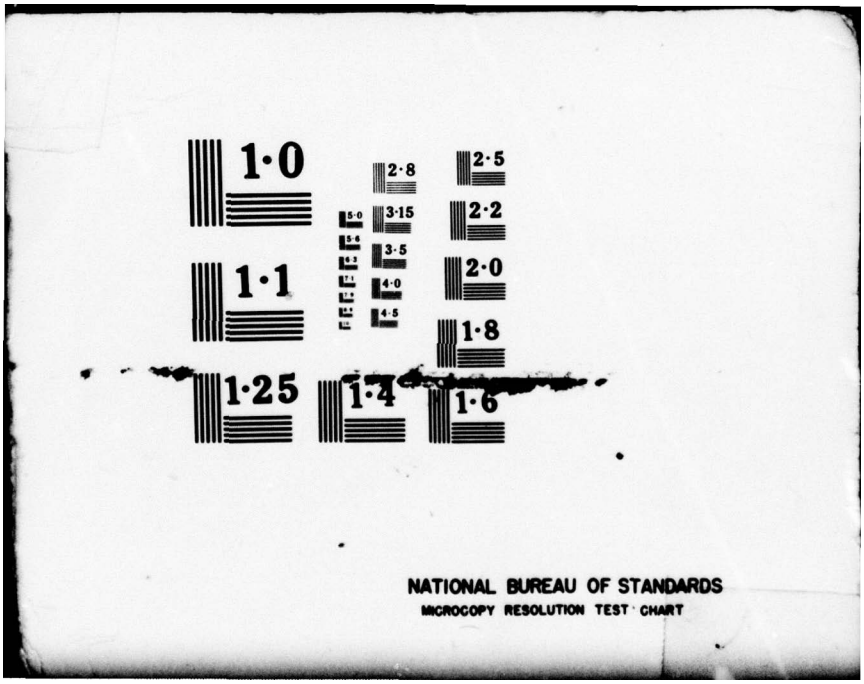
AD-A060 639 CALIFORNIA UNIV SAN DIEGO LA JOLLA COMPUTER SCIENCE DIV F/G 9/2
QM-1 MULTI MICRO-PROGRAMMER GUIDE.(U)
SEP 78 W A BURKARD , R D TUCK, R L HARTUNG N60921-77-M-B785

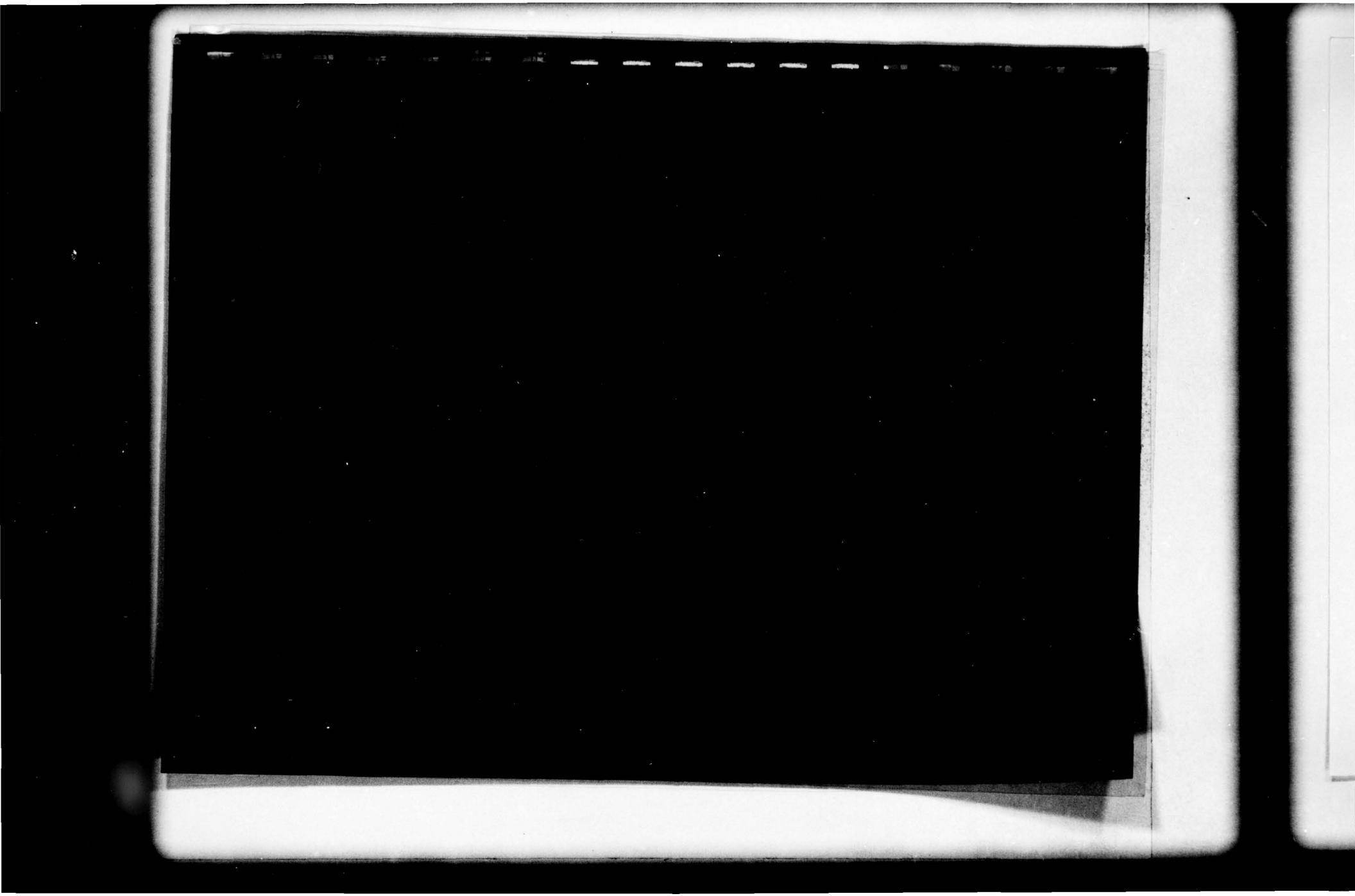
UNCLASSIFIED NSWC/DL-TR-3834 NL

1 OF 2
ADA
060639


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

DDC FILE COPY

AD A060639



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: NSWC/DL TR-3834	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): QM-1 MULTI MICRO-PROGRAMMER GUIDE	5. TYPE OF REPORT & PERIOD COVERED: Final rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): W.A. Burkard, R.D. Tuck, and R. L. Hartung	8. CONTRACT OR GRANT NUMBER(s): N60921-77-M-B785
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Surface Weapons Center (K74), Dahlgren Laboratory, Dahlgren, Virginia 22448	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: September 1978
	13. NUMBER OF PAGES: 98
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 98p.	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Micro-programming
Vertical Micro-programming
NANODATA QM-1

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

MULTI is a vertical micro-programming instruction set for the Nanodata QM-1 computer. It has been used to write computer emulations, an operating system kernal, and intermediate language machines. This document was written to serve as a reference manual for micro-programmers. It defines a standard programming environment for micro-programming the Nanodata QM-1 computer.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 912 78 10 19 058 LB

FOREWORD

This report is a micro-programmer's guide for the Nanodata Model QM-1 computer. The QM-1 is a dual-level micro-programmable computer. The higher of the two levels, called the micro level, is a vertical micro code which is interpreted by the lower level, called the nano level. This document is a description of the standard micro level called MULTI.

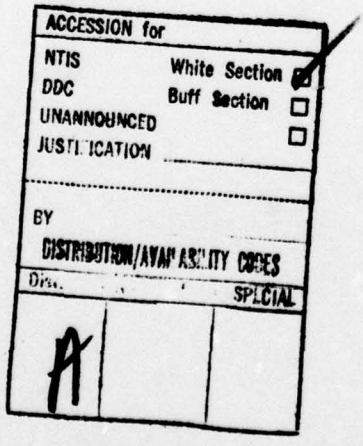
The main section of this document was produced by R. D. Tuck and W. A. Burkhard of the Computer Science Division, Department of Applied Physics and Information Science, University of California, San Diego, California, under Naval Surface Weapons Center, Dahlgren Laboratory (NSWC/DL) Contract No. N60921-77-M-B785.

This report was reviewed by H. W. Thombs, Head, Programming Systems Branch and W. P. Warner, Head, Computer Programming Division.

Released by:

Ralph A Niemann

R. A. NIEMANN, Head
Strategic Systems Department



CONTENTS

Page

INTRODUCTION 1

I. REGISTERS AND MEMORY 1

II. MULTI INSTRUCTION INVOCATION 5

III. ALU AND SHIFTER OPERATION 6

IV. THE STATUS BITS (FIST) 10

V. INTERRUPTS 11

VI. DESCRIPTION OF THE INSTRUCTIONS 13

APPENDIXES

A – MULTI NANO-PROGRAMMING CONVENTIONS 59
B – MULTI INSTRUCTION LISTS 63
C – MULTI INSTRUCTION REGISTER USAGE 69
D – QM-36 BIT ARCHITECTURE 73
E – UCSD EXTENSIONS 85

BIBLIOGRAPHY 92

DISTRIBUTION

MULTI INSTRUCTION SET SUMMARY
AND
QUICK REFERENCE INDEX

CS OPERATIONS

		PAGE
LD	Load to LS Reg.	18
ST	Store from LS Reg.	19
LDX	Load Indexed	20
STX	Store Indexed	20
LDY	Load and Index	20
STY	Store and Index	21
LDD	Load Direct	19
STD	Store Direct	19
RAD	Replace ADD	29
SW	Swap LS and CS	17
LDM	Load Multiple	21
STM	Store Multiple	22
PULL	Pop CS Stack	23

MS OPERATIONS

LDMS	Load from Main Store	24
STMS	Store to Main Store	24
SWMS	Swap with Main Store	25
LDMSX	Load MS and Index	25
STMSX	Store MS and Index	26

BRANCHING OPERATIONS

BPL	Branch Positive	41
BNG	Branch Negative	41
BZR	Branch Zero Register	42
BNZ	Branch Non Zero	42
BOS	Branch Ones Status	40
BZS	Branch Zero Status	40
BCT	Branch and Count	43

BRANCHING OPERATIONS (Continued) **PAGE**

B	Branch	43
BALN	Branch and Next Link	44
BALR	Branch and Link Reg.	44
JUMP	Table Jump	45

ARITHMETICS

LDI	Load Immediate	18
LCI	Load Complement Immediate	17
ADI	Add Immediate	28
SBI	Subtract Immediate	28
ADR	Add Register	28
SBR	Subtract Register	29
ORR	OR Register	30
ANR	AND Register	31
XOR	Exclusive OR Register	32
NTR	Not Register	30
NGR	Negate Register	30
MVR	Move Register	17
CPR	Compare Register	39
ALUX	General ALU Function and CPU Control Access	35
LDN	Load Next CS Word	18
ADN	Add Next CS Word	29
ORN	OR Next CS Word	31
ANN	AND Next CS Word	31
XON	XOR Next CS Word	32

SPECIAL OPERATIONS

EXTR	Split a Register	34
EXN	Execute Instruction in Next CS Word	45
EX	Execute Single	46
EXD	Execute Double	46
SBO	Set Bit to One	32
SBZ	Set Bit to Zero	33
TBO	Test Bit One	38

SPECIAL OPERATIONS (Continued) **PAGE**

TBZ	Test Bit Zero	39
SETALU	Set ALU Mode	47
STAT	Set Status Reg.	47
ENQ	Enqueue Function	54
DEQ	Dequeue Function	56
SYSTEM	Start System State	50

SYSTEM SUPPORT OPERATIONS

READS	Read Switches	48
AUX	Auxiliary Actions	48
SIDX	Set Nano Index	47
GENINT	General Interrupt	58
LDEI	Load ES. to LS.	27
STEI	Store LS. to ES.	27
XIO	Transmit I/O	57
RIO	Read I/O	58
HALT	Wait on Condition	48
LDMSA	Load MS Absolute	26
SAVE	Save Problem State	52
RESTORE	Restore Problem State	53
EXIT	Exit System State	53
STNS	Write Nanostore	27

SHIFTING OPERATION (Immediate)

SRAI	Single Rt. Arith.	33
SLLI	Single Lt. Logical	33
SRLI	Single Rt. Logical	34
SRCI	Single Rt. Circular	34
SHIFT	Double General Shifting	36

INTRODUCTION

MULTI is a nano-coded micro-instruction set making most of the resources of the QM-1 readily available to the micro-programmer. The MULTI architecture is vertical; each of its instructions corresponds to one micro-machine operation. It is also serial and sequential; an operation is completed by the time the next is begun, and (except for branches, skips, and interrupts) instruction words are taken from successive storage locations.

I. REGISTERS AND MEMORY

All words of Local Store, Control Store, External Store and Main Store are 18-bits wide and are numbered from 0 at the right to 17. at the left.

Central to the QM-1 architecture are 32 fast 18-bit registers collectively known as Local Store (or LS, but usually just referred to as "the registers"). Nearly all data transfers and transformations use these registers, some of which have special significance:

Symbolic		Octal		Decimal	
R.SYS	=	27	=	23.	
R.MX	=	30	=	24.	MPC Registers
R.IX	=	31	=	25.	
R.IY	=	32	=	26.	
R.MPC	=	33	=	27.	
R.ADR	=	34	=	28.	
R.TMP	=	35	=	29.	
R.SCR	=	36	=	30.	
R.IR	=	37	=	31.	

These symbols are predefined to the micro-assembler. It is strongly suggested that the user dedicate register zero to contain the value '0', as some parts of the operating system make this assumption. The symbol V is often used to designate a register containing zero and will be used in such a manner in this document. Referencing a register numbered greater than 31. as a source usually yields a '-1' result; eg: "BALN R1,77, B.ADDR+1" has the same effect as "BALN R1,V,B.ADDR."

R.SYS is used to pass information by system routines but is not actually used by MULTI.

R.MX is the address source for instructions LDMSA, LDMSX, and STMSX and is modified by them.

R.IX is used in address computations for LDX and STX but is not automatically modified by these instructions. R.IX and R.IY are used by ENQ and DEQ, as well as interrupts and are set by the EXIT instruction.

R.IY is used for address generation for LDY and STY, and is modified by these instructions, as well as being used as mentioned above.

R.MPC is the micro-program counter; during the execution of a MULTI instruction, it contains the address of that instruction (except for the object instruction of EX or EXD; see the description of those instructions for details). Programs modifying this register will generally not affect the execution of the immediately following instruction because of prefetching.

R.ADR is used as an address source by LDM, STM, PULL and STNS, and is modified by them. It is set by LDD, STD, and EXIT, used by XIO, RIO, SHIFT, ALUX, the SYSTEM instruction, and all interrupts.

R.TMP & R.SCR are used as working registers by the MULTI nano-programs. They are not saved across interrupts but, if operating with interrupts disabled, additional information may be gotten from them. In most cases, these registers may be used as sinks for unwanted data.

R.IR the instruction parameter register, contains the low-order 11. bits of an instruction word upon entry to that instruction; the top 7 bits are zero-filled. It is also used as a work register by several instructions. Modifying R.IR, in some cases, affects the execution of the following instruction.

Some instructions place additional register-use constraints on R.MX (= 24.) through R.MPC (= 27.), which are collectively referred to as the "MPC registers."

A second set of registers contained in the QM-1 are the 32., 6-bit F-registers. Most of these do not directly concern the micro-programmer, as they are invisible to him. Those which are not are:

FIDX (= F17.)

16 MODE	SUPER	RONS	NS PAGE INDEX
5	4	3	2 | 0

16 MODE = 0 – all arithmetic and tests are done on 18-bit quantities.

16 MODE = 1 – the test bits (inFIST) and the ALU are modified to reflect operations on 16-bit quantities (see the section on ALU and SHIFTER operations). This bit can be changed by the SETALU instruction.

SUPER = 0 – normal user micro-instruction execution

= 1 – supervisor state – designed to restrict user employment of certain micro-instructions. However, the hardware to support this feature has never been realized; all instructions are functionally unrestricted. This bit is automatically set by an interrupt.

RONS – controls access to read-only nano-store; normally zero.

NS PAGE INDEX – determines which nano-store page contains the entry points to the routines automatically entered by micro-instruction execution (see HLUM).

The entire FIDX register may be set by the SIDX instruction.

FIST (= F18.)

SHB	C	S	R	0	SLB
5	4	3	2	1	0

SHB – the high-order bit of the shifter output (normally bit 17. but bit 15. in 16-bit mode).

C – indicates that a carry has occurred from the high-order bit during an addition, or that a borrow has not occurred during a subtraction. The significance of this bit is altered by 16-bit mode.

S – the sign of an ALU result (bit 15. in 16-bit mode) or of a SHIFT result.

R – indicates that at least one of the bits of an ALU result to the right of the sign bit is a '1,' ie., R is set to zero for zero and the most negative 2's compliment number. The significance of this bit is altered in 16-bit mode and is greatly altered for a SHIFT.

0 – indicates that an overflow (result too large) has occurred during an ALU operation, or a double arithmetic left SHIFT. Its meaning after an ALU operation is affected by 16-bit mode.

SLB – the low-order bit of the result of a shifting operation.

C, S, R, and 0 are set by most ALU operations.

SHB and SLB are set by all shifting instructions.

All status bits are set by the SHIFT instruction.

The bits of this register are discussed in detail in the chapter on ALU and shifter operation. This register may be tested by the BZS and BOS instructions, and set by STAT. It is saved by interrupts and restored by EXIT.

Another set of QM-1 registers is the External Registers; 32 registers of 18 bits, many of which have special significance:

E0-E7 are the I/O port registers; it is through these that the CPU program communicates with the I/O channel(s) and I/O devices.

E16. & E17. are the Main Store Base Address and Field Length registers, used by MS relocation and protection (see Main Store description).

E18. & E19. are interrupt mask registers. Bits 14. through 0 of E18., when on, permit interrupts on levels 2-16., if interrupts are enabled. The corresponding bits of E19. control interrupt levels 17.-31.

E22.-E31. contain the nano-store addresses to be fetched and executed in case of an interrupt. Each register contains the six-bit representations of three addresses, one for each of levels 2 through 31. Address fields of the form 'abcdef' correspond to interrupt addresses of the form '0ab000cdef.'

These registers may be read by LDEI and SAVE, and set by STEI and RESTORE. The port registers are used by XIO and RIO, and can also be accessed by MS operators.

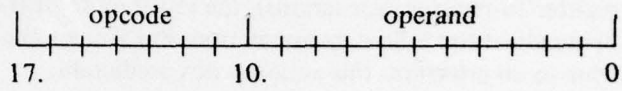
The QM-1 has three memories for program and data storage. Nano-store (NS) has a width of 360 bits, and an 80-nsec access time. It can be modified by the micro-programmer using STNS but is otherwise never changed. Micro-programs reside in Control Store (CS), which is 18 bits wide, and has 80-nsec access time. It can be read and modified by many of the load and store instructions. Interrupts store the micro-machine's status array in this storage as well. "Negative" CS addresses (those with bit 17. on) refer to Read-Only Control Store (ROCS) used for bootstrapping.

Object (emulated) machine code is usually held in Main Store (MS); 18-bit wide core storage. Access to Main Store is controlled by MS relocation and protection, if it is installed and enabled. In this case, instructions referring to MS, other than LDMSA, refer to locations whose address is the sum of the address specified in the instruction, with the contents of register E16. (MS Base). Register E17. contains the number of core locations available to the user (his field length); this inhibits his changing core locations outside of his field length. This feature is enabled and disabled by the AUX instruction. I/O data are transmitted to and from MS, and the control programs keep their overlays there.

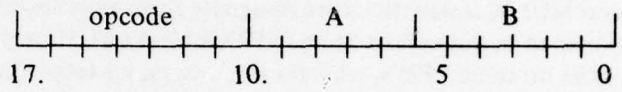
II. MULTI INSTRUCTION INVOCATION

MULTI instructions occupy either one or two 18-bit control-store words and have a seven-bit operation code. MULTI's op-codes may be changed by modifications to the nano-program. Those shown in this document are those of an unchanged MULTI machine. The individual instruction formats are shown with each instruction description, but the general types are:

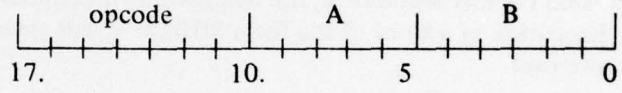
– one operand

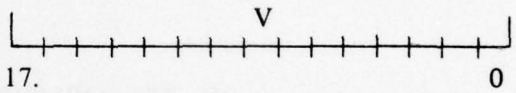


– two operands

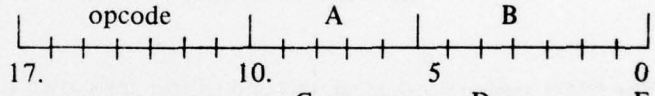


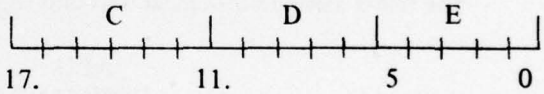
– three operands





– five operands





The first thing that most one-word micro-instructions do is to gate the control-store word (normally the following instruction) into an "invisible" CPU register. The last thing any instruction does is to start the fetching from memory of the second-next CS word (or, in the case of a branch, the word following the branched-to address) which is normally the 2nd following program word. Then, the MPC is incremented, to point to the next instruction (except for a branch), and execution of this instruction is invoked; parameters are brought from the invisible register to R.IR. Generally:

1. Using R.MPC as a source yields the address of the current instruction (first word), except when it is the object instruction of an EX or EXD.

2. Using R.MPC as a destination will not affect the fetch of the next program word, since this has already been done, unless an interrupt intercedes, breaking the fetch-ahead

chain. In this case, the program will resume at the address contained in R.MPC. Some instructions do not guarantee proper C.S. address timing if R.MPC is a destination; hence by convention, R.MPC will not be used as a destination.

3. If the immediately following program word is changed during an instruction's execution, it will nevertheless be executed as previously fetched unless, again, an interrupt intercedes.

4. Using R.IR as a destination should be done with care since, in some cases (particularly the register-to-register instructions), the result of an operation may be SCRAMBLED with the operands of the following instruction. But if the next instruction might be refetched due to an interrupt, this action is not predictable.

Several MULTI instructions are designated for supervisor use only, and supposedly can only be executed in Supervisor State (FIDX's bit 4 on). However, the protection mechanism does not exist on some CPU's, wherein the user may execute any instruction. Should an unassigned op-code be encountered, an unused nano-word will be read. Unless the user has made provision for this occurrence, the nano-word will contain all zeros and stop the machine. Execution of a word of the form $'010xxx'_8$ will cause a Program Check and the resulting interrupt.

III. ALU AND SHIFTER OPERATION

The QM-1 arithmetic-logical unit and shifter are shown in Figure 1.

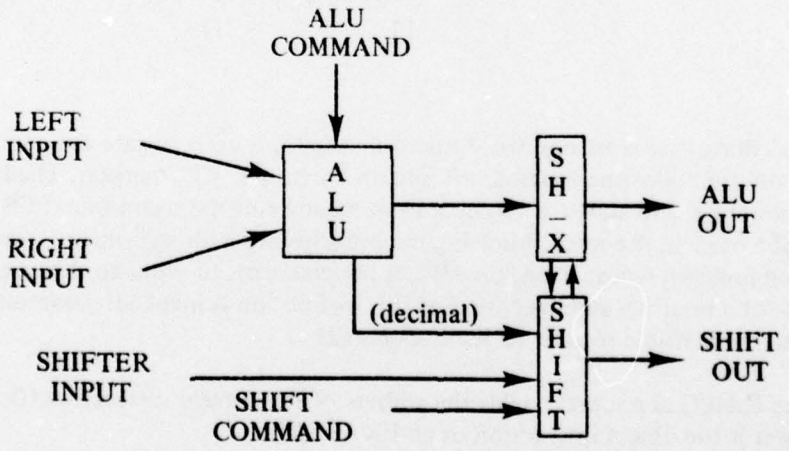


Figure 1. QM-1 Arithmetic-Logical Unit and Shifter

The ALU does 18-bit parallel operations, dependent on the ALU command code, which is invisible to the MULTI programmer, except for its use in the ALUX and SHIFT instructions. When 16-bit mode is set (bit 5 in FIDX), bit 15. of both the left and right inputs, and the output is duplicated in bits 16. and 17. of their respective busses. Thus, any arithmetic performed in 16-bit mode will still give a valid 18-bit result. This is not the case, however, for the ALU operation "PASS LEFT" (see table below); thus, MVR R,R will not sign-extend the contents of register R, whereas ADR R,V will.

When a decimal (BCD) operation is specified, a binary operation is (still) performed in the ALU, and a "decimal correction word" is generated in the shift circuit, in lieu of any shifting specified. If a carry-out is generated by a four-bit group during a decimal ALU operation, zeros are generated in the corresponding group in the shifter, otherwise '0110'$_2$ is generated; the two high-order bits are zeroed. During the ALUX instruction, any such correction factor is placed in register R.ADR. A decimal add of register B and the carry bit into register A can thus be achieved by the sequence (in 16-bit mode):

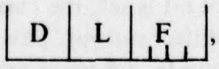
```
ALUX    A,A,51,R.SCR,V      ;   add carry-in, create fudge factor
ADR     A,R.ADR             ;   add in fudge factor
ALUX    A,A,51,R.SCR,B      ;   add augend
ADR     A,R.ADR             ;   add the correction factor, and
                            ;   set carry anew.
```

Decimal subtraction (with carry) is easier:

```
ALUX    A,A,46,R.SCR,B      ;   subtract
SBR     A,R.ADR             ;   correct
```

Recall that CARRY should normally be set before a subtraction. Note that a logical function specifying decimal correction will generate the same factor as would the corresponding arithmetic operation. A SHIFT instruction specifying a decimal ALU operation will have the following effect: the high-order destination word will receive the result of the operation; the low-order destination word will receive the correction factor; and no shifting will be done.

The ALU operation code is six bits in the form:



where D, when one, specified decimal correction, L specifies a logical operation, and F is the function code, as shown on the next page.

Function	Logical op	Arithmetic op
0	NOT L	L-1+C
1	NOT (L & R)	(L & R)-1 +C
2	NOT L v R	(L & NOT R)-1 +C
3	ALL ONES	-1 +C
4	NOT (L v R)	(L v NOT R) +1 +C
5	NOT R	(L v NOT R) + (L & R) +C
6	NOT (L x or R)	L - R -1 +C (subtract with borrow)
7	L v NOT R	(L v NOT R) +C
10	NOT L & R	(L v R) + L +C
11	L xor R	L + R +C (add with carry)
12	R	(L v R) + (L & NOT R) +C
13	L v R	(L v R) +C
14	ALL ZEROS	L + L +C
15	L & NOT R	L + (L & R) +C
16	L & R	L + (L & NOT R) +C
17	L (PASS LEFT)	L +C

("C" is the carry at the time of execution of an ALUX, or is zero for SHIFT, L is the left input to the ALU, and R its right input.)

The shifter and shifter extension operate on 18. or 36. bit quantities; their repertoire includes logical, circular, and arithmetic (sign-extending) shifts. A logical shift is equivalent to an unsigned multiply or integer divided by the specified power of two. Zeros are shifted in at either end. Right arithmetic shifts, instead of shifting in zeros on the left propagate the high-order bit to the right. A left arithmetic shift is identical with a left logical shift, except in the way the overflow bit is set (see below). Circular shifting involves the replacement of shifted-out bits at the opposite end of the shifted quantity.

Associated with the shift circuitry are the right and left control switches, which have this effect upon the result of a SHIFT instruction: when the left-control bit is set, the low-order bit of the result is obtained from the (old) value of the carry bit, rather than the shifter output. When the right-control bit is set, the carry is set from the least significant bit of the shifter input, rather than the shifter output. This circuitry facilitates shifting quantities of more than two words. For example, a 4-word arithmetic right shift of the contents of registers A, A+1, A+2, A+3 by the count contained in register C may be coded as:

```
SBI     C,1
BNG     C,*+6   ;  in case of zero shift count
SHIFT   A+1,A+1,37,13+20,1   ;  arith. double right with right control
SHIFT   A+3,A+3,37,7,1   ;  logical double right brings in carry
BCT     C,*-4
```

And a 3-word logical left shift of the contents of registers A, A+1, A+2 is:

```
SBI     C,1
BNG     C,*+6
SHIFT   A+2,A+2,37,6,1  ; logical double left sets carry
SHIFT   A,A,37,4+40,1   ; logical single left with left control
BCT     C,*-4
```

These bits also affect the value of the R (result) bit following a SHIFT.

The shifter op-code utilized in the SHIFT instruction is six bits wide and has the format:

Where

LC = left control,
RC = right control,
T = shift type: 00 – circular
01 – logical
10 – arithmetic

M = shift mode: 0 – single shift (shifter extension bypassed)
1 – double shift

and

D = shift direction: 0 – left shift
1 – right shift.

The use of the latter three fields can be summarized as:

Code	Circular Logical Arithmetic	Single Double	Left Right
0	C	S	L
1	C	S	R
2	C	D	L
3	C	D	R
4	L	S	L
5	L	S	R
6	L	D	L
7	L	D	R
10	A	S	L
11	A	S	R
12	A	D	L
13	A	D	R

Shift counts greater than the length of the quantity being shifted (i.e.; 18. or 36.) are to be avoided; circular shifts with such a count give unpredictable results.

IV. THE STATUS BITS (FIST)

SHB – shifter high bit – The high-order bit of a shifter result (low-order word of a double-length shift), this bit usually is set by result bit 17. but when in 16-bit mode is taken from bit 15.

C – carry bit – The carry-out bit of an addition, the complement of a borrow in subtraction, it is also used as a carry-in for the ALUX instruction. Double precision addition, (A-1)//(A) <= ((A-1)//(A)) + ((B-1)//(B)) is thus:

```
ADR   A,B
ALUX  A-1,A-1,11,R.SCR,B-1 ;
      and subtraction:
SBR   A,B
ALUX  A-1,A-1,6,R.SCR,B-1.
```

In 16-bit mode, carry is set to the value of bit 16. of the arithmetic result.

The carry bit also participates in the SHIFT instruction: its value at instruction invocation is used, by a double right logical shift operation, as though this bit were to the left of the most significant part of the argument (i.e.; it is the first bit shifted in and is followed by zeros). In 18-bit (normal) mode, SHIFT sets the carry as if it were to the left of the high-order word in a double logical left shift of the same argument and count, for any left shift; and to what the low-order bit for a double logical right shift would be, for any right shift, except that it is not set for a count of zero.

When in 16-bit mode, the carry is set a little differently; for any left shift, it is set to what bit 16. of the high order word of the corresponding double logical left shift would be. For a right shift of count zero, carry is zero, for a count of one it is the least significant bit of the argument and for a count of c: $2 \leqslant c \leqslant 36$. it is what the second-last bit shifted off the end would be for a corresponding double right logical shift. However, if the right control bit is set (as above), carry is set from the least significant bit of shifter input.

S – sign – Normally bit 17. of the result of an ALU operation, or of the high-order word of a SHIFT result; it is set from bit 15. in 16-bit mode.

R – result – The OR of the bits of an ALU result which are to the right of the sign bit (16.-0 in 18-bit mode, 14.-0 in 16-bit mode), or bits 16.-0 of the high-order word

of a double-length SHIFT. However, when either of the right/left control bits is set, all 18. bits of the shifter result are included in the logical sum.

O – overflow – This bit is generally used to indicate that an overflow (result too large) has occurred during an ALU operation. It is the exclusive OR of the carry into the sign bit (bit 15. of 16-bit mode) with the carry out of that same bit position. It is also set by the SHIFT instruction in the following manner: if, during a double arithmetic left shift, a bit-value is shifted into (or through) bit position 17. of the high-order word which is not the same as the starting value of that position, then overflow is signaled.

SLB – shifter low bit – This bit is set from the low-order bit of a shifter result.

V. INTERRUPTS

The QM-1 hardware has provisions for 30 different external sources of interrupts (levels 2-31.). The MULTI nano-program provides two more sources: program-detected Program Check, and the SYSTEM instruction. Each interrupt source has a priority relative to the others: Program Check has the highest and is signaled as soon as it is detected; the external interrupt levels follow, in ascending order. SYSTEM, as any other MULTI instruction, is executed when no interrupts are pending; once begun, it cannot be interrupted. The external interrupts may be enabled or disabled by:

- XIO command to the channel or device.
- use of the interrupt mask bits in registers E18. and E19.
- the AUX instruction.

The external interrupt levels on the UCSD QM-1 are assigned in the following manner:

Level	USE
8.	channel 3 data-in
9.	channel 3 data-out
10.	channel 4 data-in
12.	channel 4 data-out
14.	channel 6 data-in
18.	GENINT 18. instruction execution
19.	GENINT 19. instruction execution
25.	channel 3 status
26	channel 4 status
28.	channel 6 status
30.	clock

All other levels are unused and disabled by masking.

Data-in and data-out interrupts signify that a device on that I/O channel is ready to transmit data to or from the computer. Except for channel 6 (the "inter-computer" channel), these are handled solely by the nano-code, which supervises the transmittal of data between channel and Main Store. Thus, these interrupts are generally unnoticed by the micro-programmer, except that they will cause a refetch of his next instruction.

A status interrupt indicates a condition in the channel or device which the nano-programming will not handle: either an error has occurred, or the device is ready to accept a new command (see HLUM § 8.4.2.2).

Status interrupts, clock interrupts, Program Checks, GENINT instruction execution specifying levels 18., 19., 25., 26., 28., or 30., or SYSTEM instruction execution causes the following to happen (micro-program interrupt):

1. A micro-program state-save array is stored in CS. It is of the form:

Relative CS word	Contents
0	FUSR FACT FIST*
1	FIDX FMPC G11
2	R.ADR
3	R.MPC
4	R.IY
5	R.IX

*FUSR, FACT, FMPC, and G11 are registers F15., F14., F16., and F31., and need not concern the MULTI micro-programmer. Execution of a PCHECK word stores zeros in the space allocated for FACT and FIDX.

This array is stored in absolute CS locations, which are dependent on the type of interrupt.

2. Register R.IX is loaded from an interrupt-dependent location (to address an appropriate device control block), and R.ADR is loaded with data identifying the interrupt source. R.IY will contain TASK.BASE+100_8 (see below).

3. Micro-instruction execution is resumed at an interrupt-dependent location. Interrupts are disabled, and 18-bit, Supervisor mode is entered.

TASK.BASE is a nano-assembly-time variable which is communicated to the micro-assembler by way of the symbol table. It is the intended beginning address of the supervisor program; for a 10K CS installation, it is set to 13000_8. TASK.QUEUE is usually set to

TASK.BASE+100_8. The interrupt-dependent locations are all offsets from this address, and are summarized in Table 1.

Table 1. Interrupt-Dependent Locations

	R.IX Source	State-Save Pointer	R.MPC Source	R.ADR Contents
	TASK.BASE +	TASK.QUEUE +	TASK.QUEUE +	
PCHECK word	74	02	03	00ff74*
SYSTEM op	76	00	01	00bb74**
GENINT 18. op	70	00	01	001770
GENINT 19. op	71	00	01	001771
Channel 3 status	03	00	01	000303
Channel 4 status	04	00	01	000404
Channel 5 data-in	72	00	01	000572
Channel 5 data-out	73	00	01	000573
Clock	75	00	01	000075

* ff = contents of FUSR at error
**bb = operand of SYSTEM instruction

The return from an interupt routine is generally realized via the EXIT instruction, which restarts a micro-program using a state-save array of the format shown above.

VI. DESCRIPTION OF THE INSTRUCTIONS

Each MULTI instruction is represented as in the following model:

"HALT AB *supervisor* **OPCODE = 000**

Halt if switches set ⌊ 0000000 ⌋ ⌊ abababababa ⌋

```
procedure HALT(AB:0. .2047);
begin
      if AND(AB,SW)≠0 then
            HALT(AB)
end;
```

If the switches masked by the six low-order bits of the argument 'AB' are not all off (down), then restart this instruction."

The representation contains the following information:

– The instruction mnemonic, followed by the correct number of operands, an indication if it is a supervisor instruction, and the instruction's op-code in octal. The bits of the latter are grouped 3-3-1, as they would appear in an octal dump.

– The "title" of the instruction, and its binary representation, showing all its fields.

– A Pascal representation of the instruction's execution. In several cases, this has been omitted, as it would lend more confusion than clarity.

– A verbal description of the instruction's actions.

The PASCAL global declarations assumed are:

```
const   SYS  = 23;
        MX   = 24;
        IX   = 25;
        IY   = 26;
        MPC  = 27;
        ADR  = 28;
        TMP  = 29;
        SCR  = 30;
        IR   = 31;

const   MSBASE    = 16;
        MSFL      = 17;

const   IDX       = 17;
        IST       = 18;

const   CSMAX     = ; {Control Store size}
        COREMAX   = ; {Main Store size   }

type    FIVBIT    = 0..31;
        SIXBIT    = 0..63;
        SIGNSIX   = -32..31;
        MSAD      = 0..COREMAX; {type for Main Store addresses}

var     R array [0..31] of integer; {Local Store}
        CS array [0..CSMAX] of integer; {Control Store}
        MS array [0..COREMAX] of integer; {Main Store}
        E array [0..31] of integer; {External Store}
        F array [0..31] of SIXBIT; {F-registers}
```

```
var     RELMS, {MS relocation and protection switch}
        INTERN: boolean; {interrupt enable switch}
        SW: SIXBIT {switch register}

        function MSADDR(X:MSAD):MSAD;
        begin
              if RELMS then
                    MSADDR: =X + E[MSBASE]
              else MSADDR:=X
        end;
```

The micro-instruction counter, R.MPC, is mentioned only when sequential instruction processing is not assumed, otherwise, sequential instruction processing is assumed.

A new construction has been added to standard PASCAL: the "beginp . . endp" block. Within this compound statement, statements are separated by commas (,), and are assumed to be executed in parallel, rather than sequentially. The following integer-type functions are assumed:

COMPLEMENT(A) – The bit-inversion function. If the nth bit of its argument is a '1,' the corresponding bit of the resultant is a '0,' and vice versa. e.g.; COMPLEMENT $(1110010101000111101_2) = 0001101010111000010_2$.

OR(A,B) – If the nth bit of either or both of the argument is a '1,' then the corresponding bit of the resultant is also a '1,' otherwise, it will be a '0,' EG: OR $(1111111110000000002, 1010101010101010102) = 1111111110101010102$.

The relation is better shown in this table:

A \ B	0	1
0	0	1
1	1	1

OR

AND(A,B) – If the nth bit of both of the arguments is a '1,' then the corresponding bit of the resultant is also a '1,' otherwise it will be a '0,' EG: AND $(1111111110000000002$, $1010101010101010102) = 1010101010000000002$.

This relation can be tabulated as:

A \ B	0	1
0	0	0
1	0	1

AND

XO(A,B) – The exclusive-or function – If the nth bit of exactly one of the arguments is a '1,' then the corresponding bit of the resultant is also a '1,' otherwise it will be a '0,' e.g.;

XOR (1111111100000000_2, 1010101010101010_2) = 0101010100101010_2.

A \ B	0	1
0	0	1
1	1	0

Note that XOR(A,B) =

AND(OR(A,B),COMPLEMENT(AND(A,B)))

SRA(A,B) – Shift Right Arithmetic – Returns the value of 'A' shifted right arithmetic (sign extended) by the shift count 'B.' See the section on ALU and SHIFTER operations for an explanation of shifts.

SRL(A,B) – Shift right logical of 'A' by 'B' bits.

SLL(A,B) – Shift left logical of 'A' by 'B' bits.

SRC(A,B) – Shift right circulate (end around) of 'A' by 'B' places.

SETALUSTAT – Sets the ALU status bits in FIST (O,R,C,S) on the outcome of the preceding arithmetic or logical operation.

SETSHSTAT – Sets the shifter status bits in FIST (SLB,SHB) according to the outcome of the preceding shifter operation.

MVR A,B timing = 5T OPCODE = 450

Move register

1001010	aaaaa	bbbbbb

```
procedure MVR(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := R[B]
end;
```

Register A is loaded with the contents of register B.

SW A,B timing = 10T OPCODE = 230

Swap register with Control Store

0100110	aaaaa	bbbbbb

```
procedure SW(A:FIVBIT,B:SIXBIT);
begin
      R[IR]  := CS[R[B]];
      beginp
            CS[R[B]]  := R[A],
            R[A]  := R[IR]
      endp
end;
```

The contents of register A are exchanged with those of the Control Store word addressed by the contents of register B.

LCI A,B timing = 5T OPCODE = 030

Load Complement Immediate

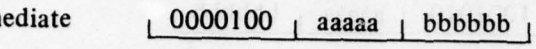
0000110	aaaaa	bbbbbb

```
procedure LDC(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := COMPLEMENT(B)
end;
```

Register A is loaded with the 18-bit complement of the six bit value 'B.'

LDI A,B Timing = 5T OPCODE = 020

Load Immediate

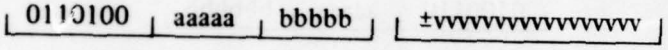



```
procedure LDI(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := B
end;
```

Register A is loaded with the six bit, unsigned, value 'B,' right justified.

LDN A,B,V timing = 8T OPCODE = 320

Load Next Word

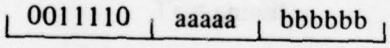



```
procedure LDN(A:FIVBIT,B:SIXBIT,V:integer);
begin
      R[A]  := R[B]  + V
end;
```

Register A is loaded with the sum of the contents of register B with the 18-bit, signed, value 'V.'

LD A,B timing = 7T OPCODE = 170

Load from Control Store

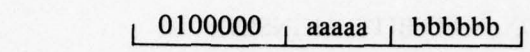



```
procedure LD(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := CS[R[B]]
end;
```

Register A is loaded with the contents of the Control Store word addressed by the contents of register B.

ST A,B timing = 7T OPCODE = 200

Store to Control Store

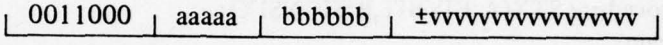



```
procedure ST(A:FIVBIT,B:SIXBIT);
begin
      CS[R[B]]  := R[A]
end;
```

The contents of register A are stored in the Control Store word addressed by the contents of register B.

LDD A,B,V timing = 10T OPCODE = 140

Load Direct

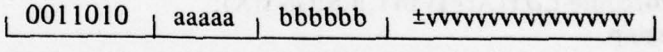



```
procedure LDD(A:FIVBIT,B:SIXBIT,V:integer)
begin
      R[ADR]  := R[B]  + V;
      R[A]  := CS[R[ADR]]
end;
```

Register R.ADR is loaded with the sum of the contents of register B with the signed 18-bit operand 'V.' Register A is loaded with the contents of the Control Store word addressed by that sum. Register B may not be an MPC register.

STD A,B,V timing = 10T OPCODE = 150

Store Direct

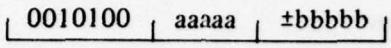



```
procedure STD(A:FIVBIT,B:SIXBIT,V:integer)
begin
      R[SCR]  := R[B]  + V;
      beginp
            CS[R[SCR]]  := R[A],
            R[ADR]  := R[SCR]
      endp
end;
```

The contents of register A are stored in the Control Store word addressed by the sum of the contents of register B with the 18-bit signed operand 'V.' Simultaneously, register R.ADR is loaded with this sum. Register B may not be an MPC register.

LDX A,B timing = 8T OPCODE = 120

Load Indexed

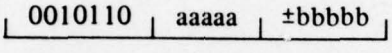



```
procedure LDX(A:FIVBIT,B:SIGNSIX);
begin
      R[A]  := CS[R[IX]  + B]
end;
```

Register A is loaded with the contents of the Control Store word addressed by the sum of the contents of register R.IX with the signed six bit operand 'B.'

STX A,B timing = 8T OPCODE = 130

Store Indexed

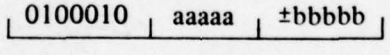



```
procedure STX(A:FIVBIT,B:SIGNSIX);
begin
      CS[R[IX]  + B]  := R[A]
end;
```

The contents of register A are stored in the Control Store word addresses by the sum of the contents of register R.IX with the signed six bit operand 'B.'

LDY A,B timing = 8T OPCODE = 210

Load using IY register

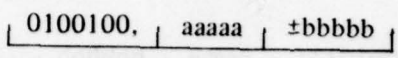



```
procedure LDY(A:FIVBIT,B:SIGNSIX);
beginp
      R[A]  := CS[R[IY]  + B],
      R[IY]  := R[IY]  + B
endp;
```

Register A is loaded with the contents of the Control Store word addressed by the sum of the contents of register R.IY with the signed six bit operand 'B.' Simultaneously, register R.IY is modified to contain that address.

STY A,B timing = 8T **OPCODE = 220**

Store using register IY

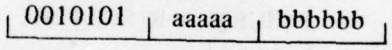



```
procedure STY(A:FIVBIT,B:SIGNSIX)
beginp
      CS[R[IY] + B] := R[A],
      R[IY] := R[IY] + B
endp;
```

The contents of register A are stored in the Control Store word addressed by the sum of the contents of register R.IY with the signed six bit operand 'B.' Simultaneously, register R.IY is modified to contain that address. Note that LDY and STY cannot be used as a "push-pop" pair of instructions, since they both use the modified address to reference Control Store.

LDM A,B timing = 10 + 3(words)T **OPCODE = 124**

Load Multiple Registers

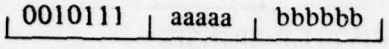



```
procedure LDM(A:FIVBIT,B:SIXBIT);
var G,C:SIXBIT;
begin
      G := A;
      C := B;
      beginp
            R[C] := CS[R[ADR]],
            R[SCR] := R[ADR] + 1,
            C := C+1
      endp;
            while G ≠ 0 do
                  beginp
                        R[C] := CS[R[SCR]],
                        R[SCR] := R[SCR] + 1,
                        C := C+1,
                        G := G-1
                  endp;
            R[ADR] := R[SCR] - 1
end;
```

Register B and the following 'A' registers are loaded with the contents of as many Control Store words: that addressed by the contents of register R.ADR, and its successors. Register R.ADR is then modified to address the CS location following the last one loaded. Loading register R.SCR using this instruction is to be avoided, since this register is being used as the CS pointer during instruction execution. This instruction is an exception to the general pre-fetching rule, and changing register R.MPC using it will result in the succeeding execution of the newly addressed program word, so long as register R.MPC is not the last register loaded.

STM A,B timing = 10 + 3(words)T OPCODE = 134

Store Multiple Registers




```
procedure STM(A:FIVBIT,B:SIXBIT);
var G,C:SIXBIT
begin
      G  := A;
      C  := B;
      beginp
            CS[R[ADR]]  := R[C],
            R[SCR]  := R[ADR]  + 1,
            C  := C+1
      endp;
            while G ≠ 0 do
                  beginp
                        CS[R[SCR]]  := R[C],
                        R[SCR]  := R[SCR]  + 1,
                        C  := C+1,
                        G  := G-1
                  endp;
            R[ADR]  := R[SCR]  -1
      end;
```

The contents of register B and the 'A' following registers are stored in the Control Store word addressed by the contents of register R.ADR, and its successor locations. The contents of the latter register are then modified to address the location following the last one stored. This instruction is an exception to the general prefetching rule, and can change the immediately following instruction.

PULL A,B timing = 12 + 3(words)T OPCODE = 114

"Pull" (i.e. pop) Registers

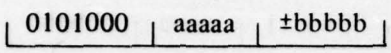
0010011	aaaaa	bbbbbb

```
procedure PULL(A:FIVBIT,B:SIXBIT);
var G,C:SIXBIT;
begin
      G  := A;
      C  := B;
      R[ADR]  := R[ADR]-A-1;
      beginp
            R[SCR]  := R[ADR]+1,
            R[C]  := CS[R[ADR]],
            C  := C+1
      endp;
            while G ≠ 0 do
                  beginp
                        R[C]  := CS[R[SCR]],
                        R[SCR]  := R[SCR]+1,
                        C  := C+1,
                        G  := G-1
                  endp
      end;
```

Register B and its 'A' successors are loaded with the contents of the 'B' Control store words preceding the one addressed by the contents of register R.ADR, in ascending order. Previous to this operation, the contents of register R.ADR are changed to address the first word loaded. PULL can be used in conjunction with STM to save and restore registers (as during a subroutine invocation) without the program's otherwise having to modify the contents of register R.ADR. This instruction is an exception to the general pre-fetch convention, and modifying register R.MPC will affect an immediate branch to the newly addressed location, provided that R.MPC is not the last register thus loaded.

LDMS A,B timing = 16T OPCODE = 240

Load from Main Store

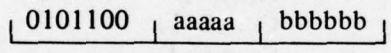



```
procedure LDMS(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := MS[MSADDR(R[B])]
end
```

Register A is loaded with the contents of the Main Store word addressed by the contents of register B. If MS relocation and protection is installed and active, then the address used is actually the sum of the contents of register B with those of the MS base register (E[16.]). If the addressed word is not installed, zero is loaded.

STMS A,B timing = 16T OPCODE = 260

Store to Main Store




```
procedure STMS(A:FIVBIT,B:SIXBIT);
begin
      if (not RELMS or R[B] ≤ E[MSFL])) then
            MS[MSADDR(R[B])  := R[A]
end;
```

The contents of register A are stored in the Main Store location addressed by the contents of register B. If MS relocation and protection is installed and active, then the address must be within the user's field length (E[17.]) for the store operation to occur, and the actual address used is the sum of the contents of register B with those of the MS base register (E[16.]).

SWMS A,B timing = 16T **OPCODE = 250**

Swap register with Main Store

0101010	aaaaa	bbbbbb

```
procedure SWMS(A:FIVBIT,B:SIXBIT);
var ADDR:0. .COREMAX;
begin
      ADDR  := MSADDR(R[B]);
      R[IR]  := MS[ADDR];
      beginp
            if not RELMS or (R[B] ≤ E[MSFL]) then
                  MS[ADDR]  := R[A],
            R[A]  := R[IR]
      endp
end;
```

The contents of register A are exchanged with those of the Main Store location addressed by the contents of register B. If MS relocation and protection is installed and active, then the address must be within the user's field length (E[17.]) for the store to take place, and the actual address used is the sum of the contents of register B with those of the MS base register (E[16.]).

LDMSX A, B timing = 16T **OPCODE = 270**

Load indexed from Main Store

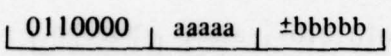
0101110	aaaaa	±bbbbb

```
procedure LDMSX(A:FIVBIT,B:SIGNSIX);
begin
      R[A]  := MS[MSADDR(R[MX])];
      R[MX]  := R[MX]  + B
end;
```

Register A is loaded with the contents of the Main Store word addressed by the contents of register R.MX. If MS relocation and protection is installed and active, then the address used is actually the sum of the contents of register R.MX with those of the MS base register (E[16.]). After the load operation is performed, the signed, six-bit operand 'B' is added into register R.MX.

STMSX A,B timing = 16T OPCODE = 300

Store indexed to Main Store

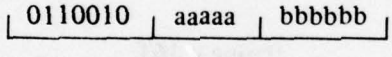



```
procedure STMSX(A:FIVBIT,B:SIGNSIX);
begin
      if not RELMS or (R[MX] ≤ E[MSFL]) then
            MS[MSADDR(R[MX])] := R[A];
end; R[MX] : R[MX] + B
```

The contents of register A are stored in the Main Store location addressed by the contents of register R.MX. If MS relocation and protection is installed and active, then the address must be within the user's field length (E[17.]) for the store to occur, and the actual address used is the sum of the contents of register R.MX with those of the MS base register (E[16.]). After the store operation is (or is not) performed, the signed, six-bit operand 'B' is added into register R.MX.

LDMSA A,B supervisor timing = 16T OPCODE = 310

Load from Main Store absolute address

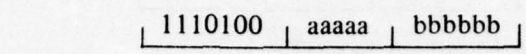



```
procedure LDMSA(A:FIVBIT,B:SIXBIT);
begin
      R[A] := MS[R[MX]];
      R[MX] := R[MX] + 1;
      R[B] := R[B] + R[A]
end;
```

Load register A with the contents of the Main Store location addressed by the contents of register R.MX, bypassing MS relocation and protection. Register R.MX is incremented, and the contents of register A are added into register B, specifically for forming a checksum while reading overlays from Main Store to Control Store.

LDEI A,B supervisor timing = 5T OPCODE = 720

Load from External Store



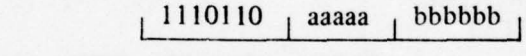

```
procedure LDEI(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := E[B]
end;
```

Register A is loaded with the contents of External Store register B.

Specifying $0 \leqslant B \leqslant 7$ designates an I/O port register.
Specifying B=16. designates the MS base register.
Specifying B=17. designates the MS filled length register.
Specifying $18. \leqslant B \leqslant 19$, or $22. \leqslant B \leqslant 31.$ designates I/O mask or interrupt nano-address register.

STEI A,B supervisor timing = 5T OPCODE = 730

Store to External Store

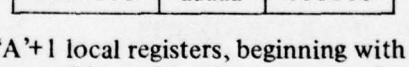



```
procedure STEI(A:FIVBIT,B:SIXBIT);
begin
      E[B]  := R[A]
end;
```

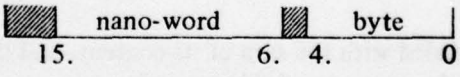
The contents of register A are stored in External Store register B.

STNS A,B supervisor timing = 7 + 5(words)T OPCODE = 064

Store to Nano-Store



Store the contents of 'A'+1 local registers, beginning with register B in nano-store locations beginning with the one addressed by the contents of register R.ADR. This address is in the following format:



The contents of register R.ADR are incremented following each store operation.

There are twenty (0-19.) 18-bit nano-bytes in each 360-bit nano-word. Byte addresses are modulo 32, and an illegal byte address (i.e. one greater than 19.) causes no data to be written. Nano-bytes are numbered left to right.

ADI A,B timing = 5T OPCODE = 040

Add Immediate | 0001000 | aaaaa | bbbbbb |

```
Procedure ADI(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := R[A]  + B;
      SETALUSTAT
end;
```

Register A is loaded with the sum of its contents with the six-bit unsigned operand 'B,' and ALU status bits are set to reflect the result.

SBI A,B timing = 5T OPCODE = 050

Subtract Immediate | 0001100 | aaaaa | bbbbbb |

```
procedure SBI(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := R[A]-B;
      SETALUSTAT
end;
```

Register A is loaded with the difference of its contents and the six-bit unsigned operand 'B.' ALU status bits are set to reflect the result of this operation.

ADR A,B timing = 5T OPCODE = 370

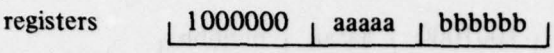
Add registers | 0111110 | aaaaa | bbbbbb |

```
procedure ADR(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := R[A]  + R[B];
      SETALUSTAT
end;
```

Register A is loaded with the sum of its contents and those of register B. ALU status bits are set to reflect the outcome of this operation.

SBR A,B timing = 5T OPCODE = 400

Subtract registers

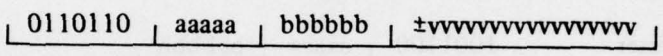



```
procedure SBR(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := R[A]-R[B];
      SETALUSTAT
end;
```

Register A is loaded with the difference of its contents and those of register B. ALU status bits are set to reflect the outcome of this operation.

ADN A,B,V timing = 8T OPCODE = 330

Add next word

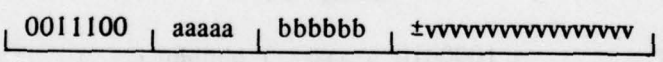



```
procedure ADN(A:FIVBIT,B:SIXBIT,V:integer);
begin
      R[A]  := R[B]  + V;
      SETALUSTAT
end;
```

Register A is loaded with the sum of the contents of register B with the 18-bit, signed, operand 'V.' ALU status bits are set to reflect the outcome of this operation. This instruction is identical with LDN except for the setting of status bits. Since V is temporarily stored in register R.IR, specifying register B as R.IR has the effect R[A] := V+V.

RAD A,B,V timing = 15T OPCODE = 160

Replacement add

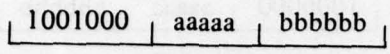



```
procedure RAD(A:FIVBIT,B:SIXBIT,V:integer);
begin
      R[SCR]  := R[B]  + V;
      R[IR]  := CS[R[SCR]];
      R[A]  := R[A]  + R[IR];
      SETALUSTAT;
      CS[R[SCR]]  := R[A]
end;
```

Register A is loaded with the sum of its contents and those of the Control Store word addressed by the sum of the contents of register B with the 18-bit signed operand 'V.' ALU status bits are set by the result of this operation. The contents of register A are then stored in the addressed Control Store word. Specifying register A as R.IR will add the contents of the storage location to itself without changing any registers.

NTR A,B timing = 5T OPCODE = 440

NOT register

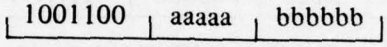



```
procedure NTR(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := COMPLEMENT(R[B]);
      SETALUSTAT
end;
```

Register A is loaded with the logical complement of the contents of register B. ALU status bits are set by the result.

NGR A,B timing = 5T OPCODE = 460

Negate register

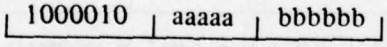



```
procedure NGR(A:FIVBIT,B:SIXBIT);
begin
      R[IR]  := 0;
      R[A]  := R[IR] - R[B];
      SETALUSTAT
end;
```

Register A is loaded with the two's complement of the contents of register B. This is achieved by subtracting the contents of register B from zero. ALU status bits are set by the result.

ORR A,B timing = 5T OPCODE = 410

OR register

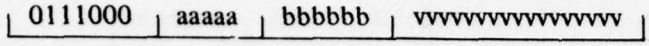



```
procedure ORR(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := OR(R[A],R[B]);
      SETALUSTAT
end;
```

Register A is loaded with the logical OR of its contents with those of register B. ALU status bits are set by this operation.

ORN A,B,V timing = 8T OPCODE = 340

OR next word

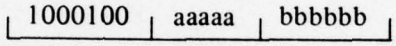



```
procedure ORN(A:FIVBIT,B:SIXBIT,V:integer);
begin
      R[A]  := OR (R[B],V);
      SETALUSTAT
end;
```

Register A is loaded with the logical OR of the contents of register B and the 18-bit operand 'V.' ALU status bits are set by its operation. Since V is temporarily stored in register R.IR, specifying register B as that register has the effect: R[A] := V.

ANR A, B timing = 5T OPCODE = 420

AND register

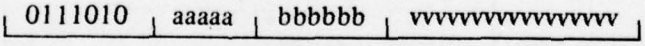



```
procedure ANR(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := AND(R[A],R[B]);
      SETALUSTAT
end;
```

Register A is loaded with the logical AND of its contents with those of register B. ALU status bits are set by this operation.

ANN A,B,V timing = 8T OPCODE = 350

AND next word

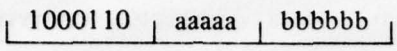



```
procedure ANN(A:FIVBIT,B:SIXBIT,V:integer);
begin
      R[A]  := AND(R[B],V);
      SETALUSTAT
end;
```

Register A is loaded with the logical AND of the contents of register B and the 18-bit operand 'V.' ALU status bits are set by this operation.

XOR A,B timing = 5T OPCODE = 430

Exclusive-OR registers

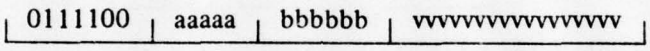



```
procedure XOR(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := XO (R[A],R[B]);
      SETALUSTAT
end;
```

Register A is loaded with the logical exclusive-OR of its contents with those of register B. ALU status bits are set by this operation.

XON A,B,V timing = 8T OPCODE = 360

Exclusive-OR next word

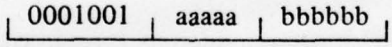



```
procedure XON(A:FIVBIT,B:SIXBIT,V:integer);
begin
      R[A]  := XO(R[B],V);
      SETALUSTAT
end;
```

Register A is loaded with the logical exclusive-OR of the contents of register B and the 18-bit operand 'V.' ALU status bits are set to reflect the outcome of this operation.

SBO A,B timing = 7T OPCODE = 044

Set bit to one

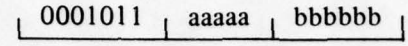



```
procedure SBO(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := OR(SRC(R[A],B),000001);
      R[A]  := SCR(R[A],18-B)
end;
```

The 'B'th bit of the contents of register A (right to left, zero origin) is set to a one. This is accomplished by shifting, OR-ing and shifting again. Avoid bit counts greater than 18.

SBZ A,B timing = 7T OPCODE = 054

Set bit to zero

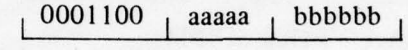



```
procedure SBZ(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := AND(SRC(R[A],B), 777776₈);
      R[S]  := SRC(R[A],18-B)
end;
```

The 'B'th bit of the contents of register A (counting right to left, zero origin) is set to a zero. This is accomplished by shifting, AND-ing and shifting again. Bit counts greater than 18 are to be avoided.

SRAI A,B timing = 5T OPCODE = 060

Shift right arithmetic

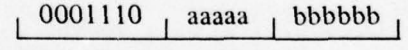



```
procedure SRAI(A:FIVBIT,B:SIXBIT)
begin
      R[A]  := SRA(R[A],B);
      SETSHSTAT
end;
```

Register A is loaded with its contents shifted right arithmetically (with sign extension) by 'B' places. Shifter status bits in FIST are set to reflect the result of this operation. Shift counts of 18. or greater will extend the sign throughout the entire register.

SLLI A,B timing = 5T OPCODE = 070

Shift left logical

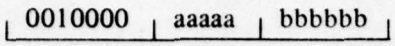



```
procedure SLLI(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := SLL(R[A],B);
      SETSHSTAT
end;
```

Register A is loaded with its contents shifted left logically (with zeros shifted in on the right) by 'B' places. Shifter status bits are set by the shifting operation. Shift counts of 18. or greater will give a zero result.

SRLI A,B timing = 5T OPCODE = 100

Shift right logical

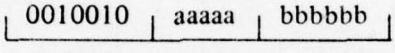



```
procedure SRLI(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := SRL(R[A],B);
      SETSHSTAT
end;
```

Register A is loaded with its contents shifted right logically (with zeros shifted in on the left) by 'B' places. Shifter status bits are set by this operation. Shift counts of 18. or greater will give a zero result.

SRCI A,B timing = 5T OPCODE = 110

Shift right circular



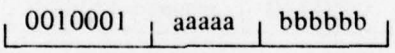

```
procedure SRCI(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := SRC(R[A],B);
      SETSHSTAT
end;
```

Register A is loaded with its contents shifted right circularly (end-around) by 'B' places. Shifter status bits are set by this operation.

****** SHIFT COUNTS OF 37. OR GREATER ARE TO BE AVOIDED!!!******

EXTR A,B timing = 5T OPCODE = 104

Extract (split register)



The 18-'B' right-most bits of the contents of register A are loaded into register A+1. Simultaneously, the 'B' high-order bits of the contents of register A are loaded into register A, right-justified with zero fill. This is accomplished by concatenating the contents of register A with a zero word on the left, then doing a circular left double shift of 'B' places on the resulting two-word quantity. The shifter output is placed in registers A and A+1. This instruction is useful for isolating the op-code of a target machine instruction. For example, if register 2 contained '$753245'_8$, the execution of a EXTR 2,6 would leave '$000075'_8$ in register 2, and '$324500'_8$ in register 3. Bit counts of greater than 36., in addition to making no sense, give unpredictable results.

ALUX A,B,C,D,E timing = 10T **OPCODE = 500**

ALU extension instruction

1010000	aaaaa	bbbbbb	cccccc	dddddd	eeeeee

ALU operation 'C' is performed on the contents of registers B (on the left) and E (on the right); the result is placed in register A, and ALU status bits are set. Also, the contents of CPU control register E are placed in register D. Setting either destination designator (A or D) to R.SCR will essentially bypass that function, except that ALU status bits will be set in any case. ALUX uses the CARRY bit (in FIST) as the carry-in to ALU operations, thus expediting multiple-precision arithmetic. If a decimal operation is specified by 'C,' the correction factor is loaded into R.ADR, otherwise the contents of this register remain unchanged. D may not designate an MPC register.

Control register codes

Value	Name	
0	PC.FLAGS	program check flags
1	PC.CSTAT	control store program check status
2	PC.CSADR	control store program check *status*
3	PC.MSTAT	main store status at least MSGO
4	PC.MSADR	main store address at least MSGO
5	PC.STATE	cpu state register

A complete explanation of the ALU operation code (parameter 'C') is included in the section on ALU and shifter operation. Some of the codes are:

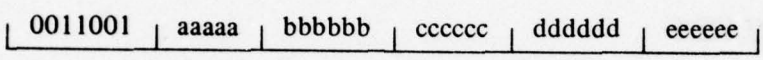
Octal Code	Operation
0	subtract borrow from left input
3	sign-extend borrow/carry
6	subtract right from left with borrow
11	add right to left with carry
17	add carry to left input
46	subtract with borrow decimal
51	add with carry decimal

For example, to do a double-precision one's complement addition, one must account for the carry between words, and the end-around carry, if any. Furthermore, to set the overflow status bit correctly, it is necessary to do the whole sum in one pass. Such an addition of registers B and B+1 into registers A and A+1 can be achieved by the sequence:

```
STAT  R.SCR,00                ;  clear carry initially
ALUX  R.SCR,A+1,11,R.SCR,B+1  ;  generate carry between words
ALUX  R.SCR,A,11,R.SCR,B      ;  generate carry-around
ALUX  A+1,A+1,11,R.SCR,B+1    ;  do low order add with carry
ALUX  A,A,11,R.SCR,B          ;  do high order add, set stat bits
```

SHIFT A,B,C,D,E timing = 11T OPCODE = 144

Shifter extension instruction



1. ALU operation 'C' is performed on the contents of register B-1 with those of register R.ADR. A carry-in of zero is used. Complete tables of both the ALU and shifter operation codes are included in the section on ALU and shifter operation.

2. Shift operation 'D,' of 'E' places is performed on a two-word operand: the result of the above operation on the left, and the contents of register B on the right. If a single-word shift is specified, the ALU result bypasses the shifter extension circuitry and is loaded directly into the high-order word of the destination pair. If a decimal ALU operation is specified by 'C,' the decimal correction factor is generated as the low-order word of the result, and no shifting takes place.

3. The result of the above operations is placed in registers A-1 and A; both shifter and ALU statuses are set. The double word shift codes are:

Octal Code	Shift Specified
2	circular left
3	circular right
6	logical left
7	logical right
12	arithmetic left
13	arithmetic right

FISTs status bits are set by SHIFT in the following manner:

SHB – Set by the sign bit of the shifter result (low-order word of SHIFT result); this is normally bit 17., but in 16-bit mode is bit 15. Note that 16-bit mode does not change the configuration of the shifter, but only the meaning of some of the status bits.

C – For all left shifts, carry is set as if by a double logical left shift of the same argument and count. In this case, it is normally set to the value of the last bit shifted out of the high-order word. But, when in 16-bit mode, it is set by bit 16. of the high-order word. For any right shift, carry is set as though by a double logical right shift of the same argument and count. For this shift, it is normally set by bit 0 of the low-order word of the result, except that a shift count of zero does not set it. When in 16-bit mode, a zero count, again, does not set carry, a count of one sets it to the last bit shifted out of the low-order word (ie., bit 0 of the argument), and for greater counts, it is set to the value of the 2nd last bit shifted out.

In any case, if the right control switch is set, carry will be set from bit 0 of the low-order word of the SHIFT argument, rather than one of the sources discussed above.

Carry bit may also be used as an argument for this operation. During double logical right shifts, it is shifted in as if to the left of the high-order word. For any shift, if the left *control switch is set, carry determines the low-order bit of the resultant.*

S – The sign bit of the high-order word of the result, normally bit 17., but bit 15. in 16-bit mode.

R – This bit is set if any of bits 16.-0 of the high-order word of the result is a '1.' If either of the control switches is set, this test includes all of the low-order word.

O – Set only by double arithmetic left shifts, it indicates that a bit was shifted into or through high-order word bit 17. different from the value of that bit of the argument.

SLB – Always set by bit 0 of the low-order word of the resultant.

Circular shift counts greater than the length of the shifted quantity should in all cases be avoided.

As an example of the use of the SHIFT instruction, consider the following routine for the multiplication of two's complement numbers. It calculated R(1) + R(2) * R.ADR and leaves the double-length result in registers 1 and 2, correct for any non-negative values of the contents of register 2. Register 10 is used to contain a counter:

```
LDI     10,21          ;  iteration count
SHIFT   2,2,37,0,0     ;  find out SLB
BOS     SLB,*+5        ;  to add or not to add?
SHIFT   2,2,37,13,1    ;  shift without add
BCT     10,*-3         ;  countdown
B       *+4            ;  exit
SHIFT   2,2,11,13,1    ;  add, then shift
BCT     10,*-7         ;  countdown
```

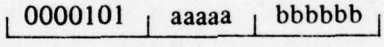
(continue)

Another example of SHIFT's use is this non-restoring division routine. It divides the quantity held in registers 1 and 2 by the contents of register 4, leaving the quotient in register 2, and a positive remainder in register 1. Register 3 is also used:

```
LDI     3,17.       ;  iteration count
SHIFT   2,2,37,2,1  ;  circularly shift around high-order bit
BOS     SLB,*+4     ;  do addition, or subtraction
SBR     1,4         ;  subtract divisor
BCT     3,*-4       ;  iterate
B       *+3         ;  go clean up
ADR     1,4         ;  add(!) divisor to high-order 18-bits of
                       dividend
BCT     3,*-7       ;  iterate
SHIFT   3,2,37,2,1  ;  get last result bit
BZS     SLB,*+2     ;  make sure we didn't subtract one
                       too many
ADR     1,4         ;  fix up remainder if we did
NTR     2,3         ;  correct quotient
   (continue)
```

TBO A,B timing = 10T(skip); 7T(no skip) OPCODE = 024

Test bit one

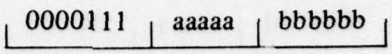



```
procedure TBO(A:FIVBIT,B:SIXBIT);
begin
      if AND(SRC(R[A],B),000001) ≠ 0 then
            R[MPC]  := R[MPC] + 2
      else R[MPC]  := R[MPC] + 1
end;
```

If the 'B'th bit (from the right, zero origin) of the contents of register A is one, then the next program word is skipped. Otherwise sequential processing continues. This instruction can thus be used for conditionally skipping single-word instructions. Bit counts of 18 or greater should be avoided, since the shifter hardware is invoked.

TBZ A,B timing = 10T(skip); 7T(no skip) OPCODE = 034

Test bit zero

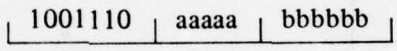



```
procedure TBZ(A:FIVBIT,B:SIXBIT);
begin
      if AND(SRC(R[A],B),000001) = 0 then
            R[MPC]  := R[MPC] + 2
      else R[MPC]  := R[MPC] + 1
end;
```

If the 'B'th bit of the contents of register A is a zero, then the next program word is skipped. Otherwise, sequential processing continues. This instruction is thus used for conditionally skipping single-word instructions. Bit counts of 18 or greater should be avoided.

CPR A,B timing = 5T OPCODE = 470

Compare registers




```
procedure CPR(A:FIVBIT,B:SIXBIT);
begin
      R[SCR]  := R[A] - R[B];
      SETALUSTAT
end;
```

The contents of register B are subtracted from those of register A, and ALU status bits are set according to the result. The outcome of this test may thus be tested by the BOS and BZS instructions. Various combinations of FIST ALU bit settings have the following meanings for two's complement arguments:

C	S	R	0	(Relation)
x	0	0	0*	R[A]=R[B]
x	1	x	0	R[A]<R[B]
1	0	x	1	R[A]<R[B]**
x	0	1	0	R[A]>R[B]
0	1	x	1	R[A]>R[B]**

(x = don't care)

*Overflow is trivially zero.

**These cases are for the maximum negative integer.

BOS A,B timing = 11T(branch); 8T(otherwise) OPCODE = 550

Branch on ones status

1011010	aaaaa	±bbbbb

```
procedure BOS(A:FIVBIT,B:SIGNSIX);
begin
      if AND(A,F[IST]) ≠ 0 then
            R[MPC]  := R[MPC] + B
      else R[MPC]  := R[MPC] +1
end;
```

The status bit in FIST are tested, and if any of the bits designated by the mask 'A' are ones, then a relative branch is taken to the word whose address is the sum of the contents of register and the six-bit signed operand 'B.' Otherwise, sequential processing continues. Note that when a branch instruction is the object of an EX,R.MPC contains the address of the executing instructions rather than that of the branch. Mnemonics known to the micro-assembler for masking are:

SLB	=	1
OVFLO	=	2
RESULT	=	4
SIGN	=	10_8
CARRY	=	20_8
SHB	=	40_8

However, since the 'B' operand has only a five-bit field, SHB can never be tested using this instruction.

BZS A,B timing = 11T(branch); 8T(otherwise) OPCODE = 560

Branch on zero status

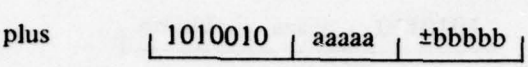
1011100	aaaaa	±bbbbb

```
procedure BZS(A:FIVBIT,B:SIGNSIX);
begin
      if AND(A,F[IST]) = 0 then
            R[MPC]  := R[MPC] + B
      else R[MPC]  := R[MPC] + 1
end;
```

The status bit in FIST are tested, and if all of the bits designated by the mask 'A' are zero, then a relative branch is taken to the word whose address is the sum of the contents of register R.MPC and the six-bit signed operand 'B.' Otherwise, sequential processing continues. Note that when a branch instruction is the object of an EX, R.MPC contains the address of the executing instruction, rather than that of the branch. Masking mnemonics are listed under the BOS instruction description. Because the A field has only five bits, SHB cannot be tested by this instruction.

BPL A,B timing = 11T(branch); 8T(otherwise) OPCODE = 510

Branch on plus

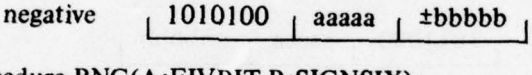



```
procedure BPL(A:FIVBIT,B:SIGNSIX);
begin
      if R[A] 0 then
            R[MPC]  := R[MPC] + B
      else R[MPC]  := R[MPC] + 1
end;
```

If the contents of register A are non-negative (sign bit zero), then a relative branch is taken to the word whose address is the sum of the contents of R.MPC and the six-bit signed operand 'B.' Otherwise, sequential processing continues. Note that when a branch instruction is the object of an EX or EXD, R.MPC contains the address of the executing instruction, rather than that of the branch.

BNG A,B timing = 11T(branch); 8T(otherwise) OPCODE = 520

Branch on negative

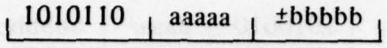



```
procedure BNG(A:FIVBIT,B:SIGNSIX);
begin
      if R[A]<0 then
            R[MPC]  := R[MPC] + B
      else R[MPC]  := R[MPC] + 1
end;
```

If the contents of register A are negative (sign bit one), then a relative branch is taken to the word whose address is the sum of the contents of register R.MPC and the six-bit signed operand 'B.' Otherwise, sequential processing continues. Note that when a branch instruction is the object of an EX, R.MPC contains the address of the executing instruction, rather than that of the branch.

BZR A,B timing = 11T(branch); 8T(otherwise) OPCODE = 530

Branch on zero

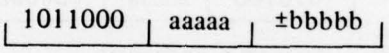



```
procedure BZR(A:FIVBIT,B:SIGNSIX);
begin
      if R[A]  := 0 then
            R[MPC]  := R[MPC] + B
      else R[MPC]  := R[MPC] + 1
end;
```

If the contents of register A are zero, then a relative branch is taken to the word whose address is the sum of the contents of register R.MPC and the six-bit signed operand 'B.' Otherwise, sequential processing continues. Note that when a branch instruction is the object of an EX, R.MPC contains the address of the executing instruction, rather than that of the branch. In 16-bit mode, the sign bit will be propagate to the left before the test is done, thus the two highest-order bits will not be correctly tested.

BNZ A,B timing = 11T(branch); 8T(otherwise) OPCODE = 540

Branch on non-zero

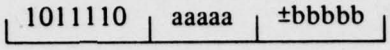



```
procedure BNZ(A:FIVBIT,B:SIGNSIX);
begin
      if R[A] ≠ 0 then
            R[MPC]  := R[MPC] + B
      else R[MPC]  := R[MPC] + 1
end;
```

If the contents of register A are non-zero, then a relative branch is taken to the location whose address is the sum of the contents of register R.MPC with the signed six-bit operand 'B.' Otherwise, sequential processing continues. Note that when a branch instruction is the object of an EX, R.MPC contains the address of the executing instruction, rather than that of the branch. In 16-bit mode, the sign bit will be propagated to the left before the test is done, thus the two highest-order bits will not be tested.

BCT A,B timing = 12T(branch); 9T(otherwise) OPCODE = 570

Branch on count

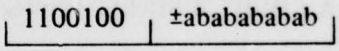



```
procedure BCT(A:FIVBIT,B:SIGNSIX);
begin
      R[A]  := R[A] - 1;
      if R[A] ≠ -1 then
            R[MPC]  := R[MPC] + B
      else beginp
            R[A]  := 0
            R[MPC]  := R[MPC] + 1
      end
end;
```

If the contents of register A upon the execution of this instruction are not zero, then they are decremented by one, and a relative branch to the location addressed by the sum of the contents of register R.MPC and the six-bit signed operand 'B.' If the contents of register A were zero, then sequential processing continues. Note that when a branch instruction is the object of an EX, R.MPC contains the address of the executing instruction, rather than that of the branch. The actual test of register A involves decrementing its contents to learn whether this operation generates a carry-out condition; if it does not, this register is presumed to have contained the value zero. FIST remains unaffected by the execution of this instruction.

B AB timing = 8T OPCODE = 620

Branch (unconditional)

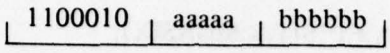



```
procedure B(AB:-1024..1023);
begin
      R[MPC]  := R[MPC] + AB
end;
```

Instruction processing continues at the location addressed by the sum of the contents of register R.MPC with the eleven-bit, signed, operand 'AB.' Note that when a branch instruction is the object of an EX, R.MPC contains the address of the executing instruction, rather than that of the branch.

BALR A,B timing = 9T OPCODE = 610

Branch and link, register

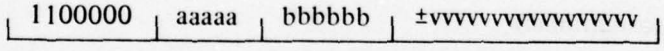



```
procedure BALR(A:FIVBIT,B:SIXBIT);
begin
      beginp
            R[IR]  := R[B],
            R[MPC] := R[MPC] + 1
      endp;
      beginp
            R[A]  := R[MPC],
            R[MPC] := R[IR]
      endp
end;
```

The address of the next sequential location is loaded, as a return link, into register A. Instruction processing continues at the word addressed by the contents of register B. Note that when a branch instruction is the object of an EX, R.MPC contains the address of the executing instruction rather than that of the branch. If a subroutine is entered, for example, by the execution of a "BALR R.RET,R.SUBR," it may be exited using "BALR R.SCR,R.RET."

BALN A,B,V timing = 10T OPCODE = 600

Branch and link, next word

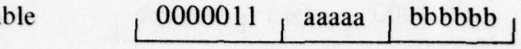



```
procedure BALN(A:FIVBIT,B:SIXBIT,V:integer);
begin
      R[IR]  := R[B] + V;
      R[MPC] := R[MPC] + 2;
      beginp
            R[A]  := R[MPC],
            R[MPC] := R[IR]
      endp
end;
```

The address of the following location is loaded, as a return link, into register A. Instruction processing continues at the word addressed by the sum of the contents of register B with the 16-bit, signed, operand 'V.' Note that when a branch instruction is the object of an EX or EXD, R.MPC contains the address of the executing instruction, rather than that of the first word of the branch instruction.

JUMP A,B timing = 15T OPCODE = 014

Jump via table

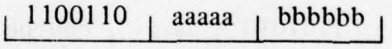



```
procedure JUMP(A:FIVBIT,B:SIXBIT);
begin
      R[MPC]  := R[MPC] + 1;
      R[IR]   := AND(R[A],B);
      R[MPC]  := CS[R[MPC] + R[IR]]
end;
```

Register R.MPC is loaded from a CS location addressed by the sum of the incremented MPC and an index formed by the AND of the contents of register A and the 6-bit operand 'B.' That is, program execution is continued at an address from a table immediately following the instruction. The table entry used is chosen using an index which is the AND of the contents of register A with the 6-bit operand 'B'; thus address arrays of one to 64. words may be used. Note that when this instruction is the object of an EX, R.MPC contains the address of the executing instruction, rather than that of the JUMP. In this case, the dispatch table should follow the EX. "JUMP R.IR,B" uses 'B' itself as the index.

EXN A,B timing = 5T OPCODE = 630

Execute next instruction

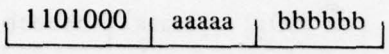



```
procedure EXN(A:FIVBIT,B:SIXBIT);
begin
      R[IR]  := CS[R[MPC] + 1];
      beginp
            R[IR]  := OR(R[IR], OR(SLL(R[A],6),R[B])),
            R[MPC]  := R[MPC] + 1
      endp
      {execute instruction as modified}
end;
```

After the next instruction is fetched into register R.IR, it is ORed with both the contents of register B, and those of register A shifted left six bits. This operation puts the low-order five bits of the contents of register A in the A-field of the next instruction, and the low-order six bits of the contents of register B in its B-field. The target instruction is then executed as modified; during its execution, register R.MPC points to the object instruction. EXN cannot alter the op-code of the object instruction, since this is kept in another register. However, it can change the high-order 7 bits of register R.IR, which are normally zero. No interrupts are permitted between the EXN and the target instruction's execution, thus the latter will not be refetched. It may be a two-word instruction, of which the second word remains unmodified.

EX A,B timing = 9T OPCODE = 640

Execute

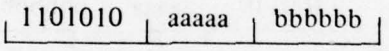



```
procedure EX(A:FIVBIT,B:SIXBIT);
begin
      beginp
            R[SCR]  := R[MPC] + 1 (for fetch-ahead),
            R[IR]   := CS[A[B]]
      endp;
      R[IR]  := R[IR] + R[A]
      {execute modified instruction}
end;
```

The control store word addressed by the contents of register B is fetched as an instruction (7 high-order bits to the op-code register, 11. low-order bits to register R.IR). Then the contents of register A are added into register R.IR, and the instruction so modified is executed. EX cannot change the op-code, but can modify the high-order seven bits of register R.IR, which normally contain zero. During execution of the object instruction, register R.MPC still points to EX, and fetch-ahead is continued (CS[R[MPC]+1] is being fetched), so that if a 2nd word is required by the object instruction, the word following the EX is used. Interrupts are not permitted between any execute and its target instruction, so that the latter will be executed as modified.

EXD A,B timing = 9T OPCODE = 650

Execute double-word instruction

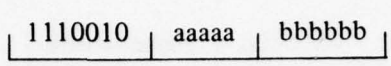



```
procedure EXD(A:FIVBIT,B:SIXBIT);
begin
      beginp
            R[SCR]  := R[B]+1, {for fetch-ahead}
            R[IR]   := CS[R[B]]
      endp;
      R[IR]  := R[IR] + R[A]
      {execute instruction as modified}
end;
```

The control store word addressed by the contents of register B is fetched as an instruction (operands into register R.IR). Then the contents of register A are added into register R.IR, and the instruction is executed as modified. The next program word fetched is CS[R[B]+1], permitting the execution of a double-word instruction. At the beginning of target instruction execution, register R.MPC addresses the EXD; a double-word instruction (or two single-word instructions) will increment this register by two, so that the word following the EXD will be skipped. (The only exception to this is the case where the addressed location contains a single-word instruction, after whose execution an interrupt occurs, causing CS[R[MPC]] to be refetched.) An interrupt cannot intervene between EXD and the (first) object instruction, which will be executed as modified. A programmer cannot change the target instruction's op-code by means of EXD, but can change the seven high-order bits in register R.IR, which are normally zero.

STAT A,B timing = 5T OPCODE = 710

Read and set status bits

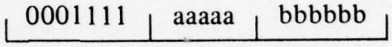



```
procedure STAT(A:FIVBIT,B:SIXBIT);
beginp
      R[A]  := F[IST],
      F[IST]  := B
endp;
```

Register A is loaded with the contents of the status register (FIST), right-justified, with zero fill. Register FIST is loaded with the six-bit operand 'B.'

SETALU A,B timing = 5T OPCODE = 074

Set ALU (16-bit) mode

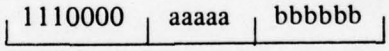



```
procedure SETALU(A:FIVBIT,B:SIXBIT);
beginp
      R[A]  := F[IDX],
      F[IDX]  := OR(AND(F[IDX],31), AND(B,32))
endp;
```

Register A is loaded with the contents of register FIDX, right-justified with zero fill. The high-order bit of FIDX (16-bit mode switch) is set or reset according to the high-order bit of the six-bit operand 'B.'

SIDX A,B supervisor timing = 5T OPCODE = 700

Set FIDX

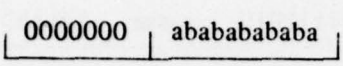



```
procedure SIDX(A:FIVBIT,B:SIXBIT);
beginp
      R[A]  := F[IDX],
      F[IDX]  := B
endp;
```

Register A is loaded with the contents of register FIDX, right-justified with zero fill. Register FIDX is loaded with the six-bit operand 'B.'

HALT AB supervisor timing ≥ 10T OPCODE = 000

Halt if switches set

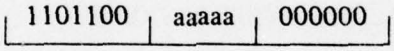



```
procedure HALT(AB:0. .2047);
begin
      while AND(AB,SW) ≠ 0 do
end;
```

If the switches masked by the six low-order bits of the argument 'AB' are not all off (down), then wait until they are, else proceed.

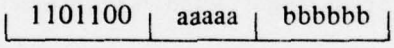
READS A supervisor timing = 7T OPCODE = 660

Read switches



AUX A,B supervisor timing = 7T OPCODE = 660

Perform auxiliary action



Execution of an AUX instruction:

–Loads register A with the six switches, right-justified with zero fill (this is the only action of the READS instruction)

–Places the contents of register R.ADR on the "index bus" and the contents of the CS location addressed by the contents of register R.ADR on the COD bus, as some auxiliary actions require.

–Performs the auxiliary action specified by the six-bit 'B' argument. Among the actions are:

		00	No Operation
. .PGMSTOP	=	40	suspend execution if PROGRAM STOP switch on, continue after START button pressed
. .PROPORT	=	41	access main store port register (†)
. .PROCPUD	=	42	disable secondary cpu control (†)
. .PROCPUE	=	43	enable secondary cpu control (†)
. .PROTDS	=	44	disable main storage protection (†)
. .PROTEN	=	45	enable main storage protection (†)
. .PROTECT	=	46	access MS protection memory (†)
. .PCKGEN	=	47	generate program check state
. .PCKCLR	=	50	clear program check flags
. .DS32	=	51	disable 32-bit shifter (+)
. .EN32	=	52	enable 32-bit shifter (+)
. .CLKSTOP	=	53	stop system clock
. .CLKRUN	=	54	load and start system clock
. .RMINDX	=	55	load RMI index (*)
. .RMINASK	=	56	load RMI mask (*)
. .RMISHIFT	=	57	load RMI shift amounts (*)
. .SEGCSA	=	60	load 'B' segment addresses (**)
. .SEGCSB	=	62	load 'B' segment addresses (**)
. .SEGEN	=	63	enable CS segmentation mode (**)
. .SLGDS	=	64	disable CS segmentation mode (**)
. .LDCSA	=	65	load control store address buffer
. .PCKENM	=	66	load program check enable mask
. .PCKMASK	=	67	load program check interrupt mask
. .PCKDS	=	70	disable program check interruption
. .PCKEN	=	71	enable program check interruption
. .MS1	=	72	select main store base 1
. .MS2	=	73	select main store base 2
. .MSABS	=	74	reference main store absolute mode (%)
. .MSREL	=	75	reference main store relative mode (%)
. .INTEN	=	76	enable external interrupts
. .INTDS	=	77	disable external interrupts

Codes marked with (†) reference the interleaved main store option.

Codes marked (+) reference the 32-bit shifter option.

Codes marked (*) reference the RMI option.

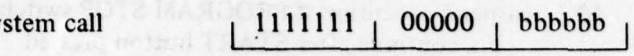
Codes marked (**) reference the CS segmentation option.

Codes marked (%) reference MS relocation and protection.

Unrecognized codes are ignored, as are those referring to uninstalled options.

SYSTEM B timing = 42T **OPCODE = 774**

System call




```
procedure SYSTEM(B:SIXBIT);
      function ENPACK(A,B,C:SIXBIT):integer;
      begin {pack 3 F-registers into an 18-bit word}
            ENPACK  := (A*64 + B)*64 + C
      end;
begin
      R[SCR]  := CS[TASK.BASE+64];
      beginp
            CS[R[SCR]]  := ENPACK(F[USR],F[ACT],F[IST]),
            R[SCR]  := R[SCR] + 1,
            F[USR]  := 0
      endp;
      beginp
            CS[R[SCR]]  := ENPACK(F[IDX],R[MPC],F[31] {is register
              "G11"})
            R[SCR]  := R[SCR]+1
      endp;
      beginp
            CS[R[SCR]]  := R[ADR],
            R[SCR]  := R[SCR] + 1
      endp;
      beginp
            CS[R[SCR]]  := R[MPC],
            R[SCR]  := R[SCR]+1
      endp;
      R[ADR]  := ENPACK(0,B,62);
      beginp
            CS[R[SCR]]  := R[IY],
            R[SCR]  := R[SCR]+1
      endp;
      CS[R[SCR]]  := R[IX];
      beginp
            R[IX]  := CS[TASK.BASE+62];
            R[IDX]  := 16 {or 20₈, signifying supervisor state}
      endp;
      R[MPC]  := CS[TASK.BASE+65];
      INTERN  := false;
      R[IY]  := TASK.BASE+64
end;
```

Execution of this instruction causes an interrupt; see the section on that topic for fuller details. Briefly, SYSTEM:

–saves a state array beginning at the CS location addressed by the contents of *CS[TASK.BASE+100$_8$]*,

–loads register R.IY with the value TASK.BASE+100$_8$, loads register R.IX with the contents of CS[TASK.BASE+76$_8$], and loads R.ADR with '000000 bbbbbb 111110'

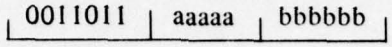
–loads register R.MPC with the contents of CS[TASK.BASE+101$_8$], enters supervisor state, disables interrupts, and continues instruction processing at the newly-addressed location.

The standard system-call codes are:

SYS.RCL	=	00
SYS.SIO	=	20
SYS.RECV	=	30
SYS.SEND	=	31
SYS.SENDW	=	32
SYS.WAIT	=	40
SYS.WAITL	=	41
SYS.SETIME	=	46
SYS.TIME	=	47
SYS.ITASK	=	50
SYS.STASK	=	51
SYS.KTASK	=	52
SYS.MSOFF	=	55
SYS.LOAD	=	60$_8$

SAVE A,B supervisor timing = 52T OPCODE = 154

Save E and G registers




```
procedure SAVE(A:FIVBIT,B:SIXBIT);
var EA, Q:SIXBIT;
    function ENPACK(A,B,C:SIXBIT):integer;
    begin {pack 3 F-registers into an 18-bit word}
        ENPACK := (A*64 + B)*64 + C
    end;
begin
    EA := B {"index register" 0 is E[8]};
    beginp
        Q := B,
        CS[R[A]] := E[EA],
        EA := EA+1,
        R[A] := R[A]+1
    endp;
    do
        beginp
            CS[R[A]] := E[EA],
            EA := EA+1,
            R[A] := R[A]+1,
            Q := Q-1
        endp
    until Q = 0;
    beginp
        CS[R[A]] := ENPACK(G[0],G[1],G[2]),
        R[A] := R[A]+1
    endp;
    beginp
        CS[R[A]] := ENPACK(G[3],G[4],G[5]),
        R[A] := R[A]+1
    endp;
    beginp
        CS[R[A]] := ENPACK(G[6],G[7],G[8]),
        R[A] := R[A]+1
    endp;
    CS[R[A]] := ENPACK(G[9],G[10],0)
end;
```

The contents of the 'B'+1 first "index registers" (EB.-E17.), and those of registers G0-G10. are stored in the 'B'+5 CS locations beginning at the one addressed by the contents of register A. This address is modified during instruction execution and points at the last word of the same array at instruction termination. G-register are saved 3 to an 18-bit CS word. The value of the B parameter must be greater than zero. The format of the save array is thus:

E8 = index 0		
E9 = index 1		
. . .		
G0	G1	G2
G3	G4	G5
G6	G7	G8
G9	G10	0

RESTORE A,B supervisor timing = 52T OPCODE = 164

Restore (un-SAVE) E and G registers

0011101	aaaaa	bbbbbb

Registers E[8] through E[8 + 'B'] are loaded from the save array in CS whose address is contained in register A. Registers G0-G10. are loaded from the four following CS locations. The contents of register A are modified to address the location following the last one loaded. This instruction is the complement of the SAVE operator, and expects a save array of the same format as is the one generated by the execution of that instruction.

EXIT A,B supervisor timing = 37T OPCODE = 670

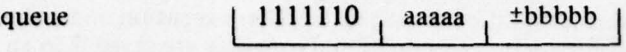
Exit from interrupt

1101110	abababababa

$0 \leqslant AB \leqslant 2047$ Control Store location (TASK.BASE+'AB') is expected to contain the address of a state-save array in CS of the format of the one generated by the occurrence of an interrupt (see the section on that subject for details). Registers FUSR, FACT, FIST, FIDX, G11, FMPC, R.ADR, R.IY, R.IX, and R.MPC are loaded from this array, interrupts are enabled, and instruction processing continues at the point where the interrupt occurred.

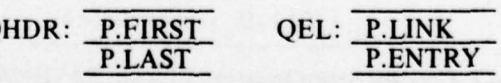
ENQ A,B timing ⩽ 29T **OPCODE = 770**

Enqueue




```
procedure ENQ(A:FIVBIT,BR:SIGNSIX);
type QEL = record
             LINK:↑QEL;
             ENTRY:↑DATA {or whatever}
             end;
      QHDR = record
             FIRST,
             LAST:↑QEL
             end;
var RY:↑QHDR; {R.IY is used as a pointer into CS}
    RX:↑QEL; {as is R.IX                        }
    RTMP:QEL; {and R.SCR, R.TMP                 }
begin
    RY  := RY+A;
    RTMP  := RY↑.LAST;
    if (RX↑.LINK ≠ nil) then begin {case 3, QEL already linked}
        RY  := COMPLEMENT (RY+1);
        B(BR)
    end
    else begin
        RY↑.LAST  := RX {link new element in}
        RX↑.LINK  := COMPLEMENT (RX) {denote this as last element}
        if (RTMP ≠ nil) then begin {wasn't list empty?}
            RTMP↑.LINK  := RX {complete link-in, case 2, non-empty
              list};
            RY  := RY+1;
            B(BR)
        end
        else begin {list was empty}
            RY↑.FIRST  := RX {complete link-in, case 1};
            RY  := RX↑.ENTRY
        end
    end
end {of ENQ};
```

Enqueue the queue element addressed by the contents of register R.IX on the linked list whose head is addressed by the sum of the operand 'A' and the contents of register R.IY. These entities are of the form:



where P.FIRST and P.LAST contain the addresses of the first and last elements on the queue, P.LINK contains the address of the next element on the queue, and P.ENTRY is assumed to contain the address of data, or a routine entry point, etc. There are three cases:

1. The queue was empty (P.LAST=0). The address of the queue element is placed in both P.FIRST and P.LAST; register R.IY is loaded with the contents of P.ENTRY, and sequential instruction processing continues.

2. The queue was not empty (P.LAST ≠ 0): The queue element is linked onto the end of the list, and a relative branch is taken to the CS location addressed by the sum of the contents of register R.MPC and the six-bit signed operand 'BR.' Note that if this instruction is the object of an EX, register R.MPC will contain the address of the execution instruction, rather than that of the ENQ.

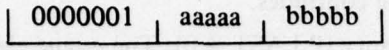
3. The queue element to be enqueued is already on a queue (P.LINK ≠ 0): Register R.IY is loaded with the complement of the sum of its contents and the five-bit unsigned 'A' parameter, plus one; and a relative branch is taken to the CS location addressed by the sum of the contents of the register R.MPC and the signed six-bit 'BR' operand. Recall the caveat concerning use of this instruction with EX.

In both of cases 1 and 2, P.LINK of the new QEL is set to the complement of its own address. Although the above program correctly gives the effect of this instruction, a closer representation of its actual workings is:

```
procedure ENQ(A:FIVBIT,B:SIGNSIX);
begin
      R[SCR]  := CS[R[IX]];
      beginp
            R[TMP]  := CS[R[IY]+A+1],
            R[IY]  := R[IY]+A+1
      endp;
      if R[SCR] ≠ 0 then begin
            R[IY]  := COMPLEMENT(R[IY]); {case 3}
            R[MPC]  := R[MPC]+B
            end
      else begin
            CS[R[IY]]  := R[IX];
            if R[TMP] ≠ 0 then begin
                  beginp
                        CS[R[TMP]]  := R[IX], {case 2}
                        R[TMP]  := COMPLEMENT(R[IX])
                  endp;
                  CS[R[IX]]  := R[TMP];
                  R[MPC]  := R[MPC]+B
                  end
            else begin {case 1}
                  beginp
                        CS[R[IY]-1]  := R[IX],
                        R[TMP]  := COMPLEMENT(R[IX])
                  endp;
                  CS[R[IX]]  := R[TMP];
                  R[IY]  := CS[R[IX]+1];
                  R[MPC]  := R[MPC]+1
            end
      end
end{ENQ};
```

DEQ A,B timing ⩽ 19T **OPCODE = 004**

Dequeue




```
procedure DEQ(A:FIVBIT,BR:SIGNSIX);
type  QEL = record
            LINK:↑QEL;
            ENTRY:↑DATA {or whatever}
            end;
      QHDR = record
            FIRST,
            LAST:↑QEL
            end;
var   RY:↑QHDR  {R.IY is used as a pointer into CS};
      RX:↑QEL   {     as is R.IX                   };
      RSCR:↑QEL {     and R.SCR,T.TMP              };
begin
      RY  := RY+A;
      RSCR  := RX↑.LINK;
      RX↑.LINK  := nil;
      RX  := RSCR;
      if (RSCR ⩾-1) then begin {case 1, not end of queue}
            RY↑.FIRST  := RX {IX pointed to its successor};
            RY  := RX↑.ENTRY {IY to point to successor's data};
            B(BR)
            end
      else {case 2, was last element of the queue}
            RY↑.LAST  := nil {queue now empty}
end {of DEQ}.
```

Dequeue the queue element (QEL) addressed by the contents of register R.IX from the linked list whose head (QHDR) is addressed by the sum of the contents of register R.IY and the five-bit 'B' operand. These entities are of the format described under ENQ. There are two cases:

1. The QEL is not at the end of the list,(i.e. it is not the only element on the list)—P.FIRST is loaded with the value contained in the QEL's P.LINK, and register R.IY is loaded with the contents of the successor element's P.ENTRY. A relative branch is taken to the location addressed by the sum of the contents of register R.MPC and the signed six-bit operand 'BR.' Recall the caveat regarding branches and EX.

2. The addressed QEL is at the end of the list (its P.LINK is negative)—P.LAST is set to zero, making the list empty.

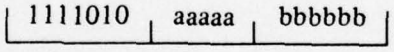
In both cases, the contents of register R.IY are modified to address the QHDR, and register R.IX is set to the addressed QEL's P.LINK, which, in turn, is set to zero.

Although the above program correctly gives the effect of this instruction, a closer representation of its actual working is:

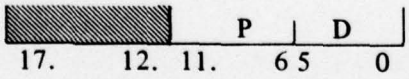
```
procedure DEQ(A:FIVBIT,B:SIGNSIX);
begin
      beginp
            R[SCR]  := CS[R[IX]],
            R[IY]   := R[IY]+A,
            R[TMP]  := 0
      endp;
      beginp
            R[IX]   := CS[R[IX]],
            CS[R[IX]]  := 0
      endp;
      if R[SCR]-1 then begin {case 1}
            beginp
                  CS[R[IY]]  := R[SCR],
                  R[IY]   := CS[R[IX]+1]
            endp;
            R[MPC]  := R[MPC]+B
            end
      else begin {case 2}
            CS[R[IY]]  := R[TMP];
            R[MPC]  := R[MPC]+1
            end
end; {of DEQ}
```

XIO A,B supervisor timing = 8T OPCODE = 750

Execute I/O command



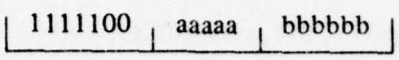
The contents of register A are loaded into external port 'P,' and I/O command 'B' is directed to channel 'P,' device 'D.' The P and D parameters are specified by the contents of register R.ADR in the following format:



This instruction is used to initiate I/O activity.

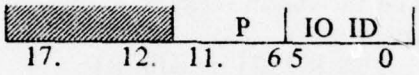
RIO A,B supervisor timing = 8T OPCODE = 760

Read I/O



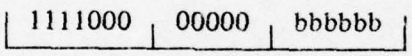
Register A is loaded with the contents of I/O port 'P.' Register R is loaded with a word containing the value on the I/O ID bus, and the value in bits 11-6 of R.ADR. An RIO pulse is directed to channel 'P.' The P parameters are specified by the contents of register R.ADR in the following format.

Register B receives a word of the form:



GENINT B supervisor timing = 6T OPCODE = 740

Generate interrupt



This instruction is used to generate, or to clear, an interrupt. The six-bit operand 'B' is of the form 'giiiii': g=0 specifies the generation of a level 'iiiii' interrupt; g=1 specifies the clearing of a level 'iiiii' interrupt.

APPENDIX A

MULTI NANO-PROGRAMMING CONVENTIONS

The MULTI instruction set is extendible by nano-programming. In addition to the many constraints placed on the nano-programmer by the hardware (e.g., not doing a READ NS during the first 2 T periods of a word which ALLOWS INTERRUPTS), there are a few conventions which must be followed for compatibility with the existing MULTI instructions.

At instruction invocation, both FAIR and FCOD point to R.IR (i.e., both contain 31.), FMPC points to R.MPC, and R.IR contains the low-order 11. bits of the instruction word (instruction parameters).

Except when a micro-instruction is entered by way of an EX or FXD, R.MPC will address the Control Store location of that instruction, and the COD bus will contain the contents of the succeeding CS word.

No assumptions should be made about the contents of the nano-program counter (NPC).

Interrupts should be permitted after each nano-routine.

Instructions operating on main store should set the CS ADDRESS field to indicate operation type in case of a p-check:

7 – I/O write
6 – I/O read
5 – MULTI write
4 – MULTI read
3,...,0 – user types or set as type 4 or 5.

PRECEDING PAGE BLANK-NOT FILMED

APPENDIX B

MULTI INSTRUCTION LISTS

PRECEDING PAGE BLANK-NOT FILMED

Ordered by Mnemonic

Mnemonic	OP-Code (3-3-1)	OP-Code (1-3-3)	Title
ADI	040	004	Add immediate
ADN	330	033	Add next
ADR	370	037	Add registers
ALUX	500	050	ALU extension
ANN	350	035	AND next
ANR	420	042	AND registers
AUX	660	066	Auxiliary action
B	620	062	Branch
BALN	600	060	Branch and link next
BALR	610	061	Branch and link register
BCT	570	057	Branch on count
BNG	520	052	Branch if negative
BNZ	540	054	Branch if not zero
BOS	550	055	Branch on one status
BPL	510	051	Branch if plus
BZR	530	053	Branch if zero
BZS	560	056	Branch on zero status
CPR	470	047	Compare
DEQ	004	100	Dequeue
ENQ	770	077	Enqueue
EX	640	064	Execute
EXD	650	065	Execute double
EXIT	670	067	Exit from interrupt
EXN	630	063	Execute next
EXTR	104	110	Extract
GENINT	740	074	Generate interrupt
HALT	000	000	Halt on switches
JUMP	014	101	Jump via table
LCI	030	003	Load complement immediate
LD	170	017	Load
LDD	140	014	Load direct
LDEI	720	072	Load from external registers
LDI	020	002	Load immediate
LDM	124	112	Load multiple
LDMS	240	024	Load from main store
LDMSA	310	031	Load from main store absolute
LDMSX	270	027	Load from main store indexed
LDN	320	032	Load next word
LDX	120	012	Load indexed
LDY	210	021	Load using IY register

PRECEDING PAGE BLANK-NOT FILMED

Ordered by Mnemonic (Continued)

Mnemonic	OP-Code (3-3-1)	OP-Code (1-3-3)	Title
MVR	450	045	Move register
NGR	460	046	Negate register
NTR	440	044	NOT register
ORN	340	034	OR next
ORR	410	041	OR registers
PULL	114	111	Pull registers
RAD	160	016	Replacement add
READS	660	066	Read switches
RESTORE	164	116	Restore saved registers
RIO	760	076	Read I/O
SAVE	154	115	Save registers
SBI	050	005	Subtract immediate
SBO	044	104	Set bit to one
SBR	400	040	Subtract registers
SBZ	054	105	Set bit to zero
SETALU	074	107	Set ALU mode
SHIFT	144	114	Shifter extension
SIDX	700	070	Set FIDX
SLLI	070	007	Shift left logical
SRAI	060	006	Shift right arithmetic
SRCI	110	011	Shift right circular
SRLI	100	010	Shift right logical
ST	200	020	Store
STAT	710	071	Read and set status
STD	150	015	Store direct
STEI	730	073	Store to External registers
STM	134	113	Store multiple
STMS	260	026	Store to main store
STMSX	300	030	Store to main store indexed
STNS	064	106	Store to nano-store
STX	130	013	Store indexed
STY	220	022	Store using IY register
SW	230	023	Swap
SWMS	250	025	Swap with main store
SYSTEM	774	177	System call
TBO	024	102	Test bit for one
TBZ	034	103	Test bit for zero
XIO	750	075	Executive I/O command
XON	360	036	Exclusive-OR next
XOR	430	043	Exclusive-OR registers

Order by OP-code

Mnemonic	OP-Code (3-3-1)	Mnemonic	OP-Code (3-3-1)
HALT	000	DEQ	004
pcheck word	010	JUMP	014
LDI	020	TBO	024
LCI	030	TBZ	034
ADI	040	SBO	044
SBI	050	SBZ	054
SRAI	060	STMS	064
SLLI	070	SETALU	074
SRLI	100	EXTR	104
SRCI	110	PULL	114
LDX	120	LDM	124
STX	130	STM	134
LDD	140	SHIFT	144
STD	150	SAVE	154
RAD	160	RESTORE	164
LD	170		
ST	200		
LDY	210	ALUX	500
STY	220	BPL	510
SW	230	BNG	520
LDMS	240	BZR	530
SWMS	250	BNZ	540
STMS	260	BOS	550
LDMSX	270	BZW	560
STMSX	300	BCT	570
LDMSA	310	BALN	600
LDN	320	BALR	610
ADN	330	B	620
ORN	340	EXN	630
ANN	350	EX	640
XON	360		
ADR	370		
SBR	400		
ORR	410		
ANR	420		
XOR	430		
NTR	440		
MVR	450		
NGR	460		
CPR	470		

Order by OP-code (Continued)

Mnemonic	OP-Code (3-3-1)	Mnemonic	OP-Code (3-3-1)
EXD	650	STEI	730
READS/AUX	660	GENINT	740
EXIT	670	XIO	750
SIDX	700	RIO	760
STAT	710	ENQ	770
LDEI	720	SYSTEM	774

APPENDIX C

MULTI INSTRUCTION REGISTER USAGE

Mnemonic	MPC register restrictions	Registers used (not R.IR nor R.SCR)	Conditions set
ADI			ALU
ADN			ALU
ADR			ALU
ALUX	yes	R.ADR, "CPU control"	ALU
ANN			ALU
ANR			ALU
AUX		R.ADR	
B			
BALN	yes		
BALR			
BCT			
BNG			
BNZ			
BOS			
BPL			
BZR			
CPR			ALU
DEQ		R.IX,R.IY,R.TMP	
ENQ		R.IX,R.IY,R.TMP	
EX			
EXD			
EXIT		R.ADR.R.IY,R.IX set	from save array
EXN			
EXTR			ALU
GENINT			
HALT			
JUMP			
LCI			
LD			
LDD	yes	R.ADR	
LDEI			
LDI			
LDM		R.ADR	
LDMS			
LDMSA		R.MX	
LDMSX		R.MX	
LDN			
LDX		R.IX	
LDY		R.IY	
MVR			
NGR			ALU
NTR			ALU

PRECEDING PAGE BLANK-NOT FILMED

Mnemonic	MPC register restrictions	Registers used (not R.IR nor R.SCR)	Conditions set
ORN			ALU
ORR			ALU
PULL		R.ADR	
RAD	yes		ALU
READS			
RESTORE			
RIO		R.ADR	
SAVE			
SBI			ALU
SBO			
SBR			ALU
SBZ			
SETALU			
SHIFT		R.ADR	ALU,SH
SIDX			
SLLI			SH
SRAI			SH
SRCI			SH
SRLI			SH
ST			
STAT			ALU,SH
STD	yes	R.ADR	
STEI			
STM		R.ADR	
STMS			
STMSX		R.MX	
STNS		R.ADR	
STX		R.IX	
STY		R.IY	
SW			
SWMS			
SYSTEM		R.ADR,R.IY,R.IX saved	
TBO			
TBZ			
XIO		R.ADR	
XON			ALU
XOR			ALU

APPENDIX D

QM-36 BIT ARCHITECTURE

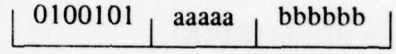
Nanodata Corporation provided a set of 36-bit instructions for MULTI. This is an extension of the MULTI instruction set and is therefore relegated to an Appendix. The QM-36 instruction set follows the conventions of MULTI and operates primarily on even/odd pairs of 18-bit words.

For the Pascal descriptions in this appendix, the following global declarations* are added to those on page

```
Type  DOUBLEINTEGER - 2^35 .. 2^35 - 1

Var   RD[0,2,4,6,8,10,12,14,16,18,20,22,23,24,26,28,30] of
      DOUBLEINTEGER;

Type  QUADINTEGER = - 2^71 .. 2^71 - 1

Var   RQ[0,4,8,12,16,20,24,28] of QUADINTEGER;
```

ANR2 A,B timing = 7T OPCODE = 224

And register 36 bit

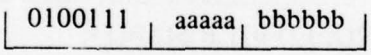



```
procedure ANR2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := AND(RD[A],RD[B];
      SETALUSTAT
end;
```

The data in local store registers (A,A+1) are replaced by the logical AND of the original contents of (A,A+1) and (B,B+1). The ALU status bits are set by this operation indicating sign and result of the 36-bit quantity produced.

ORR2 A,B timing = 7T OPCODE = 234

OR register 36 bit




```
procedure ORR2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := OR(RD[A], RD[B];
      SETALUSTAT
end;
```

Local Store Register (A,A+1) is replaced with the logical OR of Local Store Registers (A,A+1) and (B,B+1). ALU status is set indicating the sign and result of the 36-bit quantity produced.

*Note that the definitions of Doubleinteger and Quadinteger are not standard Pascal but are defined this way for clarity.

PRECEDING PAGE BLANK-NOT FILMED

XOR2 A,B timing = 7T OPCODE = 244

Exclusive–OR 36 bit | 0101001 | aaaaa | bbbbbb |

```
procedure XOR2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := XO(RD[A],RD[B]);
      SETALUSTAT
end;
```

Local store register (A,A+1) is replaced with the exclusive–or of local store registers (A,A+1) and (B,B+1). ALU status is set indicating the sign and result of the 36-bit quantity produced.

ADR2 A,B timing = 7T OPCODE = 174

Add register 36 bit | 0011111 | aaaaa | bbbbbb |

```
procedure ADR2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := RD[A]+RD[B];
      SETALUSTAT
end;
```

The data in local store registers (A,A+1) is replaced by the sum of the original content of local store registers (A,A+1) and (B,B+1). ALU status is set indicating the sign and result, carry out and overflow conditions of the 36-bit quantity produced.

SBR2 A,B timing = 7T OPCODE = 204

Subtract registers 36 bit

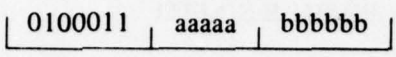
| 0100001 | aaaaa | bbbbbb |

```
procedure SBR2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := RD[A]-RD[B];
      SETALUSTAT
end;
```

The data in local store registers (A,A+1) is replaced by the original contents of local store registers (A,A+1) minus the contents of local store, registers (B,B+1). The ALU status is set indicating the sign and result, carry result and overflow conditions of the 36-bit quantity produced.

NGR2 A,B timing = 8T OPCODE = 214

Negate register 36 bit

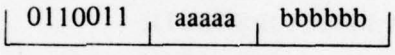



```
procedure NGR2(A:FIVBIT,B:SIXBIT);
begin
      RD[A+1]  := 0-RD[A+1]+1;
      SETALUSTAT
end;
```

The data in Local Store registers (A,A+1) is replaced by the two's complement (negative equivalent) value of registers (B,B+1). ALU status is set indicating sign and result, carry result and overflow conditions of the 36-bit quantity produced.

SLL2 A,B timing = 4T OPCODE = 314

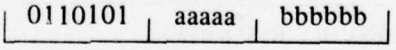
Shift left logical 36 bit




```
procedure SLL2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := SLL(RD[A],B);
      SETSHSTAT
end;
```

The data in local store registers (A,A+1) is shifted left logically, by B-bit positions. Bits are shifted to the left beyond bit position 35 are lost. Zero bits are entered from the right end of the register. Shifter status is set indicating the resulting high and low bit values from positions 35 and 0, respectively.

SRL2 A,B timing = 4T OPCODE = 324

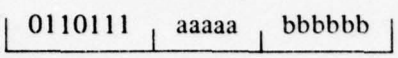
Shift right logical 36 bit 


```
procedure SRL2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := SRL(R[A],B);
      SETSHSTAT
end;
```

The data in local store registers (A,A+1) is shifted right logically, by B bit positions. Bits shifted to the right beyond bit position 0 are lost. Zero bits are shifted in at the left end. Shifter status is set indicating the resulting high- and low-bit values from positions 35 and 0, respectively.

SRA2 A,B timing = 4T OPCODE = 334

Shift right arithmetic 36 bit

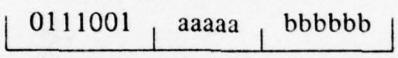



```
procedure SRA2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := SRA(RD[A],B);
      SETSHSTAT
end;
```

The data in local store registers (A,A+1) is shifted right arithmetically by B-bit positions. Bits shifted right beyond bit 0 are lost. The original value of bit 35 is propagated right. Shifter status is set indicating the resulting bit 35 and bit 0 values.

SRC2 A,B timing = 4T OPCODE = 344

Shift right circular 36 bit

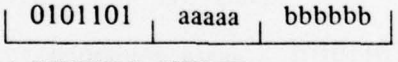



```
procedure SRC2(A:FIVBIT,B:SIXBIT);
begin
      RD[A]  := SRC(RD[A],B);
      SETSHSTAT
end;
```

The data in Local Store registers (A,A+1) is shifted right 36 bit circular by 13-bit positions. Shifter status is set indicating the resulting bit 35 and bit 0 values.

LD2 A,B timing = 8T OPCODE = 264

Load Control Store 36 bit

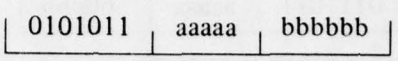



```
procedure LD2(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := CS[R[B]];
      R[A+1]  := CS[R[B] + 1]
end;
```

The Control Store word pair addressed by Local Store register B is read and placed into Local Store registers (A,A+1). No status is set.

ST2 A,B timing = 8T OPCODE = 254

Store Control Store 36 bit

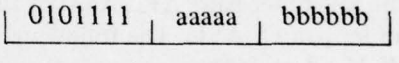



```
procedure ST2(A:FIVBIT,B:SIXBIT);
begin
      CS[R[B]]  := R[A];
      CS[R[B] +1]  := R[A+1]
end;
```

The contents of Local Store registers (A,A+1) are placed into the Control Store word pair addressed by Local Store register B. No status is set.

LDMS2 A,B timing = 22T OPCODE = 274

Load Main Store 36 bit

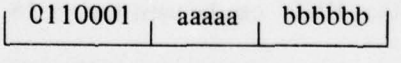



```
procedure LDMS2(A:FIVBIT,B:SIXBIT);
begin
      R[A]  := MS[MSADDR(RD[B])];
      R[A+1]  := MS[MSADDR(RD[B]+1)]
end;
```

The Main Store word pair addressed by Local Store registers (B,B+1) is read and placed into Local Store register (A,A+1). No status is set.

STMS2 A,B timing = 15T OPCODE = 304

Store Main Storage 36 bit




```
procedure STMS2(A:FIVBIT,B:SIXBIT);
begin
      MS[MSADDR[RD[B]]  := R[A];
      MS[MSADDR[RD[B] +1)]  := R[A+1]
end;
```

The contents of Local Store registers (A,A+1) are placed into the Main Store word pair addressed by Local Store register (B,B+1). No status is set.

MULC A,B timing = 13T OPCODE = 354
9T if no addition

Multiplication cycle | 0111011 | aaaaa | bbbbbb |

```
procedure MULC(A:FIVBIT,B:SIXBIT);
begin
      if STATUS (SLB)=1;
      then
            RD[A]  := RD[A]+RD[B];
      end;
      RQ[A]  := SRL(RQ[A],1);
      SETSHSTAT
end;
```

This instruction provides a building block for an add and shift multiply block for an add and shift multiply algorithm. The A parameter points to a four register "accumulator" to allow a 72-bit product to be formed. The shifter low bit (SLB) is used to test the add. Note that the user must set SLB before initiating the MULC the first time through the loop. The multiplicand is placed in the low order pair (A+2, A+3) of the four register group. The product is formed in the high order pair (A,A+1). The following example * shows the use of the MULC instruction in a 36 X 36 bit multiply.

```
PROD        =       12.             ; MULTIPLICAND-PRODUCT, 4 REG.
                                        SET
MUL         =       22.             ; MULTIPLIER, 2 REG. SET

            SBR2    PROD,PROD       ; CLEAR THE HIGH ORDER
                                      REGISTERS.
            LDI     CYCLES, 36.-1   ;   SET UP ITERATION COUNTER.
            SRCI    PROD+3,0        ;    NULL 18 BIT SHIFT SETS UP LOW
                                    ;     ORDER BIT STATUS (SLB).
MULTIPLY:   MULC    PROD, MUL       ; EXECUTE ONE MULTIPLY CYCLE,
            BCT     CYCLES,MULTIPLY ;  AND LOOP 36 TIMES.
```

The reader should note that MULC can be used for 16, 18, or other length multiplies.

*The examples were taken from the QM-36 documentation supplied by Nanodata.

DIVC A,B timing = 17T OPCODE = 364
15T if no subtract

Division Cycle | 0111101 | aaaaa | bbbbbb |

```
procedure DIVC(A:FIVBIT,B:SIXBIT);
begin
      RQ[A]  := SLL[RQ[A],1];
Status (Carry)  : = SH;
      R[SCR] =R[A];
      R[TMP]  := R[A+1];
      RD[A]  := RD[SCR]-RD[B];
      if STATUS (CARRY)=1  .OR.  COH=1
      then
      begin
      R[A+3]  := R[A+3]+1;
      R[A]  := R[SCR];
      R[A+1]  := R[TMP]
      end;
end;
end;
```

"DIVC" produces one division cycle per iteration. Each cycle consists of a four register shift, left, logically, by one-bit position, on all of the data in (A,A+1, A+2, A+3). This is followed by a conditional subtraction, of the 36-bit divisor (B,B+1) from the high order 37-bit dividend in (CARRY) + (A,A+1), if the absolute value of the divisor is less than the absolute value of the dividend. When this subtraction does take place a new quotient bit value of one is added to the rightmost position of (A+3), and the subtracted difference replaces (A,A+1).

Before beginning a division operation the user must place the 72-bit dividend into a four register array, based at a four register boundary, such as 0, 4, 8, 12, etc. This will be identified through the "A" parameter. The "B" parameter points to the even element of a 36-bit register pair, which will contain the divisor. It is the user's responsibility to test for, and avoid, division overflow, which will produce completely erroneous results without indication. This is accomplished by verifying that the contents of (A,A+1) are less than (B,B+1) prior to beginning the division. At the end of a full 36-bit division the quotient will be found in (A+2, A+3), with the remainder in (A,A+1). The contents of (B,B+1) will remain unchanged. The following is an example of a full precision divide.

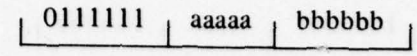
```
QUOT    =       20.                     ; DIVIDENT-QUOTIENT, 4 REG.
                                          ARRAY.
DIV     =       8.                      ; DIVISOR, 2 REG. ARRAY.
*       DIVIDEND IS IN QUOT, QUOT+1, QUOT+2, QUOT+3.
*       DIVISOR IS IN DIV, DIV+1.
*       VERIFY NO DIVIDE OVERFLOW, BEFORE DIVIDING.
        MVR     TEMP, QUOT              ; COPY THE HIGH ORDER
                                        ;  DIVIDEND
        MVR     TEMP+1, QUOT+1          ;   ELEMENTS AND THEN
        SBR2    TEMP, DIV               ;    SUBTRACT THE DIVISOR.
        BOS     CARRY, DIV. ERROR       ;     ERROR IF DIVISOR IS
                                        ;      SMALLER.
        LDI     CYCLES, 36.1            ;       SET ITERATION COUNTER.

DIVIDE: DIVC    QUOT, DIV               ; EXECUTE ONE DIVIDE
        BCT     CYCLES, DIVIDE          ;  CYCLE AND LOOP 36 TIMES.
```

When less than 36-bit precision is needed, a faster division may be realized through a smaller number of iterations. For example, a 16-bit division of a 32-bit dividend may be performed using only 16 "DIVC" iterations. The only requirement for using fewer iterations is the proper alignment of data before beginning the division. The absolute dividend must be aligned so that its unit bits is placed 16 positions to the right of the center boundary of the four register dividend array. The divisor itself is maintained right justified. Following this 16-bit division, the quotient will be found, right justified, in (A+3), and the remainder, right justified, in (A+1). All unused registers must be cleared to zero.

```
QUOT    =       12.                     ; DIVIDEND-QUOTIENT, 4 REG.
                                          ARRAY.
DIV     =       4.                      ; DIVISOR, 2 REG. ARRAY.

        SBR2    QUOT, QUOT              ; CLEAR UNUSED DIVIDEND
                                          REGISTERS
        LD2     QUOT+2,DIVIDENDS        ;   AND FETCH ACTUAL DIVIDEND.
        LDD     DIV+1,NULL,DIVISOR      ;    FETCH THE DIVISOR AND
        LD1     DIV,0                   ;     CLEAR THE HIGH ORDER
                                        ;      ELEMENT.
        SHIFT   QUOT+2,QUOT+3,PASS,LEFT+DOUBLE+LOGICAL,2
                                        ;       CENTER THE DIVIDEND
                                        ;        VALUE WITHIN 4
                                        ;         REGISTER ARRAY AND
        LDI     QUOT+3,0                ;          CLEAR QUOTIENT
                                        ;           WORD.
        LDI     CYCLES,16.-1            ;            SET ITERATION
                                        ;             COUNTER.
DIVID:  DIVC    QUOT,DIV                ; EXECUTE A DIVIDE CYCLE AND
        BCT     CYCLES,DIVID            ;  LOOP 16 TIMES.
```

LBO A,B timing = 6T to 20T OPCODE = 374

Locate one bit (36 bit)

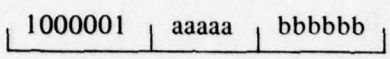



```
procedure LBO(A:FIVBIT,B:SIXBIT);
begin
      if R[A].OR.R[A+1]
            then
                  R[B]  := 0
                  while SRL(RD[A],R[B]).AND.1 < 70 do
                        R[B]  := R[B1]+1
                  end
                  else
                        R[B]  := -2
      end
end
```

Locate first one bit in R(A,A+1), and indicate in R(B). Register pair R(A,A+1) is scanned from left to right, starting at bit position 35, for the first occurrence of a bit value 1. The bit position number of the significant bit located is placed into R(B), through the index-ALU. Thus the leftmost bit position of R(A,A+1) will be denoted as the value 0, and the rightmost bit position is denoted as the value 35. Should there be no 1 bits within the register pair, a value of minus 2 will be returned in R(B). The 'A' parameter must point to the even element of the 36-bit register pair. 'B' may point to any 18-bit register accessible to the index-ALU (excludes registers 24 through 27, decimal.) ALU and shifter status are unaffected.

LBZ A,B timing = 6T to 20T OPCODE = 404

Locate zero bit (36 bit)




```
procedure LBZ(A:FIVBIT,B:SIXBIT);
begin
      if not (R[A].AND.R[A+1])
            then
                  R[B]  := 0
            while SRL(RD[A],R[B]).AND.|<>| Do
                  R[B]  := R[B]+1
            end
            else
                  R[B]  := -2
      end
end
```

Locate first zero bit in R(A,A+1), and indicate in R(B). Register Pair R(A,A+1) is scanned from left to right, starting at bit position 35, for the first occurrence of a bit value 0. The bit position number of the significant bit located is placed into R(B), through the index-ALU. Thus the leftmost bit position of R(A,A+1) will be denoted as the value 0, and the rightmost bit position is denoted as the value 35. Should there be no 0 bits within the register pair a value of minus 2 will be returned in R(B). The 'A' parameter must point to the even element of the 36-bit register pair. 'B' may point to any 18-bit register accessible to the index-ALU (excludes registers 24 through 27, decimal.) ALU and shifter status are unaffected.

APPENDIX E

UCSD EXTENSIONS

This extension was written for NSWC by University of California at San Diego (UCSD).* These have not yet been accepted as part of the standard Micro-instruction set of the QM-1. However they are included for completeness. At some time in the future they may be used to enhance the intermediate language machine called EASY which is operational on the QM-1 computer.

SWAP A,B

Swap register contents

```
procedure SWAP(A:FIVBIT,B:SIXBIT);
    beginp
    R[A]  := R[B],
    R[B]  := R[A]
    endp;
```

The contents of Local Store register A are interchanged with those of register B.

SE A,B

Sign-extend register

```
procedure SE(A:FIVBIT,B:SIXBIT);
    begin
    R[A]  := SRA(R[B],18)
    end;
```

The sign of the contents of register B are propagated throughout register A. The argument of the shifting operation (contents of register B) is passed tnrough the ALU first, so that if 16-bit mode is set, bit 15. will be the one propagated.

For the purposes of 5 new stack-oriented instructions, a Local Store register has been designated as the stack pointer:

```
const SP = 22;
```

*Under NSWC contract No. N60921-77-M-B785.

PRECEDING PAGE BLANK-NOT FILMED

CALL AB

Subroutine call

```
procedure CALL(AB:1024. .1023);
begin
      beginp
            R[SCR]  := R[MPC]+1,
            R[MPC]  := R[MPC]+AB
      endp;
      beginp
            CS[R[SP]]  := R[SCR],
            R[SP]  := R[SP]+1
      endp
end;
```

The address of the following location is pushed, as a return link, onto the stack in Control Store. The stack pointer is incremented. Instruction processing continues at the location addressed by the sub of the contents of register R.MPC with the eleven-bit, signed, operand 'AB.' Recall the caveat regarding the use of branch instructions and EX.

CALLN A,B,V

Call, next word

```
procedure CALLN(A:FIVBIT,B:SIXBIT,V:integer);
begin
      beginp
            R[SCR]  := R[MPC]+2,
            R[MPC]  := V,
            R[SYS]  := R[IR]
      endp;
      beginp
            CS[R[SP]]  := R[SCR],
            R[SP]  := R[SP]+1
      endp
end;
```

The address of the following location is pushed, as a return link, onto the stack in Control Store. The stack pointer is incremented. Register R.SYS is loaded from register R.IR, which contains the A and B parameters. Instruction processing continues at the location addressed by the 'V' operand. Recall the caveat regarding the use of branch instructions with EX.

PUSH A,B

Push registers onto stack

```
procedure PUSH(A:FIVBIT,B:SIXBIT);
var G,C:SIXBIT;
begin
      beginp
            G  := A,
            C  := B
      endp;
      while G≠D do
            beginp
                  CS[R[SP]]  := R[C],
                  R[SP]  := R[SP]+1,
                  C  .= C+1,
                  G  := G-1
            endp
      end;
```

The contents of the registers beginning with register B are stored on the Control Store stack. The stack pointer is incremented after each register is stored.

POP A,B

Pop registers from stack

```
procedure POP(A:FIVBIT,B:SIXBIT);
var G,C:SIXBIT;
begin
      beginp
            G  := A,
            C  := B
      endp;
      R[SCR]  := R[SP] -A;
      R[SP]  := R[SCR];
      while G≠O do
            beginp
                  R[C]  := CS[R[SCR]],
                  R[SCR]  := R[SCR]+1,
                  C  := C+1,
                  G  := G-1
            endp
      end;
```

The 'A' registers beginning with register B are loaded from the 'A' words on top of the Control Store stack. The contents of ≠ the stack-pointer register are diminished by 'A,' and register R.SCR is used as a pointer during instruction execution.

RETURN B

Return from subroutine via stack

```
procedure RETURN(B:SIXBIT);
beginp
      R[MPC]  := CS[R[SP]-1],
      R[SP]   := R[SP]-1
endp;
```

The program counter, R.MPC, is loaded from the top of the stack, and the stack pointer is decremented. The instruction argument is ignored. This instruction may be used to return from a subroutine invoked by the CALL or CALLN instructions.

MOVMC A,B

Move between Main and Control Stores

```
procedure MOVMC(A:FIVBIT,B:SIXBIT);
begin
      while R[A]≠O do
            begin
                  beginp
                        R[SCR]  := MS[MSADDR(R[B])],
                        R[A]    := R[A]-1,
                        R[B]    := R[B]+1
                  endp;
                  beginp
                        CS[R[B+1]]  := R[SCR],
                        R[B+1]      := R[B+1]+1
                  endp
            end
      end;
```

The contents of some Main Store words are transferred to Control Store locations. The number of words to be transferred is initially contained in register A, the initial Main Store source location address is in register B, and the Control Store beginning address is in register B+1. During each transfer operation, the count is decremented, and the two addresses augmented. Because execution of this instruction could take a very long time, it is interruptable at the end of the transfer loop, and will automatically be restarted upon the completion of interrupt processing. Registers B & B+1 may not be MPC registers.

LD8, A,B

Load Byte

An eight-bit byte is loaded from Control Store into register A, where it is right-justified with zero fill. Bytes in Control Store are considered to be packed two to a word, an even-numbered byte occupying bits 8-15., and its successor bits 0-7. The contents of register B are used as a byte offset from a word address taken from register E8 ("index register 0"). Thus the word chosen has as its address the sum: E[B]+(R[B] div 2).

MULR A,B

Multiply registers

This instruction achieves signed, two's complement multiplication on the contents of registers A and B. The product is left in registers A-1 and A (high to low). The method of shifting and adding is used, except when the multiplier (from register A) is negative; in this case, it is negated before the product is taken, and subtraction is done in lieu of addition. This instruction will not give a correct result when one of the arguments is the largest negative number, represented by 400000_8.

BIBLIOGRAPHY

Nanodata Corporation, *Task Control Program*, Williamsville, New York, 1976.

Nanodata Corporation, *MULTI Micro-Machine Description*, Williamsville, New York, 1976.

Nanodata Corporation, *Micro, QM Micro-Assembler Reference Manual*, Williamsville, New York, 1976.

Nanodata Corporation, *QM-36 Documentation*, Williamsville, New York.

DISTRIBUTION

Commander
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92152
ATTN: Code 5200
Russ Eyers

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22202
ATTN: William Carlson

Commander
U. S. Naval Electronic Systems Command
Washington, DC 20360
ATTN: John Machado (Code 330)

Computer Science Department
VPI and State University
Blacksburg, VA 24060
ATTN: Dr. Richard Nance
Dr. Thomas Wesselkamper

Commanding Officer
U. S. Army, Harry Diamond Laboratories
2800 Powder Mill Road
Adelphi, MD 20783
ATTN: Branch 520
Rick Johnson

Nanodata Corporation
2457 Wehrle Drive
Williamsville, NY 14221
ATTN: M. Brenner
W. D. Robertson
Robert C. Boe

Defense and Space Systems Group of TRW Inc.
One Space Park
Redondo Beach, CA 90278
ATTN: Barry Press
David Bixler
Herb Wangenheim

Martin-Marietta
P. O. Box 197
Denver, CO 80201
ATTN: Skip Scown
B. A. Claussen

McDonnell Douglas
5301 Bolsa Avenue
Huntington Beach, CA 92647
ATTN: Harris Dalrymple

RADC/ISCA
Griffiss Air Force Base
Rome, NY 13441
ATTN: Armand Vito

Computer Science Division
Department of Applied Physics and Information Science
University of California at San Diego
LaJolla, CA 92093
ATTN: Dr. W. A. Burkhard
Dr. Terry Miller

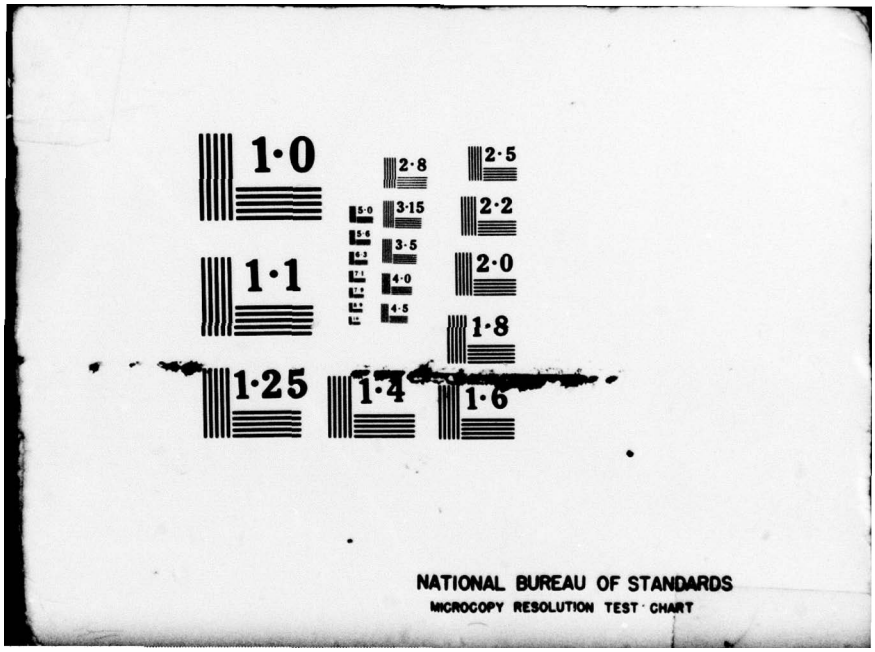
Department of Computer Science
University of Alberta
Edmonton Alberta
Canada T6G 2E1
ATTN: Dr. John Tartar
Steven Sutphen
John Demco

Hughes Aircraft Company
Bldg. 12 M/S V107
Culver City, CA 90230
ATTN: Jon Humphry

AD-A060 639 CALIFORNIA UNIV SAN DIEGO LA JOLLA COMPUTER SCIENCE DIV F/G 9/2
QM-1 MULTI MICRO-PROGRAMMER GUIDE.(U)
SEP 78 W A BURKARD , R D TUCK, R L HARTUNG N60921-77-M-B785
UNCLASSIFIED NSWC/DL-TR-3834 NL

2 OF 2
ADA
060639

END
DATE
FILMED
1 -79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

NASA
Langley Research Center
Flight Inst. Div. MS477
Langley, VA 23365
ATTN: J. Earl Migneault

Computer Science Department
University of Maryland
College Park, MD 20742
ATTN: Dr. Yaohan Chu

Director
Naval Research Laboratory
Washington, DC 20390
ATTN: Y. S. Wu
William R. Smith

USC/Information Sciences Institute
4676 Admiralty Way
Marine Del Ray, CA 90291
ATTN: Lou Gallenson
Joel Goldberg

DIT-MCO International
5612 Brighton Terrace
Kansas City, MO 64130
ATTN: Joel Herbsman

Director
Defense Mapping Agency Headquarters
U. S. Naval Observatory
Building 56
Washington, DC 20305
ATTN: Annette Krygiel

Defense Documentation Center
Cameron Station
Alexandria, VA 22314 (2)

COMPRO
Route 1 Box 690
King George, VA 22485
ATTN: A. Ammerman

University of Southwestern Louisiana
USL Box 4-4330
Lafayette, LA 70504
ATTN: Paul A. Boudreaux

Chairman Sigmicro
257 South Adelaide
Highland Park, NJ 08904
ATTN: Stanley Habib

IBM
System Architecture Department
101B559
Bodlehill Road
Owego, NY 13827
ATTN: J. Dorocak

Local:

E41
K54 (R. Pollock, W. McCoy)
K60
K61
K61 (Stemple)
K70
K71
K74 (40)
X210 (2)