7200 MICROPROCESSOR

MACRO INSTRUCTION SET DESCRIPTION

8/3/72

INSTRUCTION SET

I. General Description

The virtual machine described herein is a dual-accumulator, dual-index-register, multi-address-mode, byte-oriented structure. Most arithmetic and logical operations are performed on 8-bit bytes between memory and one of the accumulators, but several 32-bit memory-to-memory arithmetic operations and a limited number of 16-bit operations are also available. Facility is also provided for packing and unpacking 4-bit digits to and from bytes. Bit testing is facilitated by special "Test Under Mask" instructions. A 32-byte hardware stack is available for general program use, and is also used to hold subroutine and interrupt returns. The head (top two bytes) of the stack may be used as an autoincrement register to facilitate subroutine parameter pickup and sequential list operations. Memory is expandable to 65K bytes, with all locations simultaneously addressable.

All operations can be performed by pure code; that is, it is never necessary to modify instructions; hence, they can be located in Read-Only Memory.

A flexible, multi-level interrupt system and I/O instructions which can address up to 256 different devices allow sophisticated I/O operation while minimizing interface complexity and timing.

Provisions are made for a general purpose console operating on the Data Exchange Bus which is capable of loading and examining registers and memory, running in single-instruction mode, executing instructions from the console switch register, etc. Virtually all timing and control for these operations is provided by the processor itself.

II. Programmable Registers

1) Accumulators

Two 8-bit accumulators, A and B, are used for general data manipulation. Most binary operations (Add, Subtract, etc.) are performed with one of the operands in an accumulator and leave the result in the accumulator.

2) Index Registers

Two 16-bit index, or base registers, X and Y, may be used for addresses and bases. They may be loaded, stored, incremented and tested, and a displacement byte may be added to them. They contain the addresses of operands for all 32-bit operations, and may be used as base registers for all 8-, and 16-bit memory reference instructions.

3) General Stack

The General Stack, GS, is a 32-byte first-in last-out stack implemented as a 32-byte high-speed RAM and a 5-bit Stack Pointer Counter (SP) . Before a byte is stored in the Stack (pushed onto the GS), the Stack Pointer is incremented to point to the next free location in the RAM prior to the store operation. After a byte is read from the RAM (popped off of the GS), the SP is decremented to point to the byte which was pushed onto the stack immediately prior to the one that was read. Thus, pushing a byte onto the GS appends it to a list, and popping a byte deletes it from that list.

The Stack Pointer is a polynomial counter, and therefore is not "incremented" and "decremented" in the usual binary sequence. The counting sequence is listed below:

Ø	- Ø Ø Ø Ø Ø	9	- Ø Ø 1 1 Ø
1	- 1 Ø Ø Ø Ø	1Ø	- 1 Ø Ø 1 1
2	- Ø 1 Ø Ø Ø	11	- 1 1 Ø Ø 1
3	- Ø Ø 1 Ø Ø	12	- 1 1 1 Ø Ø
4	- 1 Ø Ø 1 Ø	13	- 1 1 1 1 Ø
5	- Ø 1 Ø Ø 1	14	- 1 1 1 1 1
6	- 1 Ø 1 Ø Ø	15	- Ø 1 1 1 1
7	- 1 1 Ø 1 Ø	16	- Ø Ø 1 1 1
8	- Ø 1 1 Ø 1	17	- Ø Ø Ø 1 1

18 - 1 Ø Ø Ø 1	25 - 1 Ø 1 1 1
19 - 1 1 Ø Ø Ø	26 - Ø 1 Ø 1 1
20 - Ø 1 1 Ø Ø	27 - 1 Ø 1 Ø 1
21 - 1 Ø 1 1 Ø	28 - Ø 1 Ø 1 Ø
22 - 1 1 Ø 1 1	29 - Ø Ø 1 Ø 1
23 - 1 1 1 Ø 1	3Ø - Ø Ø Ø 1 Ø
24 - Ø 1 1 1 Ø	31 - Ø Ø Ø Ø 1

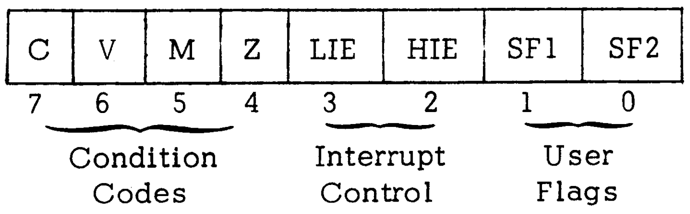
When a 16-bit quantity is pushed onto the GS, the low-order 8 bits are first pushed, and then the high-order 8 bits are pushed. When a 16-bit quantity is popped, the bytes are retrieved in the opposite order; that is, the first byte popped from the stack (the most recently pushed) is the high-order 8 bits, and tne second byte popped is the low-order 8 bits.

The top (most recently entered) two bytes of the GS are called the Stack Head, SH. The SH may be used as an autoincrement register to provide the address for most 8- and 16-bit memory reference instructions. When autoincrement addressing is specified, the SH is popped off of the stack and stored in a temporary storage register and used to provide the memory address to fetch the data byte(s). It is then incremented by the number of bytes fetched and pushed back onto the GS.

The SH may also be swapped with X and Y, incremented and decremented by one or two, and a displacement byte may be added to it, by special instructions.

4) Status Register

The Status Register, SR, is a flag register consisting of 8 bits, as shown below:



The SR consists of three types of flags, as listed and described below:

A. Condition Codes

The Condition Codes, CC, are four flags which record the result of performing an instruction and may later be used to cause conditional program branching, determined by the result of the instruction. They are:

C - The Carry bit, C, is set or cleared by most arithmetic instructions to save carries out of the most significant bit of adder (unsigned arithmetic overflows). The value of C (1 or 0) may be added to an operand by a subsequent instruction, thus facilitating multiple-precision arithmetic. It also receives the bit which is shifted out as a result of a rotate or shift instruction, and provides the bit to be shifted in by a rotate instruction. Thus, the C bit acts as a link between the most and least significant bits of data in a rotate operation.

V - The Signed Overflow bit, V, is set or cleared by most arithmetic instructions to indicate whether a signed two's-complement arithmetic overflow occured as a result of the instruction. A signed arithmetic overflow is the result of adding two numbers of the same sign whose sum exceeds the capacity of the register, yielding a result of the opposite sign, or of subtracting two numbers of different sign whose difference exceeds the capacity of the register. The V bit is irrelevant to unsigned arithmetic. Most logical instructions clear V.

Left shift and rotate instruction (but not right shift and rotate) shifts V in a special way. For these instructions V is set if the bit shifted into the signed bit (most significant bit) is difference from the bit shifted out of the signed bit, and cleared otherwise.

M - The Minus bit, M, is set or cleared by most arithmetic and logical instructions to record the value of the sign bit (most significant bit) of the result.

Z - The Zero bit, Z, is set or cleared by most arithmetic and logical instructions to record whether the result of the operation was zero. Z is set on a zero result, and cleared otherwise.

B. Interrupt Control

The Interrupt Control Flags, LIE and HIE, enable interrupts in response to requests on the Low-, and High-priority interrupt levels, respectively. Their use will be explained more fully in a later section.

C. User Flags

The User Flags, SF1 and SF0, may be set, cleared and tested by special instruction. Their use is uncommitted.

5) Program Counter

The Program Counter, PC, is a 16-bit counter which always contains the address of the next instruction byte to be fetched from memory.

6) Memory

The memory may be as large as 65K bytes. It may consist of read/write and/or read-only types of various speeds in any mixture. Memory access is not constrained by page boundaries.

7) Interval Timer/Real-Time Clock

The Interval Timer/Real-Time Clock (IT/RTC) provides a means of measuring time intervals for controlling I/O device operations, etc., or maintaining a real-time clock. It consists of a time-base generator timed by the processor input clock signals, Ø1M and Ø2M, an interval counter driven by the time base, interrupt logic, and various control flags. These registers and control flags are described below:

A. Time Base and TBS

The Time Base register is a pre-determined counter driven by the processor clock which produces time base pulses at one of two program selectable rates, determined by the state of the Time Base select flag, TBS, as follows:

TBS = Ø - Time Base frequency = 1 KHz

TBS = 1 - Time Base frequency = 1 Hz

B. Interval Counter

The interval counter, IC, is an 8-bit binary counter which is loaded under program control and incremented by time base pulses until overflow to time out the selected interval.

C. Time Out Logic and TO

The Time Out Logic examines the state of the IC and sets the Time Out flag, TO, when the selected interval has passed. It operates in one of two modes: If IC is initially loaded with zero, TO is set each time a time base pulse occurs; if IC is initially loaded with any non-zero value, TO is set upon IC overflow, and all IT/RTC operations are then halted until a new IT/RTC operation is initiated under program control. In the first (RTC) mode, the IT/RTC functions as a real-time clock, setting TO periodically at regular intervals, while in the second (IT) mode, the IT/RTC functions as an interval timer to time a single, pre-selected interval and then halt.

D. Timer Interrupt Logic and TE

The Timer Interrupt Logic is enabled and disabled by the Timer Interrupt Enable flag, TE. TE is cleared to disable timer interrupts each time a new IT/RTC operation is initiated and may subsequently be set under program control. Whenever TE is ONE and TO is ONE and low-priority interrupts are enabled (LIE is ONE), a processor interrupt occurs via an interrupt vector at location 3. Thus, the IT/RTC functions as the highest device on the low-priority interrupt bus, on a tenth level. (See section III for a discussion of interrupts.) Upon initiating the timer interrupt, both TO and TE are cleared.

III. Interrupt Operation

The interrupt system is a two-line, multiple-device priority structure. Each device which must interrupt processor operation is assigned to one of two priority lines (the high-priority line, *SI, or the low-priority line, *IA) and one of nine device levels on that line. The device may request a processor interrupt by asserting its assigned priority line. More than one device may be requesting an interrupt on the same line simultaneously. At the end of each instruction, the processor examines the *SI line and if it is asserted and high-priority interrupts are enabled (HIE = ONE), it responds to the request by initiating a Peripheral In operation with DEH ⟨7:0⟩ = peripheral address FE_{16}. (See Electrical Specification for timing details of the Peripheral In operation.) When the requesting device sees the Peripheral In FE_{16} command it identifies its device level in one of two ways: If it is assigned to the lowest device level, it makes no response at all. The processor waits approximately three microcycles after initiating the Peripheral In operation and, if no device has responded, it assumes that the lowest-level device is requesting, drops Bus Attention, and interrupts on that level. If the device is assigned to one of the eight higher device levels, it responds to the Peripheral In command by bringing the DEL ⟨7:0⟩ line corresponding to its assigned device level to ZERO and asserting *AK.

The DEL ⟨7⟩ line is the highest device level and DEL ⟨Ø⟩ , the second lowest. Upon seeing the assertion of *AK, the processor finishes the Peripheral In operation by reading the DEL ⟨7:0⟩ lines and negating Bus Attention. It then sequentially scans the eight bits, starting with bit 7, and interrupts on the level of the first zero bit encountered. If no zero bit is found, the processor interrupts on the lowest device level.

Low-priority interrupt requests are handled in a similar manner. If, at the end of an instruction, the *IA line is asserted and low-priority interrupts are enabled (LIE = ONE) and no high-priority interrupt is requested and enabled, then the processor responds with a Peripheral In FF_{16} command to determine which device level is the highest requesting. Except for the peripheral address issued (FF_{16} instead of FE_{16}) the determination of interrupt level proceeds exactly as for high-priority interrupts.

When the processor has responded to an interrupt request and determined the interrupt priority and device level, it proceeds with the following operations:

a) PC $\langle 7:0 \rangle$ is pushed onto the GS

b) PC $\langle 15:8 \rangle$ is pushed onto the GS

c) SR is pushed onto the GS

d) An interrupt vector address is determined, according to the interrupt priority and level. The interrupt vector addresses are located in the lower part of memory. Each interrupt vector consists of three consecutive bytes. The address for a low-priority interrupt on the lowest device level is 6, for a high-priority interrupt on the lowest device level it is 9, for a low-priority interrupt on the second lowest level it is 12, etc. Thus, the address for a low-priority interrupt on the highest device level is 54_{10} and that for a high-priority interrupt on the highest level is 57_{10}.

e) The contents of the byte of memory located at the interrupt vector address is loaded into PC $\langle 7:0 \rangle$.

f) The contents of the next byte, the second byte of the interrupt vector, is loaded into PC $\langle 15:8 \rangle$.

g) The contents of the third byte of the interrupt vector is loaded into the SR.

h) Program execution is resumed at the location specified by the new PC contents.

Thus, the PC and SR are saved in the stack and a new PC and SR are fetched from the interrupt vector. Normally, the new PC will be the address of the interrupt service routine for the peripheral device assigned to the corresponding priority and device level, and the new SR will disable interrupts at and below the interrupting priority while enabling higher-priority interrupts. Thus, interrupt vectors for low-priority interrupts will contain a ZERO for the LIE and a ONE for the HIE bits of the new SR, while interrupt vectors for high-priority interrupts will contain ZEROs for both the LIE and SIE bits. This allows a low-priority interrupt to be itself interrupted by a high-priority interrupt, but not the other way around.

In some cases it may be necessary or desirable to assign more than one peripheral device to the same priority/device level. This may be true if the number of peripherals which must interrupt the processor exceeds 18, or when system cost considerations dictate the minimization of peripheral interface complexity, making it desirable to put multiple devices on the lowest device level so that they need not respond to the Peripheral In FF_{16} or Peripheral In FE_{16} command. In this case, the interrupt service routine must interrogate the various devices to determine which device initiated the interrupt request.

In addition to requests on the *SI and *IA lines, interrupts may also be caused in two other ways. The first of these is an IT/RTC interrupt, as described in section II.7.d. This interrupt uses an interrupt vector at location 3. Note that timer interrupts can occur only when low-priority interrupts are enabled (LIE = ONE).

Additionally, a Power Up Start interrupt is generated upon the negation of *RESET, unless the console issues a console command. (See section VI for a description of console operations.) Power Up Start interrupts use interrupt vector address Ø, Power Up Start interrupts have the highest priority of all and cannot be disabled by either LIE or SIE.

IV. Addressing

Several different address modes may be used for memory reference instructions, as described below:

1) Immediate Addressing

In Immediate Mode, the Effective Address (EA) of the operand is contained in the PC. After fetching each byte the PC is incremented by one to point to the next location. Thus, the data is attached to, and part of the instruction.

2) Indexed Addressing

Indexed Address Mode uses one of the two index registers, X or Y, as a base register. The contents of the base register plus an additional instruction byte, regarded as an unsigned integer in the range of 255 to Ø, forms the EA.

3) Autoincremented Addressing

In Autoincremented Address Mode, the SH, the top two bytes of the GS, are used as the EA. After fetching or storing each byte the SH is incremented by one.

4) Deferred Addressing

Deferred (indirect) Addressing may be specified with any of the above modes in most instructions. If deferral is specified, the EA calculated as per sections IV.1-IV.3 above are used to fetch two bytes which then become the EA of the operand. As with all multiple-byte data structures, deferred addresses are stored in memory least-significant-byte-first; that is, the address of the low-order deferred-address byte is calculated as per one of the above modes, and the high-order deferred-address byte is fetched from the next successive location. Note that by using Immediate Mode Deferred, a full 16-bit address may be specified in an instruction.

5) Relative Addressing

Branch instructions always use Relative Addressing to specify the location at which program execution is to resume if the branch test is successful. The relative address is formed by adding a second instruction byte with sign (most significant) bit extended to form a 16-bit displacement to the PC to form the effective address. Thus, relative addressing allows any location within -128_{10} to +127 bytes of the first byte of the instruction following the branch instruction to be specified as the branch address.

V. Instructions

Several different instruction formats are used which occupy from one to four bytes of memory. The operation to be performed is specified by the first byte of the instruction; subsequent bytes, if any, specify address(es) and/or immediate data.

1) Symbols and Abbreviations

The symbols and abbreviations used in describing the instructions are listed below:

a) Subregisters and Compound Registers

x $\langle a:b \rangle$ - a "subregister" consisting of bits a through b, inclusive, of register x.

x,y - a "compund register", the more significant bit(s) of which are the bits of the register or subregister x, and the less significant bit(s) of which are those of the register or subregister y.

b) Register and Flags

A or A $\langle 7:0 \rangle$ - the contents of accumulator A

B or B $\langle 7:0 \rangle$ - the contents of accumulator B

X or X $\langle 15:0 \rangle$ - the contents of index register X

Y or Y $\langle 15:0 \rangle$ - the contents of index register Y

PC or PC $\langle 15:0 \rangle$ - the contents of the Program Counter

GS - the General Stack. When a byte from the GS is used, it is popped (removed) from the stack; when GS is loaded, the byte is pushed (added) to the stack.

SH or SH $\langle 15:0 \rangle$ - the top two bytes of the GS

SR or SR $\langle 7:0 \rangle$ - the contents of the Status Register

c) Memory Locations

EA - Effective Address

(x) - the contents of memory location x.

d) Operators

$\rightarrow$ - transfer of data; read as "replaces"

$+$ - binary two's-complement addition

$-$ - binary two's-complement subtraction

$\vee$ - logical Inclusive OR

$\wedge$ - logical AND

$\veebar$ - logical Exclusive OR (XOR)

$\leftarrow$ - sign extension; e.g., "$\leftarrow x\langle 7{:}0\rangle \rightarrow y\langle 15{:}0\rangle$" is equivalent to "$x\langle 7\rangle$, $x\langle 7\rangle$, $x\langle 7\rangle$, $x\langle 7\rangle$, $x\langle 7\rangle$, $x\langle 7\rangle$, $x\langle 7\rangle$, $x\langle 7\rangle$, $x\langle 7{:}0\rangle \rightarrow y\langle 15{:}0\rangle$". Sign extension preserves the value of signed, two's-complement number when it is transferred or added to a second register with more bits of precision than the first.

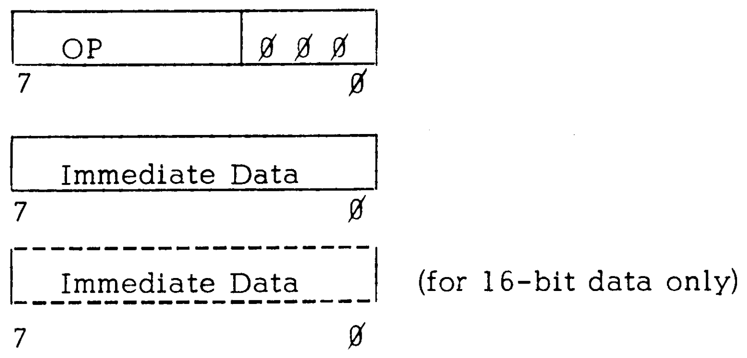
e) Unless otherwise stated, all Instruction Opcodes are listed in the hexadecimal number systems.

2) Single-Address, Multi-Mode, Memory-Reference Instructions (Multi-Mode Group)

These instructions provide the primary means of loading and storing registers and memory locations, of performing binary (two-operand) operations from memory to a register, and of performing unconditional program jumps and subroutine calls.

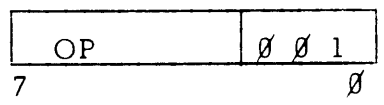
Six basic address modes may be used with most of these instructions, as shown below:

Immediate:



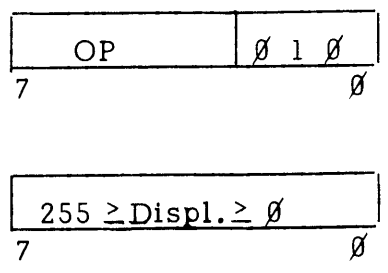
In Immediate Mode, the data, which may consist of one or two bytes, is located immediately following the instruction byte.

Autoincrement:



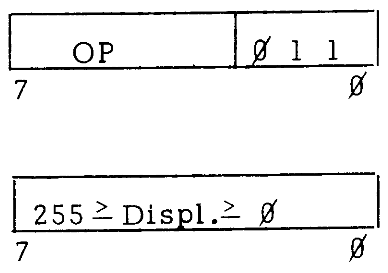
In Autoincrement Mode, the SH contains the EA. The SH is incremented after each byte of data is fetched.

X-Indexed:



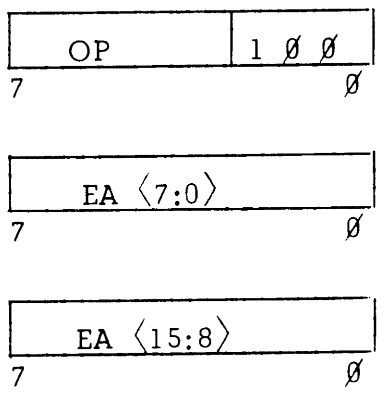
In X-Indexed Mode, the contents of X plus the displacement byte is the EA.

Y-Indexed:



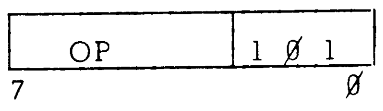
In Y-Indexed Mode, the contents of Y plus the displacement byte is the EA.

Immediate Deferred:
(Full Address)



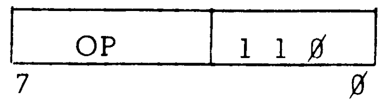
In Immediate Deferred, or Full Address Mode, the EA is contained in two bytes immediately following the instruction byte.

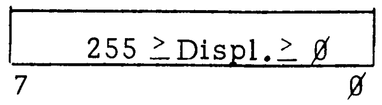
Autoincrement Deferred:



In Autoincrement Deferred Mode, the EA is contained in two consecutive bytes of memory, the address of the first of which is contained in the SH. The SH is incremented by one after each byte of the EA is fetched.

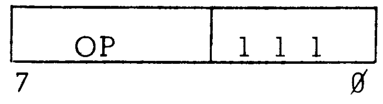
X-Indexed Deferred:

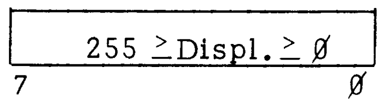




In X-Indexed Deferred Mode, the EA is contained in two consecutive bytes of memory, the address of the first of which is the contents of X plus the displacement byte.

Y-Indexed Deferred:





In Y-Indexed Deferred Mode, the EA is contained in two consecutive bytes of memory, the address of the first of which is the contents of Y plus the displacement byte.

In certain cases, wherein the use of a particular address mode with a particular instruction would be of little or no utility, that opcode/mode combination was used to implement other, Operate Group, instructions. In such a case, the statement that "____" Mode is invalid for this instruction" is included in the instruction definition.

The condition codes may be treated in one of three ways, indicated below:

"CC Unchanged" - No change to C, M, Z or V bits
"Z,M,V Loaded" - Z set if result is zero, cleared otherwise; M set equal to most significant bit of result; V cleared unconditionally; C remains unchanged.
"CC Loaded" - Z set if result is zero, cleared otherwise; M set equal to most significant bit of result; V set if signed arithmetic overflow occurs as a result of operation, cleared otherwise; C set to value of carry out of the most significant bit of the adder during operation.

In the instruction definitions below, the top line consists of the mnemonic, named, and opcode of the instruction, in that order.

a) LDX - LoaD X - ØØ + MODE
(EA + 1), (EA) →X; CC Unchanged

b) LDY - LoaD Y - Ø8 + MODE
(EA + 1), (EA)→ Y; CC Unchanged

c) LDA - LoaD A - 1Ø + MODE
(EA) → A; Z, M, V Loaded

d) LDB - LoaD B - 18 + MODE
(EA) → B; Z, M, V Loaded

e) ADA - ADd A - 2Ø + MODE
Ø, (EA) +Ø,A →C,A; CC Loaded

f) ADB - ADd B - 28 + MODE
Ø, (EA) + Ø,B→ C,B; CC Loaded

g) SBA - SuBtract A - 3Ø + MODE
Ø, A-1, (EA)→ C,A; CC Loaded

h) SBB - SuBtract B - 38 = MODE
Ø, B - 1, (EA) →C,B; CC Loaded

i) XORA - eXclusive OR A - 4Ø + MODE
A $\veebar$ (EA) →A; Z, M, V Loaded

j) IORA - Inclusive OR A - 48 + MODE
A v (EA) →A; Z, M, V Loaded

k) ANDA - AND A - 5Ø + MODE
A ^ (EA)→ A; Z,M,V Loaded

l) CMPA - CoMPare to A - 58 + MODE
Ø, A - 1, (EA) →C, Temporary register; CC Loaded
(Note: the only effect of CMPA is to set the CC as though a SBA instruction had been executed; no register is changed.)

m) PUM - PUsh Memory - 6Ø + MODE
(EA) → GS; CC Unchanged
(Pushes one byte of data onto stack)

n) PUMD - PUsh Memory Double - 68 + MODE

(EA) → GS; (EA + 1) → GS; CC Unchanged

(Pushes two bytes onto stack)

o) BTST - Bit TeST - 7Ø + MODE

(EA) $\wedge$ (PC) Temporary register; Z, M, V Loaded; PC + 1→ PC

(Note 1: This instruction has one additional byte, a mask byte, following the address byte(s), if any. This mask byte is ANDed with the data byte, and Z and M are set according to the result. No register is changed. Note 2: Immediate (non-Deferred) Mode is invalid with BTST.)

p) INC - INCrement memory - 78 + MODE

(EA) + 1 → (EA); Z, M, V Loaded

(Note: Immediate (non-Deferred) and Autoincrement (non-Deferred) Modes are invalid with INC.)

q) STA - STore A - 8Ø + MODE

A →(EA); Z, M, V Loaded

(Note: Immediate (non-Deferred) mode is invalid with STA.)

r) STB - STore B - 88 + MODE

B →(EA); Z, M, V Loaded

(Note: Immediate Mode is invalid with STB.)

s) POM - POp to Memory - 9Ø + MODE

GS → (EA); (One byte popped off stack and stored); CC Unchanged

(Note: Immediate, Autoincrement, and Autoincrement Deferred Modes are invalid with POM.)

t) POMD - POp to Memory Double - 98 + MODE

GS → (EA + 1); (Two bytes popped off stack and stored); GS → (EA); CC Unchanged

(Note: Immediate, Autoincrement, and Autoincrement Deferred Modes are invalid with POMD.)

u) JMP - JuMP - AØ + MODE

EA→ PC; CC Unchanged

(Note: Immediate and Autoincrement Modes are invalid with JMP)

v) PJMP - Push and JuMP - A8 + MODE

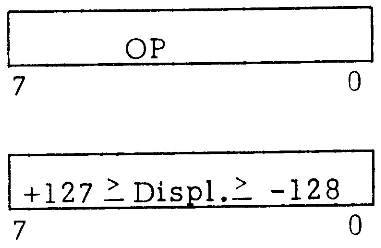
PC $\langle 7:0 \rangle$ → GS; PC $\langle 15:8 \rangle$ → GS; (PC pushed onto stack)

EA→ PC; CC Unchanged.

(Note: Immediate and Autoincrement Modes are invalid with PJMP.)

3) Branch Instructions

Branch instructions occupy two bytes, as shown below:



When the test condition is met, the displacement byte, with sign extended, is added to the PC to allow branching to any location within +127 to -128 locations of the byte following the displacement byte. Otherwise, instruction execution continues sequentially with the instruction following the displacement byte.

All Branch Instructions leave the Condition Codes unchanged.

a) BRN - BRaNch - B1

Unconditional branch

b) BRTO - BRanch on Time Out - B2 branch if TO (Time Out flag) = 1;

Ø →TO.

c) BNTO - Branch on Not Time Out - B3 branch if TO = Ø; Ø →TO.

d) IBZX - Increment and Branch on Zero X - B4

X + 1 →X; branch if X = 0.

e) IBZY - Increment and Branch on Zero Y - B5

Y + 1→Y; branch if Y = Ø

f) IBNX - Increment and Branch on Non-zero X - B6

X + 1 → X; branch if X ≠ Ø.

g) IBNY - Increment and Branch on Non-zero Y - B7

Y + 1 →Y; branch if Y ≠ Ø.

h) BRZ - BRanch on Z - B8

i) BNZ - Branch on Not Z - B9

j) BRM - BRanch on M - BA

k) BNM - Branch on Not M - BB

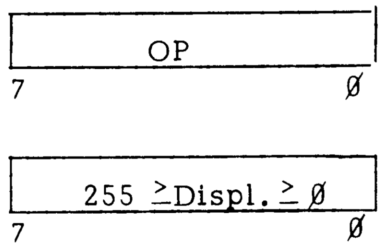
l) BRV - BRanch on V - BC

m) BNV - Branch on Not V - BD

n) BRC - BRanch on C - BE

o) BNC - Branch on Not C - BF

4) Unary, Memory-to-Memory, 32-bit Data Instructions (M-M Unary Group)
The M-M Unary instructions perform unary (single-operand) operations on 32-bit (4-byte) signed two's-complement integers stored in memory and leave the result where the operand was. These instructions are limited to one address mode, the Y-Indexed Mode. Therefore, they are two-byte instructions, as shown below:



a) SLI - Shift Left, Integer - CØ

(EA + 3), (EA + 2), EA + 1), (EA), Ø → C, (EA + 3), (EA + 2), (EA + 1), (EA);

CC Loaded.

This operation shifts left, with zero shifted into the least significant bit. As in all shifts and rotates, the bit shifted out is shifted into C.

b) RLI - Rotate Left, Integer - C1

(EA + 3), (EA + 2), (EA + 1), (EA), C $\rightarrow$ C, (EA + 3), (EA + 2), (EA + 1), (EA);
CC Loaded.

This operation rotates left through C; that is, the old C is shifted into the least significant bit, and the old most significant bit is shifted into C. All rotate operations in this machine rotate through C.

c) SRI - Shift Right, Integer - C2

(EA + 3) $\langle 7 \rangle$, (EA + 3), (EA + 2), (EA + 1), (EA) $\rightarrow$ (EA + 3), (EA + 2),
(EA + 1), (EA), C; CC Loaded

This operation shifts right, preserving (duplicating) the sign bit. The old least significant bit is shifted into C.

d) RRI - Rotate Right, Integer - C3

C, (EA + 3), (EA + 2), (EA + 1), (EA) $\rightarrow$ (EA + 3), (EA + 2), (EA + 1), (EA), C;
CC Loaded

This operation rotates right through C.

e) NGI - NeGate Integer - C4

-1, (EA + 3), (EA + 2), (EA + 1), (EA) $\rightarrow$ C, (EA + 3), (EA + 2), (EA + 1), (EA);
CC Loaded

f) ABSI - ABSolute value of Integer - C5

If (EA + 3) $\langle 7 \rangle$ = 1:

-1, (EA + 3), (EA +2), (EA + 1), (EA) $\rightarrow$ C, (EA + 3), (EA + 2), (EA + 1), (EA);
CC Loaded;

Else if (EA + 3) $\langle 7 \rangle$ = Ø:

1 $\rightarrow$ C $\rightarrow$ V $\rightarrow$ M $\rightarrow$ Z.

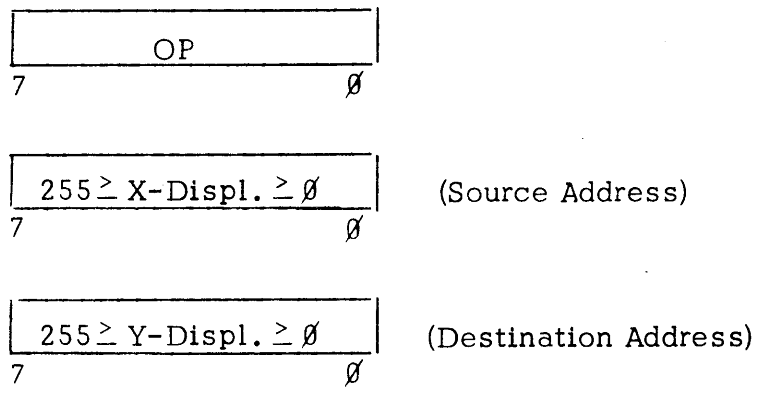
If the (old) value of the operand is positive, the CC bits are set to 1's, but no change is made in the data; otherwise, if the value of the operand is negative, then the operand is negated to produce a positive result and CC is loaded according to the result. Usually, a two's complement negation of a negative number produces a zero sign bit (M = Ø), zero carry out (C = Ø), and no signed overflow (V = Ø), but if the most negative number (in this case, -2^{31}) is negated, then M, C and V will all be 1's. However, regardless of which negative value is chosen, the result will never be zero. Hence, a BRZ or BNZ instruction following the ABSI instruction will determine the sign of the data prior to the ABSI operation. If Z = 1, the data was positive; if Z = Ø, it was negative.

g) CLI - CLear Integer - C6

Ø → (EA + 3), (EA + 2), (EA + 1), (EA); Z, M, V Loaded

5) Binary, Memory-to-Memory, 32-bit Data Instructions (M-M Binary Group)

The M-M Binary instructions perform binary (two-operand) operations on 32-bit (4-byte) signed two's complement integers stored in memory. The instructions contain two addresses, a Source and a Destination. The instruction obtains one operand from the Source and the other from the Destination, and leaves the result in the Destination. As with the M-M Unaries, the M-M Binaries are limited to a single type of addressing: the Source address uses X-Indexed Mode, and the Destination uses Y-Indexed Mode. Thus, they are three-byte instructions, as shown below:



In the instruction definitions below, the source EA is called SA and the Destination EA is called DA.

a) MVI - MoVe Integer - C7

(SA + 3), (SA + 2), (SA + 1), (SA) → (DA + 3), (DA + 2), (DA + 1), (DA); M, Z, V Loaded.

b) ADI - ADd Integer - C8

Ø, (SA + 3), (SA + 2), (SA + 1), (SA) + Ø, (DA + 3), (DA + 2), (DA + 1), (DA) → C, (DA + 3), (DA + 2), (DA + 1), (DA); CC Loaded.

c) ADCI - ADd with Carry Integer - C9

Ø, (SA + 3), (SA + 2), (SA + 1), (SA) + Ø, (DA + 3), (DA + 2), (DA + 1) (DA) + C → C, (DA + 3), (DA + 2), (DA + 1), (DA); CC Loaded

d) SBI - SuBtract Integer - CA

Ø, (DA + 3), (DA + 2), (DA + 1), (DA) -1, (SA + 3), (SA + 2), (SA + 1), (SA) → C, (DA + 3), (DA + 2), (DA + 1), (DA); CC Loaded

e) SBCI - SuBtract with Carry Integer - CB

Ø, (DA + 3), (DA + 2), (DA + 1), (DA) -1, (SA + 3), (SA + 2), (SA + 1), (SA) -1 + C → C, (DA + 3), (DA + 2), (DA + 1), (DA) CC Loaded

6) Register Operation Instructions (Operate Group)

The Operate instructions perform inter-register data transfers and unary operations of various sorts. In general, operations which can be performed on one of a pair of similar registers (A and B, X and Y) can also be performed on the other. In the listing which follows, such instructions are listed in pairs, only the first of which is fully described.

a) PUSR - PUsh SR - D6

SR → GS; CC Unchanged

b) POSR - POp SR - D7

GS → SR; CC changed in the same manner as the rest of the SR.

c) SWX - SWap X with SH - DA

X → Temporary register; SH → X; Temporary register → SH; CC Unchanged.

d) SWY - SWap Y with SH - DB

e) PUX - PUsh X - DC

X ⟨7:Ø⟩ → GS; X ⟨15:8⟩ → GS; CC Unchanged

f) PUY - PUsh Y - DD

g) POX - POp X - DE

GS → X ⟨15:8⟩ ; GS → X ⟨7:Ø⟩ ; CC Unchanged

h) POY - POp Y - DF

i) CLA - CLear A - EØ.

Ø → A; Ø → V; Ø → M; 1 → Z

j) CLB - CLear B - E1

k) RLA - Rotate Left A = E2

A,C → C,A; CC Loaded

l) RLB - Rotate Left B - E3

m) RRA - Rotate Right A - E4

C,A → A,C; CC Loaded

n) RRB - Rotate Right B - E5

o) SRA - Shift Right A - E6

A → ⟨7⟩ , A → A, C; CC Loaded

p) SRB - Shift Right B - E7

q) INA - INcrement A - E8 ; Ø, A + 1 → C, A; CC Loaded

r) INB - INcrement B - E9

s) DCA - DeCrement A - EA

Ø, A - 1 → C, A; CC Loaded

t) DCB - DeCrement B - EB

u) ACA - Add Carry to A - EC

Ø, A + C → C, A; CC Loaded

v) ACB - Add Carry to B - ED

w) DACA - Decrement and Add Carry to A - EE

Ø, A - 1 + C → C, A; CC Loaded

x) DACB - Decrement and Add Carry to B - EF

y) NGA - NeGate A - FØ

-1, A → C, A; CC Loaded

z) NGB - NeGate B - F1

aa) CMA - CoMplement A - F2

-A-1 → A; Z, M, V Loaded

ab) CMB - CoMplement B - F3

ac) PUA - PUsh A - F4

A → GS; Z, M, V Loaded

ad) PUB - PUsh B - F5

ae) POA - POp A - F6

GS → A; Z, M V Loaded

af) POB - POp B - F7

ag) SDA - Swap Digits of A - F8

A ⟨3:Ø⟩ , A ⟨7:4⟩ → A; Z, M, V Loaded

ah) SDB - Swap Digits of B - F9

ai) SAB - Swap A and B - FA

A, B → B, A; Z, M, V Loaded

(Note: M is loaded with the new value of B ⟨7⟩)

aj) RDLD - Rotate Digits Left Double - FB

B ⟨3:Ø⟩ , A, B ⟨7:4⟩ → B, A; Z, M, V Loaded

ak) RLD - Rotate Left Double - FC

B, A, C → C, B, A; CC Loaded

al) RRD - Rotate Right Double - FD

C, B, A → B, A, C; CC Loaded

am) SXY - Swap X and Y - FE

X, Y → Y, X; CC Unchanged

an) HLT - HaLT - FF

This instruction causes all processor operations to cease, with PC containing the address of the byte following the HLT.

ao) CPYA - CoPY A into B - 8Ø

A → B; M, Z, V Loaded

ap) CPYB - CoPY B into A - 88

aq) POP - POP GS - 9Ø; GS → Temporary Register; CC Unchanged

ar) POPD - POP GS Double - 98

GS → Temporary register; GS → Temporary register; CC Unchanged

as) ISH - Increment SH - 91

SH + 1 → SH; CC Unchanged

at) ISHT - Increment SH by Two - 95

SH + 2 → SH; CC Unchanged

au) DSH - Decrement SH -99

SH - 1 → SH; CC Unchanged

av) DSHT - Decrement SH by Two - 9D

SH - 2 → SH; CC Unchanged

aw) NOP - No OPeration - A∅

No operation is performed

ax) CCAL - Co-CALl - A8

SH, PC → PC, SH; CC Unchanged

ay) RTS - ReTurn from Subroutine - A1

GS → PC; CC Unchanged

az) RTI - Return from Interrupt - A9

GS → SR; GS → PC; CC Unchanged

ba) ITO - Interrupt on Time Out - D8

1 → TE; CC Unchanged

(This enables a timer interrupt when both TO (Time Out flag) and LIE (Low-priority Interrupt Enable) are 1)

bb) RSTT - ReSeT Timer - D9

All timer flags are cleared, including TO and TE, and IT/RTC operation is halted

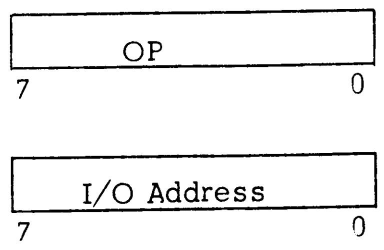
bc) CLC - CLear Carry - D3

∅ → C; Other CC Unchanged

7) Input and Output

The two I/O instructions defined below provide a means of communication between the processor and peripheral devices. All I/O operations are D.C.-interlocked to allow communication with devices of varying speeds located at varying distances from the processor. No timing circuitry or sequential logic is required to interface a peripheral to the Data Exchange Bus. See the Electrical Specification for details of the Peripheral In and Peripheral Out I/O operations.

Both I/O instructions use the following format:



The I/O Address is transmitted on the Data Exchange Bus to select the particular peripheral device or device register with which the processor is to communicate. It is in no way related to any memory address.

The I/O instructions are:

a) IN - INput from peripheral - CC

I/O Address → Data Exchange Bus ⟨15:8⟩ ;

Peripheral In Operation;

Data Exchange Bus ⟨7:Ø⟩ → A; M, Z, V Loaded

b) OUT - OUTput to peripheral - CD

I/O Address → Data Exchange Bus ⟨15:8⟩ ;

A → Data Exchange Bus ⟨7:Ø⟩

Peripheral Out Operation; M, Z, V Loaded according to A

8) Miscellaneous Two-Byte Instructions

The following instructions have only one thing in common: All of them consist of a single instruction byte followed by a single byte of immediate data. In the following instruction definitions, this immediate data is called ID.

a) TSTSR - TeST bits of SR - 7Ø

SR $\wedge$ ID → Temporary register; M, V, Z Loaded

While C, M, V, & Z can be directly tested by Branch instructions, the other four bits of SR cannot; this instruction enables the program to test those bits by loading CC bits according to their state.

b) TSTA - <u>T</u>e<u>ST</u> bits of <u>A</u> - 78

A $\wedge$ ID $\rightarrow$Temporary Register; M, Z, V Loaded

c) TSTB - <u>T</u>e<u>ST</u> bits of <u>B</u> - 79

B $\wedge$ ID $\rightarrow$Temporary Register; M, Z, V Loaded

d) LITØ - <u>L</u>oad IC, start <u>IT</u>/RTC, and set TBS to <u>Ø</u> - CE

Ø$\rightarrow$ TBS; Ø $\rightarrow$TO; Ø$\rightarrow$ TE; ID$\rightarrow$ IC; Start IT/RTC Operation;

CC Unchanged

(See section II.7, Interval Timer/Real-Time Clock)

e) LIT1 - <u>L</u>oad IC, Start <u>IT</u>/RTC, and set TBS to <u>1</u> - CF

1$\rightarrow$ TBS; Ø $\rightarrow$TO, Ø$\rightarrow$ TE; ID $\rightarrow$IC;

Start IT/RTC Operation; CC Unchanged.

f) ADSH - <u>AD</u>d to <u>SH</u> - BØ

$\pm$ ID + SH $\rightarrow$SH; CC Unchanged

This instruction allows the Stack Head, SH, to be changed by any number in the range of −128 to + 127.

g) ADX - <u>AD</u>d to <u>X</u> - DØ

$\pm$ ID + X $\rightarrow$X; CC Unchanged

h) ADY - <u>AD</u>d to <u>Y</u> - D1

$\pm$ ID + Y$\rightarrow$Y; CC Unchanged

i) ANSR - <u>AN</u>d into <u>SR</u> - D4

ID $\wedge$ SR $\rightarrow$ SR

This allows any bit or bits of the SR to be cleared without modifying other bits.

j) ORSR - Inclusive <u>OR</u> <u>SR</u> - D5

ID v SR $\rightarrow$SR.

9) Find On List Instruction

The Find On List instruction, FOL, is in a class by itself. Like Operate Group instructions, it occupies only a single byte of memory, but unlike the Operates, it references memory, using Autoincrement Mode addressing. The is defined as:

FOL - Find On List - D2

(SH) $\forall$ A → Temporary register; Z, M, V Loaded;

If Z = 1 then go on to next instruction;

Otherwise:

SH + 1 → SH; B - 1 → B;

If B = Ø then go on to next instruction;

Otherwise:

PC - 1 → PC, so that the FOL instruction will be re-fetched and re-executed.

Thus, the FOL instruction fetches data bytes from a list using Autoincrement Mode, compares them to A, and decrements B until either a byte is found on the list which is equal to A, or all elements of the list have been checked (B = Ø).

Note the following characteristics of FOL:

a) The list length may be from one to 256 characters. (If B = Ø when FOL is first executed, 256 characters will be searched.)

b) If a character matching A is found, the FOL search will stop with the address of that character in the SH; otherwise, it will stop with the address of the last character on the stack, plus one, in the SH.

c) The FOL operation may be interrupted after each character is tested, and execution of the FOL operation will resume at the point at which it left off following the return from interrupt. This feature is necessary due to the length of the FOL operation when searching a long list.

10) Instruction Set Summary

a) Multi-Mode Group

Modes:	Immediate	Ø
	Autoincrement	1
	X-Indexed	2
	Y-Indexed	3
	Full (Immediate Deferred)	4

Autoincrement Deferred	5
X-Indexed Deferred	6
Y-Indexed Deferred	7

Instructions:

LDX - Load X	ØØ + Mode
LDY - Load Y	Ø8 + Mode
LDA - Load A	1Ø + Mode
LDB - Load B	18 + Mode
ADA - Add to A	2Ø + Mode
ADB - Add to B	28 + Mode
SBA - Subtract from A	3Ø + Mode
SBB - Subtract from B	38 + Mode
XORA - Exclusive - OR into A	4Ø + Mode
IORA - Inclusive - OR into A	48 + Mode
ANDA - AND into A	5Ø + Mode
CMPA - Compare with A	58 + Mode
PUM - Push Memory onto GS	6Ø + Mode
PUMD - Push Double-byte from Memory onto GS	68 + Mode
BTST - Bit Test Note: Mode Ø (Immediate) is invalid.	7Ø + Mode
INC - Increment Memory Note: Modes Ø (Immediate) and 1 (Autoincrement) are invalid.	78 + Mode
STA - Store A in Memory Note: Mode Ø (Immediate) is invalid.	8Ø + Mode
STB - Store B in Memory Note: Mode Ø (Immediate) is invalid.	88 + Mode
POM - Pop GS and Store in Memory Note: Modes Ø (Immediate), 1 (Autoincrement), and 5 (Autoincrement Deferred) are invalid.	9Ø + Mode
POMD - Pop Double-byte from GS and Store in Memory Note: Modes Ø (Immediate), 1 (Autoin-	98 + Mode

POMD - (Continued)

crement), and 5 (Autoincrement Deferred) are invalid.

JMP - Program Jump Note: Modes Ø (Immediate) and 1 (Autoincrement) are invalid.	AØ + Mode
PJMP - Push PC onto GS and Jump Note: Modes Ø (Immediate) and 1 (Atuoincrement) are invalid.	A8 + Mode

b) Branch Group (Relative Addressing)

BRN - Branch (Unconditional)	B1
BRTO - Branch if TO = 1 and clear TO	B2
BNTO - Branch if TO = Ø and clear TO	B3
IBZX - Increment X and Branch if X = Ø	B4
IBZY - Increment Y and Branch if Y = Ø	B5
IBNX - Increment X and Branch if X ≠ Ø	B6
IBNY - Increment Y and Branch if Y ≠ Ø	B7
BRZ - Branch if Z = 1	B8
BNZ - Branch if Z = Ø	B9
BRM - Branch if M = 1	BA
BNM - Branch if M = Ø	BB
BRV - Branch if V = 1	BC
BNV - Branch if V = Ø	BD
BRC - Branch if C = 1	BE
BNC - Branch if C = Ø	BF

c) M-M Unary Group (Y-Indexed Addressing)

SLI - Shift Integer Left	CØ
RLI - Rotate Integer Left	C1
SRI - Shift Integer Right	C2
RRI - Rotate Integer Right	C3
NGI - Negate Integer	C4

	ABSI - Take the Absolute Value of Integer	C5
	CLI - Clear Integer	C6
d)	M-M Binary Group (X-Indexed Addressing for Source, Y-Indexed Addressing for Destination)	
	MVI - Move Integer	C7
	ADI - Add Integer to Integer	C8
	ADCI - Add Integer to Integer with Carry	C9
	SBI - Subtract Integer from Integer	CA
	SBCI - Subtract Integer from Integer with Carry	CB
e)	Operate Group	
	CPYA - Copy A into B	8Ø
	CPYB - Copy B into A	88
	POP - Pop and Discard a byte from GS	9Ø
	ISH - Increment SH by one	91
	ISHT - Increment SH by two	95
	POPD - Pop and Discard a Double-byte from GS	98
	DSH - Decrement SH by one	99
	DSHT - Decrement SH by two	9D
	NOP - No Operation	AØ
	RTS - Return from Subroutine	A1
	CCAL - CoCall	A8
	RTI - Return from Interrupt	A9
	CLC - Clear C	D3
	PUSR - Push SR onto GS	D6
	POSR - Pop from GS into SR	D7
	ITO - Interrupt when TO = 1	D8
	RSTT - Reset IT/RTC	D9
	SWX - Swap X and SH	DA
	SWY - Swap Y and SH	DB
	PUX - Push X onto GS	DC

PUY - Push Y onto GS	DD
POX - Pop GS to X	DE
POY - Pop GS to Y	DF
CLA - Clear A	EØ
CLB - Clear B	E1
RLA - Rotate A Left	E2
RLB - Rotate B Left	E3
RRA - Rotate A Right	E4
RRB - Rotate B Right	E5
SRA - Shift A Right	E6
SRB - Shift B Right	E7
INA - Increment A	E8
INB - Increment B	E9
DCA - Decrement A	EA
DCB - Decrement B	EB
ACA - Add Carry to A	EC
ACB - Add Carry to B	ED
DACA - Decrement and Add Carry to A	EE
DACB - Decrement and Add Carry to B	EF
NGA - Negate A	FØ
NGB - Negate B	F1
CMA - Complement A	F2
CMB - Complement B	F3
PUA - Push A onto GS	F4
PUB - Push B onto GS	F5
POA - Pop GS to A	F6
POB - Pop GS to B	F7
SDA - Swap Digits of A	F8
SDB - Swap Digits of B	F9

	SAB - Swap A and B	FA
	RDLD - Rotate Digits Left in Double Register B,A	FB
	RLD - Rotate Double Register B,A Left	FC
	RRD - Rotate Double Register B,A Right	FD
	SXY - Swap X and Y	FE
	HLT - Halt	FF
f)	I/O Instructions	
	IN - Input from Peripheral	CC
	OUT - Output to Peripheral	CD
g)	Miscellaneous Two-Byte Instructions (Immediate data forms second byte)	
	TSTSR - Test Bits of SR	7Ø
	TSTA - Test Bits of A	78
	TSTB - Test Bits of B	79
	ADSH - Add Sign - Extended Byte to SH	BØ
	LITØ - Load IC, Start IT/RTC, and Clear TBS	CE
	LIT1 - Load IC, Start IT/RTC, and Set TBS	CF
	ADX - Add Sign-Extended Byte to X	DØ
	ADY - Add Sign-Extended Byte to Y	D1
	ANSR - AND into SR	D4
	ORSR - OR into SR	D5
h)	Find On List (Repeated instruction, Autoincexed Mode Addressing)	
	FOL - Find On List	D2

VI. Console Operation

The console operations described herein allow implementation of a console which operates as a peripheral device in the system. Great flexibility is afforded the system designer in defining a console to meet his cost and performance objectives. In fact, the system may function without any console at all, provided that certain parts of memory are non-volatile so that it is not necessary to load the memory upon power-up before beginning program execution. The various console features are described below.

1) *RESET Input Line

The *RESET input line to the processor allows the processor operation to be terminated at any point. When *RESET is asserted, the processor operation stops, any I/O or memory reference operation in progress is terminated, and any IT/RTC operation is stopped. When *RESET is negated, the processor attempts to perform a Peripheral In operation from the Console Command Register, I/O Address FC_{16}. The processor then waits three microcycles for *AK to be asserted.

If the Console does not respond, the processor clears all operating registers (A, B, X etc.) and performs a Power-Up Start interrupt using interrupt vector address Ø. This allows automatic starting of systems without consoles, and unattended power-up restart of all systems.

If the Console does respond, it places a console command on the Data Exchange $\langle 7:0 \rangle$ lines, which the processor reads and executes. The console commands are described below.

2) Console Commands

Console Commands, in conjunction with the *RESET and RUN input lines to the processor, enable the console to perform all standard console operations, such as reading and loading registers and memory, stopping and starting program execution as desired, performing single instructions on demand, etc. The console command format is shown below:

Console Command Register

ANY	OP
7 5	4 Ø

Bits 7 to 5 of the command are not used. The console commands are listed and described below. Most of them use data from, or lead data into the Console Data Register, I/O Address FD_{16}. For simplicity, the Peripheral In and Peripheral Out operations necessary to read and load this register are omitted from the descriptions below. and the Console Data Register is represented as a register called CDR. Except for operations ØØ and 1Ø, all operations read the CDR, whether or not they use the data.

OP:

ØØ - RUN, fetching the instruction from the address specified in PC.

Ø1 - SR $\rightarrow$ CDR
Reads the Status Register

Ø2 - SP $\rightarrow$ CDR
Reads the Stack Pointer Counter

Ø3 - Instruction Register $\rightarrow$ CDR
Reads the Instruction Register, containing the first byte of the last program instruction (not console command) fetched.

Ø4 - GS $\rightarrow$ CDR
Pops a byte from the GS

Ø5 - Increment SP; GS $\rightarrow$ CDR
Reads the last byte popped from the GS, or the next location into which data would be pushed. SP is first incremented, and then decremented (by popping data from the GS), and therefore is not changed by this operation.

Ø6 - (PC) $\rightarrow$ CDR
Reads the memory location that's address is contained in the PC.

Ø7 - PC + 1 $\rightarrow$ PC; (PC) $\rightarrow$ CDR
Increments the PC, and then reads the memory location that address is contained in the PC.

Ø8 - PC ⟨7:Ø⟩ → CDR
Reads the low-order byte of the PC.

Ø9 - PC ⟨15:8⟩ → CDR
Reads the high-order byte of the PC.

ØA - A → CDR
Reads A.

ØB - B → CDR
Reads B.

ØC - X ⟨7:Ø⟩ → CDR
Reads the low-order byte of X.

ØD - X ⟨15:8⟩ → CDR
Reads the high-order byte of X.

ØE - Y ⟨Ø:7⟩ → CDR
Reads the low-order byte of Y.

ØF - Y ⟨15:8⟩ → CDR
Reads the high-order byte of Y.

1Ø - Run, fetching the instruction from the address specified in PC.
This operation is identical to ØØ.

11 - CDR → SR; SR → CDR
Load Status Register.

12 - CDR → SP; SP → CDR
Load SP

13 - CDR → CDR; CDR → Instruction Register; Run

The instruction byte read from the CDR is executed. If it requires additional instruction bytes, then these bytes are fetched from memory using the PC for address, and the PC incremented after each byte is fetched. If the console does not stop the processor, it will then go on executing instructions from memory. This command is identical to ØØ and 1Ø, except that the first instruction byte is fetched from the CDR.

14 - GS → Temporary register;
CDR → Temporary register;
Temporary register → GS;
Temporary register → CDR.
The top byte of the GS is changed (loaded).

15 - CDR→ Temporary register;
Temporary register→ GS;
Temporary register→ CDR.
A byte is pushed onto the GS.

16 - (PC) → Temporary register;
CDR→ Temporary register;
Temporary register→ (PC);
Temporary register→ CDR.
A byte is loaded into the memory location that's address is contained in the PC.

17 - PC + 1 →PC;
(PC) → Temporary register;
Temporary register→ (PC);
Temporary register→ CDR.

Increments the PC, and then loads the memory location that's address is contained in the PC.

18 - CDR→ PC $\langle 7:\emptyset \rangle$;
PC $\langle 7:\emptyset \rangle$ → CDR.
Loads the low-order PC.

19 - CDR→ PC $\langle 15:8 \rangle$;
PC $\langle 15:8 \rangle$ → CDR.
Loads the high-order PC.

1A - CDR→ A;
A →CDR.
Loads A.

1B - CDR→ B;
B →CDR.
Loads B.

1C - CDR→X $\langle 7:\emptyset \rangle$;
X $\langle 7:\emptyset \rangle$ →CDR.
Loads low-order X.

1D - CDR→ X $\langle 15:8 \rangle$;
X $\langle 15:8 \rangle$ → CDR.
Loads high-order X.

1E - CDR→ Y $\langle 7:\emptyset \rangle$;
Y $\langle 7:\emptyset \rangle$ →CDR.
Loads low-order Y.

1F - CDR→ Y $\langle 15:8 \rangle$;
Y $\langle 15:8 \rangle$ → CDR.
Loads high-order Y.

Note that all of the above operations, except ØØ and 1Ø, which refer to no register, load the CDR (perform a Peripheral Out/FD_{16} operation) with the final contents of the register on which the operation (read or load) is performed. In most applications the Peripheral In FD_{16} operation will read a switch register which will be the "CDR" for inputs, while the Peripheral Out FD_{16} operation will load a display register which will be the "CDR" for outputs.

At the end of each console command, the processor will begin to execute instructions, starting at the location specified in the PC, unless inhibited from doing so by the negation of RUN. (See below)

3) RUN Input Line

The RUN line enables normal processor operation. When RUN is negated, the processor operation comes to a halt the next time a "Return" microinstruction is encountered; that is, it comes to a halt at the end of execution of the current macroinstruction or console operation.

Negating the RUN line differs from asserting the *RESET line in three important ways The first is that, while asserting *RESET stops all processor operation immediately, negating RUN stops only at the end of an operation. The second is that negating RUN does not clear any processor flags nor pending interrupts nor does it stop the IT/RTC. The third is that, while negating *RESET causes the processor to poll the console and then either execute a console command or clear the processor registers and perform a Power-Up Start interrupt, reasserting RUN simply causes the processor to continue running where it left off. Thus, while asserting *RESET actually resets the processor, negating RUN simply causes it to pause between instructions or console commands.

4) Performing Console Operations

The only way to initiate a console command is to assert, and then negate *RESET. However, this cannot be done at just any point in instruction execution if the major "state" of the processor is to be preserved. For instance, during the execution of an Autoincrement Mode Instruction the SH may have been popped off of the GS and stored in a temporary register for use as an address, and may not have yet been incremented and pushed back onto the GS. If *RESET were asserted at that time, the SH would be lost and the two bytes on the GS immediately below the SH would appear to be the SH.

In order to prevent such difficulties, RUN should be negated for a length of time greater than the length of the longest instruction sequence, prior to asserting *RESET. If this is done, the processor will always have paused at the end of an instruction before *RESET is asserted, and the major registers will not be disturbed. RUN may then be reasserted at any time before or after *RESET is negated. The processor will not begin to run until RUN has been reasserted <u>and</u> *RESET has been negated.

As mentioned above, unless continued execution is somehow inhibited, the processor will begin to fetch and execute instructions as soon as it finishes executing a console command. This may be prevented by negating RUN at some time between the start of the console operation and its end. There are two easy ways of insuring this timing when performing a series of console commands. The first is to keep RUN negated except for a brief time (2 to 4Ø microcycles) at the beginning of each console operation. Then RUN is asserted long enough to start the console command, but is negated before the end of the console operation. The other method is to allow the processor itself to time the RUN signal. In this case, RUN is asserted to start the console command operation, and is negated as soon as the processor polls the console with the Peripheral In FC (Read Console Command Register) operation.

5) Single-Instruction Mode

A useful feature for debugging both hardware and software is the ability to execute a single instruction when a switch is depressed, and then to pause so that the results of executing that instruction can be observed and interpreted. This is referred to as Single Instruction Mode operation.

There are two ways to perform Single Instruction Mode operation for this processor. The first is to maintain the processor in the Paused State (RUN negated) at all times except for brief (two-, to five microcycle) periods to start the execution of instructions. In this method, the *RESET line is never asserted.

The second method is the same as the first except that Console Command ØØ or 1Ø is performed to execute an instruction. Like all other console operations, but unlike the first method, Console Commands ØØ and 1Ø stop the IT/RTC. Also, while an interrupt may occur immediately if processor operation is started by simply reasserting RUN, the second method prevents interrupts from occuring. Therefore, the second method is preferable for most program debugging functions.