MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

545 Technology Square, Cambridge, Massachusetts 02139

Computation Structures Group Memo 192-1

Packet Communication Microprocessor Programming Manual

William B. Ackerman

The preparation of this document was supported in part by the Lawrence Livermore Laboratory of the University of California under contract no. 8545403, and in part by the Department of Energy under contract no. DE-AC02-79ER10473.

20 October 1980

1. Introduction

The MP (microprocessor) is a high speed programmable 8 bit microcomputer designed for simulation of devices that communicate by byte-serial packet communication. It is designed to be "downloaded" from, and supervised by, a host computer such as a PDP-11 or LISP machine.

The MP logically consists of the following items, as shown in Figure 1:

Program memory - 4K (4096) 40 bit instructions containing the program. This is loaded by the host and does not change while the program runs.

Data memory - up to 64K (65536) 8 bit words that may be read or written by the processor. It is addressed by a 16 bit address register. This memory may also be read or written by the host while the MP is idle.

Address register - a 16 bit register giving the address of the word in data memory that the processor can read or write. The two halves of this register may be written by the processor.

Scratchpad memory - 16 words of 8 bits each, all equivalent, used for most processor operations.

"Q" register - one special 8 bit register used for shifting and special arithmetic operations.

Condition code - 4 bits, containing information about the instruction just completed. This information is useful for conditional jumps or other operations. The meaning of these four bits is explained below.

Sign compare bit - a flip-flop used during divide and normalize instructions.

Program counter - 12 bits, giving the address of the next instruction. The host can set this to zero prior to program execution.

Call stack - 5 words of 12 bits each, with a stack pointer. This is used for saving the program counter during subroutine calls, and for storage of restart addresses in iterations. Subroutine calls may thus be nested to a depth of five.

Address/count register - 12 bits, used for counting in loops, and for temporary storage of a jump address.

Offset resgister - 8 bits, used for modifying the address of a control instruction with the result of a computation.

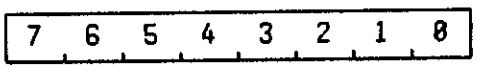
External input and output ports - these are used for transmission or reception of data bytes to or from a router, another processing element, or the host computer. There are 8 data bits plus one "last" status bit that is intended to mark the end of a packet. The processor can read the status of these ports to determine whether an input byte is available or an output byte may be sent.

Port select register - this register determines which of the two ports is to be used by IO operations. It may be written by the processor.

1.1 Number format

The MP normally uses 8 bit two's complement arithmetic. Hence the numbers that can be represented lie in the range [-128, 127] inclusive. The addition and subtraction mechanism can also deal with numbers that are considered to be unsigned, if the sign bit and condition code are interpreted differently. If this is done, the allowable range is [0, 255] inclusive. The conditional instructions can deal with either interpretation. The "HI" and "LO" conditions refer to unsigned numbers, that is, they consider 200_8 to be greater than 177_8. The "GT" and "LT" conditions refer to signed numbers. They consider 200_8 to be negative and hence smaller than 177_8.

In this document, the bits of a word are numbered as follows:



1.2 Condition code bit definitions

N - the result of the last arithmetic operation was negative, that is, its sign bit (bit 7) was on.

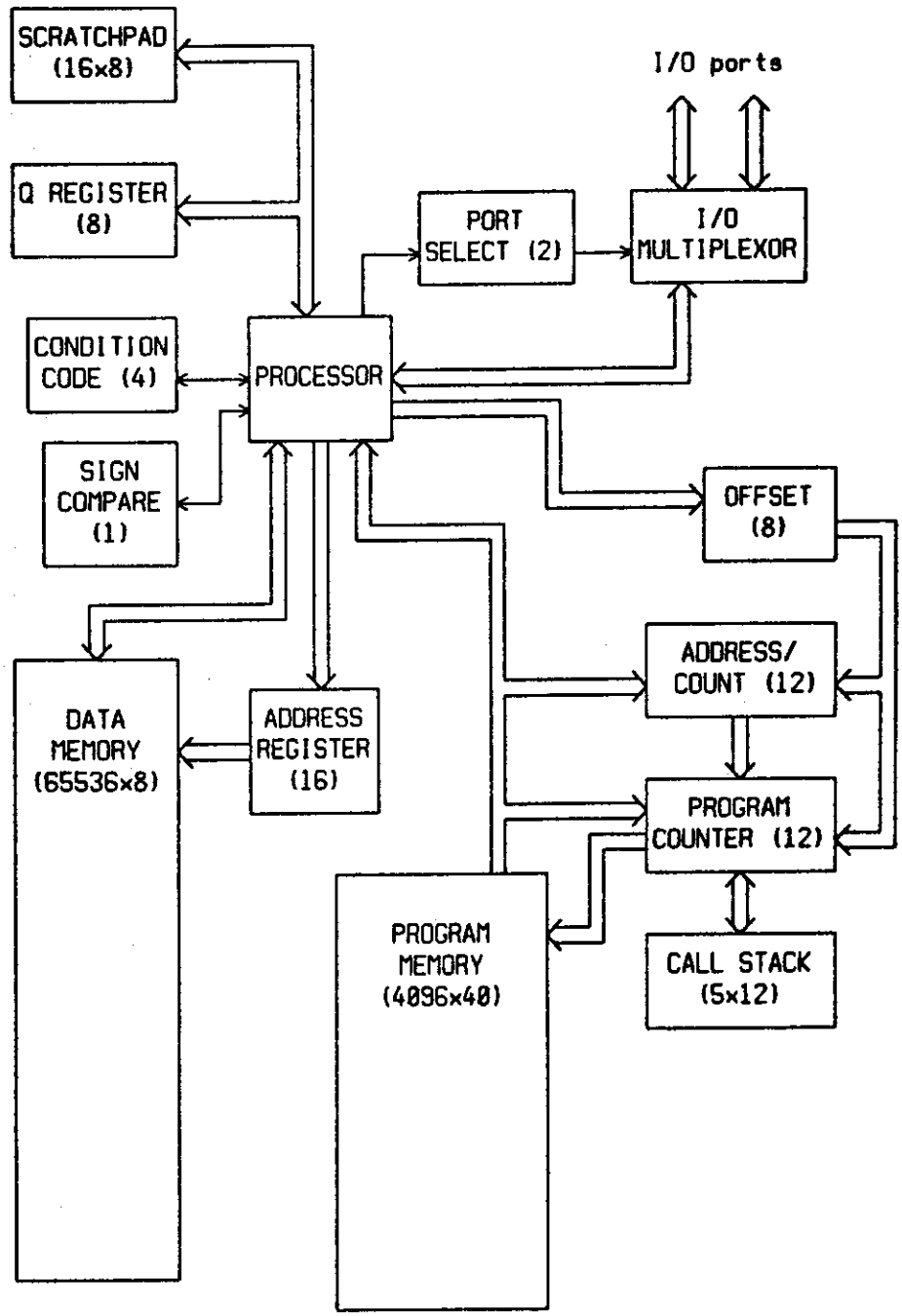
Z - the result of the last arithmetic operation was zero.

C - a carry came out of the sign bit of the last addition, considering a subtraction to be implemented as X-Y = COMP(COMP(X)+Y). (Warning: this definition for subtraction is opposite to the definition used on a PDP-11. It is the complement of the "borrow" condition.) This bit also contains the bit shifted off the end of certain shift operations.

V - the last addition or subtraction overflowed, which means that the true algebraic result is not within the range [-128, 127], and hence can not be correctly represented in two's complement. If this happens, the result that appears is the algebraically correct result plus or minus 256. The following two conditions are also equivalent definitions of overflow:

> a carry (or borrow) occurred from bit 7 but not from bit 6, or vice-versa, that is, overflow is the exclusive OR of the carry conditions from these two bits of the adder.
>
> two numbers of the same sign were added but the result had the opposite sign, or two numbers of different sign were subtracted but the result did not have the same sign as the minuend.

Figure 1. Logical diagram of the MP



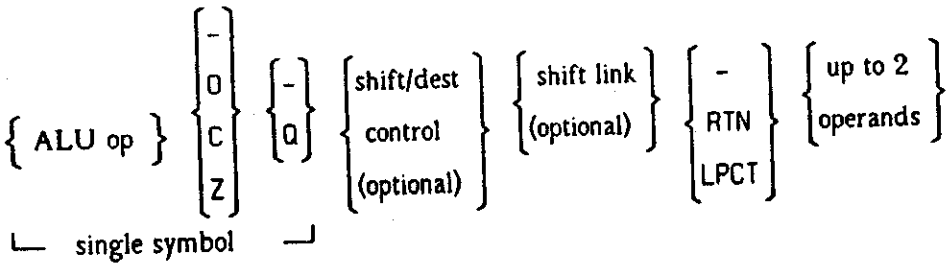
2. Summary of instruction classes

Class I - Arithmetic and Shift

This can perform the full set of arithmetic, logical, and shift instructions, manipulating scratchpad registers and the "Q" register. The "special" instructions (multiply, divide, etc.) are permitted. It can perform a few program control operations. Data may not be read from or written to data memory, IO ports or registers, or the condition code register.

Note -- in the following syntax diagrams, a hyphen is used to indicate that a symbol or option letter within a symbol may be omitted. The symbol "<null>" will be used later in the instruction tables to denote an omitted field.

The instruction format is:



Examples:

```
ADD X,Y          ; X+Y -> Y
ANDQ NQ RTN X,   ; X^Q -> Q, return from subroutine
SUB1C LS R X,Y   ; (X-Y-1+C) left rotated -> Y
NSRC LPCT X,Y    ; -X -> Y, count and loop
LDIVZ RD X,Y     ; last step of divide
```

The instruction symbols (but not the operand expressions) may be written in any order. The operands must always be written last.

The ALU operation field specifies the operation to be performed. These operations are described in sections 3 and 4. The operation symbol may be immediately followed by a "carryin" modifier: "O", "C", or "Z", specifying the initial carry for addition or subtraction. It may also be followed by a "Q" modifier specifying that the Q register, instead of the destination register, is to provide the second operand. If both modifiers are present, the carryin modifier must be first. The ALU operation and these modifiers must be written as a single symbol.

The shift/destination control field specifies the shifting of the ALU result and/or Q register, and the assignment of the ALU result. If it is not specified, no shift takes place and the ALU result is stored in the destination register. This field is described in section 5.

The shift link field specifies how the ends of the ALU result and/or Q register are linked when they shift. It is described in section 6.

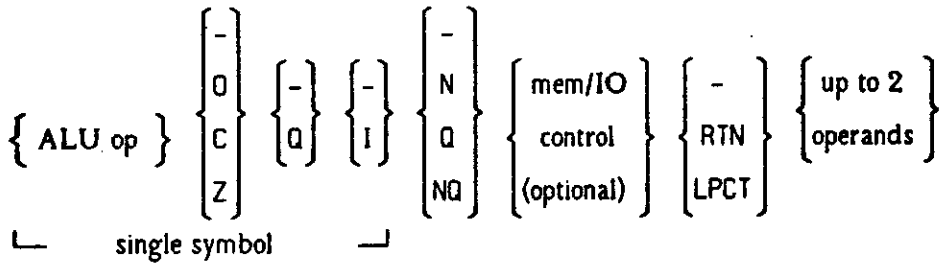
The program control field specifies the optional control action that takes place simultaneously with, and independently of, the ALU operation. There are only three choices here: nothing, LPCT, and RTN. LPCT allows the instruction to perform a loop count and iteration. RTN allows a subroutine return. These are described fully in section 9.

The operand fields are two expressions, separated by commas, which give the numbers of the "source" and "destination" scratchpad registers. For most ALU operations, the contents of these registers are the left and right operands, respectively. The result is usually stored in the destination register. If only one expression is present, it is taken as the second, and the missing first expression is assumed to be zero.

Class II - Arithmetic, IO, and Memory

This can perform arithmetic and logical instructions, manipulating scratchpad registers and the "Q" register, and can read data from or write data to the data memory, IO ports, or special registers. It can load the "IO port select" and memory address registers, and can read the IO status word and the condition code. It can also use "immediate" operands for the first operand of ALU operations. Shifts and the "special" instructions (multiply, divide, etc.) are not permitted. It can perform a few program control operations.

The instruction format is:



Examples:

```
ADDI 3,Y          ; 3+Y -> Y
ANDQI NQ 7,       ; 7^Q -> Q
SUB MR LPCT X,Y ; X-<memory> -> Y, count and loop
DST RIOSTAT Y     ; IO status -> Y
DST N RIOSTATM  ; IO status -> memory
DST N WM Y        ; Y -> <memory>
ADD N MWPSEL X, ; X+<memory> -> port select
```

The ALU operation field is the same as for class I, except that a third modifier is permitted: "I". If this is on, the first operand of the instruction is "immediate". It is 8 bits instead of four, and is the actual data to be used as the first ALU operand, rather than the number of the scratchpad register from which the first operand will be taken. The "I" modifier, if present, must follow the "carryin" and/or "Q" modifiers. The ALU operation and all modifiers must be written as a single symbol.

The destination control field specifies the assignment of the ALU result. It is similar to the "shift/destination" field in class I, except that no shift is permitted, so the only available options are nothing, "N", "Q", and "NQ". If it is not specified, the ALU result is stored in the destination register.

The memory/IO control field specifies how input data from an IO port, data memory, or other register is to be substituted for the destination register as the second ALU operand, and whether the result is to be sent to an IO port, data memory, or other register. It is described in section 7.

The program control field behaves exactly as in class I.

The operand fields are similar to the operand fields in class I, except that their interpretation is more complicated in some cases. If the "I" option is on, the first operand is immediate. Options specified in the memory/IO control field may cause the second ALU operand to be other than the scratchpad register specified by the second expression. If only one expression is present, it is taken as the second, and the missing first expression is assumed to be zero.

Class III - CC Operations

This class directly modifies condition code bits. It can set, clear, or complement any or all bits, load any or all bits from a scratchpad register, or load the "C" and/or "V" bits from each other. Like classes I and II, it can perform a few program control operations. This instruction class does not change the condition code except as specified.

The instruction format is:

{ CC operation } { - / RTN / LPCT } { operand (optional) }

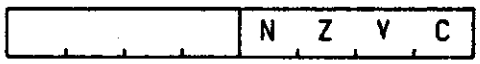
Examples:

```
SEZ             ; 1 -> Z
CLC CLN RTN     ; 0 -> C, 0 -> N, return from subroutine
LVC LCV         ; exchange C and V
IVC LPCT        ; complement C, count and loop
LCC Y           ; load N, Z, V, C from register Y
```

The CC operation field specifies the operation to be performed and the bits to be affected. Several symbols may be written here, as long as they specify the same operation. For example, "CLC CLN" causes "C" and "N" to be cleared, and "Z" and "V" to be unaffected. "CLC IVN" is illegal because the operations are different. The operations are described in section 8.

The program control field behaves exactly as in class I.

The operand field specifies the scratchpad register to be used for loading operations. Each condition code bit is loaded from a specific bit of this register, in this format:



Class IV - Program Control

This performs the full set of operations affecting program flow. It manipulates the call stack, address/counter register, and program counter, and makes decisions based on the condition code. No scratchpad register is ever changed, nor is the condition code changed, by any of these instructions.

The instruction format is:

{ control operation } { condition (optional) } { - REG } { operand (optional) }

Examples:

```
JSR FOO         ; call subroutine at location FOO
JMP GT FOO      ; if last result was > 0, jump to location FOO
LDCT 35         ; put 35 into address/counter
LSETUP CS REG   ; if C = 1, copy offset register into address/counter; push
LPCT            ; count and loop
EXIT FOO        ; exit to FOO unconditionally
```

The control operation field specifies the action to be performed. These actions are described in section 9. Many of these operations depend on a condition, and many of them use a 12 bit "effective address" either as a count or as a program address.

The condition field specifies how the condition that controls the operation is to be computed from the 4 bits of the condition code. If nothing is specified, the condition is always true. This field is described in section 10.

The "REG" option causes the effective address to be computed from the offset register, instead of (usually) being the operand field itself.

The operand field is a 12 bit quantity used to compute the effective address. If it is omitted it is taken as zero.

The effective address computation is as follows:

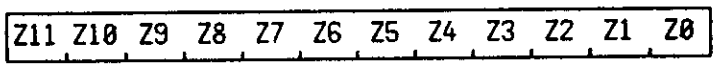
If the "REG" option is not specified and the operation is not VJMP or JCB, the effective address is the entire 12 bit operand.

If "REG" is not specified and the instruction is VJMP, the left 8 bits of the effective address are the left 8 bits of the operand, and the right 4 bits of the effective address are the right 4 bits of the offset register.

If "REG" is not specified and the instruction is JCB, the left 8 bits of the effective address are the left 8 bits of the operand. The right 4 bits of the effective address are the number of the bit position of the leftmost zero bit in the offset register, or 8 if there are no zeros.

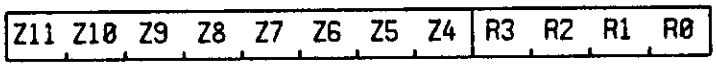
If "REG" is specified, the left 4 bits of the effective address are the left 4 bits of the operand, and the right 8 bits of the effective address are the contents of the offset register.

No REG option, instruction ≠ VJMP or JCB



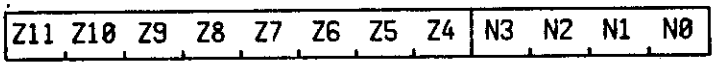
(Z = operand)

No REG option, instruction = VJMP



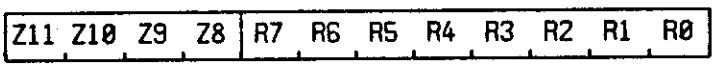
(R = contents of offset register)

No REG option, instruction = JCB



(N = bit position (7 through 0) of leftmost zero in R, or 8 if R = 11111111)

REG option used, any instruction



3. ALU OPERATIONS

These are available in classes I and II.

ZERO	zero
XFF	-1, that is, 377_8 or FF_{16} (no "Q" modifier allowed for this instruction)
SRC	SRC+carryin
NSRC	$\overline{SRC}$+1, that is, -SRC [6]
CSRC	$\overline{SRC}$+carryin, that is, -SRC-1+carryin
DST	DST+carryin
NDST	$\overline{DST}$+1, that is, -DST [6]
CDST	$\overline{DST}$+carryin, that is, -DST-1+carryin
QREG	QREG+carryin [6]
NQREG	$\overline{QREG}$+1, that is, -QREG [6]
CQREG	$\overline{QREG}$+carryin, that is, -QREG-1+carryin [6]
ADD	SRC+DST+carryin
SUB	SRC+$\overline{DST}$+1, that is, SRC-DST [6]
RSUB	$\overline{SRC}$+DST+1, that is, DST-SRC [6]
SUB1, ADDCDST	SRC+$\overline{DST}$+carryin, that is, SRC-DST-1+carryin
RSUB1, ADDCSRC	$\overline{SRC}$+DST+carryin, that is, DST-SRC-1+carryin
AND	SRC∧DST
OR, BIS	SRC∨DST
XOR	SRC⊕DST
NAND	$\overline{SRC \wedge DST}$
NOR	$\overline{SRC \vee DST}$
XNOR, EQV	$\overline{SRC \oplus DST}$
ANDCSRC, BIC	$\overline{SRC}$∧DST

Notes:

1. Those instructions which perform an addition or subtraction (SRC through RSUB1 in the above list) leave

V on if the addition overflowed
C on if a carry occurred out of the sign bit
N on if the ALU result before any shift is negative
Z on if the final result (including shift) is zero

The remaining instructions leave

V off
C off
N on if the ALU result before any shift is negative
Z on if the final result (including shift) is zero

2. Assignment to the C bit shown here is overridden if the shift link field specifies a shift into the C bit. See section 6.

3. The carryin modifier may be

<null>	add zero
0	add one
C	add one if the previous C bit is on

There is another carryin modifier, available only with the MPY, UMPY, LMPY, NORM, DNORM, SMCVT, DIV, and LDIV instructions:

Z	add one if the Z bit will be set by this instruction

(For normal instructions, the state of the Z bit upon completion depends on the result, which may depend on the carry in condition, so this modifier would be meaningless.)

The "0" carryin specification may be used to compensate for the subtraction of one that happens in some instructions, so, for example, "SUB10" computes SRC-DST. The "C" carryin specification is useful for multiple precision arithmetic. The carryin modifier is not permitted for instructions which do not perform an addition or subtraction, or for those mnemonics listed in note 6 which have implied carryin specifications.

4. If the "Q" modifier is used for any of these instructions, the second ALU operand (denoted DST in the table) is read out of the Q register (prior to any Q register shift) instead of being read out of the DST register. The result is still written back into the selected DST register if the shift/destination control field says to do so.

5. If the "I" field is on, immediate data is used instead of the contents of the SRC register. In this case, a shift must not be performed (section 5), and the shift link field (section 6) may not be specified. Immediate data is permitted only in class II instructions.

6. Some of the instructions are actually just abbreviations for certain useful combinations of carryin and "Q" modifiers. Explicit carryin or "Q" modifiers are not permitted when the modifier is implicitly specified in this way. For example, NDSTC and CQREGQ are meaningless. The abbreviations are:

NSRC = CSRC0

NDST = CDST0

QREG = DSTQ

NQREG = CDST0Q

CQREG = CDSTQ

SUB = SUB1C

RSUB = RSUB1C

4. SPECIAL ALU OPERATIONS

These are available only in class I.

They do not specify a shift/destination control field, and may not specify a "Q" modifier. They may specify a shift link (see section 6) to control the shifting that is implicitly performed.

Note that some of these operations are defined in terms of an intermediate "ALU result" which is not the final result stored in the DST register. The ALU result is generally the result of an addition or subtraction, and the final result is generally the result of a shift of the ALU result.

INC, INCO, INCC, INCZ

Result = DST+1+carryin

The result is assigned to the DST register without a shift. The parity of the shift in bit and the result is made available to the linker, as if this were a right shifting instruction, just as for the <null> shift/destination control field. The Q register is not shifted. The data presented from the Q register to the linker is undefined. Note that INCO adds 2 to the DST register.

This leaves
- V on if the addition overflowed
- C on if a carry occurred out of the sign bit
- N on if the result is negative
- Z on if the result is zero

SMCVT, SMCVTO, SMCVTC, SMCVTZ

if DST < 0, ALU result = $\overline{\text{DST}}$+carryin = -DST-1+carryin, and final result = ALU result$\oplus 200_8$
if DST ≥ 0, ALU result = DST+carryin, and final result = ALU result

The final result is stored in the DST register without a shift. The Q register is not shifted. The parity of the shift in bit and the ALU result is made available to the linker, as if this were a right shifting instruction, just as for the <null> shift/destination control field. The data presented from the Q register to the linker is undefined.

This leaves
- V on if the addition overflowed
- C on if a carry occurred out of the sign bit
- N on if the final result is negative
- Z on if the original DST was negative

This instruction is intended to be used with the "Z" carryin modifier (instruction = SMCVTZ), so the action is:

if DST < 0, ALU result = -DST, and final result = (-DST)⊕200_8
if DST ≥ 0, ALU result = DST, and final result = DST

SMCVTZ converts data in either direction between sign-magnitude and twos-complement form. If the original data is 200_8, the result will be zero and (in this case only) "V" will be set.

UMPY, UMPYO, UMPYC, UMPYZ

If Q_0 = 1, ALU result = SRC+DST+carryin
If Q_0 = 0, ALU result = DST+carryin

The ALU result is shifted right, shifting into the sign bit the "carry out" bit (i.e. what will go into the "C" bit), regardless of what shift link is selected. Bit 0 is made available to the shift linker, as usual. The Q register is shifted right, making bit zero available to the linker and shifting data from the linker into the sign bit. The result of the ALU shift is stored in the DST register.

This leaves
V on if the addition overflowed
C on if a carry occurred out of the sign bit
N on if the ALU result before the shift is negative
Z on if Q_0 was on prior to its shift

This instruction is intended to be used with no carryin modifier (instruction = UMPY), and with shift link = "D", so the action is:

If Q_0 = 1, ALU result = SRC+DST
If Q_0 = 0, ALU result = DST

and the ALU result and Q register shift right together.

MPY, MPYO, MPYC, MPYZ

This is the same as UMPY except that, when the ALU result is shifted right, the data shifted into the sign bit is the exclusive OR of the overflow condition and the previous contents of the sign bit of the ALU result (i.e. the exclusive OR of what will go into the "V" and "N" bits).

Like UMPY, this instruction is intended to be used with no carryin modifier, and with shift link = "D".

LMPY, LMPYO, LMPYC, LMPYZ

This is the same as MPY except that the ALU function is

If Q_0 = 1, ALU result = DST-SRC-1+carryin
If Q_0 = 0, ALU result = DST+carryin

The shift of the result, and the assignment to the condition code is the same as for MPY. When the ALU result is shifted right, the data shifted into the sign bit is the exclusive OR of the overflow condition and the previous contents of the sign bit of the ALU result (i.e. the exclusive OR of what will go into the "V" and "N" bits).

This instruction is intended to be used with the "Z" carryin modifier (instruction = LMPYZ), and with shift link = "D", so the action is:

If Q_0 = 1, ALU result = DST-SRC
If Q_0 = 0, ALU result = DST

and the ALU result and Q register shift right together.

NORM, NORMO, NORMC, NORMZ

result = DST+carryin

This is treated as a "left shifting" operation. The sign bit of the result is made available to the shift linker, even though the result is not shifted. The Q register is shifted left, making its sign bit available to the linker and shifting data from the linker into bit 0. The result is stored in the DST register.

This leaves
V = original $Q_6 \oplus Q_5$
C = original $Q_7 \oplus Q_6$
N = original Q_7, i.e. the bit shifted out of the Q register
Z on if the original Q register was zero

DNORM, DNORMO, DNORMC, DNORMZ

ALU result = DST+carryin

The ALU result is shifted left, shifting into bit zero the data provided by the linker. The exclusive OR of the sign bit of the ALU result and the sign bit of the SRC register is made available to the linker as the bit shifting out of the result. The complement of this is also stored in the "sign compare" flip-flop. This flip-flop is loaded only by the DNORM and DIV instructions, and is preserved by all

others. The Q register is shifted left, making its sign bit available to the linker and shifting data from the linker into bit 0. The result of the ALU shift is stored in the DST register.

This leaves

V = original ALU_6 ⊕ALU_5
C = original ALU_7 ⊕ALU_6
N on if the original ALU result is negative
Z on if the original ALU result and Q register were both zero

For a normalize or first divide step operation, this instruction is intended to be used with no carryin modifier (instruction = DNORM), and with shift link = "RD", so the action is:

The DST register and Q register are shifted left together. The exclusive OR of the bit shifting out of the DST register (its sign) and the sign bit of the SRC register shifts into Q_0 , and the complement of this is also stored in the "sign compare" flip-flop.

DIV, DIVO, DIVC, DIVZ

If sign compare flip-flop = 1, ALU result = DST+$\overline{\text{SRC}}$+carryin = DST-SRC-1+carryin
If sign compare flip-flop = 0, ALU result = DST+SRC+carryin

The ALU result is shifted left, shifting into bit zero the data provided by the linker. The complement of the exclusive OR of the sign bit of the ALU result and the sign bit of the SRC register is made available to the linker as the bit shifting out of the result. This is also stored in the "sign compare" flip-flop. This flip-flop is loaded only by the DNORM and DIV instructions, and is preserved by all others. The Q register is shifted left, making its sign bit available to the linker and shifting data from the linker into bit 0. The result of the ALU shift is stored in the DST register.

This leaves

V on if the addition overflowed
C on if a carry occurred out of the sign bit
N on if the original ALU result is negative
Z = original sign compare flip-flop

This instruction is intended to be used with the "Z" carryin modifier (instruction = DIVZ), and with shift link = "RD", so the action is:

If sign compare flip-flop = 1, ALU result = DST-SRC
If sign compare flip-flop = 0, ALU result = DST+SRC

and the combined ALU result and Q register shift left. The complement of the exclusive OR of the bit shifting out of the ALU result (its sign) and the sign bit of the SRC register shifts into Q_0 and is stored in the "sign compare" flip-flop.

LDIV, LDIVO, LDIVC, LDIVZ

If sign compare flip-flop = 1, ALU result = DST+$\overline{\text{SRC}}$+carryin = DST-SRC-1+carryin
If sign compare flip-flop = 0, ALU result = DST+SRC+carryin

The ALU result is not shifted. The sign bit of the ALU result is made available to the linker as if this were a left shifting instruction. The Q register is shifted left, making its sign bit available to the linker and shifting data from the linker into bit 0. The ALU result is stored in the DST register.

This leaves

V on if the addition overflowed
C on if a carry occurred out of the sign bit
N on if the ALU result is negative
Z = sign compare flip-flop

This instruction is intended to be used with the "Z" carryin modifier (instruction = LDIVZ), and with shift link = "0", so the action is:

If sign compare flip-flop = 1, ALU result = DST-SRC
If sign compare flip-flop = 0, ALU result = DST+SRC

and the Q register shifts left, shifting a one into Q_0 . The ALU result is stored in the DST register, and its sign in the "N" bit.

5. SHIFT/DESTINATION CONTROL

These are available only in class I, except for "<null>", "N", "Q", and "NQ", which are available in classes I and II.

These control the shifting of the ALU result to make the final result, the shifting of the Q register, and the storing of the final result into the Q register and/or the DST register.

They do not apply to special instructions, which have their own implicit shift/destination rules.

<null>	no shift, store result in DST register and compute parity (linker considers this a right shift) [3a]
N	no store (result is computed but not stored anywhere) (linker considers this a left shift) [3b]
Q	store result in Q register (as well as in DST) and compute parity (linker considers this a right shift) [3a]
NQ	no store, Q register (result is stored in Q register only, not in DST) and compute parity (linker considers this a right shift) [3a]
RS	right shift
LS	left shift
RA	right arithmetic shift [2]
LA	left arithmetic shift [2]
RSRQ	right shift, right shift Q register also
LSLQ	left shift, left shift Q register also
RARQ	right arithmetic shift, right shift Q register also [2]
LALQ	left arithmetic shift, left shift Q register also [2]
NRQ	no store (result is computed but not stored anywhere), right shift Q register, compute parity [3a]
NLQ	no store (result is computed but not stored anywhere), left shift Q register [3b]
LXT	left sign extend: the ALU result is ignored and replaced by 8 copies of the bit provided

by the shift linker. That bit is also sent to the linker as the bit shifting out of the left end. (linker considers this a left shift) [3b]

Y17 no shift, rather useless (linker considers this a left shift) [3b]

Notes:

1. All codes except those containing the letter "N" store the final result in the selected DST register.

2. Q register shifts are always 8 bits. Shifts of the ALU result are 8 bits if the shift is RS or LS, 7 bit shift leaving sign alone if RA or LA. If LA, it is bit 6 that is sent to the shift linker. If RA, the linker data goes into bit 6.

3. Codes <null>, Q, and NQ are considered to be "right" codes, and codes N, LXT, and Y17 are "left", even though no shift takes place. This is important in considering the behavior of the shift linker.

3a. Right instructions <null>, Q, NQ, and NRQ make the parity of the shift-in bit from the linker and the 8 bits of the ALU result available to the linker. Hence <null> with shift link "UN" sets the "C" bit to the parity of the ALU result.

3b. Left instructions N, NLQ, and Y17 make the sign bit (bit 7) available to the linker, even though they do not shift. Hence "N" with shift link "C" (see section 6) copies the sign bit into the C bit but otherwise ignores the result. Instruction LXT makes whatever the linker provides at the right end available to the linker at the left end. Hence "LXT" with shift link "0C" sets the result to 377_8 and shifts a one into the C bit.

4. A bit from the Q register is presented to the linker only if the Q register is shifting: RSRQ, LSLQ, RARQ, LALQ, NRQ, or NLQ. For other specifications, the data presented from the Q register to the linker is undefined.

5. The N bit is set from the ALU result before any shift, but the Z bit is set from the final result. Hence "ZERO LXT" with shift link "0C" sets C (because a one shifts into it), clears N (because the ALU result is zero, which is not negative), and clears Z (because the final result is 377_8, which is not zero).

6. SHIFT LINK field

These are available only in class I.

This field has two completely different interpretations, depending on whether the ALU instruction is considered to be a right shift or a left shift. For right shift instructions (including <null>, Q, NQ and special instructions UMPY, MPY, INC, SMCVT, and LMPY) the linker receives data that it presumes to be shifting out of bit zero of the ALU result and the Q register, and provides data to be presumably shifted into the sign bit of the ALU result and the Q register. For left shift instructions (including N, LXT, Y17, and special instructions NORM, DNORM, DIV, and LDIV) it does the opposite. In either case, if the description below shows data being shifted into the "C" bit, that shift takes precedence over the normal loading of the C bit from the carry out condition of the adder.

The letters appearing in the mnemonics have the following general meaning:

R rotate, i.e. insert the bit shifting out back in at the other end

O shift in a one every place not otherwise specified, instead of zero.

D double - ALU result is left half and Q register is right half of a 16 bit register

C consider C bit to be appended to left end of ALU result (but don't shift in a zero or one)

U "un-C" - consider C bit to be at right end of ALU result or double register

BC "branch carry" - copy whatever is shifting past the left end of the ALU result into C bit, but don't put C bit into the shift chain

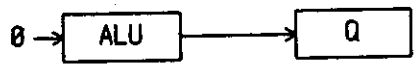
N shift the N bit in

6.1 Right shifts

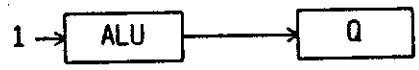
<null>

0

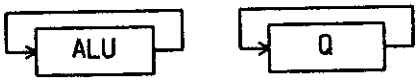
D



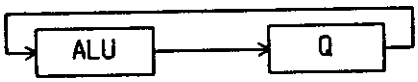
D0



R



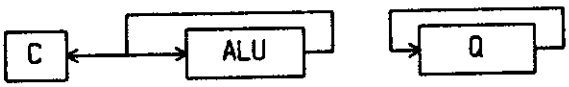
RD



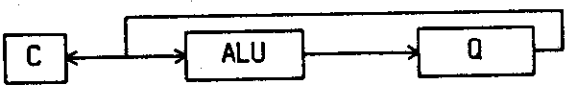
RC

RDC

RBC

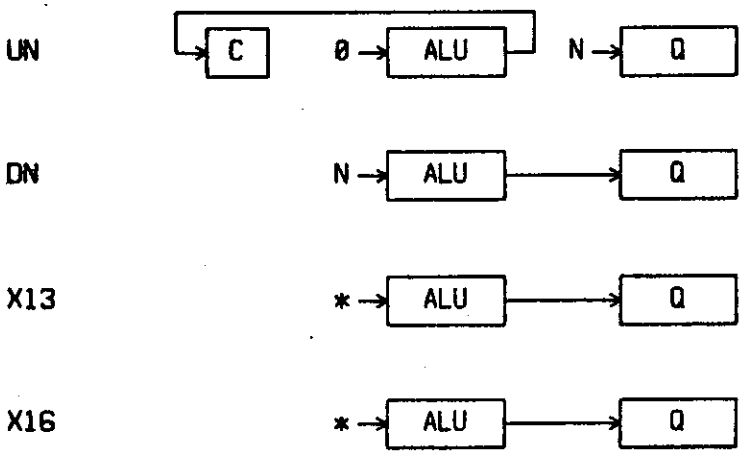


RDBC



DC

DU

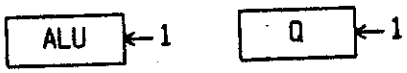


* X13 shifts the next "C" bit (the bit that is simultaneously being stored in "C") into the left end of the ALU result. X16 shifts in the exclusive OR of the next "N" bit and the next "V" bit. These codes were intended for compatibility with ALU chips that were not as sophisticated as the 2903 in performing the MPY and UMPY instructions.

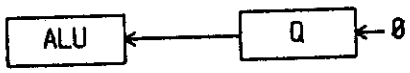
6.2 Left shifts

<null>

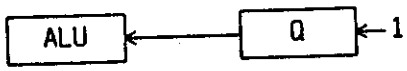
0



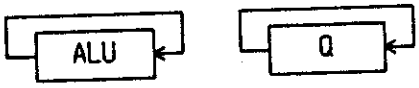
D



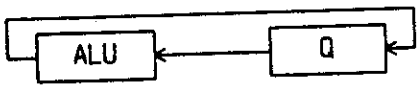
D0



R



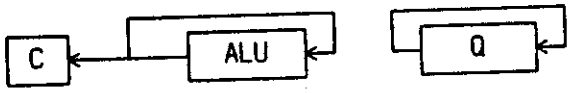
RD



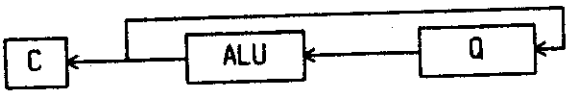
RC

RDC

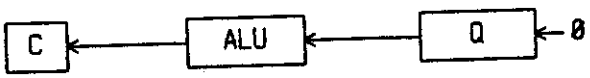
RBC



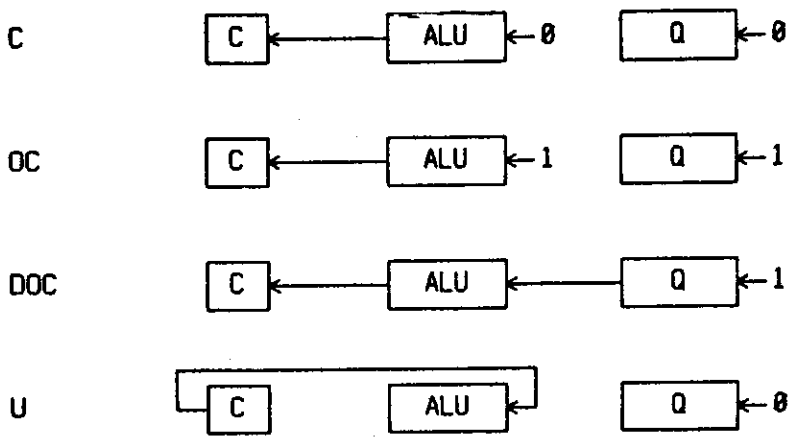
RDBC



DC



DU


C
C
ALU
0
Q
0
OC
C
ALU
1
Q
1
DOC
C
ALU
Q
1
U
C
ALU
Q
0

7. IO/memory field

These are available only in class II.

<null> Performs the same operation that would happen in class I.

MR Performs the indicated operation, but the data in the data memory is used as the second ALU operand instead of the DST register. The final result is still stored in the DST register if the destination field says to do so. The data is read from the memory at the address given by the memory address register.

WM Performs the indicated operation, but the final result is stored in the data memory, as well as being stored in the DST register and/or Q register as indicated by the destination control field. The data is stored in memory at the address given by the memory address register.

RIODAT The incoming IO data from the selected port is used as the second ALU operand, instead of the DST register. The final result is still stored in the DST register if the destination control field says to do so. The incoming data byte is acknowledged, and the "input ready" status bit is cleared. This instruction should not be executed if the "input ready" status bit for the selected port is off.

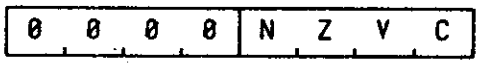
RIODATM Similar to RIODAT, but the final result is stored in the data memory, as well as being stored in the DST register and/or Q register as indicated by the destination control field. The data is stored in memory at the address given by the memory address register.

RIOSTAT Similar to RIODAT, but the status word for the IO ports is used as the second ALU operand. The status does not change, and no acknowledges are sent. The status byte is in this format:

IR0 Input port zero is ready, that is, it has a data byte available to be read by a RIODAT instruction.
IR1 Input port one is ready.
IRS The selected input port (determined by bit 1 of the port select register) is ready.
LBS If the selected input port is ready, the byte has its "last byte" bit set. This condition must be sensed before the actual data is read.
OR0 Output port zero is ready, that is, it may have a data byte transmitted by a WIODAT or WIOLAST instruction.
OR1 Output port one is ready.
ORS The selected output port (determined by bit 0 of the port select register) is ready.

RIOSTATM Similar to RIOSTAT, but the final result is stored in the data memory in addition to its other destinations.

RCC Similar to RIODAT, but the condition code is used as the second ALU operand, in this format:



The condition code is subsequently changed in accordance with the ALU operation.

RCCM Similar to RCC, but the final result is stored in the data memory in addition to its other destinations.

WIODAT The final result of the operation is transmitted on the selected IO port, as well as being stored in the destinations indicated by the destination control field. The "last byte" bit is transmitted as zero. This instruction should not be executed if the "output ready" status bit for the selected port is off.

MWIODAT Similar to WIODAT, but the data in the data memory is used as the second ALU operand instead of the DST register. The final result is still stored in the DST register (as well as being transmitted on the IO port) if the destination field says to do so. The data is read from the memory at the address given by the memory address register.

WIOLAST Similar to WIODAT, but the "last byte" bit is transmitted as one.

MWIOLAST Similar to WIOLAST, but the data in the data memory is used as the second ALU operand instead of the DST register.

WARL Similar to WIODAT, but the final result of the operation is stored in the left half of the memory address register, as well as being stored in the destinations indicated by the destination control field. The right half of the memory address register is not changed.

MWARL Similar to WARL, but the data in the data memory (using the old contents of the address register) is used as the second ALU operand instead of the DST register.

WARR Similar to WIODAT, but the final result of the operation is stored in the right half of the memory address register, as well as being stored in the destinations indicated by the destination control field. The left half of the memory address register is not changed.

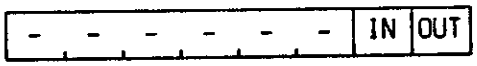
MWARR Similar to WARR, but the data in the data memory (using the old contents of the address register) is used as the second ALU operand instead of the DST register.

WOFF Similar to WIODAT, but the final result of the operation is stored in the offset register, as well as being stored in the destinations indicated by the destination control field. The

data in the offset register will be used for address modification of VJMP and JCB control instructions, or any control (class IV) instructions that use the REG option.

MWOFF Similar to WOFF, but the data in the data memory is used as the second ALU operand instead of the DST register.

WPSEL Similar to WIODAT, but the final result of the operation is stored in the IO port select register (right two bits only), as well as being stored in the destinations indicated by the destination control field. The port select register has this format:



IN Port selected for input (controls RIODAT and the "IRS" and "LBS" bits of RIOSTAT)

OUT Port selected for output (controls WIODAT, WIOLAST and the "ORS" bit of RIOSTAT)

MWPSEL Similar to WPSEL, but the data in the data memory is used as the second ALU operand instead of the DST register.

8. CC OPERATION field

These are available only in class III.

SEN, SEZ, SEV, SEC	Set the "N", "Z", "V", or "C" bit, respectively. Any or all of these may be used together.
SCC	Set all four condition code bits. Equivalent to "SEN SEZ SEV SEC".
CLN, CLZ, CLV, CLC	Clear the "N", "Z", "V", or "C" bit, respectively. Any or all of these may be used together.
CCC	Clear all four condition code bits. Equivalent to "CLN CLZ CLV CLC".
IVN, IVZ, IVV, IVC	Invert (complement) the "N", "Z", "V", or "C" bit, respectively. Any or all of these may be used together.
ICC	Invert all four condition code bits. Equivalent to "IVN IVZ IVV IVC".
LDN	Load the "N" bit from bit 3 of the indicated scratchpad register. The other bits are unaffected.
LDZ	Load the "Z" bit from bit 2 of the indicated scratchpad register. The other bits are unaffected.
LDV	Load the "V" bit from bit 1 of the indicated scratchpad register. The other bits are unaffected.
LDC	Load the "C" bit from bit 0 of the indicated scratchpad register. The other bits are unaffected.
LCC	Load the entire condition code from the low four bits of the indicated scratchpad register. Equivalent to "LDN LDZ LDV LDC".
LVC	Load "V" from "C".
LCV	Load "C" from "V". LVC and LCV together exchange "C" and "V".

9. CONTROL OPERATION field

These are available only in class IV, except for "<null>", "RTN" and "LPCT", which are available in all classes.

Many of these depend on the condition computed from the condition code and specified by the condition field described in section 10.

<null> Do nothing, continue with next instruction. The condition is ignored.

JMP (conditional) If the condition is true, jump to the effective address.

JMPR (conditional) Jump or use register. If the condition is true, jump to the effective address. If not, jump to the address contained in the address/count register.

JSR (conditional) Jump to subroutine. If the condition is true, push the PC and jump to the effective address. [1]

JSRR (conditional) Jump to subroutine or use register. If the condition is true, push the PC and jump to the effective address. If not, push the PC and jump to the address contained in the address/count register. [1]

RTN (conditional) Return. If the condition is true, pop the top address from the stack and jump there. [2]

VJMP Vector jump. Always jump to the effective address. (If the REG option is off, the low 4 bits of the effective address are the low 4 bits of the offset register.) The condition is ignored. [3]

JCB (conditional) Jump and count bits. If the condition is true, jump to the effective address. (If the REG option is off, the low 4 bits of the effective address are set to the number of the bit position of the leftmost zero bit in the offset register.) [3]

LDCT Load count. Load the address/count register with the number which is the effective address. The condition is ignored.

LSETUP (conditional) Loop setup. If the condition is true, load the address/count register with the number which is the effective address. Push the PC in any case. [1]

LOOP (conditional) If the condition is true, pop and discard the top address from the stack. If not, jump to the address on top of the stack without popping it.

LPCT Loop count. If the address/count register is nonzero, decrement it and jump to the address on top of the stack without popping it. If it is zero, pop and discard the top address from the stack. The condition is ignored.

COUNT If the address/count register is nonzero, decrement it and jump to the effective address. If it is zero, do nothing. The condition is ignored.

EXIT (conditional) If the condition is true, pop and discard the top address from the stack and jump to the effective address. If not, do nothing.

TWB (conditional) Three way branch. If the condition is true, pop and discard the top address from the stack and decrement the address/count register if it is nonzero. If not and the address/count register is nonzero, decrement it and jump to the address on top of the stack without popping it. If the condition is false and the address/count register is zero, pop and discard the top address from the stack and jump to the effective address.

RESET Jump to location zero and clear the stack. The condition is ignored.

Notes:

1. The call stack can hold five items. If too many items are pushed onto it, the oldest item is lost. No indication is given when this happens.

2. If the RTN operation is given in a class I, II, or III instruction, a condition may not be specified. The operation takes place unconditionally in this case.

3. The instructions VJMP and JCB use an unusual effective address calculation. See section 2.

10. CONDITION field

These are available only in class IV.

The condition is computed from the "N", "Z", "V", and "C" bits left by the previous instruction.

name	logical condition	meaning after SUB X, Y	meaning after DST X
<null>	Always true		
MI, NS	N		X < 0 signed
PL, NC	$\overline{N}$		X ≥ 0 signed
EQ, ZS	Z	X = Y	X = 0
NE, ZC	$\overline{Z}$	X ≠ Y	X ≠ 0
VS	V	overflow	
VC	$\overline{V}$	no overflow	
HIS, CS	C	X ≥ Y unsigned	
LO, CC	$\overline{C}$	X < Y unsigned	
HI	$C \wedge \overline{Z}$	X > Y unsigned	
LOS	$\overline{C} \vee Z$	X ≤ Y unsigned	
GT	$(\overline{N} \oplus V) \wedge \overline{Z}$	X > Y signed	X > 0 signed
GE	$\overline{N} \oplus V$	X ≥ Y signed	X ≥ 0 signed
LT	$N \oplus V$	X < Y signed	X < 0 signed
LE	$(N \oplus V) \vee Z$	X ≤ Y signed	X ≤ 0 signed
CZ	$C \vee Z$		
NCZ	$\overline{C \vee Z}$		

11. Applications

Subroutine call and return

```
        . . . . .
        JSR CS SUBR          ; call if "C" is on
        . . . . .

SUBR:   . . . . .
        . . . . .
        RTN GE               ; return if last result >= 0
        . . . . .
        ADD RTN 4,5          ; do some arithmetic and
                             ;   unconditional return
```

Loop under count (uses address/count register for count and call stack for restart address)

```
        LSETUP 5
        . . . . .            ; all of this happens 6 times
        . . . . .
        LPCT                 ; discards return address when finished
```

Loop under test (uses call stack for restart address)

```
        LSETUP
        . . . . .            ; repeats until "V" is off
        . . . . .
        LOOP VC              ; discards return address when finished
```

Loop under count without using stack (Since nothing is added to the stack, the loop may be exited at any time with a JMP or similar instruction.)

```
        LDCT 5
        . . . . .            ; all of this happens once
        . . . . .
AGAIN:  . . . . .            ; all of this happens 6 times
        . . . . .
        COUNT AGAIN
```

A loop under test without using the call stack may be written in a straightforward way with a conditional JMP instruction.

Abnormal exit from loop that is using the call stack

```
        LSETUP
        . . . . .
        EXIT EQ DONE            ; if result = 0, exit from loop
        . . . . .
        LOOP VC
        . . . . .
        . . . . .
DONE:   . . . . .
```

Three way branch

```
        LSETUP 5
        . . . . .               ; happens up to 6 times
        . . . . .
        TWB LT DONE             ; if result < 0, fall out of loop
                                ; if not, repeat the loop if count
                                ;   unexpired, else exit and go to DONE
        . . . . .
        . . . . .
DONE:   . . . . .
```

16 way "dispatch" jump

```
        ADD N WOFF 2,3          ; load offset register with R2+R3,
                                ;   for example
        VJMP TABLE              ; uses low 4 bits of offset register
        . . . . .
        . . . . .
LOC (.-1)|17+1                  ; round up to 16 instruction boundary
TABLE:  JMP X1                  ; here if low 4 bits = 0000
        JMP X2                  ; here if 0001
        JMP X3                  ; etc.
        . . . . .
```

Multiple precision addition - sets Y1Y2Y3 to X1X2X3 + Y1Y2Y3

```
        ADD X3, Y3
        ADDC X2, Y2
        ADDC X1, Y1
```

Multiple precision subtraction - sets Y1Y2Y3 to X1X2X3 - Y1Y2Y3

```
SUB X3, Y3
SUB1C X2, Y2
SUB1C X1, Y1
```

Unsigned Multiplication

Put the multiplicand in register X, put the multiplier in the Q register, and clear register Y.

```
LSETUP 7
UMPY D LPCT X,Y          ; happens 8 times
```

The 16 bit product will be computed, with its left 8 bits in Y and its right 8 bits in the Q register. Register X will be unchanged.

Twos complement multiplication

Put the multiplicand in register X, put the multiplier in the Q register, and clear register Y.

```
LSETUP 6
MPY D LPCT X,Y           ; happens 7 times
LMPYZ D X,Y
```

The 16 bit twos complement product will be computed, with its left 8 bits in Y and its right 8 bits in the Q register. Register X will be unchanged.

Division

To perform a division of a 16 bit twos complement dividend by a twos complement divisor, put the dividend in scratchpad register Y (high part) and the Q register (low part). Put the divisor in scratchpad register X.

```
DNORM RD X,Y
LSETUP 6
DIVZ RD LPCT X,Y         ; happens 7 times
LDIVZ D X,Y              ; now dividend = Q * divisor + Y
                         ; and -|divisor| ≤ Y < |divisor|
                         ; the "N" bit contains the sign of Y
```

The quotient is now in the Q register and the remainder is in register Y. Register X is unchanged. To

repair the case in which the remainder is negative, something such as the following might be executed:

```
JMP PL .+8.             ; jump if remainder OK
DST N X                 ; test divisor
JMP PL .+4
RSUB X,Y                ; divisor negative, fix remainder
QREGO NQ                ; fix quotient
JMP .+3
ADD X,Y                 ; divisor positive, fix remainder
ADDQI NQ -1,            ; fix quotient
                        ; now dividend = Q * divisor + Y
                        ; and 0 ≤ Y < |divisor|
```

Of course, different correction routines might be appropriate for different applications.

12. APPENDIX - OP CODES

The codes listed here are for reference in examining object microcode. They are not needed for writing programs, and may not be intelligible to people unfamiliar with the detailed operation of the MP hardware.

arithmetic operations (bits 27-24)

0	XFF/special	4	DST	8	ZERO	C	AND
1	RSUB1	5	CDST	9	ANDCSRC	D	NOR
2	SUB1	6	SRC	A	XNOR	E	NAND
3	ADD	7	CSRC	B	XOR	F	OR

special instructions (bits 23-20)

0	UMPY	4	INC	8	NORM	C	DIV
		5	SMCVT				
2	MPY	6	LMPY	A	DNORM	E	LDIV

carryin codes (bits 29-28)

0	<null>
1	0
2	Z
3	C

right shifts (bits 23-20)

0	RA	4	<null>
1	RS	5	NRQ
2	RARQ	6	NQ
3	RSRQ	7	Q

right shift links (bits 11-8)

0	<null>	4	DC	8	RBC	C	RDC
1	0	5	DN	9	RC	D	RDBC
2	UN	6	D	A	R	E	X16
3	DO	7	DU	B	X13	F	RD

left shifts (bits 23-20)

8	LA	C	N
9	LS	D	NLQ
A	LALQ	E	LXT
B	LSLQ	F	Y17

left shift links (bits 11-8)

0	C	4	DC	8	RBC	C	RDC
1	OC	5	DOC	9	RC	D	RDBC
2	<null>	6	D	A	R	E	DU
3	0	7	DO	B	U	F	RD

IO sources (bits 14-12)

0	RIODAT
1	RIOSTAT
2	RCC

IO destinations (bits 14-12)

0	<null>	4	WARR
1	WIODAT	5	WPSEL
2	WIOLAST	6	WOFF
3	WARL		

CC operations (bits 27-24)

0	LCC etc.	4	LVC/LCV
1	SCC etc.	5	ICC etc.
3	CCC etc.		

CC masks (bits 11-8)

1	C
2	V
4	Z
8	N

PC control codes (bits 19-16)

0	RESET	4	LSETUP	8	LPCT	C	LDCT
1	JSR	5	JSRR	9	COUNT	D	LOOP
2	VJMP	6	JCB	A	RTN	E	<null>
3	JMP	7	JMPR	B	EXIT	F	TWB

conditions (bits 27-24)

0	GT	4	NE	8	NCZ	C	HI
1	LE	5	EQ	9	CZ	D	LOS
2	GE	6	VC	A	LO	E	PL
3	LT	7	VS	B	HIS	F	MI