When designing the M6809 we looked at a lot of M6800 data that indicated that constant offsets to index registers were generally small or zero. We took this into account and made both the 5-bit and no offset addressing submodes fit entirely into the index post-byte. This really paid off, and is the main reason the average additional bytes is so low. The indexed submodes that add additional bytes (8 and 16-bit offsets and extended indirect) are low in frequency (less than 20%).

In addition to determining which indexed submodes were used the most, I was interested in which index registers were used most often. Below is the data:

Index register usage:

reg	number	%
u	2411	32.71
s	1801	24.43
x	1762	23.90
y	1149	15.59
pc	231	3.13

This data shows that all the index registers are being used in approximately equal proportions. Even the program counter is used 3.13% of the time. The predominance of U and S register indexing indicates that many M6809 programmers are creating their data areas on the stacks by creating stack

frames and then accessing their data relative to the stack frame pointers. Visual inspection of the source code further confirms this supposition indicating that the programmers are writing position independent and re-entrent programs. The high frequency of subroutine calls also indicates that the programs are modular. It was our goal as architects to encourage this type of programming. It is apparent that, if you give the programmers the tools to write good programs, they will do so.

PAIRS AND TRIPLES

Another area of interest to computer architects when analyzing a computer's instruction set is the frequency of occurrence of instruction combinations. This information can lead the architect into the definition of new instructions on future upward compatible architectures. Toward this end, I analyzed the M6809 source code and looked for instruction pairs and triples.

3.1. PAIRS

Table 3-1 contains the pairs of opcodes that appeared statically more than 70 times in the 26,330 instructions analyzed.

	ct.	pair	ct.	pair	ct.	pair
*	505	ldd std	125	beq lbsr	86	ldx stx
*	302	lbsr lbsr	124	std lbsr	85	ldd cmpd
*	289	lda sta	123	lda ldx	83	ldx tfr
*	279	ldx ldy	121	ldd lbsr	81	addd std
*	254	ldd ldx	118	pshs lbsr	* 80	cmpb bne
	248	ldy pshu	110	ldx leax	80	leax bra
*	211	cmpa bne	105	std ldd	77	lbra lbra
	172	lbsr leas	101	leax stx	75	rol rol
	164	ldx lbsr	98	ldd subd	74	bne lda
*	158	ldb stb	96	pshs ldd	71	lda pshu
*	158	ldx ldd	94	lda ora	71	ldx jsr
	150	tfr puls	93	lda anda	71	leax ldd
	146	cmpa beq	92	ora sta	70	ldb pshs
	145	ldx pshu	90	jsr jsr		
	138	leax pshs	87	lda cmpa		

* referred to in the following discussion

Table 3-1: Static Instruction Pairs

From the raw data in Appendix III we find that there are 1462 ldd instructions. The pair data indicates that 505 of these are followed by std (34.54%). There are also 1797 lda's followed by 289 sta's (16.08%) and 1017 ldb's followed by 158 stb's (14.75%). This data speaks strongly for a move instruction as found on the PDP-11 and the M68000.

The lbsr/lbsr pair indicates modular main program loops that make a series of subroutine calls.

About 2% of the total instructions are compares followed by branch equal or not equal. This is not surprising, but most attempts to combine these instructions in past archi-

tectures have created instructions that are as big and as slow as the two separate instructions.

Probably the most interesting data in the pairs tables is that there are 814 pairs of ld16/ld16. This indicates that a M68000 type of move multiple instruction would be of benefit to the M6809.

3.2. TRIPLES

Table 3-2 contains the triples that occurred (statically) over 30 times in the 26,330 total instructions analyzed.

	ct.	triple		ct.	triple
	245	ldx ldy pshu	*	41	cmpa beq cmpa
*	243	ldd ldx ldy		41	lbsr ldx leax
	121	ldd std lbsr		41	ldd std leax
	116	std lbsr leas		40	ldx lbsr lbsr
	99	beq lbsr lbsr		40	leax ldu stx
	98	lda ldx pshu		40	stb ldx ldd
	85	std ldd std		40	std leax bra
	84	ldd std ldd		40	tfr puls tfr
	78	ldx ldd std	*	39	lda anda sta
	74	ldx tfr puls		39	puls tfr puls
	70	lbra lbra lbra		38	bra bra bra
*	68	lda ora sta		37	bra ldd lbsr
	65	leax pshs lbsr		37	ldu std lbra
*	60	rol rol rol		37	leax bra ldd
	58	pshs tfr leas		36	leax pshs leax
*	53	ldd subd std		35	beq cmpa beq
*	52	ldd addd std		33	leax clrb stb
	50	ldd ldu std	*	33	ror ror ror
	48	ldx leax pshs		33	stx lbsr leas
*	43	lda cmpa bne		31	leax stx leax
	43	ldd lbsr ldx		30	ldb ldx pshu

* referred to in the following discussion

Table 3-2: Static Instruction Triples

Since there are a 243 load triples, this data also backs up the benefit of the move multiple type of instruction.

Although their frequency is not as high as might be expected, this data shows that the 'bit set' and 'bit clear' sequences of load-or-store and load-and-store did occur 107 times. This low frequency indicates that it was probably a good choice not to implement bit manipulation on the M6809. However, this conclusion might not apply to an M6809 housed in a single-chip microcomputer where the I/O tends to be more bit

oriented.

The ldd-subd-std and ldd-addd-std sequences indicate that 16-bit memory to memory adds and subtracts occur frequently.

The rotates are the only shift sequences that appeared often. This shows the possible benefits of multi-bit shifts. However, the nature of the data hides whether the rotates were being made to the same register or memory location or were part of longer multi-precision shifts. A cursory examination of the source code indicates that the rotates were used mostly for multi-precision rotates. If this is true, then multi-bit shifts would not help.

3.3. OBSERVATIONS ON PAIRS AND TRIPLES

Despite the minor suggestions made above, there was not much data that suggested that the M68Ø9 lacks any simple new instructions. No pair or triple accounted for more than 5% of the total. This is not to say that the M68Ø9 instruction set cannot be improved; only that the improvements will not be made by fusing several existing instructions. New instructions would probably be complex instructions like procedure entry, procedure exit or divide.

M6809 VS. M6800 STATIC DATA

Since many of the decisions we made in the design of the M6809 were based on the static data we took for the M6800, it is interesting to compare the two computers.[BON4] Table 4-1 contains a comparison of the static instruction set data for the two computers based on large classes.

Class	M6809%	M6800%
load	26.52	23.4
subroutine calls	12.49	10.2
conditional branches	10.07	11.0
store	9.86	15.3
push/pull	8.14	2.1
address manipulation	6.49	---
compare and test	5.63	5.3
control transfer	5.26	6.5

Table 4-1: M6800 vs. M6809 Static Instruction Set Data

The basic nature of the instruction set usage hasn't changed all that much from generation to generation. The new M6809 push/pull and address manipulation instructions (lea's) have changed the statistics some and demonstrate that the M6809 is being used in a more modular fashion than the M6800.

The comparison of the addressing modes, on the other hand, show a marked difference in the usage. Table 4-2 shows the comparison.

Addressing mode	M6809%	M6800%
indexed	27.99	8.96
immediate	19.49	14.82
short relative	13.41	15.75
inherent	13.16	11.71
long relative	11.60	0
extended	7.36	23.44
direct	3.64	18.33
accumulator a/b	3.34	6.98
indirect	0.66	0

Table 4-2: M6800 vs. M6809 Addressing Modes

From the above data we could conclude that the M6800 is a computer where most of the code was written in absolute mode (direct and extended = 41.77%) with only a small number of indexed instructions. The M6809, on the other hand, is a computer where most of the instructions are indexed, relative or immediate. This implies that most M6809 code is position independent and most M6800 code was not. Further, we know from the indexed addressing data discussed in a previous chapter, that most of the indexed addressing is of the zero or 5-bit offset variety. By observing the actual source code, the reason is apparent. Most M6809 programmers do write position independent code, and they prefer to carve data areas off of the hardware stack for their local variables. In addition, since the number of local variables is relatively small, the offsets from the hardware stack pointer or a stack mark pointer are generally small. The M6800 data shows no tendency

towards this type of programming due to the lack of the necessary addressing modes and indexable registers.

DYNAMIC ANALYSIS OF THE M68Ø9

Although the static data used in the previous chapters is useful in predicting the size of a program, data taken while a program is actually executing is more useful in determining the throughput of a computer. Also, the dynamic data can best be used to improve the performance of a follow-on or new architecture. Unfortunately, reliable dynamic data is much harder to obtain than static data; hence, not much dynamic data exists for microprocessors. [SHU]

This chapter will analyze the dynamic data I took for the M68Ø9. It begins with a discussion of the data collection methods used and the program mix. It then contains the analysis data for the instruction set and the addressing modes.

5.1. COLLECTING THE M68Ø9 DYNAMIC DATA

There are two basic ways of collecting dynamic data. One is to build special high speed hardware to monitor the instruction execution of a computer. This method has the advantage of being real-time, but has the disadvantage of being very expensive. In order to reduce hardware costs, it is usually necessary for the hardware to only take snapshots of a fixed number of cycles. It is hoped that these snapshots will faithfully represent the execution characteristics of the

whole program.

The second method is to run programs using a simulator. The simulator can be instrumented to collect the data cycle by cycle. The problem with a simulator is that is slows down the program's execution so much that the statistics may become warped. This is especially true for real-time programs. In fact, many real-time programs won't run on a simulator at all. Further, the simulator may have problems simulating the I/O and interrupt portions of the programs.

Back in 1978, we built a hardware/software simulator for the M6809. It runs on an M6800 EXORciser and faithfully emulates the M6809. It is smarter than most simulators because it includes some hardware that lets the simulated program do most I/O operations and interrupts. For this report, I modified this simulator to count the number of times each opcode and addressing mode was executed. It also counts the number of times each indexed addressing postbyte occurs. Lastly, it counts total number of cycles.

Although this simulator is a good tool, it created some problems. The address space of the M6800 EXORciser is 64K bytes, but the EXBUG firmware and the simulator hardware and firmware took up over 24K bytes (1K = 1024 bytes). If the program being emulated also required the MDOS disk operating system, then only 16-20K of space was left for the program.

This made it impossible to take dynamic data from some large programs. Further, even though the simulator can do some I/O and interrupts, it can't do them in real-time, making it impossible to emulate some real-time programs (e.g., the OS9 operating system).

Lastly, and most disappointing, the address space limitation made it impossible to simultaneously take dynamic data on the instruction set and indexed addressing both by percent of count and by percent of time. That is, the data gathered indicates exactly how many times a particular instruction was dynamically executed, but it cannot determine what percent of the total execution time was taken by the instruction. Sometimes it is possible to calculate this percentage after the fact from the available count data; sometimes it is not. For example, in the program 'chess' (described later), the 'branch equal' opcode was 9.93% of all the opcodes executed. Since all branch equals are three cycles long and since we know chess had 1,797,514 cycles, we can calculate that 6.39% of the time was taken by branch equal. However, we cannot do the same calculation for load accumulator because we do not know the average number of cycles taken by the load accumulator instruction as it has several addressing modes. We could use the average number of cycles for each addressing mode, but this would only be an approximation.

Anyone looking to expand on the work I did in this report should try to find a method of obtaining the dynamic data by percent of execution time as well as percent of count. This data would be very useful.

Once the data was taken by the EXORciser based simulator, it was serially uploaded to the UNIX system where programs similar to the ones used for the static data were used to do the dynamic analysis. This was convenient since it meant that both the static and dynamic data were available on one computer.

5.2. PROGRAM MIX

Due to the limitations mentioned above, I was only able to analyze five programs dynamically. they are:

Chess	chess playing program
Ed	line editor
Mon	small monitor
Mopet	automatic test generator
M6839	floating point package

The chess program is a very high speed world-class chess program. The simulation consisted of simulating one medium complexity move in the middle of a game.

The editor is a simple line editor similar to the one described in Software Tools. [KER] It is written in assembly language. The simulation consisted of executing each editor

command at least once on a small file.

The monitor is a simple monitor used for debugging in a system where most of the programs are written in 'C'. It is written in assembly language. The simulation consisted of executing each monitor command once.

Mopet is a program that automatically generates and then executes breadboard tests. Mopet compiles its input, automatically generates the test cases, and then executes them. The simulation generated some test cases for a fictional breadboard, then tried to execute them. It is written in a mixture of Pascal and modular assembly language.

The M6839 is a floating point package that implements the IEEE standard for floating point. It was written in structured assembly language. The simulation consisted of calling each available function six times with varing precision data.

In addition to the dynamic data for each of these programs, I also have the static data. This allows comparisons to be made between the static and the dynamic data.

Although I would like to have analyzed more programs, I feel these programs are representative. However, I did not feel justified in concatenating all the data into one set of data as I did in the static data. In the following sections the dynamic data will be presented independently for each

simulated program.

The number of instructions and cycles in the simulation for each program is given below:

program	instructions	cycles
mopet	451,131	2,015,244
chess	385,698	1,797,514
M6839	163,370	719,976
ed	149,521	702,876
mon	86,406	477,887
total	1,236,126	5,713,497

5.3. CYCLES / INSTRUCTION AND MIPS

A metric of interest for a processor is the number of cycles taken by the average instruction and the millions of instructions per second (MIPS) for each program . Table 5-1 contains this data:

program	avg. cycles/instr.	MIPS at 2 Mhz
M6839	4.41	.454
mopet	4.47	.447
chess	4.66	.429
ed	4.70	.425
mon	5.53	.362
average	4.75	.423

Table 5-1: Cycles/Instruction and MIPS for the M6809

This data is fairly consistent and a MIPS rate of approximately .42 can probably be used to successfully estimate the execution speed of an M6809 program. The slightly higher number of cycles per instruction for the monitor seems to stem from the fact that it had a large number of relatively slow subroutine calls and returns.

5.4. MOST FREQUENTLY EXECUTED SINGLE OPCODES

Table 5-2 gives a matrix of the 10 most frequently executed single opcodes for the five simulated programs:

	chess	ed	mon	mopet	M6839
1.	beq	cmpa x	lda x	beq	ror x
2.	lda x	bne	cmpa #	leax	rol x
3.	ldb x	lda x	lbsr	bne	bra
4.	bne	beq	rts	cmpa #	blt
5.	leau	bra	pshs	lda x	lda x
6.	bmi	bsr	puls	bra	cmpa #
7.	jsr	ldx x	ldx x	cmpx dir	decb
8.	ldx #	sta x	ror	stx dir	bne
9.	inc	pshs	leax	ldx x	sta x
10.	sta x	bge	blo	ldx dir	cmpa #

where:
x = indexed
# = immediate
dir = direct

Table 5-2: Top 10 Most Frequently Executed Single Opcodes

The following single opcodes appeared at least three times in the top 10 single opcodes: lda x (5), cmpa # (4), bne

(4), sta x (3), beq (3), bra (3), and ldx x (3).

There are far fewer conclusions that can be drawn from this data than from the top 10 static data. I'm not sure whether this is due to the statistically smaller sample of data or whether it is characteristic of all dynamic data. I tend to favor the later explanation. I think the individual characteristics of the programmers and the applications begin to surface more in the execution of programs.

5.5. MOST FREQUENTLY EXECUTED OPCODES BY CLASS

Using the same classes of opcodes as described in the static data, we can determine what classes of instructions are most frequently executed. Table 5-3 is the union of the top 10 classes for each program. Note, that it takes 24 separate classes to get the union of the top 10 classes. This indicates, once again, that the dynamic data is not as consistent as the static data where the union of the top 10 only included 14 separate classes:

units = percent of executed instructions

class	chess	ed	mon	mopet	M6839
ld	19.20	11.87	10.10	8.04	8.76
beq	9.93	6.56	1.44	12.07	.59
ldl6	7.81	8.80	10.41	7.35	2.05
lea	5.06	4.82	3.72	10.95	2.92
st	4.85	5.01	1.49	2.19	4.22
stl6	4.23	3.58	3.29	4.26	1.88
bne	4.21	9.32	2.68	6.99	3.80
bit	3.49	0	.59	.32	.01
bmi	3.22	.06	.06	.04	.03
inc	2.89	.46	1.48	.32	.92
cmp	2.74	13.18	6.56	8.68	9.35
jsr	2.64	6.45	3.29	2.60	.05
psh	2.19	3.37	4.20	1.45	1.55
rts	1.97	3.04	4.90	2.48	2.30
pul	1.84	2.49	3.90	.77	.95
cmpl6	1.40	5.62	1.39	11.63	1.25
dec	1.21	0	.08	.44	4.97
bra	.95	6.47	.64	4.64	6.20
blt	.19	.09	0	.01	5.62
bcc(bhs)	.11	.27	.30	3.10	1.68
bcs(blo)	.07	.41	3.56	1.68	.27
lbsr	.04	.73	5.15	1.78	1.15
rol	.02	.08	.06	.09	10.96
ror	0	0	3.77	0	10.63
other	19.74	7.32	26.94	8.12	16.01

Table 5-3: Union of the Top Ten Classes (Dynamic)

Some of the top 24 classes seem to be prevalent in all the programs. Some of the other classes seem to be unique to a particular program and may not be valid data points for the average program. The classes that seem significant in all the programs are:

ld
ld16
st
st16
cmp
bne/beq
lea
psh/pul
rts
bra

Also from the union of the top 10 we see that the conditional branches are executed often, but it appears every programmer/compiler has its favorite type of conditional branches. For example, the monitor has 3.56% branch on carry sets and .30% branch on carry clears while the M6839 has .27% branch on carry sets and 1.68% branch on carry clears.

5.6. MOST FREQUENTLY EXECUTED BY LARGE CLASS

Even though the dynamic data for small classes showed a larger variation than it did for static, the variation for large classes is smaller. The larger classes remove more of the programmer and compiler preferred individual small classes. For example, whereas the small class data did not show any particular conditional branch that was very high in execution frequency, the larger classes clearly show that the conditional branches, as a large group, are very high in execution frequency. Table 5-4 contains the dynamic data for the large classes.

units = percent of total instructions executed

Class	chess	editor	monitor	mopet	M6839
load	27.01	20.67	20.51	15.40	10.81
cond_br	21.81	21.12	9.24	24.76	15.86
store	9.08	8.59	4.78	6.46	6.10
cmp_tst	6.07	18.97	8.32	22.14	12.63
arith	5.15	1.01	1.41	0.48	1.86
addr	5.06	4.82	3.72	10.95	2.92
inc_dec	4.10	0.46	1.56	0.76	5.89
psh_pul	4.03	5.87	8.10	2.23	2.50
call	3.18	7.18	10.66	5.54	2.97
logical	2.65	0.50	2.51	0.46	1.27
xfr	2.24	6.52	3.97	4.98	6.46
shifts	0.33	0.16	9.50	0.45	23.33
total	90.72	95.87	84.27	94.59	92.60

Table 5-4: Dynamic Data By Large Classes

Below is the union of the top three large classes for each program:

Class	chess	editor	monitor	mopet	M6839
load	27.01	20.67	20.51	15.40	10.81
cond_br	21.81	21.12	9.24	24.76	15.86
store	9.08	8.59	4.78	6.46	6.10
cmp_tst	6.07	18.97	8.32	22.14	12.63
call	3.18	7.18	10.66	5.54	2.97
shifts	0.33	0.16	9.50	0.45	23.33

In both static and dynamic frequency, loads and stores make up the largest class of instructions by far. Next in frequency in the dynamic data is the conditional branches. Compares and tests also have a high dynamic frequency. Calls have a high frequency, but not as high as in static. Probably

the most surprising result is that, in programs that have a lot of shifts to begin with, the dynamic frequency of the shifts is high.

5.7. DYNAMIC STATISTICS BY PERCENT OF CYCLES

As mentioned in the beginning of this chapter, due to memory constraints I was not able to instrument my simulator to take the dynamic data in such a way as to determine the percent of time (cycles) actually taken by a given opcode. I did have the total cycles, however, and with this information I was able to approximate the percent of cycles for some individual classes of opcodes and for some larger classes for the chess program.

As mentioned previously, some of the data is exact, some is not. For instructions that take an invariant number of cycles to execute, the percent of time can be calculated exactly. For instructions with memory reference addressing modes, I used the average values for the cycles. For example, load has 4 basic addressing modes (immediate, direct, extended, and indexed). The dynamic addressing mode data (in a later section) gives the percentage of execution of these modes. Using this percentage, I calculated the average number of cycles required by a load.

Table 5-5 contains the execution data for the chess

program for some classes. Table 5-6 contains the same data for larger classes. Both percent of instructions and percent of cycles is given in both tables, and both are sorted by the percent of cycles.

op	% instr.	% cycles
ld	19.20	18.00 *
ld16	7.81	9.11 *
beq	9.93	6.39
lea	5.06	5.65 *
st16	4.23	5.47 *
st	4.85	5.21 *
jsr	2.64	4.54 *
inc	2.89	4.31 *
psh	2.19	3.37
bit	3.49	3.28 *
clr	1.97	2.93 *
tst	1.93	2.87 *
bne	4.21	2.71
cmp	2.74	2.57 *
add	2.59	2.43 *
rts	1.97	2.12
bmi	3.22	2.07
cmp16	1.40	2.01 *
dec	1.21	1.80 *
and	1.32	1.16 *

* estimated

Table 5-5: Chess by Percent of Instructions and Cycles

Class	% instr.	% cycles
load	27.01	27.11
cond_br	21.81	14.60
store	9.08	10.68
cmp_tst	6.07	7.45
psh_pul	4.03	6.19
inc_dec	4.10	6.11
addr	5.06	5.69
arith	5.15	5.65
call	3.18	5.36
logical	2.65	2.48
xfr	2.24	1.77
shifts	0.33	.49
total	90.72	93.58

Table 5-6: Percent of Cycles for Large Class (Chess)

This data indicates that although conditional branches are a large percentage of the instructions executed, they are not as large a percentage by cycles. Further, the 16-bit operations and read/modify/write instructions are larger by cycles than by execution count. Stores take slightly more cycles than their count might suggest. This probably occurs because stores don't include the faster immediate addressing mode. Also the subroutine calls, returns and push/pulls take a larger percent of cycles.

5.8. DYNAMIC EXECUTION OF ADDRESSING MODES

This section presents the dynamic addressing mode data collected by the simulator. Table 5-7 contains the frequency of execution of the various addressing modes for the

five programs.

mode	chess	ed	mon	mopet	M6839
indexed	40.79	33.74	29.76	31.05	41.46
short_relative	22.23	27.48	12.09	30.46	23.78
immediate	14.23	18.47	15.46	12.12	11.49
direct	8.33	0.00	0.00	14.29	0.00
inherent	7.21	9.77	23.19	5.77	6.73
extended	3.90	8.93	3.29	2.90	0.24
accumulator_a	1.51	0.09	10.33	0.84	9.07
long_relative	1.08	0.88	5.18	1.98	1.27
accumulator_b	0.72	0.64	0.70	0.59	5.95
indirect	0.69	0.00	0.00	0.04	0.15

Table 5-7: Dynamic Addressing Mode Usage

In all five programs indexed addressing is by far the most executed addressing mode. In fact, its dynamic frequency is about 10% higher than its static frequency. Short relative addressing is a strong second with immediate third. If a future architect were looking to improve the performance of the M6809, it would be advantageous to look at speeding up relative and indexed addressing. (More about this later.) Indirect addressing is the big loser.

5.8.1. Indexed Addressing Dynamic Statistics

Since the frequency of indexed addressing is so high, it is worthwhile to see how indexed addressing is being used dynamically. The first parameter of interest is how many

additional cycles are added by the average indexed instruction. The following data is from the simulations:

program	average cycles
mopet	.77
M6839	1.ØØ
ed	1.16
chess	1.21
mon	1.47

This indicates that every indexed instruction adds between .77 and 1.47 cycles to its base time. This is reasonable in light of the features provided by indexed addressing on the M68Ø9. Table 5-8 contains the indexed addressing breakdown for the five programs analyzed.

addr mode	chess	ed	mon	mopet	M6839
+	2.70	34.76	21.26	6.78	1.68
++	1.57	1.46	0.00	3.06	0.06
-	1.15	0.02	0.05	4.26	0.19
--	1.60	0.00	0.00	0.00	0.38
0 offset	17.03	23.34	7.47	45.95	10.96
5 offset	62.00	39.80	37.82	37.35	62.32
8 offset	2.30	0.01	0.00	0.22	2.55
16 offset	6.77	0.01	0.00	1.21	0.00
pc8 off.	0.00	0.06	11.21	0.02	0.14
pc16 off.	0.00	0.02	0.00	0.23	1.53
a offset	3.34	0.08	11.06	0.37	3.91
b offset	0.35	0.00	0.00	0.26	15.93
d offset	0.42	0.43	11.13	0.27	0.35
ext. ind.	0.77	0.00	0.00	0.00	0.00

Table 5-8: Dynamic Indexed Addressing Statistics

The 5-bit and no offset indexed addressing are by far the most frequently executed. If it were possible to make these faster, it would certainly improve the M6809's performance. Auto increment by 1 is used fairly often in an executing program. This is expected as almost all auto increment and decrements are in loops by definition. The accumulator indexed frequencies are lower than expected and are eclipsed by the offset varieties.

POSSIBLE PERFORMANCE IMPROVEMENTS TO THE M68Ø9

The dynamic data in the previous chapter lead me to consider what improvements could be made to the M68Ø9 instruction set to increase its performance. Admittedly, I know some of the details of the internals of the M68Ø9 which makes it easier for me to predict what is possible and what is not. Unfortunately, I cannot divulge in this report what I know for proprietary reasons.

The data I will use to justify these recommendations is the dynamic data. I can calculate the total cycles saved by a proposed improvement and compare that with the total number of cycles for all the dynamic data (5,713,497 cycles). This gives me the percentage of the throughput improvement.

6.1. INSTRUCTION SET IMPROVEMENTS

When looking at the static and dynamic data for the long conditional branches, it is obvious that long branch equal and long branch not equal occur more frequently than any other page 2 conditional branches. Should lbne and lbeq be included on page 1 in future M68Ø9's? If this made lbne and lbeq one cycle faster, it would improve performance .Ø7%. Even if it saved two cycles, it would only amount to .14%. This improvement is not worth it.

What if all long branches were made 1 cycle faster?

This would save 20,940 total cycles for a 2.73% throughput improvement. Definitely worthwhile if it can be done.

By comparing the M6809 to the internally faster M6801, it seems reasonable to assume that one cycle could also be removed from the instructions listed below.

	throughput improvement
All load effective address	1.47 %
All subroutine calls (jsr, bsr, lbsr)	1.09 %
All pushes and pulls	.80 %
	3.36 %

Adding these savings to the savings for long branches we get 6.09 % improvement by speeding up the suggested instructions.

6.2. ADDRESSING MODE IMPROVEMENTS

Again the M6801 leads me to believe that it should be possible to remove one cycle from all direct and extended instructions. Further, with the proper changes to the internal architecture of the M6809, I think we could save a cycle on every indexed instruction. This gives the following savings:

```
All direct    = 1.69 %
All extended =  .78 %
All indexed   = 7.70 %
                -----
                10.17 %
```

In addition, I think the 5-bit offset indexed mode could be reduced by one additional cycle. Although their frequencies are low, I think the change necessary to the internal architecture to speed up the 5-bit offset would also speed up the auto increments and decrements by two. These improvements would improve the performance by 4.05%.

6.3. PUTTING IT ALL TOGETHER

The following table summarizes all the proposed performance improvements:

Remove 1 cycle from:

All long branches (including lbra and lbsr)	=	2.73 %
All subroutine calls (including lbsr)	=	1.09 %
All load effective addresses	=	1.47 %
All pushes and pulls	=	.80 %
All direct addresses	=	1.69 %
All extended addresses	=	.78 %
All indexed	=	7.70 %
5-bit indexed and auto inc/dec by 2	=	4.05 %
total throughput improvement	=	20.31 %

Table 6-1: Possible Performance Improvements

A 20% throughput improvement may not sound like much compared to the typical marketing claims of 5, 10, 100 times throughput improvement, but in reality a 20% throughput improvement is significant.

In closing this chapter, let me make it clear that these are only the improvements that jumped out at me as I was doing this report. I do not consider the main objective of this report to be to make recommendations to future architects. The main objective is to present the data so that future architects can draw their own conclusions.

COMPARISON OF DYNAMIC VS. STATIC DATA

This chapter compares the dynamic data gathered on the five programs that were simulated to the static data taken from the same five programs. The purpose is to help answer the question of whether static data can be used to predict the dynamic characteristics of a processor.

7.1. INSTRUCTION SET COMPARISON

In the following sections the dynamic data for the larger classes for each program will be compared to the static data for the same program.

7.1.1. Chess

Below is the dynamic and static data for large classes for the chess program.

Class	dynamic % (by ct.)	static % (by ct.)
load	27.01	24.17
cond_br	21.81	17.44
store	9.08	7.36
cmp_tst	6.07	5.60
arith	5.15	4.30
addr	5.06	5.39
inc_dec	4.10	6.14
psh_pul	4.03	3.03
call	3.18	9.48
logical	2.65	2.90
cntrl_xfr	2.24	4.33
shifts	0.33	0.73
total	90.72	90.87

Table 7-1: Dynamic and Static Data For: Chess

The loads and conditional branches are slightly higher in dynamic execution. The calls and control transfers are higher in static appearance.

7.1.2. Editor

Below is the dynamic and static data for large classes for the editor program.

Class	dynamic % (by ct.)	static % (by ct.)
cond_br	21.12	13.58
load	20.67	24.93
cmp_tst	18.97	12.60
store	8.59	10.86
call	7.18	9.26
cntrl_xfr	6.52	4.94
psh_pul	5.87	10.58
addr	4.82	5.43
arith	1.01	1.74
logical	0.50	0.14
inc_dec	0.46	0.42
shifts	0.16	0.42
total	95.87	92.20

Table 7-2: Dynamic and Static Data For: Ed

Again this program indicates that the dynamic execution of conditional branches is greater than the static appearance and that the calls and pushes and pulls are less. A new data point here is the higher percentage of compares and tests in the dynamic data.

7.1.3. Monitor

Below is the dynamic and static data for large classes for the monitor program.

Class	dynamic % (by ct.)	static % (by ct.)
load	20.51	13.95
call	10.66	14.09
shifts	9.50	2.37
cond_br	9.24	12.27
cmp_tst	8.32	5.30
psh_pul	8.10	7.53
store	4.78	8.51
cntrl_xfr	3.97	6.56
addr	3.72	7.11
logical	2.51	1.53
inc_dec	1.56	1.39
arith	1.41	2.23
total	84.27	82.84

Table 7-3: Dynamic and Static Data For: Monitor

The dynamic loads are much higher in this case than the static. The calls and control transfers are lower. The shifts are much higher in execution frequency than in appearance. Strangely, the conditional branches are actually less in dynamic execution for this program. This is counter to all the other programs. Both the store and address manipulation instructions were higher in static appearance than in dynamic execution.

7.1.4. Mopet

Below is the dynamic and static data for large classes for the mopet program.

Class	dynamic % (by ct.)	static % (by ct.)
cond_br	24.76	12.72
cmp_tst	22.14	9.01
load	15.40	19.52
addr	10.95	8.02
store	6.46	7.43
call	5.54	15.51
cntrl_xfr	4.98	8.68
psh_pul	2.23	7.43
inc_dec	0.76	0.96
arith	0.48	0.42
logical	0.46	1.05
shifts	0.45	0.45
total	94.59	91.20

Table 7-4: Dynamic and Static Data For: Mopet

This program conforms to the trend of branches and compares/tests being higher in dynamic percent; and calls, control transfers, and push/pulls being lower.

7.1.5. M6839

Below is the dynamic and static data for large classes for the M6839 program.

Class	dynamic % (by ct.)	static % (by ct.)
shifts	23.33	7.62
cond_br	15.86	11.15
cmp_tst	12.63	6.10
load	10.81	19.28
cntrl_xfr	6.46	6.86
store	6.10	10.21
inc_dec	5.89	3.02
call	2.97	10.18
addr	2.92	8.46
psh_pul	2.50	3.14
arith	1.86	2.24
logical	1.27	3.93
total	92.60	92.19

Table 7-5: Dynamic and Static Data For: M6839

This program also has more dynamic conditional branches, compare/tests, and shifts than static. It has fewer dynamic loads, stores, calls, pushes and pulls, and control transfers in dynamic than static.

7.1.6. Short vs. Long Relative

Table 7-6 contains the comparison of the short and long relative branches for both the dynamic and static data.

program	short		long	
	dynamic	static	dynamic	static
chess	95.38	94.89	4.62	5.11
ed	96.88	80.90	3.12	19.10
mon	70.02	49.57	29.98	50.43
mopet	93.91	59.64	6.09	40.36
M6839	94.93	65.09	5.07	34.91

Table 7-6: Short vs Long Branches

The percentage of short branches increases when the program is executed. The percentage of long branches decreases when the program is executed. I feel this occurs because the short branches are in the loops and the long branches are not.

7.1.7. Instruction Set Comparison Conclusions

None of the program statistics for dynamic looked greatly different than those for static. In both cases, the large classes selected account for about 90% of all instructions. However, the following characteristics seemed to run through most of the programs:

1. Conditional branches, compares, and tests have a higher dynamic percentage of execution than static percentage of appearances in the source program. This is intuitive since the branches, compares and tests are likely to be controlling the loops.

2. In programs that have a lot of shifts statically, the number of shifts executed when the program is simulated is even higher. This conclusion isn't as easy to understand; possibly, shifts are done in loops which means they are probably multi-precision or multi-bit shifts.

3. Subroutine calls, control transfers, and pushes and pulls appear more often in a source than they are actually executed. Apparently subroutine calls are generally not found in the tight loops because of their inefficiency (in-line code is used instead). In structured programs, it would be expected that control transfers would not be in loops.The pushes and pulls are usually setup and cleanup code and, therefore, do not appear in loops.

4. In most cases, but not all, the number of loads and stores that appear in the source are a greater percentage than the percentage executed. Apparently this occurs because of the large number of loads that are used in the initialization of loops.

5. Short relative is also executed more often while long relative is executed less often. I suspect this indicates that the short relative branches are used in loops, and the long relative branches are used to

get from one part of the program to another.

Knowing these basic facts, it should be possible to do a static analysis on some program and predict, fairly accurately, what the dynamic instruction set characteristics will be.

7.2. ADDRESSING MODE COMPARISON

Table 7-7 contains the percentage difference between the static and dynamic data for the addressing modes. A plus number indicates that the dynamic data had a higher percentage of that particular addressing mode than did the static data. A minus indicates that the particular addressing mode appeared more frequently in the source than it was executed. The number in the table is the amount the two percentages differed.

Addr. Mode	chess	ed	mon	mopet	M6839	all
indexed	+11.02	+ 7.35	+ 7.17	+ 5.15	- 1.97	+
short rel.	+ 1.06	+11.25	- 4.09	+ 9.44	+ 5.53	+
immediate	- 3.13	- 3.47	- .58	- 3.33	- 2.32	-
direct	- 1.93	- 6.41	- .56	+10.25	0	?
inherent	+ .27	- 4.78	+ 2.83	- 8.15	- 0.34	?
extended	- 5.89	- .19	+ .50	- .51	+ .24	-
accum. a	- .41	- .68	+ 6.98	- .18	+ 5.05	?
accum. b	- .94	- .13	- .97	- .43	+ 2.32	?
long rel.	- .06	- 2.95	-11.28	-12.24	- 8.52	-

\+ = more dynamic
\- = more static
value = % dynamic - % static

Table 7-7: Comparison of Addressing Modes

In most cases indexed addressing is used more in actual execution than its static appearance would suggest. Short relative is also executed more often than anticipated while long relative is not. Immediate and extended were executed less than would be expected. For the other addressing modes the data is inconclusive.

7.2.1. Indexed Addressing Comparison

Now we will make a similar type of table for indexed addressing:

addr mode	chess	ed	mon	mopet	M6839	all
+	.00	+27.37	+10.77	+ 3.54	+ 1.05	+
++	+ .44	+ .14	- 1.23	+ 1.67	- .15	?
-	- .24	- .24	- 3.65	+ 3.10	+ .05	?
--	+ 1.16	0	0	- .46	+ .38	?
0 offset	+ 1.71	- 4.89	-17.84	+25.83	- .73	?
5 offset	+ 5.26	-19.30	- 1.69	-10.28	+ 5.05	-
8 offset	- 4.14	- .78	- 1.85	- 3.71	-14.29	-
16 offset	- 6.28	- 1.05	- 3.70	-12.55	0	-
pc8 off.	0	- .47	- 3.19	- .33	- .71	-
pc16 off.	0	- .24	- 4.32	- 4.63	- 2.02	-
a offset	+ 2.03	- .16	+10.44	+ .25	- .20	+
b offset	- .17	0	0	- 1.71	+11.55	+
d offset	+ .16	- .36	+ 9.90	- .65	+ .07	+

\+ = more dynamic
\- = more static
value = % dynamic - % static

Table 7-8: Comparison of Indexed Addressing Modes

There is a wider variation here than for either the instructions or the addressing modes in general. The particular indexed addressing mode used in loops seems to be either application or programmer dependent.

Auto increment by one is used more in execution than the static data would indicate. For the other increments and decrements the data is inconclusive.

Generally speaking, the constant offset variety indexed instructions had a slightly lower dynamic than static frequency. I'm not quite sure I know why.

The register offset indexing had a higher dynamic execution frequency than static appearance.

7.2.2. Addressing Modes Conclusions

The following conclusions can be gathered from the comparison of the addressing modes in static and dynamic modes.

1. Indexed addressing occurs more frequently in the dynamic cases.

2. Immediate and extended were executed less frequently in the dynamic case.

3. Auto increment and the register offset indexed are executed more frequently in the dynamic case.

4. The constant offset indexed are executed less frequently in the dynamic case.

It is not clear whether it would be safe to use static data to predict the dynamic characteristics of addressing modes. The data was more variable than the instruction set data and seemed more application dependent.

CONCLUSIONS

The following sections contain a summary of the conclusions for each section of the paper.

8.1. STATIC DATA CONCLUSIONS

The average instruction size for the M68Ø9 is approximately 2.3 bytes.

As is the case with most Von Neuman architectures, a very few single opcodes make up a large percentage of all the instructions that appear. For the M68Ø9 the top 2Ø opcodes accounted for over 58% of all the instructions. Three new M68Ø9 instructions headed up the list of the most frequently appearing single opcodes. They were: lbsr, leax and pshs. The rest of the top 2Ø was composed of loads, stores, branches, compares, subroutine calls, and subroutine returns.

By larger classes of instructions, the following statistics were the approximate values for the top 5 classes:

loads and stores	= 36 %
subroutine calls	= 12 %
conditional branches	= 1Ø %
pushes and pulls	= 8 %
Load effective address	= 6 %

The arithmetic and logical instructions had low frequencies of occurence. The long conditional branches, except

for lbne and lbeq, had very low static frequencies.

When analyzing the percentage of bytes taken for page 1 opcodes versus 2 and 3, we find that over 90 % of all the bytes were for page 1 instructions.

When the data was analyzed by groups to see what instructions were present most often, the following conclusions were drawn:

1. There are three times more loads than stores.

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

3. Programmers are using relative and long relative addressing ; and, therefore, are writing position independent code.

The multi-register push/pull statistics indicate that 2.25 registers are pushed or pulled per push/pull instruction. I would have been happier if this ratio had been higher; but, at least, it is greater than the 2.0 break even point.

The direct page register statistics, on the other hand, are a big disappointment. Clearly, it was a mistake to include the direct page register in the M6809 and to expand the read-modify-write instructions to include direct addressing.

The static addressing mode data indicates that the most common addressing mode is indexed (30%), followed by relative (24%), and immediate (20%). The number of direct and extended instructions combined was only 10%. Indirect was the big loser with practically zero occurences.

In the static indexed addressing data we find that 5-bit and no offset indexed account for 66% of all indexed instructions. Including the 8-bit, 16-bit, 8-bit program counter relative, and 16-bit program counter relative modes we find 86% are constant offset or have no offset. The average number of bytes added for each indexed instruction is 1.17 bytes.

8.2. PAIRS AND TRIPLES CONCLUSIONS

In general, not much showed up to indicate that there are any easy new instructions that could be included in future M6809's.

The pairs data did indicate that a memory to memory move instruction would be useful. I don't think this should be a surprise to anyone. Unfortunately, there is no way to implement a memory to memory move on the M6809 except to use a two byte opcode which would defeat most of the benefit.

Both the pairs and the triple data indicate that an M68000 type move multiple instruction would be useful. This

may be feasible on future M6809's.

The triple data did not show a great need for the bit manipulation operations. There was some evidence that 16-bit memory to memory adds and subtracts would be useful. Lastly, there is some inconclusive evidence that a multi-bit or multi-precision shift would be useful.

8.3. M6809 STATIC VS. M6800 STATIC DATA CONCLUSIONS

The instruction set usage has not changed much between the two processors except for the greater number of pushes and pulls and address manipulation instructions on the M6809. Also, the relative forms of the subroutine calls and control transfers have replaced the absolute versions found on the M6800.

The addressing mode usage varies a great deal. The data indicates that the M6809 programs use more indexed and relative addressing and less absolute addressing than do comparable M6800 programs. This implies that the M6800 was basically an absolute address machine without much modularity. The M6809 is a relative, position independent, modular machine. The new addressing modes on the M6809 have substantially changed the 'feel' of the M6809 to the programmer as compared to the other 8-bit Motorola microcomputers.

8.4. DYNAMIC DATA CONCLUSIONS

There was a larger variation in the dynamic data. I feel there are two reasons. One is that I had less dynamic data points. The other is that dynamic data seems to be more dependent on the application and programmer style.

The average number of cycles for an M68Ø9 instruction is approximately 4.75. This gives a throughput of .423 MIPS with a 2 Mhz. M68Ø9.

There was a larger variety of single opcodes executed in the five programs analyzed for the dynamic data than for the static data. The top runners were: load indexed, store indexed, compare immediate, and branches.

By class the data was: load and store, compare, beq, bne, lea, rts, bra and the pushes and pulls. Further, there were a lot of conditional branches, but they were varied.

By larger classes the conditional branches all combined to form the second most executed group, second only to the loads and stores. The other large classes were compare and test, the calls, and the shifts.

If the dynamic data is calculated to determine which opcodes spend the largest time actually being executed, we find that the percentage of time taken by the conditional

branches is less than the dynamic data taken by percent of instructions executed would indicate. On the other hand, the 16-bit operations, the subroutine calls and returns, and the pushes and pulls take more time than their frequency by count would indicate.

In dynamic execution the indexed addressing mode accounts for approximately 35% of all addressing modes. Short relative is about 25%, and immediate is 15%. Long relative addressing usage is fairly low. Indirect is the big loser again.

In indexed addressing, the offset varieties accounted for 72%. This is down from the static data, but is still impressive. Auto increment and the accumulator offsets make up most of the rest of the indexed data.

8.5. POSSIBLE PERFORMANCE IMPROVEMENTS CONCLUSIONS

The table from chapter 7 is repeated here:

Remove 1 cycle from:

All long branches (including lbra and lbsr)	= 2.73 %
All subroutine calls (including lbsr)	= 1.09 %
All load effective addresses	= 1.47 %
All pushes and pulls	= .80 %
All direct addresses	= 1.69 %
All extended addresses	= .78 %
All indexed	= 7.70 %
5-bit indexed and auto inc/dec by 2	= 4.05 %
total throughput improvement	=20.31 %

Table 8-1: Possible Performance Improvements

8.6. COMPARISON OF DYNAMIC VS. STATIC - CONCLUSIONS

The following instruction set conclusions seem to hold for all the data:

1. Conditional branches, compares and tests have a higher dynamic frequency than static frequency.

2. Calls, control transfers, address manipulation, and the pushes and pulls have a lower dynamic frequency than static frequency.

3. In programs that have a lot of shifts in the source code, there will be an even greater number actually