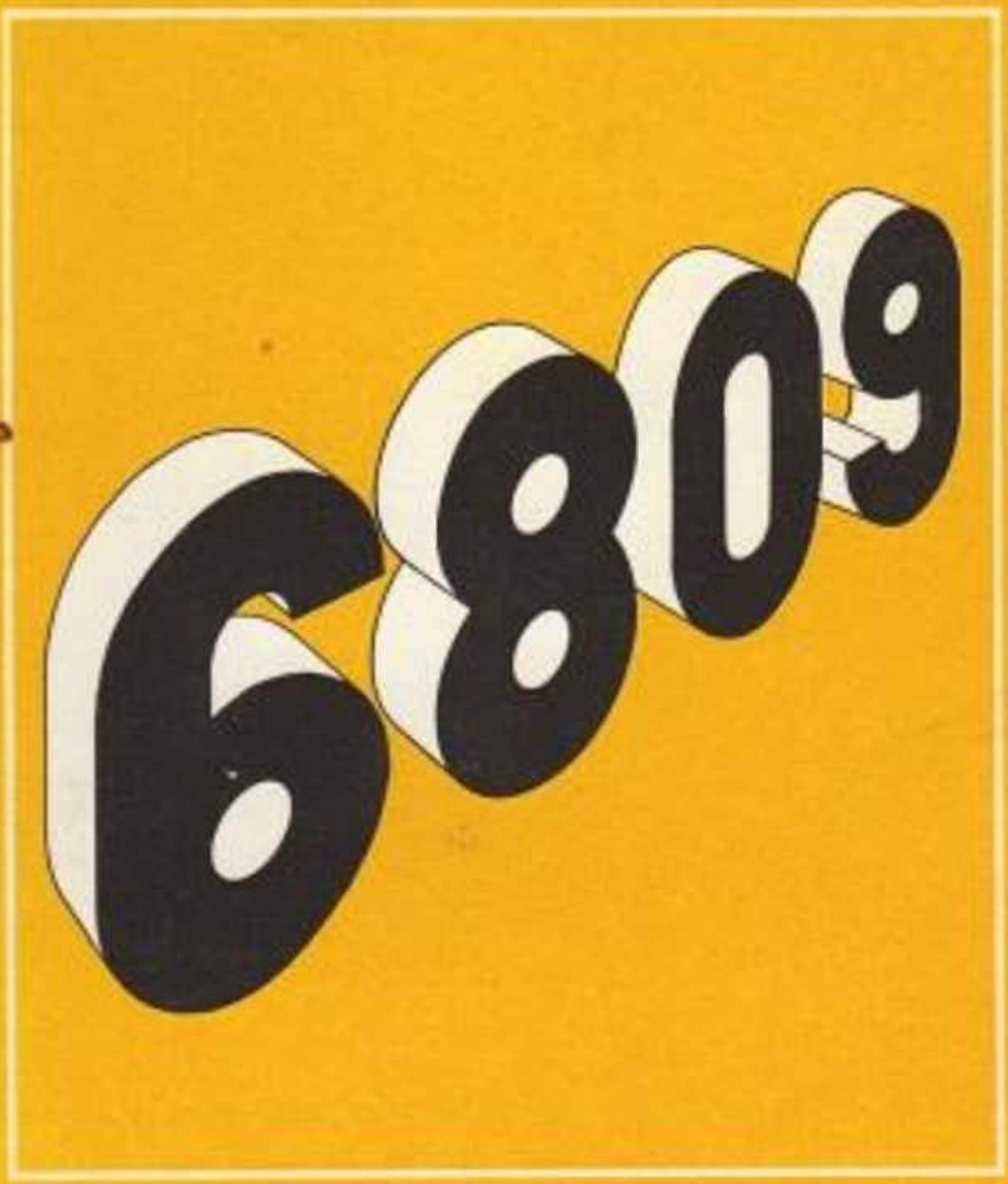
The 6809 Companion

M. JAMES



THE
ABBEY
COMPANION

by

B. JAMES, D.Phil., F.R.E.C.S.

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

First Published - February 1982

British Library Cataloguing in Publication Data
James, M
The 6809 companion.
1. Motorola 6809 (computer)
I. Title
001.64'04 QA76.8.M6/

ISBN 0 85934 077 5

Printed and Bound in Great Britain by Mayhew McCrimmon Printers Limited

PREFACE

"The 6809 Companion" was written for the average assembly language programmer. It is not a beginner's introduction to microprocessors in general but a discussion of the features of the 6809 and a reference work for the 6809 programmer in particular.

The 6809 is becoming more and more important as industry and technology moves on to more sophisticated microprocessors. Tandy have recently introduced the TSR 80 color computer based on the 6809, Commodore have chosen a 6502/6809 combination for their Micro/main frame MOMF 9000, a 6809 card, "The Mill", has been made available for the Apple, and Apple themselves are rumoured to be working on a new 68000 based system. With the big three of personal computing leaving the 8080/Z80 behind, there must be something good about Motorola's products !

The early chapters cover the elementary detail of the 6809 and its assembly language. Later chapters deal with more advanced topics such as interrupt handling and programming style. Chapter seven is included for the 6800 programmer needing help in converting programs to the 6809. Although the subject of this book is tackled from a programmer's point of view a brief examination of some common hardware using the 6809 is given in chapter eight.

I have attempted to include in one book as much as possible of the information that I use in my own everyday programming of the 6809. This is a difficult task in that in order to keep the book to a reasonable length much of the obvious and exotic has to be excluded, thus risking making the text frustrating to the beginner and expert alike ! My selection of important items has been guided by consideration of what a programmer used to programming another machine would need to know to recognise what was special about the 6809.

This text was prepared using a combination of the following equipment and software:

SWTP 128K UNIFLEX SYSTEM, TSC TEXT PROCESSOR, CT82 VDU
GIMIX 56K FLEX SYSTEM, TSC TEXT PROCESSOR/EDITOR
CENTRONICS 737 PRINTER
DATAMAX TEXT PROCESSOR

CONTENTS

CHAPTER ONE
INTRODUCTION TO THE 6809

The first microprocessors were designed as replacements for the large numbers of integrated circuits employed as control devices, in for example traffic light sequencers. They hardly resembled computers at all and as a result were mainly used by electronic engineers and ignored by programmers. As they developed, the early micros had difficulty in forgetting their early incarnations and so tended to be odd mixtures of data processing computers and process control devices. The difficulty was that once a manufacturer had introduced a processor that was as advanced as current technology would allow, they later found it difficult to produce a radically new device. This was because their existing customers were all reluctant to change to a new device. Thus the idea of upward compatibility was introduced.

Instead of making completely new devices, manufacturers would improve (sometimes considerably) their early micros by adding new features. Old customers could make use of these extensions as and when they chose. This upward compatibility approach means that, although improvements are possible, it is impossible to design out an early fault or shortcoming. A typical example of this problem is the 8080 family of micros by INTEL. The 8080 is a development of the very early 8008 device but it is still very popular. To overcome its limitations Zilog introduced the Z80 which contained all of the features of the 8080 plus a lot more and is generally considered to be a very powerful micro. However the Z80 is often considered to be the best of a bad job. In other words if the Z80 did not have to imitate the very successful 8080 it could be a much easier to use and more powerful device.

THE 6800

Motorola came to the microprocessor market rather later than INTEL and designed their first microprocessor with a great deal more care. The resulting product, the 6800, seemed at first sight less powerful than its direct competitor the 8080. It had fewer registers and seemed a lot simpler. However both of these apparent failings contributed to why the 6800 is powerful. Each one of its small number of registers had a particular well defined usage and its instruction set reflected these functions. In other words its architecture was well thought out. Other reasons for the 6800's popularity were the existence of a wide range of peripheral chips for various input/output functions and a mass of well written documentation, both making system design simple.

The 6800 had a number of imitators such as the 65xx range from MOS Technology but Motorola's own improvements remained on the lines of special purpose versions of the 6800 rather than new devices. This again reflects the good initial design of the 6800. It must be admitted, however, that the 65xx range has become better known than the 6800 mainly because the 6502 device has been used in a number of personal

computers e.g. PET, VIC, APPLE, SUPERBOARD, ATOM, etc. For anyone used to the 6800, the 6502 is a very poor second, from the point of view of ease of use, and the marginal increase in power is obtained at a very high price.

THE 6809

Even though the 6800 was well designed, integrated circuit technology moved on and it became possible to produce more powerful processors. The 6800 had a number of well known design difficulties - only one index register, no direct transfer from accumulators to index register, not enough addressing modes, no sixteen bit arithmetic, incomplete test and compare commands on the index register etc., etc. In answer to criticisms, Motorola produced the 6809 (in 1980) as its first real alternative to the 6800. It was not upward compatible with the 6800 in the usual sense - i.e. a program written on the 6800 will not run on the 6809 - but it is clearly related to it and anyone familiar with the 6800 has little difficulty in using the new device. The standard 6800 registers are all present A, B, X and S along with the much-wanted new index register Y and a new stack pointer U. New instructions have been added along with many new addressing modes. Some of the 6800 instructions are no longer available and the operation codes of those that are, are different on the 6809. However, because of the similar architecture of the 6809, any 6800 programs can be converted from source code very easily (see chapter seven).

The 6809 is a microprocessor suitable both for large systems, data processing, high-level languages etc. and small scale process control applications. The reason for this ability to cope with both types of problem is the unique mixture of sophisticated features, such as the second stack pointer and the ability to push or pull register sets onto the stack, and the attention to basic design features, such as the provision of a fast interrupt line and the direct page register. It is probably true to say that the success of the 6809 processor is due to it being a sophisticated device that hasn't forgotten some simple microprocessor necessities.

THE 68000

The announcement of Motorola's plans for the 68000 microprocessor has tended to overshadow the 6809. So much has been written about the 68000 in the popular computer press concerning how powerful it is and whether or not it is within the limits of the current technology, that the fact that the 6809 is in use in all sorts of systems has taken a back seat. The 68000 is, or will be, a very powerful micro. In fact it promises to be closer to a mini or a small mainframe processor. With the promise of the first of the true sixteen bit processors on the horizon it might be difficult to see where the 6809 fits into the future. Will the changeover to the 6809 be short-lived, followed by a changeover to the 68000? For some users this will be the case, but for the majority the 68000 will probably be too powerful and too expensive for standard applications. After all the 6800 is still with us !

COMPARATIVE POWER OF THE 6809

It is very difficult to say how "powerful" a microprocessor is. It depends very much on the application to which the processor is to be put. The standard way of comparing computers is to use a "benchmark". A benchmark is a problem or a program that can be run on a range of computers so that their performance can be measured. The things that interest us about the way the benchmarks run are various but include how fast it runs, how short the resulting code is, how simple the code is etc. The trouble with benchmarks is that they can be very misleading. If you want to make a particular computer look good, it is always possible to find a problem that it solves faster, or in a smaller space than any other. The only safeguard against this sort of bias is to average the results of a wide range of benchmarks. Of course this doesn't imply that the computer that performs well over a wide range is the one to choose for your particular very specalised application ! With these comments in mind, table one gives the relative speeds for some popular processors for a range of benchmarks. It is also important to compare the speeds with which processors execute their high-level languages. Table two gives timings of a standard benchmark on a range of processors including some mainframes.

Table One

6809	100%
Z80	64%
6502	46%
6800	58%
8080	42%

Note : each percentage refers to the average amount of the benchmarks each processor completes (with the 6809 as 100%).

Table Two

Program and Processor	Time
TRS-80 Level II Basic	6h 21m 10s
Z80 assembler	22m 50s
6809 TSC Basic	4h 17m 10s
6809 Lucidata Pascal	2h 16m 0s
6809 assembler-pos.indep	6m 50s
6809 assembler-direct	6m 10s
IBM 370/148 assembler	56s
6809 Unifloc Pascal	34m 25s

(This table first appeared in 68' Micro Journal Vol 3 no 4)

CHAPTER TWO
REGISTERS AND OPERATIONS

The 6809 has relatively few registers when compared to a micro such as the Z80 but each register has a well thought out function. As will become clear later, the design of the 6809 was influenced by the needs of programmers rather than what was easy to fabricate on a single chip. For a programmer, computer architecture is about what registers and operations are available on a given machine. The registers can be of three types :-

Accumulators - for carrying out data manipulation such as arithmetic, logic, etc.
Address pointers - for pointing to areas of memory and manipulating addresses
Status registers - for indicating the past/future condition of the machine.

In real machines not all of the registers fall clearly into these types but it does help to know if a register is more like an accumulator than an address pointer.

A PROGRAMMING MODEL

From the programmers' point of view of the 6809 there are five sixteen bit registers and four eight bit registers, two of which can be used as an additional sixteen bit register. These are:

NAME	FUNCTION	SIZE	
X	index	16	
Y	index	16	
U	user stack pointer	16	
S	hardware stack pointer	16	
PC	program counter	16	
D	D accumulator	16	(A + B accumulators)
A	A accumulator	8	
B	B accumulator	8	
DP	direct page register	8	
CC	condition code register	8	

The registers can be grouped into two main types - address pointer registers X, Y, U and S and the accumulators A, B and D. The program counter, direct page and condition code register have to be treated separately. The pointer registers can be further subdivided into index registers X, Y and stack pointers U, S. We will deal with each type in turn.

THE ACCUMULATORS A, B, D

The A and B registers are eight bit general purpose accumulators capable of being used for arithmetic and logic calculations. The usual form of operation is between a memory location and the current contents of the accumulator with the result being left in the accumulator. The range of operations available are: binary addition (with or without carry); decimal addition via a decimal adjust instruction; binary subtraction (with or without borrow); binary negation (2's complement); decimal addition via a decimal adjust instruction; AND; OR; NOT; BCD arithmetic and logical shifts. The most remarkable accumulator operation on a machine of this size is an unsigned multiply. This instruction multiplies the two accumulators together and places the answer in the D register (see below). In general the A and B registers are identical and may be used interchangeably. The exceptions to this rule are DDA (decimal adjust the A accumulator) and ABX (add the B register to the X index register).

A feature which increases the power of the 6809 considerably is the ability to use the A and B registers as one sixteen bit accumulator for some operations. This is referred to as the D register, and is formed with the A register as the most significant byte. The range of arithmetic and logical operations that involve the D register includes: sixteen bit binary addition; sixteen bit binary subtraction; sign extended; sixteen bit compare; and the usual transfer, exchange and load operations. No logical operations or shifts are available on the D register.

THE DIRECT PAGE REGISTER (DP)

The direct page register is a single eight bit register which is used to specify bits eight to fifteen in direct paged addressing (see - addressing modes). The direct page register is cleared on hardware reset. Only four instructions can refer to the direct page register:- PUS/PUL and TFR/EXG. This limited set of operations is perfectly adequate for the uses to which the direct page register is put.

THE INDEX REGISTERS X, Y

The index registers X and Y are used in the indexed mode of addressing to specify sixteen bit addresses in a wide variety of ways. Although only the X and Y registers have been included in this section, both the S and U registers can be used as index registers with no restrictions. The reason that the S and U registers have a section to themselves is that they have some extra instructions which makes them more useful as stack pointers.

The range of operations that can be carried out on the index registers is limited when compared with the accumulators, but is adequate to their purpose. The index registers (X, Y, U & S) can be loaded and stored, exchanged and transferred as would be expected. In addition there are a number of important operations that are worth separate comment. First, the 6809 has a true sixteen bit compare instruction which can be used to test not only for equality

but also for greater than and less than relationships. Second, the 6809 has a Load Effective Address instruction, LEA. This calculates the address specified by an index addressing mode (see chapter Three), and LOADS it into the index register specified. This allows any of the index registers to be manipulated in a way that is more useful for an addressing register than simple arithmetic and logical operations. For example - LEAY 4,Y means add four to the contents of the Y register and store the result in Y. i.e increment the Y register by four. The LEA instruction is very powerful and deserves careful study. The final instruction is ABX, add the B register to the X register and place the result in the X register. As there is no equivalent instruction (i.e. ABY) for the Y register, this is the only instruction which treats the index registers differently from one another. Thus, with this one exception, the index registers may be taken to be identical to one another.

THE STACK POINTERS U, S

The stack pointers are both sixteen bit registers which can be used as index registers but they have two extra instructions PSH (PUSH) and PUL (PULL) and S is used implicitly by a number of other instructions. A stack pointer is a register that is used to hold the address of the top of an area of memory that can be used for temporary storage. The area of memory is known as the stack. Any register (or set of registers) can be stored in the stack by the use of a PSH instruction. This first decrements the stack pointer and then stores the specified register in the location that the stack pointer addresses, and so on until all of the registers have been stored. Registers can be loaded from the stack by a PUL instruction which loads the register specified from the location that the stack pointer addresses and then increments the stack pointer, and so on until all of the registers specified have been loaded. For example, PSH U A means decrement the U register and store the A register at the location whose address is stored in the U register. PUL U A means load the A register from the location that U addresses and then increment U. A set of registers can be specified as part of a PSH or PUL instruction and they are dealt with, stacked or unstacked in a specified order (see chapter three).

The two stack pointers are NOT identical. The U stack pointer is available for use by the programmer only. The S stack pointer is used by the 6809 for operations other than explicit PSH and PULs. The 6809 stores the contents of the PC register on the S stack when a subroutine jump is executed and loads the PC from the S stack when a return from subroutine is executed. The S stack is also used during hardware and software interrupts to store the entire set of registers (except for FIRQ which stores only PC and CC). Because of these additional uses the S stack pointer is known as the SYSTEM stack pointer and its associated stack is the SYSTEM stack. The U register is known as the USER register and associated with it is the USER stack.

The presence of two stack pointers in the 6809, one for the system and one for the user, makes the implementation of high-level languages very efficient for a machine of this size.

THE PROGRAM COUNTER PC

The program counter is used by the 6809 to point to the next instruction to be executed. The PC is more often referred to implicitly by being loaded during a branch or jump instruction. However it can be used in constant offset indexed and indirect constant offset indexed addressing (see chapter three), so it shares some of the properties of the general index registers.

THE CONDITION CODE REGISTER

The condition code register is a very special eight bit register used to store certain information about the state of the processor and results of compare instructions. Each bit has a specific meaning which we will describe in turn.

Bit Zero: The Carry Flag (C), is used to store the carry from additions. It is also used to store the borrow from subtract-like operations such as CMP, NEG, SUB, SBC. It is also used as a ninth bit in shift and rotate operations.

Bit One: The Overflow Flag (V), is set to a one by an operation which causes a signed two's complement arithmetic overflow.

Bit Two: The Zero Flag (Z), is set to one if the result of the previous operation was identically zero.

Bit Three: The Negative Flag (N), is the most significant bit (7 or 15) of the result of the previous operation. Thus a negative two's complement result will leave N set to one.

Bit Four: The interrupt mask bit (I), is used to determine how the processor reacts to hardware interrupts (IRQ). The processor will not recognise interrupts on the IRQ line if this bit is set to a one. NMI, FIRQ, IRQ, RESET and SWI all set I to one (after stacking the CC register) so as to disable IRQ type interrupts. Note that I is set after the CC register is stacked. SWI2 and SWI3 do not affect I.

Bit Five: The half carry bit (H), stores a carry from bit 3 as a result of an eight bit addition. This bit is used by the DDA (decimal adjust) instruction to perform BCD addition. The state of this flag is undefined following all subtract-like operations.

Bit Six: The FIRQ mask (F), is like the IRQ mask bit. It is used to disable or enable the FIRQ interrupt line. (The FIRQ interrupt line is simply a fast interrupt line that does not stack the entire machine state i.e. the complete set of registers. See chapter five on interrupts.) The processor will not recognise FIRQ interrupts if the FIRQ mask bit is set to one. NMI, FIRQ, SWI and RESET all set F to one. IRQ, SWI2 and SWI3 do not affect F. Note that the F bit is set after the CC register is stacked.

Bit Seven: The entire flag (E), if set to one indicates that the complete set of

registers was stacked as opposed to a subset (PC and CC). The E bit is used by the RTI (return from interrupt) instruction to determine the extent of the unstacking necessary. Therefore the current E value represents the previous state of the machine. Note that the E bit is set to one BEFORE the CC register is stacked during an interrupt (see chapter five).

The CC register can be exchanged or transferred to any of the other eight bit registers. Also four special instructions are provided to allow the condition codes to be manipulated separately. ANDCC and ORCC are restricted logical operations which AND or OR the CC register with the data byte following the instruction. CWAI (AND condition code register and wait for interrupt) is an extremely useful instruction and can result in time saving when servicing interrupts. The CWAI instruction first ANDs the CC register with the data byte following the instruction, then stacks the entire machine state and waits for an interrupt. When a (non-masked) interrupt occurs, no further machine states are saved thus reducing the time between the reception of an interrupt and its servicing.

Using the ANDCC and ORCC instructions, any bit or number of bits of the CC register can be set or reset. Some assemblers include extra instructions such as CLC (clear carry) to enable the programmer to set and reset bits without working out the data byte to be ANDed or ORed with the CC.

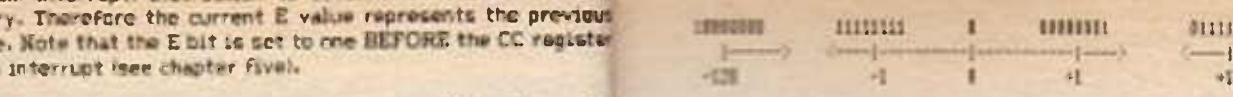
The rest of this chapter is a more detailed discussion of the condition code register and how it interacts with other 6809 features. The novice 6809 programmer is advised to skip these sections until after reading later chapters describing these features in more detail.

ARITHMETIC AND THE CONDITION CODES

The 6809's CC register includes a large number of flags concerned with indicating the state of an arithmetic operation - C, I, N and H. Some of these flags may be unfamiliar to a programmer new to the 6800/6809, for example the 8080 has only the C, N, I, H and a new flag P for parity. Thus the V flag in particular is likely to be unknown to most 8080 programmers. A more detailed explanation of the workings of the arithmetic flags seems in order.

All arithmetic operations are carried out to a limited precision. For the 6809 arithmetic is done (in one operation) to only eight or sixteen bits. To represent negative number we must choose a convention that makes arithmetic easy. The usual one, and the one used by the 6809, is TWO'S COMPLEMENT FORM. A number in two's complement form can either be positive or negative. To decide which, we examine the most significant bit - if it's zero then the number is positive, if it's one then the number is taken to be negative. If the number is positive the remaining bits represent its magnitude. For example 0111 is 7. If the number is negative then the remaining bits represent not the magnitude but the logical complement of the magnitude less one. For example 1111 is -1 not -7; the most significant bit is 1 so the number is negative and therefore the magnitude is obtained by taking the logical complement of 111 which is 000, and adding one to give 001. The largest positive number that can be represented by 8 bits is 01111111 or +127 and the largest magnitude negative number is 10000000 or -128. The reason for this odd coding of negative numbers is to make the order relationships between negative numbers

true in the coding. For example -1 > -2 and 1111 > 1110. The eight bit two's complement number system looks like this:



Obviously it is possible to carry out arithmetic on two numbers within this range and get a result outside this range. For example 124 + 4 = 128 or -128 -1 = -129. The 6809 V flag is used to detect this occurrence. An overflow has occurred if the carry from the most significant bit differs from the most significant bit - i.e. the sign bit of the result. The reason for this is that the carry is what the sign bit OUGHT to be if we were doing our arithmetic to one extra bit of precision - if they are the same then our result is correct if they are different we NEED to carry out our arithmetic to one extra place! Thus for ANY arithmetic operation where an overflow can occur V = N (+) C ((+) means exclusive OR). To emphasise the ANY consider the arithmetic shift left operation. Shifting left is the same as multiplying by two so an overflow condition can occur and this is indicated by V = N (+) C. Shifting right however is the same as division by two and so no overflow can occur and the V flag is unaffected. (Underflow can occur and this is spotted by testing for a zero result.)

THE CONDITION CODE REGISTER AND BRANCH INSTRUCTIONS

The N, Z and V flags are mostly used by the branch group of instructions to decide when a branch should or should not be taken. A branch instruction such as BGE - branch if greater than or equal to - implies that the two numbers can be compared by subtraction. This in turn implies that the numbers are represented either in absolute form (i.e. are positive) or in two's complement form (i.e. are positive or negative) and in either case the result of the compare can be represented by the precision available. If the result cannot be represented by the compare, then two options are open to us - first we can ignore any overflow that occurred during the compare and branch on the possibly invalid result - or we can use the information in the N and V flags to discover if the result WOULD have been positive or negative and thus branch correctly even if the result is invalid.

Some of the 6809 branches work with unsigned binary numbers and hence no overflow is possible from a compare. Some work with two's complement numbers and all of these use the V flag to determine the correct branch action on an invalid result. It is important that a 6809 programmer is aware of the simple fact that after arithmetic operations and compares the result may be invalid but a signed branch operation will still work correctly.

BCD ARITHMETIC AND THE HALF CARRY FLAG

Although it is faster and more efficient to use two's complement binary numbers for arithmetic it is sometimes not worth converting decimal numbers typed in from a VDU to binary if only a small amount of arithmetic is to be carried out. The answer is to use BCD (Binary Coded Decimal) arithmetic. A BCD digit is simply the binary representation of a decimal digit 0-9. For

example 0010 = 2, 1000 = 8, 1100 = 9. As only four bits are required two BCD digits can be stored in one byte and hence in one accumulator. An instruction to add two BCD digits to two other BCD digits would be an advantage. The 6809 however uses the H flag and the usual ADD instruction. If two pairs of valid BCD digits are added together using the binary add instruction the result is not necessarily a pair of valid BCD digits. This is because adding two BCD digits together can give an answer bigger than 9, the largest valid BCD digit. The DDA instruction examines the result of the ADD instruction and the state of the H flag to adjust the result to be valid BCD digits.

FURTHER READING In the final sections of this chapter I have tried to outline some of the important concepts of binary arithmetic for the 6809 programmer, but the reader who wants to learn more about this topic is advised to look elsewhere. A useful introduction is contained in "Beginners Guide to Microprocessors and Computing" by E. F. Scott (Babani, 1980). Alternatively, sunilar information is given in "Elements of Electronics, Book 4" by F. A. Wilson (Babani, 1980).

CHAPTER THREE
ADDRESSING

How efficient and easy to use a computer is, is determined by the overall architecture, the range of the instruction set and the ways in which memory locations can be specified as part of an instruction. The ways in which memory locations can be specified are usually referred to as the ADDRESSING MODES of a machine. For the 6809 a good understanding of the available addressing modes and how they can be used is the key to writing shorter, faster and more transportable programs. We will first examine some basic concepts of addressing.

WAYS OF ADDRESSING MEMORY

Broadly speaking, there are two basic ways a computer can address memory: ABSOLUTE and BASE RELATIVE.

Absolute addressing is the most obvious way of specifying where an operation should be carried out and involves stating the complete address of the operand. "LDA $4533" meaning load the A register from the location whose address IS $4533, is an example of absolute addressing, so is "LDA TEMP" where TEMP is defined (somewhere else in the program) to be an address. Absolute addressing has a number of disadvantages. It can use more memory than necessary to store the address, it can make the addressing of sequential memory locations difficult and it makes the relocation of programs (see below - position independence) a major task. Even though it has these problems it is still the most frequently used method of addressing.

Base relative addressing involves the use of a register, known as the base register, that is used to hold a full address, the base address and a number, referred to as the offset, which is the number of locations above or below the base address that the operand location is to be found.

Both absolute and base relative addressing offer many variations on a theme and it is often difficult to decide which method is being used by any given addressing mode. Base relative has however one important form, that merits mention, PC RELATIVE. PC relative is simply base relative using the program counter as the base register. Thus every address in PC relative is specified as the number of locations above or below the current location that the operand is. PC relative is often referred to simply as relative addressing and its importance is that it makes program relocation easy (see below - position independence).

The final concept required for an understanding of addressing modes is INDIRECTION. If we allow any addressing mode to specify not the memory location required but the address of a memory location that contains the address of the memory location required then we have INDIRECT addressing. Although it sounds complicated indirect addressing is fairly straightforward if we follow its logic step by step :

1) work out the location specified by the instruction in the usual way
2) use the contents of the location specified in step one as the address of the location that the instruction refers to.

Because indirection can be applied to any addressing mode it can even be applied to indirect addressing, leading to a second level of indirection. It is left to the reader to work out where the final address comes from in this case ! This can be repeated as many times as required giving any number of levels of indirection. In practice however, indirection is rarely used more than once and indeed most computers will not allow its use more than once and many popular micros do not allow ANY indirection. It may not be immediately obvious when indirection is useful but it is a powerful programming technique allowing parameters to be passed to subroutines and enabling the efficient use of look-up tables and is therefore a desirable extra to any computer's capabilities.

ADDRESSING MODES OF THE 6809

The 6809 has the most varied and extensive set of addressing modes of any of the current micros. In addition, as far as is possible, the addressing modes are "uniform" across the instruction set. That is, where it makes sense, any addressing mode can be used with any instruction. This obvious simplicity is NOT always true of other micros. For example, programming the 8080 usually requires all but the expert programmer to have a list of which addressing modes can be used with which instructions. This makes the assembly language more difficult to learn than it need be and the micro less powerful.

We will deal with each of the 6809's addressing modes in turn.

INHERENT

This is the simplest addressing mode. The address of the operand is contained in the instruction itself and cannot be modified in any way. For example, "MUL" means multiply accumulators A and B together and place the answer in the D register.

ACCUMULATOR

Accumulator addressing is similar to inherent except that one of a number of alternative registers (the accumulators) may be specified. For the 6809 either the A or B (and sometimes D) register may be specified in accumulator addressing. For example, "INC A" means increment the A register; "INC B" means increment the B register.

REGISTER

Register addressing is like accumulator addressing but any of the 6809's registers - A, B, D, X, Y, S, U, PC can be specified. For example, TFR X,Y means transfer the X register to the Y register.

DIRECT EXTENDED

In direct extended the sixteen bit number following the instruction is used as the address of the operand. For historical reasons the direct extended mode is often referred to simply as extended addressing. For example, "STA TEMP" means store the A register in the memory location whose address is TEMP (TEMP would be defined as a sixteen bit number somewhere else in the program).

DIRECT PAGED

In direct paged addressing the eight bit number following the instruction is taken to be the lower eight bits of the address of the operand. The upper eight bits used to make the full sixteen bit address are taken from a special register DP, the direct page register. For example, "STA $44" means store the A register in the memory location given by $XX44 where XX is the number stored in the direct page register. Notice that direct page mode is not specified explicitly but is implied by the size of the address (eight bits). When direct page mode is used depends on how clever the available 6809 assembler is. A good assembler will keep track of the value stored in the DP register and use direct paged mode whenever the address specified in a direct extended mode can be converted. This saves storage without the programmer having to worry too much. If it is necessary to explicitly force either direct paged or direct extended to, then the symbols < and > are often used. For example, "LDA <$5566" means load the A register from $XX66 where XX is the contents of the DP register and "LDA >$66 means load the A register from $0066.

IMMEDIATE

In immediate addressing the one or two bytes following the instruction are used as the operand. Immediate addressing is distinguished from direct by the symbol #. The number of bytes used depends on the instruction. For example, "LDA #$44" means load the A register WITH 44 and "LDX#$44" means load the X register WITH $0044. Notice that the number of bytes used depends on the size of the register specified in the instruction.

RELATIVE

In relative addressing the eight or sixteen bits following the instruction, the offset, are added to the current address to give the address of the operand. The offset is regarded as a two's complement number, so if the offset is eight bits, then memory within +127 and -128 bytes of the current location can be addressed. This is known as short relative addressing. If the offset is sixteen bits then memory within +32767 and -32768 bytes of the current location can be addressed. This is usually the entire memory for a standard 6809 system. This is known as long relative addressing.

For the 6809 relative addressing can only be used in branch instructions (but see below - PC relative). Long relative mode is indicated by an L in front of the standard branch instruction. For example, "BRA LOOP" would cause a branch to the memory location whose address was LOOP and if LOOP was within plus or minus 128 locations of the current address short relative addressing would be used. If LOOP was further away than this an error would be reported by most 6809 assemblers. "LBRA LOOP" would have the same effect except that long relative addressing would be used and no error could occur. Short relative addressing saves one byte over long relative addressing and should be used where possible.

CONSTANT INDEXED

In constant indexed addressing a two's complement constant up to sixteen bits long is added to the contents of any of the 6809's true sixteen bit registers (i.e. X, Y, U, S or PC but NOT D), and the result is used as the address of the operand. It should be noted that the indexing register is NOT ALTERED IN ANY WAY - the addition of the constant is a temporary one. For example, "STA 0,X" means store the A register in the address given by adding zero to the contents of the X register, "STA TEMP,S" means store the A register in the address given by adding the value of the symbol TEMP to the contents of the user stack pointer.

The number of bytes used to store the offset depends on its magnitude. The 6809 supports four distinct constant offset sizes each with a different size and speed overhead. A brief description is given below. For more information consult the later section on efficiency.

OFFSET	EXTRA STORAGE	EXTRA TIME	RANGE
zero	zero	zero	zero
five bit	zero	one cycle	-16 to +15
eight bit	one byte	one cycle	-128 to +127
sixteen bit	two bytes	four cycles	-32768 to +32767

ACCUMULATOR INDEXED

Accumulator indexed is like constant indexed except that the contents of the specified accumulator A, B or D (treated as a two's complement number) is added to the contents of one of the true sixteen bit registers (but not PC) to obtain the address of the operand. Note that D can be used to specify a sixteen bit variable offset! It should be noted that neither the accumulator nor the indexing register are altered by this addition which is temporary. For example, "STA D,X" means store the A register in the address given by adding the contents of the D register to the contents of the X register.

AUTO INCREMENT/DECREMENT ZERO OFFSET INDEXED

In this addressing mode any of the true sixteen bit (but not PC) registers can be used in a constant indexed mode but the constant MUST be zero. The register chosen may be followed by one or two plus signs or preceded by one or two minus signs. If plus signs are used the index register specified is INCREMENTED once for each plus sign AFTER the instruction is completed. That is the contents of the index register are used as the address of the operand and THEN incremented. If minus signs are used the index register is FIRST DECREMENTED by one for each minus sign. The contents of the register is then used as the address of the operand in the usual way.

The important points to remember about this mode is that only a ZERO constant offset is allowed, minus signs come BEFORE the name of the index register and decrementation is done FIRST and plus signs come AFTER and incrementation is done LAST. For example, "STA 0,X+" means store the accumulator in the location given by the contents of X and then increment X by one, or "STA 0,X++" would increment X by two, "STA 0,--X" means FIRST decrement X by two and then uses the contents of X as the address to store the contents of A in.

INDIRECT ADDRESSING

Some of the 6809 addressing modes described above, with some restrictions, can be used with one level of indirection. This is indicated by [ ] around the address that is to be used as the address of the required address. To work out what is going on in indirect addressing simply remember to work out the address within the square bracket as a normal address and then use that as the location of the two bytes containing the address of the operand. The addressing modes that support indirection are :

DIRECT EXTENDED
CONSTANT-OFFSET INDEXED
ACCUMULATOR INDEXED
AUTO-INCREMENT/DECREMENT ZERO OFFSET INDEXED

We will give an example of each one in turn :

INDIRECT EXTENDED

"STA [TEMP]" means store the A register in the location whose address is stored in TEMP. Although indirect extended addressing is logically an extension of direct extended it is in fact implemented by the 6809 as a special case of indirect indexed addressing.

CONSTANT OFFSET INDEXED INDIRECT

"STA [$FF,Y]" means store the A register in the location whose address is stored in the location whose address is the contents of the Y register plus $FF.

ACCUMULATOR INDEXED INDIRECT

"STA [A,X]" means store the A register in the location whose address is stored in the location whose address is the contents of the X register plus the contents of the A register.

AUTO INCREMENT/DECREMENT INDIRECT

"STA [0,S++]" means store the A register in the location whose address is stored in the location whose address is the contents of the S register and then increment the S register by two. NOTE that only an increment or decrement of two makes any sense in this type of indirection.

POSITION INDEPENDENT CODE

One of the advantages of using the 6809 is that it is very easy to write position independent programs. A position independent program is capable of being run at any memory location WITHOUT modification. This is obviously very important when a library of subroutines is being used or in time sharing. Good programming practice for the 6809 is to always write position independent code unless there is a good reason to the contrary.

For a program to be position independent it must deal with two types of problem. Addresses that don't move with the program, absolute addresses, must be referred to in a way that does not change as the program moves and addresses that do move with the program must be referred to in such a way that changes correctly as the program moves. For jumps within the program a relative branch instruction will always be position independent because the offset from the current position to the destination address always remains the same. The problem of referring to data bytes within a program in a position independent way, requires relative addressing to be available to types of instruction other than branches. Surprisingly, because the constant index mode can use the PC register, this problem is already solved.

PC RELATIVE

Although we have listed all of the addressing modes of the 6809, the use of the PC register as an index register deserves special mention. If we use the PC register in a constant index mode to refer to the address of a data value, the reference is obviously position independent because, as with relative addressing, the offset does not depend on where the program is loaded. Most 6809 assemblers therefore extend the available addressing modes by automatically converting an absolute address to an offset when used in PC relative addressing. PC relative addressing is indicated by calling the PC register PCR. For example, "STA TEMP,PC" is constant indexed addressing using the PC register. It simply stores the contents of the A register in the location addressed by the contents of the PC register added to the number that TEMP represents. However "STA TEMP,PCR" would store the contents of the A register in the location whose address WAS TEMP, because the assembler recognises the "PCR" and automatically replaces the value of TEMP used in constant indexed by the difference between the current value of the PC and the value of TEMP, which is the number of bytes TEMP is from the current location. Thus when the instruction is executed, the constant offset added to the PC gives the original value of TEMP. If the program is moved by ten bytes say, then the current value of the PC would be changed by ten, but so would the location of TEMP ie location within the program and thus the A register would STILL be stored in TEMP, hence giving position independent code.

In short, to produce position independent code, any address that moves when the program moves should be addressed using relative or PC relative addressing - immediate data is a special case of relative addressing - any address that does not move when the program does, such as an I/O port or interrupt vector, can be addressed by any other method EXCEPT relative or PC relative.

EFFICIENCY

To compare the efficiency of the various addressing modes it is necessary to list the execution times and memory requirements of a single instruction with each of the addressing modes. It is therefore impossible to compare all of the addressing modes with each other because no one instruction can be used with all the addressing modes. The most we can do is to compare the groups of addressing modes that can be used with a single instruction. We can gain slightly more information if we also consider other methods of achieving the same end that use other addressing modes. If we consider loading the A register from various locations we can compare the times for transferring one byte. In the table below the total number of bytes for each instruction and the total number of machine cycles (one cycle equals one micro second on a single speed 6809) are given; the numbers in square brackets give the same information for one level of indirection where possible.

MODE		Number of bytes Total		Number of cycles Total	
register	TBF B,A	2		6	
direct ext.	LDA TEMP	3	[4]	5	[7]
direct paged	LDA <TEMP	2		4	
immediate	LDA #$FF	2		2	
const.index 0 bits	LDA 0,X	2	[2]	2	[5]
const.index 5 bits	LDA 15,X	2	[3]	3	[6]
const.index 8 bits	LDA 127,X	3	[3]	3	[6]
const.index 16 bits	LDA 32767,X	4	[4]	6	[9]
accu. indexed 8 bits	LDA B,X	2	[2]	3	[6]
accu. indexed 16 bits	LDA D,X	2	[2]	6	[9]
auto inc/dec indexed	LDA B,X+	2		4	
auto inc/dec indexed	LDA B,X++	2	[2]	5	[8]
PC relative 8 bits	LDA $FF,PC	3	[3]	3	[6]
PC relative 16 bits	LDA $FFFF,PC	4	[4]	7	[10]

The main observations to be drawn from the table are that indirection is expensive on time and immediate addressing is fast. Other storage and timing information can be gained from chapter four.

CODING

For most of the addressing modes the coding is completely part of the OP CODE byte. However for indexed and register addressing the OP CODE is followed by a POST BYTE which contains extra information about the addressing mode. (This in fact makes the 6809 a variable length instruction machine.)

The details for the indexed addressing post byte are as follows:

post byte bit number 7 6 5 4 3 2 1 0								
0	R	R	(	offset	)			5 bit constant offset
1	R	R	0	0	0	0	0	auto increment by one
1	R	R	x	0	0	0	1	auto increment by two
1	R	R	0	0	0	1	0	auto decrement by one
1	R	R	x	0	0	1	1	auto decrement by two
1	R	R	x	0	1	0	0	constant zero offset
1	R	R	x	0	1	0	1	accumulator B offset
1	R	R	x	0	1	1	0	accumulator A offset
1	R	R	x	1	0	0	0	8 bit constant offset
1	R	R	x	1	0	0	1	16 bit constant offset
1	R	R	x	1	0	1	1	accumulator D offset
1	x	x	x	1	1	0	0	PC Relative with 8 bit offset
1	x	x	x	1	1	0	1	PC Relative with 16 bit offset
1	0	0	1	1	1	1	1	indirect extended addressing
	F1		F2		F3			

F1 = Index register specification field 00 = X
01 = Y
10 = U
11 = S

F2 = Indirection field when bit 7=1, F2=0 direct F2=1 indirect

F3 = Addressing mode field

x means that the bit can be a zero or a one

Notice that indirect extended addressing is implemented as a special case of indexed addressing.

An example of coding an instruction may help to make things clear! LDA [A,Y] is a two byte instruction, the first byte is the op code $A6 (obtained from chapter four), the second byte is the post byte which is 10100110, as the register is Y, RR=01 and as indirection is required X=1, we have 10110110 or $B6. Hence the complete two byte instruction code is $A6 B6.

Two types of register addressing post bytes are used. The first is used with the push/pull instructions to indicate which registers are to be stacked/unstacked.

The format is:

post byte bit 7	6	5	4	3	2	1	0
PC	U/S	Y	X	DP	B	A	CC

A one in any position causes the appropriate register to be included in the pull or push. If bit 6 is set then either U or S is stacked depending on which register is being used as the stack pointer (it is obvious that that stack pointer is not stacked!). The stacking order is from higher bit registers first during a push and lower bit registers during a pull.

For example, the coding of PSHS A,B,PC would be two bytes, the first being $34 (from chapter four) and the second 10000110 or $86 giving $34 86 for the complete instruction.

The second form is used by transfer and exchange instructions to specify a pair of registers to be moved. The top four bits of the post byte (4-7) define the source register and the bottom four (0-3) define the destination register.

These are coded as follows:

0000 - D	0101 - PC
0001 - X	1000 - A
0010 - Y	1001 - B
0011 - U	1010 - CC
0100 - S	1011 - DP

All other codes are invalid.

For example, TRF X,Y is a two byte instruction - the first byte is $1F (from chapter four) and the post byte is 00010010 or $12, giving $1F 12 for the complete instruction.

CHAPTER FOUR
THE INSTRUCTION SET

This chapter forms the bulk of the reference part of this book. It is a strictly alphabetic listing of the 6809's instruction set. It is not suggested that a beginner reads through from ABX to TST trying to take in every detail. To make the best use of this chapter it is probably enough to review each page briefly, noticing any special comments, and from then on simply look up any instruction as and when it causes difficulty. The only way to learn a new machine language is to write programs and make mistakes !

Each instruction is given along with its effect on the condition codes and a table of op codes. The table of op codes includes an indication of which addressing modes are allowed with each instruction, how much space it takes, and how many cycles it takes to complete. For a single speed 6809 (MC6809) the number of cycles can be translated directly into micro-seconds. The times given in column C of the table are the MINIMUM times that any instruction might take. ADDITIONAL CYCLES MIGHT HAVE TO BE ADDED FOR SOME INDEXED ADDRESSING MODES. The number of additional cycles can be found at the end of the chapter. In the case of branch instructions sometimes two timings are given. The first refers to the time if the branch is not taken and the second in brackets refers to the time if the branch is taken.

The same is also true of the memory requirements listed in column B in the tables in that extra bytes could be required for any given indexed addressing mode. This information for indexed addressing is again given at the end of the chapter.

Abbreviations used

Operators

(- = assignment or is transferred to
& = Boolean AND
+ = Boolean OR
(+) = Boolean EXCLUSIVE OR
: = concatenation
( ) = indirect contents of e.g. (X) is the content of the location whose address is in X.
H = high byte of e.g. XH = most significant byte of X
L = low byte of e.g. XL = least significant byte of X
' = after operation e.g. A' is the state of A after the operation
$ = Hexadecimal number e.g. $F = 15

Flags

H = Half carry flag
N = Negative flag
Z = Zero flag
V = Overflow flag
C = Carry flag
I = Interrupt flag
F = Fast interrupt flag
E = Entire state flag

Registers and Memory

A = Accumulator A
B = Accumulator B
D = Accumulator D (= A:B)
X = Index register X
Y = Index register Y
S = system stack pointer
U = user stack pointer
DP = direct page register
CC = condition code register
PC = program counter
SP = system stack pointer

R = a register (specified later)
M = a general memory reference
EA = effective address - the address specified by whatever addressing mode is in use.
IM = an immediate memory reference
# = an immediate memory reference

General

MS = most significant
LS = least significant
iff = if and only if

6809 Operation Codes

ABX Add the unsigned value in B into X

OPERATION: X' <- X + B

CONDITION CODES: Not affected

NOTES: This is the one instruction which makes the B register different from A.

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ABX	3A	3	1												

ADC Add with carry memory into accumulator

OPERATION: R' <- R + M + C

CONDITION CODES:
HC: Set iff the operation caused a carry from bit 3
NC: Set iff bit seven of the result is set
ZC: Set iff result is zero
VC: Set iff the operation caused an 8 bit 2's complement overflow
CC: Set iff the operation caused a carry from bit 7

NOTES: Notice that there is no 16 bit form of this instruction

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ADCA				99	4	2	B9	5	3	B9	2	2	A9	4+	2+
ADCB				D9	4	2	F9	5	3	C9	2	2	E9	4+	2+

ADD Add memory (without carry) into accumulator A,B or D

OPERATION: For an 8 bit operation R' <- R + M
For a 16 bit operation R' <- R + M:M+1

CONDITION CODES:
H: Affected only by an 8 bit operation - Set iff a carry from bit 3 occurred
N: Set iff MSB of result is set
Z: Set iff result is zero
V: Set iff there was an 8/16 bit 2's complement overflow
C: Set iff the operation caused a carry from the MSB.

NOTES: Notice that the H bit is only affected by an 8 bit addition hence BCD arithmetic can only be carried out two digits at a time.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ADDA				9B	4	2	BB	5	3	8B	2	2	AB	4+	2+
ADDB				DB	4	2	FB	5	3	CB	2	2	EB	4+	2+
ADDD				D3	6	2	F3	7	3	C3	4	3	E3	6+	2+

AND Logical AND of memory into accumulator A or B or CC

OPERATION: R' <- R & M

CONDITION CODES:

If R is not the condition code register

N: not affected
N: set iff bit 7 of result is set
Z: set iff result is zero
V: cleared
C: not affected

If R is condition code register

CC' <- CC & IM (immediate memory)

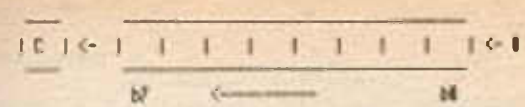
NOTES: ANDCC can only be used in immediate form.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ANDA				94	4	2	B4	5	3	84	2	2	A4	4+	2+
ANDB				D4	4	2	F4	5	3	C4	2	2	E4	4+	2+
ANDCC										1C	3	2			

ASL Arithmetic shift left

OPERATION:



C'<-b7,b7'<-b6,b6'<-b5.....b0'<-0

CONDITION CODES:
H: undefined
N: set iff bit 7 of the result is set
Z: set iff result is zero
V: loaded with b7 (+) b6 of the original operand
C: loaded with bit 7 of the original operand

NOTES: There is no 16 bit version of this instruction. For ASLD use:

ASLB
ROLA

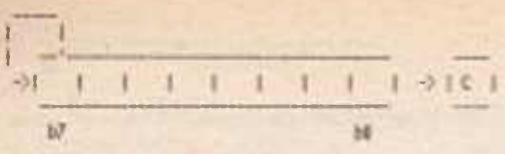
but notice that the flags are set for the byte in the A register only.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ASLA	48	2	1												
ASLB	58	2	1												
ASL				08	6	2	78	7	3				68	6+	2+

ASR Arithmetic shift right

OPERATION:



CONDITION CODES:
H: undefined
N: set iff bit 7 of the result is clear
Z: set iff result is zero
V: not affected
C: loaded with bit zero of original operand

NOTES: Notice that the V flag is not affected. the 6809/11/12/13/18 do affect the V flag.(See chapter 7) There is no 16 bit version of this instruction. For ASRD use:

ASRA
ASRB

but note that the flags are set only for the B register.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ASRA	47	2	1												
ASRB	57	2	1												
ASR				07	6	2	77	7	3				67	6+	2+

BCC Branch on carry clear

OPERATION: iff C=0 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: Used after a subtract or compare on unsigned binary values BCC branches if the register was higher or the same as the memory operand. This instruction is not useful after INC/DEC, LD/ST, TST/CLR/COM.

Relative addressing

Mode	OP	C	B
BCC	24	3	2
LBCC	1024	5(6)	4

BCS Branch on carry set

OPERATION: iff C=1 then PC' <- PC +IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare BCS causes a branch if the unsigned value in the register was lower than the unsigned memory value. This instruction is not useful after INC/DEC, LD/ST, TST/CLR/COM.

Relative addressing

Mode	OP	C	B
BCS	25	3	2
LBCS	1025	5(6)	4

BEQ Branch on equal to zero

OPERATION: iff Z=1 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare BEQ will cause a branch if the two values signed or unsigned were equal.

Relative addressing

Mode	OP	C	B
BEQ	27	3	2
LBEQ	1027	5(6)	4

BGE Branch on greater than or equal to zero

OPERATION: iff (N (+) V) = 0 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a compare or subtract operation BGE will cause a branch if in two's complement arithmetic the contents of the register were greater than or equal to the contents of the memory.

	Relative addressing		
Mode	OP	C	B
BGE	2C	3	2
LBGE	102C	5(6)	4

BGT Branch on greater than

OPERATION: iff Z & (N (+) V) = 0 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare instruction BGT causes a branch if the register was greater than the memory in two's complement arithmetic.

	Relative addressing		
Mode	OP	C	B
BGT	2E	3	2
LBGT	102E	5(6)	4

BHI Branch if higher

OPERATION: iff (C v Z) = 0 then PC <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare instruction BHI will cause a branch if the unsigned value of the register was higher than the memory value. It is not useful after INC/DEC, LD/ST, TST/CLR/COM.

	Relative addressing		
Mode	OP	C	B
BHI	22	3	2
LBHI	1022	5(6)	4

BHS Branch if higher or same

OPERATION: iff C = 0 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: Used after a subtract or compare on unsigned binary values BCC branches if the register was higher or the same as the memory operand. This is an assembly language duplicate of BCC and similarly is not useful after INC/DEC, LD/ST, TST/CLR/COM.

	Relative addressing		
Mode	OP	C	B
BHS	24	3	2
LBHS	1024	5(6)	4

BIT Bit test

OPERATION: R & M (no assignment)

CONDITION CODES:
N set iff bit 7 of result is set
Z set iff result is zero
V cleared
C not affected

NOTES: Only the condition codes are changed by this instruction

MEMORY ADDRESSING MODES

	IMMEDIATE			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
BITA				95	4	2	B5	5	3	85	2	2	A5	4+	2+
BITB				D5	4	2	F5	5	3	C5	2	2	E5	4+	2+

BLE Branch on less than or equal to

OPERATION: iff Z v (N (+) V) = 1 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare on two's complement values BLE causes a branch if the register was less than or equal to the memory operand

	Relative addressing		
Mode	OP	C	B
BLE	2F	3	2
LBLE	102F	5(6)	4

BLO Branch on lower

OPERATION: iff C=1 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare on unsigned binary values BLO causes a branch if the register was lower than the memory operand. This is a duplicate assembly language instruction for BCC and is similarly not useful after INC/DEC, LD/ST, TST/CLR/COM.

	Relative addressing		
Mode	OP	C	B
BLO	25	3	2
LBLO	1025	5(6)	4

BLS Branch on lower or same

OPERATION: iff (C v Z) = 1 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare on unsigned binary values BLS causes a branch if the register was lower or the same as the memory operand. Not useful after INC/DEC, LD/ST, TST/CLR/COM.

	Relative addressing		
Mode	OP	C	B
BLS	22	3	2
LBLS	1023	5(6)	4

BLT Branch on less than

OPERATION: iff (N (+) V) = 1 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare operation on two's complement values BLT will cause a branch if the register was less than the memory operand.

	Relative addressing		
Mode	OP	C	B
BLT	2D	3	2
LBLT	102D	5(6)	4

BMI Branch on minus

OPERATION: iff N = 1 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After an operation on a two's complement value BMI causes a branch if the result (possibly invalid) was negative. Notice that an overflow condition is not checked for.

	Relative addressing		
Mode	OP	C	B
BMI	2B	3	2
LBMI	102B	5(6)	4

BNE Branch not equal (to zero)

OPERATION: iff Z=0 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a subtract or compare operation on ANY binary values BNE causes a branch if the register is (or would be) not equal to the memory operand.

	Relative addressing		
Mode	OP	C	B
BNE	26	3	2
LBNE	1026	5(6)	4

BPL Branch on plus

OPERATION: iff N = 0 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a two's complement operation BPL causes a branch if the (possibly invalid) result is positive. Note that BPL does not check for overflow.

	Relative addressing		
Mode	OP	C	B
BPL	2A	3	2
LBPL	102A	5(6)	4

BRA Branch (always)

OPERATION: PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: Causes an unconditional branch.

	Relative addressing		
Mode	OP	C	B
BRA	20	3	2
LBRA	16	5	3

BRN Branch never

OPERATION: none

CONDITION CODES: Not affected

NOTES: BRN is essentially a NO-OP command but is useful in testing programs, timing loops etc.

	Relative addressing		
Mode	OP	C	B
BRN	21	3	2
LBRN	1021	5	4

BSR Branch to subroutine

OPERATION: SP' <- SP - 1, (SP) <- PCL
SP' <- SP - 1, (SP) <- PCH
PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: The PC is pushed onto the stack and a relative or long relative jump is executed.

	Relative addressing		
Mode	OP	C	B
BSR	8D	7	2
LBSR	17	9	3

BVC Branch on overflow clear

OPERATION: iff V = 0 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a two's complement binary operation BVC will cause a branch if there was no overflow.

	Relative addressing		
Mode	OP	C	B
BVC	28	3	2
LBVC	1028	5(6)	4

BVS Branch on overflow set

OPERATION: iff V = 1 then PC' <- PC + IM

CONDITION CODES: Not affected

NOTES: After a two's complement operation BVS will cause a branch if an overflow occurred.

	Relative addressing		
Mode	OP	C	B
BVS	29	3	2
LBVS	1029	5(6)	4

CLR Clear (set to zero)

OPERATION: M <- 0

CONDITION CODES:
N: not affected
Z: set
V: cleared
C: cleared

NOTES: The C flag is cleared for 6800 compatibility. Notice that the memory location is read during a clear.

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
CLRA	4F	2	1												
CLRB	5F	2	1												
CLR				0F	6	2	7F	7	3				6F	6+	2+

CMP Compare memory to register

OPERATION: For an 8 bit operation R - M (no assignment)
For a 16 bit operation R - M:M+1 (no assignment)

CONDITION CODES:
HI undefined
NI set iff MSB of result is set
ZI set iff result is zero
VI set iff the operation caused a 2's complement overflow
CI set iff the operation DID NOT cause a carry from the MSB

NOTES: Nothing is altered except the condition code register. Notice that the 16 bit form of this instruction is a true 16 bit compare unlike the 6800 family.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
CMPA		91 4 2	B1 5 3	81 2 2	A1 4+ 2+
CMPB		D1 4 2	F1 5 3	C1 2 2	E1 4+ 2+
CMPD		1093 7 3	10B3 8 4	1083 5 4	10A3 7+ 3+
CMPS		119C 7 3	11BC 8 4	118C 5 4	11AC 7+ 3+
CMPU		1193 7 3	11B3 8 4	1183 5 4	11A3 7+ 3+
CMPX		9C 6 2	BC 7 3	8C 4 3	AC 6+ 2+
CMPY		109C 7 3	10BC 8 4	108C 5 4	10AC 7+ 3+

COM Ones or logical complement

OPERATION:
M' <- M̄
CONDITION CODES:
HI not affected
NI set iff bit 7 of the result is set
ZI set iff result is zero
CI set

NOTES: The C flag is set for 6800 compatibility. There is no COMD instruction.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
COMA	43 2 1				
COMB	53 2 1				
COM		03 6 2	73 7 3		63 6+ 2+

CWAI Clear and wait for interrupt

OPERATION: CC <- CC & IM (possibly clear masks)
Set E (entire state saved)
SP' <- SP - 1, (SP) <- PCL
SP' <- SP - 1, (SP) <- PCH
SP' <- SP - 1, (SP) <- UL
SP' <- SP - 1, (SP) <- UH
SP' <- SP - 1, (SP) <- YL
SP' <- SP - 1, (SP) <- YH
SP' <- SP - 1, (SP) <- XL
SP' <- SP - 1, (SP) <- XH
SP' <- SP - 1, (SP) <- DP
SP' <- SP - 1, (SP) <- B
SP' <- SP - 1, (SP) <- A
SP' <- SP - 1, (SP) <- CC

wait for a non-masked interrupt to occur

CONDITION CODES:
possibly cleared by immediate byte

NOTES: After a CWAI an FIRQ interrupt may enter its interrupt handler with the entire machine state saved. The following RTI will return the entire machine state automatically however. This instruction replaces the 6809's CLI;WAI sequence.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
CWAI				3C 20 2	

DAA Decimal adjust A

OPERATION: A' <- A + CF1:CF2

where CF1 = 6 iff H=1
or LSN > 9

and = 0 Otherwise

CF2 = 6 iff C = 1
or MSN >9
or MSN >8 and LSN >9

and = 0 otherwise

MSN/LSN are the most/least significant 4 bits of the original operand

CONDITION CODES:
H: not affected
N: set iff MSB of result is set
Z: set iff result is zero
V: not defined
C: set if a carry from the MSB occurred or if the carry flag was set before the operation

NOTES: The DAA instruction is used after an ADDA or and ADCA to correct the result to BCD form. Before the addition operation both of the operands must be in a correct BCD format (i.e. each nibble must be between 0 and 9). Multiple precision operations can be carried out by using the carry generated by the DAA.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
DAA	19 2 1				

DEC Decrement

OPERATION: M' <- M - 1

CONDITION CODES:
H: not affected
N: set iff bit 7 is set
Z: set iff result is zero
V: set iff the original operand was $80
C: not affected

NOTES: The carry flag is not affected allowing the DEC instruction to be used as a loop counter in multiple precision operations. When operating on unsigned values only BEQ and BNE can be expected to work properly. When operating on two's complement values all signed branches work. This is because of the setting of the V flag for two's complement overflow and the C flag not being used to indicate a carry. Notice that BHI,BLS,BHS,BLO do not work consistently after a DEC. Notice that there is no 16 bit DECD instruction. Use LEAr -1,r for the index registers and stack pointers.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
DECA	4A 2 1				
DECB	5A 2 1				
DEC		0A 6 2	7A 7 3		6A 6+ 2+

EOR Exclusive OR of A or B with memory

OPERATION: R' <- R (+) M

CONDITION CODES:
H: not affected
N: set iff bit 7 of result is set
Z: set iff result is zero
V: cleared
C: not affected

NOTES: No 16 bit form of this instruction.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
EORA		98 4 2	B8 5 3	88 2 2	A8 4+ 2+
EORB		D8 4 2	F8 5 3	C8 2 2	E8 4+ 2+

EXG Exchange registers

OPERATION: R1 <--> R2 (note double assignment)

CONDITION CODES:
Not affected unless one of the registers is the CC'

NOTES: Only registers of the same size can be exchanged.

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
EXG															
R1,R2	1E	7	2												

INC Increment

OPERATION: M' <- M + 1

CONDITION CODES:
H: not affected
N: set iff bit 7 of the result is set
Z: set iff result is zero
C: not affected

NOTES: The carry flag is not affected allowing INC to be used as a loop counter in multiple precision arithmetic. Notice that there is no 16 bit form of this instruction, however LEAr 1,r can be used for the true 16 bit registers.

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
INCA	4C	2	1												
INCB	5C	2	1												
INC				0C	6	2	7C	7	3				6C	6+	2+

JMP Jump to the effective address

OPERATION: PC' <- EA

CONDITION CODES: Not affected

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
JMP				0E	3	2	7E	4	3				6E	3+	2+

JSR Jump to subroutine

OPERATION: SP' <- SP - 1 (SP) <- PCL
SP' <- SP - 1 (SP) <- PCH
PC' <- EA

CONDITION CODES: Not affected

NOTES: The return address is stored on the stack and a jump to the effective address occurs. If it is required to save all of the registers then use:

PSHS CC,A,B,DP,X,Y,U
JSR subroutine

and use PULS ALL as return.

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
JSR				9D	7	2	BD	8	3				AD	7+	2+

LD Load register from memory

OPERATION: For an 8 bit operation R' <- M
For a 16 bit operation R' <- M:M+1

CONDITION CODES:
H: not affected
N: set iff MS bit of loaded data is set
Z: set iff loaded data is zero
V: cleared
C: not affected

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
LDA				96	4	2	B6	5	3	86	2	2	A6	4+	2+
LDB				D6	4	2	F6	5	3	C6	2	2	E6	4+	2+
LDD				DC	5	2	FC	6	3	CC	3	3	EC	5+	2+
LDS				10DE	6	3	10FE	7	4	10CE	4	4	10EE	6+	3+
LDU				DE	5	2	FE	6	3	CE	3	3	EE	5+	2+
LDX				9E	5	2	BE	6	3	8E	3	3	AE	5+	2+
LDY				109E	6	3	10BE	7	4	108E	4	4	10AE	6+	3+

LEA — Load effective address

OPERATION: R' <- EA

CONDITION CODES:
H: not affected
N: not affected
Z: LEAX,LEAY: set iff result is zero
   LEAS,LEAU: not affected
V: not affected
C: not affected

NOTES: The LEA instruction is special in that it allows access to the address of a location rather than the location. The effective address is computed using whatever addressing mode is specified and the result is loaded into the register specified. Notice that instructions like LEAY 1,Y are allowed. The LEAX and LEAY instructions affect the Z bit to allow 6800 compatibility with INX/DEX replaced by instructions like LEAX 1,X. LEAU and LEAS do not affect any of the condition codes to allow the stacks to be cleaned up with the condition codes returned as results to a calling routine and for 6800 INS/DES compatibility.

It is important to be aware of a pitfall in the use of the LEA instruction. LEAr ,r++ and LEAr ,r+ do not change the register r because the effective address is loaded before the auto incrementation is done on an internal register. Instead, use LEAr 2,r and LEAr 1,r which are faster in any case.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
LEAS													32	4+	2+
LEAU													33	4+	2+
LEAX													30	4+	2+
LEAY													31	4+	2+

LSL — Logical shift left

OPERATION:

C'<-b7,b7'<-b6,b6'<-b5.....b1'<-0

CONDITION CODES:
H: undefined
N: set iff bit 7 of the result is set
Z: set iff result is zero
V: loaded with b7 (+) b6 of the original operand
C: loaded with bit 7 of the original operand

NOTES: This is a convenient assembly language duplicate of ASL which performs the same function. See ASL for more details.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ASLA	48	2	1												
ASLB	58	2	1												
ASL				08	6	2	78	7	3				68	6+	2+

LSR Logical shift right

OPERATION:

CONDITION CODES:
H: not affected
N: cleared
Z: set iff result is zero
C: loaded with bit 0 of the original operand

NOTES: The &888 affects the V flag. There is no 16 bit version of this instruction. To shift D (LSRD) use:

LSRA
RORB

but notice that flags only reflect the condition of the byte in the B register.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
LSRA	44	2	1												
LSRB	54	2	1												
LSR				04	6	2	74	7	3				64	6+	2+

MUL Multiply accumulators

OPERATION: A':B' <- A * B

CONDITION CODES:
H: not affected
N: not affected
Z: set iff the result is zero
V: not affected
C: set iff bit 7 of B' is set

NOTES: The multiply operation is on unsigned binary numbers. The carry flag allows the result to be rounded back to 8 bits by:

MUL
ADCA #0

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
MUL	3D	11	1												

NEG Arithmetic or two's complement (negate)

OPERATION: M' <- 0 - M

CONDITION CODES:
H: undefined
N: set iff bit 7 of result is set
Z: set iff result is zero
V: set iff original operand was $80
C: set iff the operation did not cause a carry from bit 7.

NOTES: This is a two's complement negate.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
NEGA	40	2	1												
NEGB	50	2	1												
NEG				00	6	2	70	7	3				60	6+	2+

NOP No operation

OPERATION: none

CONDITION CODES: Not affected

NOTES: No registers (except for the PC) or memory contents are affected.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
NOP	12	2	1												

OR Inclusive OR memory into A, B or CC

OPERATION: R' <- R v M
R' <- R v IM if R is CC

CONDITION CODES:
if R is not CC
H: not affected
N: set iff bit 7 of result is set
Z: set iff result is zero
C: not affected

if R is CC then

CC' <- CC v IM

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ORA				9A	4	2	BA	5	3	8A	2	2	AA	4+	2+
ORB				DA	4	2	FA	5	3	CA	2	2	EA	4+	2+
ORCC										1A	3	2			

PSH= Push registers onto = stack

OPERATION: = = S or U
if ==S then #=U
if ==U then #=S

if PC is to be pushed then =' <- = - 1, (=) <- PCL
=' <- = - 1, (=) <- PCH
if D is to be pushed then =' <- = - 1, (=) <- DL
=' <- = - 1, (=) <- DH
if Y is to be pushed then =' <- = - 1, (=) <- YL
=' <- = - 1, (=) <- YH
if X is to be pushed then =' <- = - 1, (=) <- XL
=' <- = - 1, (=) <- XH
if DP is to be pushed then =' <- = - 1, (=) <- DP
if B is to be pushed then =' <- = - 1, (=) <- B
if A is to be pushed then =' <- = - 1, (=) <- A
if CC is to be pushed then =' <- = - 1, (=) <- CC

Notice that only the stack pointer not being used can be stacked !

CONDITION CODES: Not affected

NOTES: Any, all or none of the registers may be pushed onto the stack. For any register not included in the push the appropriate cycle is ignored. The order the registers are pushed is strictly as above and does not depend on the way the assembly language instruction is written e.g. PSHU CC,Y,PC is the same as PSHU PC,CC,Y and the order is first PC, second Y, third CC.

A single register can be stored on the stack with the condition codes set by using an auto increment store, e.g. STA ,-S; STX ,--S.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
PSHU	36	5+	2												
PSHS	34	5+	2												

A push requires 5 cycles + one cycle for every BYTE pushed.
e.g. PSHS A requires 6 cycles, PSHS X,A requires 7 cycles.

PUL_* Pull registers from a stack

OPERATION: x = S or U
if x=S then #=U
if x=U then #=S

if CC is to be pulled then CC <- (x), x' <- x + 1
if A is to be pulled then A <- (x), x' <- x + 1
if B is to be pulled then B <- (x), x' <- x + 1
if DP is to be pulled then DP <- (x), x' <- x + 1
if X is to be pulled then XH <- (x), x' <- x + 1
XL <- (x), x' <- x + 1
if Y is to be pulled then YH <- (x), x' <- x + 1
YL <- (x), x' <- x + 1
if U is to be pulled then UH <- (x), x' <- x + 1
UL <- (x), x' <- x + 1
if PC is to be pulled then PCH<- (x), x' <- x + 1
PCL<- (x), x' <- x + 1

Notice that only the stack pointer not being used can be pulled from the stack !

CONDITION CODES: Not affected

NOTES: Any, all or none of the registers may be pulled from the stack. For any register not included in the pull the appropriate cycle is ignored. The order the registers are pulled is strictly as above and does not depend on the way the assembly language instruction is written e.g. PULU CC,Y,PC is the same as PULU PC,CC,Y and the order is first CC, second Y, third PC.

A single register can be pulled from the stack with the condition codes set by an auto increment load, e.g. LDA ,S+; LDX ,S++.

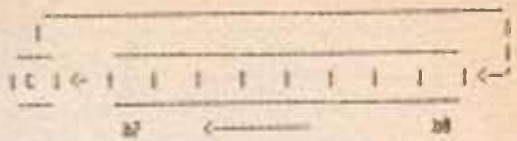
MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
PULS	35	5+	1												
PULU	37	5+	1												

A pull requires 5 cycles + one cycle for every byte pulled.
e.g. PULS A requires 6 cycles, PULS X,A requires 7 cycles.

ROCR_ Rotate left

OPERATION:



C'<-b7,b7'<-b6,b6'<-b5.....,b0'<-C

CONDITION CODES:
HC not affected
NC set iff bit 7 of result is set
ZC set iff result is zero
VC loaded with b7 (+) b6 of the original operand
CC loaded with b7 of original operand

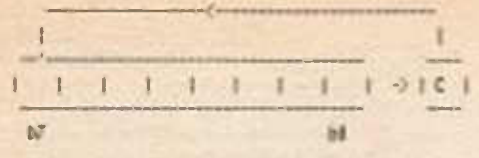
NOTES: A 16 bit version of this operation is not available.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
ROLA	49	¢Z	1												
ROLB	59	Z	1												
ROL				09	6	Z	79	7	3				69	6+	2+

ROR Rotate right

OPERATION:



b7'<-C,b6'<-b7,b5'<-b6,b4'<-b5....b0'<-b1,C'<-b0

CONDITION CODES:
H: not affected
N: set iff bit 7 of result set
Z: set iff result zero
V: not affected
C: loaded with bit zero of operand

NOTES: The 6800 sets the V bit.

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
RORA	46	2	1												
RORB	56	2	1												
ROR				06	6	2	76	7	3				66	6+	2+

RTI Return from interrupt

OPERATION: CC' <- (SP) , SP' <- SP + 1

Then if E flag is set

A <- (SP), SP' <- SP + 1
B <- (SP), SP' <- SP + 1
DP <- (SP), SP' <- SP + 1
XH <- (SP), SP' <- SP + 1
XL <- (SP), SP' <- SP + 1
YH <- (SP), SP' <- SP + 1
YL <- (SP), SP' <- SP + 1
UH <- (SP), SP' <- SP + 1
UL <- (SP), SP' <- SP + 1
PCH <- (SP), SP' <- SP + 1
PCL <- (SP), SP' <- SP + 1

if CC bit E is clear then

PCH <- (SP), SP' <- SP + 1
PCL <- (SP), SP' <- SP + 1

CONDITION CODES:
recovered from stack

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
RTI entire	3B	15	1												
RTI partial	3B	6	1												

RTS Return from subroutine

OPERATION: PCH' <- (SP), SP' <- SP + 1
PCL' <- (SP), SP' <- SP + 1

CONDITION CODES: Not affected

MEMORY ADDRESSING MODES

	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
RTS	39	5	1												

SBBC Subtract with borrow

OPERATION: R' <- R - M - C

CONDITION CODES:
H: undefined
N: set iff bit 7 of the result is set
Z: set iff result is zero
V: set iff the operation causes a two's complement overflow
C: set iff the operation did not cause a carry from bit 7

NOTES: There is no 16 bit version of this instruction.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
SBCA		92 4 2	B2 5 3	82 2 2	A2 4+ 2+
SBCB		D2 4 2	F2 5 3	C2 2 2	E2 4+ 2+

SEX Sign extend lower eight bits in B register

OPERATION: If bit 7 of B is set then A' <- $FF
else A' <- $00

CONDITION CODES:
H: not affected
N: set iff MSB of result is set
Z: set iff result is zero
V: not affected
C: not affected

NOTES: This instruction converts a two's complement number in B into a 16 bit two's complement number in D.

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
SEX	1D 2 1				

ST Store register into memory

OPERATION: For an 8 bit operation M' <- R
For a 16 bit operation M':M+1' <- R

CONDITION CODES:
H: not affected
N: set iff MS bit of stored data was set
Z: set iff stored data was zero
V: cleared
C: not affected

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
STA		97 4 2	B7 5 3		A7 4+ 2+
STB		D7 4 2	F7 5 3		E7 4+ 2+
STD		DD 5 2	FD 6 3		ED 5+ 2+
STS		109F 6 3	10FF 7 4		10AF 6+ 3+
STU		DF 5 2	FF 6 3		EF 5+ 2+
STX		9F 5 2	BF 6 3		AF 5+ 2+
STY		109F 6 3	10BF 7 4		10AF 6+ 3+

SUBB Subtract memory from register

OPERATION: R' <- R - M

CONDITION CODES:
H: undefined
N: set iff MS bit of result is set
Z: set iff result is zero
V: set iff the operation caused a two's complement overflow
C: set iff the operation did not cause a carry from the MS bit

MEMORY ADDRESSING MODES

	INHERENT	DIRECT	EXTENDED	IMMEDIATE	INDEXED
Modes	OP C B	OP C B	OP C B	OP C B	OP C B
SUBA		90 4 2	B0 5 3	80 2 2	A0 4+ 2+
SUBB		D0 4 2	F0 5 3	C0 2 2	E0 4+ 2+
SUBD		93 6 2	B3 7 3	83 4 3	A3 6+ 2+

SWI 2/3 Software interrupt

OPERATION: Set E flag

```
SP' <- SP - 1, (SP) <- PCL
SP' <- SP - 1, (SP) <- PCH
SP' <- SP - 1, (SP) <- UL
SP' <- SP - 1, (SP) <- UH
SP' <- SP - 1, (SP) <- YL
SP' <- SP - 1, (SP) <- YH
SP' <- SP - 1, (SP) <- XL
SP' <- SP - 1, (SP) <- XH
SP' <- SP - 1, (SP) <- DP
SP' <- SP - 1, (SP) <- B
SP' <- SP - 1, (SP) <- A
SP' <- SP - 1, (SP) <- CC

iff SWI then
 set I, F (mask interrupts)
 PC' <- (FFFA):(FFFB)
iff SWI2 then
 PC' <- (FFF4):(FFF5)
iff SWI3 then
 PC' <- (FFF2):(FFF3)
```

CONDITION CODES: Not affected

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
SWI	3F	19	1												
SWI2	103F	20	2												
SWI3	113F	20	2												

SYNC Synchronize to external event

OPERATION: All processing stops until an interrupt occurs
If the interrupt is enabled and lasts longer than 3 cycles
the appropriate interrupt routine is executed.
If the interrupt is masked or shorter than 3 cycles execution
continues at the next instruction (without stacking the registers).

CONDITION CODES: Not affected

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
SYNC	13	>2	1												

TRF Transfer register to register

OPERATION: R2 <- R1

CONDITION CODES:
not affected unless R2 = CC

NOTES: Only registers of the same size may be transferred. The D register can be used as a link between the 8 and 16 bit registers. For example TFR DP,X can be implemented as TFR DP,B;TRF B,X

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
TFR R1,R2	1F	?	2												

TST Test

OPERATION: M - 0 no assignment

CONDITION CODES:
HC not affected
NC set iff bit 7 of result is set
ZC set iff result is zero
VC cleared
CC not affected

NOTES: The 6800 also clears the C flag. The TST instruction is of less use on unsigned values than signed because unsigned values cannot be less than zero. Because the C flag is unaffected the only unsigned branches that will work after TST are BEQ and BNE. All the signed branches are available.

	MEMORY ADDRESSING MODES											
	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED		
Modes	OP	C	B	OP	C	B	OP	C	B	OP	C	B	OP	C	B
TSTA	4D	2	1												
TSTB	5D	2	1												
TST				0D	6	2	7D	7	3				6D	6+	2+

Indexed addressing Modes extra time and memory

Type	Form	non-indirect C	non-indirect B	indirect C	indirect B
Constant offset	Zero offset	0	0	3	0
	5 bit offset	1	0	4	1#
	8 bit offset	1	1	4	1
	16 bit offset	4	2	7	2
Accumulator offset	A/B register	1	0	4	0
	D register	4	0	7	0
Auto inc/dec	inc/dec by 1	2	0	not allowed	
	inc/dec by 2	3	0	6	0
const. offset from PC	8 bit offset	1	1	4	1
	16 bit offset	5	2	8	2
extended indirect	16 bit address			5	2

Stacking order

High memory

	PCL		
10,5	PCH		
	U/SL		
8,5	U/SH		
	YL		
6,5	YH		
	XL		
4,5	XH		
3,5	DP		
2,5	B		
1,5	A		^
0,5	CC	<- SP or U	Pull |
			push |
			V

Branch Groups

Simple Conditional Branches

BEQ	Z=1	BNE
BMI	N=1	BPL
BCS	C=1	BCC
BVS	V=1	BVC

Signed Conditional Branches

BGT	$\overline{N \oplus V} \cdot \overline{Z} = 1$	BLE
BGE	$\overline{N \oplus V} = 1$	BLT
BEQ	Z=1	BNE
BLE	$N \oplus V = 1$	
BLT	$N \oplus V = 1$	BGE

Unsigned conditional branches *

BHI	$\overline{C} \cdot \overline{Z} = 1$	BLS
BHS	$\overline{C} = 1$	BLO
BEQ	Z = 1	BNE
BLS	$C + Z = 1$	BHI
BLO	C = 1	BHS

* not useful after INC/DEC, LD/ST, TST/CLR/COM

CHAPTER FIVE
INTERRUPT HANDLING

The way in which a microprocessor handles interrupts is one of the many details which determines whether or not a processor is suitable for a particular application. For the microprocessor designer there are a number of choices to be made concerning interrupts. In the early days micros could only have one, or at most two, sorts of interrupt because of the lack of space on the chip and the shortage of spare pins on a standard integrated circuit package. Ideally an interrupt should be serviced by first stacking the entire machine state and then jumping to an appropriate interrupt handling program. However the requirement for an interrupt to be serviced quickly would make the facility to stack only a subset of the registers very desirable. Also the limitations on chip design and speed make the use of a fixed interrupt address the usual way of servicing the interrupt. A fast simple interrupt technique generally makes a processor suitable for control aplications. A sophisticated interrupt technique generally makes a processor suitable for time sharing and multi-tasking.

The 6809 has a number of interrupt handling techniques that make it suitable for dedicated control operations where speed is important and a number that make it suitable for large operating systems. We will deal with each method in turn but first we will give a brief description of what an interrupt is.

INTERRUPTS

The invention of the interrupt is one of the few new ideas to be introduced to computing hardware since Babage! If a computer is executing a program and an external event requires that it does something else at ONCE then an interrupt is the simplest method of achieving this. Usually a number of connections, interrupt lines, are provided to the computer and the event which requires immediate attention is allowed to pulse one of the lines when it requires that attention. On receiving a pulse on an interrupt line the computer should stop what it is doing, transfer its attention to the action required by the external event, and, at the completion of the action, transfer its attention back to its original task. This is all there is to an interrupt ! an interrupt signal (or cause)! a saving of the current machine state! a jump to the interrupt servicing routine! and finally a restoration of the original machine state.

Of course real interrupts have a number of problems to overcome and the programmer should ask the following questions of any interrupt method :

1) On receiving an interrupt signal how long before action is taken ? Is the current instruction finished first ?
2) How much, if any, of the machine's state is stored during an interrupt?

3) Where does the machine jump to on receipt of an interrupt, to a fixed address or a variable one ?
4) How many sources of interrupts are there ?
5) How is the source of an interrupt identified ?
6) What happens if an interrupt occurs during an interrupt ? Can an interrupt be interrupted?
7) Can interrupts be disabled ?

6809 INTERRUPTS

The 6809 has three interrupt lines: NMI - Non Maskable Interrupt, IRQ - Interrupt Request and FIRQ - Fast Interrupt Request. In addition it has three software interrupt commands! SWI, SWI2 and SWI3. A software interrupt is not really an interrupt in the conventional sense of a response to an external event but is a very useful version of the same mechanism. A software interrupt is an instruction which causes a saving of the machine status and a jump to a fixed location. In some senses it is more like an extended jump to subroutine instruction but with a fixed destination as in an interrupt. An important feature of interrupts on the 6809 is that the interrupt addresses are indirect. That is an interrupt obtains its destination address from a fixed pair of bytes rather than jumping to a fixed location.

There are also a number of instructions concerned with interrupts and interrupt processing. These are RTI, SYNC and CWAIT.

NON MASKABLE INTERRUPT (NMI)

On receipt of a pulse on the NMI line the processor sets the E bit in the CC register and then stacks the entire register set in the usual order and jumps to the location whose address is contained in $FFFD -$FFFC. The only changes that are made to the machine state following the stacking are the setting of the I, F and E bits in the CC register. This interrupt cannot be masked (disabled) and therefore is always honoured by the processor. Also it has the highest priority and will always be dealt with first. Following a RESET an NMI will be ignored until the first loading of the system stack pointer S.

FAST INTERRUPT REQUEST (FIRQ)

FIRQ has a lower priority than NMI but a higher priority than other interrupts. A low level on the FIRQ input line causes the E bit to be cleared, then the PC and CC registers to be stacked and a jump to the location whose address is stored in $FFF6-FFF7. The FIRQ is fast in the sense that it only stacks the PC and CC registers. Bits I and F are the only CC register bits set. Note that the F bit disables any further FIRQ interrupts unless it is explicitly cleared by the interrupt service routine.

INTERRUPT REQUEST (IRQ)

A low input on the IRQ line causes the processor to set the E bit in the CC register and to stack the entire set of registers on the system stack and a jump to the location whose address is stored at $FFF8-$FFF9. The IRQ sequence can be disabled by setting the I bit in the CC register. It also has a lower priority than FIRQ and NMI. IRQ sets the I and E bits in the CC register. Notice that FIRQ is not disabled following an IRQ service. If it is desired to mask FIRQ then the F bit should be set by an ORCC instruction. Since IRQ stacks the entire machine state its response is slower than FIRQ.

SOFTWARE INTERRUPTS

Three software interrupts SWI, SWI2 and SWI3 are provided in the 6809. Their order of priority is SWI, SWI2, SWI3 and their main difference is that SWI disables further interrupts via IRQ and FIRQ. When any SWI instruction is encountered the entire set of registers is stacked. (The E bit is set before the CC register is stacked.) The addresses that are used by the SWI instructions are given in table one. Software interrupts are not usually of use in normal programming but help no end with system design, making single step hardware debugging, trace and memory mapping easy.

INTERRUPT RELATED INSTRUCTIONS

RTI This is the usual way of terminating an interrupt routine. An RTI removes from the system stack whatever was placed there by the last interrupt. As the first byte pulled from the stack is always the condition code register, the E bit can be tested to see if any other registers have to be pulled before the PC register. Apart from its regular use the RTI instruction is often used as a way of jumping to a program with the machine state set to a particular condition.

SYNC The sync instruction is a very special and new instruction to the 68XX family. When a SYNC instruction is executed processing halts and the processor waits for an interrupt. When any interrupt occurs processing continues. If the interrupt was enabled and if it lasted for more than three machine cycles then the normal interrupt sequence occurs. If the interrupt was not enabled or lasted for less than three machine cycles then the processor goes on to execute the next instruction as if nothing had happened. This remarkable instruction is obviously of great value in syncronising processor operation to the outside world but with the option of taking some emergency action if need be.

CWAIT This instruction has been dealt with in chapter two in connection with the CC register, but it is worth pointing out its importance. A CWAIT causes the condition code register to be anded with the following immediate byte and then stacks the entire machine state

and waits for a non-masked interrupt to occur. It is important to note that this can save valuable time in the servicing of an interrupt but cannot increase the number of interrupts per second that can be handled. Indeed because the FIRQ interrupt can be forced to stack and unstack the entire machine state, it can reduce the number of interrupts per second that can be handled.

NOTES ON INTERRUPT HARDWARE

Although this book is not about 6809 hardware a few details of how interrupts interact with the hardware will help the software expert appreciate the types of action that are possible.

The 6809 has two signal lines that are relevant to interrupt operation the BA (Bus Available) line and the BS (Bus State) line. The state of these two lines reflects the state of the processor as given below:

BA	BS	Processor State
0	0	Normal (running)
0	1	IACK (interrupt acknowledge)
1	1	HALT (not running)
1	0	SYNC (SYNC acknowledge)

It is not difficult to see that an interrupting device could examine the state of the BA and BS lines to discover if its interrupt request had been accepted. This opens the possibility of the device taking control of the data bus at this point and supplying the address to be used as the interrupt jump address. (Notice that this is possible not only because of the IACK signal but because the 6809 interrupts are VECTORED, that is the processor does not jump to a fixed location but obtains its jump address from a fixed location.) This frees the programmer from the task of finding the source of the interrupt and speeds interrupt servicing time. This technique is known as vectored interrupts. The other point worth noticing is the fact that the processor is in a SYNC state can be detected by external equipment, effectively saying that the processor is waiting for something to happen. This indication is not given however for a CWAIT instruction!

GENERAL CONSIDERATIONS ON INTERRUPT PROGRAMMING

Writing programs that handle interrupts is one of the most tricky areas of programming. Hardware experts often think of a system based on interrupts before anything else because from a hardware point of view it forms an easy solution. Two facts should be borne in mind by anyone considering using interrupts for any reason. First, the time spent in saving machine states and jumping about may be longer than the time spent in processing the result of an interrupt. Second, interaction effects are often difficult to predict in systems capable of multiple interrupt sources; these effects are usually due to inadequate software design but this should not be suprising for to design good interrupt handlers

requires a fair degree of hardware knowledge. The point is that it is often better to construct a program that polls devices in a fixed sequence than allow the demand servicing of interrupts.

The warnings having been given, it must be admitted that there are things that simply cannot be done without interrupts and the 6809 has a wide range of interrupt options that suit most problems. It is clear that the NMI interrupt should be used for real emergencies e.g. power failure, and the FIRQ interrupt should be used for fast servicing of devices. The IRQ interrupt is a general purpose interrupt for slow to medium external devices - it is easier on the programmer than FIRQ but a lot more sluggish. The CWAI instruction should be used if the processor can't find anything else to do between interrupts or if a fast response to an IRQ/NMI is required. It doesn't really improve an FIRQ because of the extra time spent in unstacking. The possibilities of the SYNC instruction have to be explored by inventive systems designers but its dual function of "wait" or "do interrupt" should be kept in mind.

Table One

Interrupt	Location of jump address	timings
SWI	$FFFA,$FFFB	20
SWI2	$FFF4,$FFF5	20
SWI3	$FFF2,$FFF3	20
IRQ	$FFF8,$FFF9	20
NMI	$FFFC,$FFFD	20
FIRQ	$FFF6,$FFF7	11

CHAPTER SIX
PROGRAMMING STYLE

The 6809 is probably the first popular microprocessor that warrants a consideration of programming style. For other microprocessors a programming guide would consist of a list of tricks (both dirty and clean), but the 6809 has a sufficiently "interesting" architecture for a more advanced guide to be useful. That is not to say that the 6809 does not have as many clever tricks as any other micro but I hope that at the end of this chapter the reader will see why they are no part of "good" programming style.

EFFICIENCY OR ELEGANCE

In most cases a program can either be efficient (fast, small) or elegant (easy to understand, easy to modify, easy to correct). The reason for this is that, given a program that is elegant, it is usually possible to introduce some programming trick or method that speeds the program up or makes it smaller. However the use of such tricks often makes a program "messy" and much more difficult to understand. In the early days of computing, programming was mainly about writing efficent programs. Later, as computer time became cheaper and programmer time more expensive the emphasis moved to how quickly a reliable program could be written. This pattern was repeated in the early days of microcomputing. Not so much because of the cost of microcomputer time, but because the early micros were very slow and were being used mostly for real time control applications. It is still true that for some applications quick tricks are required to enable a processor such as the 6809 to keep up with the real world. However, there is also a tendency to use this as an excuse for sloppy programming. For most applications speed is not the most important element of implementation and a "good" progam is the most elegant. A slogan that all programmers should take to heart is "speed is a hardware problem - programming is about quality".

STRUCTURE

The question of elegant programming is traditionally dealt with by demanding that a design method is employed. The best known and most used design method is Top Down Structured Programming (TDSP) which is explained in many software textbooks. It is usually assumed that assembly language programming doesn't require a design method and if it did it wouldn't be TDSP which is most concerned with high level languages such as Pascal. This simply isn't true. The design processes involved in TDSP can and should be applied in any language. However there are special considerations to be kept in mind when using a low level language, that relate to the architecture of the particular machine being used. These special considerations will be the subject of the rest of this chapter.

ELEGANT PROGRAMMING

Elegant programming at the assembly language level is especially about how the machine's architecture can be used to simplify program construction. Apart from this rather general consideration we may also ask that a program at the assembly level is position independent, modular or re-entrant. We will deal with these specific requirements first.

POSITION INDEPENDENCE

Position Independent Code (PIC) has already been discussed in chapter three. A position independent program can be loaded into any area of memory and will run correctly without any modification. This is an important property if the program has to be made available on a range of hardware where the user memory is likely to be in different places. It also makes the distribution of software in EPROM or ROM easier. Instead of distributing a program EPROM/ROM for every starting address possible, one is enough if it contains PIC. Another advantage of PIC becomes evident when a library of programs or subroutines is being established for, if each program required a particular area of memory to function, it's very clear that things become very complicated.

To repeat the information from chapter three about producing position independent code :-

any address that moves when the program moves should be addressed using relative or PC relative addressing (immediate addressing is a special case of relative).
any address that doesn't move with the program such as an I/O port can be addressed by any other method EXCEPT relative or PC relative.

MODULAR PROGRAMMING

Modular programming is really a part of the TDSP method but requires some extra thought at the assembly language level. A program is modular if it can be split up into a number of other programs (modules), each one carrying out a particular job and interacting with the others only at its start and end. Modules are usually subroutines but the reverse need not be true. The advantages of modular programming are obvious - a number of programmers can each produce modules to construct a larger program, program errors can be isolated to individual modules, subroutine/module libraries can be set up. The problem with modules at the assembly language level is defining conventions to be used for linking modules together. Obviously a JSR/BSR should be used to enter a module and an RTS for leaving it but what about passing inputs and results to and from the module? Also, what registers can the module use without destroying information belonging to the calling program? One possible linkage convention is to push all of the registers used by the module onto the system stack as the first operation of a module and pull the same set of registers plus the PC register (doing the return automatically) as the last instruction. This has the advantage of freeing any or all of the registers

for use by the module. This doesn't solve the problem of passing parameters between modules to which there are a number of solutions.

1) pass parameters in defined registers - this has the disadvantages of requiring each module to specify which registers it will use for what and restricting the size and number of parameters to whatever registers are available.

2) pass the parameter's ADDRESS in a register - this has the advantage of not restricting the size of the parameter to the size of the register but does not solve the problem of the number of parameters.

3) The parameters can be sent and returned to an area of storage following the subroutine call. This is referred to as in line parameter passing. For example:

```
        BSR MODULE
        BRA RTN1
        FCB SIZE
        FCC /this is a string parameter/

        as many extra parameters as necessary

RTN1    EQU *       Return point jumping round parameters after
                    module has finished.
```

4) Parameters, or their addresses, can be passed on the stack. This has the advantage that any number of parameters can be passed and returned. The only thing to be remembered is that the stack must be cleaned up at some point after the module is finished.

There are probably many variations on the theme of parameter passing and programmers are likely to have their own personal favourite on the grounds of efficiency or ease of use.

The question of whether to pass parameters or the addresses of parameters is a difficult one and has been a topic of discussion ever since computers were programmed! It is worth noting that the on the 6809 passing parameters by address is fairly easy. For example:

```
        PSHS X,Y    * X and Y point to the start of lists say
        JSR COMPARE * compare is a routine which compares two lists

        rest of main program

COMPARE LDA [2,S] * load A with first item in string
        LDB [4,S] * load B with first item in second string
        rest of subroutine
```

and apart from remembering to clean up the stack that's all there is to it. The use of one level of indirection is all that's necessary to get a parameter from its address on the stack!

RE-ENTRANT CODE

Re-entrancy is a fairly advanced concept and was once only discussed by system programmers. However with the 6809 it becomes useful not only for system applications but for real time control as well. A module is re-entrant if it can be stopped, by an interrupt say, used by another calling program and restarted, by a return from interrupt say, without any special precautions any number of times. Re-entrancy is useful in a timeshare operating system for example. If a BASIC interpreter is fully re-entrant then any number of users can use one copy of the program - a great saving in memory! A more ordinary example of the usefulness of re-entrancy is in a real time control system where a number of channels need roughly the same sort of attention but at different times and priority levels. If a channel of a lower priority is interrupted by one of a higher priority then if the processing program is re-entrant it can simply be called again and the processing of the lower priority channel will continue from where it left off following the RTI instruction.

It is very easy to produce re-entrant code for the 6809. When a program is interrupted (except by a FIRQ) all of the registers are stacked (and hence saved) so if a program uses only the registers for data storage it may be called again without changing the old state of the program, now stored on the stack. However if it does use areas of memory for data storage then we have to find a way of using such areas in a re-entrant fashion. (See next section.)

Re-entrancy is closely related to recursion - the ability of a program to call itself. Recursion is also something that is usually considered to be alien to assembly language programming, it is usually discussed in high level language manuals and handbooks. However if an assembly language program is re-entrant then there is no reason why it should not call itself - so suspending operation of its current form and restarting itself with a new set of parameters.

Some programmers consider recursion an unnecessarily complex way of writing a simple program and their advice is to avoid it at all costs. This does not mean that re-entrancy is similarly damned!

LOCAL, GLOBAL AND TEMPORARY STORAGE

In the previous sections we have discussed various properties that we might like a program to have. It may not be obvious from our discussion that an important factor in achieving these objectives is the method used to store data. In the writing of a modular program, for example, the non interaction of the modules demands that each one has its own areas for storing data. In re-entrancy it is clear that each time the program is called it must create a brand new data area that is protected if the program is stopped and recalled. In PIC we must distinguish and treat differently data areas that move with the program and those that don't.

Because of this, it is worth examining ways of creating and using data areas.

A location that moves with a program is in some sense "inside" the program. Such locations are often referred to as LOCAL storage. Local storage may be simply an area within the program set aside for storing data. As long as this area is referred to by PC relative addressing it is position independent. It is also modular as long as no other program references it, but it is not re-entrant. A recall of the program would overwrite any information in the local storage.

The term local storage can be extended to include any storage that a program module has total control over, i.e that has been created by it and that it is the sole user of. This reinforces the idea that local storage "belongs" to a program module, rather than just being inside it. A method of gaining this more general form of local storage in a position independent way is to use one of the stacks. For example if we want three bytes of local storage - SIZE, COUNT and LOC - then the following code is position independent:

```
        LEAS -3,S       * reserve 3 bytes on the system stack
SIZE    EQU  0          * define SIZE as 0,S
COUNT   EQU  1          * define COUNT as 1,S
LOC     EQU  2          * define LOC as 2,S
        STA COUNT,S     * for example store A into COUNT
        CLR SIZE,S      * or clear SIZE
        .
        .
        .
        rest of program
```

The position independence of this local storage comes from the stack pointer being unaffected by the program moving. Notice that the stack must not be used with negative offsets for local storage because an interrupt would overwrite the same area! Also the system stack must not be used during the program and any information pushed onto the stack before the local storage was allocated cannot be recovered by PULLs. Apart from these problems this method of local storage is obviously modular and re-entrant. If the progam is interrupted and restarted then a new area of stack is allocated and the old area is restored intact after an RTI. The objection that this method of local storage paralyses the system stack can easily be overcome by using the user stack pointer. Not in the obvious way of simply using the user stack instead of the system stack, because the user stack does not automatically allocate new storage following an interrupt, it is in fact completely unaffected. Rather, the user stack pointer can be used to mark the position of the local storage on the sytem stack.

For example:

```
        LEAS -3,S  * reserve storage
SIZE    EQU 1
COUNT   EQU 2
LOC     EQU 3
        TFR S,U    * mark temps position on stack
        STA COUNT,U * store A in count for example
        PSHS B     * system stack may be used without altering Locals

          rest of program
```

This form of local storage is obviously position independent and re-entrant. The X or Y registers could be used in place of the U pointer if desired but this is a fairly typical use of U.

Using the stack for local storage is reminiscent of parameter passing in modular programming and indeed we can ask if parameters can be passed in a re-entrant way. The answer is that if parameters, or preferably their addresses, are passed on the stack, the subroutine can be interrupted, recalled and then restarted, i.e. they are re-entrant.

Sometimes local storage is only needed for a short time during a program. This is usually referred to as temporary storage. Again the stack is our most useful method of managing temporary storage. If it is required to store the value of any of the registers for a while then simply push them onto the system stack. The only things to watch out for are that the system stack is free to grow - i.e. it's not being used as a marker for local storage and that the temporary items are removed from the stack at the end. This method of temporary storage is obviously position independent and re-entrant.

The only sorts of location that we have not dealt with are the ones that do not move with a program. These are obviously outside the program in some sense and are often called GLOBAL locations. It may seem at first sight as if there are two sorts of global locations, those that are absolutely fixed by some hardware definition such as an I/O port, and those that are fixed relative to some OTHER program, say an operating system. In fact this is an unnecessary distinction because we can imagine a hierarchy of programs calling lower programs. At each level a storage location is either LOCAL to that level, i.e. created and owned by that level, or owned by a higher level and is GLOBAL. For example, in a user program some variables are obviously LOCAL; others belong to a higher level, the operating system or the machine hardware, and are GLOBAL. Moving up one level to the operating system any variables that were LOCAL to the lower levels are inaccessible, some variables are again obviously LOCAL and other variables belong to a higher level, the machine hardware, and are again GLOBAL. If we adopt this view, then machine defined constants are simply GLOBAL locations defined at the highest possible level!

Methods of allocating global storage can obviously be considered from the point of view of local allocation if we move up the hierachy to a level

where the storage is local. We can therefore ask global storage to be position independent and re-entrant at its appropriate level but what about the way in which it interacts with the program levels in which it is global ?

The first important requirement is that global locations should be accessible in the same way at all the lower levels. This is different from passing parameters to subroutines where only the subroutine is given the values of the parameters, any lower levels must have the values passed on down to them. From this point of view passing parameters is simply allowing a program to initialise the values of the local storage of a program lower in the hierachy.

There are several ways of defining global locations. First we can use the fact that the highest program in the hierachy never actually moves! We can define a number of global locations at the machine (or absolute) level and use these to hold the locations of the globals currently required. If the locations are the start addresses of other programs this is referred to as an indirect jump table, otherwise it is known as an indirect data table. For example, it is common for operating systems to define a set of memory locations containing the addresses of their major subroutines (read a character, print a line etc). This technique is ready made for the 6809 with its one level of indirection and has the added advantage that if the location of one of the globals changes then programs lower in the hierachy are unaffected as long as the jump vector is altered. In fact the advantages are so great that JUMP TABLES AND INDIRECT DATA TABLES SHOULD BE USED WHEREVER POSSIBLE.

A second method of defining global locations is to reserve local storage on the user stack. Programs lower in the hierarchy can access the global locations as long as the user stack pointer is not moved. This is a very restricting way of using the U register.

EFFICIENCY

This section on efficiency is shorter than the section on elegance because optimising a program for size or speed is a very specialised topic and depends very much on the type of program being optimised. As for elegance and efficiency, there is usually a trade off between speed and size. If a program needs to be optimised then the following strategies should be considered:

1) For an increase in speed and saving in memory try to use the direct page register for addressing as often as possible.

2) For an increase in speed try to use the registers as much as possible without swapping to and from memory.

3) For a saving in memory convert similar pieces of code to one subroutine.

4) For an increase in speed try to "unwind" the program where possible. Unwinding means changing subroutine calls to copies of the subroutine placed in the program where it is called and writing out loops explicitly i.e. if something has to be done five times then write it out five times !

5) For speed optimisation only, examine the parts of a program executed repeatedly. Examine loops in order of use and number of repeats. Never examine single instructions unless critical in some timing operation. Try to remove as much code as possible from inside loops, such as parts of arithmetic expressions that do not depend on the loop index.

6) For an increase in speed avoid the repeated use of expensive addressing modes such as indirection.
e.g. instead of :

```
        LDB #4
LOOP    ROR [TOP]
        DECB
        BNE LOOP
```

use :

```
        LEAX [TOP]
        LDB #4
LOOP    ROR 0,X
        DECB
        BNE LOOP
```

which saves a total of 11 cycles.
In general it is often quicker to use LEA to compute an address once before using it repeatedly in a loop.

7) If all else fails, try to find another algorithm to do the same thing. For example, to save space evaluate a function such as sin(x) by a formula but to increase speed use a look up table of values.

The 6809 is such a versatile micro that a whole book could be devoted to programming tricks and every programmer will develop his own special favourites. The comments made earlier in the chapter should be kept in mind however, and tricks should be avoided unless necessary and then should be well documented as part of the program. If speed or space optimisation is a crucial problem when using the 6809 always remember that a double speed version is available and going from single to double speed is likely to show a bigger improvement than any software change could make.

SPECIAL DATA TYPES - ONE- AND TWO-DIMENSIONAL ARRAYS

The 6809 has many features that make it especially easy to handle more advanced data types, such as tables etc. One data type that is so important that it deserves a mention is the array. The one- and two-dimensional array is usually thought of as the province of high level languages but the 6809's multiply instruction makes array access easy. We will assume that the reader is familiar with the concept of an array and go directly to the details of implementation.

Arrays are often dealt with by the use of a storage mapping function. A storage mapping function relates the value of the current index(s) to the area of memory allocated to the corresponding array element. For example, a one-dimensional array of N elements each of M bytes in length, the first corresponding to an index of 0 (the modification for 1 is easy), may be implemented by :

address of start = start address of array + iM
of ith element

A two-dimensional array of N1 by N2 elements each M bytes in length, the first corresponding to indices of 0,0, may be implemented by :

address of start = start address of array + iM + (N1+1)Mj
of i,jth element

The translation of these two storage mapping functions into a 6809 program is easy. In the one-dimensional case :

```
LDB SIZE    * Size=number of bytes per element
LDA EYE     * EYE= index of required element
MUL         * form iM in D register
ADDD START  * add to start address of array to give the
            * the required address in the D register
```

In the two dimensional case :

```
LDB SIZE    * SIZE = number of bytes per element
LDA EYE     * EYE = first index of required element
MUL         * form iM in D register
PSHS A,B    * save D register
LDB N11     * N11 = N1+1
LDA JAY     * JAY = second index of required element
MUL         * form (N1+1)Mj in D register
ADDD ,S++   * add  iM to (N1+1)Mj, result in D and restore stack
ADDD START  * add to start address of array to give the
            * required address in the D register
```

(Note: EYE and JAY must be less than 256.)

CHAPTER SEVEN
CONVERTING 6800 PROGRAMS

Although the 6809 has a similar architecture to the 6800 it is not upward compatible in the sense explained in chapter one. Most of the 6809's operation codes are different from the 6800's so it is not possible to take a 6800 machine code program and run it on the 6809. It is however easy to convert a 6800 assembly language (source code) program to a 6809 assembly language program and then assemble it to a 6809 machine code program. The trouble is that very often users only have access to the machine code form of the program. In this case either the program must be disassembled on a 6800 system and then converted to the 6809, or the originator of the program must be contacted. Most software suppliers are willing to replace a 6800 program with a current 6809 version for a small update charge - some will ask you to buy the program again!

SIMULATED 6800 INSTRUCTIONS

One of the first problems in converting 6800 programs for the 6809 is the absence of certain standard 6800 instructions. For example the 6809 has no ABA (add B register to the A register) instruction. This shortage of instructions is easy to solve by using other 6809 instructions to the same effect. A number of 6809 Assemblers automatically translate any 6800 instructions encountered in a program to 6809 equivalents. For general interest and for assemblers with only 6809 instructions the standard equivalents are:

6800 Instruction	6809 Equivalent	
ABA	PSHS B	
	ADDA ,S+	
CBA	PSHS B	
	CMPA ,S+	
CLC	ANDCC #$FE	
CLI	ANDCC #$EF	
CLV	ANDCC #$FD	
CPX	CMPX	(Note!sets all condition codes see later)
DES	LEAS -1,S	
DEX	LEAX -1,X	
INS	LEAS 1,S	
INX	LEAX 1,X	
LDAA	LDA	
LDAB	LDB	
STAA	STA	
STAB	STB	
PSHA	PSHS A	
PSHB	PSHS B	

PULA	PULS A
PULB	PULS B
SBA	PSHS B
	SUBA ,S+
SEC	ORCC #$01
SEI	ORCC #$10
SEV	ORCC #$02
STAA	STA
STAB	STB
TAB	TFR A,B
	TST A
TAP	TFR A,CC
TBA	TFR B,A
	TST A
TPA	TFR CC,A
TSX	TFR S,X
TXS	TFR X,S
WAI	ORCC #$FF

The 6809 equivalents are mostly simple substitutions. The only ones likely to cause any trouble are the composite ones that make use of the system stack - be careful that there is enough stack - and the CPX instruction. On the 6800 the CPX instruction was not a true sixteen bit operation and only set the Z bit in the condition code register correctly. Thus one could check to see if the X register was the same as some value but not smaller/greater than it. On the 6809 the equivalent CMPX sets all of the condition code bits correctly. It is difficult to think of a correct 6800 program where this would make any difference but it is worth bearing in mind during debugging. The TXS and TSX pair are also difficult to translate exactly but the reason for this is better dealt with in the next section.

THE SYSTEM STACK

The differences between the operation of the 6809's system stack and the 6800's stack are the most troublesome in converting programs. The 6809 stack pointer S points at the LAST item placed on the stack instead of the location below the last item (i.e. the next free location) as in the 6800. The consequences of this difference are:

1) The stack pointer can be initialised one location higher on the 6809.

2) Any 6800 program that does all its stack manipulation through the X register (i.e. LDX #TEMP, TXS instead of LDS #TEMP) will have a correct stack translation when assembled for the 6809. The reason for this is that the 6800's TXS instruction automatically decremented the contents of the X register. (Thus if X points at TEMP a LDAA 0,X will load A from TEMP and following a TXS a PULA also loads TEMP into A.) Similarly a TSX instruction automatically loads the X register with one plus the contents of the Stack pointer - leaving the X register pointing at the first item on the stack. The 6809 equivalents of TSX and TXS given above

do NOT add or subtract one to the value of X or S and thus give the same results as the 6800 case. That is, transfers between the S and X registers always result in the destination register pointing to the same location as the source register.

The first problem is slight compared to the second. Unfortunately most medium to large 6800 programs do not do all their stack manipulation via TXS and TSX. The reason for this is simple. One of the problems with the 6800 is the lack of a second index register and many programmers tended to use the stack pointer as an occasional second index register. The X register was therefore often occupied by the time the stack pointer was brought into use and an immediate load of S seemed the best way to bring it into action. For example a typical untidy use of the 6800's stack pointer is to compare two strings of characters:

```
        LDX #STRG1          #STRG1=start of first string
        STS SVSP            #Save current stack pointer in SVSP
        LDS #STRG2-1        #STRG2=start of second string
        LOOP LDAA 0,X       #Load A with character from string 1
        INS                 #Move to next character in string 1
        CPX #LEN            #Last character ?
        BEQ END             #Branch to equal exit
        PUL B               #Load B with character from string 2
        CBA                 #Compare two characters
        BNE NEQ             #Branch to not equal exit
        #
        END LDS SVSP
            rest of program
        NEQ LDS SVSP
            rest of program
```

Notice that this program cannot be interrupted because the saved registers would overwrite the second string! Also notice that the stack pointer is loaded with #STRG2-1 so that the first PUL B gives the first character of string 2 and that the stack pointer moves its way along without any INS or DES instruction.

To translate this program to the 6809 all that is necessary is to load the stack pointer with #STRG2 rather than #STRG2-1. Although this seems simple enough real programs can become rather more confusing. Instead of loading S with #STRG2-1 a 6800 programmer might have loaded it with #STRG2 and then carried out a DES to correct the stack pointer. The DES instruction could be placed well away from the first loading of the stack pointer and cause the programmer carrying out the translation some time to work out if the DES should be deleted or not. The only way to translate a general stack using program is to follow the logic through and adjust the stack pointer when necessary.

My personal preference is to replace any section of a program that uses the stack pointer as an additional index register by an equivalent section of code using the 6809's Y register.

The stack causes trouble in two other areas - stacking order and

stacking length. The 6809 stacks five more bytes for each NMI,IRQ, or SWI than the 6800 and this must be allowed for in the memory allocated to the system stack. A more difficult problem is the difference between the 6800 and the 6809's stacking order. The 6809's stacking order can be seen at the end of chapter four. The most important points are:

1) the stacking order of the A and B register is reversed this allows A to stack as the most significant byte of the D register.
2) the stacking order invalidates any program that accessed the X or PC on the stack

The changes that are necessary to make any such programs work are obvious:

If X points to the first item on the stack then use any of

1,X change to 2,X
2,X change to 1,X
3,X change to 4,X
4,X change to 1R,X.

CONDITION CODES AND BRANCHES

There are a number of differences between the 6800 and 6809's use of the condition code register.

The first difference is that the 6809 uses the top two bits (b7,b6) of the condition code register whereas the 6800 ignores them. The only place that this can cause problems is if TPA or TAP instructions are used to manipulate the condition codes via the A register. The solution to this problem is to simply ensure that the top two bits of the A register are set to the required values before the TAP instruction and ensure that the top two bits are not used in the result of a TST or CMP instruction after the TPA.

A more difficult set of problems arises from the different ways some 6809 instructions affect the condition codes.

1) The 6809 TST instruction does not affect the C flag whereas the 6800 clears it.
2) All of the 6809 right shifts (ASR,LSR,ROR) do not affect the V flag whereas the 6800 sets it equal to the exclusive OR of bits seven and zero.
3) The 6809 H flag is not defined as having any particular state after subtract-like operations (CMP,NEG,SBC,SUB) the 6800 clears the H flag under these conditions.
4) The 6809 CMPX instruction sets all flags correctly whereas the 6800 only sets the Z flag correctly.

These differences should not cause any trouble in translation of correct 6800 programs. Any correct 6800 program would not use the state of the C bit following a TST instruction because a carry cannot result from subtracting zero, or use the state of the V flag following a shift right because an overflow condition cannot exist after a shift right (division by 2), or use the state of the H flag following a subtract-like instruction because a half carry flag is only used for BCD addition (BCD subtraction is done by adding nine's complement numbers).

The 6800's actions on the condition code bits are either due to an extension of what happens on related instructions or a rigid application of the flag's meaning. For example, an overflow condition can occur on a shift left instruction (this corresponds to multiplication by two) and this can be detected by the exclusive OR of the N and C flags after the shift has occurred. This approach carries over to shift rights even though it is difficult to give meaning to it. In the case of the TST instruction, if no carry can be generated by subtracting zero then the carry must be zero and the C flag should be reset. The 6809's approach to flags is best summed up by, "if it doesn't convey any information leave it alone".

The only problem is that 6800 programmers do not always produce correct 6800 code. However, even incorrect code sometimes works. This can happen for two reasons, either the misunderstanding does not matter or the incorrect code is being used in some very clever way. An example of the first reason may be found in the TINY ASSEMBLER 6800 by J Emmerichs, published by Byte, 1977) where we find the following code:

```
        rest of program

        TST ERRSW
        BNE END

        rest of program
```

On the 6800, the BHI instruction following a TST has the same effect as a BNE instruction (C is always zero) which is what the programmer intended. However on the 6809 the setting of the C flag depends on the last instruction that affected it and the operation of the BHI is erratic. Examples such as this can be found throughout the TINY ASSEMBLER 6800 code involving BHI used as BNE and BLS as BEQ. The remedy for this sort of problem is simple, either follow each TST instruction by an ANDCC #$FE or change each branch to its appropriate correct form.

I have not come across any careless or intentional misuses of the other condition code anomalies, but they should be borne in mind if an otherwise correct program behaves erratically after conversion.

PURE AND IMPURE PROGRAMS

An impure program is one which modifies itself in the course of running. Although impure programs are always to be discouraged, the 6800's shortcomings often forced clever programmers to use self-modifying techniques to gain speed or save memory. One of the most

common forms of impure code involved the run time modification of the constant offset byte in an indexed instruction. For example suppose A contains the desired offset then:

```
        STAA INCR
INCR    EQU *+1
        LDAB 0,X
```

would cause B to be loaded from the location given by the contents of X plus the contents of A. Notice that the label INCR is set to equal the current program position + plus one, that is the second byte of the LDAB 0,X instruction and hence the instruction changes each time the program is carried out. On the 6809 the same effect can be obtained by LDAB A,X and there is no need for impure code. The 6800 code as given above will not work on the 6809 because the second byte of the LDB 0,X instruction contains other information than the value of the constant offset - indeed zero offset is treated as a special case, see chapter three. A direct translation would have to force the index mode to go to the eight bit constant offset mode and remember that three bytes are then used, the third entirely for the offset value. Hence a correct translation would be :

```
        STA INCR
INCR    EQU *+2
        LDB $FF,X
```

Many other similar examples can be found in commercial code.

The main trouble with the conversion of impure programs is detecting that they are impure in the first place. The sure sign that a program is self-modifying is the presence of a label that is attatched to the middle of an instruction(but, apart from that it's simply a question of checking that no part of a program stores data in or modifies (for example, increments) any other part of the program.

TIME

Many applications of the 6800 use the time which an instruction takes to complete as a method of delaying a known amount of time. For example a typical delay loop might be:

```
        LDAB #100
DELAY   DEC B
        BNE DELAY
```

Obviously on the 6809 the delay produced would be different. The answer in this case is simply to work out how long the 6800 code took to execute in total and how long a single loop would take on the 6809 and then adjust the value loaded into B to give as near to the same time as possible. It is difficult to give any general advice on the question of translation of system timing, except to say that such timing loops are not good system design practice and should be replaced by the use of system clocks, SYNC instructions etc. wherever possible.

LENGTH OF CODE

A direct conversion of a 6800 program usually produces a 6809 program that takes up more space. The reason for this is that some 6800 instructions have to be coded as two 6809 instructions; also the often used "LD- constant,X" can take one extra byte and instructions such as INX now take the two byte form LEAX 1,X. In general a completely rewritten 6809 program doing the same task would take less or, at the worst, the same memory space. The reason for this is that the more powerful 6809 instructions/addressing modes can be used in place of a set of simpler 6800 instructions. For example LEAX 1,X would usually be absorbed into an LD- ,X+ instruction.

This change in code length can produce four sorts of problem:

1) lack of total memory space
2) overlapping of program and data areas
3) the destination of relative branches can be made greater than plus or minus 128 bytes
4) fixed size relative branches can be invalidated

Problem one can be solved by either optimising the code or acquiring more memory space.

Problem two is slightly more difficult. If a data area is defined to start at a fixed location (usually by an ORG - origin statement) it is possible for a program area to grow sufficiently to overlap. The solution is to move either the data area or a section of the program code to somewhere safe. The most important point is to check for ORG statements embedded in the program and see if they are still valid.

The third problem is trivial. The solution is to simply replace every branch causing an error by a long relative branch. Remember however, that this increases the length of the program and branches that did not cause problems before may cause errors at the next assembly.

The fourth and last problem is difficult in that it can be hard to spot. A fixed length relative branch is one that is not automatically computed by the assembler. For example:

```
BNE *+3
LDX $FF,X
NOP
```

(where * means the current point in the program). This will cause a branch to the NOP instruction. When converted to 6809 code the +3 offset is too small because LDX $FF,X takes an extra byte. The solution in this case is to change the 3 to a 4 or label the NOP instruction and branch to the label.

The trouble with fixed length relative addressing is that it often goes unnoticed in the middle of an otherwise trouble-free program. So check for any arithmetic expressions in the address field of ANY instruction.

There are a wide range of computer systems based on the 6809, from small industrial units for process control to large timeshare systems. Until recently the 6809 has been associated with a particular hardware bus standard - the S50 - and a particular operating system - FLEX. However the introduction of popular units by Tandy and Acorn have allowed a wider selection of basic hardware and the introduction of OS9 by Microware, Uniflex by TSC and the availability of full UCSD PASCAL have widened the range of operating systems. This increased choice is both a blessing and a curse. Previously the 6809 community was more or less committed to the S50 and FLEX. Thus a reasonable degree of standardisation was possible and hardware and software could be exchanged between systems with little difficulty. Before going on to describe the "other" systems we will examine the S50 bus - where it all started.

THE S50 BUS

The first popular microcomputers were bus oriented devices. That is, they used a standard set of plug in cards to allow the user to select the required performance. This approach is less common these days because of the reduction in hardware costs. It is now easier to supply as much hardware as possible - even if it is not required - on one printed circuit board. This is often referred to as the one board approach. The most successful bus standard was the S100. It was used mainly with the 8080 and Z80 processors and is an obvious standard by which to judge any other bus. A brief comparative history of the S100 and S50 bus can be seen in Table One.

The basic structure of the S50 bus can be seen in Table Two. Nearly all of the bus lines are derived from the 6800 MPU's connections. Sixteen address lines provide the same amount of addressing as the S100. Eight bi-directional data lines contrast with the S100's sixteen uni-directional data lines. Most of the other lines are fairly straightforward and self-explanatory. Anyone familiar with the S100 will be surprised at the relatively few control lines used. That they are enough, is something that can only be proved by experience.

The greatest difference between the S50 and the S100 is, in fact, not part of the main bus definition at all. The S50 bus has an auxiliary I/O bus consisting of 30 pins. (Not strictly a bus at all because not all the pins are paralleled.) This is sometimes referred to as the S30 bus and its specifications can be seen in Table Three. The most unusual feature of the S30 bus is the presence of pin 1 an I/O select pin. The S50 bus is so organised that every S30 bus slot occupies a certain number of address locations (usually four, but see the definition of the S50C later) and when an address in the slot's range is output on the main bus the I/O select pin goes low. This means that any I/O card plugged into an S30 slot need

only examine pin 1 to discover if it is being addressed or not. Thus I/O cards need very little circuitry for this purpose.

Although not part of the S50 standard, most S50 computers have eight S30 I/O ports, usually at the rear of the main chassis. As the S50 bus is organised around the 6800 MPU the S30 I/O bus is organised around the 6800's peripheral chips - the 6820 PIA, and the 6850 ACIA. Thus RS0 and RS1 are used as register select lines to determine which control/data register of a 6820 is being addressed. Having only two register selects, means that each S30 slot can only access four I/O registers. Thus, more advanced peripheral chips, such as the MOSTEK 6522 VIA, cannot be used. (A problem overcome with the advent of the S50C extended bus, see later). To recap, each S30 slot has one I/O select pin which goes low when the slot is addressed and occupies four distinct addresses in the main memory space, usually referred to as an I/O port.

EXTENDED ADDRESSING - THE S50C BUS

With the 6809 came the need to increase the addressing range of the S50 bus. Also some extra control lines used by the 6809 are not included in the S50 bus definition. These problems have been overcome by the S50C bus definition, the main features of which can be seen in Table Four. The new S30C bus definition is given in Table Five. The main improvements are the provison of four extra address lines, giving access to one megabyte of main memory, and two extra register select lines, giving each I/O port sixteen memory locations. These two details make the S50C bus ready for the next generation of micros. Comparing the S50C with the S50 definition indicates that S50/S30 devices will work on the S50C/S30C bus with little or no modification. Going the other way is not always so easy but some manufacturers make plug-in cards that can be used on both versions of the S50.

SOME REAL PRODUCTS

S50 BUS

South West Technical Products Corp
219 W. Rapsody,
San Antonio,
TEXAS 78216 (512) 344-0241

SWTPC is the main manufacturer of S50 devices. Their product line includes standard S50 systems of 8-56k bytes, multi-user systems of 128k and larger, a "super intelligent" VDU, eight inch double sided/double density disks, five inch double sided/double density disks, and a wide range of interfaces.
Special points to note: they produce the lowest cost 6809 CPU card and an excellent process control 6809 CPU card with on-board 8k RAM.

Smoke Signal Broadcasting
31336 Via Colinas,
Westlake Village,
CA 91361 (213) 889-9340

Smoke Signal offer a complete range of systems. Included in their product range are an interesting disk controller, and an advanced CPU card.

Midwest Scientific Instruments
220 W. Cedar,
Olathe,
Kansas,
66061 (913) 764-3273

MSI offer a complete range of systems including disk drives etc.

GIMIX Inc
1337 West 37th Place,
Chicago,
IL 60609 (312) 927-5510

The GIMIX product range includes complete systems from 8k to over 128k bytes, a superb main chassis-box, mother board and constant voltage power supply, a high resolution (256x256) graphics board, a very powerful and versatile CPU card, disk controllers etc. Special point to note: GIMIX produce some of the highest quality boards available for the S50 bus. They are very committed to the scientific/process control side of computing.

S100 BUS

Ackerman Digital Systems, Inc.,
110 N. York Rd,
Suite 208
Elmhurst,
Illinois 60126

An S100 compatible 6809 card is available from Ackerman Digital Systems. This includes a serial port, 2k of RAM, and up to 16k of ROM. It makes a good alternative for anyone with a standard S100 system already. The main problem of the 6809 on the S100 bus is the lack of standard software. This can be overcome to some extent if the FLEX operating system can be customised to run with the available disk drives but this would have to be done for every S100 disk controller before FLEX software could be generally transferred.

NON-STANDARD BUSES

Acorn Computers,
4a Market Hill,
Cambridge,
ENGLAND
CB2 3NJ

ACORN computers are best known for their low cost 6502 based machine the ATOM. However they also produce a number of EUROCARD size modules based on the 6800 and the 6809. These are ideal for process control type work and may be mounted in a standard 19" lab rack. The 6809 system can run FLEX so it is fully software compatible with most other 6809 systems. The only drawback of the ACORN product is that it is a non-standard bus and any add-ons etc. must (at the moment) be purchased from ACORN.

Motorola
(contact through local agent or distributor)

Motorola has always supplied computer systems based on its own products. These have traditionally been called development systems and have been intended as ways of developing dedicated hardware/software for control applications. Their first range of products were based on the 6800 and given the general name of "Exorciser" boards. These were of a high quality but rather expensive. The bus standard used is similar to the S50 but not identical and cards are not exchangable. A 6809 CPU is available in this range.

More recently, Motorola has introduced the Exorset 30 for development work using the 6809. Unlike previous Motorola systems the Exorset looks like a computer, with everything in one VDU-like case. It comes complete with a pair of 5" floppy disk drives and a cassette interface. Indeed to use the Exorset no extra hardware is required. (Although extra I/O interfaces etc. can be installed inside the case via a Motorola defined 86 pin bus.) The Exorset is excellent in terms of hardware quality and design but the same cannot be said for its supplied software. The operating system XDOS is primitive when compared to FLEX and the standard editor/assembler pair are nothing special. Motorola recommend their new basic interpreter/compiler, BASICM for use with the 6809 and it does have some interesting features. It is capable of producing a position independent (and hence ROMable) pseudo code which is a very desirable feature for development work but all in all I would rather see a true compiler for BASIC. BASICM would be even more interesting if a FLEX version were available.

The main problem with all Motorola hardware has always been the relatively high cost (an equivalent S50 based system is approximately half the price) and the relatively poor software available.

Tandy
(contact through local agent or distributor)

One of the most important things to happen to the 6809 was its selection by TANDY for use in its colour computer. Tandy are best known for their range of TRS80 computers - the best selling Z80 computers in the world. The switch of Tandy to a new CPU, the 6809, when they have had such success with the Z80 says much for the 6809's quality. Some features of the Tandy colour machine are: 4-32k RAM, colour BASIC in ROM, cassette I/O port, six bit D to A converter, four channel six bit A to D converter and a number of colour display modes from 256 x 192 of four colours to a more coarse diplay with eight colours. The colour computer is neatly packaged with a full sized keyboard and rear exit connectors for expansion etc. The Tandy colour machine is aimed at the home market and computer game playing so it is relatively cheap (about £300) and Tandy expect sales to exceed those of the TRS80 models. Even though game playing is its main market, the colour computer should not be dismissed as a toy. It is powerful and sophisticated and capable of much. As it becomes more popular other manufacturers, including Tandy, should produce add-ons. One firm (PERCOM) have already produced an interface to the S50 bus making available the wide range of S50 peripheral cards usable from the Tandy Colour Computer.

Percom Data Company Inc
211 N Kirby Garland,
Texas 75042 (214) 272-3421

Percom manufacture a range of S50 boards, CPU cards and disk controllers.

SOFTWARE

To cover the full range of 6809 software in a book of this size is impossible. All that can be done is to recommend one or two pieces of good software and leave the user to follow up the addresses given.

FLEX

FLEX is, almost without argument, the standard operating system for the 6800 and certainly the best single-user operating system for the 6809. It is simple to use, user-friendly, reliable and technically sophisticated. Some of the software available under FLEX: BASIC (TSC), EXTENDED BASIC (TSC), PASCAL (Lucidata), ASSEMBLER (TSC), EDITOR (TSC), DEBUG - machine simulator (TSC), LAB BASIC - control language, TEXT PROCESSOR (TSC) DIAGNOSTICS (TSC), etc.

UNIFLEX

UniFLEX is TSC's answer to the need for a time share/multi-tasking operating system for the 6809. It is based on UNIX and is therefore powerful. However, it cannot be said to be as easy to use as FLEX. The maximum number of users is 12 and hard disks etc. can be supported. Software available under UniFLEX: BASIC (TSC), EDITOR (TSC), ASSEMBLER (TSC), TEXT PROCESSOR (TSC), PASCAL (TSC) etc.

OS9

OS9 is a new multi-user/multi-tasking operating system based on UNIX for the 6809. It is, like any UNIX system, more difficult to use than FLEX but has the advantage over UniFLEX in that it is available in a single-user version, making the changeover to larger systems less painful. Software available under OS9 is more restricted than FLEX but includes BASIC09, EDIT, ASSEMBER, DEBUGGER all from Microware.

UCSD PASCAL

UCSD PASCAL comes complete with an operating system of its own. It is unique in being available on all of the current micros. Software available under UCSD includes: BASIC, FORTRAN, EDITOR, PASCAL and a number of assemblers.

SOFTWARE SUPPLIERS

For FLEX, UniFLEX etc:

Technical Systems Consultants, Inc.
Box 2570,
West Lafayette,
IN 47906 (317) 463-2502

For OS9, etc:

Microware,
5835 Grand Avenue,
Des Moines,
Iowa 50304 (515) 279-8844

For LAB BASIC:

International Software,
P.O. Box 160,
Welwyn Garden City,
Herts, England (07073) 26633

For FLEX Pascal:

Lucidata Ltd,
P.O. Box 128,
Cambridge,
CB2 5EZ, England

For UCSD:

Tallgrass Technologies Corporation,
7623 W. 86th St.,
P.O. Box 12047,
Overland Park,
Kansas 66212 (913) 381-5588

INFORMATION

One of the most valuable sources of information on both 6809 hardware and software is 68 Micro Journal:

68 Micro Journal
5900 Cassandra Smith Rd,
Hixson,
TN 37343
U.S.A

My only comment is - subscribe!

Table One

	S100	SS50
Produced by	ALTAIR	South West Tech. Products (SWTP)
1st CPU	8080	6800
2nd CPU	Z80	6809
Other CPUs	8085	6502/Z80 (not popular)
I/O	256 undecoded	8 fully decoded 4/16 registers each
Improvements	IEEE S100	SS3C
Manufacturers	Many	Few large companies
Main OS	CP/M	FLEX
Software	wide range	not much applications software

Table Two

SS4 NAME (pins 1 to 50)	DESCRIPTION
D0	
D1	
D2	
D3	eight bi-directional
D4	inverted
D5	data lines
D6	
D7	
A15	
.	
to	16 address lines
.	
A0	
GRD	
GRD	Ground
GRD	
+8V	
+8V	
+8V	
-16V	
+16V	
not used	Location (index) pin
MRST	Manual Reset
NMI	Non Maskable Interrupt
IRQ	Interrupt
UD2	User defined
UD1	User defined
Ø2	Phase two clock (1-2 MHZ)
VMA	Valid address indication
R/W	Read/Write
Reset	
BA	Bus Available
Ø1	Phase one clock
HALT	
110b	110 baud line
150b	150 baud line
300b	300 baud line
600b	600 baud line
1200b	1200 baud line

Table Three

S38 NAME (pins 1 to 38)	DESCRIPTION
UD3	user defined
UD4	user defined
-12V	
+12V	
GND	
GND	
not used	Location (index) pin
NMI	
IRQ	
RS0	Register select 0
RS1	Register select 1
D0	
D1	
D2	
D3	eight bit
D4	bi-directional
D5	data lines
D6	
D7	
E2	
R/W	
+8V	
+8V	
128Kb	
64Kb	
32Kb	
16Kb	
1Kb	
RESET	
I/O SELECT	

Table Four

(only the changes are shown)

OLD S38 NAME	NEW S38C	COMMENTS
MRST	MRDY	Memory ready line (for slow memory)
NMI	BUSY	Bus in use
UD2	FIRQ	Fast interrupt request (New 6809 interrupt)
UD1	Q	Clock line
E2	E	Clock line
R1	BS	Bus status
1Kb	BUSRQ	Bus request
16Kb	S3	A19
32Kb	S2	A18
64Kb	S1	A17
128Kb	S0	A16

1

Table Five

(only the changes are shown)

OLD S38 NAME	NEW S38C	COMMENTS
UD3	RS2	Register select line two
UD4	RS3	Register select line three
NMI	FIRQ	Fast interrupt request
600b	4800b	
150b	9600b	

Table Six

CARD	COMMENTS	AVAILABILITY
A800 CPU	At least three types	NOW
6809 CPU	Four current, more planned	NOW
A8000 CPU	Not much information yet	4th Q 1981
Memory	All types from 4K to 128K with many different features	NOW
SERIAL	One, two or eight channel RS232	NOW
PARALLEL	One or four (2K bit) channels	NOW
TIMERS	Interrupt and interval	NOW
EPROM PRG	Both 2716 & 2708	NOW
EPROM CARDS	Both 2716 & 2708	NOW
A to D's	Fast 12 and 8 bit types	NOW
D to A's	Fast 12 and 8 bit types	NOW
VDU CARD	With low res graphics	NOW
HIGH RES	High resolution graphics card	NOW
DISK CONTROL	With drives for both 8" and 5"	NOW
DISK CONTROL	Without drives for 8" and 5"	NOW
PROTOTYPE	S30 and S50	NOW
EXTENDER	S30 and S50	NOW

Please note overleaf is a list of other titles that are available in our range of Radio, Electronics and Computer books.

These should be available from most good Booksellers, Radio Components Dealers and Mail Order Companies.

However, should you experience difficulty in obtaining any title in your area, then please write directly to the publishers enclosing payment to cover the cost of the book plus adequate postage.

If you would like a copy of our latest catalogue of Radio, Electronics and Computer books, then please send a stamped, addressed envelope to:

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

180	Coil Design and Construction Manual	1.50p
202	Handbook of Integrated Circuits (IC's) Equivalents & Substitutes	1.75p
205	Pocket Book of Hi-Fi Loudspeaker Enclosures	95p
221	30 Tested Transistor Projects	1.25p
222	Solid State Short Wave Receivers for Beginners	1.25p
223	50 Projects Using IC CA3130	1.25p
224	50 CMOS IC Projects	1.25p
225	A Practical Introduction to Digital IC's	1.25p
226	How to Build Advanced Short Wave Receivers	1.25p
227	Beginners Guide to Building Electronic Projects	1.50p
228	Essential Theory for the Electronics Hobbyist	1.25p
RCC	Resistor Colour Code Disc	20p
BP1	Pocket Book of Transistor Equivalents and Substitutes	95p
BP2	Handbook of Radio, TV & Ind. & Transmitting Tube & Valve Equiv.	60p
BP6	Engineers and Mechanics Reference Tables	75p
BP7	Radio and Electronic Colour Codes and Data Chart	40p
BP14	Second Book of Transistor Equivalents and Substitutes	1.75p
BP24	52 Projects Using IC 741	1.25p
BP27	Chart of Radio Electronic Semiconductor and Logic Symbols	60p
BP32	How to Build Your Own Metal and Treasure Locators	1.15p
BP33	Electronic Calculator Users Handbook	1.50p
BP34	Practical Repair and Renovation of Colour TV's	1.25p
BP35	Handbook of IC Audio Preamplifier & Power Amplifier Construction	1.75p
BP36	50 Circuits Using Germanium, Silicon and Zener Diodes	95p
BP37	50 Projects Using Relays, SCR's and TRIACs	1.75p
BP38	50 (FET) Field Effect Transistor Projects	1.75p
BP40	Digital IC Equivalents and Pin Connections	2.50p
BP41	Linear IC Equivalents and Pin Connections	3.75p
BP42	50 Simple L.E.D. Circuits	99p
BP43	How to Make Walkie-Talkies	1.50p
BP44	ICL8038 Projects	1.85p
BP45	Projects in Opto-Electronics	1.75p
BP46	Radio Circuits Using IC's	1.35p
BP47	Mobile Discotheque Handbook	1.35p
BP48	Electronic Projects for Beginners	1.25p
BP49	Popular Electronic Projects	1.45p
BP50	IC LM3900 Projects	1.25p
BP51	Electronic Music and Creative Tape Recording	1.75p
BP52	Long Distance Television Reception (TVDX) for the Enthusiast	1.95p
BP53	Practical Electronic Calculations and Formulae	2.95p
BP55	Radio Stations Guide	1.75p
BP56	Electronic Security Devices	1.75p
BP57	How to Build Your Own Solid State Oscilloscope	1.50p
BP58	50 Circuits Using 7400 Series IC's	1.50p
BP59	Second Book of CMOS IC Projects	1.50p
BP60	Practical Construction of Preamps, Tone Controls, Filters & Amps	1.45p
BP61	Beginners Guide to Digital Techniques	99p
BP62	Elements of Electronics – Book 1	2.25p
BP63	Elements of Electronics – Book 2	2.25p
BP64	Elements of Electronics – Book 3	2.25p
BP65	Single IC Projects	1.50p
BP66	Beginners Guide to Microprocessors and Computing	1.75p
BP67	Counter Driver and Numeral Display Projects	1.75p
BP68	Choosing and Using Your Hi-Fi	1.65p
BP69	Electronic Games	1.75p
BP70	Transistor Radio Fault-Finding Chart	95p
BP71	Electronic Household Projects	1.75p
BP72	A Microprocessor Primer	1.75p
BP73	Pumps Owners' Projects	1.95p
BP74	Electronic Music Projects	1.75p
BP75	Electronic Test Equipment Construction	1.75p
BP76	Power Supply Projects	1.75p
BP77	Elements of Electronics – Book 4	2.95p
BP78	Practical Computer Experiments	1.75p
BP79	Radio Control for Beginners	1.75p
BP80	Popular Electronic Circuits – Book 1	1.95p
BP81	Electronic Sandwicher Projects	1.75p
BP82	Electronic Projects Using Solar Cells	1.95p
BP83	VMOS Projects	1.95p
BP84	Digital IC Projects	1.95p
BP85	International Transistor Equivalents Guide	2.95p
BP86	An Introduction to Basic Programming Techniques	1.95p
BP87	Simple L.E.D. Circuits – Book 2	1.35p
BP88	How to Use Op-Amps	2.25p
BP89	Elements of Electronics – Book 5	2.95p
BP90	Audio Projects	1.95p
BP91	An Introduction to Radio DX-ing	1.95p
BP92	Electronics Simplified – Crystal Set Construction	1.95p
BP93	Electronic Timer Projects	1.95p
BP94	Electronic Projects for Cars and Boats	1.95p
BP95	Model Railway Projects	1.95p
BP96	C & Projects	1.95p
BP97	IC Projects for Beginners	1.95p
BP98	Popular Electronic Circuits – Book 2	2.25p
BP99	Mini-Matrix Board Projects	1.95p
BP100	An Introduction to Video	1.95p
BP101	How to Identify Unmarked IC's	95p
BP102	The 6809 Companion	1.95p
BP103	Multi-Circuit Board Projects	1.95p
BP104	Electronic Science Projects	2.25p
BP105	Aerial Projects	1.95p
BP106	Modern Op-Amp Projects	1.95p

BERNARD BABANI BP102

The 6809 Companion

- The 6809 microprocessor is becoming increasingly important as more and more manufacturers use it in popular machines.

- Mike James has been an enthusiastic user of the 6809 since it was first introduced and this book has been written for programmers who want to make the most of this powerful microprocessor.

- It is a work of reference which includes the topics needed in a machine companion: history, architecture, addressing modes and the instruction set (fully commented). In addition there are chapters on converting programs from the 6800, programming style, interrupt handling, and about the 6809 hardware and software that are available.

ISBN 0 85934 077 5

BERNARD BABANI (publishing) LTD
The Grampians
Shepherds Bush Road
London W6 7NF
England

£1.95