ANALYSIS OF THE M6809 INSTRUCTION SET

BY

JOEL FREDRICK BONEY, B.S.E.

DECEMBER 1981 CS-TR-206

ANALYSIS OF THE M68Ø9 INSTRUCTION SET

BY

JOEL FREDRICK BONEY B.S.E

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

December 1981

PREFACE

In the spring of 1977 several of us at Motorola felt it was time to plan the follow-on part to the successful M68ØØ microprocessor. We were not the first to envision such a part, but we were the first to actually have the time and resources to proceed with the design. The new part was labeled the M68Ø9. Terry Ritter and I were assigned the task of defining the new architecture.

As part of the preliminary design of the M68Ø9 we did an analysis of the then existing programs written for the M68ØØ. Much of the data we gathered from this analysis was very helpful in the design of the M68Ø9.

Several years have now passed since the introduction of the M68Ø9 and I felt it was time to analyze how the M68Ø9 is actually being used. I hope this data will be as useful to those computer architects that follow as the M68ØØ data was to us. It should enable a future designer to make appropriate choices regarding his architecture.

During preparation of this paper I solicited help from many sources. Some people provided their programs for me to analyze, some people helped me write the programs that collect the data, and some people did both. Special thanks to Ed Rupp, Mike Cruess, Dave Trissell, Clay Huntsman, Cedell Alex-

ander, Greg Stevens and Hunter Scales for their assistance. Also, I would like to thank my management at Motorola for providing both time and the computer systems necessary to gather and analyze the data.

Lastly, I wish to thank my wife Becka and daughters Robin and Gena for their patience these last six years.

-- December 2, 1981

TABLE OF CONTENTS

INTRODUCTION 11

1.1 THE M68Ø9 DESIGN PROJECT 13

1.1.1 Goals and Constraints of the M68Ø9 Project 13

1.2 BRIEF OVERVIEW OF THE M68Ø9 ARCHITECTURE 16

1.2.1 Programmer's Model 16

1.2.2 Instruction Classes 18

1.2.3 Addressing Modes 2Ø

1.2.4 Opcode Size 23

1.3 M68ØØ ARCHITECTURE 24

STATIC ANALYSIS OF THE M68Ø9 25

2.1 COLLECTING THE M68Ø9 STATIC DATA 25

2.2 PROGRAM MIX 26

2.3 AVERAGE INSTRUCTION SIZE 29

2.4 MOST FREQUENTLY APPEARING SINGLE OPCODES 3Ø

2.5 MOST FREQUENTLY APPEARING OPCODES BY CLASS 34

2.6 MOST FREQUENTLY APPEARING BY LARGE CLASS 38

2.7 MOST FREQUENTLY APPEARING BY INSTRUCTION GROUP 41

2.8 OTHER INSTRUCTION SET DATA 43

2.8.1 Opcode Page Usage 43

2.8.2 Push / Pull Statistics 44

2.8.3 Direct Page Register Usage 46

2.9 STATIC APPEARANCE OF ADDRESSING MODES 47

2.9.1 Indexed Addressing Static Statistics 49

PAIRS AND TRIPLES 53

3.1 PAIRS 53

3.2 TRIPLES 55

3.3 OBSERVATIONS ON PAIRS AND TRIPLES 57

M6809 VS. M6800 STATIC DATA 58

DYNAMIC ANALYSIS OF THE M6809 61

5.1 COLLECTING THE M6809 DYNAMIC DATA 61

5.2 PROGRAM MIX 64

5.3 CYCLES / INSTRUCTION AND MIPS 66

5.4 MOST FREQUENTLY EXECUTED SINGLE OPCODES 67

5.5 MOST FREQUENTLY EXECUTED OPCODES BY CLASS 68

5.6 MOST FREQUENTLY EXECUTED BY LARGE CLASS 70

5.7 DYNAMIC STATISTICS BY PERCENT OF CYCLES 72

5.8 DYNAMIC EXECUTION OF ADDRESSING MODES 74

5.8.1 Indexed Addressing Dynamic Statistics 75

POSSIBLE PERFORMANCE IMPROVEMENTS TO THE M6809 78

6.1 INSTRUCTION SET IMPROVEMENTS 78

6.2 ADDRESSING MODE IMPROVEMENTS 79

6.3 PUTTING IT ALL TOGETHER 80

COMPARISON OF DYNAMIC VS. STATIC DATA 82

7.1 INSTRUCTION SET COMPARISON 82

7.1.1 Chess 82

7.1.2 Editor 83

7.1.3 Monitor 84

7.1.4 Mopet 85

7.1.5 M6839 86

7.1.6 Short vs. Long Relative 87

7.1.7 Instruction Set Comparison Conclusions 88

7.2 ADDRESSING MODE COMPARISON 90

7.2.1 Indexed Addressing Comparison 91

7.2.2 Addressing Modes Conclusions 92

CONCLUSIONS 94

8.1 STATIC DATA CONCLUSIONS 94

8.2 PAIRS AND TRIPLES CONCLUSIONS 96

8.3 M6809 STATIC VS. M6800 STATIC DATA CONCLUSIONS 97

8.4 DYNAMIC DATA CONCLUSIONS 97

8.5 POSSIBLE PERFORMANCE IMPROVEMENTS CONCLUSIONS 100

8.6 COMPARISON OF DYNAMIC VS. STATIC - CONCLUSIONS 100

8.7 HOW WELL DID I DO? 101

APPENDIX I -- THE M6809 INSTRUCTION SET 105

APPENDIX II -- OPCODE MAP AND INDEXING FORMATS 107

APPENDIX III -- STATIC DATA 108

BIBLIOGRAPHY 118

VITA 12Ø

LIST OF FIGURES

1-1 M6809 Programmer's Model 17

10-1 The M6809 Opcode Map. 107

10-2 Indexed Addressing Mode Formats. 107

LIST OF TABLES

2-1 Average Instruction Size 29

2-2 Top 10 Most Frequently Appearing Opcodes 31

2-3 Second 10 Most Frequently Appearing Opcodes 32

2-4 Top 10 Opcodes By Percent of Bytes 33

2-5 Top 10 Classes Sorted By Count 35

2-6 Union of the Top 10 Classes for Each Program Class ... 36

2-7 Second 10 Most Frequently Appearing Classes 37

2-8 Most Frequently Appearing By Large Class 38

2-9 Comparison of Static Data 40

2-10 Number of Opcodes per Page 44

2-11 Push Pull Statistics 45

2-12 M6809 Static Addressing Mode Usage 48

2-13 Static Indexed Addressing Data 49

3-1 Static Instruction Pairs 53

3-2 Static Instruction Triples 56

4-1 M6800 vs. M6809 Static Instruction Set Data 58

4-2 M6800 vs. M6809 Addressing Modes 59

5-1 Cycles/Instruction and MIPS for the M6809 66

5-2 Top 1Ø Most Frequently Executed Single Opcodes 67
5-3 Union of the Top Ten Classes (Dynamic) 69
5-4 Dynamic Data By Large Classes 71
5-5 Chess by Percent of Instructions and Cycles 73
5-6 Percent of Cycles for Large Class (Chess) 73
5-7 Dynamic Addressing Mode Usage 75
5-8 Dynamic Indexed Addressing Statistics 76
6-1 Possible Performance Improvements 8Ø
7-1 Dynamic and Static Data For: Chess 83
7-2 Dynamic and Static Data For: Ed 84
7-3 Dynamic and Static Data For: Monitor 85
7-4 Dynamic and Static Data For: Mopet 86
7-5 Dynamic and Static Data For: M6839 87
7-6 Short vs Long Branches 87
7-7 Comparison of Addressing Modes 9Ø
7-8 Comparison of Indexed Addressing Modes 91
8-1 Possible Performance Improvements 1ØØ

INTRODUCTION

In 1973 Motorola produced the first M68ØØ microprocessor and its related peripherals. The M68ØØ design was done at the Motorola facility in Phoenix, Arizona. In 1976 Motorola moved its microprocessor operations to Austin, Texas and I transferred into the Microprocessor Design group. In early 1977 I was assigned to the group that was responsible for the architectural design of the follow on processor to the M68ØØ -- the M68Ø9. Since I had several years of software experience at the time, I became most interested in the design of the instruction set.

From very early in the project we felt the need for more data on which to base design decisions, but none existed. Some instruction set analyses had been done on large computers, but no data existed for microprocessors and we weren't sure the data from the big computers was applicable to microprocessors. To help alleviate this problem we undertook a small project to do a static analysis of the existing M68ØØ programs to see how the M68ØØ was being used. Our tools were crude, and it was difficult to find enough code to analyze.We were, however, able to get some rudimentary data. We relied heavily on this data in the design of the M68Ø9.

After nearly two years of intensive design, the M68Ø9 became silicon in January of 1979. By now it has been in the marketplace for over two years, and is one of the most successful microprocessors that Motorola produces. I feel that the static analysis of the M68ØØ code we did in the early design was instrumental in that success. Therefore, the major purpose of this report is to do a similar, more extensive analysis of the M68Ø9 architecture for the use of future architects.

Another purpose of this paper is to determine just how good the assumptions we made about the nature of the programs we expected to be written for the M68Ø9 really was. If the data shows the M68Ø9 to be much as we expected, then it seems reasonable that this type of instruction set and addressing mode data can be used for future designs. Another assumption I would like to test is whether it is possible to use static data, which is relatively easy to obtain, to predict the dynamic nature of a machine. This is exactly what we did in the M68Ø9 design. This paper will do both a static and a dynamic analysis of the M68Ø9 and compare the results.

Lastly, I want to provide enough raw data for the future M68Ø9 architects so that they can formulate their own conclusions. Although I will offer some suggestions about how to improve the performance of the M68Ø9 instruction set, much of the data I have taken will be presented in its raw form. (For the interested reader, I have much more raw data than it is practical to include in this paper and will gladly provide it upon request.)

1.1. THE M6809 DESIGN PROJECT

In order to understand the analysis of the M6809 that will follow it is necessary to have some understanding of the goals and constraints of the M6809 design project. Further, it will be necessary to have some knowledge of the basic architecture and instruction set of the M6809.

1.1.1. Goals and Constraints of the M6809 Project

Every design project in industry begins with some goals that are shared by the designers, the marketers and, hopefully, by the customer. Every project also has some design constraints that must be adhered to in order to design a product that is producible.

A personal goal held by both Terry Ritter and me for the M6809 project was that we wanted to prove that it was possible to produce an inexpensive microprocessor that was also easy to program. We felt too many of the existing microprocessors were needlessly difficult to program. We suspected that the reason was not that it was impossible to make a microcomputer that was easy to program, but, rather, that the architects of the early microprocessors were generally more hardware oriented than software oriented. We both had good software backgrounds and were determined that the M68Ø9 would be easier to program than the other 8-bit microprocessors.

Our experience told us that the consistency (regularity or orthogonality) of the instruction set was one of the features of a computer that made it easy to program. We wanted all the instructions, addressing modes and system resources, such as registers, to be treated in a consistent way. The M68ØØ data analysis showed that instructions in the M68ØØ such as 'add B to A' were rarely used despite the fact that they provided a useful function with better than average performance. The reason was that these instructions were 'unusual' and behaved differently than other instructions. It was our observation that programmers will not use instructions that are hard to use or that require the programmer to remember peculiarities about their execution. Consistency became the battle cry of the M68Ø9 designers!

The second goal of the design team was to support the improvements we saw that were rapidly taking place in the design of microprocessor software. We wanted the architecture to efficiently support modern block structured high level languages. Features such as stack addressing were included for this purpose. We also wanted to better support assembly language with the ability to write recursive and reentrant programs. Foremost, we wanted to support the concept of position independent code so that our customers and Motorola could create libraries of M68Ø9 ROMs that could be placed anywhere in their address space.

Another goal was to greatly improve the performance of the M68Ø9 as compared to the M68ØØ. The M68Ø9 had to be the highest performance 8-bit microprocessor available. Not only did the circuit designers need to make the M68Ø9 a producible 2 Mhz. part, but we also needed to improve the instruction set and addressing modes to increase throughput. Further, we wanted to provide the rudiments of 16-bit support without increasing the cost significantly.

Along with the goals must come some constraints. It was felt that the M68Ø9 must be compatible with the M68ØØ at either the assembler source or machine code level. The machine code level was preferable. However, as it was impossible to get the necessary throughput improvements and remain machine code compatible, we selected assembler source code compatibility.

Since the company was already involved in the technically risky M68ØØØ project, Motorola's management felt that the M68Ø9 should be produced with as little technical risk as possible. Therefore, the M68Ø9 had to be built with the same basic structure and technology as the M68ØØ. It would contain dynamic nodes and be implemented in random logic rather than microcode. It had to be producible in either the HMOS or NMOS production process. Lastly, the most overriding constraint of all, the die size had to be such that the price of the M6809 would lie between the M6800 and the M68000.

The success of the M6809 in the marketplace indicates that these constraints were valid and that most of the goals of the project were met.

For those interested, more details regarding the design stage of the M6809 can be found in the articles that Terry Ritter and I published in "Byte" magazine in January, February and March of 1979. [BON1] [BON2] [BON3]

1.2. BRIEF OVERVIEW OF THE M6809 ARCHITECTURE

In order to understand the data analysis that follows, it will be helpful to have a working knowledge of the M6809 architecture. The following sections will describe the programmer's model, the instruction classes, and the addressing modes of the M6809.

1.2.1. Programmer's Model

The M6809 is a one and a half address, 8-bit Von Neuman architecture microcomputer. Figure 1-1 is the programmer's model of the M6809.

```
+-----------------------+-----------------------+
|    A accumulator      |     B accumulator     |
+-----------------------+-----------------------+

+-----------------------+-----------------------+
|               X index register                |
+-----------------------+-----------------------+

+-----------------------+-----------------------+
|               Y index register                |
+-----------------------+-----------------------+

+-----------------------+-----------------------+
|        U stack pointer/index register         |
+-----------------------+-----------------------+

+-----------------------+-----------------------+
|        S stack pointer/index register         |
+-----------------------+-----------------------+

+-----------------------+-----------------------+
|               program counter                 |
+-----------------------+-----------------------+

                        +-----------------------+
                        |  direct page register |
                        +-----------------------+

                        +-----------------------+
                        |condition code register|
                        +-----------------------+
```

Figure 1-1: M6809 Programmer's Model

The A and B accumulators are general purpose 8-bit accumulators that can be considered as one 16-bit accumulator for 16-bit operations. When used as a 16-bit accumulator it is called the D accumulator.

The X and Y index registers are general purpose index registers used in the various forms of indexed addressing. (Indexed addressing will be explained later.) The U and S registers are also index registers, but they have the additional quality that they can be used as stack pointers. The U register is called the user stack pointer. The S register is the hardware stack pointer and is also used by the hardware to store machine state during subroutine calls and interrupts.

The program counter on the M68Ø9 is 16-bits wide, thus supporting an address space of 65,536 bytes. All addresses on the M68Ø9 are 16-bits wide.

The address field of an instruction with direct addressing on the M68Ø9 is only 8-bits wide. The direct page register is a base register that provides the most significant 8-bits of address for direct addressing. The condition code register contains the results from the last arithmetic or logical operation as well as interrupt masks and other control bits.

1.2.2. Instruction Classes

The 68Ø9 has 7 major classes of instructions:

Arithmetic, Logical, Load and Store
Read / Modify / Write
Conditional Branch
Load Effective Address
Push / Pull
Control Transfer
Miscellaneous

The arithmetic, logical, load and store instructions make up the largest set of instructions. They are one and a half address instructions that get one of their operands from memory and the other from an accumulator and store the result, if any, in the accumulator.

The read/modify/write instructions read a memory location or accumulator, perform some operation on its contents (e.g., clear, shift, increment), and store the result back to the same memory location or accumulator.

The conditional branch instructions are used for conditional program control transfer. The address field of the branch instructions contains an offset to be added to the program counter rather than specifing an absolute address. These instructions are inherently position independent.

The load effective address instructions evaluate the effective address of an indexed addressing mode instruction and return the effective address to an index register. This makes the powerful address calculation hardware already present for indexed addressing available for address manipulation.

The push/pull instructions allow one or several of the registers to be pushed or pulled on the stacks pointed to by the U or S stack pointers. A single push or pull instruction can push or pull from 1 to 8 registers.

The control transfer instructions include the subroutine calls as well as the unconditional jumps and branches. The miscellaneous category includes instructions such as sign extend, no-operation, transfer register to register, etc. Their addressing mode, if any, is inherent.

1.2.3. Addressing Modes

The M68Ø9 supports a wide variety of addressing modes. There are 7 major types with several sub-types:

1. Inherent
2. Accumulator
3. Register
4. Immediate
5. Absolute
    - Extended
    - Direct
6. Relative
    - Long
    - Short
7. Indexed
    - Constant offset
    - Constant offset from the PC
    - Accumulator offset
    - Auto increment / decrement

Inherent addressing includes those instructions which have no addressing options. Accumulator addressing is similar to inherent except that an accumulator is specified (e.g., CLRA, CLRB). Some M68Ø9 instructions specify one or several of the registers as the operands (e.g., TFR D,X - transfer D to X). This is called register addressing. In immediate addressing the source operand is assumed to be in the memory location immediately following the current opcode. The M6809 supports both 8 and 16-bit immediate values.

In absolute addressing all or part of the absolute memory address is included in the instruction. In extended addressing the full 16-bit address is included in the instruction. In direct addressing only the lower 8 bits of the address are included in the instruction. The upper 8 bits of the address are supplied by the direct page register.

Relative addressing is used for branches. In relative addressing the 8 or 16-bit signed offset that follows in the instruction is added to the current program counter if the branch condition is true or if the branch was unconditional. The short form branch has an 8-bit offset and the long form branch has a 16-bit offset.

Many of the new features supported by the M6809 in comparison to the M6800 lie in its greatly expanded indexed addressing modes. In the M6800 the second byte of an indexed instruction contains an 8-bit offset. In the M6809 this _post-byte_ contains additional addressing information.

In the constant offset indexed addressing modes a constant value of length 0, 5, 8 or 16 bits is added to an index register to obtain the effective address that is used to fetch the operand. The original contents of the index register are unchanged by this addressing mode.

Constant offset from the program counter (program counter relative) works in much the same way except the program counter is used as the index register. This addressing mode is used most often by the 'load effective address' instruction to find the starting address of tables in a position independent ROM. The original contents of the program counter are unchanged by this addressing mode.

In accumulator offset mode the effective address is the sum of the signed accumulator and the specified index register. For example, 'LDA D,X' calculates the address of the value to be loaded into the A accumulator by adding the contents of the D accumulator to the contents of X index register. The original contents of the index register and the accumulator are unchanged by this addressing mode.

In auto increment mode the contents of the specified index register are used as the effective address; then it is incremented by 1 or 2 (post-increment). In auto decrement the contents of the index register specified are first decremented by 1 or 2 and then used as the effective address (pre-decrement). In both cases the contents of the index register are permanently changed.

As a further feature, all the indexed addressing modes and the extended addressing mode of the M6809 provide for an additional level of indirection. That is, the original effective address calculated by the addressing mode can be used as the address of another 16-bit value that specifies the final effective address.

1.2.4. Opcode Size

Most opcodes on the M6809 are one byte long plus the number of bytes required for the addressing mode. Some opcodes, however, are two bytes long. It was necessary on the M6809 to go to some two byte opcodes so that we could include the new instructions we felt were important. The two byte opcodes are implemented by defining two separate one byte opcodes to be 'escape' opcodes. Conceptually, we viewed these opcodes as escaping us into a whole new page of 256 possible opcodes. Therefore, we refer to these escape opcodes as the 'page 2' and 'page 3' opcode pre-bytes. We tried to keep the most frequently executed instructions as one byte (page 1) opcodes and let the less frequently used ones become two bytes long (page 2 or 3). The data presented in this report will indicate just how good a job we did of selecting which instructions should be on which page.

For more detailed information about the M6809 see Appendix I for a list of M6809 instructions. Appendix II contains an opcode map of the M6809 and the format of the postbyte for indexed instructions.

1.3. M6800 ARCHITECTURE

Although it is not my intent to describe the M6800 architecture in detail here, it is useful to remember that the M6809 is upward source compatible to the M6800. The major differences are that the M6800 has only the A and B accumulators, the X index register, the S stack pointer, the program counter and a simpler condition code register. There is no D, Y, U or direct page register. The instruction set of the M6800 contains almost no 16-bit operations. Many individual inherent instructions that occur on the M6800 were combined into several more powerful instructions on the M6809. Lastly, the indexed mode of addressing on the M6800 includes only the 8-bit constant offset mode and the X register is the only indexable register.

STATIC ANALYSIS OF THE M68Ø9

There are essentially two types of analyses that can be performed on an instruction set --- static and dynamic. In static analysis either the source or object code of a program or programs is analyzed to determine the frequency of appearance of various instructions, addressing modes, registers, etc. In dynamic analysis data is taken during the actual execution of a program and is used to determine the frequency of execution of an instruction, addressing mode, etc.

Both types of data are useful for specific purposes. The static data can lead to improvements in the architecture that will reduce the size of the average program. The dynamic data can lead to a reduction in the execution time of the average program. They are also related. If a program is smaller, it generally has to fetch fewer bytes of opcode and, hence, runs faster. This relationship will be analyzed further in a later chapter on the dynamic analysis.

2.1. COLLECTING THE M68Ø9 STATIC DATA

This section will briefly describe the mechanical methods used to collect the static data analyzed in this report.

Static analysis can be performed on either object or source code. Object code is easier to obtain and more compact than source code, but it is impossible to determine whether the object code being analyzed is really instructions or is data such as tables. Therefore, I decided to analyze source code. The source data was collected and processed on Motorola's PDP-11/70 under the UNIX operating system. Unfortunately, many of the source programs I wished to analyze were not resident on the UNIX system nor were they all written for the same M6809 assembler. This required uploading many of the programs and preprocessing them with a series of UNIX filters to remove the variation in assemblers.

Once the data was all in the same format and on UNIX, it was processed by a program called STAT9. STAT9 is a modified assembler that takes M6809 source as its input and outputs the object code in ASCII with each instruction's object code on a separate line. It only outputs the object code for instructions and throws away the object code for data and tables.

Next, the various output files of STAT9 were concatenated into files representing different program classes and into a file with all the static data. Another program, RPT, then took this data and generated the data used in the analysis. Appendix III contains the data for the concatenation of all the static data.

2.2. PROGRAM MIX

In order to make the data as useful as possible, I analyzed static code from several different classes of programs. I tried to balance the amount of code in each class so that one class of program would not pervert the data. For example, I could have included over 1ØØ,ØØØ bytes of compiler generated code which would have accounted for two thirds of the total, but marketing information does not indicate that two thirds of the M68Ø9 programs are written in high level language. Therefore, in some classes I deliberately reduced the amount of code in a given class to give what I considered a better mix. The following paragraphs will describe each class and its constituents. I classified the programs into the following classes:

Program Class	Number of Bytes
Compiler generated code	14549
Compiler code	7695
Application code	263Ø5
Monitor code	6293
Numeric code	7135
	61977

Compiler generated code (compiled code) is any code that is generated by a high level language compiler. The two compilers used were the Motorola M68Ø9 Pascal compiler and a version of the Portable 'C' Compiler. The data consists of the compiled code from two programs. The first program was written in Pascal and is a calculator interface to the M6839 floating point ROM. The second program is portion of an operating system written in 'C'.

Compiler code is the actual compiler itself. The compiler used for analysis is an in-house compiler used by Motorola to generate automatic testing sequences for new microprocessors and peripherals. The algorithms for this compiler were written in Pascal, but the code itself was written in a modular assembly language.

Five application programs were analyzed. One is a high speed world-class chess program. It was coded for maximum speed and by nature is recursive. The second program is a screen editor. It was written mainly just to get the job done and was not optimized for either speed or size. The third is a small segment of the diagnostics for a complex serial peripheral part under development at Motorola. The fourth is a graphics package and the fifth is the on-board program that controls a smart terminal.

Three monitors were selected because they represent a class of small programs that are I/O and interrupt intensive. The first monitor is a general purpose, position independent, ROMable monitor called AssistØ9. [MOT2]. The second monitor is a minimal monitor for use in a system where most programs are in high level language. The third monitor is a special purpose monitor used as a tool for executing large diagnostics.

The numeric category only includes one program, but it is fairly large for this type of program. It is the M6839 IEEE compatible floating point package. This package not only includes the basic add, sub, multiply and divide operations, but also includes all the format conversions including BCD to floating point and floating point to BCD. The program is position independent, modular and was written in a structured assembly language.

2.3. AVERAGE INSTRUCTION SIZE

One parameter of interest is the average size of an M68Ø9 instruction. This data can be useful in helping to estimate the memory needed for an application. The size of the average instruction for the various program classes and for all classes combined is given in table 2-1.

Class	Average Size
numeric	2.16 bytes
monitor	2.27 bytes
compiler	2.3Ø bytes
application	2.4Ø bytes
compiled	2.43 bytes
all	2.35 bytes

Table 2-1: Average Instruction Size

These numbers are very close to one another. The larger size of the compiled code is probably due to the compilers' inability to determine when the shorter form of some instructions could be used. Smarter compilers might do better. The small average size of the numeric code is probably due to the fact that it doesn't use extended addressing (3 or 4 bytes long) and uses more indexed instructions.

An average instruction size of 2.Ø would be nearly optimal. The slightly larger size seems to be caused by the large number of the 3 byte 'long branch to subroutine', 16-bit immediate, and 'jump to subroutine extended' instructions. Since it would be impossible to make any of these instructions smaller, the size is probably as close to the minimum as possible. Only a small percentage of the size is due to page 2 and page 3 opcodes and to indexed addressing.

2.4. MOST FREQUENTLY APPEARING SINGLE OPCODES

There are two ways of looking at the static data. One is to order the frequency of appearance based soley on the percent of the total number of opcodes represented by a particular opcode. I call this the percent by count. The other is to order the frequency based on the percent of the bytes actually taken by an instruction. I call this the percent by bytes. I have generally sorted the data by count unless otherwise noted; however, both the 'percent by count' and 'percent by bytes' are given in the data.

Table 2-2 is the data from the 1Ø most frequently appearing single opcodes in the concatenation of all of the static data:

Opcode	Instr.	By Count	Percent	By Bytes	Percent
17	lbsr	23Ø7	8.76	6921	11.17
3Ø	leax	922	3.5Ø	2653	4.28
34	pshs	91Ø	3.46	182Ø	2.94
86	lda imm.	9Ø6	3.44	1812	2.92
2Ø	bra	877	3.33	1754	2.83
8e	ldx imm.	862	3.27	2586	4.17
26	bne	8Ø4	3.Ø5	16Ø8	2.59
27	beq	8ØØ	3.Ø4	16ØØ	2.58
ed	std index	739	2.81	1584	2.56
cc	ldd imm.	722	2.74	2166	3.49
	total		37.4Ø		39.53

Table 2-2: Top 1Ø Most Frequently Appearing Opcodes

For those not familiar with the M6809 instruction set, a brief description is given in Appendix I.

It is interesting to note that the top 10 opcodes represent 37.4% of all instructions. Since there are 266 possible opcodes in the M6809, these 10 opcodes are only 3.76% of all possible opcodes. The top three opcodes are new M6809 instructions that were not available on the M6800. They account for 15.72% of all opcodes. Clearly there was a need for these instructions. Table 2-3 contains the next 10 most frequently appearing single opcodes.

Opcode	Instr.	By Count	Percent	By Bytes	Percent
ec	ldd index	663	2.52	1451	2.34
35	puls	645	2.45	1290	2.08
ae	ldx index	613	2.33	1272	2.05
bd	jsr ext.	605	2.30	1815	2.93
a6	lda index	588	2.23	1261	2.03
c6	ldb imm.	564	2.14	1128	1.82
81	cmpa imm.	517	1.96	1034	1.67
36	pshu	516	1.96	1032	1.67
1f	tfr	486	1.85	972	1.57
39	rts	434	1.65	434	0.70
			-----		-----
total from 2nd 10			21.39		18.86
total from 1st 10			37.40		39.53
			-----		-----
total for 1st 20			58.79		58.39

Table 2-3: Second 10 Most Frequently Appearing Opcodes

The next 10 opcodes account for another 21.39% of all opcodes. The top 20 opcodes account for over 58% of all opcodes. It is interesting to note that the percentage by count and the percentage by bytes is almost identical for he top 2Ø. Since there are no opcodes in the top 2Ø that are from page 2 or page 3, we did a reasonable job of selecting which opcodes should be on page 1. The most frequently executed page 2 opcode was ldy immediate, and it accounted for 1.3% by count and 2.2% by bytes. On page 3 it was cmpu immediate, and it accounted for only .Ø3% by count and .Ø5% by bytes.

By observing the raw data in Appendix III, it can be seen that some opcodes did not appear at all. Only 214 of the possible 266 opcodes were used (8Ø%). Possibly the largest surprise is that cmps (compare stack pointer) was never used.

Table 2-4 contains the top 1Ø most frequently appearing single opcodes sorted by percent of bytes taken.

Opcode	Instr.	By Count	Percent	By Bytes	Percent
17	lbsr	2307	8.76	6921	11.17
30	leax	922	3.50	2653	4.28
8e	ldx imm.	862	3.27	2586	4.17
cc	ldd imm.	722	2.74	2166	3.49
34	pshs	910	3.46	1820	2.94
bd	jsr ext	605	2.30	1815	2.93
86	lda imm.	906	3.44	1812	2.92
20	bra	877	3.33	1754	2.83
26	bne	804	3.05	1608	2.59
27	beq	800	3.04	1600	2.58

Table 2-4: Top 10 Opcodes By Percent of Bytes

Table 2-4 looks much like the ones that were ordered by count, except that the 16-bit immediates have risen higher up the list and jsr extended moved from 14th to 6th place overall.

2.5. MOST FREQUENTLY APPEARING OPCODES BY CLASS

Although it is useful to know which individual opcodes occur most frequently, it is more useful to have the data broken down into slightly larger classes. While there are some cases where an architect may be able to reduce the size or execution speed of some single opcode, it is more likely that he will be able to change a whole class of related instructions. For example, from the previous data we find the single opcode for lbsr is the most frequently appearing, but we also find three other relative branches in the top 20. Therefore, it would be better if all branches could be reduced in size or speed.

The data for all the classes can be found in Appendix III. Table 2-5 contains the top 10 classes sorted by count for the concatenation of all the static data:

class	count	%	bytes	%
ld16	4114	15.62	11291	18.22
ld	2868	10.89	6144	9.91
lbsr	2307	8.76	6921	11.17
lea	1708	6.49	4629	7.47
psh	1426	5.42	2852	4.60
st16	1376	5.23	3155	5.09
st	1219	4.63	2991	4.83
bra	877	3.33	1754	2.83
cmp	860	3.27	1792	2.89
bne	804	3.05	1608	2.59
total		66.69		69.60

Table 2-5: Top 10 Classes Sorted By Count

The classes 'ld16' and 'st16' are the 16-bit loads and stores, and 'ld' and 'st' are the 8-bit loads and stores. The 'lea' class contains the load effective address instructions.

The first six classes account for over 52% of all instructions and the top 10 for 66.69%. Conclusion: The M6809 behaves like most computers in that a very few instruction types account for most of the instructions [STO]. Furthermore, most of the instructions are in the load-store category. (Pushes and pulls are also classified as loads and stores in some literature.)

The new M68Ø9 instruction 'lea' (load effective address) is a cross between a load instruction and instruction that performs arithmetic on an index register. It is probably best classified as an address manipulation instruction.

It might be valid at this point to ask just how valid is this data? Is it consistent across the program classes that go together to make up the concatenated data? (i.e., What is the variation from program type to program type?) Table 2-6 was compiled by taking a union of the top 1Ø instruction classes in each program type. It is interesting to note that the union of the top 1Ø contains only 14 unique classes.

class	all	compiler	numeric	appl.	monitor	compiled
ld16	15.62	9.76	6.13	16.30	7.49	27.04
ld	10.89	9.76	13.15	14.42	9.40	4.20
lbsr	8.76	10.90	8.07	9.07	7.17	7.78
lea	6.49	8.02	8.46	3.10	5.04	11.12
psh	5.42	3.92	1.72	6.53	5.69	5.97
st16	5.23	4.40	4.26	2.60	3.28	12.64
st	4.63	3.02	5.95	5.09	5.19	3.45
bra	3.33	5.60	5.98	2.71	3.67	1.34
cmp	3.27	5.45	3.54	3.77	4.72	.24
bne	3.05	4.40	3.51	3.38	4.03	1.19*
beq	3.04	5.06	1.24	4.21	4.18	1.24*
pul	2.72	3.50	1.42	2.09	5.44	3.05
jsr	2.41	1.35	.06	4.76	2.16	.13
rol	.56	.03	3.41	.16	.22	.14

* used lbne and lbeq since the compilers didn't generate short conditional branches.

Table 2-6: Union of the Top 10 Classes for Each Program Class

Only jsr (jump to subroutine) and rol (rotate left) are suspect data points. Jsr seems to vary from program to program depending on whether the program is position independent or not. Rol is undoubtedly an application dependent data point and probably should be ignored. Otherwise, the data looks consistent.

Table 2-7 contains the next 10 most frequently appearing opcode classes for all the static data.

class	count	%	bytes	%
beq	8ØØ	3.Ø4	16ØØ	2.58
pul	716	2.72	1432	2.31
jsr	635	2.41	19Ø5	3.Ø7
clr	521	1.98	853	1.38
tfr	486	1.85	972	1.57
rts	434	1.65	434	Ø.7Ø
cmp16	422	1.6Ø	14Ø9	2.27
lbra	4Ø3	1.53	12Ø9	1.95
bsr	347	1.32	694	1.12
inc	27Ø	1.Ø3	518	Ø.84

Table 2-7: Second 1Ø Most Frequently Appearing Classes

Note: no arithmetic or logical instructions show up in the top 2Ø except increment, compare, and clear. This certainly casts doubt on the old benchmarking method that considers adds and subtracts as representative instructions.

2.6. MOST FREQUENTLY APPEARING BY LARGE CLASS

We can take an even bigger view of the instruction classes. This data is useful for comparing the usage of the M68Ø9 to other computers. Table 2-8 contains this information for all the static data.

Class	count	%	bytes	%
load	6982	26.52	17435	28.13
call	3289	12.49	9520	15.36
conditional branch	2652	10.07	5992	9.67
store	2595	9.86	6146	9.92
push/pull	2142	8.14	4284	6.91
addr. manipulation	1708	6.49	4629	7.47
compare/test	1482	5.63	3614	5.83
control xfr	1384	5.26	3256	5.25
arithmetic	538	2.04	1315	2.12
logical	526	2.00	1073	1.73
inc/dec	520	1.97	903	1.46
shifts	378	1.44	542	0.87
total		84.44		94.73

Table 2-8: Most Frequently Appearing By Large Class

The 'call' class includes lbsr, bsr and jsr. The address manipulation class includes the lea's. The 'control xfr' class is all the unconditional control transfers: bra, lbra and jmp. Arithmetic includes add, add16, sub, sub16, adc and sbc. Logical includes 'and' 'or' and 'exclusive or'.

If the pushes and pulls and the load effective address are included in the loads and stores, and if arithmetic and logical are combined, then a simpler list looks like:

Class	% by count	% by bytes
load/store	5Ø.83	52.43
call	12.49	15.36
cond. branch	1Ø.Ø7	9.67
control xfr.	5.26	5.25
cmp/tst	5.63	5.83
arith/logical	4.Ø4	3.85
other	11.68	7.61

The large classes above account for 94.73% of all the bytes in an M68Ø9 program, but only use 127 of the 266 possible opcodes (47.74%).

Table 2-9 compares this data with the static data gathered by Leonard Shustek for the IBM37Ø and the PDP-11. [SHU]

Class	M68Ø9	IBM37Ø	PDP-11
load/store	5Ø.83	48.ØØ	32.8Ø
call	12.49	5.5Ø	6.3Ø
cond branch	1Ø.Ø7	15.3Ø	2Ø.1Ø
cntrol xfr	5.26	?	**
cmp/tst	5.63	8.8Ø	6.5Ø
arith/logical	4.Ø4	3.5Ø*	3.ØØ***
other	11.68	18.9Ø	26.3Ø

* subtract only
** control transfer included in conditional branch
*** add only

Table 2-9: Comparison of Static Data

From the above data we can deduce that the M68Ø9 is not much different from other Von Neuman machines. All three machines have a high percentage of loads and stores, subroutine calls, conditional branches, and compares/tests. Further, the amount of arithmetic and logical instructions is low.

2.7. MOST FREQUENTLY APPEARING BY INSTRUCTION GROUP

It might also be useful to see which instructions are most used in each M68Ø9 instruction group. Below are several groups and the top several instruction classes in each group listed with their percent of bytes taken.

Load and Stores:

ld16	18.22
ld	9.91
st16	5.Ø9
st	4.83

Read - Modify - Write:

clr	1.38
inc	.84
dec	.62
tst	.67
rol	.39
ror	.25
asl	.12

Arithmetic:

cmp	2.89
cmp16	2.27
sub16	.66
add16	.63
add	.57

Calls and Control Transfers:

lbsr	11.17
jsr	3.07
bra	2.83
lbra	1.95
bsr	1.12
jmp	.47

Push / Pulls:

psh	4.60
pul	2.31

Conditional Branches:

bne	2.59
beq	2.58
lbne	.81
lbeq	.72
bmi	.41
bcs(blo)	.35
bcc(bhs)	.35

Some conclusions that can be drawn from the above data are:

1. The average program has approximately three times more loads than stores.

2. In read/modify/write the simple functions (clr, inc, dec) occur much more often than do the shifts.

3. Compares are the most frequent arithmetic operation (if you consider them to be arithmetic).

4. There are more 16-bit adds, subtracts, loads, and stores than 8-bit ones.

5. Most programmers prefer to use lbsr and lbra rather than jsr and jmp.

6. There are twice as many pushes as pulls. This is a little puzzling.

7. The second page opcodes 'long branch equal' and 'long branch not equal' have a higher frequency than all the other conditional branches except their short counterparts. In fact when all the long conditional branches are added up, lbne and lbeq account for 68.9% of all long conditional branches.

2.8. OTHER INSTRUCTION SET DATA

During the design of the M6809 we made several decisions based on rather tenuous M6800 data or on no data at all. To verify the correctness of these decisions, I instrumented the static data taking to gather some additional data.

2.8.1. Opcode Page Usage

As mentioned previously, we tried to determined which opcodes would occur most frequently so we could decide which opcodes should have 1 byte opcodes and which opcodes should have two. The 1 byte opcodes are called page 1 opcodes and the two byte opcodes are called page 2 and page 3 opcodes depending on their escape opcode. Below is a table of the number of opcodes that appear in each page and their percentages.

Page	Count	Percent	Bytes	Percent
1	24960	94.80	56863	91.75
2	1354	5.14	5058	8.16
3	16	.06	56	.09

Table 2-10: Number of Opcodes per Page

These results are encouraging. It appears less than 10% of all bytes in an average program are due to page 2 or 3 opcodes. Ten percent seems a fair price to pay for the new instructions on the M6809.

2.8.2. <u>Push</u> / <u>Pull</u> <u>Statistics</u>

The M6800 had four one byte instructions to push and pull the A and B accumulator. The M6801 added two more to push and pull the X index register. When we added the additional registers for the M6809, we knew that we would either have to come up with enough single byte opcodes so that we could have an individual push and pull for each register or we would have to have a two byte instruction. Analysis indicated that there were barely enough unimplemented page 1 opcodes to assign separate push and pull opcodes for each register. We didn't want to use up all of the page 1 opcodes with pushes and pulls if at all possible. Then we realized that, if we had to have a two byte instruction, we could push or pull more than 1 register at a time. So which was better? It depended on whether programmers would take advantage of the multiple register capabilities of the two byte instruction. Table 2-11 contains the usage statistics.

Total number of push/pulls = 2142

Average number of registers push/pulled per instruction: 2.25

Register push/pulled	count	percent
a	1216	25.23
x	1Ø91	22.63
b	844	17.51
y	837	17.37
pc	442	9.17
u	299	6.2Ø
cc	62	1.29
dpr	27	Ø.56
s	2	Ø.Ø4

Table 2-11: Push Pull Statistics

The break even point is 2.Ø registers pushed/pulled per instruction. Even though 2.25 registers per instruction is somewhat disappointing, it is apparent from marketing input that the multi-register push/pull is well received by the user. However, a future M68Ø9 architect might want to consider implementing additional individual one byte pushes and pulls for the A, B, X, and Y registers. This needs some additional study since the data above doesn't tell enough about how the registers are paired. For example, if A and B are pushed simultaneously most of the time, then separate instructions would be of little benefit.

The large number of uses of the PC register is due to the ability to exit a subroutine on the M68Ø9 by pulling the PC along with the other saved registers. (e.g., PULS A,B,X,PC pulls A, B and X from the hardware stack as well as returning from a subroutine.)

2.8.3. Direct Page Register Usage

The decision to include the direct page register on the M68Ø9 was a controversial one. The M68ØØ data clearly indicated that it would be used often, but we also suspected that the M68ØØ data was influenced by the small amount of data memory that was used in the early M68ØØ programs due to the high cost of memory in those days. Would programmers still use direct addressing even if memory costs dropped? Several customers said they would if we provided more 'pages' of memory that could be accessed by direct addressing. Therefore, after much debate we included the direct page register (DPR).

It is a little hard to judge how often the DPR register is used since we didn't provide a separate load DPR instruction. However most (all?) loads of the DPR take place by transferring some register to the DPR (e.g., TFR A,DP). The statistics gathering program looked for and counted these transfers. Further, we can look at the percentage of all instructions that use direct addressing. The two statistics gathered are:

1. The DPR register was only transferred to 9 times.

2. Direct addressing accounted for only 3.64% of all addressing

The percent of direct addressing is much less than on the M68ØØ, and the number of transfers to the DPR is almost zero. Conclusion: the direct page register was a mistake.

2.9. STATIC APPEARANCE OF ADDRESSING MODES

Another major area of interest to an architect is the use of the addressing modes of a computer. They not only indicate areas of the processor that could be optimized, but they also indicate how the machine is being used. Table 2-12 is the addressing mode statistics for the concatenation of all the static data.

Addressing mode	count	percent
indexed	7371	27.99
immediate	5132	19.49
short_relative	3532	13.41
inherent	3466	13.16
long_relative	3054	11.60
extended	1937	7.36
direct	958	3.64
accumulator_b	456	1.73
accumulator_a	424	1.61
indirect	175	0.66

Table 2-12: M6809 Static Addressing Mode Usage

Indexed addressing is by far the most frequently appearing addressing mode because many of the unique features of the M6809 addressing modes are hidden under the umbrella of indexed addressing. For this reason indexed addressing will be discussed in more detail in a later section.

The relative addressing modes account for 25.01% of all addressing modes. This indicates that the programmers are using the relative rather than the absolute control transfers and subroutine calls. In short, programmers are writing a lot of position independent code for the M6809. The relatively small amounts of extended and direct addressing also back up this conclusion.

The accumulator addressing modes supported for the read / modify / write instructions are not used much.

When we designed the M68Ø9, one of the most hotly debated areas of the architecture was whether to include indirect addressing. Some PDP-11 dynamic statistics indicated that indirect addressing was used less than 2.2% of the time. [SHU] [MAR] However, we felt that the stack and stack-frame addressing capabilities of the M68Ø9 would cause programmers to put addresses on the stacks and then to reference their data indirectly through the stacked addresses. The data above, however, does not back this up. The M68Ø9 programmers only used indirect addressing .66% of the time. The inclusion of indirect addressing did not cost a lot of silicon on the M68Ø9, but it did cause some cultural incompatibilities with the M68ØØØ which doesn't support indirect addressing.

2.9.1. Indexed Addressing Static Statistics

Since indexed addressing represents about 72% of all the addressing modes that reference memory (direct, extended, and indexed), we will spend some more time looking at the indexed addressing data. Table 2-13 breaks the indexed addressing down into its subgroups. The basic subgroups are the auto increment/decrement, the no offset, the constant offset, the register offset, and extended indirect.

subgroup	addr mode	number	% of total	% of subgroup
inc/dec	+	286	3.88	39.61
inc/dec	++	12Ø	1.63	16.62
inc/dec	-	43	Ø.58	5.96
inc/dec	--	273	3.7Ø	37.81
no_offset		961	13.Ø4	1ØØ.ØØ
offset	5	394Ø	53.45	73.32
offset	8	631	8.56	11.74
offset	16	572	7.76	1Ø.64
offset	pc8	92	1.25	1.71
offset	pc16	139	1.89	2.59
reg_offset	a	85	1.15	28.62
reg_offset	b	99	1.34	33.33
reg_offset	d	113	1.53	38.Ø5
ext_indirect		17	Ø.23	1ØØ.ØØ

Average additional bytes for indexed = 1.17

Table 2-13: Static Indexed Addressing Data

The constant offset varieties account for 72.91% of the total. If 'no offset' is combined with the constant offset subgroup, we find that 85.95% of the indexed instructions are of a simple type. As is indicated in the table, the program that took the data also calculated the average number of bytes that are added for each indexed addressing mode above the base opcode. The average is 1.17 bytes. Since the minimum possible is 1.Ø bytes, this is a very encouraging statistic. The code size penalty for providing all the new M68Ø9 indexed addressing modes is minimal.