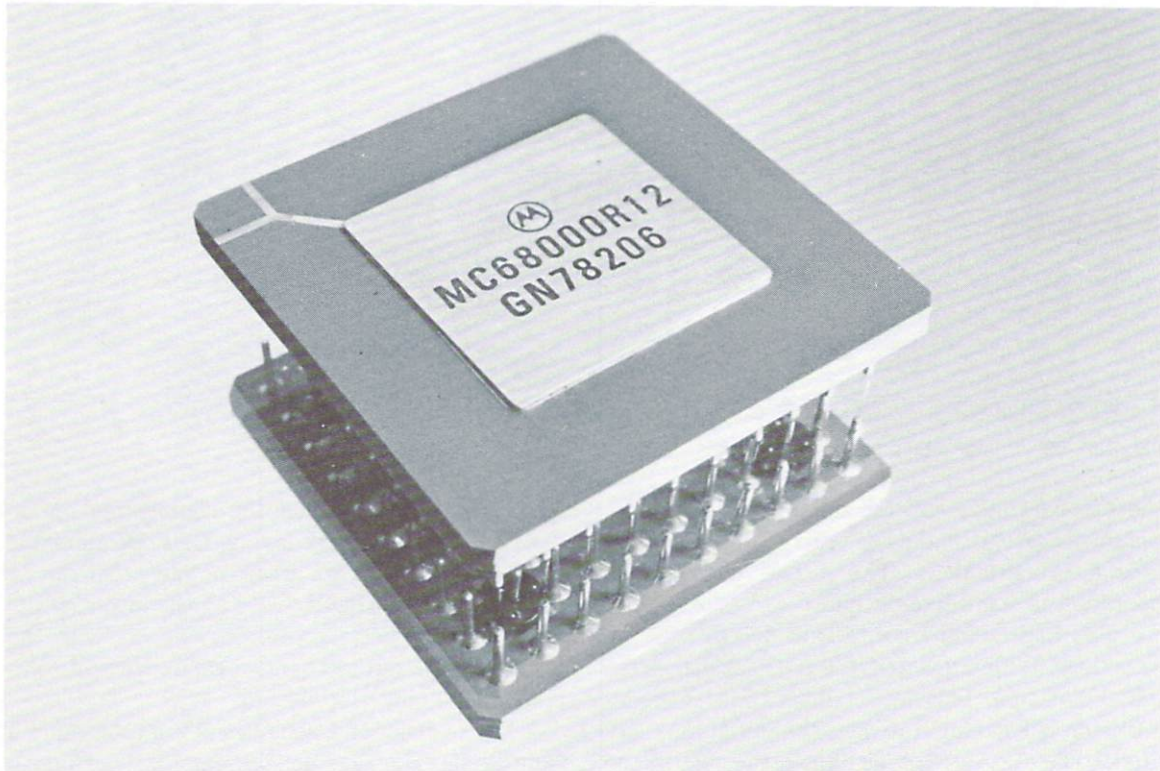


Fig. 8-11. Pin-grid array package (courtesy of Motorola).

Improvements	Factor (68020/68000 8 MHz)
Doubled Clock Frequency	2.0
32-bit Data Bus	1.3
On-board Cache	1.25
New Instructions	1.25

Fig. 8-12. 68020 improved performance factor estimates.

Motorola feels that the 68020 is 2.5 times more powerful than any other chip on the market. (The newest member of the very popular DEC VAX minicomputer line, the 11/785, is rated at 1.35 MIPS.) The 68020 MIPS measurement depends on application. The 8 MHz 68000 runs at 0.5 to 0.75 MIPS. Motorola estimates that the 68020 runs better because of the factors shown in Fig. 8-12. These factors would add up to 5.8 times the performance of the 68000. This means that the 68020 will run at 2 to 3 MIPS with typical instructions, no waits, and no MMU (which would slow it down).

Memory Space. Because it is a completely 32-bit chip, it can address 4 gigabytes of memory directly. There are no instruction timing differences for byte, word, and long-word operations. The 68020 is also built to use Virtual Memory. That is, though it can address 4 gigabytes, it doesn't have to have that much memory directly available as chips. If an area of memory is addressed that isn't on the chips in the system, but is within the 4 gigabytes, the computer system can be set up to grab that part of memory from whatever source it is on and bring it into active chip memory.

Data Bus Adjustment. An unusual feature of the 68020 is its ability to adjust the data bus width to whatever is needed; 8-bit, 16-bit, or 32-bit. That doesn't mean the pins disappear, just that the proper lines are all that is used. At every cycle, the bus can be adjusted by the 68020 itself. Besides easing the programmer's job, that means that 8- and 16-bit peripheral chips will be easy to hook to the 68020. That is important because most peripheral chips probably will be 8- or 16-bit: more bits aren't really useful for most I/O functions.

New Addressing and Instruction Features. The 68020 has some new addressing modes, such as full displacements, true memory indirection, and scaled indexing. It also has new instructions such as bit-field operators, double-ended bounds checking, BCD data compression and expansion, module support, and enhanced system calling functions. Both of these additions help high-level languages work more easily with the 68020. Some instructions that did exist on the previous 68000 family chips, but could not work with long-words, are extended to work with a full 32-bits on the 68020.

Another surprise waiting for 68020 programmers is the existence of 2 Supervisor system stack pointers. These are included to make task switching easier, and to separate task-related exceptions from system-related exceptions. The master stack pointer is active with user tasks so all task-related exceptions are within a user's control block. Other exceptions are handled by the interrupt stack.

Price. As with all microprocessors, the 68020 is expensive: The introductory price for 12.5 MHz samples in late 1984 was $487. But as production experience is gained, and the volume sold climbs, that price will drop. The 6800 started at $450. The 68000 first sold, in 1979, for nearly $450 and now costs around $50 at local electronic hobbyshops. Motorola thinks that by 1989 the 68020 will cost approximately $50.

At that price, and with its speed and memory

addressing ability, the 68020 is bound to show up in engineering workstations, high-performance computers, and communications and control systems. Eventually, there is little doubt it will find its way into the personal computer field as users demand more and more power. Motorola expects to see MC68000 based CAD, CAM, and CAE workstations (individual computers dedicated to Computer Aided Design, Manufacturing, and Engineering), fault-tolerant processors, graphics processing, small to intermediate business computer systems, robotics, and telecommunications switching networks.

Compatibility. Because the 68020 has the same architecture as the other 68000 chips all object code written for the others will run without change on the 68020. Because of its new features, however, some programs written for the 68020 will not run on the previous chips. In fact, because of improvements in clock speed and other new features, that code will probably run faster. The Cache helps speed regular operation and makes multiprocessing easier to accomplish. The new addressing modes help with full flexibility and make high-level languages run better on the chip. The new instructions make complex data manipulations, graphics, robotics, and high-speed controllers work better. Operating systems will be easier to implement because of the program counter, and complete control of the onchip instruction cache. Also, because the user, I/O, and supervisor information are separated.

At the time this book was written (late 1984) there was not yet a second source for the 68020. There will certainly soon be one.

The 68020 has the elements that are found in the earlier 68000 family CPU chips: the 7 address registers, 7 data registers, and 2 stack pointers. It now has an architecture that includes (and all of these are 32-bit) a program counter, a user stack pointer, an interrupt stack pointer, a master stack pointer, a ALU, a cache control register, a cache address register, 7 address registers, 7 data registers, address bus, and data bus.

Cache. The 68020 has a 64-word, on-chip, direct-mapped, instruction cache. The cache—which takes 120 nanoseconds (ns) for access—is faster than external memory that requires a minimum of 180 ns for access (and the minimum can only be reached if very expensive 90 ns RAMs are used for no wait state accessing).

The cache holds recently used instruction sequences. The 68000, executing typical instructions, uses the bus 95% or the time, leaving only 5% for DMA. The 68020 with cache and prefetch, uses the bus as little as 65-70% of the time. The pipelining is done in 3 stages: prefetch, decode (and address calculations), and execute. The 68020 has a pipeline that is 4 active words long where the 68000 has a 3 word long pipeline. Caches and pipelines let the CPU get data and instructions faster: look for a computer science text for more details of these structures.

Peripherals and Coprocessors. This is short and sweet: the present peripheral chips that work with the 68000 will also work with the 68020. In addition, because of its coprocessor interface and huge memory address space, it is easy to interface other processors, chips, or systems to the 68020.

The 68020 coprocessor interface allows the chip to be easily attached to other processors. Because the interface is generalized, the 68020 can work with different coprocessors. (Some other microprocessors are customized to work only with certain coprocessors.) A 68020 system will support up to 8 coprocessors. The way the interface works is that when the 68020 doesn't know how an instruction works, it passes that instruction to coprocessor which decodes it and tells CPU what to do.

The most important coprocessor is the 68881 FPC (Floating Point mathematical processor) which is described in detail later in this chapter. The 68881, 68851 PMMU (Paged Memory Management Unit: also described in this chapter) and the 68020 chip set will rival the performance of any superminicomputer. Motorola claims that it will have the power as a DEC VAX 11/785 minicomputer. (The 688 prefix is Motorola's code for coprocessors.)

68200

This is a 16-bit, single-chip microcomputer from

Mostek. Microcomputers is a term sometimes used for chips that integrate a microprocessor and various peripheral functions onto a single silicon piece. Microcomputers are typically used as controllers for equipment.

In the expanded bus mode, the 68200 directly interfaces to the 68000. It has three timers and a full duplex USART with address wake-up. It has a 128 word RAM, timers and a serial port on the chip.

There are two versions of the 68200. The first has an onchip ROM in a 48-pin plastic DIP. The second is an emulator version in a 84-pin ceramic LCC.

Most instructions operate on both bytes and words. Several 68200 chips can be connected by a single serial channel or a shared parallel bus. In the expandable parallel mode, RAM, ROM, and I/O on the chip are accessed without using the shared bus. It has 4K of onchip ROM and 256 bytes of RAM and addresses a full 64K bytes of memory. Denser memory, faster, and CMOS versions will appear in the future. It has more than 50 instruction types and a number of addressing modes. Most instructions are kept to one word to minimize the use of memory for programs. There are rapid bit-manipulation instructions for both registers and memory.

Instead of the memory organization of the 68000 (which is made for larger systems), the 68200 addresses 64K as 32K × 16-bits. All I/O is memory mapped. The top 1K bytes hold the onchip I/O. There are nine addressing modes including a short-form address that takes only a single word to reach frequently used I/O data. Single-chip microcomputers spend a lot of time getting I/O, so this addressing mode improves performance. Mostek claims the 68200 will perform mathematics faster than the Intel 8096 microcontroller.

The 68200 has an extensive and flexible I/O capability including a serial channel, 2 parallel ports, an interrupt controller, and three 16-bit binary timers for internal timing, pulse-width measurement and generation functions. Every I/O device is programmable. Up to 40 pins are available for I/O.

An interesting new trick is that the serial channel has a wake-up mode. By adding a wake-up bit to each data word, it can transmit and receive wake-up signals. This is an efficient and expedient way to interrupt and process new data, particularly when 68200s are interconnected serially.

The onchip interrupt capability has a reset, a nonmaskable interrupt, and 14 independent vectored interrupts (about twice as many as the competing 8096). The 68200 and the 8096 can be placed in external bus mode for addressing additional memory or for operating standalone. The 68200 can be used as a universal peripheral controller; the 8096 cannot. The onchip bus arbitration logic lets it do DMA transfer to and from system memory.

The 68200 is not 68000 compatible; it is modeled after the 68000. The registers, instructions, and addressing are similar to those of the 68000. The instructions use the same mnemonics to make it easier to use for 68000 programmers.

PERIPHERAL CHIPS

Some of the peripheral chips of the 68000 family were listed in Fig. 8-1. Several of the most important devices are detailed in this section. These chips are designed to do specialized tasks for the 68000 microprocessors and thus ease their processing burden. Some are microprocessors in their own right.

68881 FPC

The Floating-Point Coprocessor (FPC) chip is a special processor that is used for very fast floating-point arithmetic calculations. It is made to support all required and most suggested features of the IEEE proposed floating-point standard. All features are built into the hardware and don't depend on special programs from the outside. Because it can interface directly to the 68020's special bus, it is also known as a *coprocessor.*

Floating Point Numbers. There are two fundamental kinds of arithmetic processing in computers. The first is called integer arithmetic. That is the simpler form and consists of work on numbers that don't have any exponents or fractional parts. For instance, the following numbers are integers:

1
2
64,378
1,024,000,000,000

Integers can be as large as you want, but making them larger requires more and more bits in the number. That makes them awkward for computing where the large number of bits in the numbers makes integer calculations slow. Another disadvantage of integers is that they cannot represent those numbers less than 1 and more than zero: fractions.

Floating-point numbers can represent larger and smaller values within the same number of bits. The *floating-point* description refers to the fact that these numbers don't automatically have the radix point (called the decimal point for those humans amongst us who work with base 10) at the right hand end of the number. Instead, the position of the radix point is determined by the exponent value.

For instance 1×10^4 is a floating point number. This would frequently be shown, in computer books, as 1E4. You could also find the floating point number 1E-4 which stands for 1×10^{-4}, or 0.0001.

(In computers, however, the actual representation of floating point numbers is more complex than this. They are often figured to base 16 exponents and have an automatic value subtracted from that.)

Anyway, lots of computer arithmetic calls out for the size and flexibility of floating point numbers. However, that arithmetic can be slow and cumbersome for the CPU because the mantissa (main value) and its sign, and the exponent and its sign (as well as the various bases involved) have to be remembered and manipulated by a whole list of rules. The MC68881 is faster at performing such arithmetic than a CPU is because it is dedicated to that purpose only.

Architecture. The Motorola 68881 is made in the HCMOS technology (high-performance CMOS) and is specially designed to work with the 68020. That doesn't stop it from working with other CPUs too. It provides a wide range of abilities that match those found in some large computers. It is about as complex as the 68020. Inside, it has a high-speed 65-bit ALU for mantissa arithmetic. There is a barrel shifter that can handle a shift of from 1 to 67 bits in a single machine cycle. This shifter speeds standard arithmetic and is fundamental to the transcendental functions (such as sin, cos, tan).

Because it was designed as an extension of the 68000 family, it keeps many of the same architectural hallmarks. It is, for instance, a register based processor, with 8 80-bit floating-point data registers (shown in Fig. 8-13). These must be so long because they contain a very precise mantissa, an exponent, and the two sign bits. They hold what is called *full extended-precision numbers.*

The 68881 also contains three special 32-bit registers: the control, status, and instruction address registers. The control register contains bits for mode selection and exception enabling. The status register contains a condition code byte (for flags similar to those in the 68020), the FREM and FMOD quotient bits, the exception byte, and the accrued-exception byte. These help control arithmetic and exception handling for the processor. The instruction address register holds the address of the last instruction executed. Because it hangs on to that information, the instruction address register can be useful in tracing the faulty instruction that causes an exception.

The 68881 is internally divided into the Bus Instruction Unit (BIU) and Execution Unit (EU). The EU executes the instructions while the BIU communicates with the CPU. When the 68020 detects a 68881 instruction, it writes the instruction to the memory mapped coprocessor interface command register and reads the coprocessor interface response register. The BIU encodes any addition action the 68020 must do for the 68881.

The 68881 also supports the virtual machine architecture. If the 68020 finds a page fault, and/or a task time out, the main processor can stop the 68881 at any time—even in the middle of execution—and save its internal state. It can also reload the 68881 state.

Data Types. The 68881 can handle four new data types: Single Precision Real (referred to as S), Double Precision Real (D), Extended Precision Real (X), and Packed Real Decimal String (P). The codes

(S, D, X, and P) are used in assembly language programming just as the B, W, and L codes were used before: they are appended to the end of opcodes.

The first three data types use the organization shown in Fig. 8-14. All numbers are converted to full precision, though, when they enter the floating point registers. This means mixed type arithmetic is possible and that there will be no loss of precision (even of integers and BCD strings).

Operation Types. There are five major operation types. Dyadic (2 operands) operations have a source argument that is a 68020 memory location, data register, or floating-point data register. If the source isn't already in extended precision form, it is converted. The destination argument is always one of the floating-point registers. The result, also in extended-precision form, is stored in the destination.

Monadic (1 operand) operations have a single 68020 memory, data register, or floating-point register argument. Again, it is converted to extended precision form and then the result is stored in the destination (a floating point register). Moves and Conversions can move and convert (from one data type form to another) anything in the floating-point registers.

Conditional tests (FBcc, FScc, FDBcc, FTcc, and FTPcc) are identical to the same conditional tests in the 68020 except that the condition code register referred to is the 68881's. Control operations read and write the control status and the instruction address registers and the full 68881 context.

Coprocessor Interface. The special coprocessor interface is built into both the 68020 and the 68881. It is a hardware construction that programmers don't need to worry about. This interface

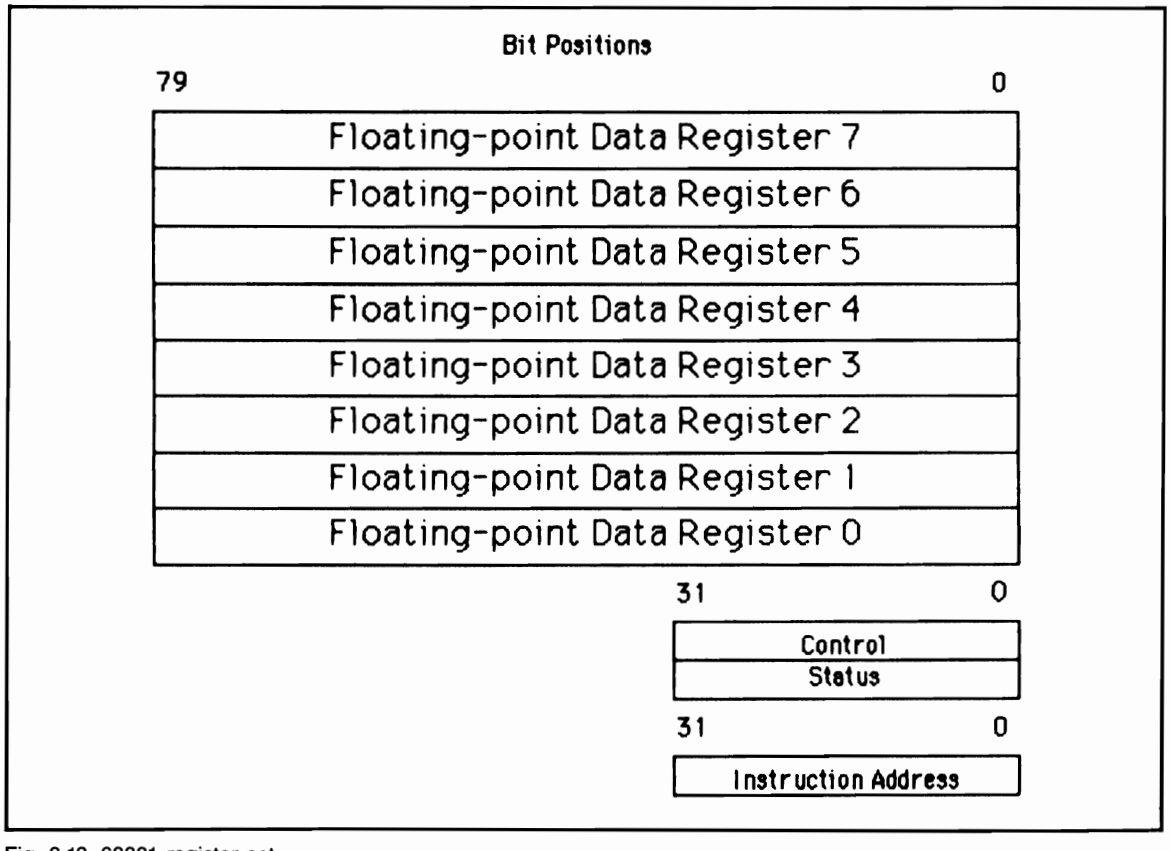


Fig. 8-13. 68881 register set.

A number such as this:

$-5.763 * 10^{8}$

Breaks down this way:

- is the Sign
5.763 is the Mantissa
8 is the Exponent

Which would be arranged this way:

Sign	Exponent	Mantissa

Data Type	Number of Bits Sign	Exponent	Mantissa	Total
Single	1	8	23	32
Double	1	11	52	64
Extended	1	15	64	80

Fig. 8-14. 68881 data types.

allows the two chips to specialize in what they do best. When the 68881 requires certain services that are ably handled by the 68020, it requests and receives those services. The 68881 is a full processor in its own right. Once it gets its instructions it can process without direct help from the 68020. In fact, depending on the instructions, the 68881 can process concurrently with the 68020, overlapping the processing and speeding overall performance. (The great majority of 68881 instructions do overlap.)

Because the interface is simple, though, you can design and use your own coprocessors. Also, multiple coprocessors are allowed. The 68881 can be treated as peripheral in other systems by software reproduction of the handshaking that takes place between the 68020 and the 68881.

IEEE Floating Point Standard. The IEEE standard requires and the 68881 performs the following:

1. Recognition of these data types: Positive True Zero, Negative True Zero, Plus Infinity, Minus Infinity, Denormalized Numbers, Not-a-Numbers (NaN's).

2. Performance of these operations (in full precision): add, subtract, multiply, divide, remainder, compare, square root, integer part.

3. Performance of these rounding modes: to nearest, towards plus infinity, toward minus infinity, towards zero.

4. Performance of these rounding precisions (even though the 68881 makes all calculations to 80 bits of precision, it can emulate narrower precision by appropriate rounding): Round to extended (this is the default), Round to double, Round to single. The traps for exceptions are handled through the 68020, which is signalled and given a vector by the 68881, and handles the exceptions just like any other traps.

But the 68881 goes beyond those requirements. It also has additional instructions and transcendental support.

Additional instructions: Absolute value, negate, scale, exponent, set byte determined by floating-point condition, branch on floating-point condition, move constant to floating-point register, get fraction of floating-point number, get exponent of floating-point number, modulo, test, single precision fast multiply, and single precision fast divide.

Transcendental support. Mathematical functions such as sine, cosine, and logarithms are called transcendental functions. The 68881 includes hardware that will find the value, to double precision of sine, cosine, arctangent, log base 2, log base e, log base 10, 2^x, e^x, 10^x, tangent, hyperbolic arctangent,

hyperbolic sine, hyperbolic cosine, hyperbolic tangent, arccosine, arcsine, log base $e^{(x+1)}$, and simultaneous Sine and Cosine.

68851 PMMU

Like the other 68020 peripheral chips and the 68020 itself, the 68851 is made by the HCMOS (High-performance Complementary Metallic Oxide Semiconductor) technology. The chip first became available in 1985. It is a vital keystone to a system built on the 68020.

The 68851 is a paged memory management unit (PMMU). That is, it helps logically organize the huge memory space that the 68020 can address and translates logical addresses from the 68020 into physical addresses for the RAM and ROM chips. It is specifically intended to help implement a virtual memory scheme. The 68851 can also be used with other CPUs such as the 68010.

Virtual Memory. Virtual memory is called *virtual* because it is a design scheme where the memory chips aren't all actually, physically in the system. In other words, although the 68020 can address 4 gigabytes of memory, most systems won't want to put that many chips into the computer. (If they did it would be very expensive and probably wouldn't fit on a desktop). Instead, a reasonable amount of memory is built into the computer, say 512K bytes or even a 1 Megabyte RAM. Whenever the CPU asks for something that is within that memory space, the request passes through the MMU which refers it directly to the memory chips. If the CPU asked for something within the second megabyte of RAM, however, the MMU would receive the request and realize that the second megabyte worth of memory wasn't within the actual chip space. Instead, the contents of those memory addresses would be on a longer term storage device, such as a disk.

The MMU would execute the proper instructions to move the contents of the second megabyte of memory into the actual chips, swapping it for the first megabyte of memory. Once the new information was within the chips, the MMU would relay the desired address and the memory chips would supply the needed values to the CPU.

In particular, the 68851 is paged memory management unit. It can move single pages into the out of actual physical memory. It doesn't have to move the whole ball of wax. Up to 6 68851s can be used in a system to handle a huge memory space.

Features. Figure 8-15 shows the major structures of the 68851 that are characterized by the following features:

☐ High speed. It translates logical addresses into physical addresses very quickly. If the memory management scheme isn't quick, all the speed of the CPU, such as the 68020, can be wasted waiting for information from memory.

☐ Logical Addresses that consist of a 4-bit function code and a 32-bit address.

☐ A full 32-bit physical address. This means the 68851 won't hold back advanced processors such as the 68020 that can address a full 4 gigabytes of memory.

☐ Eight different page sizes (from 256 bytes to 32K bytes). This means it can swap a variety of sizes of data into and out of physical memory. This allows flexibility in programming. Swapping larger pages takes longer than swapping smaller pages, but may be more efficient in some cases because of the way a particular program accesses memory.

☐ A fully associative 64 entry onchip translation cache. As with the 68020, having an onchip cache speeds up the chip performance.

☐ A translation cache can hold descriptors for multiple processes. The translation cache holds the information the chip needs to decode logical address requests from the CPU into physical address for the memory.

☐ Internal hardware that maintains translation tables and the onboard cache.

☐ A MC68020 instruction set extension and instruction oriented interface using M68000 family coprocessor interface. This simplifies interfacing because all external control chips such as this and the 68881 floating point coprocessor attach to the 68020 in basically the same way.

☐ A linear address space of 4 gigabytes or a hierarchical protection mechanism with eight levels of privilege and protection. Another duty that an MMU can carry out for a microcomputer system is

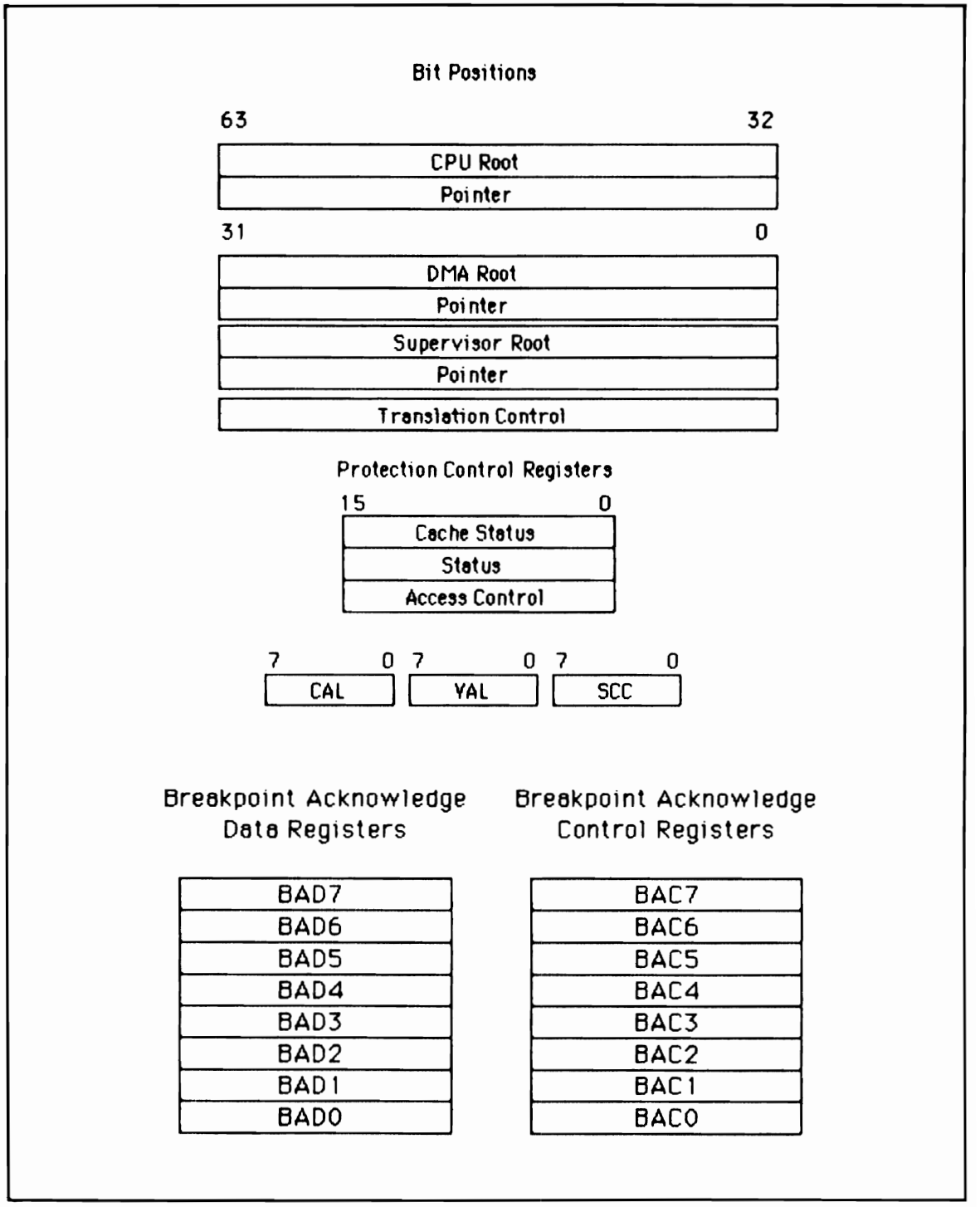


Fig. 8-15. 68851 block diagram.

to know what users (or programs) are authorized to use what parts of memory. This can help protect programs from crashing, simplify multitasking and multiuser systems, and provide data security and privacy.

☐ Support of multiple logical and physical bus masters.

☐ Support of logical and physical data cache.

☐ Support of instruction breakpoints for software debugging and program control.

Purposes. In fact, all of these features come down to three basic purposes of the PMMU:

1. It translates logical into physical addresses.
2. It provides a protection and privilege mechanism for memory.
3. It supports breakpoint operations to make programming easier.

The most important task of these three is the first, and so the chip is optimized to perform at very high speed. It takes the 32-bit logical address (from the CPU) and the 4-bit function code and then begins a translation. It searches for the page descriptor corresponding to the logical-to-physical mapping in the onchip (on the 68851) translation-lookaside module (TLM). This is a very fast 64-entry fully-associative cache memory (just described in the features) that stores recently used page descriptors. By keeping recently used page descriptors in a fast memory, the speed of the overall process of translanting an address is increased. Most programs will use many of translating an address is increased. Most programs will use many of the addresses from the same pages frequently, when in a certain part of the program.

If the descriptor isn't found in the TLM, the bus cycle of the logical bus master is aborted and the 68851 executes enough bus cycles to find the descriptor in the translation table in physical memory. This table is hierarchical and contains the page descriptors that control the logical-to-physical address translations. The 68851 has 64 bit primary root pointer registers that point to the head of the translation tables. Once the proper page descriptor is found, it is loaded into the TLM and the logical bus master retries its bus cycle. This should result in the correct translation.

Protection Mechanism. The 68851 has a hierarchical protection scheme that examines and enforces the access rights of the currently executing process cycle-by-cycle. There are eight levels of privilege and the levels are coded in the upper three bits of the incoming logical address LA (31-29). The 68851 compares those three bits against the value in the current access level register (CAL in Fig. 8-15) and if the priority level of the incoming address is less incoming the current access level, the 68851 will terminate the access as a fault. The 68020 module call and return functions (CALLM/RTM) are supported and this means you can thus change privilege levels during module operation.

Coprocessor Interface. The 68851 uses the 6800 family coprocessor interface. This interface is built into the 68020, 68881, and 68851 chips and allows instructions to be put in a program that are not executed by the main CPU. Each of the coprocessors has a special set of instructions that are customized for its task. When the 68020 runs into one of these instructions in the program and tries to decode it, it will automatically request the special help of the coprocessor. Whatever part of the instruction can be carried out efficiently by the 68020 will be carried out by the 68020.

Programmers do not need to worry about the coprocessors. All they need to know is that they have more instructions to work with when the coprocessors are included in the hardware. The coprocessor interface can be used with these chips, future Motorola chips, and any special processors the user wants to implement.

New Instructions. The 68851 extends the 68000 instruction set. The new instructions let you control the following:

1. Loading and storing of values in the MMU registers.
2. Testing access rights and conditionals based on the result of the tests.
3. MMU control functions.

The new instructions are as follows:

PMOVE. This moves data to or from a 68851 register.

PVALID. This compares access rights requested by logical address and traps if it is less than the current access level.

PTEST. This searches the translation tables to determine the access rights to an effective address. It also sets the 68851 status register according to the results.

PFLUSH. This flushes translation cache entries by any of a number of methods: root pointer; root pointer and effective address; or root pointer, effective address, and function code.

PSAVE. Saves the internal state of the coprocessor interface (for support of 68020 virtual memory).

PRESTORE. Restores the internal state of the coprocessor interface (the inverse of the PSAVE instruction).

PBcc. Branches conditionally on 68851 condition.

PDBcc. Tests 68851 condition, decrements, and then branches.

PScc. Tests the operand according to the 68851 condition.

PTRAPcc. Traps according to the 68851 condition.

68451 MMU

The 68451, like the 68851, is a memory management chip to control the large memory space that the 68000 family can address. This chip does two things: it translates addresses and it provides address protection. The 68451 comes in 4, 6, 8, and 10 megahertz versions.

Each processor in the 68000-based system sends a function code and an address during each bus cycle. The function code tells what address space to use and the address specifies an address within that space. The function codes determine whether User or Supervisor space (and then whether data or program space) is addressed. By separating memory this way, the operating system can be protected from application programs, and individuals memory can be protected from unauthorized access. Special provision has even been made for a separate address space for employing the 68450 DMAC (Direct Memory Access Controller). The protection and control of memory provided by an MMU simplifies the creation of multitasking and multiuser operating systems.

The 68451 has the following features:

1. Provides efficient memory allocation.
2. Separates address spaces of system and other user resources.
3. Provides write protection.
4. Supports paging and segmentation. It can work with 32 segments of variable size with each MMU. Multiple MMUs can therefore expand the system to any number of segments. Intertask communication is simple through the use of shared segments.
5. Is DMA compatible.

68452 BAM

The 68452 is the sort of chip you wouldn't have to worry about in an 8-bit system: the systems just didn't get that complicated. The more complex 16- and 32-bit 68000 systems, however, need control of the bus. For instance, a shared memory area shouldn't be accessed by two different chips at once. A chip such as the 68452 is needed to keep that from happening.

The Bus Arbitration Module (BAM) is an asynchronous controller that allows multiple local buses to be multiplexed onto a common global bus. The local buses to be multiplexed onto a common global bus. The local buses can then share memory and I/O and can also communicate with one another. One 68452 BAM arbitrates for up to eight local buses. Those are assigned a priority, from zero to seven, and the higher priority unit takes precedence over a lower priority unit when both try to access a common site at the same time.

The 68452 words in one of two modes: cycle-by-cycle or block. Cycle-by-cycle arbitrates after every transfer. This could slow down fast devices, so systems with speedy chips (such as DMA and disk controllers) often use block mode. In that mode, a device has the global bus for a number of cycles. Even in block mode, memory access will be slowed because another layer of logic has to be worked through.

68120 IPC

The 68120 and 68121 are Intelligent Peripheral Controllers (IPC). There are slight differences between the two. The 68120 has 2K bytes of ROM on the chip, the 68121 does not. The 68121 has 5 parallel I/O lines, the 68120 has 21 such lines. IPCs are used to harness other peripheral devices, which then don't have to be directly connected to the CPU.

The IPCs have the following features:

1. Bus compatibility with 68000 (asynchronous), 6809 and 6800 chips.
2. 6809 source and object code compatibility.
3. 128 bytes of dual-ported RAM.
4. Multiple operation modes from single chip to expanded.
5. Six shared semaphore registers. (These hold messages from CPU to IPC or IPC to CPU).
6. Parallel I/O lines (21 on the 68120 and 5 on the 68121).
7. A 16-bit three function timer.
8. A serial communications interface.
9. An 8 × 8 multiply instruction.
10. External and internal interrupts.
11. Halt/Bus available capability control.

68440 DDMA

If microprocessors were restricted to moving a single byte or word at a time, the data movement bottleneck would severely hurt system performance. Moving a lot of data in a hurry is a very important computer function. For example, while many people believe the main advantage to having a hard disk drive is that it can store a huge amount of information. That isn't accurate. In fact, the much higher speed of data transfer that hard disks are capable of actually does more to improve system performance.

Moving data is such a simple task that it doesn't make sense to use a complicated and powerful CPU to do it. Therefore, advanced systems take advantage of DMA (Direct Memory Access) where the CPU turns over bus control to another chip. That chip, called the DMAC (DMA Controller) quickly shuffles large sequences of information from input to memory, memory to output, or from one part of memory to another.

Typically, DMA takes place between a disk drive and memory. Instead of having a CPU loop through some MOVE instructions, the CPU signals the DMAC that it wants a certain amount of data moved from a source to a destination. The DMAC then controls the entire movement, and returns control to the CPU when the transfer is complete. This sort of transfer is much faster, and the CPU can even be attending to other business while the DMAC is handling the transfer.

A DMAC is a complicated, specialized processor. The 68000 family has several DMACs. The first is the 68440 DDMA (Dual Direct Memory Access Controller) which is a subset of the 68450 DMAC. The 68440 chip has these features:

1. Bus compatibility with 68000, 68008, 68010.
2. 16 Megabyte addressing range.
3. Byte or word transfers.
4. Two independent channels.
5. Onchip registers for complete program control by system MPU (microprocessing unit).
6. Memory-to-memory, memory-to-peripheral, and peripheral-to-memory transfer capability.
7. Programmable channel prioritization.
8. Vectored Interrupt Capabilities with two vectors per channel.

A transfer operation has three phases: initialization, transfer, and termination. During initialization, the CPU loads the DDMAC registers with control information and address pointers for the device address, memory address, and memory transfer count. Then bus control is given to the DDMAC which provides the addressing and bus controls for the transfer. When the transfer is complete, the termination phase begins. The DDMAC sends status information to the CPU, returns bus control to the CPU, and then idles until it is called again.

68450 DMAC

The other Direct Memory Access Controller

(DMA) chip in the 68000 family is the 68450. This chip has these features:

1. Four independent DMA channels.
2. Memory-to-memory, memory-to-peripheral, and peripheral-to-memory capability.
3. Array-chained and linked-array-chained ability.
4. On-chip registers for complete programmability by the MPU (microprocessing unit).
5. Ability to transfer to 68000 or 6800 peripherals.
6. Programmable channel prioritization.
7. Two vectored interrupts for each channel.
8. Up to 4 Megabytes/second transfer rate.

68230 PI/T

The peripheral chips used most often are those that help handle I/O tasks. Even small systems which don't need the raw horsepower of the DMA, FPC, and MMU chips still have considerable I/O tasks. The 68230 Parallel Interface/Timer uses the following features to handle two common I/O jobs:

1. A variety of port modes: bit I/O, Unidirectional 8-bit and 16-bit, Bidirectional 8-bit and 16-bit.
2. Selectable handshaking.
3. A 24-bit programmable timer.
4. Programmable timer modes.
5. Interrupt Vector generation logic.
6. Separate port and timer interrupt service requests.
7. Onchip registers that are directly addressable from the 68000.
8. Direct DMA compatibility.

Timers are registers that the programmer puts a value into. The value of the timer register will then be regularly decremented. Typically, once that value reaches zero, it returns to the original value and begins counting down again. The zero point can be used to generate periodic interrupts, a single interrupt, or square waves.

The onchip registers of any of these peripheral devices are treated just as memory locations by the 68000. But once you move the proper data to those locations, the peripheral chip can work as a separate processor, controlling memory, buses, or timing within the system.

DATA COMMUNICATIONS CHIPS

With the need to communicate between systems, there are a number of communications and support chips available. This section covers six such chips to support the 68000 microprocessors.

68652 MPCC

The 68652 MultiProtocol Communications Controller (also known as the 2652) formats, transmits, and receives synchronous serial data and uses Bit-Oriented (BOP) or Byte-Control (BCP) protocol. It has a parallel bus which will work with 6800 or 68000 microprocessors.

68653 PGC

The 68653 is a good example of the specialized chips included in the 68000 family. A PGC is a polynominal generator checker and character comparator circuit that is used with a Receiver/Transmitter (R/T, UART, USRT, or USART). What does all of that mean? The 68653 monitors the characters that are transferred between the microprocessor and the R/T chip. It checks for errors or searches for particular characters by performing the block check character (BCC) operation and a parity check on the transferred data.

68661 EPCI

Enhanced Programmable Communications Interface is an enhanced version of the popular Signetics 2651 communications controller chip. It can be hooked to 8-bit or 16-bit microprocessors and will work in polled or interrupt-driven systems. The 68661 (also called the 2661) can be programmed and will handle both synchronous and asynchronous serial protocols at full- or half-duplex mode. The EPCI can simultaneously translate serial data into parallel and parallel into serial.

There are three versions of this chip: A, B, and

C. Each has a different set of baud rates (which can be set internally or externally).

1. Synchronous operation.
2. Asynchronous operation.
3. All operations.

68681 DUART

The Dual Asynchronous Receiver/Transmitter has two UARTs on the chip that are independent and full-duplex. The chip is compatible both with the 68000 family and with many other microprocessors. It can be used in a polled or an interrupt driven system. A UART (pronounced "you-art") is a very common microcomputer system chip because it is the foundation of communication between different computer systems and subsystems.

In a polled system, the microprocessor polls or asks the peripheral devices if they have anything to say. Typically, a timer is set up and each time it counts to zero, the microprocessor asks each of the peripherals in turn if they have new information to report.

An interrupt drive system lets the peripherals tell the microprocessor about new information at any time. When any one of them has something to report, they assert an interrupt to the CPU. If the priority of the interrupt is high enough, the interrupt is acknowledged and the CPU listens to the peripheral. The advantage of this system is that important messages don't have to wait for a polling, they can be received and acted upon right away.

Some of the features of the 68681 are as follows:

1. Quadruple buffered receiver data registers.
2. Programmable data format.
3. Programmable baud rate for each receiver and transmitter.
4. External 1× or 16× clock.
5. Parity, framing, and overrun error detection.
6. False start bit detection.
7. Line break detection and generation.
8. Programmable channel mode.
9. Multi-function 6-bit input port.
10. Multi-function 8-bit output port.
11. Versatile interrupt system.
12. Single interrupt output with eight maskable interrupting conditions.
13. Automatic wake-up mode for multidrop applications.

68562 DUSCC

The 68562 Dual Universal Serial Communications Controller chip puts two independent, multiprotocol, full duplex receiver/transmitter controllers on a single chip. It can handle asynchronous and synchronous communications protocols and will format, synchronize, and validate data. It can work in polled, interrupt drive, or DMA (Direct Memory Access) systems.

Each channel has the following:

1. Receiver.
2. Transmitter.
3. 16-bit multifunction counter/timer.
4. Digital phase-locked loop (DPLL).
5. Parity/CRC generator and checker.

Though both channels share a bit rate generator, they can be programmed for different data formats and operating modes.

68564 SIO

One of the first peripheral chips provided for any microprocessor is a SIO (Serial Input/Output) controller. The 68564 handles this chore for the 68000 family, and is, in fact, two SIOs on a single chip. It can work with asynchronous, byte synchronous (bisync), and synchronous bit-oriented protocols (HDLC and SDLC). It can also handle almost any serial protocol including noncommunications protocols such as floppy disk interfacing. The 68564 has these features:

1. Self test built-in.
2. Directly addressable registers.
3. Two independent full-duplex channels.
4. Quadruple buffered receiver registers and double buffered transmitter registers.
5. Daisy-chain priority interrupt logic.

6. Baud-rate generators.
7. Asynchronous, byte synchronous, and bit snychronous.
8. Address field recognition.
9. CRC generation and checking.

MOSTEK PERIPHERAL CHIPS

As I mentioned once before, second source manufacturers of the 68000 don't only make the CPUs. They also produce and design peripherals. The following pages describe some example peripheral chips from Mostek (which makes the 68000 and 68008 CPUs).

68901 MFP

The Mostek 68901 Multi-Function Peripheral has four 8-bit timers with preprogrammed scalers, an interrupt controller for 16 sources, eight parallel I/O lines and a full duplex USART with programmable DMA signals all in a 48-pin plastic DIP. It is intended for small applications such as instrumentation and personal computers and packs a variety of functions into one box to make system design simple.

Mostek's 68901 MFP combines several important functions on a single chip.

1. Four timers: two multimode timers and two delay timers.
2. An interrupt controller (for 16 sources).
3. Eight parallel I/O lines.
4. A single channel USART.

In many cases, this one chip will handle all of the extra functions a system needs. The 68901 has 24 directly addressable internal registers for both controlling the chip and monitoring its status. These registers are connected to the system bus and can be loaded, checked, and manipulated by the CPU.

68564 SIO

The 68564 Serial I/O Controller has two independent, full duplex serial channels. It can handle asynchronous and a number of synchronous communications protocols and was designed for high-level protocol applications. It has directly addressable registers and can be used in polled, interrupt (vectored and non-vectored) or DMA transfer systems.

68345 FIFO

The 68345 is the highest density FIFO (First-In First-Out) (512×9) memory on the market. It is used in high-speed parallel I/O applications where one data rate needs to be synchronized with another. By using this one chip, more complicated and costly interface circuitry may be eliminated.

68590 LANCE

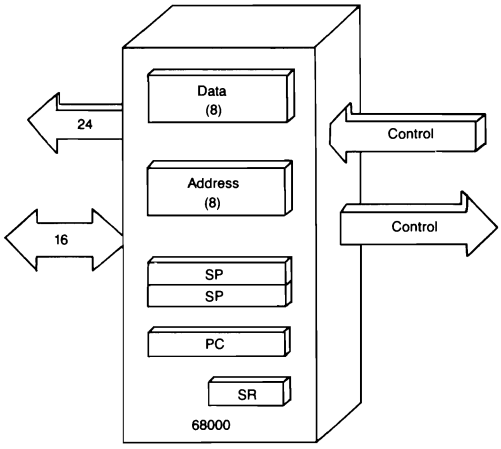
The 68590 Local Area Network Controller for Ethernet combined with an SIA (Serial Interface Adapter) will handle the physical and data link levels of Ethernet. This is a second-sourced chip that interfaces to other 16-bit microprocessors too. Because it has a DMA controller on the chip, it can handle up to 128 messages in a queue without bothering the CPU. It also has a 48 byte buffer.

SUMMARY

There are many, many more 6800 and 68000 family chips all of which are used in 68000 systems. Except for the programmable coprocessors, you (the programmer) don't have to know much about these chips. There are times when you will read or write a particular SIO or timer register, but the documentation of the computer system you are working with should cover that task as a part of the I/O memory map.

But even though you won't be soldering these chips into printed circuit boards, you should be aware of their existence and uses. They are as vital to microelectronic systems as are microprocessors themselves.

9



Assembly Language

THIS CHAPTER WILL DESCRIBE WHAT A COMputer language is and what the different language levels are. Then it will introduce you to assembly language. Don't expect to be able to program in assembly language just from reading this book. You also need the documentation for a particular assembler, some fundamental routines (from magazines or books), and lots of practice. Beyond that, to write efficient and useful programs, you'll need some acquaintance with general programming principles.

Don't be overawed by those requirements. This book is a good place to start. It will allow you to write very simple routines and to understand the assembler documentation and subroutines you will read in the future.

COMPUTER LANGUAGES

Digital computers process 1s and 0s. All of their information is represented by those two symbols, whether that information is a set of population statistics from 18th century Russia, a love letter written today, or a color picture of the rings of Saturn.

People, however, don't speak or think in 1s and 0s. So between the people and the computers there has to be a translation. You might think that a single translation between the human language and the computer language is all that is necessary. It is not that simple. Not only are the people and computer languages widely separated—which suggests the possibility of intermediate languages—but different computers don't even speak the same language.

Machine Language

The first computer programs were written directly in the 1s and 0s that make up *machine language*. The programmers had to be dedicated, highly-trained workers: yet their productivity was severely limited. Programming was slow and error-filled—almost a black art.

Figure 9-1 shows the most common hierarchical breakdown of computer languages. The absolute bottom is represented by binary machine code.

Level	Language Type	Examples	Symbols
Highest Level	Procedural	Pascal, BASIC	Similar to English
	Macro-Assembly		
	Assembly	68000 Assembly	Mnemonics
	Hex Machine		Hexadecimal Numbers
Lowest Level	Machine	68000 Machine	Binary Numbers

Fig. 9-1. Hierarchy of computer languages.

While a small amount of machine level programming is necessary for assembly language I/O operations and hardware debugging, with the analysis and software utility programs available today, no one needs to program entirely in binary machine language. Figure 9-2 shows some of the advantages and disadvantages of programming at this level.

As you can see, the disadvantages of machine-level programming are considerable, including deficiencies in programming speed, accuracy, and por-

Level	Language	Advantages	Disadvantages
High	BASIC	Efficient: Uses little programmer time. Easy-to-read: Uses English expressions. Powerful: Individual instructions translate into many machine instructions. Portable: Can be easily adapted to many different computers.	Memory Inefficient: Uses more memory than low-level language. Lack of control: Difficult to control single bits, especially for timing and I/O. Slow: Cannot be optimized for speed as much as a low-level.
	Macro-assembly	Speed: Combines assembly speed with some of High-level programmer's time efficiency.	Skill Required: Harder to learn and to use than High-level.
	Assembly	Speed: Runs much faster than high-level. Memory efficiency: Uses much less memory than high-level. Control: Allows control of every bit and address.	Skill Required: Harder to learn and to use than High-level. Inefficient: Uses lots of programmer's time. Not Portable: Different language for most microprocessors.
Low	Hex Machine and Binary Machine	Control: Working with every bit inside and outside of the microprocessor. Speed: Can run faster than any other.	Inefficient: Very difficult to work with -- uses maximum time. Not Portable: Different language for every system and every microprocessor.

Fig. 9-2. Advantages and disadvantages of computer language levels.

tability. Those first two disadvantages are opposite sides of the same coin. The machine language programmer must stare at columns and pages of 1s and 0s for hours, days, and weeks. Out of that confusing welter he must recognize instructions, addressing modes, relative branches, and data tables. To do this for a long program is simply not a human activity. Not only will the programmer take a long time to write anything, he will make endless mistakes.

Remember, a single bit misplaced or otherwise in error can completely derail a program. If you are not convinced, try something far simpler than machine-level coding. Write a page full of 8-bit groups of 1s and 0s. Don't try to make them meaningful by looking up op codes, just take the easy route and write random numbers. Now, try to make an exact copy of that page to another page. (And not with a copy machine.) Finally, imagine doing that over and over, often working with a page full of numbers that someone else wrote.

There is a slightly higher level of machine language available. Hexadecimal representation of the bytes in a program will reduce the programming time and programmer's mistakes. This is still machine-level coding, but the distinction between symbols is improved enormously. It didn't take long for hexadecimal-binary translation programs to be written. These programs, sometimes called hex loaders, translate hexadecimal numbers into binary. They allow the programmer to write his lines of instruction in hexadecimal symbols.

Because one hex symbol stands in for four binary symbols, the ocean of 1s and 0s on a page is quickly refined to a river. Still, programmers must make all the translations between numbers and operations in their mind and specify everything. They must work out the mathematics of relative addressing instructions, the destination of each jump or subroutine, and even the exact location that the program will occupy in memory. On computers that don't have an assembler program, the best you can do for direct coding in the microprocessor is to use hexadecimal machine language. (Some personal computers give memory locations and internal data in decimal or octal code instead of hex.)

Portability is a concern when working with machine language. A program is portable if it is written for one computer and yet will run on another computer. This is a different level of relations between systems than the concept of software compatibility described in Chapter 2, but it is related.

Like *compatibility*, portability is rarely 100 percent complete. But writing a program in a portable language means that only slight modifications will be necessary to run it on another computer. High-level languages are very portable. Almost any computer runs some versions of BASIC, for instance. As explained below, if you use the standard rules for BASIC, the program you write can be translated and then run on most any other computer.

Assembly language is far less portable than high-level languages. Because it is basically a faster, clearer way to write machine language programs (and every CPU or microprocessor has a different machine language) assembly and machine language programs cannot easily move from one machine to another.

Assembly Language

The next step after machine language was a natural: a program was written that translates abbreviated names into hexadecimal machine language. That program is called an assembler. Every different microprocessor or CPU has a different assembly language because the assembly language is just an easier way of writing and symbolizing the machine language instructions. Therefore, every different CPU will need a different assembler program.

An assembler is used in the following way:

1. The program is written with the special abbreviations and symbols. This can be done with an editor (the name for a simple word-processing program). As long as the proper abbreviations and punctuation rules are followed, any editor can be used. The finished program is called the source code.
2. The assembler program is run on a computer. This computer can be the computer the final

program is to run on, or it can be any other computer. At this stage, the program isn't going to be used, it is just going to be translated from one bunch of symbols to another. A huge mainframe computer could be, and often is, used to run an assembler that can translate the *source code* into the *object code*. In fact, a large computer will often do the job faster and more efficiently (if you have access to a large computer, that is). The object code is machine language. You are now done with the assembler.

3. A program called a *loader* is used to put the new object code machine language program into the right place in computer memory. If you have more than one piece of object code, and you need to put them together, you can use another program called a linker. There are even linking loaders.

4. You run the program. If there are any problems (and there always are), you return to step 1, use the editor, and debug/rewrite the source code. You will then have to assemble again, load again, and run again to see if you fixed the program.

Any computer equipped with a program called an assembler can be programmed in assembly language. The programmer writes an assembly language program and the assembler translates it into the machine language for the computer. A *disassembler* is a program that accomplishes the reverse. It translates machine language into assembly language. That is useful for modifying or understanding a program; reading assembly is much easier than reading machine code.

Figure 9-2 also lists the advantages and disadvantages of assembly language. The two most important advantages are complete control of the microprocessor (which yields faster programs and efficient use of memory). By working directly with the raw material registers, addresses, and flags, the programmer can observe and manipulate each individual byte or word.

Not all programs have to work quickly, but many do. Even though some of the newer high-level languages create speedy programs, no one will argue with you that an experienced assembly language programmer can write a faster, leaner program than any assembler or compiler can turn out.

Early microcomputers, and all computers, were limited severly by memory considerations. Even today, although bigger memories are much cheaper and more widely available than they used to be, increased program complexity still means that a programmer cannot afford to waste memory. On machines with small memory, the problem becomes not one of cost, but of feasibility.

A program written in assembly language by a competent programmer occupies the least memory. Also, a well-written assembly language program uses the least RAM space during operation.

Symbolic naming is the premier strength of assembly language. Not all assemblers allow symbolic names for everything. But they all allow symbolic instruction codes. These are called *mnemonics*. They stand for the instructions and are sometimes called opcodes. (Sometimes that name is applied to the binary instruction; it depends on who you talk to.) A mnemonic is an easier form to use and remember.

An example of symbolic coding demonstrates another facet of assembly language. An assembly instruction can be translated into more than a single byte of machine code. Symbolic naming almost always extends beyond the actual names of instructions. The next most common use of symbolic names is in addressing. More often than not, this involves jumps or subroutine calls. Instead of calculating the destination address that must be loaded into the program counter and then inserting that number in the program, you can simply assign a name such as DEST and then later either define the name or put a label in the program. This facility also means that the program will be relocatable.

Fundamentally, assembly language programming preserves the best parts of machine language and adds facilities such as symbolic addressing that will ease most program jobs. Because it caters to the human programmer more than machine language and because it is easier to read, use, and debug, it is a higher level of language than machine language.

High-Level Languages

It didn't take long for programmers to chafe at

the restrictions of assembly language. Its abbreviated forms, its strict adherence to the machine language functions, and its elemental operations all kept programmers working long hours. The next step was to write translator programs that could do even more of the work.

High-level languages are the outcome of this drive for productivity. A program called a compiler (or an interpreter) directly translates the high-level *source code* into machine language.

Instead of using the abbreviations of assembly languages, high-level languages frequently allow full words, standard mathematical operations, direct printing of letters, and other features dear to the hearts of programmers. Because the high-level languages offer such flexibility and allow the programmers to use words and numbers that they are all familiar with from other human disciplines, programmers can write far more program in far less time.

However, because the compiler has to do so much work, and has so much to understand, compiling takes more time, and the compiler takes up much more memory than an assembler would. Interpreters are a special sort of compiler. They are explained in more depth in the following paragraphs.

There are many high-level languages. Although a few of the most famous high-level languages like BASIC and Pascal are used for many tasks, many high-level languages are designed for specific purposes. LISP, for instance, is primarily used for Artificial Intelligence work. GPSS is used for simulation. In fact, high-level languages are also called problem-oriented or procedure-oriented languages.

Like assembly language, these languages must be translated into the machine code of binary that a computer can understand. The program that does the translating is called a compiler or an interpreter. A compiler waits for an entire program to be complete before translating it: an interpreter repeatedly compiles the same code. That makes the interpreter slower but more interactive. The following procedure is used for working with a high-level language:

1. Write the source code. This is done using an editor. Most high-level languages have a built-in editor (word-processing program) for writing the symbols.
2. Compile the source code into object code. The object code is the machine language that the computer can understand. An interpreter translates each line as it is entered. Interpreters produce a final object program that takes up more memory space and doesn't run as fast as a compiled code, but the instant translation helps while you are writing and debugging the program.
3. Load the compiled program.
4. Run the compiled program.
5. If there are any problems, return to step 1 to debug/edit the source code. Then you have to recompile, load, and run it again to see if it is fixed. If you use an interpreter, you only have to fix the code and run it. The intervening steps aren't necessary.

BASIC is the most famous and widely used high-level language for microcomputers. Most microcomputers have BASIC compilers and interpreters available. Figure 9-3 shows the brevity of a BASIC program compared to the equivalent program in assembly or machine language. The ability to write easy-to-read, short, and portable programs makes high-level languages ideal for speed of programming. On the other hand, as listed in Fig. 9-2, the disadvantages of high-level languages include a lack of programmer control and of memory efficiency.

The first of these problems is evident in languages such as BASIC where the programmer has no easy way of finding out what is in a particular register. Efficiency is limited because a compiler or interpreter must be very careful to translate the source code into a machine code that will always adhere to the programmer's intentions. That cautious attitude shackles the translation and produces careful code that is rarely as optimized as that produced by human programmer. Optimization is the act of tightening and shortening a program by eliminating unnecessary instructions, changing addressing modes to use shorter and quicker instruc-

BASIC	Assembly	Machine Hexadecimal	Machine Binary
10 LET A = B*4	ANDI.L #0,D0	0280	0000001010000000
		0000	0000000000000000
		0000	0000000000000000
	ANDI.L #0,D1	0281	0000001010000001
		0000	0000000000000000
		0000	0000000000000000
	MOVE.L B,D0	2039	0010000000111001
		bbbb	bbbbbbbbbbbbbbbb
		bbbb	bbbbbbbbbbbbbbbb
	MOVE.L #4,D1	223C	0010001000111100
		0000	0000000000000100
		0000	0000000000000000
	MULS D1,D0	C1C1	1100000111000001
	MOVE.L D0,A	23C0	0010001111000000
		aaaa	aaaaaaaaaaaaaaaa
		aaaa	aaaaaaaaaaaaaaaa
	RTS	4E75	0100111001110101

Fig. 9-3. A program fragment in several computer languages.

tions, and generally editing a program.

Because a single line of BASIC can be translated into a dozen machine language instructions, there is often ample opportunity for optimization. There are always a number of ways to do something in assembly language, and some of them take much longer than others. A programmer can decide to sacrifice memory space or calculation precision to gain some extra speed in a certain predicament. A compiler or interpreter cannot.

In fact, for many real-time applications such as graphics or process control, high-level languages are often just not fast enough.

A Quick Comparison

Figure 9-3 shows a piece of a program written at four different levels: straight binary machine language, hex machine language, assembler, and (high-level) BASIC. Notice the following:

1. It is easier to find a mistake in lines of hex code than in endless binary.
2. It is easier to understand the assembly code than the machine language code.
3. The high-level language program is shorter and clearer than any of the other languages.

LANGUAGE SELECTION

There is no way to pick the one and only best computer language. Selection depends on what you want to accomplish. Although there are languages that are admittedly almost never a popular choice of programmers, you probably haven't heard of them. A language that is unusable dies before reaching any sizable audience. Of the languages you hear about, both low and high level, each has some

advantages and disadvantages.

Perhaps there is some truth to the observation that newer languages are improved over earlier languages and that FORTRAN is passed up by Pascal, but the observation is a lame truth. Of course the faults of previous languages are important to language designers of today, but that doesn't mean you shouldn't learn a language such as FORTRAN. Computer languages are not as complicated as foreign languages. Once you learn a mainstream language such as BASIC, Pascal, FORTRAN, or Forth, other languages will be much easier to learn.

A simple language may be easy to learn, but it won't support complicated data structures that the programmer wants to use later. A complicated language may allow you to use layers of algorithms and file constructs not possible with a simple language, but the complexity of the language means it will take a long time to learn and master; and then the program will be hard to maintain; no one else will know the language.

Don't think of computer languages as you think of foreign languages. Learning another computer language, or several, is not a sign of culture that you can use when traveling or when eating in a fancy restaurant. Computer languages are more like modes of transportation. There is no best mode; the ideal circumstance is to have all modes available to you: feet, bicycle, car, and plane.

In summary, use the level of language that is best suited to your task. Understanding all levels is a good idea, even if you don't have immediate plans to use them. But don't use machine language unless you have to or you will spend far too much time debugging your code of irritant and impossible to find errors. Sometimes, simple computers are not equipped with assemblers, and hand assembly, writing and placing the instructions into the computer byte by byte is necessary. Avoid machine language and stick to assembly language unless you really want to know what is happening on each wire coming out of the chip.

USING A 68000 ASSEMBLER

Assembly language is a symbolic language. To use an assembler you have to learn the symbols it recognizes. Those symbols include opcode mnemonics, directives, and formating symbols, and they differ from assembler to assembler.

Opcode Mnemonics

Each assembler program has its own symbols and practices. In fact, the mnemonics used to represent an instruction doesn't have to match those in this book. Whatever mnemonics you want to use are perfectly OK. You will have a hard time discussing your program with anyone else if you don't use symbols that are at least partially standard, though. And your assembler won't understand you unless you use the mnemonics it expects. Also, if you write in nonstandard mnemonics, you won't be able to transport your program to another system or assembler: you will have lost all portability.

Since Motorola invented the 68000 chip and provided the first documentation, its mnemonics are the most often used. Motorola opcode mnemonics are three, four, or five letters long and are always capitalized. They are generally acronyms for the operations performed by the instruction. Chapter 6 describes all of the instructions the 68000 can perform: they are listed alphabetically by their mnemonics. Figure 9-4 presents a complete list of 68000 mnemonics.

Directives

Directives are assembler instructions. They are not part of the CPU instruction set. These commands are used to specify the address for the beginning of a program, set variable values, reserve memory space for data structures, or define macros. (Macros are compound instruction sequences that are used to save time in programming.) Figure 9-5 lists and defines the directives common to most assemblers.

Syntax

Syntax is the set of rules for correctly putting symbols together in a way an assembler can recognize, work on, and properly translate. Chapter

ABCD	CLR	LSR	ORI to SR	TRAP
ADD	CMP	MOVE	PEA	TRAPV
ADDA	CMPA	MOVE to CCR	RESET	TST
ADDE	CMPI	MOVE to SR	ROL	UNLK
ADDQ	CMPM	MOVE from SR	ROR	
ADDX	DBcc	MOVE USP	ROXL	
AND	DIVS	MOVEA	ROXR	
ANDI	DIVU	MOVEM	RTE	
ANDI to CCR	EOR	MOVEP	RTR	
ANDI to SR	EORI	MOVEQ	RTS	
ASL	EORI to CCR	MULS	SBCD	
ASR	EORI to SR	MULU	Scc	
Bcc	EXG	NBCD	STOP	
BCHG	EXT	NEG	SUB	
BCLR	ILLEGAL	NEGX	SUBA	
BRA	JMP	NOP	SUBI	
BSET	JSR	NOT	SUBQ	
BSR	LEA	OR	SUBX	
BTST	LINK	ORI	SWAP	
CHK	LSL	ORI to CCR	TAS	

Fig. 9-4. 68000 mnemonics.

Directive	Abbreviation	Definition
DATA	DATA	Enters data into fixed program memory.
EQUATE	EQU	Relates symbolic names to addresses or data.
END	END	Marks the end of a program.
ENTRY	XDEF	Shows that name is available for use.
EXTERNAL	XREF	Shows the name is defined somewhere else.
LIST	LIST	Prints the source program.
NAME	NAME	Prints the program name at the top of each page.
ORIGIN	ORG	Specifies memory location where program or data will sit.
PAGE	PAGE	Skips listing to next page.
RESERVE	RESERVE	Allocates memory.

Fig. 9-5. Common assembler directives.

4 covers some of the major points of assembler syntax. The most important point that you should remember is syntax varies from assembler to assembler. While the most basic functions will almost always have the same representations, more advanced functions will not be the same. The source code you write and edit for one assembler may just not run on another assembler. Worse yet, another assembler may make different assumptions about default addresses and values. That could lull you into thinking everything was fine when the second assembler translated your source code into object code when, unfortunately, that object code may not work or may not produce the result you need. Read the documentation for your assembler. Figure 9-6 lists some common syntax rules.

Format

Assemblers organize the assembly language program in divisions called fields. These are not part of the actual code; they are visual structures that the assembler uses to simplify communication with the programmer. When you are programming, you have to enter the information in the proper fields or the assembler will not understand it. When the assembler is printing out a program listing for you to read, it prints in these fields so the program makes sense to you.

There are three main fields: label, instruction, and comment. Two other sections, the line numbers and the addresses, are important but are rarely called fields. Figure 9-7 shows the structure of a simple assembly program. This is the organization of

1. Use symbols to show what number system is used:
   B or % = binary
   Q or @ = octal
   D = decimal
   H or $ = hexadecimal
2. Write the opcode of the instruction first, then write the operands.
3. Separate source and destination operands by a comma.
4. Use an extension letter to show the operand size (no letter indicates a word operation).
   .B = Byte
   .W = Word
   .L = Long-word
5. Use parentheses to show indirection. A pair of parentheses around a register sign shows that the register value is to be used as an indirect address.
6. Use signs to show postincrement and predecrement addressing:
   + after the parentheses to show postincrementing.
   - before the parentheses to show predecrementing.
7. Use a space or colon after a label.
8. Use a space after the opcode.

Fig. 9-6. Common assembler syntax rules.

```
Address  Labels    Opcode    Operands     Comments
                   DATA      EQU          $0010000   SET WHERE TO STORE DATA
                   PROGRAM   EQU          $0012000   SET WHERE TO STORE PROGRAM

                             ORG          DATA
0010000  TEMP      DS.W      1            VALUE TO ROTATE
0010002  COUNT     DS.W      1            NUMBER OF POSITIONS TO ROTATE

                             ORG          PROGRAM
0012000  SHIFTER   MOVE.W    TEMP,D0      GET VALUE TO ROTATE
0012004            MOVE.W    COUNT,D1     GET ROTATION COUNT
0012008            ROR.B     D1,D0        ROTATE
001200A            MOVE.W    D0,VALUE     REPLACE OLD TEMP WITH NEW ROTATED TEMP
001200E            RTS

                   END       SHIFTER
```

Fig. 9-7. Assembly language editing structure.

the source code that the assembler will translate into object code. The line number is almost always the leftmost area of the display. The numbers count up from 1 and are used only for organizing the program on the page. They don't directly affect the code.

The memory locations are frequently the next column of the information. They are given in hexadecimal or decimal values and identify the address in memory where the program lines will be stored. Sometimes the addresses are not specified because the program is not destined for a particular place in memory.

The next area is the label field. This is used to contain symbolic labels or addresses of the various instructions. the assembler can then use the symbolic addresses to specify jump. branch, or return movements. Labels are optional: you can choose when to use them and when to leave this field blank.

The next field is truly the most important. The instruction field breaks down into two portions. The first is the opcode listing. The next portion is the operands listing. Some assemblers call this another field. The operands that relate to the previously listed opcodes are listed here.

Finally, the comment field finishes the display. Comments are optional, and don't change the translation or operation of the program. But you should use them. With the inclusion of explanatory and descriptive comments your program will be easier to write, to read, and to debug. If you have to modify your own program later, you'll be forever thankful that you added comments.

MACRO ASSEMBLERS AND CROSS ASSEMBLERS

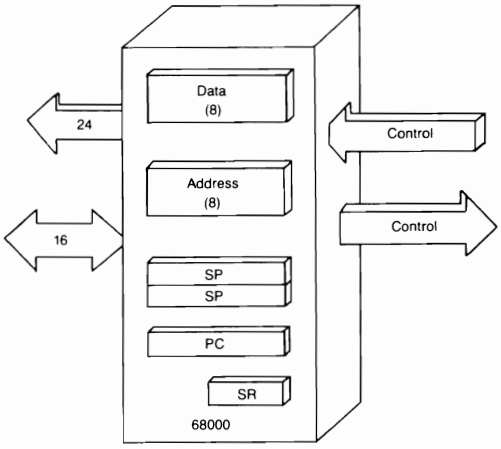
There are several types of assemblers. A powerful assembler known as the *macro assembler* that you may hear of lets you write macros and reuse them. These macros are short pieces of assembly code which are given a name. Once you have included a macro a single time in the source code, all you have to do is give the macro name, and the assembler will make a copy of that routine wherever you put the name. This is not the same thing as subroutines in BASIC because the macro is actually written into the object code in every place it has to be used.

Another important type of assembler is the *cross assembler.* While many programmers work on the

system they are going to run the program on using resident assembler programs, this is not a requirement. In fact, since creating, editing, and assembling source code into object code is abstract (it doesn't require the actual microprocessor), programmers frequently use assemblers that are written on large, powerful minicomputers. That way, the editing and assembling are simplified by the speed and utilities of the minicomputer. Once the assembly is complete, the program will normally be run on the actual microprocessor to test execution speed and to see that the program really works. Minicomputers even have emulation programs that would allow the programmer to run the object code on an imaginary microprocessor that the minicomputer simulates with software.

Programmers that don't have minicomputers can still work with *development systems.* These are complete microcomputer systems that have the same microprocessor as the target computer and yet have more power in the form of more memory, programs, I/O devices, and disk space. Microprocessor manufacturing firms such as Motorola supply development systems for their chips.

10



68000-Based Systems

THE 68000 FAMILY OF MICROPROCESSORS HAS already appeared in many systems and continues to occupy a favored position in the hearts of computer designers. Its powerful 32-bit architecture, its orthogonal, mainframe-like instruction set, and its range of chips (8-bit, 16-bit, and 32-bit) have made it prominent as one of the top two microprocessor families (the other being the 8086 family from Intel).

The 68000 appears in systems ranging from industrial controllers to minicomputers. This chapter attempts to show a few examples of its use in computers. Because the microprocessor world changes so quickly, however, some of these machines may well be defunct by the time you are reading this chapter. Others will have changed designs. Nonetheless, new systems that perform much the same tasks as these will no doubt appear.

Don't forget that the 68000 can also be found in many systems such as robotic controllers and laboratory instruments. However, because fewer readers of this book are likely to be programming such machines, I haven't included examples of these here.

Each example system is discussed and, where possible, illustrated by a photo. Some of the systems are allotted more space than others. The Sinclair QL, is described in detail. It is an example of a 68008 system and will probably be the cheapest complete 68000 system available for some time (several firms offer add-on 68008 boards for the Apple II and IBM PC).

Other systems, such as the Synapse N+1 minicomputer system, are important examples of the high-end power of the 68000, but are not described in too much detail because most people who read this book will rarely encounter, and never program one.

There are far more systems than are even mentioned here. Choosing the Sinclair, Apple, and IBM systems was simple. Deciding which minicomputer and which of the larger microcomputers to discuss was not. The systems shown here were chosen to illustrate the diversity of 68000 uses.

SINCLAIR QL

The first system example is the Sinclair QL. Sinclair chose the 68008 for a CPU because it was the most advanced 8-bit data bus microprocessor on the market and seemed destined to be a future industry standard. It incorporates the power of the 68000 family but can be designed into an inexpensive system. The Sinclair QL could well compete with the Apple Macintosh for the title of best-selling 68000-based system.

The Sinclair QL was announced at the beginning of 1984 with an estimated U.S. price of $500. QL stands for Quantum Leap, which is supposed to be in computing performance. It is aimed at serious home, business, or educational users. Sinclair Research Limited of London is the same company that put out the phenomenally successful rock-bottom priced ZX80 and ZX81 (which became the Timex/Sinclair 1000). These machines, based on the Z80 microprocessor, were introduced as the first computers under $200 at a time when others almost all cost at least five times that much. It is clear that with the QL they are again attempting to make a revolutionary jump instead of an evolutionary step.

The software supplied with it is its own integrated set written by Psion. The programs are called QL Abacus (spreadsheet), Archive (database management), Easel (graphics), and Quill (word-processing).

The QL has high resolution color graphics, 128K RAM memory (expandable externally by the 0.5MB RAM pack shown in Fig. 10-1 to 640K; 32K of this RAM is dedicated to the screen bit map, two

Fig. 10-1. Sinclair QL with 0.5 megabyte add-on RAM (courtesy of Sinclair).

built-in 100K QL Microdrives, and a full-size, professional, QWERTY keyboard.

The QL is 138 × 46 × 472 mm (5 3/8 NCH × 1 3/4″ × 18 3/8″) and weighs 1388 gms (3.055 lbs.). It has rear peripherals ports for full networking, dual joystick, and ROM cartridge expansion.

It has standard RS-232C interfaces (for printers, modems, or other computers), and an RGB monitor and TV port for color or monochrome monitor or TV. The microdrive expansion slot lets you add up to a total of six Microdrives stacked externally for 800K mass storage.

It is built around the 68008, four Sinclair-designed semicustom ICs, and a 32K SuperROM that contains the Sinclair QDOS and Sinclair SuperBASIC (an enhanced version of Spectrum BASIC). ROM is expandable by the ROM cartridge to 64K. QDOS was developed by Sinclair and handles single-user multiple tasking, time-sliced priority job scheduling, display handling for multiple screen windows, and device-independent I/O.

The semicustom ICs are made by several firms. The first is made by both Plessey and Synertek, and controls both display and memory. The second, made by NCR and Synertek, controls the microdrives, LAN, and RS-232C transmission. The third and fourth, made by Ferranti, provide the analog functions required by the Microdrives.

The Microdrives have a capacity of 100K bytes each, 3.5 seconds average access time, and load programs or data into internal RAM at up to 15K bytes/second.

The serial ports are 2 standard RS-232C interfaces that transmit from 75 to 19200 baud or full duplex transmit/receive at seven rates up to 9600 baud. Up to 64 Sinclair QL or ZX Spectrums can be connected to the LAN; data transmission over the net is at 100K baud.

Sinclair Research claims "potential expansion for other peripherals including, say, a memory manager, is almost unlimited due to the QL's advanced Motorola 68008 32-bit processor with its one megabyte linear address capability." The 68008 runs at 7.5 MHz for all principal functions. As described in Chapter 8, the 68008 is the full 32-bit 68000 architecture with a 8-bit external data bus. A second processor, the Intel 8049) controls the keyboard, sound, RS-232C receive, and real-time clock functions.

IBM SYSTEM 9000

This computer was originally designed as a laboratory computer. It is essentially aimed at automating the laboratory as the IBM PC is aimed at automating the office. Because of that emphasis, it has quite a few ports: three RS-232C, bidirectional 8-bit parallel, IEEE-488, three timers, clock, 32 programmable interrupts, and four DMA channels.

It runs on a real-time, multitasking operating system (called CSOS) so that it can collect, store, process, analyze, display, and output data all at the same time.

There are two versions. A lab model and an office model. The 9001 benchtop holds the computer, a display, and a multicolor printer/plotter. The 9002 is smaller and is intended for desktops. XENIX (a version of the UNIX operating system) is available for the 9000, but while still offering multiuser, multitasking capacity, it isn't as good as CSOS for real-time control. To use XENIX, you need a hard-disk, memory management card, 640 K and an 8-inch floppy. The 9000 has a large membrane keypad available with 57 user programmable keys and overlays.

The 9000 measures 6 × 18 × 22″, and weighs 64 lbs. Its CPU is a 68000 running at 8 MHz with four DMA channels. The memory is 128K of ROM with 128K RAM. The memory can be expanded in 256K increments. Three RS-232C, one bidirectional 8-bit parallel, one IEEE-488 port make up the I/O capacity. There are 10 user-definable function keys below the screen as well as an 83-key keyboard with a numeric/cursor keypad. The mass storage includes an optional 640K 5 1/4″ floppy of 985K 8″ floppy. Inside the 9000 are five expansion slots. The 9000 costs approximately $7000.

APPLE LISA AND APPLE MACINTOSH

Apple Computer Corporation's first great success was the Apple II. Kept alive by a number of im-

Fig. 10-2. Apple LISA (courtesy of Apple).

provements (resulting in the IIc model), the Apple II line is built around the 6502 microprocessor. This 8-bit chip was originally chosen because it could do the job and was cheap.

When Apple looked around for a more powerful chip for its more advanced computers, if latched onto the 68000 family. As this book points out, even though the original member of the family (the 68000 chip itself) is a 16-bit microprocessor externally; it is intentionally a 32-bit chip. So Apple was able to leapfrog from 8-bit systems to 16/32-bit systems.

The first 68000-based system from Apple was the LISA shown in Fig. 10-2. The LISA appeared in 1983. Based on a number of ideas like icons, mice, and windows that were originally developed by Xerox, the LISA was unlike any other personal computer. It was designed to be extremely easy to learn to use. It was also designed to sell to office workers who didn't know much about computers.

Unfortunately, it did not sell very well. The reason may have been the high price (approximately $10,000), the slow processing (the 68000 had voluminous software and a screen to handle), or its lack of IBM PC compatibility (the PC is the standard in many companies). Whatever the cause, when January 1984 rolled around and Apple was set to release the Macintosh (described below), it also rolled out three new, more powerful, but cheaper version of the LISA: the Lisa 2, Lisa 2/5, and Lisa 2/10. All include the same icons and windowing software as the original LISA. The 2/5 has a 5 megabyte external hard disk drive and the 2/10 has a 10 megabyte internal hard disk drive.

The Apple Macintosh, shown in Fig. 10-3, was released in January of 1984. Priced at $2500 and

incorporating many of the software ideas of the LISA, the Macintosh depends on a 68000 to handle both screen display and processing chores. The enormous amount of processing necessary to handle overlapping windows and the complex graphics of the Macintosh require a chip with 68000 power. The Macintosh now comes with 512K RAM though the original version sold with only 128K. Newer Macintosh features include built-in hard disks, a second floppy disk drive, and a color display. The black and white screen has an unusually high resolution (for personal computers) and the mouse can be used for many manipulations.

The Macintosh has a standard computer-human interface that is embedded in ROM and used by most programs. This interface includes pull-down menus, windows, icons, and click commands.

Together the LISA and the Macintosh make up Apple's line of 32-bit supermicros. New versions of these machines will undoubtedly appear, including a Macintosh with more memory. In addition, Apple will freely admit that it is looking at the more powerful chips in the 68000 family, including the 68020, for future designs. Because the Macintosh is already designed around the 68000, switching shouldn't be too difficult.

DIMENSION

The Dimension 68000, shown in Fig. 10-4, is an attempt to make a microcomputer that will run software written for any of the popular microcomputers. One of the problems in microcomputing is that programs written for one computer, say the IBM PC, will not run on another, such as

Fig. 10-3. Apple Macintosh (courtesy of Apple).

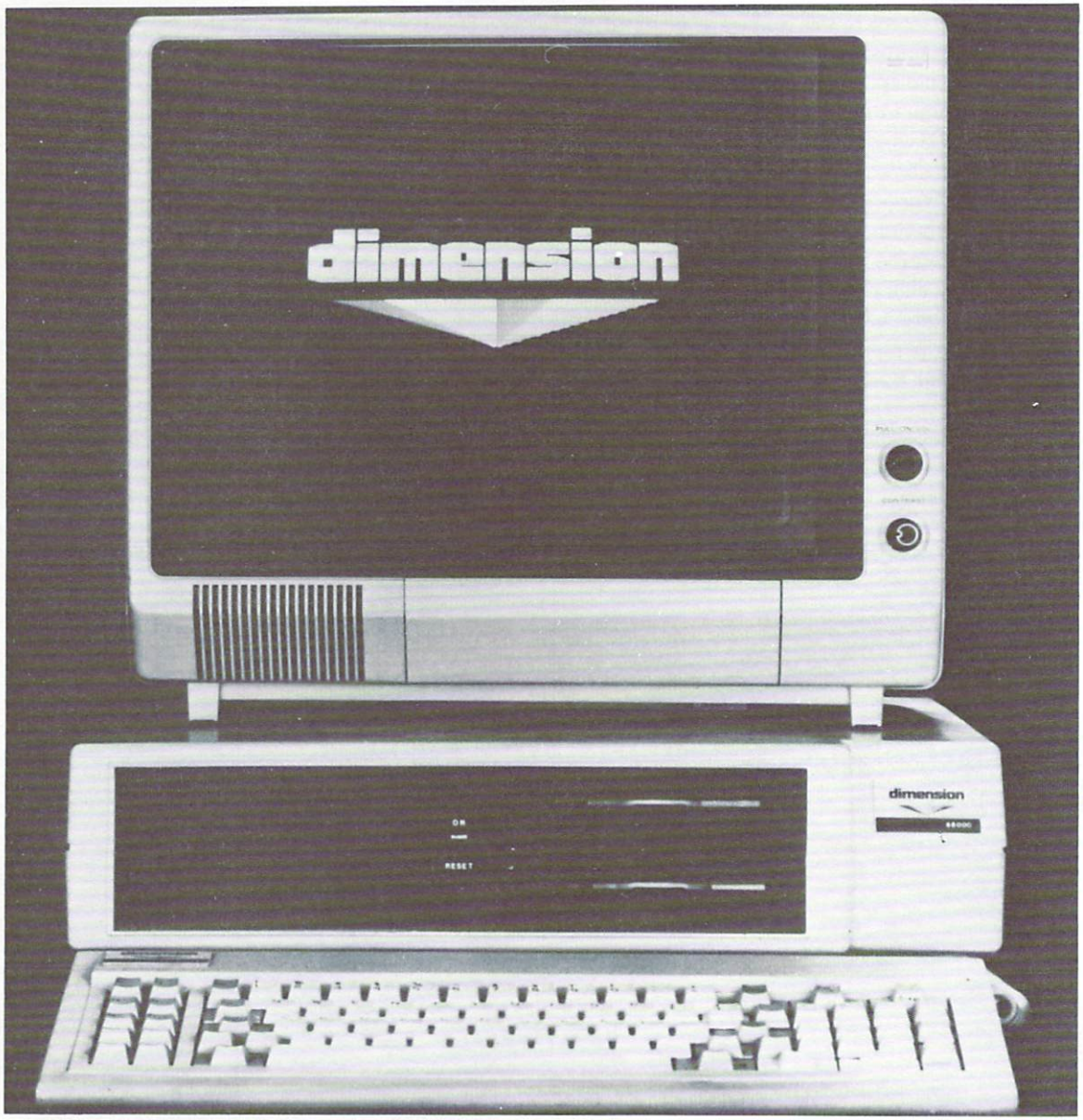


Fig. 10-4. Dimension 68000 (courtesy of Micro Craft).

the Apple II. Dimension advertises that its 68000 computer (built around its namesake 68000 chip) will run software written for Apple II, IBM PC, TRS-80, Osborne, Kaypro, and many other computers.

The 68000 chip in the Dimension is supported by a disk controller, Centronics-style parallel port, a RS-232 serial port, a real-time clock, a 83-key Keyboard with 10 programmable function keys, a ten key numeric pad, a game control port, a CRT display controller for composite color or monochrome, and 6 expansion slots. The amount of memory your Dimension will have, both solid state and disk, depends on what you pay. You can

get 256 K or 512K RAM, and 2 floppy disk drives (either 400KB or 800KB each). You can also get a 20MB or a 50MB hard disk drive with the controller and the cables.

The way the 68000 runs software for the other computer systems is by emulating them with the help of an additional processor card. You have to buy extra processor cards to plug into your Dimension: an 8086 card if you want to run IBM PC software, a 6502 card to run Apple II software, or a Z-80 card to run CP/M-80 software (such as the Kaypro or Osborne runs). Having the other microprocessor is necessary because the programs are specific to a particular chip. The instruction sets and the addressing modes differ. Along with the circuit board (called a card), you get the proper software to allow the new processor, the 68000 processor, and the application program to operate.

The 68000 chip in the Dimension is a Motorola MC68000LB running at 7.19 MHz. The Dimension comes with the CP/M-68K operating system software, UniBASIC, and various utility programs.

CONVERGENT TECHNOLOGIES MINIFRAME AND MEGAFRAME

Convergent Technologies is the sort of computer company that many people never hear of. These firms build computers for other computer companies. For example, Convergent Technologies manufactures the MiniFrame microcomputer and then sells it, in large numbers, to another computer company. That other company then adds some software, terminals, and perhaps a few minor hardware features and resells the systems to the public.

The MiniFrame, shown in Fig. 10-5, is built around a 10 MHz 68010. Its other hardware is designed to allow the 68010 to run with no wait states, or in other words, at full speed. Because of all that power, the MiniFrame can support up to 8 users. The 68010 is a Virtual Memory Microprocessor and the MiniFrame makes heavy use of that capability.

The I/O capacity includes 2 RS-232C ports, a parallel Centronics-compatible printer port, and a RS-422 line. The standard memory of 0.5 MB can be expanded to 2 MB. The system is available with 5.25" hard disk drives which hold 13, 26, or 50 MB. A 5.25" floppy disk drive that will store 640 KB is also built-in.

Convergent Technologies also make the MegaFrame computer. This system, that runs up to 8 MIPS, is comparable to a superminicomputer.

Fig. 10-5. Convergent Technologies Miniframe (courtesy of Convergent Technologies).

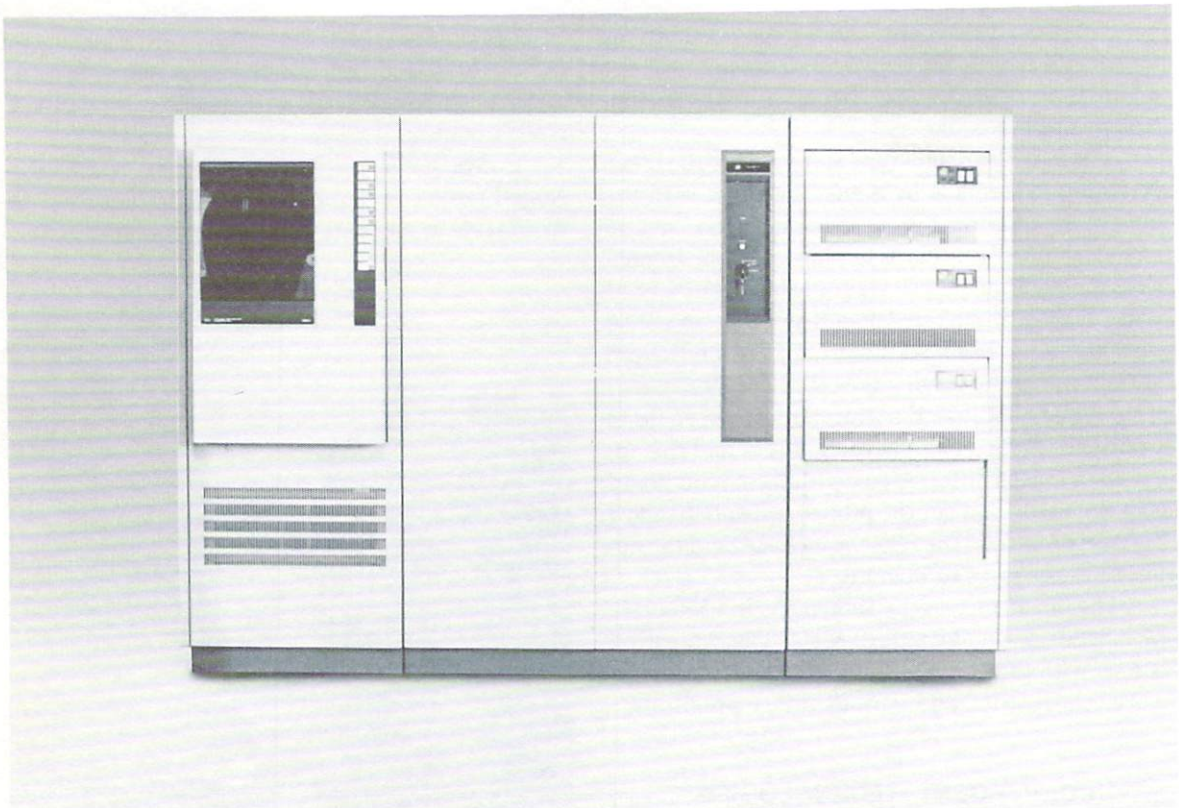


Fig. 10-6. Synapse N+1 (courtesy of Synapse).

It uses up to sixteen 68010s all integrated together and has up to 28 megabytes of RAM. 128 users can work simultaneously on the MegaFrame. Again, the virtual memory capacity of the 68010 is put to good use. Virtual memory is almost mandatory to systems that handle so many users. The boards are designed so that the 68020 can be plugged into them when it becomes available.

SYNAPSE N + 1

The Synapse N + 1 online transaction-processing system (shown in Fig. 10-6) is a very powerful minicomputer built around 68000 family chips. This computer is dedicated to *fault tolerance.* That is a branch of computers that is growing very rapidly. Fault tolerant machines are able to keep working even if some part of the software or hardware breaks down.

There are many schemes to realize fault tolerance. Synapse's scheme involves connecting many 68010 based processors together. Total systems can be built with just a few processors or with many. That way, the system can grow with the users needs. And if one processor stops working, another takes over its load. The software for this must necessarily be more complex and expensive than that for single processor systems, but certain online uses, such as airline reservations and bank transactions, cannot afford to miss any processing time. If their computers are down, they lose business and data. They are willing, therefore, to pay a premium for reliable machines.

Index

Index

OTHER POPULAR TAB BOOKS OF INTEREST

The Computer Era—1985 Calendar Robotics and Artificial Intelligence (No. 8031—$6.95)

Using and Programming the IBM PCjr®, including 77 Ready-to-Run Programs (No. 1830—$11.50 paper; $16.95 hard)

Word Processing with Your ADAM™ (No. 1766—$9.25 paper; $15.95 hard)

The First Book of the IBM PCjr® (No. 1760—$9.95 paper; $14.95 hard)

Going On-Line with Your Micro (No. 1746—$12.50 paper; $17.95 hard)

Mastering Multiplan™ (No. 1743—$11.50 paper; $16.95 hard)

The Master Handbook of High-Level Microcomputer Languages (No. 1733—$15.50 paper; $21.95 hard)

Apple Logo for Kids (No. 1728—$11.50 paper; $16.95 hard)

Fundamentals of TI-99/4A Assembly Language (No. 1722—$11.50 paper; $16.95 hard)

The First Book of ADAM™ the Computer (No. 1720—$9.25 paper; $14.95 hard)

BASIC Basic Programs for the ADAM™ (No. 1716—$8.25 paper; $12.95 hard)

101 Programming Surprises & Tricks for Your Apple II®/II®e Computer (No. 1711—$11.50 paper)

Personal Money Management with Your Micro (No. 1709—$13.50 paper; $18.95 hard)

Computer Programs for the Kitchen (No. 1707—$13.50 paper; $18.95 hard)

Using and Programming the VIC-20®, including Ready-to-Run Programs (No. 1702—$10.25 paper; $15.95 hard)

25 Games for Your TRS-80™ Model 100 (No. 1698—$10.25 paper; $15.95 hard)

Apple® Lisa™: A User-Friendly Handbook (No. 1691—$16.95 paper; $24.95 hard)

TRS-80 Model 100—A User's Guide (No. 1651—$15.50 paper; $21.95 hard)

How To Create Your Own Computer Bulletin Board (No. 1633—$12.95 paper; $19.95 hard)

Using and Programming the Macintosh™, with 32 Ready-to-Run Programs (No. 1840—$12.50 paper; $16.95 hard)

Programming with dBASE II® (No. 1776—$16.50 paper; $26.95 hard)

Making CP/M-80® Work for You (No. 1764—$9.25 paper; $16.95 hard)

Lotus 1-2-3™ Simplified (No. 1748—$10.25 paper; $15.95 hard)

The Last Word on the TI-99/4A (No. 1745—$11.50 paper; $16.95 hard)

101 Programming Surprises & Tricks for Your TRS-80™ Computer (No. 1741—$11.50 paper)

101 Programming Surprises & Tricks for Your ATARI® Computer (No. 1731—$11.50 paper)

How to Document Your Software (No. 1724—$13.50 paper; $19.95 hard)

101 Programming Surprises & Tricks for Your Apple II®/II®e Computer (No. 1721—$11.50 paper)

Scuttle the Computer Pirates: Software Protection Schemes (No. 1718—$15.50 paper; $21.95 hard)

Using & Programming the Commodore 64, including Ready-to-Run Programs (No. 1712—$9.25 paper; $13.95 hard)

Fundamentals of IBM PC® Assembly Language (No. 1710—$15.50 paper; $19.95 hard)

A Kid's First Book to the Timex/Sinclair 2068 (No. 1708—$9.95 paper; $15.95 hard)

Using and Programming the ADAM™, including Ready-to-Run Programs (No. 1706—$7.95 paper; $14.95 hard)

MicroProgrammer's Market 1984 (No. 1700—$13.50 paper; $18.95 hard)

Beginner's Guide to Microprocessors—2nd Edition (No. 1695—$9.95 paper; $14.95 hard)

The Complete Guide to Satellite TV (No. 1685—$11.50 paper; $17.95 hard)

Commodore 64 Graphics and Sound Programming (No. 1640—$15.50 paper; $21.95 hard)

TAB **TAB BOOKS Inc.**

Blue Ridge Summit, Pa. 17214

Send for FREE TAB Catalog describing over 750 current titles in print.

Mastering the 68000™ Microprocessor

by Phillip R. Robinson

- A complete introduction to microprocessors highlighting Motorola's advanced 68000 family of CPUs and support chips.
- Covers the 68000, the 68008, the 68010, and the 68020—the first fully supported full 32-bit chip in practical use—as well as peripherals.
- Provides a complete, detailed breakdown of the 68000 architecture—registers, memory addressing space, flags, exception processing (interrupts), flags, and buses.
- Fully analyzes the 68000 instruction set with definitions, condition code effects, allowable addressing modes, and object codes.
- Introduces assembly language techniques as they apply to the 68000 microprocessors.

If you're looking for expert guidance and to-the-point advice on mastering the use of the 68000 microprocessors, you'll find this an exceptionally complete resource. It's an ideal reference for those with a special interest in the 68000 family of chips for hardware design *and* for assembly language programmers of 68000-based micro and mini computers including: Apple's Macintosh, ATARI's new ST series, the Commodore Amiga, the IBM 9000, the Sinclair QL, new CAD/CAM micros from Apollo and Imagatex, the Synapse® N + 1, the Convergent Technologies series, and other state-of-the-art machines.

Putting his emphasis on the software aspects of the 68000, the author covers the entire family of chips, including the 32-bit 68020 with explanations of currently available packages and applications. In addition he provides detailed illustrations and diagrams and invaluable data tables referencing 68000 chips, computers, and information sources.

Phillip R. Robinson is an engineer and electronics professional whose previous books for TAB include *The Programming Guide to the Z80™ Chip.*

TAB TAB BOOKS Inc.
Blue Ridge Summit, Pa. 17214

FPT > $16.95 ISBN 0-8306-1886-4

PRICES HIGHER IN CANADA 1645-0585