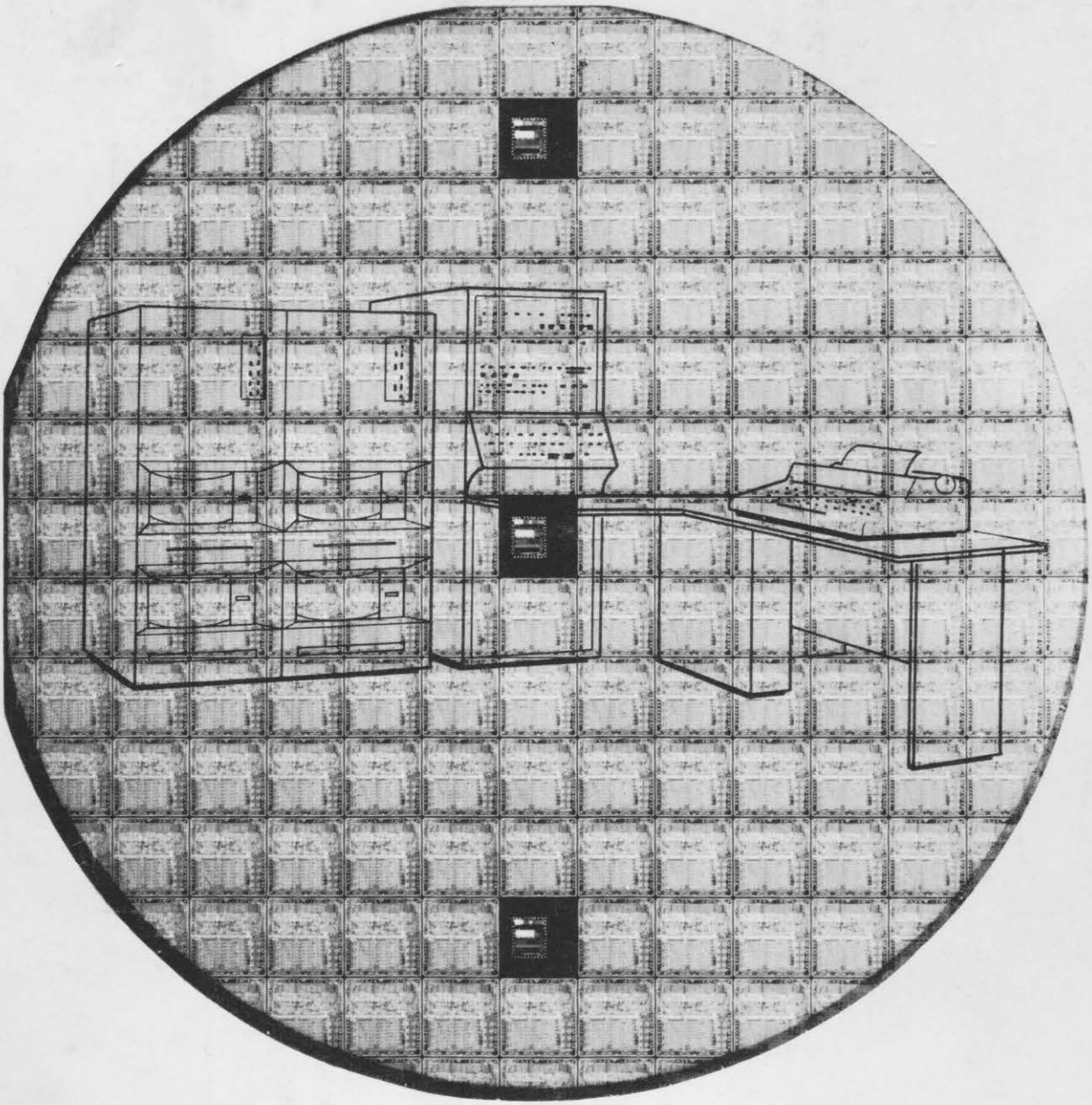
MC6809

PRELIMINARY PROGRAMMING MANUAL



MOTOROLA

MC6809

PRELIMINARY PROGRAMMING

MANUAL

Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Although the information in this document has been carefully reviewed for broad application, Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.

First Edition

FOREWORD

This Preliminary programming manual was excerpted from the system design specification for the M6809 and as such occasionally betrays its origin. It is, however, complete and correct and contains all the information necessary to construct a M6809 system and to write the software for that system. References made in this manual to the MC6801 also apply to the MC6803, and references the MC6802 also apply to the MC6808.

When a discrepancy is found between this preliminary manual and the MC6809 Advance Information Data Sheet, the data sheet takes precedence.

Further details pertaining to the assembly language syntax and M6809 assembler operation can be found in "Macro Assemblers Reference Manual", part no. M68MASR(D).

TABLE OF CONTENTS

1.0 OVERVIEW

The 6809 is an 8-bit NMOS microprocessor designed with particular attention to real-time programming and character-manipulation data processing. It is compatible with the 6800 microprocessor bus and family parts, and is capable of superior computing performance.

Even people who have not previously used the 6800 will find the 6809 a serious contender for their microprocessor business. The consistent and powerful instruction set makes our computer easy -- and even fun! -- to program. The enhanced architecture allows programming techniques that reduce the risk and increase the life of the programming investment. The resultant programs are fast and efficient. And, since our machine is byte-oriented (as opposed to 16-bit) it is best at processing byte quantities -- exactly the facility required for High-Level-Language and business data-manipulation.

People who <u>have</u> used the 6800 will find the 6809 very familiar and easy-to-learn. For example: the 6800 had one stack pointer; now the 6809 has <u>two</u> stack pointers, and a single instruction can push a register, a couple of registers, or the entire machine state (all visible registers) onto the stack. Another example: the 6800 had one index register; now the 6809 has <u>two</u> index registers. And <u>both</u> stack pointers are indexable. And so is the program counter. So the 6809 is not <u>different</u> from the 6800, just tremendously more capable.

1.1 DESIGN TARGET

The principal thrust for the design of the 6809 MPU was to create a processor which would improve our position in present markets, and the vast consumer markets still to come. We expect that markets such as Business Accounting, Word Processing, Scientific/ Business Programming, Medical Analysis, Communications Switching, etc., will find the 6809 an optimal choice.

1.1.1 Results of 6800 Analysis

Extensive analysis of difficulties in using the 6800 brought out a number of more-specific design goals for the 6809. These ranged from rather obvious improvements (like "greater throughput," "more registers," and "PUSH X") through those typical of professional architectural design ("consistancy," and "powerful addressing") to innovative attempts to crack the problem of expensive software ("position-independence," and "indirect addressing for I/O"). Next, we examine some of the ramifications of these improvements.

1.1.2 Hardware Improvements

A number of hardware difficulties are resolved from the original 6800 system: R/C $\overline{\text{RESET}}$, on-chip clocks, and improved bus-timing specs make the system easier to use and easier to run faster. Extensive analysis of the interaction between various control/response signals (interrupts, HALT, BA, RESET, IACK, etc.) has the new signals (READY) work with the old to handle multiple-processor and other new applications.

1.1.3 Throughput Improvements

The 6809 can provide a radical throughput improvement that qualifies it for a number of tasks previously unsuited to microprocessors. The enhanced architecture (additional index registers and stack pointers) and greatly-expanded addressing capabilities simplify algorithms and programming while speeding processing. New instructions and better bus-timing give us an even more powerful machine. And "optimizing" code using the new Direct Page Register can further increase speed and reduce program size.

But no matter how fast the machine goes, there will always be some application just out of reach, and it will always be "nice" to have the same job done in half the time. Many systems will use multiple processors for just this reason. But the fact of the matter is, once any machine can do your job in the time you require, throughput has ceased to be important. It is more important that the machine be easy to use and easy to program. The hardware designer can verify his work -- each system signal, if necessary -- by experiment. Not so the software designer, who can easily build systems that would take longer to exhaustively test than there has so far been life on Earth.

1.1.4 Software Improvements

Some things which facilitate program correctness are: Block Structure, High-Level-Language; and, at the machine level, a regular architecture, consistent instruction-set and logical assembly language. We have made a conscious attempt to minimize the number of assembly-language mnemonics, and to make those which remain apply consistently, both functionally and syntactically, to similar registers. We have nevertheless added some redundant mnemonics (LSL, BHS, BLO, BRN) to fill out particular instruction types, making them easy to remember and available for compiler-produced code.

1.1.5 Architectural Improvements

Perhaps the most powerful improvement we have made was to greatly expand the 6809's addressing capabilities over the 6800. Let's talk a little about "state-information". The true description of the state of a computer program includes the description of every bit in both the memory and the CPU. Compared to the memory environment in which it processes data, even register-oriented computers have a very limited amount of program state information available internally. By vastly-expanding the addressing modes, and making each apply to any of the four pointer registers, we orient the machine to saving most program state information in memory, where there is plenty of space, as opposed to in the CPU itself where it is very expensive.

1.1.5 (Continued)

Some CPU designers have gone even further, effectively placing their registers in memory, on the assumption that if a little of something is good, a lot is better. These machines must fetch data from memory, operate on it, then put it back - and they are inevitably slower.

1.1.6 Innovative Improvements

Perhaps most intriguing from an architectural point of view, are the features we included to attack the problem of high-cost software. While microprocessor-family sales would seem to be a business capable of exponential expansion, vast applications markets are still closed due to the unavailability of quality software. And the software is unavailable because of its high development costs and very low security.

1.1.6.1 ROM's For Low-Cost Software

One attack on reducing development costs is to move the results into mass production -- in this case, Read-Only-Memories. But ROM's are risky; if the software is not carefully designed, it will only apply to one system -- a custom product at custom economics. And a single software error could conceivably require that every unit in the field be recalled; the risk of software error cannot be amortized over the number of units produced.

1.1.6.1 (Continued)

The error problem will always require very careful modular testing, but by insisting on a regular architecture and logical assembly language, that risk is noticeably reduced. The problem of making the ROM applicable to large numbers of arbitrary hardware designs requires a solution to the problem of Position-Independent-Code (PIC).

1.1.6.2 Position Independence

By Position-Independent we mean that the exact same machine-language code can be placed anywhere in memory and still function correctly (PIC is also called "self-relative" code). The 6800 has a limited form of position-independent control-transfer in its branch instructions, and we have added long branches to complete this capability. But that is only part of the problem: it is also crucial that RAM storage for global, permanent, and temporary values be easily available in a position-independent manner. We suggest placing this data on the stack, since the stacked data is exceedingly easy to access and manipulate. It is suitable to stack the absolute addresses of I/O devices before calling a standard software package, and the package can use the stacked addresses for I/O in any system.

It is also necessary to be able to gain access to tables or data or immediate values in the text of the program; the LEA instructions allow the

1.1.6.2 (Continued)

user to point at data in a position-independent manner, as, for example:

```
        }
        LEAX    MSG1,PCR
        LBSR    PDATA
        }
MSG1    FCC     /PRINT THIS!/
```

Here we wish to point at a message to be printed from the body of the program. By writing "MSG1, PCR" we signal the assembler to compute the distance between the present address (the address of the LBSR) and MSG1. This result is inserted as a constant into the LEA instruction which will be indexed from the program counter value at the time of execution. Now, no matter where the code is located, when it is executed the computed offset from the program counter will point at MSG1. This code is position-independent.

1.1.7 Summary

In short, the 6809 microprocessor will provide the user with greatly-improved performance, reduced system-complexity, and radically new capabilities. Its innovative features will allow deep inroads to be made in quality low-cost programs.

1.2 SUMMARY OF FEATURES

1.2.1 Hardware

- o 8-Bit Data / 16-Bit Address Bus
- o MC6800 Bus Compatible
- o Single 5v Supply / 40 pins
- o TTL - Compatible
- o Fast Interrupt Request Input
- o Interrupts may be Vectored by Device
- o Two Status Outputs (BA and BS)
- o On-Chip Clock Version 4 x f_0
  - MRDY input for slow memory
  - DMA/BREQ input for DMA

1.2.2 Software

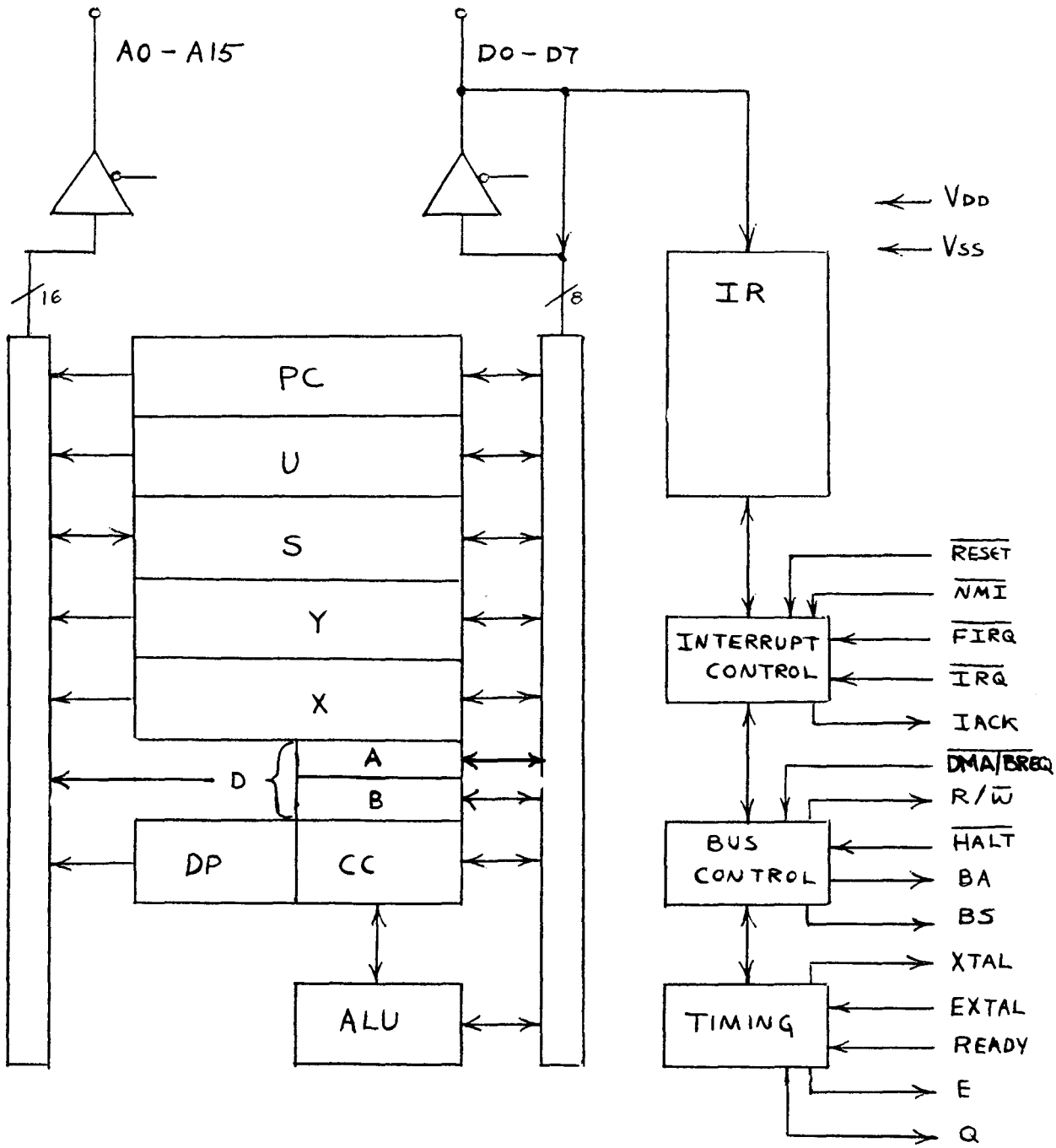
- o MC6800 Upward-Compatible Architecture
  - Two 8-Bit Accumulators
  - Two 16-Bit Index Registers
  - Two 16-Bit Stack Pointers (with index capability)
  - Programmable Direct Page Register
- o MC6800 Upward-Compatible Instruction-Set
  - 59 Instruction Mnemonics
  - 268 Opcodes
  - 1464 Instructions w/different addressing modes
  - 8x8 Unsigned Multiply
  - 16-Bit Arithmetic (Load, Store, Add, Subtract,Compare
  - Powerful Push/Pull Instructions
  - Powerful Register Transfers and Exchanges
  - Powerful Address-Manipulation Instructions
  - Extended-Range Long Branches

1.2.2 (Continued)

- MC6800 Upward-Compatible Addressing
  - 10 Addressing Modes
  - 24 Indexed Sub-modes
  - Indexing Applied From Either Index Register or Either Stack Pointer
  - Constant Indexing From PC
  - Indirect Addressing (Post-Indirection)
  - Up to 16-Bit Indexed Offsets
  - Auto-Increment/Decrement
- Fully-Supports Various Software Disciplines
  - Position-Independent Code
  - Non-Self-Modifying Code
  - Structured, Highly-Subroutined Code
  - Multi-Task and Multi-Processor Organization
  - Stack-Oriented Compiler Instructions
  - Re-Entrancy and Recursion

2.0 CHIP ARCHITECTURE

2.1 6809 BLOCK DIAGRAM



2.2 Pin Description

2.2.1 Signals of the 6809

2 - Power

16 - Address Bus

8 - Data Bus

1 - R/$\overline{W}$

1 - $\overline{RESET}$

1 - $\overline{NMI}$

1 - $\overline{FIRQ}$

1 - $\overline{IRQ}$

1 - $\overline{DMA/BREQ}$

1 - $\overline{HALT}$

1 - BA

1 - BS

1 - XTAL
1 - EXTAL
} 4x f0 only

1 - MRDY

1 - E out

1 - Q out

2.2.2 Power (V_{ss}, V_{dd})

Two pins are used to supply power to the part: V_{ss} is ground or 0v, while V_{dd} is +5.0v ±5%.

2.2.3 Clock (XTAL, EXTAL, E, Q, $\overline{DMA/BREQ}$, MRDY)

The pins XTAL and EXTAL are used to connect the on-chip oscillator to an external parallel-resonant crystal; this oscillator may take as long as 20 msec to become operational after power-on. Alternately, the pin EXTAL may be used as a TTL-level input for external timing; the crystal frequency or external input is 4x the bus frequency.

E is the standard 6800-bus system timing signal. The leading edge of E indicates to memory and peripherals that the address is (should be) sufficiently set-up to begin with operations ($\overline{E} \wedge Q$ is the address set-up time for peripherals). Data flows on the data bus during E and is latched on the trailing edge of E.

Q is a quadrature clock signal which leads E and which has no parallel on the 6800. Addresses from the MPU will be guaranteed good with the leading edge of Q.

2.2.3 (Continued)

$\overline{\text{DMA/BREQ}}$ is a request to temporarily suspend MPU operation and take the MPU off of the MOS bus. A $\overline{\text{DMA/BREQ}}$ is always accepted "immediately" (at the end of the next E) to insure a maximum asynchronous latency of one bus cycle (although the system bus will typically require a "dead" cycle before beginning an actual transfer).

The user may decode the bus grant state (BA $\wedge$ BA one-half-cycle-delayed) to place the DMA device on the MPU buses; this will be appropriate timing so as to eliminate bus contention both into and out of DMA. The MPU has an internal counter which will periodically switch the MPU back onto the bus, execute one cycle, then return to DMA operation. This automatic MPU refresh allows DMA operations of arbitrary length.

MRDY - Memory Ready is designed to extend the required data access time for use with slow memory (it does not increase address set-up time). is also designed to extend a memory access until a multi-processor shared-memory can respond to the access request.

When a memory-access is to be extended, MRDY should be LOW some setup time before the trailing-edge of E of that access cycle; the clocks will then be held in the $E \wedge \overline{Q}$ state. After MRDY is made HIGH, up to one-quarter bus cycle will elapse before the memory access is completed (at the trailing-edge of E). MRDY can only extend the memory access to 10 microseconds for the standard part (a 100 microsecond extension capability may be available as a selected version at increased cost).

2.2.4 Address Bus (A0 - A15)

Sixteen pins are used to place information from the MPU onto the address bus. Each pin will drive one standard TTL load (or four LS loads) plus eight 6800-family devices at rated bus speed. Additional MOS devices may be driven by eliminating the TTL load, or by reducing the bus rates. All address drivers are made high-impedence when output BA is HIGH. The address pins may start to change an address hold-time after the trailing edge of E, and they will be stable with the leading edge of Q.

2.2.4 (Continued)

When the processor does not need to use the bus for a data transfer it will send address $FFFF_{16}$ and $R/\overline{W} = 1$; this will replace the VMA function on the 6800. This dummy access may be differentiated from a RESET by not being acknowledged as an interrupt; i.e., the dummy access will have a $\overline{BA} \wedge \overline{BS}$ status, while RESET vector fetch will have $\overline{BA} \wedge BS$. It is recommended that the user not otherwise read access location $FFFF_{16}$ when decoding $FFFF_{16}$ as non-VMA.

2.2.5 Data Bus (DØ - D7)

Eight pins provide communication with the bi-directional data bus. Each pin will drive one standard TTL load plus eight 6800-family devices at rated bus speed. All data bus drivers are made high impedence when the BA output is HIGH. The period $\overline{E} \wedge \overline{Q}$ is used to tri-state the data bus to allow data bus turnaround without contention. The MPU will start to propagate data to the data bus with the leading edge of Q, but peripherals generally propagate data only during E. All data receivers require data to be valid some set-up time before E goes LOW, when data is latched in the receiving device.

2.2.6 Read/$\overline{\text{Write}}$ (R/$\overline{\text{W}}$)

One output pin indicates the direction of data transfer on the data bus; a LOW level on this line indicates that the MPU is sending data on the data bus. R/$\overline{\text{W}}$ is made high-impedance when the output BA is HIGH. R/$\overline{\text{W}}$ is good with the leading edge of Q, the same as the address bus.

2.2.7 Reset ($\overline{\text{RESET}}$)

A LOW-level on this Schmitt-trigger input (for at least one cycle) will Reset the MPU. The MPU will take 5 bus cycles for a complete Reset; this will abort the present instruction, jam $\emptyset\emptyset_{16}$ into the Direct Page Register, set the F and I mask bits in the Condition Code Register, and disable the $\overline{\text{NMI}}$ (until after the first load into the stack pointer).

Assuming that neither the $\overline{\text{HALT}}$ nor the $\overline{\text{DMA/BREQ}}$ pins are LOW, the MPU will begin operation immediately after $\overline{\text{RESET}}$ goes HIGH. The MPU will read data from locations $FFFE_{16}$ and $FFFF_{16}$, then use this data as the address of the first opcode to be executed.

Because $\overline{\text{RESET}}$ on the MPU is a Schmitt-trigger input which needs a higher '1' level than is required by the peripherals, a simple RC network can be used to

2.2.7 (Continued)

Reset the entire system. The peripherals will be fully out of Reset before the MPU can start operation and therefore before the MPU can attempt peripheral initialization.

During initial power-on, the RESET line should be held LOW until the clock oscillator is fully operational, and only then released.

If the $\overline{\text{HALT}}$ or $\overline{\text{DMA/BREQ}}$ pins are LOW when $\overline{\text{RESET}}$ returns to a HIGH level, the $\overline{\text{RESET}}$ positive-edge will be latched. The MPU will then wait until resumption of a Running state before completing the Reset. The MPU will not come out of tri-state during HALT or DMA even if RESET.

Since DMA operation may occur during RESET, DMA or $\overline{\text{MRDY}}$ may lengthen the total bus transaction period. A full Reset will take, therefore, correspondingly longer in terms of real time.

2.2.8 Halt ($\overline{\text{HALT}}$, BA, BS)

A LOW level on the $\overline{\text{HALT}}$ input causes a running MPU to halt at the end of the present instruction, and remain halted indefinitely without loss of data, until the $\overline{\text{HALT}}$ pin is driven HIGH. When the MPU is

2.2.8 (Continued)

halted, the BA output is driven HIGH (which indicates) that the buses are tri-stated) and BS is driven HIGH to indicate a HALT or DMA state. While halted, the MPU cannot respond to some real-time requests although a $\overline{\text{DMA/BREQ}}$ will always be accepted, and $\overline{\text{NMI}}$ or $\overline{\text{RESET}}$ will be latched for later response. Conversely, if the MPU is not running ($\overline{\text{DMA/BREQ}}$ or $\overline{\text{RESET}}$) the HALT state will not be achieved until the MPU is released with $\overline{\text{HALT}}$ LOW.

BA (Bus Available) is an indication of an internal control signal which tri-states the MOS buses(address, data, R/$\overline{\text{W}}$) on the MPU. This is a valuable signal for any form of bus-sharing or DMA, but does not imply that the bus will be available for more than one cycle. When BA transitions from a HIGH to a LOW state, an additional cycle will always elapse before the MPU regains the bus.

BS (Bus State) is an encoded pin which, in conjunction with BA, indicates the present MPU state.

Status indications are valid with the leading edge of Q.

2.2.8 (Continued)

BA	BS	MPU STATE
0	0	Normal (Running)
0	1	IACK
1	1	HALT + BUS GRANT
1	0	SYNC Acknowledge

SYNC Acknowledge is indicated on pins BA and BS ($BA \wedge \overline{BS}$) while the MPU is waiting for external synchronization (on an interrupt line). CWAI does not tri-state the buses and is not acknowledged.

Interrupt Acknowledge is indicated on pins BA and BS, ($\overline{BA} \wedge BS$) during both cycles of a hardware-vector-fetch (RESET, NMI, SWI, etc.).

Because the 6800 family does fetch vectors (most other MPU's do not) this signal, plus decoding of the lower four bits of the address bus, can provide high-speed interrupt capability (vectored by device) which other MPU's do not have.

External decoding logic can indicate which vector is being used (thus, which interrupt-level has been accepted), turn-off the vector-ROM (if ROM), and jam onto the data bus the address of the desired interrupt handler. This technique could drastically decrease interrupt latency compared to a polled approach.

2.2.8 (Continued)

It is not sufficient merely to decode a vector address to indicate a vector-fetch, since normal accesses, including indirect JUMPS, can be made to these locations. Such a normal access may well occur even after an external interrupt request has been received (it may be masked!).

2.2.9 Interrupts ($\overline{NMI}$, $\overline{FIRQ}$, $\overline{IRQ}$)

The interrupt system on the 6809 has been extensively analyzed to eliminate any unknown states from any combination of hardware signals and valid instruction operations. All interrupt inputs are latched during every Q, and will be delayed another bus cycle before they are seen by the MPU. $\overline{NMI}$ is edge-sensitive in the sense that if it is sampled LOW one cycle after it has been sampled HIGH, a NMI interrupt will be triggered. Because $\overline{NMI}$ is not masked by execution of a NMI, it is possible to take another NMI interrupt before executing the first instruction of the NMI routine. A fatal error will exist if an NMI is allowed to occur regularly before completing the RTI of the previous NMI, since the stack will surely overflow.

FIRQ and IRQ are both level-sensitive in the sense that the interrupt will be accepted anytime the running

2.2.9 (Continued)

processor sees FIRQ or IRQ and the associated mask bit both LOW. This means that the associated interrupt handler must cancel the original interrupt, or the program will never return to the interrupted routine.

FIRQ provides fast interrupt response by stacking only the return address and condition-codes. This will allow read-modify-write operations (like CLR, TST, INC, DEC, rotates, etc) with minimal overhead. Alternately, any desired subset of registers may be saved (and later recovered) using PSH/PUL instruction.

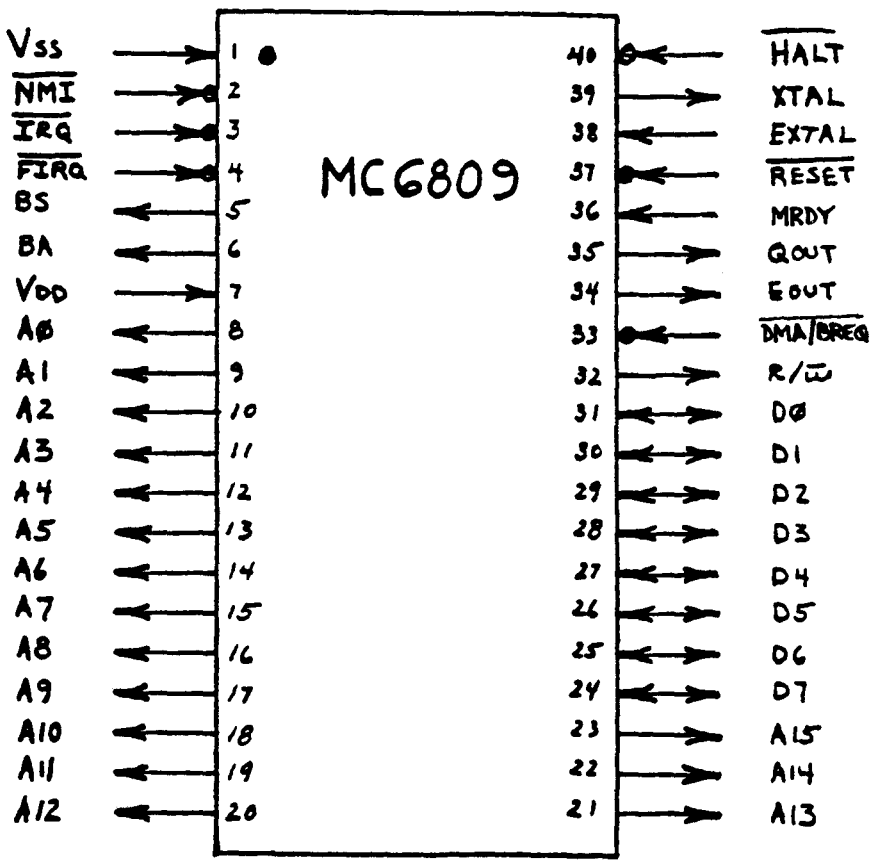
IRQ provides a slower response to interrupts, but stacks the entire machine state. This means that interrupting routines can use all CPU resources without fear of damaging the processing of the interrupted routine.

All interrupt pins can be used with the SYNC instruction which causes the processor to stop processing and tri-state its buses; any interrupt input then causes processing to resume. If that input was masked, the processor will simply execute the next in-line instruction. If that input was not masked (or was NMI) the interrupt sequence will occur. This means that the same interrupt line that is used for arbitrary interrupts can be used for periods of high-throughput program/device synchronization. Naturally, other devices on the same

2.2.9 (Continued)

line must be disarmed (disabled at the source).

All interrupt-handling routines should return to the formerly-executing task using an RTI instruction.


MC6809
VSS 1
NMI 2
IRQ 3
FIRQ 4
BS 5
BA 6
VDD 7
A0 8
A1 9
A2 10
A3 11
A4 12
A5 13
A6 14
A7 15
A8 16
A9 17
A10 18
A11 19
A12 20
40 HALT
39 XTAL
38 EXTAL
37 RESET
36 MRDY
35 QOUT
34 EOUT
33 DMA/BREQ
32 R/W
31 D0
30 D1
29 D2
28 D3
27 D4
26 D5
25 D6
24 D7
23 A15
22 A14
21 A13

2.4 USING 6809 BUS TIMING

2.4.1 DMA

The three 6800 methods of DMA (HALT-mode, cycle-stealing, and bus multiplexing) are also available on the 6809, and cycle-stealing is controlled by the chip itself (in the on-chip clock version).

Halt-mode DMA is achieved by pulling the HALT line LOW and waiting for a Halt+DMA acknowledge (BA $\wedge$ BS) =1 which will occur after the last cycle of the current instruction. The MPU will tri-state its buses to allow a DMA device to take over the MOS bus, and the bus clocks (E and Q) from the chip will continue to run to provide system timing for DMA transfers. The MPU may be held in HALT indefinitely, but the worst-case latency into Halt-mode DMA is 20 cycles (SWI2). The Halt-mode is terminated by bringing the $\overline{\text{HALT}}$ line HIGH; the MPU will resume normal operation one cycle after goes LOW.

Cycle-stealing DMA is handled (in the on-chip-clock version) by pulling the $\overline{\text{DMA/BREQ}}$ line LOW with the trailing edge of Q. The internal MPU clocks will stop and the MPU will start to tri-state its MOS drivers a hold-time after the trailing-edge of E (BA will go LOW). An external $\overline{\text{DMAVMA}}$ must be generated to disable the memory during the 'dead'

2.4.1 (Continued)

cycle between different bus masters. External logic may place the DMA device on the bus sometime during the last half of the dead cycle. The E and Q bus clock signals from the chip continue to run to provide bus timing for DMA transfers.

Synchronous latency into Cycle-stealing DMA is less than one-quarter bus cycle; asynchronous latency may be a full cycle longer. Cycle-stealing DMA is terminated by returning $\overline{\text{DMA/BREQ}}$ to a HIGH level with the trailing edge of Q, the DMA device must get off the bus a hold-time after the trailing-edge of E of the same cycle (BA = LOW). The MPU will start to come out of three-state at the end of the dead cycle. (Meanwhile, an external $\overline{\text{DMAVMA}}$ must be generated to eliminate the false memory access).

Cycle-stealing DMA is similarly available in the off-chip-clock version of the 6809, with the exception that all control and timing occurs external to the chip. This circuitry must assure that the MPU is suspended with clock signals $\overline{E} \wedge \overline{Q}$, while continuing to generate E and Q clocks for the system.

Bus-multiplexing DMA requires external buffers from the MPU which are gated onto the system buses during a portion of the MPU cycle (usually during E). Buffers

2.4.1 (Continued)

from DMA devices are gated on the system buses during $\overline{E}$, thus allowing 50% of the bus bandwidth for DMA.

2.4.2 Dynamic Memory

Dynamic memory is usually considered to be a high-priority form of cycle-steal DMA. That is, the refresh controller (possibly a DMA chip) accesses either 64 (for 4K RAM's) or 128 (for 16K's) consecutive locations within each 2 millisecond interval.

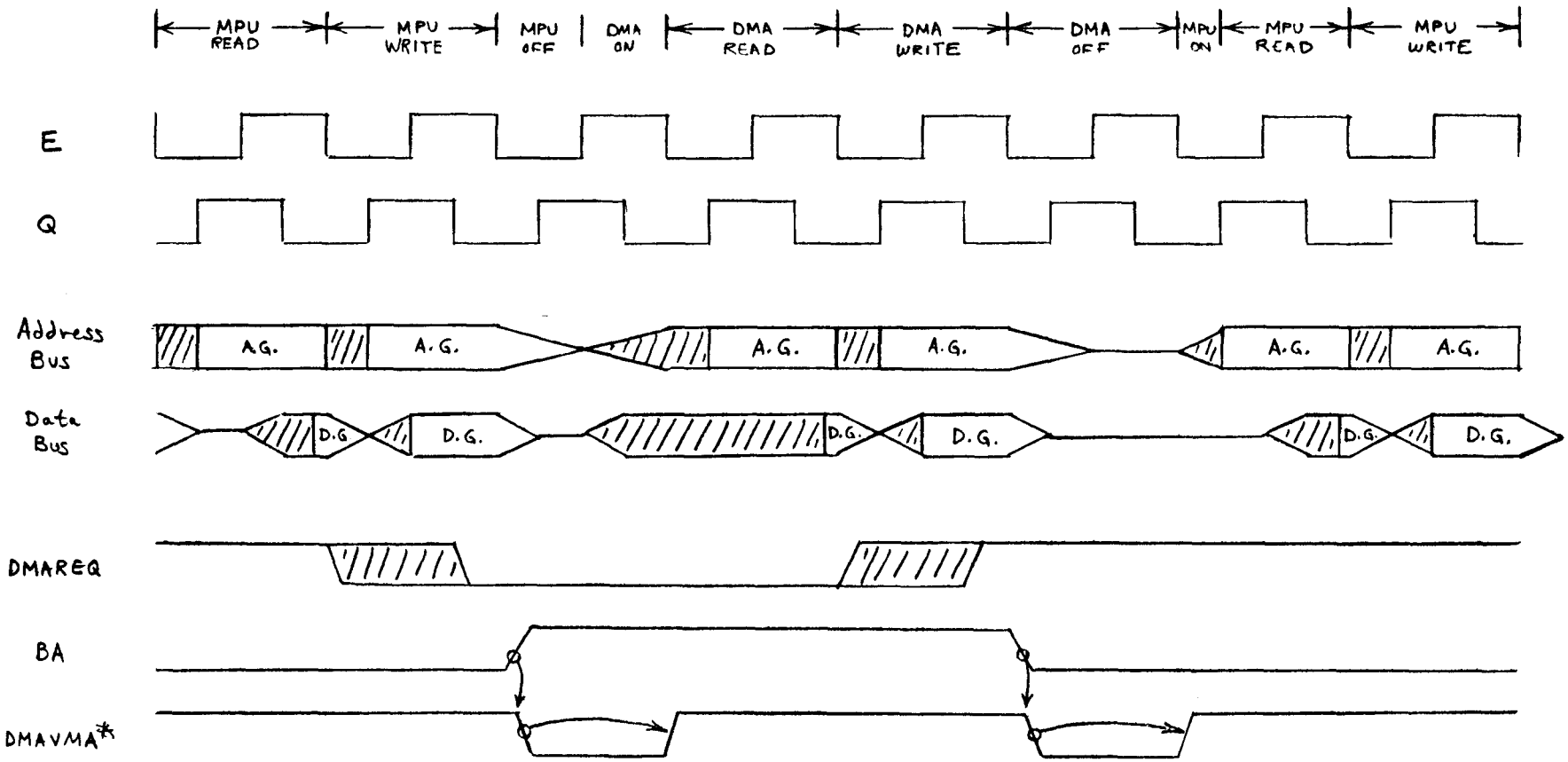
Another form of dynamic memory refresh is to guarantee a software access of the required number of consecutive locations every 2 milliseconds. This can be done by using a real-time clock to cause a FIRQ interrupt, then using 63 or 127 consecutive PAGE 2 pre-bytes followed by an RTI; this sequence is not interruptable (and must not be interruptable, if memory integrity is to be guaranteed).

2.4.3 Slow Devices

Various clock signals from the 6809 MPU allow for increasing memory timing parameters, including both access time and set-up time.

Access-time extension is provided by pulling the MREADY pin LOW in response to the leading-edge of

MOS BUS DMA SEQUENCE

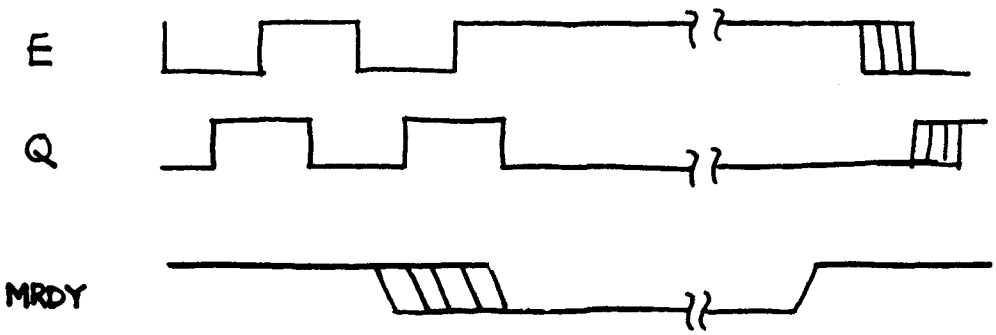


A.G. ≡ Address Good
D.G. ≡ Data Good

* Externally generated

11/29/78 /TFR

Slow Memory



2.4.3 (Continued)

E. The memory-access will be extended, in integral multiples of the high-frequency clock, until some period (0-1 H.F. cycles) after the MR line is returned HIGH. Note that the MPU may only be held not-ready for 10 microseconds.

Furthermore, the Memory Ready function actually changes the system E signal; devices which require a real-time clock must use a different clock source.

Address Set-up time can be easily increased from one-quarter bus cycle to one-half bus cycle by forming a new E' signal, $E' = E \wedge \overline{Q}$. Since this reduces E' up-time to one-quarter bus cycle, Memory Ready can be used to regain the minimum E-time, or increase it, as necessary. It is also possible to use additional timing circuitry to apportion set-up and enable performance as desired.

2.4.4 Multi-Processors

Shared-bus multiprocessor systems must arbitrate between possibly multiple and simultaneous requests for memory access. Exactly one processor must then gain the (temporary) use of the bus; remaining processors are "held off" using the Memory Ready Control signal. Naturally, any processor can only be held

2.4.4 (Continued)

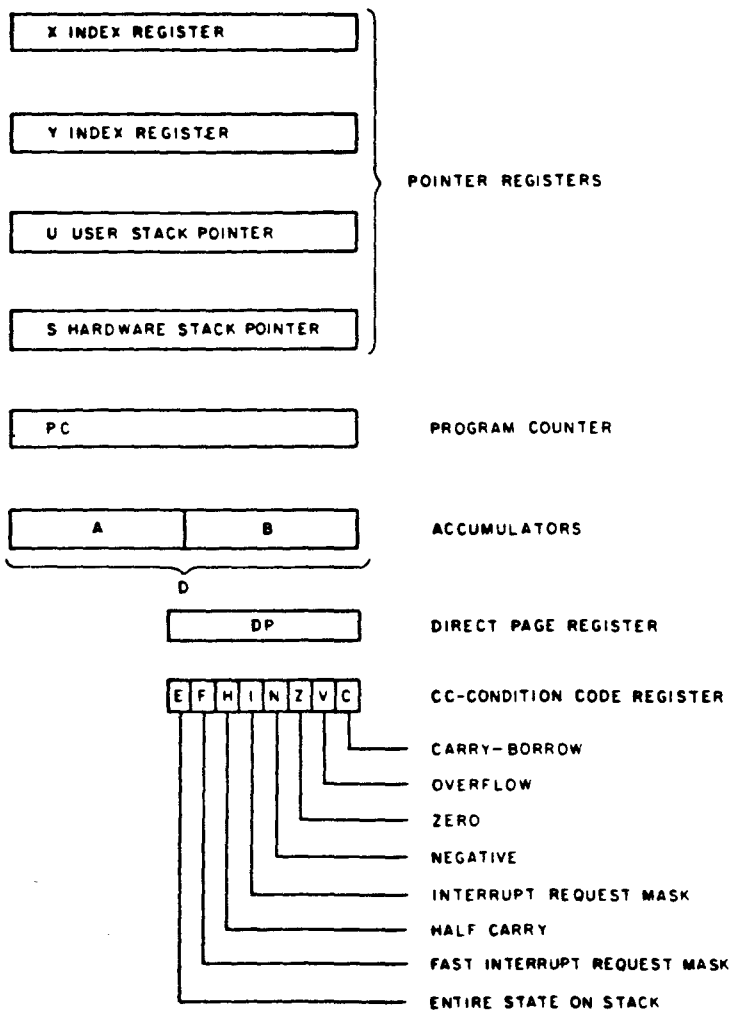
not-ready a maximum of 10 microseconds.

As each memory request is resolved, MREADY for that processor is brought HIGH, and that processor delivers the trailing edge of E which completes the data transfer.

3.0 SOFTWARE ARCHITECTURE

3.1 6809 PROGRAMMING MODEL

The 6809 contains four 8-bit registers and five 16-bit registers which are visible to the programmer:



The Double-Accumulator D consits of the two 8-bit accumulators concatenated A:B. The A-register is the MS byte of the pair while the B-register is the LS byte.

3.1.1 Accumulators (A, B & D)

The A and B registers are general purpose accumulators used for arithmetic calculations and data manipulation. With the exception of ABX, DAA and 16-bit operations, the two accumulators are completely interchangeable. In the catenated form the A-register is the MS byte of the pair thru forming the 16-bit Double Accumulator, or D-register.

3.1.2 Direct Page Register (DP)

The Direct Page register defines the MS byte to be used in the direct mode of addressing; the DP is catenated with the byte following the direct-mode op code to form a 16-bit effective address. The DP will be initialized to $00 by $\overline{\text{RESET}}$ for 6800 compatibility.

3.1.3 Condition Code Register (CC)

The Condition Code register defines the state of the processor flags at any given time. The bits in the CC are:

B7	B6	B5	B4	B3	B2	B1	B0
E	F	H	I	N	Z	V	C

Bit 5 and bits 3-0 are set as the result of instructions that manipulate data; for details, see condition code section for each instruction.

3.1.3.1 Bit 0 (C)

b0 is the Carry flag, and is usually generated by the binary carry from the MSB of the operation (ADC, ADD) -- this is an unsigned overflow. However, C is also used to represent a 'borrow' (a NOT-carry) to and from subtract-like instructions (CMP, NEG, SBC, SUB), and MUL uses C to represent b7 of the result for round-off purposes. Data-movement and logical operations do not affect C, while arithmetic operations set C, if appropriate.

3.1.3.2 Bit 1 (V)

b1 is the overflow flag, and is set by an operation which causes a two's-complement arithmetic overflow. The overflow is, of course, detected in an operation if the carry from the MSB in the binary ALU does not match the carry from the MSB-1. Loads, stores, and logical operations clear V, while arithmetic operations set V if appropriate.

3.1.3.3 Bit 2 (Z)

b2 is the zero flag, and is set if the result of the previous operation was identically zero. Loads, stores, logical and arithmetic operations set Z if appropriate.

3.1.3.4 Bit 3 (N)

b3 is the negative flag, which contains exactly the value of the MSB of the result of the preceeding operation. Thus, a negative two's complement result will leave N set. Loads, stores, logical and arithmetic operations all set N if appropriate. If a two's complement overflow occurs, the sign of the result (and the N-flag) will be incorrect. For this reason two's complement branches use the expression (N⊕V) to obtain an always-valid sign result.

3.1.3.5 Bit 4(I)

b4 is the IRQ mask bit. The processor will not recognize interrupts from the IRQ line if this bit is Set. NMI, FIRQ, IRQ, RESET and SWI all Set I; SWI2 and SWI3 do not affect I.

3.1.3.6 Bit 5 (H)

b5 is the half-carry bit, and is used to indicate a carry from b3 in the ALU as a result of an 8-bit addition only (ADC or ADD). This bit is used by the DAA instruction to perform a (BCD) decimal add adjust operation. The state of the H flag is undefined in all subtract-like instructions to allow for future expansion; software must not depend upon a particular state of the H flag after subtract operations.

3.1.3.7 Bit 6 (F)

b6 is the FIRQ mask bit. The processor will not recognize interrupts from the FIRQ line if this bit is set. NMI, FIRQ, SWI and RESET all Set F; IRQ, SWI2 and SWI3 do not affect F.

3.1.3.8 Bit 7 (E)

b7 is the entire flag, and indicates either the complete machine state (all the registers) or the subset state (PC and CC) is being stacked. E is used by the RTI instruction to determine the extent of the unstacking, thus allowing some interrupt-handling routines which work with both fast and slow interrupts. FIRQ will clear E while IRQ, NMI, SWI, SWI2, and SWI3 will set E before stacking. The E bit associated with the saved registers is in the E flag position in the CC of the stacked state; the E bit in the processor has little meaning.

3.1.3.9 Interrupt Effects on CC

After accepting an IRQ interrupt, the processor will set the E flag, save the entire machine state, then set the I mask bit to mask out the present and further IRQ interrupts. After clearing the original interrupt, the user may reset the I mask bit to allow multiple-level IRQ interrupts. The IRQ interrupt will not affect the F mask bit, thus, in general a FIRQ may interrupt an IRQ handler. The machine state as it was before the interrupt will be recovered by the associated RTI.

3.1.3.9 (Continued)

After accepting a FIRQ interrupt, the processor will clear the E flag, save the subset machine state (return address and CC), then set both the I and F-bits to mask out the present FIRQ and further IRQ and FIRQ interrupts. After clearing the original interrupt, the user may reset the I and F bits to allow multiple-level interrupts. The PC and CC (including the previous state of the mask bits) will be recovered by the associated RTI.

3.1.4 Index Register (X, Y)

The index registers are used in indexed mode addressing. They provide a 16-bit address to be added to an optional offset (of up to 16-bits) for indexed instructions; the result of the addition is the effective address of the instruction. For more details see the section on addressing modes. The X and Y registers are essentially equivalent in usage and support the same instructions. Because automatic pre-increment and post-decrement options are available on indexed-mode operations, these registers may be used to easily implement software stacks, queues, and buffers.

3.1.5 Stack Pointers (U , S)

The stack pointer registers contain addresses that point to the top of a push-down/pop-up stack. Data and machine state can be pushed onto the stack (stored at the next memory address to that "pointed" to by the U or S) or pulled from the stack in a last-in first-out manner. Pushes decrement the stack pointer before the data is stored while pulls increment the stack pointer after the data is recovered; the stack pointers point at the last byte placed on the stack. The S is used by the hardware to automatically store subset or entire machine states during subroutines and interrupts. The User Stack (U) is controlled exclusively by the programmer and can be used to pass arguments to and from subroutines. Both the U and S have the same indexed-mode addressing capabilities as

3.1.5 (Continued)

the X and Y index registers; the stack pointers are enhanced index registers (although the operation as LEA is slightly different on the stack registers). This allows the 6809 to be used efficiently as a stack processor, greatly enhancing its ability to support higher level languages.

3.1.6 Program Counter (PC)

The PC is used by the hardware to point to the next instruction to be executed by the processor. Limited indexed-mode addressing is available on the PC (i.e., auto-increment/decrement is not available). For notational convenience the description of each instruction assumes that the program counter points one location past the last byte of the op code, as it would after decoding the instruction. As additional bytes are used by the instruction the PC always points to the next unused byte.

EXAMPLE: The branch instructions are available in either short or long forms; in general the short form takes a one-byte opcode, while the long form takes two bytes. After decoding the opcode, the PC points at either a one- (short branch) or two-byte (long) immediate value, which is taken into the machine for addition to the PC. If the branch is not taken, the addition never happens and the PC remains pointing to the next instruction. Indexed-mode instructions also have variable length fields.

3.1.7 Stack Programming Techniques

Good programming practice indicates use of space in the hardware stack for temporary storage. The stack pointer is decremented by the amount of storage required (LEAS -TEMPS, S) making space for temporaries from 0,S through TEMPS-1,S. This technique is structured, position-independent, and allows recursion.

Global variables may be considered local to the highest-level routine, and allocated storage there. Unfortunately, access to these same variables requires different offset values depending upon subroutine depth, itself a dynamic parameter which may not be readily available. This problem can be solved by assigning one pointer to mark a location (TFR S,U) on the hardware stack. If this is done immediately prior to allocating global storage, all variables will be available at a constant, positively-offset location from the stack mark. Unstructured multi-level returns are also available; this feature may be useful for aborting the entire package and cleaning up the stack.

Because the hardware stack pointer may be preempted at any time by hardware interrupts, it is an extremely dangerous practice to utilize data referred to by negative offset with respect to the hardware stack pointer (SP).

3.2 ADDRESSING

3.2.1 Register Addressing Notation*

Accumulator	ACCA or ACCB (A or B)
Double Accumulator	ACCA:ACCB or ACCD (D)
Index Register	IX or IY (X or Y)
Stack Register	SP or US (S or U)
Program Counter	PC (PC)
Direct Page Register	DPR (DP)
Condition Code Register	CCR (CC)

3.2.2 Register Addressing Modes

3.2.2.1 Accumulator

3.2.2.2 Double-Accumulator

3.2.2.3 Inherent

* The longer-form notation (i.e., ACCA, ACCB, ACCD, IX, IY, SP, US, PC, DPR, CCR) is used by this document to describe the CPU registers. The short-form notation (i.e., A, B, D, X, Y, S, U, PC, DP, CC) is used by the 6809 Assembler.

3.2.3 Memory Addressing Notation

() = The (8-Bit) data pointed to by the enclosed (16-Bit) address

EA = The Effective Address; a pointer into memory created as a result of an addressing mode.

M = (EA) = the data in the address space ("MEMORY") pointed to by the effective address

MI = Memory Immediate Addressing; the data immediately following the last byte of the op code

dd = 8-Bit Offset (or a relative distance to a label which evaluates to 8-bits)

DDDD = 16-Bit Offset (or a relative distance to a label)

P = Immediate, Direct, Indexed, Extended

Q = Accumulator, Direct, Indexed, Extended

YYYY = Offset such that $-64K \leq YYYY \leq 64K$

ZZ = Any indexable register (IX,IY,SP, or US)

XX = 8-Bit hex value

* = PC at start of present instruction

*' = Start of next instruction

IN = Indexed Addressing only.

= Immediate Addressing Byte(s) Follow(s)

$ = Hex Value Follows

% = Binary Value Follows

< = Before indexing: force one-byte offset form (for known forward reference)

= Before absolute address; force direct addressing (obtain warning if SETDP ≠ MS Byte value)

= Before indexing; force two-byte offset form

> = Before absolute address; force extended addressing.

, = Indexing symbol

[ ] = Indirection

3.2.3 (Continued)

It is understood for convenience of description that the PC points one byte past the last byte of the instruction op code at the beginning of instruction execution.

* The assembler uses brackets "[]" to indicate indirection. This avoids evaluation confusion with parentheses "()" which are allowed in expressions.

3.2.4 Memory Addressing Modes

3.2.4.1 Inherent

Example: MUL

Inherent addressing includes those instructions which have no addressing options.

3.2.4.2 Accumulator

```
Example:  CLRA
          CLRB
```

Accumulator addressing includes those instructions which operate on an accumulator.

3.2.4.3 Immediate EA = PC

```
Example:  LDA #CR
          LDB #7
          LDA #$F0
          LDB #%11110000
          LDX #$8004
```

Immediate addressing refers to the location(s) following the last byte of the op code. This mode is used to hold a value which is known at assembly time and which will not be changed during program execution.

3.2.4.4 Absolute (Immediate Indirect)

```
Example:  LDA  $8004
          LDB  CAT
```

Absolute addressing refers to an exact 16-bit location in the memory address space, and is especially useful for transactions with peripherals (I/O).

3.2.4.4 (Continued)

There are three program-selectable modes of absolute addressing, namely: Direct, Extended, and Extended Indirect. Certain instructions (SWI, SWI2, SWI3), and the interrupts, use an inherent absolute address to function similarly to Extended Indirect mode addressing. These instructions are said to have "Absolute Indirect" addressing.

3.2.4.4.1 Direct EA = DPR:(PC)

LDA <CAT

Direct addressing uses the immediate byte of the instruction as a one-byte pointer into a single 256-byte "page" of memory. (The term "page" refers to one of the 256 possible combinations of the high-order address bits.) The particular page in use is fixed by loading the Direct Page Register with the desired high-order byte (by transferring from or exchanging with another register.) Thus, the effective address consists of a high-order byte (from the Direct Page Register) catenated with a low-order byte (from the instruction).

This mode may allow economies of both program space and excution time as compared to other absolute or indexed modes.

3.2.4.4.2 Extended EA = (PC):(PC+1)

Example: LDA >CAT

Extended addressing uses a 16-bit immediate value (and thus contained in the two bytes following the last byte of the op code) as the exact memory address value.

3.2.4.4.3 Extended Indirect EA = ((PC):(PC+1))

Example: LDA [$F000]

Extended indirect addressing uses a 16-bit immediate value as an absolute address from which to recover the effective address.

3.2.4.4.3 (Continued)

This mode is inherently used by interrupts to vector to the handling routine; and may be used to create vector tables in a customized system which allow the use of standard software packages.

Although Extended Indirect is a logical extension of Extended addressing, this mode is implemented using an encoding of the postbyte under the indexed addressing group.

3.2.4.5 Register

Example: TFR X,Y

Register addressing refers to the selection of various on-board registers.

3.2.4.6 Indexed (Register Indirect)

The 6809 includes extremely powerful indexing capabilities. There are five indexable registers (X,Y,S,U, and PC) with many options (constant-offset, accumulator offset using A,B, or D, auto-increment or -decrement, and indirection). These options are selected by complex coding of the first byte after the op code byte(s) of indexed-mode instructions. Most 6800 indexed-mode instructions will map into an equivalent two bytes on the 6809.

3.2.4.6.1 Constant-Offset Indexed

```
Examples:  LDA     ,X
           LDB     0,Y
           LDX     64000,S
           LDY     -64000,U
           LDA     17,PC
           LDA     THERE,PCR
```

Constant-offset indexing uses an optional two's complement offset contained in either the post byte of the instruction as a bit-field or as an immediate value. This offset may be an absolute quantity, a symbol, or an expression and may range from zero to a 16-bit binary value which may be specified either positive or negative with an absolute value less or equal to 2^{16}. The offset value is temporarily added to the pointer value from the selected register (X,Y,U,S, or PC); the result is the effective address which points into memory.

A number of hardware modes are available to reduce the number of instruction bytes for various options. The majority of 6800 indexed-mode instructions will still need only two bytes on the 6809.

The notation THERE, PCR causes the assembler to compute the relative distance between the location of the symbol THERE elsewhere in the program, and the present value of the program

3.2.4.6.1 (Continued)

counter. The computed value is used as an immediate value in the instruction, indexed from the program-counter. This notation is painlessly position-independent.

Because a 16-bit offset is allowed, the (necessarily absolute) address of the indexable data may be carried as a constant value in the indexing instructions. This would allow the "index register" to be simultaneously used for indexing and counting using LEA.

* With exceptions for 6800 compatibility, the 6809 assembly language uses a comma (,) to indicate a single level of indexed indirection. That is, LDX ,Y should be interpreted as: X←(Y):(Y+1) while LDX Y could be: X←Y. This symbology allows the programmer access to a large number of language-compatible macros, and forces the addressing symbology to be apparent for many different instructions. The instructions PSH, PUL, TFR, and EXG are also exceptions.

3.2.4.6.2 Constant-Offset Indexed Indirect

```
Examples:  LDA  [,X]*
           LDB  [Ø,Y]
           LDX  [64000,S]
           LDY  [-64000,U]
           LDA  [17,PC]
           LDA  [THERE,PCR]
```

* Brackets indicate indirection to the assembler.

3.2.4.6.2 (Continued)

Constant-offset indexed indirect addressing functions in two stages (like all indirects). First an indexed address is formed by temporarily adding the offset-value contained in the addressing byte(s) to the value from the selected pointer register (X,Y,S,U, or PC). Second, this address is used to recover a two-byte absolute pointer which is used as the "effective address."

This mode allows the programmer to use a "table of pointers" data structure, or to do I/O through absolute values stored on the stack.

3.2.4.6.3 Accumulator Indexed

```
Examples:  LDA   A,X
           LDA   B,Y
           LDA   D,U
```

Accumulator-indexed addressing uses an accumulator (A,B, or D) as a two's complement offset which is temporarily added to the value from the selected pointer register (X,Y,S, or U) to form the effective address.

3.2.4.6.4 Accumulator Indexed Indirect

```
Examples:  LDA  [A,X]
           LDA  [B,Y]
           LDA  [D,U]
```

Accumulator-indexed indirect addressing uses an accumulator (A,B,or D) as a two's complement

3.2.4.6.4 (Continued)

offset which is temporarily added to the value from the selected pointer register (X,Y,S, or U). The resulting pointer is then used to recover another pointer from memory (thus, the indirect designation) which is then used as the effective address.

3.2.4.6.5 Auto-Increment

```
Examples:  LDA    ,X+     LDX    ,X++
           LDA    ,Y+     LDX    ,Y++
           LDA    ,S+     LDX    ,U++
           LDA    ,U+     LDX    ,S++
```

Auto-increment addressing uses the value in the selected pointer register (X,Y,S, or U) to address a one-or two-byte value in memory. The register is then incremented by one (single +) or two (two +'s). No offset is permitted.

3.2.4.6.6 Auto-Increment Indirect

```
Examples:  LDA    [,X++]
           LDB    [,Y++]
           LDD    [,S++]
           LDX    [,U++]
```

Auto-increment indirect addressing uses the value in the selected pointer register (X,Y,S, or U) to recover an address value from memory. This value is used as the effective address. The register is then incremented by two (++); the indirected increment by one is illegal. No offset is permitted.

3.2.4.6.7 Auto-Decrement

```
Examples:  LDA    ,-X     LDX    ,--X
           LDA    ,-Y     LDX    ,--Y
           LDA    ,-U     LDX    ,--U
           LDA    ,-S     LDX    ,--S
```

Auto-decrement addressing first decrements the selected pointer register (X,Y,S, or U) by one (-) or two (--) as selected by the user. The resulting value is then used as the effective address. No offset is permitted.

3.2.4.6.8 Auto-Decrement Indirect

```
Examples:  LDA    [,--X]
           LDB    [,--Y]
           LDD    [,--U]
           LDX    [,--S]
```

Auto-decrement indirect addressing first decrements the selected pointer register by two (--). Auto-decrement by one indirect is prohibited in the assembly language. The resulting value is used to recover a pointer value from memory; this value is used as the effective address. No offset is permitted.

3.2.4.7 Relative

```
Example:  BRA    POLE
```

(Short) Relative addressing adds the value of the immediate byte of the instruction (an 8-bit two's complement value) to the value of the program counter to produce an absolute address. This addressing mode is always position-independent.

3.2.4.8 Long Relative

Example: LBRA CAT

Long Relative addressing adds the value of the immediate bytes of the instructions (a 16-bit two's complement value) to the value of the program counter to produce an absolute address. This addressing mode is always position-independent.

3.3 INSTRUCTION SET

3.3.1 Operation Notation

← = is Transferred to
∧ = Boolean AND
V = Boolean OR
⊕ = Boolean EXCLUSIVE-OR
¯ = (overline) = Boolean NOT
: = Concatenation

3.3.2 Register Notation

ACCA = A =	Accumulator A
ACCB = B =	Accumulator B
ACCX =	Either ACCA or ACCB
ACCA:ACCB = D =	Double Accumulator
IX = X =	Index Register X
IY = Y =	Index Register Y
SP = S =	Hardware Stack Pointer
US = U =	User Stack Pointer
DPR = DP =	Direct Page Register
CCR = CC =	Condition Code Register
PC =	Program Counter
R =	A Register before the operation; A,B,D,X,Y,U,S,PC,DP or CC (usually, only a subset of registers is legal, these are specified by "Register Addressing Mode" in the individual instructions)
R' =	A Register <u>after</u> the operation
ALL =	All Registers; i.e.,A,B,D,X,Y,U,S,PC,DP & CC
ZZ =	A Pointer Register; i.e., X,Y,U,S
MSB =	Most-Significant <u>BIT</u>
MS BYTE =	Most-Significant BYTE
LS BYTE -	Least-Significant BYTE
IXH =	MS Byte of Index X
IXL =	LS Byte of Index X

ABX Add ACCB Into IX

SOURCE FORM: ABX

OPERATION: IX' ← IX + ACCB

CONDITION CODES: Not Affected

DESCRIPTION:

Add the 8-bit unsigned value in Accumulator B into the X index register.

ADDRESSING MODE: Inherent

ADC Add With Carry Memory Into Register

SOURCE FORMS: ADCA P; ADCB P

OPERATION: R' ← R + M + C

CONDITION CODES:

H: Set IFF the operation caused a carry from bit 3 in the ALU
N: Set IFF bit 7 of the result is Set.
Z: Set IFF all bits of the result are Clear
V. Set IFF the operation caused an 8-bit two's complement arithmetic overflow.
C: Set IFF the operation caused a carry from bit 7 in the ALU

DESCRIPTION:

Adds the contents of the carry flag and the memory byte into an 8-bit register.

REGISTER ADDRESSING MODE: Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

ADD Add Memory Into Register - 8 Bit

SOURCE FORMS: ADDA P; ADDB P

OPERATION: R' ← R + M

CONDITION CODES:

H: Set IFF the operation caused a carry from bit 3 in the ALU
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Set IFF the operation caused an 8-bit two's complement arithmetic overflow.
C: Set IFF the operation caused a carry from bit 7 in the ALU

DESCRIPTION:

Adds the memory byte into an 8-bit register.

REGISTER ADDRESSING MODE: Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

ADD Add Memory Into Register - 16 Bits

SOURCE FORM: ADDD P

OPERATION: R' ← R + M:M+1

CONDITION CODES:

H: Not Affected
N: Set IFF bit 15 of the result if Set
Z: Set IFF all bits of the result are Clear
V: Set IFF there was a 16-bit two's complement arithmetic overflow
C: Set IFF the operation on the MS Byte caused a carry from bit 7 in the ALU.

DESCRIPTION:

Adds the 16-bit memory value into the 16-bit accumulator.

REGISTER ADDRESSING MODE: Double Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

AND Logical AND Memory Into Register

SOURCE FORMS: ANDA P; ANDB P

OPERATION: R' ← R ∧ M

CONDITION CODES:

H: Not Affected
N: Set IFF bit 7 of result is Set
Z: Set IFF all bits of result are Clear
V: Cleared
C: Not Affected

DESCRIPTION:

Performs the logical "AND" operation between the contents of ACCX and the contents of M and the result is stored in ACCX.

REGISTER ADDRESSING MODE: Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

AND Logical AND Immediate Memory Into CCR

SOURCE FORM: ANDCC #XX

OPERATION: R' ← R ∧ MI

CONDITION CODES: CCR' ← CCR ∧ MI

DESCRIPTION:

Performs a logical "AND" between the CCR and the MI byte and places the result in the CCR.

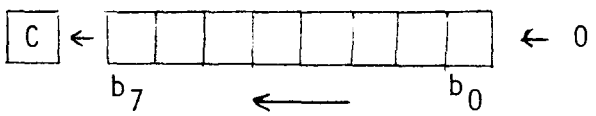
REGISTER ADDRESSING MODES: CCR

MEMORY ADDRESSING MODE: Memory Immediate

ASL Arithmetic Shift Left

SOURCE FORM: ASL Q

OPERATION:



$C' \leftarrow b_7,\ b_7' \ldots b_1' \leftarrow b_6 \ldots b_0,\ b_0' \leftarrow 0$

CONDITION CODES:

H: Undefined
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Loaded with the result of $(b_7 \oplus b_6)$ of the original operand.
C: Loaded with bit 7 of the original operand.

DESCRIPTION:

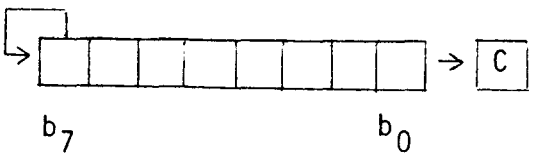
Shifts all bits of the operand one place to the left. Bit 0 is loaded with a zero. Bit 7 of the operand is shifted into the carry flag.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

ASR Arithmetic Shift Right

SOURCE FORM: ASR Q

OPERATION:



$C' \leftarrow b_0,\ b_6' \ldots b_0' \leftarrow b_7 \ldots b_1,\ b_7' \leftarrow b_7$

CONDITION CODES:

H: Undefined
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of result are Clear
V: Not Affected
C: Loaded with bit 0 of the original operand.

DESCRIPTION:

Shifts all bits of the operand right one place. Bit 7 is held constant. Bit 0 is shifted into the carry flag. The 6800/01/02/03/08 processors do affect the V flag.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

BCC Branch on Carry Clear

SOURCE FORMS: BCC dd ; LBCC DDDD

OPERATION: TEMP ← MI
IFF C = 0 then PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Tests the state of the C bit and causes a branch if C is clear.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

When used after a subtract or compare on unsigned binary values, this instruction could be called "branch if the register was higher or the same as the memory operand".

BCS Branch on Carry Set

SOURCE FORMS: BCS dd ; LBCS DDDD

OPERATION: TEMP ← MI
IFF C = 1 then PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Tests the state of the C bit and causes a branch if C is set.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

When used after a subtract or compare on unsigned binary values, this instruction could be called "branch if the register was lower then the memory operand".

BEQ Branch on Equal

SOURCE FORMS: BEQ dd ; LBEQ DDDD

OPERATION: TEMP ← MI
IFF Z = 1 then PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Tests the state of the Z bit and causes a branch if the Z bit is set.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation, this instruction will branch if the compared values - signed or unsigned - were exactly the same.

BGE Branch on Greater than or Equal to Zero

SOURCE FORMS: BGE dd ; LBGE DDDD

OPERATION: TEMP ← MI
IFF [N ⊕ V] = 0 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if N and V are either both set or both clear (i.e., branch if the sign of a valid two's complement result is - or would be - positive).

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation on two's complement values, this instruction will "branch if the register was greater than or equal to the memory operand."

BGT Branch on Greater

SOURCE FORMS: BGT dd; LBGT DDDD

OPERATION: TEMP ← MI
IFF Z v [N ⊕ V] = 0 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if (N and V are either both set or both clear) and Z is clear. In other words, branch if the sign of a valid two's complement result is- or would be - positiv and non-zero.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation on two's complement values, this instruction will "branch if the register was greater than the memory operand".

BHI Branch if Higher

SOURCE FORMS: BHI dd; LBHI DDDD

OPERATION: TEMP ← MI
IFF [C v Z] = 0 then PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Causes a branch if the previous operation caused neither a carry nor a zero result.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation on unsigned binary values this instruction will "branch if the register was higher than the memory operand." Not useful, in general after INC/DEC, LD/ST, TST/CLR/COM.

BHS Branch if Higher or Same

SOURCE FORM: BHS dd; LBHS DDDD

OPERATION: TEMP ← MI
IFF C = 0 then PC' ← PC + MI

CONDITION CODES: Not Affected

DESCRIPTION:

Tests the state of the C-bit and causes a branch if C is clear.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

When used after a subtract or compare on unsigned binary values, this instruction will "branch if register was higher than or same as the memory operand." This is a duplicate assembly-language mnemonic for the single machine instruction BCC. Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

BIT Bit Test

SOURCE FORM: BIT P

OPERATION: TEMP ← R ∧ M

CONDITION CODES:

H: Not Affected
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Cleared
C: Not Affected

DESCRIPTION:

Performs the logical "AND" of the contents of ACCX and the contents of M and modifies condition codes accordingly. The contents of ACCX or M are not affected.

REGISTER ADDRESSING MODE: Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

BLE Branch on Less than or Equal to Zero

SOURCE FORM: BLE dd; LBLE DDDD

OPERATION: TEMP ← MI
IFF Z v [N ⊕ V] = 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if the "Exclusive OR" of the N and V bits is 1 or if Z = 1. That is, branch if the sign of a valid two's complement result is - or would be - negative.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation on two's complement values, this instruction will "branch if the register was less than or equal to the memory operand".

BLO Branch on Lower

SOURCE FORM: BLO dd; LBLO DDDD

OPERATION: TEMP ← MI
IFF C = 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Tests the state of the C bit and causes a branch if C is Set.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

When used after a subtract or compare on unsigned binary values, this instruction will "branch if the register was lower" than the memory operand. Note that this is a duplicate assembly-language mnemonic for the single machine instruction BCS. Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

BLS Branch on Lower or Same

SOURCE FORM: BLS dd; LBLS DDDD

OPERATION: TEMP ← MI
IFF (C v Z) = 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if the previous operation caused either a carry or a zero result.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation on unsigned binary values, this instruction will "branch if the register was lower than or the same as the memory operand." Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

BLT Branch on Less than Zero

SOURCE FORMS: BLT dd; LBLT DDDD

OPERATION: TEMP ← MI
IFF [N ⊕ V]= 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if either, but not both, of the N or V bits is '1.' That is, branch if the sign of a valid two's complement result is - or would - negative.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation on two's complement binary values, this instruction will "branch if the register was less than the memory operand."

BMI Branch on Minus

SOURCE FORM: BMI dd; LBMI DDDD

OPERATION: TEMP ← MI
IFF N = 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Tests the state of the N bit and causes a branch if N is set. That is, branch if the sign of the two's complement result is negative.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after an operation on two's complement binary values, this instruction will "branch if the (possibly invalid) result is minus."

BNE Branch Not Equal

SOURCE FORMS: BNE dd; LBNE DDDD

OPERATION: TEMP ← MI
IFF Z = 0 then PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Tests the state of the Z bit and causes a branch if the Z bit is clear.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after a subtract or compare operation on any binary values, this instruction will "branch if the register is (or would be) not equal to the memory operand."

BPL Branch on Plus

SOURCE FORM: BPL dd; LBPL DDDD

OPERATION: TEMP ← MI
IFF N = 0 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Tests the state of the N bit and causes a branch if N is clear. That is, branch if the sign of the two's complement result is positive.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after an operation on two's complement binary values, this instruction will "branch if the possibly invalid result is positive."

BRA Branch Always

SOURCE FORMS: BRA dd; LBRA DDDD

OPERATION: TEMP ← MI
PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Causes an unconditional branch.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

BRN Branch Never

SOURCE FORM: BRN dd; LBRN DDDD

OPERATION: TEMP ← MI

CONDITION CODES: Not affected

DESCRIPTION:

Does not cause a branch. This instruction is essentially a NO-OP, but has a bit pattern logically related to BRA.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

BSR Branch to Subroutine

SOURCE FORM: BSR dd; LBSR DDDD

OPERATION: TEMP ← MI
SP' ← SP-1, (SP) ← PCL
SP' ← SP-1, (SP) ← PCH
PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

The program counter is pushed onto the stack. The program counter is then loaded with the sum of the program counter and the memory immediate offset.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

BVC Branch on Overflow Clear

SOURCE FORM: BVC dd; LBVC DDDD

OPERATION: TEMP ← MI
IFF V = 0 then PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Tests the state of the V bit and causes a branch if the V bit is clear. That is, branch if the two's complement result was valid.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after an operation on two's complement binary values, this instruciton will "branch if there was no overflow".

BVS Branch on Overflow Set

SOURCE FORM: BVS dd; LBVS DDDD

OPERATION: TEMP ← MI
IFF V = 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Tests the state of the V bit and causes a branch if the V bit is set. That is, branch if the two's complement result was invalid.

MEMORY ADDRESSING MODE: Memory Immediate

EFFECTIVE ADDRESSING MODES: Relative
Long Relative

COMMENTS:

Used after an operation on two's complement binary values, this instruction will "branch if there was an overflow." This instruction is also used after ASL or LSL to detect binary floating-point normalization.

CLR Clear

SOURCE FORM: CLR Q

OPERATION: TEMP ← M
M ← 00_{16}

CONDITION CODES:

H: Not affected
N: Cleared
Z: Set
V: Cleared
C: Cleared

DESCRIPTION:

ACCX or M is loaded with 00000000. The C-flag is cleared for 6800 compatibility.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

CMP Compare Memory from a Register - 8 Bits

SOURCE FORM: CMPA P; CMPB P

OPERATION: TEMP ← R - M [i.e., TEMP ← R + $\overline{M}$ + 1]

CONDITION CODES:

H: Undefined
N: Set IFF bit 7 of the result is Set.
Z: Set IFF all bits of the result are Clear.
V: Set IFF the operation caused an 8-bit two's complement overflow
C: Set IFF the subtraction did not cause a carry from bit 7 in the ALU

DESCRIPTION:

Compares the contents of M from the contents of the specified register and sets appropriate condition codes. Neither M nor R is modified. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING: Accumulator

MEMORY ADDRESSING: Immediate
Direct
Indexed
Extended

FLAG RESULTS:

(N⊕V) = 1 R .LT. M (2's comp)
C = 1 R .LO. M (unsigned)
Z = 1 R .EQ. M

CMP Compare Memory From a Register - 16 Bits

SOURCE FORMS: CMPD P; CMPX P, CMPY P; CMPU P; CMPS P

OPERATION: TEMP ← R - M:M+1 [i.e., TEMP ← R + $\overline{M:M+1}$ +1]

CONDITION CODES:

H: Unaffected
N: Set IFF bit 15 of the result is Set
Z: Set IFF all bits of the result are Clear.
V: Set IFF the operation caused a 16-bit two's complement overflow
C: Set IFF the operation on the MS byte did not cause a carry from bit 7 in the ALU

DESCRIPTION:

Compares the 16-bit contents of M:M+1 from the contents of the specified register and sets appropriate condition codes. Neither R nor M:M+1 is modified. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING: Double Accumulator
Pointer (X, Y, S, or U)

MEMORY ADDRESSING: Immediate
Direct
Indexed
Extended

FLAG RESULTS:

(N⊕V) = 1 R .LT. M (2's comp)
C = 1 R .LO. M (unsigned)
Z = 1 R .EQ. M

COM Complement

SOURCE FORM: COM Q

OPERATION: $M' \leftarrow 0 + \overline{M}$

CONDITION CODES:

H: Not affected
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Cleared
C: Set

DESCRIPTION:

Replaces the contents of M or ACCX with its one's complement (also called the logical complement). The carry flag is set for 6800 compatibility.

MEMORY ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

COMMENTS:

When operating on unsigned values, only BEQ and BNE branches can be expected to behave properly. When operating on two's complement values, all signed branches are available.

CWAI Clear and Wait for Interrupt

SOURCE FORM: CWAI #$XX

E	F	H	I	N	Z	V	C

OPERATION: CCR ← CCR ∧ MI (Possibly clear masks)
Set E (entire state saved)
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
SP' ← SP - 1, (SP) ← DPR
SP' ← SP - 1, (SP) ← ACCB
SP' ← SP - 1, (SP) ← ACCA
SP' ← SP - 1, (SP) ← CCR

FF = enable neither
EF = enable IRQ
BF = enable FIRQ
AF = enable both

CONDITION CODES: Possibly Cleared by the immediate byte.

DESCRIPTION:

The CWAI instruction ANDs an immediate byte with the condition code register which may clear interrupt mask bit(s), stacks the entire machine state on the hardware stack then looks for an interrupt. When a (non-masked) interrupt occurs, no further machine state will be saved before vectoring to the interrupt handling routine. This instruction replaced the 6800's CLI WAI sequence , but does not tri-state the buses.

ADDRESSING MODE: Memory Immediate

COMMENTS:

An FIRQ interrupt may enter its interrupt handler with its entire machine state saved. The RTI will automatically return the entire machine state after testing the E bit of the recovered CCR.

DA Decimal Addition Adjust

SOURCE FORM: DAA

OPERATION: ACCA' ← ACCA + CF(MSN):CF(LSN)
where CF is a Correction Factor, as follows:
The C.F. for each nybble (BCD digit) is determined separately, and is either 6 or 0.

Least Significant Nybble

CF(LSN) = 6 IFF 1) H = 1
or 2) LSN > 9

Most Significant Nybble

CF(MSN) = 6 IFF 1) C = 1
or 2) MSN > 9
or 3) MSN > 8 and LSN > 9

CONDITION CODES:

H: Not affected

N: Set IFF MSB of result is Set

Z: Set IFF all bits of the result are Clear

V: Not defined.

C: Set if the operation caused a carry from bit 7 in the ALU, or if the carry flag was Set before the operation.

DESCRIPTION:

The sequence of a single-byte add instruction on ACCA (either ADDA or ADCA) and a following DAA instruction results in a BCD addition with appropriate carry flag. Both values to be added must be in proper BCD form (each nybble such that: $0 \leq$ nybble ≤ 9). Multiple-precision additions must add the carry generated by this DA into the next higher digit during the add operation immediately prior to the next DA.

ADDRESSING MODE: ACCA

DEC Decrement

SOURCE FORM: DEC Q

OPERATION: $M' \leftarrow M - 1$ [i.e., $M' \leftarrow M + FF_{16}$]

CONDITION CODES:

H: Not affected
N: Set IFF bit 7 of result is Set
Z: Set IFF all bits of result are Clear
V: Set IFF the original operand was 10000000
C: Not affected

DESCRIPTION:

Subtract one from the operand. The carry flag is not affected, thus allowing DEC to be a loop-counter in multiple-precision computations.

MEMORY ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

COMMENTS:

When operating on unsigned values only BEQ and BNE branches can be expected to behave consistently. When operating on two's complement values, all signed branches are available.

EOR Exclusive OR

SOURCE FORMS: EORA P; EORB P

OPERATION: R' ← R ⊕ M

CONDITION CODES:

H: Not affected
N: Set IFF bit 7 of result is Set
Z: Set IFF all bits of result are Clear
V: Cleared
C: Not affected

DESCRIPTION:

The contents of memory is exclusive - ORed into an 8-bit register.

REGISTER ADDRESSING MODES: Accumulator

MEMORY ADDRESSING MODES: Direct
Extended
Immediate
Indexed

EXG Exchange Registers

SOURCE FORM: EXG R1, R2

OPERATION: R1 ↔ R2

CONDITION CODES: Not affected (unless one of the registers is CCR)

DESCRIPTION:

Bits 3-0 of the immediate byte of the instruction define one register, while bits 7-4 define the other, as follows:

0000 = A:B	1000 = A
0001 = X	1001 = B
0010 = Y	1010 = CCR
0011 = US	1011 = DPR
0100 = SP	1100 = Undefined
0101 = PC	1101 = Undefined
0110 = Undefined	1110 = Undefined
0111 = Undefined	1111 = Undefined

Registers may only be exchanged with registers of like size; i.e., 8-bit with 8-bit, or 16 with 16.

ADDRESSING MODES: Inherent

INC Increment

SOURCE FORM: INC Q

OPERATION: M' ← M + 1

CONDITION CODE:

H: Not affected
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Set IFF the original operand was 01111111.
C: Not affected

DESCRIPTION:

Add one to the operand. The carry flag is not affected, thus allowing INC to be used as a loop-counter in multiple-precision computations.

MEMORY ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

COMMENTS:

When operating on unsigned values, only the BEQ and BNE branches can be expected to behave consistently. When operating on two's complement values, all signed branches are correctly available.

JMP Jump to Effective Address

SOURCE FORM: JMP

OPERATION: PC' ← EA

CONDITION CODES: Not affected

DESCRIPTION:

Program control is transferred to the location equivalent to the effective address.

ADDRESSING MODES: Direct
Indexed
Extended

JSR Jump to Subroutine at Effective Address

SOURCE FORM: JSR

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

Condition Codes Not affected

DESCRIPTION:

Program control is transferred to the Effective Address after storing the return address on the hardware stack.

ADDRESSING MODES: Direct
Indexed
Extended

LD Load Register from Memory - 8 Bit

SOURCE FORMS: LDA P; LDB P

OPERATION: R' ← M

CONDITION CODES:

H: Not affected
N: Set IFF bit 7 of loaded data is Set
Z: Set IFF all bits of loaded data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Load the contents of the addressed memory into the register.

REGISTER ADDRESSING MODE: Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

LD Load Register from Memory - 16 Bit

SOURCE FORM: LDD P; LDX P; LDY P; LDS P; LDU P

OPEATION: R' ← M:M+1

CONDITION CODES:

H: Not affected
N: Set IFF bit 15 of loaded data is Set
Z: Set IFF all bits of loaded data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Load the contents of the addressed memory (two consecutive memory locations) into the 16-bit register.

REGISTER ADDRESSING MODES: Double Accumulator
Pointer (X, Y, S, or U)

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

LEA Load Effective Address

SOURCE FORM: LEAX, LEAY, LEAS, LEAU

OPERATION: R' ← EA

CONDITION CODES:

H: Not affected
N: Not affected
Z: LEAX, LEAY: Set IFF all bits of the result are Clear.
LEAS, LEAU: Not affected
V: Not affected
C: Not affected

DESCRIPTION:

Form the effective address to data using the memory addressing mode. Load that address, not the data itself, into the pointer register.

LEAX and LEAY affect Z to allow use as counters and for 6800 INX/DEX compatibility. LEAU and LEAS do not affect Z to allow for cleaning up the stack while returning Z as a parameter to a calling routine, and for 6800 INS/DES compatibility.

REGISTER ADDRESSING MODE: Pointer (X, Y, S, or U)

MEMORY ADDRESSING MODE: Indexed

LSL Logical Shift Left

SOURCE FORM: LSL Q

OPERATION:

$C' \leftarrow b_7,\ b_7' \ldots b_1' \leftarrow b_6 \ldots b_0,\ b_0' \leftarrow 0$

CONDITION CODES:

H: Undefined
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Loaded with the result of $(b_7 \oplus b_6)$ of the original operand.
C: Loaded with bit 7 of the original operand.

DESCRIPTION:

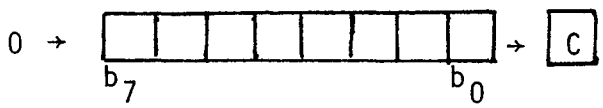
Shifts all bits of ACCX or M one place to the left. Bit 0 is loaded with a zero. Bit 7 of ACCX or M is shifted into the carry flag. This is a duplicate assembly-language mnemonic for the single machine instruction ASL.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

LSR Logical Shift Right

SOURCE FORM: LSR Q

OPERATION:



$C' \leftarrow b_0,\ b_0' \ldots b_6' \leftarrow b_1 \ldots b_7,\ b_7' \leftarrow 0$

CONDITION CODES:

H: Not affected
N: Cleared
Z: Set IFF all bits of the result are Clear
V: Not affected
C: Loaded with bit 0 of the original operand

DESCRIPTION:

Performs a logical shift right on the operand. Shifts a zero into bit 7 and bit 0 into the carry flag. The 6800 processor also affects the V flag.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

MUL Multiply Accumulators

SOURCE FORM: MUL

OPERATION ACCA':ACCB' ← ACCA x ACCB

CONDITION CODES:

H: Not affected
N: Not affected
Z: Set IFF all bits of the result are Clear
V: Not affected
C: Set IFF ACCB bit 7 of result is Set.

DESCRIPTION:

Multiply the unsigned binary numbers in the accumulators and place the result in both accumulators. Unsigned multiply allows multiple - precision operations. The Carry flag allows rounding the MS byte through the sequence: MUL, ADCA #0.

ADDRESSING MODES: Inherent

NEG Negate

SOURCE FORM: NEG Q

OPERATION: $M' \leftarrow 0 - M$ i.e., $M' \leftarrow \overline{M} + 1$

CONDITION CODES:

H: Undefined
N: Set IFF bit 7 of result is Set
Z: Set IFF all bits of result are Clear
V: Set IFF the original operand was 10000000
C: Set IFF the operation did not cause a carry from bit 7 in the ALU.

DESCRIPTION:

Replaces the operand with its two's complement. The C-flag represents a borrow and is set inverse to the resulting binary carry. Note that 80_{16} is replaced by itself and only in this case is V Set. The value 00_{16} is also replaced by itself, and only in this case is C cleared.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

FLAG RESULTS:

(N⊕V) = 1 if Ø .LT. M (2's comp)
C = 1 if Ø .LO. M (unsigned)
Z = 1 if Ø .EQ. M

NOP No Operation

SOURCE FORM: NOP

CONDITION CODES: Not affected

DESCRIPTION:

This is a single-byte instruction that causes only the program counter to be incremented. No other registers or memory contents are affected.

ADDRESSING MODES: Inherent

OR Inclusive OR Memory into Register

SOURCE FORMS: ORA P; ORB P

OPERATION: R' ← R v M

CONDITION CODES:

- H: Not affected
- N: Set IFF high order bit of result Set
- Z: Set IFF all bits of result are Clear
- V: Cleared
- C: Not affected

DESCRIPTION:

Performs an "Inclusive OR" operation between the contents of ACCX and the contents of M and the result is stored in ACCX.

REGISTER ADDRESS MODE: Accumulator

MEMORY ADDRESS MODES: Immediate
Direct
Indexed
Extended

OR Inclusive OR Memory-Immediate into CCR

SOURCE FORM: ORCC #XX

OPERATION: R ← R v MI

CONDITION CODES: CCR' ← CCR v MI

DESCRIPTION:

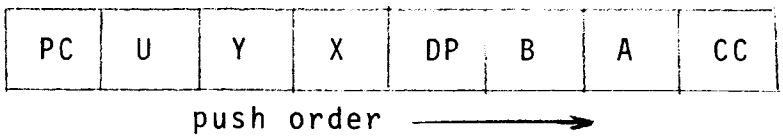
Performs an "Inclusive OR" operation between the contents of CCR and the contents of MI, and the result is placed in CCR. This instruction may be used to Set interrrupt masks (disable interrupts) or any other flag(s).

REGISTER ADDRESSING MODE: CCR

MEMORY ADDRESSING MODE: Memory Immediate

PSHS Push Registers on the Hardware Stack

SOURCE FORM: PSHS register list
PSHS #Label



OPERATION:

IFF B7 of MI set, then: SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
IFF B6 of MI set, then; SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
IFF B5 of MI set, then: SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
IFF B4 of MI set, then: SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
IFF B3 of MI set, then: SP' ← SP - 1, (SP) ← DPR
IFF B2 of MI set, then: SP' ← SP - 1, (SP) ← ACCB
IFF B1 of MI set, then: SP' ← SP - 1, (SP) ← ACCA
IFF B0 of MI set, then: SP' ← SP - 1, (SP) ← CCR

CONDITION CODES: Not affected

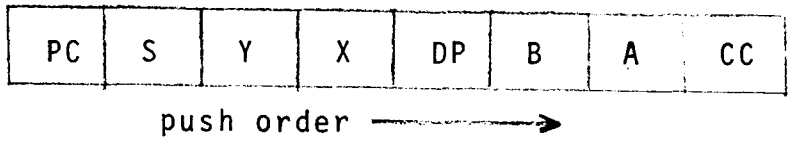
DESCRIPTION:

Any, all, any subset, or none of the MPU registers are pushed onto the hardware stack, (excepting only the hardware stack pointer itself).

MEMORY ADDRESSING MODE: Memory Immediate

PSHU Push Registers on the User Stack

SOURCE FORM: PSHU register list
PSHU #LABEL



OPERATION:

IFF B7 of MI set, then: US' ← US - 1, (US) ← PCL
US' ← US - 1, (US) ← PCH
IFF B6 of MI set, then: US' ← US - 1, (US) ← SPL
US' ← US - 1, (US) ← SPH
IFF B5 of MI set, then: US' ← US - 1, (US) ← IYL
US' ← US - 1, (US) ← IYH
IFF B4 of MI set, then: US' ← US - 1, (US) ← IXL
US' ← US - 1, (US) ← IXH
IFF B3 of MI set, then: US' ← US - 1, (US) ← DPR
IFF B2 of MI set, then: US' ← US - 1, (US) ← ACCB
IFF B1 of MI set, then: US' ← US - 1, (US) ← ACCA
IFF B0 of MI set, then: US' ← US - 1, (US) ← CCR

CONDITION CODES: Not affected

DESCRIPTION:

Any, all, any subset, or none of the MPU registers are pushed onto the user stack (excepting only the user stack pointer itself).

MEMORY ADDRESSING MODE: Memory Immediate

PULS Pull Registers from the Hardware Stack

SOURCE FORM: PULS register list
PULS #LABEL

pull order

OPERATION:

IFF B0 of MI set, then: CCR' ← (SP), SP' ← SP + 1
IFF B1 of MI set, then: ACCA' ← (SP), SP' ← SP + 1
IFF B2 of MI set, then: ACCB' ← (SP), SP' ← SP + 1
IFF B3 of MI set, then: DPR' ← (SP), SP' ← SP + 1
IFF B4 of Mi set, then: IXH' ← (SP), SP' ← SP + 1
IXL' ← (SP), SP' ← SP + 1
IFF B5 of MI set, then: IYH' ← (SP), SP' ← SP + 1
IYL' ← (SP), SP' ← SP + 1
IFF B6 of MI set, then: USH' ← (SP), SP' ← SP + 1
USL' ← (SP), SP' ← SP + 1
IFF B7 of MI set, then: PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

CONDITION CODES:

May be pulled from stack, otherwise unaffected.

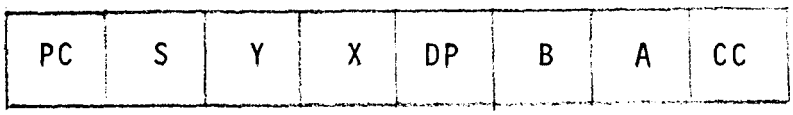
DESCRIPTION:

Any, all, any subset, or none of the MPU registers are pulled from the hardware stack, (excepting only the hardware stack pointer itself). A single register may be "PULLED" with condition-flags set by loading auto-increment from stack (EX: LDA, S+).

MEMORY ADDRESSING MODE: Memory Immediate

PULU Pull Registers from the User Stack

SOURCE FORM: PULU register list
PULU #LABEL



OPERATION:

IFF B0 of MI set, then: CCR' ← (US), US' ← US + 1
IFF B1 of MI set, then: ACCA' ← (US), US' ← US + 1
IFF B2 of MI set, then: ACCB' ← (US), US' ← US + 1
IFF B3 of MI set, then: DPR' ← (US), US' ← US + 1
IFF B4 of MI set, then: IXH' ← (US), US' ← US + 1
IXL' ← (US), US' ← US + 1
IFF B5 of MI set, then: IYH' ← (US), US' ← US + 1
IYL' ← (US), US' ← US + 1
IFF B6 of MI set, then: SPH' ← (US), US' ← US + 1
SPL' ← (US), US' ← US + 1
IFF B7 of MI set, then: PCH' ← (US), US' ← US + 1
PCL' ← (US), US' ← US + 1

CONDITION CODES:

May be pulled from stack, otherwise unaffected.

DESCRIPTION:

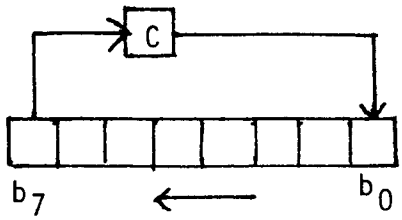
Any all, any subset, or none of the MPU registers are pulled from the user stack (excepting only the user stack pointer itself). A single register may be "PULLED" with condition-flags set by doing an auto-increment load from the stack (EX: LDX, U++).

MEMORY ADDRESSING MODE: Memory Immediate

ROL Rotate Left

SOURCE FORM: ROL Q

OPERATION:



$C' \leftarrow b_7,\ b_7' \ldots b_1' \leftarrow b_6 \ldots b_0,\ b_0' \leftarrow C$

CONDITION CODES:

- H: Not affected
- N: Set IFF bit 7 of the result is Set
- Z: Set IFF all bits of the result are Clear
- V: Loaded with the result of $(b_7 \oplus b_6)$ of the orginal operand.
- C: Loaded with bit 7 of the original operand

DESCRIPTION:

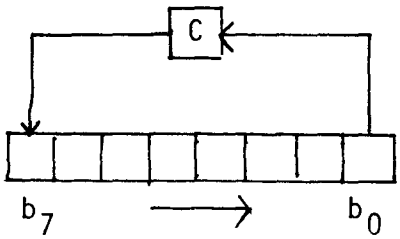
Rotate all bits of the operand one place left through the carry flag; this is a nine-bit rotation.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

ROR Rotate Right

SOURCE FORM: ROR Q

OPERATION:



$C' \leftarrow b_0,\ b_6' \ldots b_0' \leftarrow b_7 \ldots b_1,\ b_7' \leftarrow C$

CONDITION CODES:

- H: Not affected
- N: Set IFF bit 7 of result is Set
- Z: Set IFF all bits of result are Clear
- V: Not affected
- C: Loaded with bit 0 of the previous operand

DESCRIPTION:

Rotates all bits of the operand right one place through the carry flag; this is a nine-bit rotation. The 6800 processor also affects the V flag.

ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

RTI Return from Interrupt

SOURCE FORM: RTI

OPERATION: CCR' ← (SP), SP' ← SP + 1

IFF CCR bit E is SET then: ACCA' ← (SP), SP' ← SP + 1
ACCB' ← (SP), SP' ← SP + 1
DPR' ← (SP), SP' ← SP + 1
IXH' ← (SP), SP' ← SP + 1
IXL' ← (SP), SP' ← SP + 1
IYH' ← (SP), SP' ← SP + 1
IYL' ← (SP), SP' ← SP + 1
USH' ← (SP), SP' ← SP + 1
USL' ← (SP), SP' ← SP + 1
PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

IFF CCR bit E is CLEAR then:
PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

CONDITION CODES: Recovered from stack

DESCRIPTION:

The saved machine state is recovered from the hardware stack and control is returned to the interrupted program. If the recovered E bit is CLEAR, it indicates that only a subset of the machine state was saved (return address and condition codes) and only that subset is to be recovered.

ADDRESSING MODE: Inherent

RTS Return from Subroutine

SOURCE FORM: RTS

OPERATION: PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

CONDITION CODES: Not affected

DESCRIPTION:

Program control is returned from the subroutine to the calling program. The return address is pulled from the stack.

ADDRESSING MODE: Inherent

SBC Subtract with Borrow

SOURCE FORMS: SBCA P; SBCB P

OPERATION: $R' \leftarrow R - M - C$ [i.e., $R' \leftarrow R + \overline{M} + \overline{C}$]

CONDITION CODES:

- H: Undefined
- N: Set IFF bit 7 of the result if Set
- Z: Set IFF all bits of the result are Clear
- V: Set IFF the operation causes an 8-bit two's complement overflow
- C: Set IFF the operation did not cause a carry from bit 7 in the ALU

DESCRIPTION:

Subtracts the contents of M and the borrow (in the carry flag) from the contents of an 8-bit register, and places the result in that register. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING MODE: Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

SEX Sign Extended

SOURCE FORM: SEX

OPERATION: If bit 7 of ACCB is set then ACCA' ← FF_{16}
else ACCA' ← 00_{16}

CONDITION CODES:

H: Not affected
N: Set IFF the MSB of the result is Set
Z: Set IFF all bits of ACCD are Clear
V: Not affected
C: Not affected

DESCRIPTION:

This instruction transforms a two's complement eight-bit value in ACCB into a two's complement sixteen-bit value in the double accumulator.

ADDRESSING: Inherent

ST Store Register Into Memory - 8 Bits

SOURCE FORM: STA P; STB P

OPERATION: M' ← R

CONDITION CODES:

H: Not affected
N: Set IFF bit 7 of stored data was Set
Z: Set IFF all bits of stored data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Writes the contents of an MPU register into a memory location.

REGISTER ADDRESSING MODES: Accumulator

MEMORY ADDRESSING MODES:

Direct
Indexed
Extended

ST Store Register Into Memory - 16 Bit

SOURCE FORM: STD P; STX P; STY P; STS P; STU P

OPERATION: M':M+1' ← R

CONDITION CODES:

H: Not affected
N: Set IFF bit 15 of stored data was Set
Z: Set IFF all bits of stored data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Write the 16 bit register into consecutive memory locations

REGISTER ADDRESSING MODES: Double Accumulator
Pointer (X, Y, S, or U)

MEMORY ADDRESSING MODES:

Direct
Indexed
Extended

SUB Subtract Memory from Register - 8 Bit

SOURCE FORMS: SUBA P; SUBB P

OPERATION: R' ← R - M [i.e., R' ← R + $\overline{M}$ + 1]

CONDITION CODES:

H: Undefined
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Set IFF the operation caused an 8-bit two's complement overflow
C: Set IFF the operation did not cause a carry from bit 7 in the ALU

DESCRIPTION:

Subtracts the value in M from the contents of an 8-bit register. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING MODE: Accumulator

FLAG RESULTS:

(N ⊕ V) = 1 if R .LT. M (2's comp)
C = 1 if R .LO. M (unsigned)
Z = 1 if R .EQ. M

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

SUB Subtract Memory from Register - 16 Bit

SOURCE FORM: SUBD P

OPERATION: R' ← R - M:M+1 [i.e., R' ← R + $\overline{M:M+1}$ + 1]

CONDITION CODES:

H: Unaffected
N: Set IFF bit 15 of result is Set
Z: Set IFF all bits of result are Clear
V: Set IFF the operation caused a 16-bit two's complement overflow.
C: Set IFF the operation on the MS byte did not cause a carry from bit 7 in the ALU

DESCRIPTION:

This information subtracts the value in M:M+1 from the 16-bit accumulator. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING MODE: Double Accumulator

MEMORY ADDRESSING MODES: Immediate
Direct
Indexed
Extended

SUBTRACT SETS:

(N⊕V)= 1 if R .LT. M (2's comp)
C = 1 if R .LO. M (unsigned)
Z = 1 if R .EQ. M

SWI Software Interrupt

SOURCE FORM: SWI

OPERATION: Set E (entire state will be saved)
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
SP' ← SP - 1, (SP) ← DPR
SP' ← SP - 1, (SP) ← ACCB
SP' ← SP - 1, (SP) ← ACCA
SP' ← SP - 1, (SP) ← CCR
Set I, F (mask interrupts)
PC' ← (FFFA):(FFFB)

CONDITION CODES: Not affected

DESCRIPTION:

All of the MPU registers are pushed onto the hardware stack (excepting only the hardware stack pointer itself), and control is transferred through the SWI vector.

ADDRESSING MODE: Absolute Indirect

SWI2 Software Interrupt 2

SOURCE FORM: SWI2

OPERATION: Set E (entire state saved)
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
SP' ← SP - 1, (SP) ← DPR
SP' ← SP - 1, (SP) ← ACCB
SP' ← SP - 1, (SP) ← ACCA
SP' ← SP - 1, (SP) ← CCR
PC' ← (FFF4):(FFF5)

CONDITION CODES: Not affected

DESCRIPTION:

All of the MPU registers are pushed onto the hardware stack (excepting only the hardware stack pointer itself), and control is transferred through the SWI2 vector. SWI2 is available to the end user and must not be used in packaged software.

ADDRESSING MODE: Absolute Indirect

SWI3 Software Interrupt

SOURCE FORM: SWI3

OPERATION: Set E (entire state will be saved)
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
SP' ← SP - 1, (SP) ← DPR
SP' ← SP - 1, (SP) ← ACCB
SP' ← SP - 1, (SP) ← ACCA
SP' ← SP - 1, (SP) ← CCR
PC' ← (FFF2):(FFF3)

CONDITION CODES: Not affected

DESCRIPTION:

All of the MPU registers are pushed onto the hardware stack (excepting only the hardware stack pointer itself), and control is transferred through the SWI3 vector.

ADDRESSING MODE: Absolute Indirect

SYNC Synchronize to External Event

SOURCE FORM: SYNC

OPERATION: Stop processing instructions

CONDITION CODES: Unaffected

DESCRIPTION:

When a SYNC instruction is executed, the MPU enters a SYNCING state, stops processing instructions, and waits on an interrupt. When an interrupt occurs, the SYNCING state is cleared and processing continues. IF the interrupt is enabled, and the interrupt lasts 3 cycles or more, the processor will perform the interrupt routine. If the interrupt is masked or is shorter than 3 cycles long, the processor simply continues to the next instruction (without stacking registers). While SYNCING, the address and data buses are tri-state.

ADDRESSING MODES: Inherent

COMMENTS:

This instruction provides software synchronization with a hardware process. Consider the high-speed acquisition of data:

```
FAST  SYNC          WAIT FOR DATA <------ interrupt!
      LDA   DISC    DATA FROM DISC AND CLEAR INTERRUPT
      STA   ,X+     PUT IN BUFFER
      DECB          COUNT IT, DONE?
      BNE   FAST    GO AGAIN IF NOT.
```

The SYNCING state is cleared by any interrupt, and any enabled interrupt will probably destroy the transfer (this may be used to provide MPU response to an emergency condition).

The same connection used for interrupt-driven I/O service may thus be used for high-speed data transfers by setting the interrupt mask and using SYNC.

TFR Transfer Register to Register

SOURCE FORM: TFR R_1,R_2

OPERATION: $R_2 \leftarrow R_1$

CONDITION CODES: Not affected (Unless R_2 = CCR)

DESCRIPTION:

Bits 7-4 of the immediate byte of the instruction define the source register, while bits 3-0 define the destination register, as follows:

0000 = A:B	1000 = A
0001 = X	1001 = B
0010 = Y	1010 = CCR
0011 = US	1011 = DPR
0100 = SP	1100 = Undefined
0101 = PC	1101 = Undefined
0110 = Undefined	1110 = Undefined
0111 = Undefined	1111 = Undefined

Registers may only be transferred between registers of like size; i.e., 8-bit to 8-bit, and 16 to 16.

ADDRESSING MODES: Inherent

TST Test

SOURCE FORM: TST Q

OPERATION: TEMP ← M - 0

CONDITION CODES:

H: Not affected
N: Set IFF bit 7 of the result is Set
Z: Set IFF all bits of the result are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Set condition code flags N and Z according to the contents of M, and clear the V flag. The 6800 processor clears the C flag.

MEMORY ADDRESSING MODES: Accumulator
Direct
Indexed
Extended

COMMENTS:

The TST instruction provides only minimum information when testing unsigned values; since no unsigned value is less than zero, BLO and BLS have no utility. While BHI could be used after TST, it provides exactly the same control as BNE, which is preferred. The signed branches are available.

HARDWARE INSTRUCTION: FIRQ Fast Interrupt Request

OPERATION: IFF F bit CLEAR, then: SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
Clear E (subset state is saved)
SP' ← SP - 1, (SP) ← CCR
Set F, I (mask further interrupts)
PC' ← (FFF6):(FFF7)

CONDITION CODES: Not affected

DESCRIPTION:

A low level on the FIRQ input with the F bit clear causes this interrupt sequence to occur at the end of the current instruction. The program counter and condition code register are pushed onto the hardware stack. Program control is transferred through the FIRQ vector. An RTI returns to the original task. It is possible to enter an FIRQ handler with the entire state saved if the FIRQ occurs after CWAI.

ADDRESSING MODE: Absolute Indirect

COMMENTS:

An IRQ interrupt, having lower priority then the FIRQ, is prevented from interrupting the FIRQ handling routine by automatic setting of the I flag. This mask bit could then be reset if priority was not desired. The FIRQ allows operations on memory, TST, INC, DEC, etc. without the overhead of saving the entire machine state on the stack.

HARDWARE INSTRUCTION: IRQ Interrupt Request

OPERATION: IFF I bit CLEAR, then:

SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
SP' ← SP - 1, (SP) ← DPR
SP' ← SP - 1, (SP) ← ACCB
SP' ← SP - 1, (SP) ← ACCA
Set E (entire state saved)
SP' ← SP - 1, (SP) ← CCR
Set I (mask further IRQ interrupts)
PC' ← (FFF8):(FFF9)

CONDITION CODES: Not affected

DESCRIPTION:

If the IRQ mask bit I is clear, a low level on the IRQ input causes this interrupt sequence to occur at the end of the current instruction. Control is returned to the interrupted program via an RTI. An FIRQ may interrupt an IRQ handling routine and be recognized anytime after the IRQ vector is taken.

ADDRESSING MODE: Absolute Indirect

HARDWARE INSTRUCTION: NMI Non-Maskable Interrupt

OPERATION:
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
SP' ← SP - 1, (SP) ← DPR
SP' ← SP - 1, (SP) ← ACCB
SP' ← SP - 1, (SP) ← ACCA
Set E (entire state save)
SP' ← SP - 1, (SP) ← CCR
Set I, F (mask interrupts)
PC' ← (FFFC):(FFFD)

CONDITION CODES: Not affected

DESCRIPTION:

A negative edge on the NMI input causes all of the MPU registers (except the hardware stack pointer SP) to be pushed onto the hardware stack, starting at the end of the current instruction. Program control is transferred through the NMI vector. Successive negative edges on the NMI input will cause successive NMI operations. The NMI operation is internally blocked by RESET, any NMI-edge will be latched, and the operation will occur after the first load into the stack pointer (LDS; TFR r,s; EXG r,s; etc.).

ADDRESSING MODE: Absolute Indirect

HARDWARE INSTRUCTION: RESTART

OPERATION: CCR' ← X1X1XXXX
DPR' ← 00_{16}
PC' ← (FFFE):(FFFF)

CONDITION CODES: Not affected

DESCRIPTION:

The MPU is initialized (required after power-on) to start program execution.

ADDRESSING MODE: Absolute Indirect

6809 STACKING ORDER

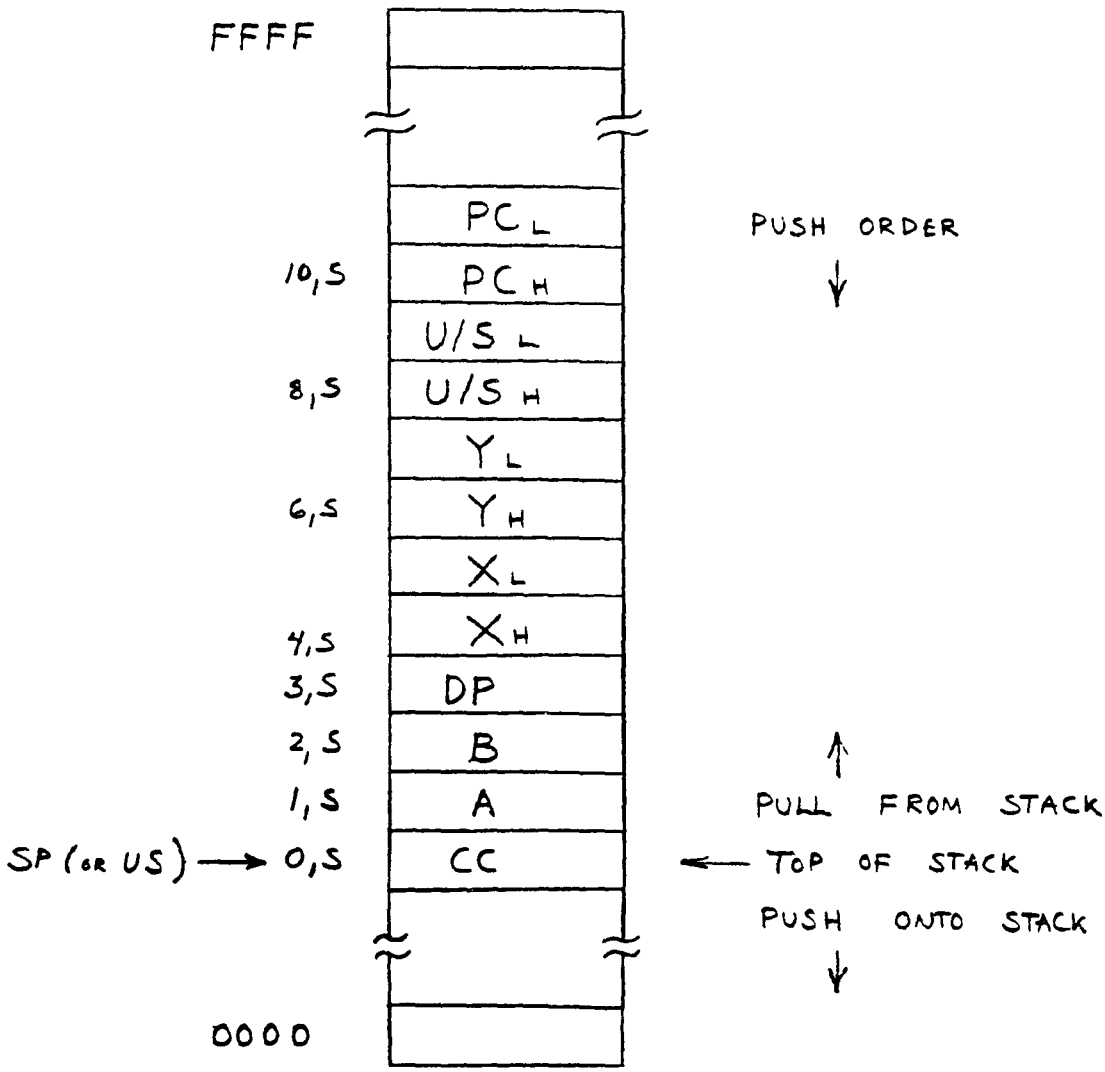


Figure 7: 6809 Push/Pull and Interrupt Stacking Order.

3.5 HARDWARE INCOMPATABILITIES WITH 6800/6801/6802

1. VMA is not used on the on-chip clock 6809; the processor sends $FFFF_{16}$ and R/$\overline{W}$=1 when no valid access is occurring. This dummy access can be differentiated from a valid RESET access by using the IACK signal.

   Since the MREADY line is inhibited internally during dummy access cycles, a slow ROM located in high memory will not extend dummy cycles.

2. While 6800 required a DBE signal (Data Bus Enable and strobe), 6801/6802/6809 generate DBE internally.

3.6 SOFTWARE INCOMPATABILITIES WITH 6800/6801/6802

1. The new stacking order on the 6809 exchanges the order of ACCA and ACCB; this allows ACCA to stack as the MS byte of the pair.

2. The new stacking order on the 6809 invalidates previous 6800 code which displayed X or PC from the stack.

3. Additional stacking length on the 6809 stacks five more bytes for each NMI, IRQ, or SWI when compared to 6800/6801/6802.

4. The 6809 stack pointer points directly at the last item placed on the stack, instead of the location before the last item as in 6800/6801/6802. In general this is not a problem since the most-usual method of pointing at the stack in the 6800/6801/6802 is to execute a TSX. The TSX increments the value during

the transfer, making X point directly at the last item on the stack.

The stack pointer may thus be initialized one location higher on the 6809 than in the 6800/6801/6802; similarly, comparison values may need to be one location higher.

Any 6800/6801 program which does all stack manipulation through X (i.e., LDX #CAT, TXS instead of LDS #CAT) will have an exactly-correct stack translation when assembled for 6809.

5. Instruction timings in 6809 will, in general, be different from other 6800-family processors.

6. The 6809 uses the two high-order condition code register bits. Consequently, these will not, in general, appear as 1's as on the 6800/6801/6802.

7. The 6809 MUL instruction sets the Z-flag (if appropriate); the 6801 MUL does not.

8. The 6809 TST instruction does not affect the Z-flag, while 6800/6801/6802 TST does clear the C-flag.

9. The 6809 right shifts (ASR, LSR, ROR) do not affect V; the 6800/6801/6802 shifts set $V = b_7 \oplus b_6$.

10. The 6801 double-length shift instructions (ASLD, LSRD) are not exactly emulated by the 6800/6802/6809 sequences ASLB, ROLA; and LSRA, ROLB. In particular, the Z-flag represents only the last 8-bit result, and not the 16-bit quantity.

11. The 6809 H-flag is not defined as having any particular state after subtract-like operations (CMP, NEG, SBC, SUB); the 6800/6801/6802 clear the H-flag under these conditions.

12. The 6800/6802 CPX instruction compared MS byte than LS byte; consequently only the Z-flag was set correctly for branching. The 6801/6809 instructions (CPX/CMPX) set all flags correctly.

13. The 6809 instruction LEA may or may not affect the Z-flag depending upon which register is being loaded; LEAX and LEAY do affect the Z-flag, while LEAS and LEAU do not. Thus, the User Stack does not exactly emulate the index registers in this respect.

the transfer, making X point directly at the last item on the stack.

The stack pointer may thus be initialized one location higher on the 6809 than in the 6800/6801/6802; similarly, comparison values may need to be one location higher.

Any 6800/6801 program which does all stack manipulation through X (i.e., LDX #CAT, TXS instead of LDS #CAT) will have an exactly-correct stack translation when assembled for 6809.

5. Instruction timings in 6809 will, in general, be different from other 6800-family processors.

6. The 6809 uses the two high-order condition code register bits. Consequently, these will not, in general, appear as 1's as on the 6800/6801/6802.

7. The 6809 MUL instruction sets the Z-flag (if appropriate); the 6801 MUL does not.

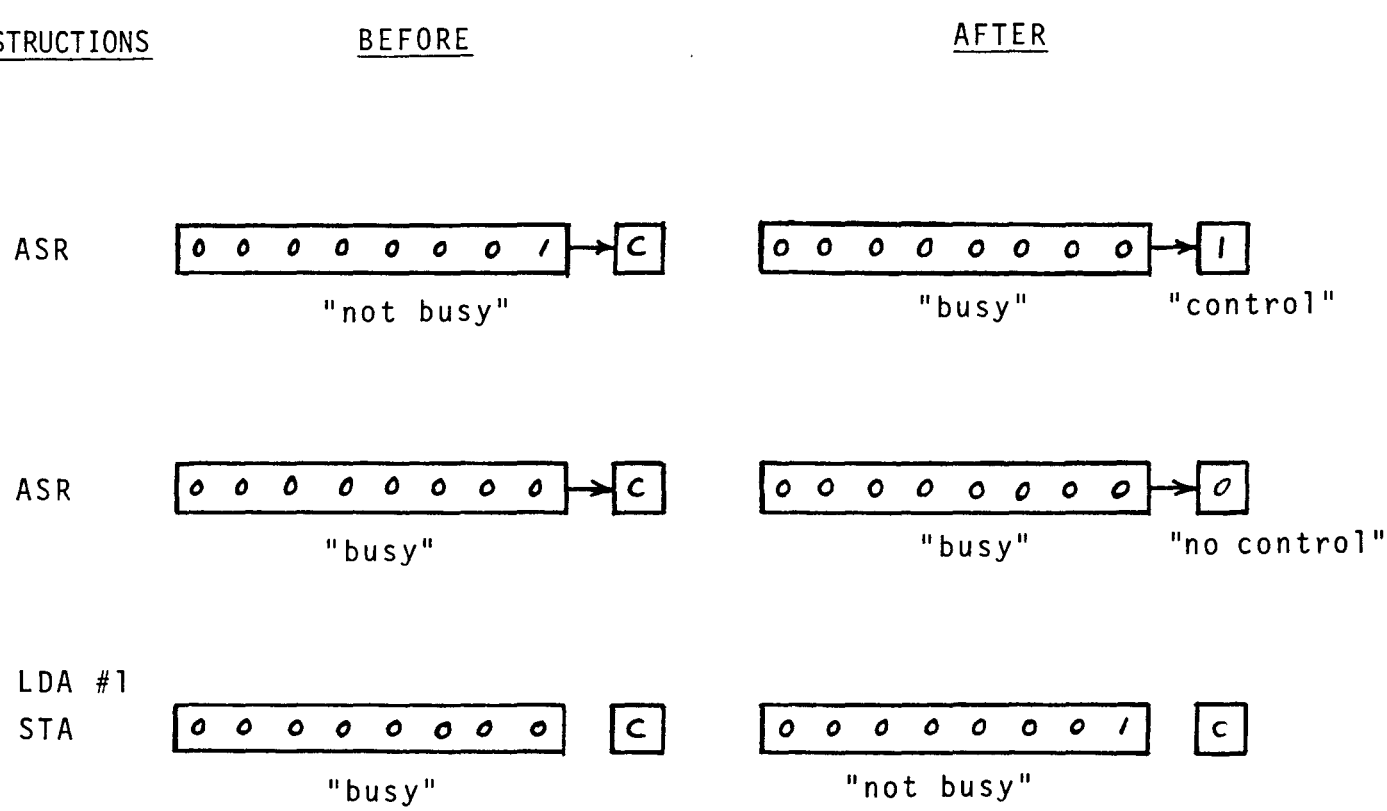
8. The 6809 TST instruction does not affect the Z-flag, while 6800/6801/6802 TST does clear the C-flag.

9. The 6809 right shifts (ASR, LSR, ROR) do not affect V; the 6800/6801/6802 shifts set $V' = b_7 \oplus b_6$.

10. The 6801 double-length shift instructions (ASLD, LSRD) are not exactly emulated by the 6800/6802/6809 sequences ASLB, ROLA; and LSRA, ROLB. In particular, the Z-flag represents only the last 8-bit result, and not the 16-bit quantity.

3.7 MULTI-PROCESS SYNCHRONIZATION

ASR used as "Test and Clear"
ST used as "Unbusy"



3.8 6809 ASSEMBLY-LANGUAGE SYNTAX

ABX	ABX						
ADC	ADCA	P;	ADCB	P			
ADD	ADDA	P;	ADDB	P;	ADDD	P	
AND	ANDA	P;	ANDB	P;	ANDCC	#XX	
ASL	ASL	Q					
ASR	ASR	Q					
BCC	BCC	dd;	LBCC	DDDD			
BCS	BCS	dd;	LBCS	DDDD			
BEQ	BEQ	dd;	LBEQ	DDDD			
BGE	BGE	dd;	LBGE	DDDD			
BGT	BGT	dd;	LBGT	DDDD			
BHI	BHI	dd;	LBHI	DDDD			
BHS	BHS	dd;	LBHS	DDDD			
BIT	BITA	P;	BITB	P			
BLE	BLE	dd;	LBLE	DDDD			
BLO	BLO	dd;	LBLO	DDDD			
BLS	BLS	dd;	LBLS	DDDD			
BLT	BLT	dd;	LBLT	DDDD			
BMI	BMI	dd;	LBMI	DDDD			
BNE	BNE	dd;	LBNE	DDDD			
BPL	BPL	dd;	LBPL	DDDD			
BRA	BRA	dd;	LBRA	DDDD			
BRN	BRN	dd;	LBRN	DDDD			
BSR	BSR	dd;	LBSR	DDDD			
BVC	BVC	dd;	LBVC	DDDD			
BVS	BVS	dd;	LBVS	DDDD			
CLR	CLR	Q					
CMP	CMPA	P;	CMPB	P;	CMPD	P	
	CMPX	P;	CMPY	P;	CMPS	P	
	CMPU	P					
COM	COM	Q					
CWAI	CWAI	#XX					

3.8 (Continued)

```
DA          DAA
DEC         DEC   Q
EOR         EORA  P;     EORB  P
EXG         EXG   R,R
INC         INC   Q
JMP         JMP   XXXX
JSR         JSR   XXXX
LD          LDA   P;     LDB   P;     LDD   P
            LDX   P;     LDY   P;     LDS   P
            LDU   P
LEA         LEAX  IN;    LEAY  IN;    LEAU  IN
            LEAS  IN
LSL         LSL   Q
LSR         LSR   Q
MUL         MUL
NEG         NEG   Q
NOP         NOP
OR          ORA   P;     ORB   P;     ORCC  #XX
PUL         PULS  R{,R} ;  PULU R{,R}
PSH         PSHS  R{,R} ;  PSHU R{,R}
ROL         ROL   Q
ROR         ROR   Q
RTI         RTI
RTS         RTS
SBC         SBCA  P;     SBCB  P
SEX         SEX
ST          STA   P;     STB   P;     STD   P
            STX   P;     STY   P;     STS   P
            STU   P
SUB         SUBA  P;     SUBB  P;     SUBD  P
SWI         SWI
SWI2        SWI2
SWI3        SWI3
```

3.8 (Continued)

```
SYNC          SNYC
TFR           TFR    R,R
TST           TST    Q
```

3.9 MC6800 - Equivalent Instructions

MC6800 mnemonics which are not included in the MC6809 assembly-language are handled by automatically translating the 6800 instruction into functionally-equivalent 6809 instructions, as described:

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 P
DES	LEAS -1,S
DEX	LEAX -1,X
INS	LEAS 1,S
INX	LEAX 1,X
LDAA	LDA
LDAB	LDB
ORAA	ORA
ORAB	ORB
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

3.9 (Continued)

6800 Instruction	6809 Equivalent
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	*CWAI #$FF

* The interrupt structure on the 6809 has been extensively analyzed and improved compared to the 6800. While with the 6800 it was useful to execute the sequence: CLI, WAI; the 6809 logically-equivalent sequence (ANDCC #$EF, CWAI #$FF) would allow on IRQ interrupt to occur after the ANDCC instruction. If this is not desired, the 6809 instruction CWAI #$EF should be used to replace the logically-equivalent sequence.

6809 op code map and cycle counts. The numbers by each op code indicate the number of machine cycles required to execute each instruction. When the number contains an I (eg: 4 + I), an additional number of machine cycles equaling I may be required. The presence of two numbers, with the second one in parentheses, indicates that the instruction involves a branch. The larger number applies if the branch is taken. The notation first page/second page/third page has the following meaning: first page op codes have only one byte of op code (eg: load A immediate has an op code of hexadecimal 86). All page 2 op codes are preceded by a page op code of hexadecimal 10 (eg: the op code for CMPD immediate is hexadecimal 1083– two bytes). Similarly third page op codes are preceded by a hexadecimal 11. A CMPU immediate is 1183. Some instructions are given two mnemonics as a programmer convenience (eg: ASL and LSL are equivalent). Notice that the long branch op codes LBRA and LBSR were brought onto the first page for increased code efficiency.

Least Significant Four Bits	Most Significant Four Bits: 0000 (0)	0001 (1)	0010 (2)	0011 (3)	0100 (4)	0101 (5)	0110 (6)	0111 (7)	1000 (8)	1001 (9)	1010 (A)	1011 (B)	1100 (C)	1101 (D)	1110 (E)	1111 (F)	Least Significant Four Bits
	DIR		REL		ACCA	ACCB	IND	EXT	IMM	DIR	IND	EXT	IMM	DIR	IND	EXT	
0000 (0)	6 NEG	PAGE2	3 BRA	4+1 LEAX	2	2	6+1 NEG	7	2	4	4+1 SUBA	5	2	4	4+1 SUBB	5	C
0001 (1)	——	PAGE3	3 BRN/ 5 LBRN	4+1 LEAY	——	——	——	——	2	4	4+1 CMPA	5	2	4	4+1 CMPB	5	1
0010 (2)	——	2 NOP	3 BHI/ 5(6) LBHI	4+1 LEAS	——	——	——	——	2	4	4+1 SBCA	5	2	4	4+1 SBCB	5	2
0011 (3)	6 COM	2 SYNC	3 BLS/ 5(6) LBLS	4+1 LEAU	2	2	6+1 COM	7	4,6,6+1,7 SUBD /	5,7,7+1,8 CMPD /	5,7,7+1,8 CMPU		4	6	6+1 ADDD	7	3
0100 (4)	6 LSR	——	3 BHS 5(6) (BCC)	5+1/by PSHS	2	2	6+1 LSR	7	2	4	4+1 ANDA	5	2	4	4+1 ANDB	5	4
0101 (5)	——	——	3 BLO 5(6) (BCS)	5+1/by PULS	——	——	——	——	2	4	4+1 BITA	5	2	4	4+1 BITB	5	5
0110 (6)	6 ROR	5 LBRA	3 BNE/ 5(6) LBNE	5+1/by PSHU	2	2	6+1 ROR	7	2	4	4+1 LDA	5	2	4	4+1 LDB	5	6
0111 (7)	6 ASR	9 LBSR	3 BEQ/ 5(6) LBEQ	5+1/by PULU	2	2	6+1 ASR	7	——	4	4+1 STA	5	——	4	4+1 STB	5	7
1000 (8)	6 ASL (LSL)	——	3 BVC/ 5(6) LBVC	——	2	2	6+1 ASL(LSL)	7	2	4	4+1 EORA	5	2	4	4+1 EORB	5	8
1001 (9)	6 ROL	2 DAA	3 BVS/ 5(6) LBVS	5 RTS	2	2	6+1 ROL	7	2	4	4+1 ADCA	5	2	4	4+1 ADCB	5	9
1010 (A)	6 DEC	3 ORCC	3 BPL/ 5(6) LBPL	3 ABX	2	2	6+1 DEC	7	2	4	4+1 ORA	5	2	4	4+1 ORB	5	A
1011 (B)	——	——	3 BMI/ 5(6) LBMI	6/15 RTI	——	——	——	——	2	4	4+1 ADDA	5	2	4	4+1 ADDB	5	B
1100 (C)	6 INC	3 ANDCC	3 BGE/ 5(6) LBGE	20 CWAI	2	2	6+1 INC	7	4,6,6+1,7 CMPX /	5,7,7+1,8 CMPY /	5,7,7+1,8 CMPS		3	5	5+1 LDD	6	C
1101 (D)	6 TST	2 SEX	3 BLT/ 5(6) LBLT	11 MUL	2	2	6+1 TST	7	7 BSR	7	7+1 JSR	8	——	5	5+1 STD	6	D
1110 (E)	3 JMP	8 EXG	3 BGT/ 5(6) LBGT	——	——	——	3+1 JMP	4	3,5,5+1,6 LDX	/	4,6,6+1,7 LDY		3,5,5+1,6 LDU	/	4,6,6+1,7 LDS		E
1111 (F)	6 CLR	7 TFR	3 BLE/ 5(6) LBLE	19/20/20 SWI/2/3	2	2	6+1 CLR	7	——	5,5+1,6 STX	/	6,6+1,7 STY	——	5,5+1,6 STU	/	6,6+1,7 STS	F

6809 INDEXED ADDRESSING

TYPE	FORMS	NON-INDIRECT				INDIRECT			
		SOURCE	POST-BYTE	$\overset{+}{\sim}$	$\overset{+}{\#}$	SOURCE	POST-BYTE	$\overset{+}{\sim}$	$\overset{+}{\#}$
CONSTANT OFFSET FROM R	NO OFFSET	,R	1RR00100	0	0	[,R]	1RR10100	3	0
	5-BIT OFFSET	n,R	0RRnnnnn	1	0	defaults to 8-bit			
	8-BIT OFFSET	n,R	1RR01000	1	1	[n,R]	1RR11000	4	1
	16-BIT OFFSET	n,R	1RR01001	4	2	[n,R]	1RR11001	7	2
ACCUMULATOR OFFSET FROM R	A-REGISTER OFFSET	A,R	1RR00110	1	0	[A,R]	1RR10110	4	0
	B-REGISTER OFFSET	B,R	1RR00101	1	0	[B,R]	1RR10101	4	0
	D-REGISTER OFFSET	D,R	1RR01011	4	0	[D,R]	1RR11011	7	0
AUTO-INCREMENT/ -DECREMENT R	INCREMENT BY 1	,R+	1RR00000	2	0	not allowed			
	INCREMENT BY 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	DECREMENT BY 1	,-R	1RR00010	2	0	not allowed			
	DECREMENT BY 2	,--R	1RR00011	3	0	[,--R]	1RR10011	6	0
CONSTANT OFFSET FROM PC	8-BIT OFFSET	n,PCR	1XX01100	1	1	[n,PCR]	1XX11100	4	1
	16-BIT OFFSET	n,PCR	1XX01101	5	2	[n,PCR]	1XX11101	8	2
EXTENDED		use non-indexed				[n]	10011111	5	2

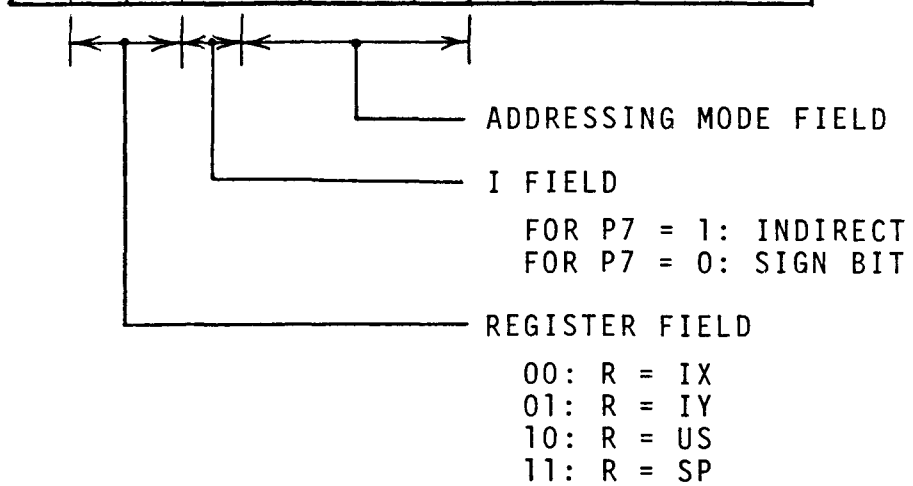
Figure 4: Indexed Addressing Modes. All instructions with indexed addressing have a base size and number of cycles. The $\overset{+}{\sim}$ and $\overset{+}{\#}$ columns indicate the number of additional cycles and bytes for the particular variation. The post byte opcode is the byte that immediately follows the normal opcode.

3.12 INDEXED-MODE POST-BYTE

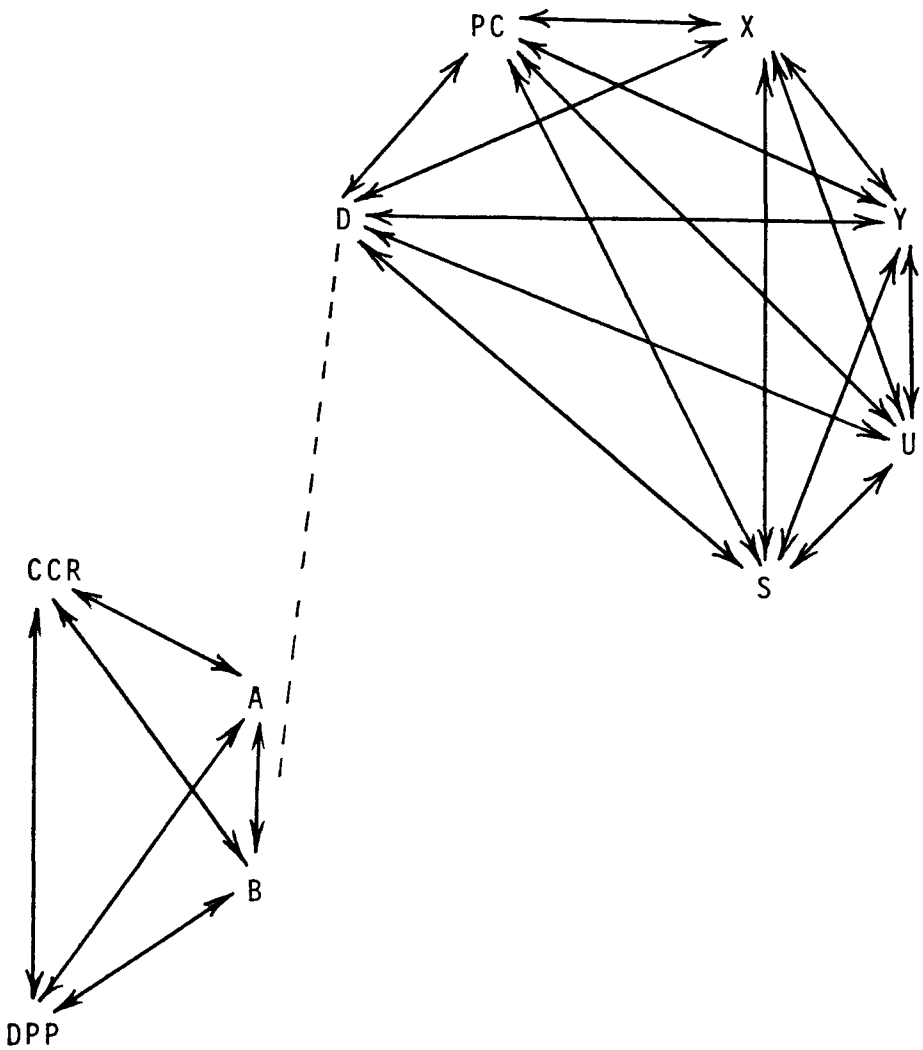
POST BYTE REGISTER

BIT ASSIGNMENTS

POST-BYTE REGISTER BIT								INDEXED ADDRESSING MODE
7	6	5	4	3	2	1	0	
1	X	X	X	0	0	0	1	,R++
1	X	X	0	0	0	0	0	,R+
1	X	X	0	0	0	1	0	,-R
1	X	X	X	0	0	1	1	,--R
1	X	X	X	0	1	0	0	EA=(R ± 0 OFFSET)
1	X	X	X	0	1	0	1	EA=(R ± ACCB OFFSET)
1	X	X	X	1	0	0	0	EA=(R±7BIT OFFSET)
1	X	X	X	1	0	0	1	EA=(R±15 BIT OFFSET)
1	X	X	X	1	1	0	0	EA=(PC ±7 BIT OFFSET)
1	X	X	X	1	1	0	1	EA=(PC ±15 BIT OFFSET)
0	X	X	X	X	X	X	X	EA=(R±4 BIT OFFSET)
1	X	X	X	0	1	1	0	EA=(R±ACCA OFFSET)
1	X	X	X	1	0	1	1	EA=(R±D OFFSET)
1	X	X	1	1	1	1	1	EA=(ADDRESS)



3.13 LEGAL TRANSFER AND EXCHANGE PATHS



3.14 BRANCH GROUPS

Simple Conditional Branches

Condition	Complement
BEQ {Z=1}	BNE
BMI {N=1}	BPL
BCS {C=1}	BCC
BVS {V=1}	BVC

Signed Conditional Branches

Condition	Complement
BGT $\{(\overline{N \oplus V}) \wedge \overline{Z}=1\}$	BLE
BGE $\{(\overline{N \oplus V})=1\}$	BLT
BEQ {Z=1}	BNE
BLE $\{(N \oplus V) \vee Z=1\}$	BGT
BLT $\{(N \oplus V)=1\}$	BGE

Unsigned Conditional Branches*

Condition	Complement
BHI $\{(\overline{C} \wedge \overline{Z})=1\}$	BLS
BHS $\{\overline{C}=1\}$	BLO
BEQ {Z=1}	BNE
BLS $\{C \vee Z=1\}$	BHI
BLO {C=1}	BHS

* Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

ABX	Add B-register to X-register unsigned
ADCA,ADCB	Add memory to accumulator with carry
ADDA,ADDB	Add memory to accumulator
ANDA,ANDB	And memory with accumulator
ANDCC	And immediate with condition code register
ASLA,ASLB,ASL	Arithmetic shift left accumulator or memory
ASRA,ASRB,ASR	Arithmetic shift right accumulator or memory
BITA,BITB	Bit test memory with accumulator
CLRA,CLRB,CLR	Clear accumulator or memory
CMPA,CMPB	Compare memory with accumulator
COMA,COMB,COM	Complement accumulator or memory
DAA	Decimal Adjust A-accumulator
DECA,DECB,DEC	Decrement accumulator or memory
EORA,EORB	Exclusive or memory with accumulator
EXG R1,R2	Exchange R1 with R2
INCA,INCB,INC	Increment accumulator or memory
LDA,LDB	Load accumulator from memory
LSLA,LSLB,LSL	Logical shift left accumulator or memory
LSRA,LSRB,LSR	Logical shift right accumulator or memory
MUL	Unsigned multiply (8 bit x 8 bit = 16 bit)
NEGA,NEGB,NEG	Negate accumulator or memory
ORA,ORB	Or memory with accumulator
ORCC	Or immediate with condition code register
PSHS {register}$^{8}_{0}$	Push register(s) on hardware stack
PSHU {register}$^{8}_{0}$	Push register(s) on user stack
PULS {register}$^{8}_{0}$	Pull register(s) from hardware stack
PULU {register}$^{8}_{0}$	Pull register(s) from user stack
ROLA,ROLB,ROL	Rotate accumulator or memory left
RORA,RORB,ROR	Rotate accumulator or memory right
SBCA,SBCB	Subtract memory from accumulator with borrow
STA,STB	Store accumulator to memory
SUBA,SUBB	Subtract memory from accumulator
TSTA,TSTB,TST	Test accumulator or memory
TFR R1,R2	Transfer register R1 to register R2

FIGURE 1 8-BIT OPERATIONS

ADDD	Add to D accumulator
SUBD	Subtract from D accumulator
LDD	Load D accumulator
STD	Store D accumulator
CMPD	Compare D accumulator
LDX,LDY,LDS,LDU	Load pointer register
STX,STY,STS,STU	Store pointer register
CMPX,CMPY,CMPU,CMPS	Compare pointer register
LEAX,LEAY,LEAS,LEAU	Load effective address into pointer register
SEX	Sign Extend
TFR register,register	Transfer register to register
EXG register,register	Exchange register to register
PSHS (register)$^{8}_{0}$	Push register(s) onto hardware stack
PSHU (register)$^{8}_{0}$	Push register(s) onto user stack
PULS (register)$^{8}_{0}$	Pull register(s) from hardware stack
PULU (register)$^{8}_{0}$	Pull register(s) from user stack

FIGURE 2 16-BIT OPERATIONS

0,R	indexed with zero offset
[0,R]	indexed with zero offset indirect
,R+	auto increment by 1
,R++	auto increment by 2
[,R++]	auto increment by 2 indirect
,-R	auto decrement by 1
,--R	auto decrement by 2
[,--R]	auto decrement by 2 indirect
n,P	indexed with signed n as offset (n=5,8, or 16-bits)
[n,P]	indexed with signed n as offset indirect
A,R	indexed with accumulator A as offset
[A,R]	indexed with accumulator A as offset indirect
B,R	indexed with accumulator B as offset
[B,R]	indexed with accumulator B as offset indirect
D,R	indexed with accumulator D as offset
[D,R]	indexed with accumulator D as offset indirect

R = X, Y, U or S

P = PC, X, Y, U or S

FIGURE 3 INDEXED ADDRESSING MODES

BCC,LBCC	Branch if carry clear
BCS,LBCS	Branch if carry set
BEQ,LBEQ	Branch if equal
BGE,LBGE	Branch if greater than or equal (signed)
BGT,LBGT	Branch if greater (signed)
BHI,LBHI	Branch if higher (unsigned)
BHS,LBHS	Branch if higher or same (unsigned)
BLE,LBLE	Branch if less than or equal (signed)
BLO,LBLO	Branch if lower (unsigned)
BLS,LBLS	Branch if lower or same (unsigned)
BLT,LBLT	Branch if less than (signed)
BMI,LBMI	Branch if minus
BNE,LBNE	Branch is not equal
BPL,LBPL	Branch if plus
BRA,LBRA	Branch always
BRN,LBRN	Branch never
BSR,LBSR	Branch to subroutine
BVC,LBVC	Branch if overflow clear
BVS,LBVS	Branch if overflow set

FIGURE 4 RELATIVE SHORT AND LONG BRANCHES

CWAI	Clear condition code register bits and wait for interrupt
NOP	No-operation
JMP	Jump
JSR	Jump to subroutine
RTI	Return from interrupt
RTS	Return from subroutine
SEX	Sign extend B-register into A-register
SWI,SWI2,SWI3	Software interrupts
SYNC	Synchronize with interrupt line

FIGURE 5 MISCELLANEOUS INSTRUCTIONS

4.0 SYSTEMS INTERFACING

4.1 INTERRUPTS

Three different classes of prioritized vectored interrupts are included in the 6809 MPU. In decreasing priority these are: NMI (Non-Maskable Interrupt), FIRQ (Fast Interrupt Request), and IRQ (Interrupt Request) and are more fully defined in the "Hardware Instructions" section.

Using the processor signal line Interrupt Acknowledge (IACK) and decoding four bits of the Address Bus, the interrupt response may be vectored by the interrupting device to anywhere in the address-space. This technique can be used to greatly expand the number of prioritized hardware-vectored interrupts.

The NMI is especially applicable to gaining immediate (non-inhibitable) MPU response for power-fail, software dynamic memory refresh, or other non-delayable events. FIRQ is a maskable fast interrupt which saves only a return address and condition codes, making it much faster than NMI or IRQ. IRQ is a maskable interrupt which saves a complete MPU state.

Two types of external-process synchronization are also provided by the interrupt system. The CWAI command saves the entire MPU state, then waits until a non-inhibited interrupt occurs before vectoring to the interrupt routine. A SYNC instruction stops the MPU from executing code until an interrupt is received. If the interrupt is masked, the MPU simply resumes execution. If the interrupt is enabled, the interrupt response is performed.

```
00007                          ********** I/O HANDLER **********
00008                          *
00009                          *     A SINGLE INPUT INTERRUPT IS ARMED.  RECEIVE
00010                          *     AN INTERRUPT, SAVE REGISTERS, INPUT A CHAR,
00011                          *     CLEAR THE INTERRUPT, PUT THE CHAR IN A
00012                          *     SOFTWARE BUFFER, INCREMENT THE BUFFER PTR,
00013                          *     TEST FOR END OF LINE, RECOVER REGISTERS,
00014                          *     AND RETURN.
00015                          *
00016                          *     SETUP:  NONE
00017                          *     TOTAL:  7 LN, 16 BY, 62 CY
00018                          *
00019                          ********************************


00021              000D        EOL     EQU   $0D       ASCII CR
00022 1004         00          MODEM   FCB   0
00023 1005         0100        BUFPTR  FDB   $100


00025                          * ASSUME IRQ FROM PIA (19 CY)

00027 1007 B6   1004      5 BEGIN  LDA   MODEM    CLEARS PIA IRQ
00028 100A BE   1005      6        LDX   BUFPTR   GET PTR
00029 100D A7   80        6        STA   ,X+      STORE CHAR
00030 100F BF   1005      6        STX   BUFPTR   UPDATE PTR
00031 1012 81   0D        2        CMPA  #EOL     END OF LINE?
00032 1014 27   01        3        BEQ   EOLGP    IF YES, MORE TO DO
00033 1016 3B            15        RTI            ELSE, RETURN


00035 1017 20   FE        3 EOLGP  BRA   *
```

```
00038                          ********** CHARACTER SEARCH **********
00039                          *
00040                          *     SEARCH A TABLE OF N CHARACTERS FOR A SPECIFIC
00041                          *     CHARACTER.  IF FOUND, RETURN THE ADDRESS OF
00042                          *     THE MATCH, ELSE RETURN ZERO.  LET N BE 40.
00043                          *     LET THE SEARCH FAIL.
00044                          *
00045                          *     SETUP:        3 LN,  7 BY,   7 CY
00046                          *     OPERATION:    6 LN, 12 BY, (14*40)+8=568 CY
00047                          *     TOTAL:        9 LN, 19 BY, 575 CY
00048                          *
00049                          ************************************

00051 1019 86   4A       2 CSRCH  LDA   #CHAR     CHAR TO FIND
00052 101B 8E   102E     3        LDX   #BUF      PTR INTO TABLE
00053 101E C6   28       2        LDB   #40       LENGTH OF TABLE

00055 1020 A1   80       6 CS1    CMPA  ,X+       SAME CHAR?
00056 1022 27   06       3        BEQ   CS2       IF YES, POINT AT IT
00057 1024 5A            2        DECB             ANOTHER ONE DOWN
00058 1025 26   F9       3        BNE   CS1       ALL DONE?
00059 1027 8E   0001     3        LDX   #1        TRICKY CLRX
00060 102A 30   1F       5 CS2    LEAX  -1,X      WENT PAST!

00062 102C 20   FE       3        BRA   *

00064          004A       CHAR   EQU   'J
00065 102E     00         BUF    FCB   0,,,,,,,,,,,,,,,,,,,,0
00066 1042     00                FCB   0,,,,,,,,,,,,,,,,,,,,0
```

```
00069                          ********** COMPUTED GO TO **********
00070                          *
00071                          *      LSB FIRST, TEST A CONTROL BYTE WHICH HAS
00072                          *      HAS EXACTLY ONE BIT TRUE.  THE POSITION
00073                          *      OF THE TRUE BIT DETERMINES WHICH OF EIGHT
00074                          *      TABLE VECTORS IS USED FOR CONTROL-TRANSFER
00075                          *      LET B7 BE TRUE.
00076                          *
00077                          *      SETUP:        2 LN,  5 BY,   5 CY
00078                          *      OPERATION:    5 LN,  8 BY, 2+(7*8)+7=65 CY
00079                          *      TOTAL:        7 LN, 13 BY,  70 CY
00080                          *
00081                          ************************************

00083 1056 86   80       2 COMPGO LDA   #CONTBY
00084 1058 8E   1061     3        LDX   #TABLE-2 START OF TABLE

00086 105B 5F            2        CLRB
00087 105C CB   02       2 CO1    ADDB  #2       TWO BYTES / VECTOR
00088 105E 44            2        LSRA
00089 105F 24   FB       3        BCC   CO1
00090 1061 6E   95       7        JMP   [B,X]    REGISTER-OFFSET INDIRECT


00092           0080       CONTBY EQU   $80
00093 1063      1073       TABLE  FDB   ERR,ERR,ERR,ERR,ERR,ERR,ERR
00094 1071      1075              FDB   NOERR
00095 1073 20   FE       3 ERR    BRA   *
00096 1075 20   FE       3 NOERR  BRA   *
```

```
00099                          ********** VECTOR ADDITION / 16-BIT **********
00100                          *
00101                          *     PERFORM AN ELEMENT-BY-ELEMENT ADDITION ON
00102                          *     TWO VECTORS OF N 16-BIT ELEMENTS EACH.
00103                          *     PLACE THE RESULT IN A DIFFERENT VECTOR.
00104                          *     LET N BE 20.
00105                          *
00106                          *     SETUP:        3 LN, 10 BY,  10 CY
00107                          *     OPERATION:    5 LN, 11 BY, 32*20=640 CY
00108                          *     TOTAL:        8 LN, 21 BY, 650 CY
00109                          *
00110                          ***********************************************


00112 1077 8E   108E      3 ANBNCN LDX   #TABLEA
00113 107A 108E 10B6      4        LDY   #TABLEB
00114 107E CE   10DE      3        LDU   #TABLEC


00116 1081 EC   81        8 AN1    LDD   ,X++
00117 1083 E3   A1        9        ADDD  ,Y++
00118 1085 ED   C1        8        STD   ,U++
00119 1087 8C   10B6      4        CMPX  #2*20+TABLEA
00120 108A 26   F5        3        BNE   AN1


00122 108C 20   FE        3        BRA   *




00124 108E      0000        TABLEA FDB   $00,$01,$02,$03,$04
00125 1098      0005               FDB   $05,$06,$07,$08,$09
00126 10A2      0010               FDB   $10,$11,$12,$13,$14
00127 10AC      0015               FDB   $15,$16,$17,$18,$19
00128 10B6      0099        TABLEB FDB   $99,$98,$97,$96,$95
00129 10C0      0094               FDB   $94,$93,$92,$91,$90
00130 10CA      0089               FDB   $89,$88,$87,$86,$85
00131 10D4      0084               FDB   $84,$83,$82,$81,$80
00132 10DE      0000        TABLEC FDB   0,,,,,,,,,,,,,,,,,,,,0
```

```
00135                          ********** VECTOR ADDITION / 8-BIT **********
00136                          *
00137                          *     PERFORM AN ELEMENT-BY-ELEMENT ADDDITION
00138                          *     ON TWO VECTORS OF N 8-BIT ELEMENTS EACH.
00139                          *     PLACE THE RESULT IN A DIFFERENT VECTOR.
00140                          *     LET N BE 20.
00141                          *
00142                          *     SETUP:        3 LN, 10 BY,  10 CY
00143                          *     OPERATION:    6 LN, 13 BY, 10*35=350 CY
00144                          *     TOTAL:        9 LN, 23 BY, 360 CY
00145                          *
00146                          ******************************************

00148 1106 8E   111F     3 ABCNNN LDX   #TABLA
00149 1109 108E 1133     4        LDY   #TABLB
00150 110D CE   1147     3        LDU   #TABLC

00152 1110 EC   81       8 ABC1   LDD   ,X++
00153 1112 AB   A0       6        ADDA  ,Y+
00154 1114 EB   A0       6        ADDB  ,Y+
00155 1116 ED   C1       8        STD   ,U++
00156 1118 8C   1133     4        CMPX  #TABLA+20
00157 111B 26   F3       3        BNE   ABC1

00159 111D 20   FE       3        BRA   *



00161 111F      00         TABLA  FCB   $00,$01,$02,$03,$04
00162 1124      05                FCB   $05,$06,$07,$08,$09
00163 1129      10                FCB   $10,$11,$12,$13,$14
00164 112E      15                FCB   $15,$16,$17,$18,$19
00165 1133      99         TABLB  FCB   $99,$98,$97,$96,$95
00166 1138      94                FCB   $94,$93,$92,$91,$90
00167 113D      89                FCB   $89,$88,$87,$86,$85
00168 1142      84                FCB   $84,$83,$82,$81,$80
00169 1147      00         TABLC  FCB   0,,,,,,,,,,,,,,,,,,,,0
```

```
00172                          ********** 16-BIT SHIFTS **********
00173                          *
00174                          *     LOGICALLY SHIFT A 16-BIT QUANTITY FROM
00175                          *     MEMORY RIGHT N PLACES.  (ZERO FILLS ON
00176                          *     LEFT).  PLACE THE RESULT IN MEMORY.
00177                          *     LET N BE 5.
00178                          *
00179                          *     SETUP:        1 LN,  2 BY,   2 CY
00180                          *     OPERATION:    8 LN, 16 BY, (13*5)+7 =88 CY
00181                          *     TOTAL:        9 LN, 18 BY,  90 CY
00182                          *
00183                          ************************************

00185 115B C6   05       2 BEG    LDB   #5

00187 115D 34   04       6        PSHS  B
00188 115F FC   116F     6        LDD   DWORD
00189 1162 44            2 BE1    LSRA
00190 1163 56            2        RORB
00191 1164 6A   E4       6        DEC   0,S
00192 1166 26   FA       3        BNE   BE1
00193 1168 FD   116F     6        STD   DWORD
00194 116B 32   61       5        LEAS  1,S       CLEAN UP STACK

00196 116D 20   FE       3        BRA   *

00198 116F      F1CD       DWORD  FDB   $F1CD
```

```
00201                          ********** DOUBLE SHIFT RIGHT FIVE PLACES ********
00202                          *
00203                          *      LOGICALLY SHIFT RIGHT A 16-BIT QUANTITY
00204                          *      FROM MEMORY EXACTLY 5 PLACES.
00205                          *      REPLACE THE RESULT IN MEMORY.
00206                          *
00207                          *      SETUP:  NONE
00208                          *      TOTAL:  12 LN, 16 BY, 30 CY
00209                          *
00210                          ************************************************


00212 1171 FC   1183     6          LDD   WORD     GET DOUBLE BYTE
00213 1174 44            2          LSRA           : 16-BIT SHIFT
00214 1175 56            2          RORB           :
00215 1176 44            2          LSRA           AGAIN
00216 1177 56            2          RORB
00217 1178 44            2          LSRA           AGAIN
00218 1179 56            2          RORB
00219 117A 44            2          LSRA           AGAIN
00220 117B 56            2          RORB
00221 117C 44            2          LSRA           AGAIN
00222 117D 56            2          RORB
00223 117E FD   1183     6          STD   WORD     STORE DOUBLE BYTE


00225 1181 20   FE       3          BRA   *



00227 1183      F1CD          WORD   FDB   $F1CD
```

```
00230                          ********** 16 X 16 MULTIPLY **********
00231                          *
00232                          *      MULTIPLY TWO 16-BIT POSITIVE VALUES
00233                          *      TO GENERATE A 32-BIT PRODUCT.
00234                          *      AT TERMINATION, BOTH INPUT VALUES
00235                          *      AND THE RESULT WILL BE IN MEMORY.
00236                          *
00237                          *      (A:B) X (C:D) =          BDH:BDL
00238                          *                    +      BCH:BCL
00239                          *                    +      ADH:ADL
00240                          *                    + ACH:ACL
00241                          *                      ---------------
00242                          *
00243                          *      SETUP:       3 LN, 10 BY,  10 CY
00244                          *      OPERATION:  25 LN, 46 BY, 154 CY
00245                          *      TOTAL:      28 LN, 56 BY, 164 CY
00246                          *
00247                          ************************************

00249 1185 8E   11BF     3 ABC    LDX   #AA       POINTER TO A (MS BYTE)
00250 1188 108E 11C1     4        LDY   #BB
00251 118C CE   11C3     3        LDU   #C

00253 118F 6F   C4       6        CLR   0,U
00254 1191 6F   41       7        CLR   1,U
00255 1193 A6   01       5        LDA   1,X       : #A LS BYTE
00256 1195 E6   21       5        LDB   1,Y       : #B LS BYTE
00257 1197 3D           11        MUL
00258 1198 ED   42       6        STD   2,U
00259 119A A6   84       4        LDA   0,X       : #A MS BYTE
00260 119C E6   21       5        LDB   1,Y       : #B LS BYTE
00261 119E 3D           11        MUL
00262 119F E3   41       7        ADDD  1,U
00263 11A1 ED   41       6        STD   1,U
00264 11A3 24   02       3        BCC   AB1
00265 11A5 6C   C4       6        INC   0,U
00266 11A7 A6   01       5 AB1    LDA   1,X       : #A LS BYTE
00267 11A9 E6   A4       4        LDB   0,Y       : #B MS BYTE
00268 11AB 3D           11        MUL
00269 11AC E3   41       7        ADDD  1,U
00270 11AE ED   41       6        STD   1,U
00271 11B0 24   02       3        BCC   AB2
00272 11B2 6C   C4       6        INC   0,U
00273 11B4 A6   84       4 AB2    LDA   0,X       : #A MS BYTE
00274 11B6 E6   A4       4        LDB   0,Y       : #B MS BYTE
00275 11B8 3D           11        MUL
00276 11B9 E3   C4       6        ADDD  0,U
00277 11BB ED   C4       5        STD   0,U

00279 11BD 20   FE       3        BRA   *

00281 11BF      03E8       AA     FDB   1000
00282 11C1      01F4       BB     FDB   500
00283 11C3      0000       C      FDB   0,0
```

```
*T
END ADDR 11BD
>1185;G
LDX    #11BF  P-1188 X-11BF Y-11C1 A-00 B-07 C-D0 D-00 U-11C3 S-2000
LDY    #11C1  P-118C X-11BF Y-11C1 A-00 B-07 C-D0 D-00 U-11C3 S-2000
LDU    #11C3  P-118F X-11BF Y-11C1 A-00 B-07 C-D0 D-00 U-11C3 S-2000
CLR     11C3  P-1191 X-11BF Y-11C1 A-00 B-07 C-D4 D-00 U-11C3 S-2000
CLR     11C4  P-1193 X-11BF Y-11C1 A-00 B-07 C-D4 D-00 U-11C3 S-2000
LDA     11C0  P-1195 X-11BF Y-11C1 A-E8 B-07 C-D8 D-00 U-11C3 S-2000
LDB     11C2  P-1197 X-11BF Y-11C1 A-E8 B-F4 C-D8 D-00 U-11C3 S-2000
MUL           P-1198 X-11BF Y-11C1 A-DD B-20 C-D8 D-00 U-11C3 S-2000
STD     11C5  P-119A X-11BF Y-11C1 A-DD B-20 C-D8 D-00 U-11C3 S-2000
LDA     11BF  P-119C X-11BF Y-11C1 A-03 B-20 C-D0 D-00 U-11C3 S-2000
LDB     11C2  P-119E X-11BF Y-11C1 A-03 B-F4 C-D8 D-00 U-11C3 S-2000
MUL           P-119F X-11BF Y-11C1 A-02 B-DC C-D9 D-00 U-11C3 S-2000
ADDD    11C4  P-11A1 X-11BF Y-11C1 A-03 B-B9 C-D0 D-00 U-11C3 S-2000
STD     11C4  P-11A3 X-11BF Y-11C1 A-03 B-B9 C-D0 D-00 U-11C3 S-2000
BCC     11A7  P-11A7 X-11BF Y-11C1 A-03 B-B9 C-D0 D-00 U-11C3 S-2000
LDA     11C0  P-11A9 X-11BF Y-11C1 A-E8 B-B9 C-D8 D-00 U-11C3 S-2000
LDB     11C1  P-11AB X-11BF Y-11C1 A-E8 B-01 C-D0 D-00 U-11C3 S-2000
MUL           P-11AC X-11BF Y-11C1 A-00 B-E8 C-D1 D-00 U-11C3 S-2000
ADDD    11C4  P-11AE X-11BF Y-11C1 A-04 B-A1 C-D0 D-00 U-11C3 S-2000
STD     11C4  P-11B0 X-11BF Y-11C1 A-04 B-A1 C-D0 D-00 U-11C3 S-2000
BCC     11B4  P-11B4 X-11BF Y-11C1 A-04 B-A1 C-D0 D-00 U-11C3 S-2000
LDA     11BF  P-11B6 X-11BF Y-11C1 A-03 B-A1 C-D0 D-00 U-11C3 S-2000
LDB     11C1  P-11B8 X-11BF Y-11C1 A-03 B-01 C-D0 D-00 U-11C3 S-2000
MUL           P-11B9 X-11BF Y-11C1 A-00 B-03 C-D0 D-00 U-11C3 S-2000
ADDD    11C3  P-11BB X-11BF Y-11C1 A-00 B-07 C-D0 D-00 U-11C3 S-2000
STD     11C3  P-11BD X-11BF Y-11C1 A-00 B-07 C-D0 D-00 U-11C3 S-2000
```

```
00286                          ********** MOVE BLOCK **********
00287                          *
00288                          *     COPY N BYTES TO ANOTHER LOCATION
00289                          *     LET N BE 64.
00290                          *
00291                          *     SETUP:       3 LN, 10 BY,  10 CY
00292                          *     OPERATION: 7LN,  11 BY, 2+(21*32)+5=679 CY
00293                          *     TOTAL:     10 LN, 21 BY, 689 CY
00294                          *
00295                          ********************************

00297 11C7 CC   0020     3           LDD   #LENGTH/2
00298 11CA 108E 0100     4           LDY   #FROM
00299 11CE CE   0200     3           LDU   #TO

00301 11D1 4C            2           INCA              MS COUNT CORRECTION
00302 11D2 AE   A1       8 B1        LDX   ,Y++        GET TWO BYTES
00303 11D4 AF   C1       8           STX   ,U++        PUT TWO BYTES
00304 11D6 5A            2           DECB              LS COUNT
00305 11D7 26   F9       3           BNE   B1
00306 11D9 4A            2           DECA              MS COUNT
00307 11DA 26   F6       3           BNE   B1

00309 11DC 20   FE       3           BRA   *

00311           0100       FROM    EQU   $100
00312           0200       TO      EQU   $200
00313           0040       LENGTH  EQU   64
```

6.2 PROGRAM SEGMENTS

These small segments of code are less well-suited for benchmarks as they are more complex, harder to fairly define, and perhaps more dependent on the structure of an individual machine. They do represent a demonstration of useful, powerful 6809 subroutine techniques.

```
00008            000D        CR      EQU   $0D       ASCII CR




00010                        *
00011                        * COPYLN COPIES A TEXT LINE TO A NEW LOCATION
00012                        *
00013                        *      A TEXT LINE IS A SEQUENCE OF CHARS
00014                        *      ENDING WITH A CARRIAGE-RETURN
00015                        *
00016 1404 30    8D 0011   9          LEAX  FROM,PCR
00017 1408 31    8D 0022   9          LEAY  TO,PCR
00018 140C 8D    02        7          BSR   COPYLN
00019 140E 20    FE        3          BRA   *
00020                        *
00021                        *
00022 1410 A6    80        6 COPYLN   LDA   ,X+        GET A BYTE
00023 1412 A7    A0        6          STA   ,Y+        STORE IT
00024 1414 81    0D        2          CMPA  #CR        END OF LINE?
00025 1416 26    F8        3          BNE   COPYLN     NOPE, GO AGAIN
00026 1418 39              5          RTS
00027 1419       54          FROM     FCC   /THIS IS A TEXT LINE./
00028 142D       0D                   FCB   CR
00029 142E       00          TO       FCB   0,,,,,,,,,,,,,,,,,,,,,0
```

```
00032                          *
00033                          * SEARCH LOOKS FOR A PARTICULAR TEXT STRING
00034                          *      IN A BLOCK OF DATA.
00035                          *      RETURNS Z=1 IFF FOUND.
00036                          *      X POINTS AT NEXT CHAR PAST STRING.
00037                          *
00038 1443 30    8D 0038   9 START  LEAX  BLOCK,PCR  DATA BLOCK START ADDR
00039 1447 33    8D 0061   9        LEAU  END,PCR  DATA BLOCK END ADDR
00040 144B 31    8D 005E   9        LEAY  STRING,PCR  ADDR OF STRING TO BE FOUNI
00041 144F C6    05        2        LDB   #LENGTH
00042 1451 8D    02        7        BSR   SEARCH
00043 1453 20    FE        3        BRA   *

00045 1455 34    74       11 SEARCH PSHS  U,Y,X,B
00046                          *
00047                          * (SP+0) = LENGTH
00048                          * (SP+1) = RESTART BLOCK SEARCH (H)
00049                          *          RESTART BLOCK SEARCH (L)
00050                          * (SP+3) = STRING (H)
00051                          *          STRING (L)
00052                          * (SP+5) = END (H)
00053                          *          END (L)
00054 1457 AE    61        6 AGAIN  LDX   1,S
00055 1459 10AE  63        7        LDY   3,S       RESET STRING PTR
00056 145C E6    E4        4        LDB   0,S       RESET STRING LENGTH
00057                          * THIS LOOP SEARCHES AFTER MISMATCH
00058 145E AC    65        7 LOOP1  CMPX  5,S       END OF DATA?
00059 1460 2E    1A        3        BGT   EXIT      IF YES, EXIT NOT FOUND
00060 1462 A6    80        6        LDA   ,X+       GET BYTE AND INC
00061 1464 AF    61        6        STX   1,S       STORE RESTART LOCATION
00062 1466 A1    A4        4        CMPA  0,Y       SAME AS STRING?
00063 1468 26    F4        3        BNE   LOOP1     BRANCH IF NOT
00064 146A 31    21        5        LEAY  1,Y       POINT TO 2ND CHAR
00065 146C 5A              2        DECB
00066 146D 27    0D        3        BEQ   EXIT      FOR 1-BYTE SEARCH
00067                          * THIS LOOP SEARCHES AFTER MATCH
00068 146F AC    65        7 LOOP2  CMPX  5,S       END OF DATA?
00069 1471 2E    09        3        BGT   EXIT      IF YES, EXIT NOT FOUND
00070 1473 A6    80        6        LDA   ,X+       GET BYTE AND INC
00071 1475 A1    A0        6        CMPA  ,Y+       SAME AS STRING?
00072 1477 26    DE        3        BNE   AGAIN     IF NO, START OVER
00073 1479 5A              2        DECB            DONE?
00074 147A 26    F3        3        BNE   LOOP2     IF NO, KEEP GOING
00075 147C 32    67        5 EXIT   LEAS  7,S       CLEAN UF STACK
00076 147E 39              5        RTS

00078 147F       54          BLOCK  FCC   /THIS IS A BLOCK OF DATIVE /
00079 1499       44                 FCC   /DATA TO BE SEARCHED./
00080            14AC        END    EQU   *-1
00081 14AD       44          STRING FCC   /DATA /
00082            0005        LENGTH EQU   *-STRING
```

```
00085                          *
00086                          * ADDSEQ ADDS A SEQUENCE OF DECIMAL DIGITS
00087                          *     (FIRSTG + SECSTG = THIRST)
00088                          *
00089                          *  ALL PTRS ARE PAST LS BYTE OF STRING
00090                          *
00091 14B2 30   8D 0025   9          LEAX  FIRSTG+LEN,PCR  PTR TO 1ST STRING
00092 14B6 31   8D 002B   9          LEAY  SECSTG+LEN,PCR  PTR TO 2ND STRING
00093 14BA 33   8D 0031   9          LEAU  THIRST+LEN,PCR  PTR TO 3RD STRING
00094 14BE C6   0A        2          LDB   #LEN      BYTES IN STRING
00095 14C0 8D   02        7          BSR   ADDSEQ
00096 14C2 20   FE        3          BRA   *
00097                            *
00098                            *
00099 14C4 1C   FE        3 ADDSEQ CLC             CLEAR CARRY
00100 14C6 A6   82        6 DOG    LDA   ,-X       GET 2 DIGITS
00101 14C8 A9   A2        6        ADCA  ,-Y       ADD W/OTHERS
00102 14CA 19             2        DAA             MAKE DECIMAL
00103 14CB A7   C2        6        STA   ,-U       STORE 2 DIGITS
00104 14CD 5A             2        DECB            DONE? (CARRY UNAFFECTED)
00105 14CE 26   F6        3        BNE   DOG       BRANCH IF NO
00106 14D0 39             5        RTS



00108 14D1      01          FIRSTG FCB   $01,$23,$45,$99,$99
00109 14D6      00                 FCB   $00,$00,$99,$99,$99
00110 14DB      88          SECSTG FCB   $88,$76,$54,$00,$01
00111 14E0      01                 FCB   $01,$23,$45,$67,$89
00112 14E5      00          THIRST FCB   0,,,,,,,,,,0
00113           000A        LEN    EQU   10        DECIMAL DIGITS = 20
```

```
00116                          *
00117                          * SUBSEQ SUBTRACTS A SEQUENCE OF DECIMAL DIGITS (I'
00118                          *      FROM ANOTHER SEQUENCE OF DECIMAL DIGITS (IX)
00119                          *      AND STORES THE RESULT (US), ALL STRINGS
00120                          *      BEING COUNT BYTES LONG.
00121                          *
00122 14EF 30   8D 002E   9            LEAX  MINUEN+COUNT,PCR
00123 14F3 31   8D 0034   9            LEAY  SUBTRA+COUNT,PCR
00124 14F7 33   8D 003A   9            LEAU  RESULT+COUNT,PCR
00125 14FB C6   0A        2            LDB   #COUNT
00126 14FD 8D   02        7            BSR   SUBSEQ
00127 14FF 20   FE        3            BRA   *
00128                          *
00129                          *
00130 1501 1A   01        3 SUBSEQ SEC                 SET CARRY
00131 1503 34   01        5        PSHS  CC            CARRY TEMP
00132 1505 86   99        2 LOOPS  LDA   #$99          THE TEN'S COMPLEMENT
00133 1507 A0   A2        6        SUBA  ,-Y           NO CARRY POSSIBLE
00134 1509 35   01        5        PULS  CC            THE SAVED CARRY
00135 150B A9   82        6        ADCA  ,-X           DO A BINARY ADD
00136 150D 19             2        DAA                 BACK TO BCD
00137 150E 34   01        5        PSHS  CC            SAVE THE CARRY!
00138 1510 A7   C2        6        STA   ,-U           STORE THE RESULT
00139 1512 5A             2        DECB                DONE?
00140 1513 26   F0        3        BNE   LOOPS         IF NOT, GO AGAIN
00141 1515 35   81        7        PULS  CC,PC         CLEAN UP STACK, RTS



00143 1517      99          MINUEN FCB   $99,$99,$99,$99,$99
00144 151C      99                 FCB   $99,$09,$00,$00,$00
00145 1521      01          SUBTRA FCB   $01,$23,$45,$67,$00
00146 1526      99                 FCB   $99,$00,$54,$32,$11
00147 152B      00          RESULT FCB   0,,,,,,,,,0
00148           000A        COUNT  EQU   10              DECIMAL DIGITS = 20
```

```
00151 1535 30   8D 003B  9        LEAX  INPUT,PCR
00152 1539 31   8D 0047  9        LEAY  OUTPUT,PCR
00153 153D CC   0004     3        LDD   #CHARS4
00154 1540 8D   23       7        BSR   PACKS
00155 1542 20   FE       3        BRA   *



00157                             *
00158                             * PACK PUTS FOUR RIGHT-JUSTIFIED 6-BIT CHARS (IX)
00159                             *      INTO THREE PACKED 8-BIT BYTES (IY)
00160                             *
00161 1544 EC   81       8 PACK   LDD   ,X++      GET FIRST TWO CHARS
00162 1546 58            2        ASLB
00163 1547 58            2        ASLB
00164 1548 58            2        ASLB
00165 1549 49            2        ROLA
00166 154A 58            2        ASLB
00167 154B 49            2        ROLA
00168                             * HERE ACCA IS PACKED AND ACCB = XXXX0000
00169                             *
00170 154C A7   A0       6        STA   ,Y+       STORE FIRST PACKED BYTE
00171 154E A6   84       4        LDA   0,X       GET THIRD CHAR
00172 1550 44            2        LSRA
00173 1551 44            2        LSRA
00174 1552 84   0F       2        ANDA  #$0F      MAKE MS NYBBLE CLEAN
00175                             * HERE ACCB HOLDS MS NYBBLE
00176                             *      AND ACCA HOLDS LS NYBBLE
00177                             *
00178 1554 34   04       5        PSHS  B         PUT B IN STACK TEMP
00179 1556 AA   E0       6        ORA   ,S+       NOW TOGETHER, CLEAN STACK
00180 1558 A7   A0       6        STA   ,Y+       STORE SECOND PACKED BYTE
00181 155A EC   81       8        LDD   ,X++      GET LAST TWO CHARS
00182                             *
00183                             * PICK UP 2 LSB FROM ACCA AS 2 MSB IN ACCB
00184 155C 58            2        ASLB
00185 155D 58            2        ASLB
00186 155E 44            2        LSRA
00187 155F 56            2        RORB
00188 1560 44            2        LSRA
00189 1561 56            2        RORB
00190 1562 E7   A0       6        STB   ,Y+       STORE THIRD PACKED BYTE
00191 1564 39            5        RTS
```

```
00194                          *
00195                          * PACKS TAKES 4 * ACCD 6-BIT CHARS (IX) AND PACKS
00196                          *      THEM INTO 3 * ACCD 8-BIT BYTES (IY)
00197                          *
00198 1565 4C              2 PACKS  INCA            ADJUST COUNT MS BYTE
00199 1566 34    06        6        PSHS  D         COUNT ON THE STACK
00200 1568 8D    DA        7 PAC1   BSR   PACK      PACK 4 INTO 3
00201 156A 6A    61        7        DEC   1,S       LS COUNT
00202 156C 26    FA        3        BNE   PAC1
00203 156E 6A    E4        6        DEC   0,S       MS COUNT
00204 1570 26    F6        3        BNE   PAC1
00205 1572 35    86        8        PULS  D,PC      CLEAN UP STACK, RETURN


00207 1574       50          INPUT  FCC   /PACK THESE CHARS/
00208            0004        CHARS4 EQU   *-INPUT/4
00209 1584       00          OUTPUT FCB   0,,,,,,,,,,,0
```

```
00212 1590 30   8D FFF0  9         LEAX  IN,PCR
00213 1594 31   8D 0045  9         LEAY  OUT,PCR
00214 1598 CC   0004     3         LDD   #BYTES3
00215 159B 34   20       6         PSHS  Y
00216 159D 8D   2F       7         BSR   UNPAKS
00217 159F A6   A2       6 TOASC   LDA   ,-Y       GET A CHAR
00218 15A1 85   20       2         BITA  #$20      IF B5 NOT...
00219 15A3 26   04       3         BNE   TO1       ...THEN B6
00220 15A5 8A   40       2         ORA   #$40      (INTO ASCII)
00221 15A7 A7   A4       4         STA   ,Y
00222 15A9 10AC E4       7 TO1     CMPY  ,S        DONE GOING BACK?
00223 15AC 22   F1       3         BHI   TOASC
00224 15AE 32   62       5         LEAS  2,S
00225 15B0 20   FE       3         BRA   *



00227                              *
00228                              * UNPACK RETURNS THREE PACKED 8-BIT BYTES (IX)
00229                              *      INTO FOUR RIGHT-JUSTIFIED 6-BIT CHARS (IY)
00230                              *
00231 15B2 34   06       6 UNPACK  PSHS  D         SAVE ACCD
00232 15B4 EC   80       7         LDD   ,X+       GET 1ST + 2ND BYTES
00233 15B6 44            2         LSRA            :
00234 15B7 56            2         RORB            : 16-BIT SHIFT, TWO PLACES
00235 15B8 44            2         LSRA            :
00236 15B9 56            2         RORB            :
00237                              * HERE ACCA IS AN UNPACKED BYTE
00238                              *
00239 15BA 54            2         LSRB
00240 15BB 54            2         LSRB
00241                              * NOW ACCB IS ALSO UNPACKED
00242                              *
00243 15BC ED   A1       8         STD   ,Y++      STORE 1ST + 2ND CHARS
00244 15BE EC   80       7         LDD   ,X+       GET 2ND + 3RD BYTES
00245 15C0 58            2         ASLB            :
00246 15C1 49            2         ROLA            : ANOTHER SHIFT, TWO PLACES
00247 15C2 58            2         ASLB            :
00248 15C3 49            2         ROLA            :
00249 15C4 84   3F       2         ANDA  #$3F      CLEAR TOP TWO BITS
00250                              * HERE ACCA IS UNPACKED
00251                              *
00252 15C6 E6   80       6         LDB   ,X+       GET 3RD BYTE AGAIN
00253 15C8 C4   3F       2         ANDB  #$3F
00254                              * NOW BOTH ARE UNPACKED
00255 15CA ED   A1       8         STD   ,Y++      STORE 3RD + 4TH CHARS
00256 15CC 35   86       8         PULS  D,PC      RECOVER ACCD, RETURN
```

```
00259                          *
00260                          * UNPAKS TAKES 3 * ACCD 8-BIT BYTES (IX) AND PUTS
00261                          *      6-BIT CHARS INTO 4 * ACCD BYTES
00262                          *
00263 15CE 4C              2 UNPAKS INCA            ADJUST CTR MS BYTE
00264 15CF 34   06         6        PSHS  D         COUNT ON THE STACK
00265 15D1 8D   DF         7 UNP1   BSR   UNPACK    UNPACK 3 INTO 4
00266 15D3 6A   61         7        DEC   1,S       LS COUNT
00267 15D5 26   FA         3        BNE   UNP1
00268 15D7 6A   E4         6        DEC   0,S       MS COUNT
00269 15D9 26   F6         3        BNE   UNP1
00270 15DB 35   86         8        PULS  D,PC      CLEAN UP STACK, RETURN


00272           1584         IN     EQU   OUTPUT
00273           0004         BYTES3 EQU   CHARS4
00274 15DD      00           OUT    FCB   0,0,0,,,,,,,,,,,,,,,0,0,0
```

```
00277            0400        DELTA0 EQU   $400     START OF DELTA0 TABLE


00279                        *
00280                        * SUB-LINEAR STRING SEARCH
00281                        *
00282 15F1 30    8D 0061   9 SETUP  LEAX  TEXT,PCR START OF TEXT STRING
00283 15F5 31    8D 0089   9        LEAY  TEXTEN,PCR  END OF TEXT STRING
00284 15F9 34    30        8        PSHS  Y,X
00285 15FB 31    8D 004F   9        LEAY  PAT,PCR  START OF PATTERN
00286 15FF CE    0400      3        LDU   #DELTA0  POINT AT OFFSET TABLE
00287 1602 86    08        2        LDA   #PATLEN  GET PATTERN LENGTH (.LE. 255!)
00288 1604 34    62        9        PSHS  U,Y,A
00289 1606 8D    04        7        BSR   SLSS
00290 1608 32    69        5        LEAS  9,S
00291 160A 20    FE        3        BRA   *
00292                        *
00293                        *  BOYER + MOORE, "A FAST STRING SEARCHING
00294                        *     ALGORITHM"  COMM. ACM  VOL.20 NO.10,
00295                        *     OCT. '77  PP.762-772.
00296                        *
00297                        *
00298                        * (SP+0) = RETURN (H)
00299                        *           RETURN (L)
00300                        * (SP+2) = PATLEN
00301                        * (SP+3) = PAT (H)
00302                        *           PAT (L)
00303                        * (SP+5) = DELTA0 (H)
00304                        *           DELTA0 (L)
00305                        * (SP+7) = TEXT (H)
00306                        *           TEXT (L)
00307                        * (SP+9) = TEXTEN (H)
00308                        *           TEXTEN (L)
00309                        *
00310                        *  INITIALIZE DELTA0 TABLE
00311 160C A6    62        5 SLSS   LDA   2,S       GET PATTERN LENGTH
00312 160E C6    80        2        LDB   #128      TABLE SIZE
00313 1610 A7    C0        6 SE1    STA   ,U+       )
00314 1612 5A              2        DECB             ) FILL TABLE WITH
00315 1613 26    FB        3        BNE   SE1        )    PATTERN LENGTH
00316                        *
00317                        *  FINISH DELTA0 OFFSET TABLE
00318 1615 E6    62        5        LDB   2,S       GET PATTERN LENGTH
00319 1617 EE    65        6        LDU   5,S       POINT AT DELTA0 TABLE
00320 1619 5A              2 SE2    DECB
00321 161A A6    A0        6        LDA   ,Y+       GET A CHAR
00322 161C 84    7F        2        ANDA  #$7F      MASK MSB
00323 161E E7    C6        5        STB   A,U       STORE COUNT AT DELTA0 (CHAR)
00324 1620 5D              2        TSTB
00325 1621 26    F6        3        BNE   SE2
```

```
00328 1623 31   3F       5         LEAY  -1,Y     WENT PAST!
00329 1625 10AF 63       7         STY   3,S      SAVE END-OF-PATTERN
00330 1628 AE   67       6         LDX   7,S      START OF TEXT STRING
00331 162A 4F            2         CLRA
00332 162B E6   62       5         LDB   2,S      PATTERN LENGTH
00333                      *
00334                      * (IX) = START OF 'TEXT STRING'
00335                      *      AND WILL SEARCH 'TEXT STRING'
00336                      * (US) = THE DELTA0 TABLE
00337                      * (IY) = LAST CHAR OF 'PATTERN'
00338                      *      AND WILL DECR AS MATCH IS FOUND
00339                      *
00340 162D 5A            2          DECB
00341 162E 30   85       5 FAST    LEAX  B,X      POINT AT NEXT TRY IN TEXT
00342 1630 AC   69       7         CMPX  9,S      PAST THE END OF TEXT?
00343 1632 22   19       3         BHI   NOTFND   YES, NOT FOUND (Z=0)
00344 1634 E6   84       4         LDB   0,X      GET CHAR INTO B
00345 1636 E6   C5       5         LDB   B,U      GET DELTA0 OF CHAR
00346 1638 26   F4       3         BNE   FAST     BRANCH IF NOT SAME
00347                      *
00348                      * DELTA1 (CHAR) = 0 IFF CHAR = PAT (PATLEN)
00349                      *      HERE B IS OBVIOUSLY ZERO, SO. . .
00350                      *
00351 163A 5C            2 SLOW    INCB           ONE MATCH ALREADY
00352 163B E1   62       5         CMPB  2,S      GOT ENOUGH MATCHES?
00353 163D 24   0C       3         BHS   FOUND    YES, RETURN FOUND
00354 163F A6   82       6         LDA   ,-X      GET ANOTHER CHAR
00355 1641 A1   A2       6         CMPA  ,-Y      IS IT MATCHED?
00356 1643 27   F5       3         BEQ   SLOW     IF YES, GO SLOW
00357 1645 5C            2         INCB           PAST ORIGINAL MATCH
00358 1646 10AE 63       7         LDY   3,S      : END-OF-PATTERN
00359 1649 20   E3       3         BRA   FAST     :  (RESET IY)
00360                      *
00361                      *
00362 164B 1A   04       3 FOUND   ORCC  #$04     RETURN Z=1
00363 164D 39            5 NOTFND  RTS



00365 164E       50        PAT     FCC   /PATTERN /
00366            0008      PATLEN  EQU   *-PAT
00367 1656       20        TEXT    FCC   / A STERN EXAMPLE OF A /
00368 166C       50                FCC   /PATTERN SEARCH IN TEXT./
00369            1682      TEXTEN  EQU   *-1
```

```
00372 1683 CC   0005     3         LDD   #LONG/2
00373 1686 30   8D 0019  9         LEAX  ORIGIN,PCR
00374 168A 31   8D 001F  9         LEAY  DEST,PCR
00375 168E 8D   02       7         BSR   DCPY
00376 1690 20   FE       3         BRA   *


00378                               *
00379                               * DCPY COPIES 2*ACCD BYTES FROM (IX+) TO (IY+)
00380                               *
00381 1692 4C            2 DCPY    INCA              MS COUNT CORRECTION
00382 1693 34   06       6         PSHS  D           SAVE D
00383 1695 EC   81       8 DC1     LDD   ,X++        GET TWO BYTES
00384 1697 ED   A1       8         STD   ,Y++        PUT TWO BYTES
00385 1699 6A   61       7         DEC   1,S         COUNT LS BYTE
00386 169B 26   F8       3         BNE   DC1
00387 169D 6A   E4       6         DEC   0,S         COUNT MS BYTE
00388 169F 26   F4       3         BNE   DC1
00389 16A1 35   86       8         PULS  D,PC        CLEAN STACK, RETURN



00391 16A3      01         ORIGIN FCB   1,1,2,3,4,5,6,7,8,9
00392 16AD      00         DEST   FCB   0,,,,,,,,,,0
00393           000A       LONG   EQU   10
```

6.3 SYSTEM EXAMPLE -- MTEST

MTEST is a nice, fast (proportional to N rather than N^2) memory test system. The package has self-contained I/O routines, is completely position-independent, and uses no absolute RAM (all parameters and temporary variables exist on the stack).

Note the use of LEA to point at text strings in a position-independent manner. Note also the use of a branch table near the start of the program which allows external access to internal subroutines. This allows MTEST to be updated without requiring changes in code that may use MTEST subroutines. And note that the I/O routines use absolute values on the stack to point at I/O devices. By using a PROM to set up these values (and the stack pointer itself), the same code can be used in a large number of diverse systems.

The User Stack Pointer is used to mark the original top of the stack (the stack bottom for this system) so that temporary locations may be accessed with similiar offsets from different subroutine levels. The stack mark technique also allows the unstructured system-abort technique which requires no knowledge of present subroutine level to completely clean up the stack.

```
00001                                   NAM   MTEST9
00003                           *
00004                           * COPYRIGHT (C) 1978 MOTOROLA INC, AUSTIN, TX
00005                           *  MPU SYSTEMS DESIGN, T. F. RITTER
00006                           *  3.0/01/27/78/TFR
00007                           *  3.1/03/08/78/TFR+WMK


00009                           *     MTEST9 IS A FAST MEMORY TEST SYSTEM.  IT HAS
00010                           *  SELF-CONTAINED I/O, IS COMPLETELY POSITION-
00011                           *  INDEPENDENT, AND USES NO ABSOLUTE RAM.  IT MAY
00012                           *  BE PLACED IN UNDER 1K OF ROM.
00013                           *
00014                           *     MTEST9 IS ENTERED AT ITS FIRST LOCATION,
00015                           *  AND ASKS FOR START/STOP ADDRESSES FOR THE
00016                           *  TEST.  THE LAST FOUR HEX CHARS BEFORE <CR>
00017                           *  ARE ACCUMULATED; A NULL ENTRY PRESERVES THE
00018                           *  ORIGINAL ADDRESSES.  IF AN 'M' IS ENTERED,
00019                           *  MTEST9 WILL COPY ITSELF INTO A NEW LOCATION
00020                           *  BEGINNING AT THE CURRENT START ADDRESS, AND
00021                           *  RESTART AT THAT ADDRESS.
00022                           *
00023                           *     MTEST9 STORES A SEQUENCE OF BYTES THROUGH-
00024                           *  OUT THE MEMORY TEST AREA, THEN COMPARES THE
00025                           *  RECOVERED SEQUENCE TO THE INTERNALLY-GENERATED
00026                           *  SEQUENCE.  ANY ERRORS CAUSE DISPLAY OF THE
00027                           *  ERROR ADDRESS AND THE BITS IN ERROR; ALL STUCK
00028                           *  BITS AND IMPROPER ADDRESS-DECODE ERRORS CAN
00029                           *  BE FOUND, AND SOME PATTERN-SENSITIVITIES ARE
00030                           *  ALSO EXERCISED.  AN 'X' IS PRINTED FOR EACH
00031                           *  PASS THROUGH MEMORY;  EIGHT X'S IS A FUNCTIONAL
00032                           *  TEST,  AND 'ALL DONE!' WILL PRINT AFTER THE
00033                           *  FULL SEQUENCE OF 211 PASSES; THEN MTEST9 WILL
00034                           *  START OVER.  AN <ESC> ALWAYS RESTARTS MTEST9;
00035                           *  <CONTROL X> ALWAYS RETURNS TO THE CALLING
00036                           *  SYSTEM (MAID, IN THE EXORCISOR).
00037                           *
00038                           *     A SHORT INITIALIZATION ROUTINE IS USED TO
00039                           *  CONFIGURE MTEST9 FOR THE EXORCISOR; CONTROL
00040                           *  THEN FALLS INTO M0, WHICH IS THE GENERAL TEST
00041                           *  SYSTEM.  DIFFERENT HARDWARE CONFIGURATIONS
00042                           *  NEED ONLY SET UP THE STACK, PUSH A ZERO MODE
00043                           *  BYTE, PUSH THE ABSOLUTE ADDRESSES OF THE ACIA
00044                           *  CONTROL AND DATA PORTS, THEN CALL TVM0 AT
00045                           *  MTEST+3.  ALTERNATELY, PUSHING A NON-ZERO
00046                           *  MODE BYTE AND ABSOLUTE ADDRESSES OF INPUT AND
00047                           *  OUTPUT ROUTINES WILL ALLOW ALL I/O TO BE DONE
00048                           *  EXTERNALLY (NOTICE THE SPECIAL PARAMETER
00049                           *  REQUIREMENTS OF INCH: ACCA IS SENT TO INCH AS
00050                           *  A PARAMETER.  IFF B7 OF ACCA IS 0, INCH WILL
00051                           *  WAIT FOR A NEW CHAR.  IFF ACCA=0, INCH WILL
00052                           *  ECHO CHAR TO OUTCH.  INCH RETURNS THE RECOVERED
00053                           *  CHAR IN ACCA.)
```

```
00056            F11E        MAID    EQU   $F11E     REENTRY ADDRESS
00057            000D        CR      EQU   $0D       ASCII CR
00058            0D0A        CRLF    EQU   $0D0A     ASCII CRLF
00059            0018        CTLX    EQU   $18       ASCII CANCEL (CONTROL X)
00060            001B        ESC     EQU   $1B       ASCII ESCAPE
00061            0020        SPACE   EQU   $20       ASCII SPACE
00062            0024        STACKS  EQU   $24       STACK AREA (MAX SIZE)


00064            FCF4        ACIAC   EQU   $FCF4     ACIA CONTROL REGISTER
00065            FCF5        ACIAD   EQU   $FCF5     ACIA DATA REGISTER


00067                        *  CONDITION CODE BITS
00068            0080        E       EQU   $80
00069            0040        F       EQU   $40
00070            0020        H       EQU   $20
00071            0010        I       EQU   $10
00072            0008        N       EQU   $08
00073            0004        Z       EQU   $04
00074            0002        V       EQU   $02
00075            0001        C       EQU   $01

00077                        *  CONDITION CODE BITS (NOT)
00078            007F        NE      EQU   $7F
00079            00BF        NF      EQU   $BF
00080            00DF        NH      EQU   $DF
00081            00EF        NI      EQU   $EF
00082            00F7        NN      EQU   $F7
00083            00FB        NZ      EQU   $FB
00084            00FD        NV      EQU   $FD
00085            00FE        NC      EQU   $FE
```

```
00088 0400                              ORG    $0400     POSITION INDEPENDENT
00089 0400 16   003F      5 MTEST   LBRA   M2

00091                                   *
00092                                   *  TRANSFER VECTORS
00093                                   *
00094 0403 16   004D      5 TVM0    LBRA   M0         GENERAL PURPOSE ENTRY
00095 0406 16   028D      5 TVINIA  LBRA   INITAC     INIT. ACIA
00096 0409 16   0281      5 TVGCH   LBRA   GCH        GET PRESENT CHAR IN ACCA
00097 040C 16   0300      5 TVINCH  LBRA   INCH       A=0 FOR ECHO, BIT7=0 FOR WAIT
00098 040F 16   02F7      5 TVINNP  LBRA   INCHNP     CHAR W/O PARITY IN ACCA
00099 0412 16   031D      5 TVIN1H  LBRA   IN1H       CHR IN A, HEX IN B, NEG IF BAD
00100 0415 16   0207      5 TVINAD  LBRA   INADDR     GET CHARS UNTIL NON-HEX
00101 0418 16   01AB      5 TVBEGE  LBRA   BEGEND     GET ADDRESSES IN 0,X - 3,X


00103 041B 16   0241      5 TVOUT   LBRA   OUT        SEND CHAR FROM ACCA NOW
00104 041E 16   0283      5 TVOUTC  LBRA   OUTCH      SEND CHAR WHEN READY
00105 0421 16   029A      5 TVHEXL  LBRA   CHEXL      CONVERT ACCA MSN TO HEX (ASCII)
00106 0424 16   029B      5 TVHEXR  LBRA   CHEXR      CONVERT RIGHT NYBBLE
00107 0427 16   02A5      5 TVOUT2  LBRA   OUT2H      SEND 2 HEX (IX)
00108 042A 16   02A0      5 TVOUT4  LBRA   OUT4H      SEND 4 HEX (IX)
00109 042D 16   02AF      5 TVPDAT  LBRA   PDATA      SEND CRLF, DATA ...
00110 0430 16   02AE      5 TVPDA1  LBRA   PDATA1     SEND DATA ...THRU MSB=1
00111 0433 16   02B8      5 TVPCRL  LBRA   PCRLF      SEND CRLF NULLS
00112 0436 16   02C9      5 TVREP   LBRA   REPEAT     SEND ACCA, B TIMES
00113 0439 16   02C4      5 TVRSR   LBRA   RSPACE     SEND SPACE, B TIMES


00115 043C 16   0125      5 TVPRIN  LBRA   PRINBI     SEND ACCB AS BINARY
00116 043F 16   0116      5 TVRAND  LBRA   RAND       PSEUDO-RANDOM ACCA




00118                                   *
00119                                   *  M2 CONFIGURES FOR EXORCISOR
00120                                   *
00121                                   *     (ANOTHER SYSTEM MIGHT INITIALIZE
00122                                   *     THE STACK, STACK I/O ADDRESSES,
00123                                   *     THEN CALL TVM0 IN A SMALL PROM).
00124                                   *
00125 0442 32   8C BB      7 M2      LEAS   MTEST,PCR  STACK BELOW PROGRAM
00126 0445 6F   E2         8         CLR    ,-S        INTERNAL I/O MODE
00127 0447 CE   FCF4      3         LDU    #ACIAC
00128 044A 108E FCF5      4         LDY    #ACIAD
00129 044E 8E   F11E      3         LDX    #MAID
00130 0451 34   70       10         PSHS   U,Y,X      ABSOLUTES ON STACK
00131                                   *  FALL INTO THE GENERAL-PURPOSE PACKAGE M0
```

```
00134 0453 1F   43       6 M0     TFR   S,U       MARK STACK
00135 0455 32   77       5        LEAS  -9,S      AREA FOR TEMP GLOBALS


00137                             *
00138                             *  EQUATES ARE RELATIVE POSITION
00139                             *     FROM USER STACK POINTER.
00140                             *
00141            0006             MODE   EQU   6          I/O SELECT (0 MEANS ACIA AD
00142            0004             CIAC   EQU   4          ACIA CONTROL
00143            0004             INSUB  EQU   4          GET CHAR IN A
00144            0002             CIAD   EQU   2          ACIA DATA
00145            0002             OUTSUB EQU   2          SEND CHAR FROM A
00146            0000             MAI    EQU   0          MAID RETURN
00147            FFFF             SEED   EQU   -1         STARTING PSEUDO-RANDOM VALUI
00148            FFFE             FLAG   EQU   -2         ERROR HEADING PRINTED? FLAG
00149            FFFD             XCOUNT EQU   -3         NO. OF X'S ON LINE
00150            FFFB             ENDAD  EQU   -5         END ADDRESS
00151            FFF9             BEGAD  EQU   -7         BEGIN ADDRESS
00152            FFF8             NUCH   EQU   -8         NEW CHAR (ESCAPE TEST)
00153            FFF7             OLCH   EQU   -9         OLD CHAR


00155                             *
00156                             *  VERIFY PROGRAM CORRECTNESS
00157 0457 17   0300     9 M1     LBSR  VERPGM
00158                             *
00159                             *  INITIALIZE ACIA
00160 045A 17   FFA9     9        LBSR  TVINIA
00161                             *
00162                             *  PRINT PROGRAM ID
00163 045D 30   8D 0068  9        LEAX  MSG1,PCR POINT AT MSG1
00164 0461 17   027B     9        LBSR  PDATA    PRINT IT
00165                             *
00166                             *  PRINT PROGRAM LOCATION
00167 0464 30   8C 99    7        LEAX  MTEST,PCR
00168 0467 17   02E7     9        LBSR  PRNTIX
00169 046A 86   2D       2        LDA   #'-
00170 046C 17   0235     9        LBSR  OUTCH
00171 046F 30   8D 0319  9        LEAX  PGMEND,PCR
00172 0473 17   02DB     9        LBSR  PRNTIX
00173                             *
00174                             *  GET ADDRESSES
00175 0476 17   010F     9        LBSR  GETAD
00176                             *
00177                             *  INITIALIZE
00178 0479 86   01       2 TST5   LDA   #1
00179 047B A7   5F       5        STA   SEED,U   SEED VALUE
00180 047D A7   5D       5        STA   XCOUNT,U CRLF ON NEXT X
00181 047F 6F   5E       7        CLR   FLAG,U   NO HEADING YET
00182                             *
00183                             *  STORE PSEUDO-RANDOM SEQUENCE
00184 0481 A6   5F       5 TESTM  LDA   SEED,U   GET SEED
00185 0483 AE   59       6        LDX   BEGAD,U
```

```
00186 0485 30   1F       5          LEAX  -1,X     DEX
00187 0487 30   01       5 TST1     LEAX  1,X      NEXT LOCATION
00188 0489 17   00CC     9          LBSR  RAND     DO PSEUDO-RANDOM IN A
00189 048C A7   84       4          STA   0,X
00190 048E AC   5B       7          CMPX  ENDAD,U  ALL DONE ENTRY SWEEP?
00191 0490 2D   F5       3          BLT   TST1     NO, GO AGAIN
00192                               *
00193                               *  CHECK RECOVERED SEQUENCE
00194 0492 A6   5F       5          LDA   SEED,U   GET SEED AGAIN
00195 0494 AE   59       6          LDX   BEGAD,U
00196 0496 30   1F       5          LEAX  -1,X
00197 0498 30   01       5 TST2     LEAX  1,X
00198 049A 17   00BB     9          LBSR  RAND
00199 049D E6   84       4          LDB   0,X      SAVE CHAR FROM MEM
00200 049F 34   04       5          PSHS  B        :  CBA
00201 04A1 A1   E0       6          CMPA  ,S+      :
00202 04A3 27   03       3          BEQ   TST3
00203 04A5 17   005F     9          LBSR  ERR
00204 04A8 AC   5B       7 TST3     CMPX  ENDAD,U  ALL DONE CHECK SWEEP?
00205 04AA 2D   EC       3          BLT   TST2     NO, GO AGAIN
00206                               *
00207                               *  DO PASSES UNTIL END OF SEQUENCE
00208 04AC 17   0048     9          LBSR  PRNTX    SINGLE PASS DONE
00209 04AF 1027 FFA0     6          LBEQ  M0       OUT IFF ESC
00210 04B3 A6   5F       5          LDA   SEED,U   :
00211 04B5 17   00A0     9          LBSR  RAND     :  UPDATE SEED
00212 04B8 A7   5F       5          STA   SEED,U   :
00213 04BA 81   01       2          CMPA  #1       END OF PSEUDO-RANDOM SEQUENCE
00214 04BC 26   C3       3          BNE   TESTM    DO ANOTHER PASS
00215                               *
00216                               *  PRINT DONE, THEN START OVER
00217 04BE 30   8D 002B  9          LEAX  ENDM,PCR POINT AT END MESSAGE
00218 04C2 17   021C     9          LBSR  PDATA1
00219 04C5 30   59       5          LEAX  BEGAD,U
00220 04C7 20   B0       3          BRA   TST5     ANOTHER COMPLETE TEST
00221                               *
00222                               *
00223 04C9      0D0A               MSG1  FDB   CRLF
00224 04CB      50                       FCC   /PSEUDO-RANDOM MEMORY TEST 3.1 AT /
00225 04EC      A4                       FCB   $A4      $ W/MSB=1
00226 04ED      41                 ENDM  FCC   /ALL DONE!/
00227 04F6      A0                       FCB   $A0
```

```
00230                          *
00231                          *  PRINT AN X FOR EACH PATTERN-TEST
00232                          *
00233                          *  BLOWS A
00234                          *
00235 04F7 6A   5D        7 PRNTX  DEC   XCOUNT,U LINE FULL?
00236 04F9 26   07        3        BNE   PR1       NO, NEED NO CRLF
00237 04FB 86   40        2        LDA   #64       CRLF IMPLIES NEW CHAR CNT
00238 04FD A7   5D        5        STA   XCOUNT,U
00239 04FF 17   01EC      9        LBSR  PCRLF
00240 0502 86   58        2 PR1    LDA   #'X
00241 0504 16   019D      5        LBRA  OUTCH     PRINT X



00243                          *
00244                          *
00245                          *  ERR PRINTS DATA FOUND IN ERROR
00246                          *    IX (2,S) = LOCATION OF ERROR
00247                          *  ACCB (1,S) = VALUE READ FROM MEMORY
00248                          *  ACCA (0,S) = PSEUDO-RANDOM VALUE
00249                          *
00250 0507 34   16        8 ERR    PSHS  X,B,A
00251 0509 86   01        2        LDA   #1        :  CRLF ON NEXT X
00252 050B A7   5D        5        STA   XCOUNT,U  :
00253 050D 6D   5E        7        TST   FLAG,U    ERROR HEADING PRINTED?
00254 050F 26   09        3        BNE   E1        YES, DON'T PRINT AGAIN!
00255 0511 6C   5E        7        INC   FLAG,U    REMEMBER, "IT'S PRINTED!"
00256 0513 30   8D 001A   9        LEAX  HDR,PCR   POINT AT HEADER MSG
00257 0517 17   01C5      9        LBSR  PDATA
00258 051A 17   01D1      9 E1     LBSR  PCRLF
00259 051D 30   62        5        LEAX  2,S       POINT AT SAVED X
00260 051F 17   01AB      9        LBSR  OUT4H     PRINT ADDRESS
00261 0522 C6   03        2        LDB   #3
00262 0524 17   01D9      9        LBSR  RSPACE
00263 0527 E6   61        5        LDB   1,S       MEMORY VALUE
00264 0529 8D   39        7        BSR   PRINBI    PRINT MEMORY VALUE
00265 052B E8   E4        4        EORB  0,S       DESIRED VALUE
00266 052D 8D   35        7        BSR   PRINBI    PRINT ERRORS AS 1'S
00267 052F 35   96       10        PULS  A,B,X,PC GET SAVED REGS, RET
00268                          *
00269                          *
00270 0531      0D0A             HDR    FDB   CRLF
00271 0533      41               FCC   /ADDRESS    READS
00272 054B      42               FCC   /BIT-IN-ERROR/
00273 0557      A0               FCB   $A0
```

```
00276                          *
00277                          *  RAND GENERATES A 211 BYTE
00278                          *     SEQUENCE IN ACCA
00279                          *
00280 0558 34   02        5 RAND   PSHS  A         SAVE CURRENT #
00281 055A 46             2        RORA
00282 055B 46             2        RORA
00283 055C 46             2        RORA
00284 055D A8   E4        4        EORA  0,S
00285 055F 46             2        RORA
00286 0560 35   02        5        PULS  A         GET SAVED #, CLEAN STACK
00287 0562 46             2        RORA            ROTATED, 1 BIT CHANGED
00288 0563 39             5        RTS



00290                          *
00291                          *  PRINBI OUTPUTS THE VALUE IN B
00292                          *     AS BINARY ASCII, MSB FIRST,
00293                          *     THROUGH ACCA, TO SUBROUTINE OUTCH
00294                          *
00295 0564 34   06        6 PRINBI PSHS  B,A       SAVE STATE
00296 0566 86   08        2        LDA   #$08
00297 0568 34   02        5        PSHS  A         SAVE BIT COUNTER
00298 056A 58             2 T1     LSLB            GET NEXT BIT
00299                          *  BRANCH IF CARRY A ONE
00300 056B 25   04        3        BCS   T2
00301 056D 86   30        2        LDA   #'0
00302 056F 20   02        3        BRA   T3
00303 0571 86   31        2 T2     LDA   #'1
00304 0573 17   012E      9 T3     LBSR  OUTCH     SEND IT
00305 0576 86   20        2        LDA   #SPACE
00306 0578 17   0129      9        LBSR  OUTCH
00307 057B 6A   E4        6        DEC   0,S       COUNT IT
00308                          *  BRANCH IF NOT A WHOLE BYTE DONE
00309 057D 26   EB        3        BNE   T1
00310 057F 35   02        5        PULS  A         CLEAN UP COUNTER
00311 0581 C6   02        2        LDB   #2        2 SPACES
00312 0583 17   017A      9        LBSR  RSPACE
00313 0586 35   86        8        PULS  A,B,PC    RECOVER STATE, RET
```

```
00316                          *
00317                          *  GETAD GETS ADDRESSES INTO 0,X - 3,X.
00318                          *     GOES AGAIN IF TEST WOULD OVERWRITE
00319                          *     MTEST9.
00320                          *
00321 0588 30   59        5 GETAD  LEAX  BEGAD,U
00322 058A 17   0039      9        LBSR  BEGEND
00323 058D 30   8D 01FB   9        LEAX  PGMEND,PCR  END OF MTEST
00324 0591 AC   59        7        CMPX  BEGAD,U
00325 0593 25   18        3        BLO   OK        TESTING AFTER MTEST

00327 0595 30   8D FE67   9        LEAX  MTEST,PCR  START OF MTEST
00328 0599 30   88 DC     6        LEAX  -STACKS,X  ENCLOSE THE STACK
00329 059C AC   59        7        CMPX  BEGAD,U
00330 059E 23   04        3        BLS   NOPE      TESTING INSIDE MTEST!

00332 05A0 AC   5B        7        CMPX  ENDAD,U
00333 05A2 22   09        3        BHI   OK
00334 05A4 30   8D 0006   9 NOPE   LEAX  DANMSG,PCR  DANGER MESSAGE!
00335 05A8 17   0134      9        LBSR  PDATA
00336 05AB 20   DB        3        BRA   GETAD

00338 05AD 39             5 OK     RTS


00340 05AE      44          DANMSG FCC   /DON'T OVERWRITE MTEST9!/
00341 05C5      A0                 FCB   $A0




00343                          *
00344                          *  BEGEND GETS BEGIN AND END
00345                          *     ADDRESSES FROM KEYBOARD
00346                          *     AND PUTS THEM IN RAM (IX).
00347                          *     BEGIN .LE. END OR TRYS AGAIN.
00348                          *
00349                          *       (X):(X+1) = BEGIN
00350                          *     (X+2):(X+3) = END
00351                          *
00352 05C6 34   06        6 BEGEND PSHS  B,A       SAVE STATE
00353 05C8 8D   1F        7        BSR   INST      GET BEGIN ADDR
00354 05CA 30   02        5        LEAX  2,X
00355 05CC 8D   32        7        BSR   INFIN     GET END ADDR
00356 05CE EC   84        5        LDD   0,X
00357 05D0 A3   83        9        SUBD  ,--X      BEGIN .LE. END?
00358 05D2 25   F2        3        BLO   BEGEND
00359 05D4 35   86        8        PULS  A,B,PC    RECOVER STATE, RET
```

```
00361                          *
00362                          *  ASLM4 SHIFTS TWO BYTES (X):(X+1)
00363                          *     LEFT FOUR PLACES
00364                          *
00365 05D6 34    02        5 ASLM4  PSHS  A          SAVE A
00366 05D8 86    04        2        LDA   #4         SHIFT COUNT
00367 05DA 68    01        7 AS1    LSL   1,X        :    16-BIT SHIFT
00368 05DC 69    84        6        ROL   0,X        :
00369 05DE 4A              2        DECA              DONE?
00370 05DF 26    F9        3        BNE   AS1        AGAIN GO IF NO
00371 05E1 35    82        7        PULS  A,PC       RECOVER A, RETURN


00373                          *
00374                          *  INST PRINTS START MESSAGE AND COLLECTS
00375                          *     A HEX ADDRESS (X):(X+1).  ASCII
00376                          *     CR RETURNS, OTHER NON-HEX STARTS OVER.
00377                          *     M TRANSFERS CONTROL TO SUBROUTINE MOVE.
00378                          *
00379                          *     BLOWS A,B
00380                          *
00381 05E3 81    4D        2 INS1   CMPA  #'M
00382 05E5 1027 005D       6        LBEQ  MOVE
00383 05E9 34    10        6 INST   PSHS  X
00384 05EB 30    8D 0022   9        LEAX  STARTM,PCR  ENTER HERE
00385 05EF 17    00ED      9        LBSR  PDATA
00386 05F2 35    10        6        PULS  X
00387 05F4 17    0028      9        LBSR  INADDR
00388 05F7 26    EA        3        BNE   INS1      : AGAIN IFF BAD
00389 05F9 39              5        RTS             :   HEX .NE. CR
00390                          *
00391                          *  INFIN PRINTS END MESSAGE AND
00392                          *     COLLECTS A HEX ADDRESS (X):(X+1).
00393                          *     ASCII CR RETURNS, OTHER NON-HEX STARTS OVER.
00394                          *     M TRANSFERS CONTROL TO SR MOVE.
00395                          *
00396                          *     BLOWS A,B
00397                          *
00398 05FA 81    4D        2 INF1   CMPA  #'M
00399 05FC 1027 0046       6        LBEQ  MOVE

00401 0600 34    10        6 INFIN  PSHS  X
00402 0602 30    8D 0013   9        LEAX  FINM,PCR ENTER HERE
00403 0606 17    00D6      9        LBSR  PDATA
00404 0609 35    10        6        PULS  X
00405 060B 17    0011      9        LBSR  INADDR    : AGAIN IFF BAD
00406 060E 26    EA        3        BNE   INF1      :   HEX .NE. CR
00407 0610 39              5        RTS

00409 0611       20          STARTM FCC   / BEGIN:/
00410 0618       A0                 FCB   $A0
00411 0619       20          FINM   FCC   / END:/
00412 061E       A0                 FCB   $A0
```

```
00415                          *
00416                          *  INADDR INPUTS HEX ADDRESS FROM KEYBOARD
00417                          *     UNTIL NON-HEX.  RETURNS NON-HEX IN
00418                          *     ACCA AND ALSO Z=1 IFF CR.
00419                          *     LAST 4 CHARS ARE COLLECTED
00420                          *     IN BINARY AT 0,X AND 1,X.
00421                          *
00422                          *     BLOWS A,B
00423                          *
00424 061F 34    10        6 INADDR PSHS  X
00425 0621 17    00A9      9        LBSR  OUT4H    PRESENT ADDRESS
00426 0624 C6    02        2        LDB   #2
00427 0626 17    00D7      9        LBSR  RSPACE
00428 0629 35    10        6        PULS  X
00429 062B 17    0104      9        LBSR  IN1H     GET CHAR IN A, HEX IN B
00430 062E 2B    13        3        BMI   INA2     RETURN IFF NOT HEX
00431 0630 6F    84        6        CLR   0,X      INITIALIZE ADDR=0
00432 0632 6F    01        7        CLR   1,X
00433 0634 20    05        3        BRA   INA3     IS HEX, SO ACCUMULATE
00434 0636 17    00F9      9 INA1   LBSR  IN1H     GET CHAR IN A, HEX IN B
00435 0639 2B    08        3        BMI   INA2     RETURN IFF NOT HEX
00436 063B 8D    99        7 INA3   BSR   ASLM4    MAKE A PLACE
00437 063D EA    01        5        ORB   1,X      CATENATE HEX
00438 063F E7    01        5        STB   1,X
00439 0641 20    F3        3        BRA   INA1
00440 0643 81    0D        2 INA2   CMPA  #CR      RETURN Z=1 IFF CR
00441 0645 39              5        RTS


00443                          *
00444                          *  MOVE RE-POSITIONS MTEST9 TO BEGIN ADDRESS!
00445                          *
00446 0646 30    8D FDB6   9 MOVE   LEAX  MTEST,PCR  START OF MTEST
00447 064A 34    40        6        PSHS  U
00448 064C EE    59        6        LDU   BEGAD,U  MOVE-TO LOCATION
00449 064E 108E 038D       4        LDY   #PGMEND+1-MTEST  LENGTH OF MTEST
00450 0652 A6    80        6 MO1    LDA   ,X+      GET A BYTE
00451 0654 A7    C0        6        STA   ,U+      MOVE IT
00452 0656 31    3F        5        LEAY  -1,Y     COUNT IT
00453 0658 26    F8        3        BNE   MO1      BRANCH IF NOT DONE
00454 065A 35    40        6        PULS  U
00455 065C 6E    D8 F9     8        JMP   [BEGAD,U]  RE-INITIALIZE EVERYTHING


00457                          *
00458                          *  OUT SENDS CHAR NOW
00459                          *
00460 065F A7    D8 02     9 OUT    STA   [CIAD,U]
00461 0662 39              5        RTS
```

```
00464                          *
00465                          *  CKESC CHECKS FOR ESCAPE OR CONTROL X
00466                          *     ESC RETURNS Z=1 (RESTART MEMTEST)
00467                          *     CTLX = RETURN (JUMP TO MAID)
00468                          *
00469 0663 34    04        5 CKESC  PSHS  B
00470 0665 E6    58        5        LDB   NUCH,U    SET UP...
00471 0667 E7    57        5        STB   OLCH,U    SOFTWARE EDGE-DETECTOR
00472 0669 1E    89        7        EXG   A,B
00473 066B 86    80        2        LDA   #$80      GET CHAR NOW
00474 066D 17    009F      9        LBSR  INCH      (NO ECHO)
00475 0670 1E    89        7        EXG   A,B
00476 0672 C4    7F        2        ANDB  #$7F
00477 0674 E7    58        5        STB   NUCH,U
00478 0676 C1    18        2        CMPB  #CTLX
00479 0678 26    03        3        BNE   CK1       ABORT MTEST9 PACKAGE?
00480 067A 32    C4        4        LEAS  0,U       FUNNY TFR U,S
00481 067C 39              5        RTS             RETURN TO CALLING SYSTEM

00483 067D C1    1B        2 CK1    CMPB  #ESC      IS THE NEW CHAR ESC?
00484 067F 26    0A        3        BNE   CK2
00485                          *  HERE ACCB = NUCH = ESC
00486 0681 E1    57        5        CMPB  OLCH,U    THE OLD CHAR ALSO ESC?
00487 0683 27    06        3        BEQ   CK2       RESTART IFF FIRST ESC CHAR
00488 0685 1F    34        6        TFR   U,S       ABSOLUTES REMAIN
00489 0687 6E    8D FDC8   8        JMP   M0,PCR    FUNNY LBRA M0

00491 068B 35    84        7 CK2    PULS  B,PC      RETURN NO ACTION


00493                          *
00494                          *  GCH GETS PRESENT CHAR INTO ACCA
00495                          *
00496 068D A6    D8 02     9 GCH    LDA   [CIAD,U] GET THE CHAR NOW!
00497 0690 A1    D8 02     9        CMPA  [CIAD,U] STILL THE SAME?
00498 0693 26    F8        3        BNE   GCH       IF NOT, GO AGAIN
00499 0695 39              5        RTS


00501                          *
00502                          *  INITIALIZE ACIA
00503                          *
00504 0696 34    02        5 INITAC PSHS  A
00505 0698 86    03        2        LDA   #3        RESET ACIA
00506 069A A7    D8 04     9        STA   [CIAC,U]
00507                          *
00508                          *     X X X  X X X  0 1   DIVIDE BY 16
00509                          *     X X X  1 0 1  X X   8 DATA + 1 STOP
00510                          *     0 0 0  X X X  X X   READER OFF, BOTH
00511                          *                           INTERRUPTS DISABLED
00512 069D 86    15        2        LDA   #%00010101
00513 069F A7    D8 04     9        STA   [CIAC,U]
00514 06A2 35    82        7        PULS  A,PC
```

```
00517                          *
00518                          *  OUTCH WAITS TILL ACIA IS READY
00519                          *     THEN SENDS A TO ACIA
00520                          *
00521                          *     CHANGES A
00522                          *
00523 06A4 8A   80        2 OUTCH  ORA   #$80      DONT SEND MSB
00524 06A6 6D   46        7        TST   MODE,U
00525 06A8 27   03        3        BEQ   OUTC2
00526 06AA 6E   D8 02     8        JMP   [OUTSUB,U]  ALTERNATE I/O
00527 06AD 34   04        5 OUTC2  PSHS  B          SAVE B IN STACK
00528 06AF E6   D8 04     9 OUTC1  LDB   [CIAC,U] ACIA CONTROL
00529 06B2 C4   02        2        ANDB  #2         CHECK XMIT STATUS
00530 06B4 27   F9        3        BEQ   OUTC1      LOOP IF XMIT NOT READY
00531 06B6 A7   D8 02     9        STA   [CIAD,U] ACIA DATA
00532 06B9 17   FFA7      9        LBSR  CKESC
00533 06BC 35   84        7        PULS  B,PC       RECOVER B, RETURN


00535                          *
00536                          *  CHEXL MAKES LEFT NYBBLE ACCA
00537                          *     ASCII HEX
00538                          *  CHEXR MAKES RIGHT NYBBLE ACCA
00539                          *     ASCII HEX
00540                          *
00541                          *     BLOWS A
00542                          *
00543 06BE 44             2 CHEXL  LSRA             LEFT NYBBLE BECOMES RIGHT
00544 06BF 44             2        LSRA
00545 06C0 44             2        LSRA
00546 06C1 44             2        LSRA
00547 06C2 84   0F        2 CHEXR  ANDA  #$F        RIGHT NYBBLE ONLY
00548 06C4 8B   30        2        ADDA  #$30       OFFSET TO ASCII 0
00549 06C6 81   39        2        CMPA  #'9        LARGER THAN ASCII 9?
00550 06C8 23   02        3        BLS   CHEX1
00551 06CA 8B   07        2        ADDA  #7         ADDITIONAL OFFSET TO ASCII A-
00552 06CC 39             5 CHEX1  RTS


00554                          *
00555                          *  OUT4H DOES OUT2H TWICE
00556                          *  OUT2H SENDS (IX) AS
00557                          *     2 ASCII HEX CHARS.
00558                          *
00559                          *     BLOWS A, MOVES X
00560                          *
00561 06CD 8D   00        7 OUT4H  BSR   OUT2H     2H X 2 = 4H
00562 06CF A6   80        6 OUT2H  LDA   ,X+
00563 06D1 34   02        5        PSHS  A          SAVE A
00564 06D3 8D   E9        7        BSR   CHEXL      GET MS BYTE
00565 06D5 17   FFCC      9        LBSR  OUTCH      SEND IT
00566 06D8 35   02        5        PULS  A
00567 06DA 8D   E6        7        BSR   CHEXR      GET LS BYTE
00568 06DC 16   FFC5      5        LBRA  OUTCH      SEND IT
```

```
00571                          *
00572                          *  PDATA PRINTS CRLF, TEXT STRING
00573                          *     (IX) IS START
00574                          *     B7 = 1 IS LAST CHAR PRINTED
00575 06DF 8D   0D          7 PDATA  BSR   PCRLF
00576                          *  FALL INTO PDATA1


00578                          *
00579                          *  PDATA1 PRINTS TEXT STRING
00580 06E1 34   12          7 PDATA1 PSHS  A,X        SAVE STATE
00581 06E3 A6   84          4 PD1    LDA   ,X         GET A CHAR
00582 06E5 17   FFBC        9        LBSR  OUTCH      SEND IT
00583 06E8 6D   80          8        TST   ,X+        TEST MSB
00584 06EA 2A   F7          3        BPL   PD1        ANOTHER CHAR IF B7=0
00585 06EC 35   92          9        PULS  A,X,PC     RECOVER STATE, RETURN


00587                          *
00588                          *  PRINT CRLF
00589 06EE 34   10          6 PCRLF  PSHS  X          SAVE PRESENT X
00590 06F0 30   8D 0004     9        LEAX  TCRLF,PCR  POINT AT CRLF TEXT
00591 06F4 8D   EB          7        BSR   PDATA1     PRINT IT
00592 06F6 35   90          8        PULS  X,PC       RECOVER STATE, RETURN

00594 06F8      0D0A          TCRLF  FDB   CRLF
00595 06FA      00                   FCB   0,,,$80


00597                          *
00598                          *  PSPACE PRINTS ONE SPACE
00599                          *  RSPACE PRINTS B SPACES
00600                          *  REPEAT PRINTS ACCA, B TIMES
00601                          *
00602                          *     THESE ALL BLOW A,B
00603                          *
00604 06FE C6   01          2 PSPACE LDB   #1         SET COUNT TO 1
00605 0700 86   20          2 RSPACE LDA   #SPACE     LOAD A WITH ASCII SPACE
00606 0702 17   FF9F        9 REPEAT LBSR  OUTCH      PRINT ACCA
00607 0705 5A               2        DECB             DONE?
00608 0706 26   FA          3        BNE   REPEAT     LOOP TILL DONE
00609 0708 39               5        RTS              RETURN
00610                          *
00611                          *  INCHNP GETS A CHAR (NO PARITY)
00612                          *
00613 0709 4F               2 INCHNP CLRA             SET UP ECHO, WAIT FOR CHAR
00614 070A 8D   03          7        BSR   INCH       GET BYTE FROM ACIA
00615 070C 84   7F          2        ANDA  #$7F       CLEAR BIT7
00616 070E 39               5        RTS
```

```
00619                                  *
00620                                  *  INCH RETURNS CHAR IN ACCA.
00621                                  *     ECHOS IFF OLD ACCA=0.
00622                                  *     WAITS FOR CHAR IFF OLD ACCA B7=0
00623                                  *
00624 070F 6D   46          7 INCH     TST    MODE,U
00625 0711 27   03          3          BEQ    INCH3
00626 0713 6E   D8 04       8          JMP    [INSUB,U]
00627 0716 4D               2 INCH3    TSTA
00628 0717 2A   03          3          BPL    INCH4
00629 0719 16   FF71        5          LBRA   GCH
00630 071C 34   02          5 INCH4    PSHA
00631 071E A6   D8 04       9 INCH1    LDA    [CIAC,U] ACIA STATUS
00632                                  *  IFF DATA READY, B0=1
00633 0721 44               2          LSRA             B0 INTO CARRY
00634 0722 24   FA          3          BCC    INCH1
00635 0724 A6   E0          6          LDA    ,S+       SNEAKY PULL
00636 0726 26   06          3          BNE    INCH2     SHALL WE ECHO?
00637 0728 A6   D8 02       9          LDA    [CIAD,U]
00638 072B 16   FF76        5          LBRA   OUTCH

00640 072E A6   D8 02       9 INCH2    LDA    [CIAD,U] DATA INTO ACCA
00641 0731 39               5          RTS


00643                                  *
00644                                  *  IN1H WAITS FOR NEW CHAR FROM ACIA IN ACCA,
00645                                  *     THEN TRANSLATES CHAR TO HEX IN ACCB.
00646                                  *     IN1H RETURNS NEG IFF NOT HEX.
00647                                  *
00648 0732 8D   D5          7 IN1H     BSR    INCHNP    WAIT FOR CHAR AND ECHO
00649 0734 1F   89          6          TFR    A,B
00650 0736 17   FF2A        9          LBSR   CKESC
00651                                  *
00652                                  *  CHECK AND CONVERT FOR VALID HEX CHAR
00653 0739 C1   30          2 CMPINP   CMPB   #'0
00654 073B 25   11          3          BLO    INBAD     BAD IF UNDER ASCII 0
00655 073D C1   39          2          CMPB   #'9
00656 073F 23   0A          3          BLS    INGD      GOOD IF 0-9
00657 0741 C1   41          2          CMPB   #'A
00658 0743 25   09          3          BLO    INBAD     BAD IF BETWEEN 9,A
00659 0745 C1   46          2          CMPB   #'F
00660 0747 22   05          3          BHI    INBAD     BAD IF OVER F
00661 0749 C0   07          2          SUBB   #7        LETTERS TO BINARY
00662 074B C4   0F          2 INGD     ANDB   #$F       RETURN POS IFF GOOD
00663 074D 39               5          RTS
00664 074E 1A   08          3 INBAD    ORCC   #N        RETURN NEG IFF BAD
00665 0750 39               5          RTS
```

```
00668                          *
00669                          *  PRINTIX PRINTS THE VALUE IN X
00670                          *     AS 4 HEX DIGITS
00671                          *
00672 0751 34    10        6 PRNTIX PSHS  X         SAVE X
00673 0753 1F    41        6        TFR   S,X       POINT AT SAVED X
00674 0755 17    FF75      9        LBSR  OUT4H     PRINT IT
00675 0758 35    90        8        PULS  X,PC      RECOVER X, RETURN.



00677                          *
00678                          *  VERPGM VERIFYS PROGRAM CORRECTNESS BY
00679                          *     COMPUTING PARITY OVER ENTIRE PGM
00680                          *     (PGM HAS BEEN MADE ODD PARITY)
00681                          *
00682 075A 30    8D 002E   9 VERPGM LEAX  PGMEND,PCR  LAST ADDRESS
00683 075E 34    10        6        PSHS  X         (PARITY BYTE)
00684 0760 30    8D FC9C   9        LEAX  MTEST,PCR
00685 0764 4F              2        CLRA
00686 0765 A8    80        6 VER1   EORA  ,X+
00687 0767 AC    E4        6        CMPX  0,S       DONE?
00688 0769 23    FA        3        BLS   VER1
00689 076B 32    62        5        LEAS  2,S       CLEAN UP STACK
00690 076D 4C              2        INCA             ODD PARITY NOW 0'S
00691 076E 27    09        3        BEQ   VER2      NORMAL RETURN
00692 0770 30    8D 0006   9        LEAX  VERMSG,PCR
00693 0774 17    FF68      9        LBSR  PDATA
00694 0777 1F    34        6        TFR   U,S       RETURN TO MAIN SYSTEM
00695 0779 39              5 VER2   RTS

00697 077A       49          VERMSG FCC   /INVALID PGM LOAD!/
00698 078B       A0                 FCB   $A0


00700 078C       93          PGMEND FCB   $93       ODD PARITY BYTE
```

```
00702             0000                END
```

TOTAL ERRORS   00000
TOTAL WARNINGS 00000

7.0 PROGRAMMING TRICKS 'N TREATS

7.1 INSTRUCTION EQUIVALENTS

```
JMP     0,X          =        TFR     X,PC

                     =        PSHS    X
                              PULS    PC

                     =        PSHS    X
                              RTS

LBRA    CAT          =        JMP     CAT,PCR

LBRA    *+5          =        JMP     2,PC

LBSR    DOG          =        JSR     DOG,PCR

LDX     #PIG         ≈        LEAX    PIG,PCR
     ↑                                 ↑
the loaded value will not       the loaded value will
change when executed in         change when executed in
different locations             different locations

PSHS    A                            STA     ,-S
(shorter)                            (affects flags)
PULS    A                            LDA     ,S+
(shorter)                            (affects flags)
RTI                  ≈        PULS    ALL

                     =        TST     0,S
                              BMI     RAT
                              PULS    CC,PC
                         RAT  PULS    ALL
```

7.1 (Continued)

```
RTS                 =           PULS    PC

SEX                 =           CLRA
                                TSTB
                                BPL     COW
                                DECA
                        COW     EQU     *

                    =           CLRA
                                TSTB
                                BPL     BULL
                                COMA
                        BULL    EQU     *

                    =           PSHS    X
                                LDX     #0
                                LEAX    B,X
                                TFR     X,D
                                PULS    X

SWI                 ≈           PSHS    ALL
                                JMP     [$FFF8]

                    =           PSHS    ALL
                                LDX     POSUM,PCR
                                STX     10,S
                                JMP     [$FFF8]
                        POSUM   EQU     *
TFR     Y,X         =           LEAX    0,Y
                                (shorter, may affect flags)
```

7.2 COMPATIBLE MACROS

7.2.1 Monadic:

```
ASLD            =       ASLB
                        ROLA

TSTA            ≠       CLC

CLRD            =       LDD     #0

CLRX            =       LDX     #0

DBNE    MOOSE   =       DECB
                        BNE     MOOSE

DDBN    MOUSE   =       DECB
                        BNE     MOUSE
                        DECA
                        BNE     MOUSE

DECD            ≈       TSTB
                        BNE     ROACH
                        DECA
                ROACH   DECB

                ≈       EXG     D,X
                        LEAX    -1,X
                        EXG     D,X

                ≈≈      TSTB
                        BNE     DE1
                        DECA
                DE1     DECB
                        BNE     DE2
                        TSTA
                DE2     EQU     *
```

7.2.1 (Continued)

```
        INCD                    ~               INCB
                                                BNE     COON
                                                INCA
                                        COON    EQU     *

                                ~               EXG     D,X
                                                LEAX    1,X
                                                EXG     D,X

        JMP     [[0,X]]         ~               BRA     DBLIND

                                        DBLIND  LDX     0,X
                                                LDX     0,X
                                                JMP     0,X

        LDDP    #VALU           =               EXG     A,DP
                                                LDA     #VALU
                                                EXG     A,DP

        LDPC    CHICK           =               JMP     [CHICK]

        LEAPC   EEL             =               JMP     EEL

        LSRD                    ~               LSRA
                                                RORB
```

7.2.1 (Continued)

```
        NEGD                    ≠               COMA
                                                NEGB
                                                ADCA    #0

                                ≈               COMA
                                                COMB
                                                ADDD    #1

                                =               STD     BEE
                                                COM     BEE
                                                COM     BEE+1
                                                INC     BEE+1
                                                BNE     BONNET
                                                INC     BEE
                                    BONNET      BVS     ERR
                                                LDD     BEE

        NEGX                    =               EXG     D,X
                                                COMA
                                                COMB
                                                ADDD    #1
                                                EXG     D,X

        STDP    DILLO           =               EXG     A,DP
                                                STA     DILLO
                                                EXG     A,DP

        TGC (toggle carry)      =       C       EQU     $01
                                                PSHS    A
                                                TFR     CC,A
                                                EORA    #C
                                                TFR     A,CC
                                                PULS    A
```

7.2.1 (Continued)

```
TGC                     =       C       EQU     $01
                                NOTC    EQU     $FE
                                        BCC     TOAD
                                        ANDCC   #NOTC
                                        BRA     FROG
                                TOAD    ORCC    #C
                                FROG    EQU     *
```

7.2.2 Dyadic:

```
ADDB    A               =               PSHS    A
(B←B+A)                                 ADDB    ,S+

ADDD    X               ≠               ADDD    0,X
(D←D+X)                 ≠               ADDD    ,X

                        =               PSHS    X
                                        ADDD    ,S++

ADDX    D               =               LEAX    D,X
(X←X+D)

ADDX    Y               ≠               ADDX    ,Y
(X←X+Y)

                        =               EXG     D,Y
                                        LEAX    D,X
                                        EXG     D,Y

ANDA    B               =               PSHS    B
(A←A ∧ B)                               ANDA    ,S+

ANDB    A               =               PSHS    A
(B←B ∧ A)                               ANDB    ,S+
```

7.2.2 (Continued)

BITA B	=	PSHS	B
(TEMP←A ∧ B)		BITA	,S+
CMPA B	=	PSHS	B
(TEMP←A-B)		CMPA	,S+
CMPB A	=	PSHS	A
(TEMP←B-A)		CMPB	,S+
CMPX Y		PSHS	Y
(TEMP←X-Y)		CMPX	,S++
EXG A,X	=	PSHS	A,B
		TFR	X,D
(A←XH)		PULS	A
(X←A:XL)		TFR	D,X
		PULS	B
EXG B,X	=	PSHS	A
		PSHS	B
(B←XL)		TFR	X,D
(X←XH:B)		PULS	B
		TFR	D,X
		PULS	A
JMP X,PC	≈	TFR	PC,D
		LEAX	D,X
(PC←PC+X)		TFR	X,PC
(destroys D,X)			

7.2.2 (Continued)

LDDP	#ADDR	=	EXG A,DP LDA #ADDR EXG A,DP
LEAD	SOW,X	=	EXG X,D LEAX SOW,X EXG X,D
(D←EA,EA=X+SOW)			
LEAP	GUPPY,X	=	PSHS X,PC LEAX GUPPY,X STX 2,S PULS X,PC
(PC←EA,EA=X+GUPPY)			
SUBD	X	≠	SUBD ,X
(D←D-X)			
		=	PSHS X SUBD ,S++
SUBX	D	=	PSHS D COMA COMB ADDD #1 LEAX D,X PULS D
(X←X-D)			

7.2.2 (Continued)

SUBX Y (X←X-Y)	≠	SUBX ,Y
	=	PSHS D TFR Y,D COMA COMB ADDD #1 LEAX D,X PULS D
TFR A,X (X←A:XL)	=	PSHS X STA 0,S PULS X
TFR B,X (X←XH:B)	=	PSHS X STB 1,S PULS X

7.3 PROGRAM FLOW MANIPULATIONS

- error return
  (return to a different location if error--return with offset)

```
RTO         =           PSHS    D
                        LDD     2,S
                        ADDD    #OFFSET
                        STD     2,S
                        PULS    D,PC

            =           PSHS    X
                        LDX     2,S
                        LEAX    OFFSET,X
                        STX     2,S
                        PULS    X,PC

            ≈           PULS    X
                        JMP     OFFSET,X

            ≈           INC     1,S
                        BNE     ANT
    (if offset = 2)     INC     0,S
                ANT     INC     1,S
                        BNE     EATER
                        INC     0,S
                EATER   RTS
```

7.3 (Continued)

- pass parameters in-line

```
(destroys X)  LEAX      RTN,PCR
              PSHS      X
              LBRA      SUB
              FCB       MOO
              FCB       MEOW
              FCB       CRUNCH
        RTN   EQU       *
```

- alternately

```
              LBSR      SUB
              BRA       NXT
              FCB       OINK
              FCB       WOOF
              FCB       SQUEEK
        NXT   EQU       *
```

- pass parameters on stack

```
(destroys X,A) LDX      #CRT
               LDA      #TYPE
               PSHS     X,A
               LBSR     SUB2
               LEAS     3,S
```

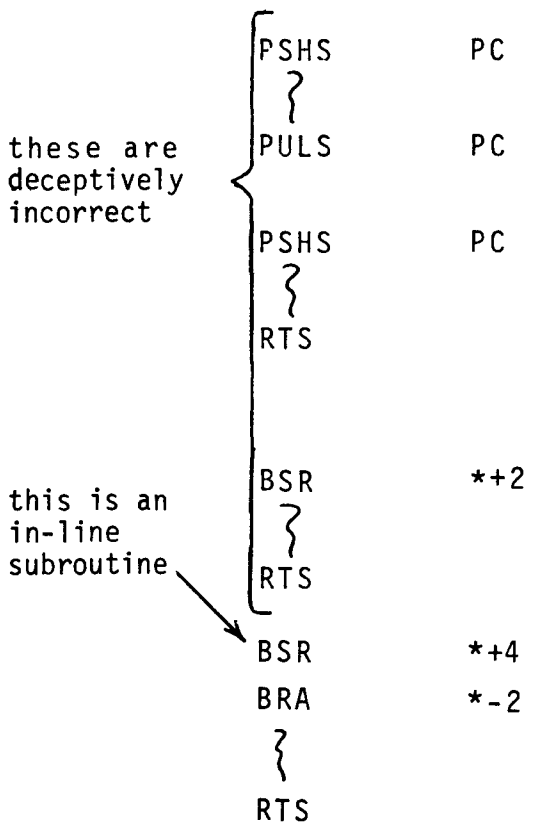
7.3 (Continued)

- subroutine skips past in-line arguments after operating - system "interrupt"

```
WAY1        LDX         7,S         RETURN PC
            LEAX        B,X         COMPUTED OFFSET
            STX         7,S
            PULS        ALL

WAY2        LDX         7,S         RETURN PC
            LEAX        OFFSET,X    FIXED OFFSET
            STX         7,S
            RTI
```

- alternate forms for loop construction



7.3 (Continued)

- pass parameters in-line

```
(destroys X)  LEAX      RTN,PCR
              PSHS      X
              LBRA      SUB
              FCB       MOO
              FCB       MEOW
              FCB       CRUNCH
        RTN   EQU       *
```

- alternately

```
              LBSR      SUB
              BRA       NXT
              FCB       OINK
              FCB       WOOF
              FCB       SQUEEK
        NXT   EQU       *
```

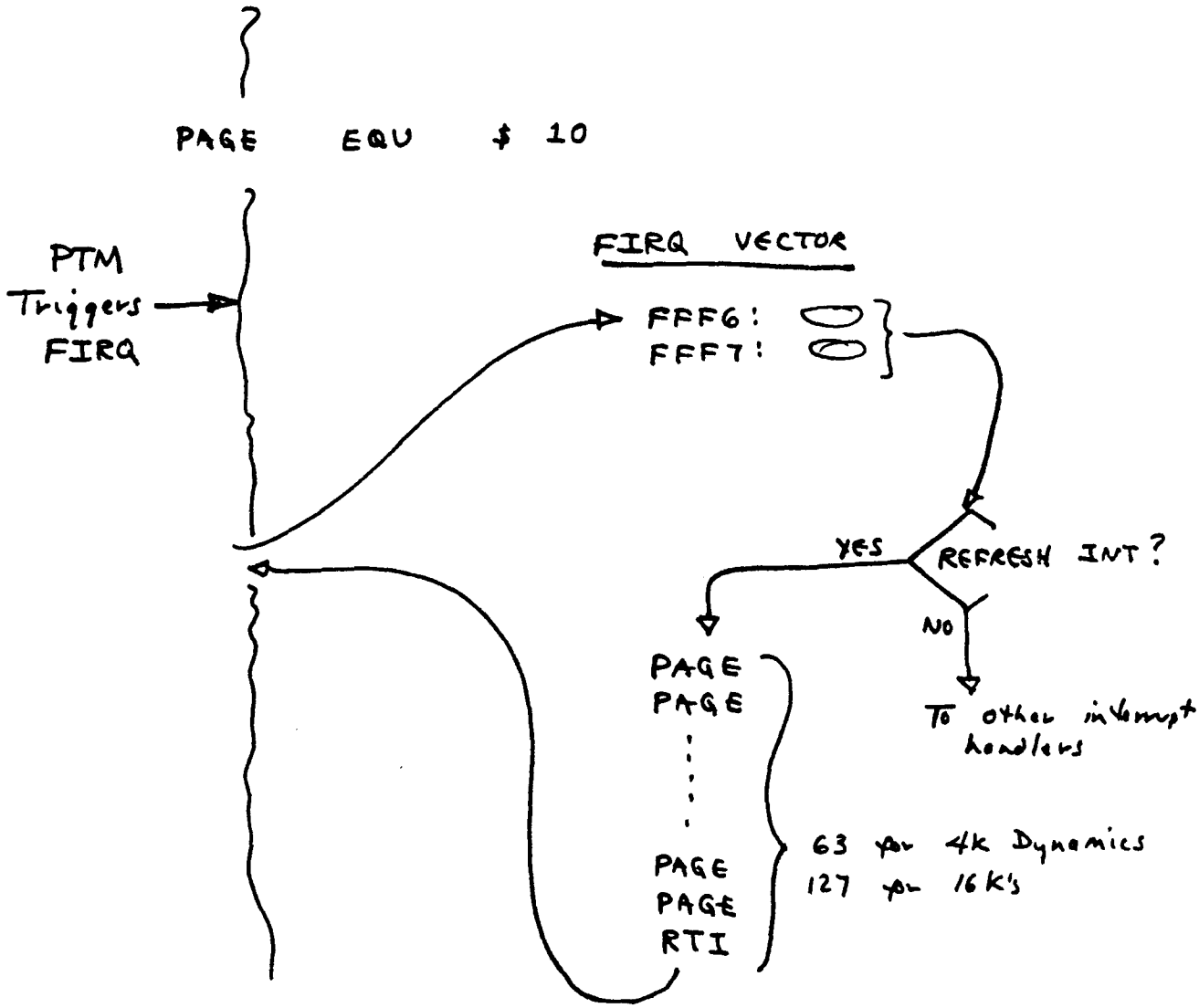
- pass parameters on stack

```
(destroys X,A) LDX      #CRT
               LDA      #TYPE
               PSHS     X,A
               LBSR     SUB2
               LEAS     3,S
```

7.4 PROGRAMMING HINTS: Wise And Other Whys

Go to co-routine	=	EXG	X,PC
Call operating system	=	SWI	
		FCB	SQUID
		FCB	WHALE
		}	
		FCB	GNAT
double exchange top-of-stack	≈	LDD	2,S
		LDX	0,S
		STX	2,S
		STD	0,S
ACCD exchange top-of-stack	≈	LDX	0,S
		STD	0,S
		TFR	X,D
point to PC-relative table	=	LEAX	21,PC
	=	LEAX	CAT,PCR
add top top bytes on stack and push result	=	LDA	,S+
		ADDA	,S
		STA	,S
exchange PC with top-of-stack	=	JSR	[,S++]

Refresh Dynamic Memory



7.6 SOFTWARE DOCUMENTATION STANDARDS FOR 6809

1. Each subroutine should have an associated header block containing at least the following elements:

   a) A full specification for this subroutine - including associated data structures - such that from this description alone replacement code can be generated.

   b) All usage of memory resources must be defined, including:

      i) All RAM needed from Temporary (local) storage used during execution of this subroutine or called subroutines).

      ii) All RAM needed for Permanent storage (used to transfer values from one execution of the subroutine to future executions).

      iii) All RAM accessed as Global Storage (used to transfer values from or to higher-level subroutines).

      iv) All possible exit-state conditions, if these are to be used by calling routines to test occurrences internal to the subroutine.

2. Code internal to each subroutine should have sufficient associated line-comments to help in understanding the code.

3. All code must be non-self-modifying and position-independent.

4. Each subroutine which includes a loop must be separately documented by flow-chart.

5. The main program should be executable starting at the first location and should include an I/O jump table immediately thereafter.

6. When any single routine begins to approach the length of one listing page, it becomes candidate for further subroutining.

7.7 ADDITIONAL TRICKS 'N TREATS

7.7.1 Instruction Equivalents

```
LEAX  ,--X     =        LEAX  -2,X

LEAX  ,--Y     =        LEAY  -2,Y
                        TFR    Y,X

LEAX  ,X++     =        LEAX   2,X

LEAX  ,Y++     =        TFR    Y,X
                        LEAX   2,X

NOP            =        TFR    X,X
               =        LEAX   0,X
```

7.7.2 Monadic Compatible Macros

```
ABSA           =        TSTA
                        BPL    AB1
                        NEGA
                 AB1    EQU    *

AAX            =        EXG    A,B
                        ABX
                        EXG    A,B

                        NEGA
                        NEGB
                        SBCA   #0
```

MOTOROLA ***Semiconductor Products Inc.***

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721 • A SUBSIDIARY OF MOTOROLA INC.

13157 PRINTED IN USA 10-79 IMPERIAL LITHO B82406