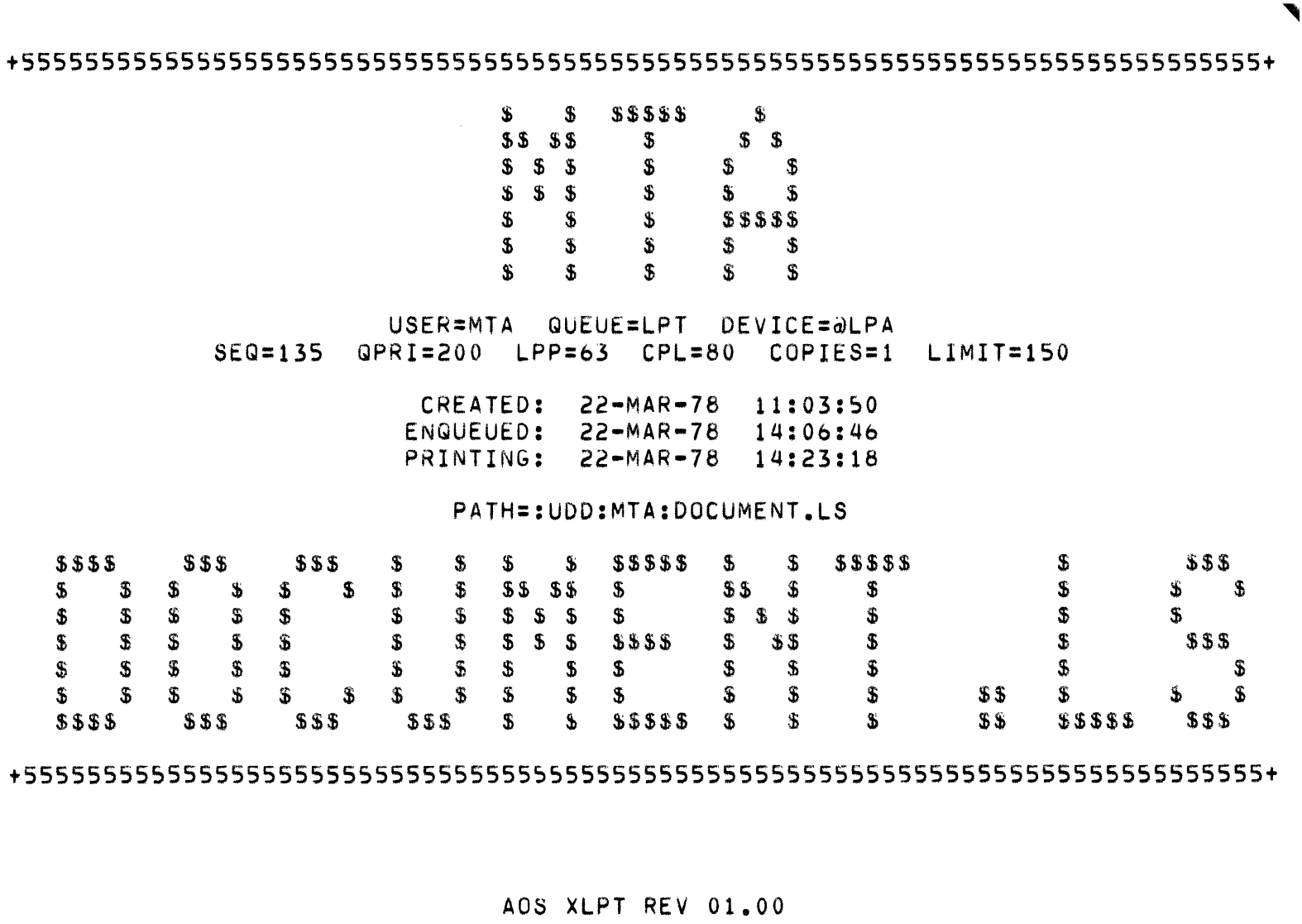

USER=MTA QUEUE=LPT DEVICE=@LPA
SEQ=135 QPRI=200 LPP=63 CPL=80 COPIES=1 LIMIT=150
CREATED: 22-MAR-78 11:03:50
ENQUEUED: 22-MAR-78 14:06:46
PRINTING: 22-MAR-78 14:23:18
PATH=:UDD:MTA:DOCUMENT.LS
AOS XLPT REV 01.00

CHAPTER 1 - Data Types and Formats

1.1 Basic Allowable Types

In this architecture, a word has 4 bytes; a half word has 2; a character, one. Double precision refers to 2-word quantities; single precision refers to 1-word quantities. This chapter enumerates the data types supported by the architecture, and describes the formats they take while residing in system memory.

1.1.1 Floating Point

Real numbers are represented in standard Data General (and IBM) format.

Figure 1-1, "Floating Point Format"

Both single precision and double precision will be suported.

1.1.2 Fixed

Fixed point numbers are supported in 2's complement integer representation. Direct support is provided for 8-, 16-, 32-, and 64-bit container sizes.

Figure 1-2, "Fixed Point Format"

1.1.3 Boolean

Boolean values occupy a one bit container and have the value zero or one.

1.1.4 String

Byte strings of lengths from 1 to 2**15 bytes are supported in the architecture. A string descriptor is associated with each byte string.

Figure 1-3, "String Format"

1.1.5 Commercial

The architecture provides direct support for the data types listed below:

* Unpacked decimal, low-order sign/overpunch
* Unpacked decimal, high-order sign/overpunch
* Unpacked decimal, trailing sign
* Unpacked decimal, leading sign
* Unpacked decimal, unsigned
* Packed decimal (IBM format)
* Binary integer, signed
* Binary integer, unsigned

Eighteen digits of precision are supported. The formats are as follows:

Figure 1-4, "Unpacked decimal, low-order sign/overpunch"

Figure 1-5, "Unpacked decimal, high-order sign/overpunch"

Figure 1-6, "Unpacked decimal, trailing sign"

Figure 1-7, "Unpacked decimal, leading sign"

Figure 1-8, "Unpacked decimal, unsigned"

Figure 1-9, "Packed decimal"

Figure 1-10, "Binary integer, signed"

Figure 1-11, "Binary integer, unsigned"

A commercial descriptor is always accociated with a commercial word.

1.1.6 Bit Numerics

The architecture supports the manipulation of a string of consecutive bits for 1 to 32 bits. When the operation specified calls for a numeric value, the bit string is considered to be in two's complement form.

1.2 Rounding

Two guard digits are provided for floating point operations, with the following rounding modes provided:

* Truncation
* Round toward zero.
* Round away from zero.
* Round toward plus infinity.
* Round toward minus infinity.
* Unbiased round.

Truncation is the only legal form of rounding in implimentations with only a single guard digit. A trap will occur if another

form of rounding is specified.

--End of Chapter--

CHAPTER 2 - Addressing

2.1 Address Structure

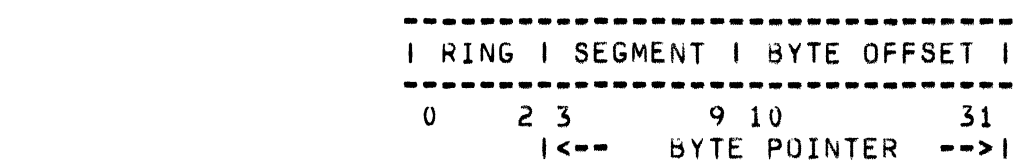
The MTA architecture supports an address space of 512 Mbytes (Mbyte = 2**20 bytes) divided into 128 segments, each containing up to 4 Mbytes. A segment can contain either procedure or data.

The basic addressing granularity is to the byte, although bit, byte, word (32 bits), half-word (16 bits) and double word (64 bit) operations are defined. The address mechanism of the memory system is always presented with a virtual address comprised of segment and byte offset within the segment. This logical address is 29 bits in length. (See Memory Management Chapter for a detailed description of memory management and the translation of the logical address to a physical address).

2.2 Architecturally Defined Registers

The processor contains the following registers for use with the standard instruction set:

* Base Registers (BR) - The 8 base registers are 32 bits wide. Their contents are interpreted as follows:



In the above format, the three bit ring field provides protection information, while the segment and byte offset fields comprise a 29 bit byte pointer.

Four base registers are allocated to the following architectural pointers:

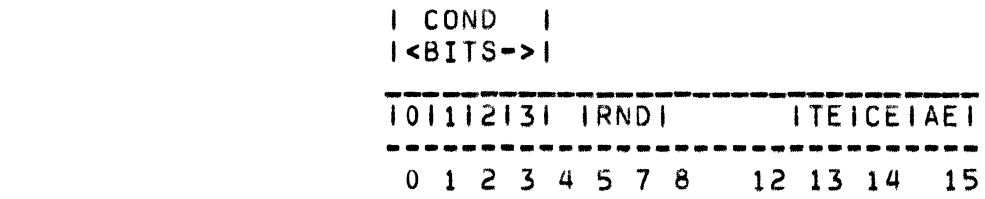
BR0 - Program Counter (PC)
BR1 - Frame Pointer (FP)
BR2 - Argument Pointer (AP)
BR3 - Global Pointer (GP)

The PC can only be modified as a result of a branch type instruction. BR1 through BR7 can only be modified by pointer-specific instructions. Violation of these restrictions is signalled by a specification error.

* General Registers (GR) - Eight 32-bit registers are defined for use in indexing and temporary storage. 64-bit entities may only be moved to an even-odd pair of GR's; this provides four 64-bit registers (DP0, DP1, DP2, DP3). String operands and commercial operands may not be referenced in a GR. Bit numeric types may only be moved to or from a GR using the bit numeric move instructions with a length field less than than 32 bits.

* Interrupt Status Register (ISR) - This 17 bit register contains an interrupt disable bit (interrupts are inhibited when this bit is set) , and 16 bits specifying which interrupts will be honored if interrupts are enabled.

* User Status Resister (USR) - The User Status Register contains user relevant status in the following format:



In this register, bits 0 - 3 are condition bits CB0, CB1, CB2, and CB3, respectively. They are altered at the conclusion of each instruction in a predefined fashion. Bits 5 - 7 specify the rounding mode of the arithmetic unit. Chapter 2 details the rounding modes. Bits 13 - 15 enable (or disable) the trace,

commercial, and arithmetic traps respectively. All bits not specified are reserved by the architecture, and must be cleared to zero.

* Stack Pointer (SP) - This register contains an address which points to the last filled location on the top of the current stack. The format of the SP is identical to that of a BR. A complete definition of the stack mechanism is provided later.

2.3 Data Address Formation

An in-line data reference is self-describing and falls into one of six categories:

a) based,

b) absolute,

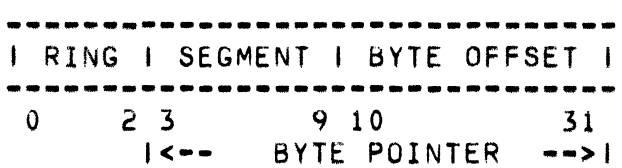
c) register,

d) local variable,

e) argument, and

f) literal.

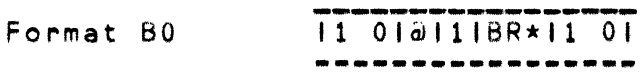
When indirection is specified (i.e. the bit labeled "@" is set), the in-line reference points to a 32-bit intermediate address which points to the desired operand; multiple-level indirection is not provided. An intermediate address has the following format:



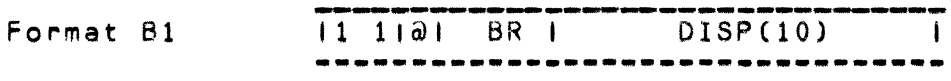
2.3.1 Based Data References

The following formats are used for based address generation:

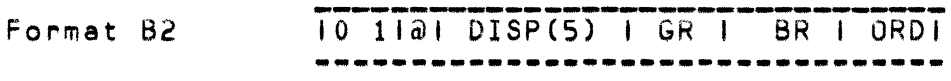
Format B0



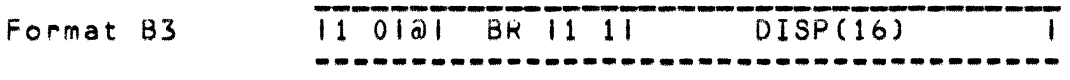
Format B1



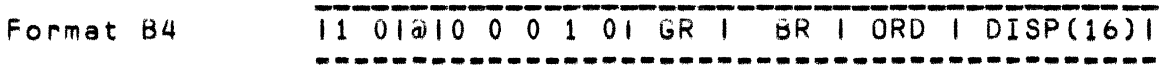
Format B2



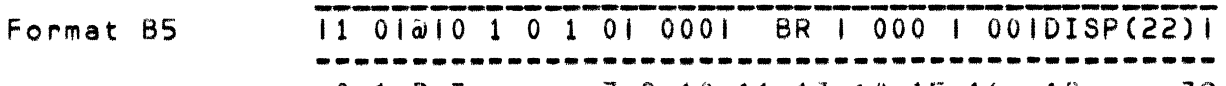
Format B3



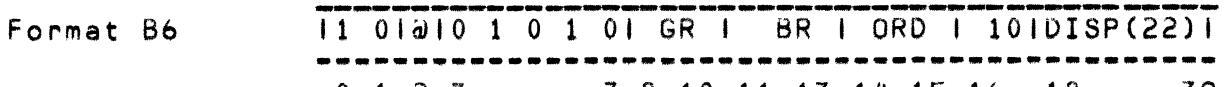
Format B4



Format B5



Format B6



Every format specifies a BR (format B0 can only specify BR4, BR5, BR6, or BR7) while some formats also specify a GR and/or a twos-complement byte displacement. When PC relative addressing is specified (the specified BR is BR0), the value of the PC used in the address resolution is the address the the instruction's first opcode byte.

Each based data reference is resolved into an effective byte address consisting of two fields: a 7-bit effective segment, and a 22-bit effective byte offset. The effective segment is simply the segment number (bits 3-9) of the byte pointer contained in the specified BR. The effective byte offset is computed usung the following terminology:

[BR(X,Y)]	represents bits X through Y of the contents of the specified BR
[GR(X,Y)]	represents bits X through Y of the contents of the specified GR
DISP(X)	represents the X bits of the specified displacement, sign extended to 22 bits.
ORD	represents the contents of the specified ordinal multiplier
+,*,**	represent twos-complement addition, multiplication, and exponentiation, respectively.

The formulas for determining the effective byte offset are:

Format	Effective Byte Offset
B0	[BR*(10,31)]
B1	[BR(10,31)] + DISP(10)
B2	[BR(10,31)] + DISP(5) + [GR(10,31)]*(2**ORD)
B3	[BR(10,31)] + DISP(16)
B4	[BR(10,31)] + DISP(16) + [GR(10,31)]*(2**ORD)
B5	[BR(10,31)] + DISP(22)
B6	[BR(10,31)] + DISP(22) + [GR(10,31)]*(2**ORD)

Formats B2, B4, and B6 provide indexed addressing by incorporating the contents of a GR in the effective byte offset computation. The two-bit ORD field permits ordinal addressing of elements in an array by specifying that the contents of the indexing GR be multiplied by a constant to account for element length. The original GR contents are not modified.

When indirection and indexing are both specified, post-indexing occurs. The intermediate address is located by:

Effective segment = [BR(3,9)]

Effective byte offset = [BR(10,31)] + DISP

When the intermediate address has been fetched, the operand address is computed as:

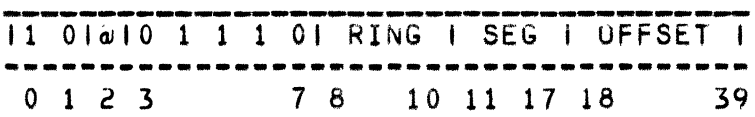
Effective segment = INT(3,9)

Effective byte offset = INT(10,31) + [GR(10,31)]*(2**ORD)

where INT(0,31) is the intermediate address

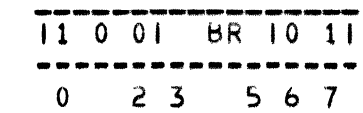
2.3.2 Absolute Data References

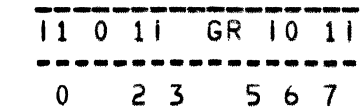
The following format is used for absolute address generation:



2.3.3 Register Data References

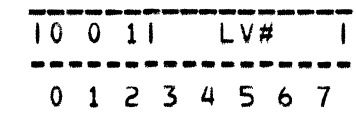
The BRs and GRs are addressed using the following formats:





2.3.4 Local Variable Data References

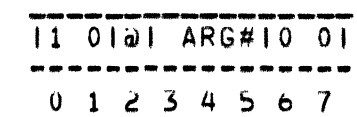
Local variables in the stack frame can be addressed using the following format:



The LV# field is interpreted as a word offset relative to BR1, the frame pointer. Thus, the effective byte offset is computed by multiplying the LV# field by 4 (forming a byte displacement) and adding it to the byte offset contained in FP.

2.3.5 Argument Data References

Arguments passed to a subroutine are addressed using the following format:

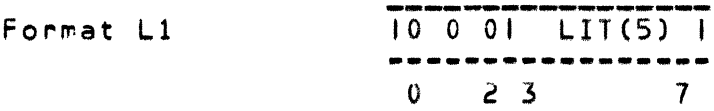


The ARG# field is interpreted as a word offset relative to BR2, the argument pointer. Thus, the effective byte offset is computed by multiplying the ARG# field by 4 (forming a byte displacement) and adding it to the byte offset contained in AP.

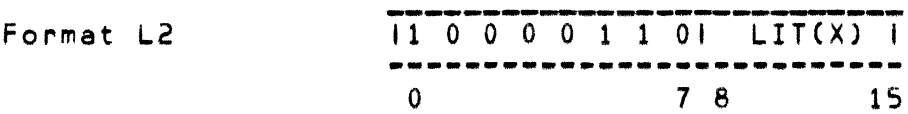
2.3.6 Literal Data References

In-line literals are generated by operand references of the following formats:

Format L1



Format L2



The length of the literal contained in format L2 is defined as the length of the data type associated with the instruction's opcode. Possible lengths are 1, 2, 3, or 4 bytes.

2.4 Operand References

An operand is specified by one, two, or three data references, depending on its data type.

2.4.1 Fixed-point Operand References

A fixed-point operand is specified by a single data reference.

2.4.2 Floating-point Operand References

A floating-point operand is specified by a single data reference.

2.4.3 Boolean Operand References

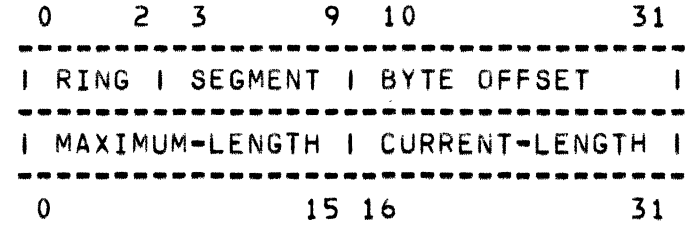
A boolean operand is specified by two data references. The first reference is a byte pointer to the base of a bit table; the second data reference provides a 32-bit bit offset from the base. The two references are referred to as a bit-ref.

2.4.4 Bit-numeric Operand References

A bit-numeric operand is specified by three data references. The first two are a byte pointer and bit offset, as for a boolean operand. The third data reference provides the 5-bit operand bit-length (in an 8-bit container). The three references are referred to as a bit-numeric-ref.

2.4.5 String Operand References

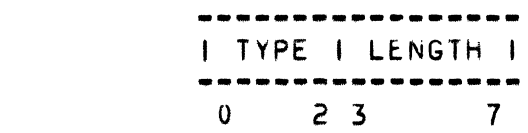
A string operand is specified by a single data reference, which points to a two-word string descriptor having the following format:



The descriptor's first word is the absolute address of the first byte of the string. Twos-complement maximum-length and current-length fields are contained in the descriptor's second word. Reference to and modification of these fields is specified by the instruction being executed.

2.4.6 Decimal Operand References

A decimal operand is specified by two data references. The first is a pointer to the first byte of the operand; the second provides a 3-bit type and a 5-bit length in the following format:



The type field is encoded as:

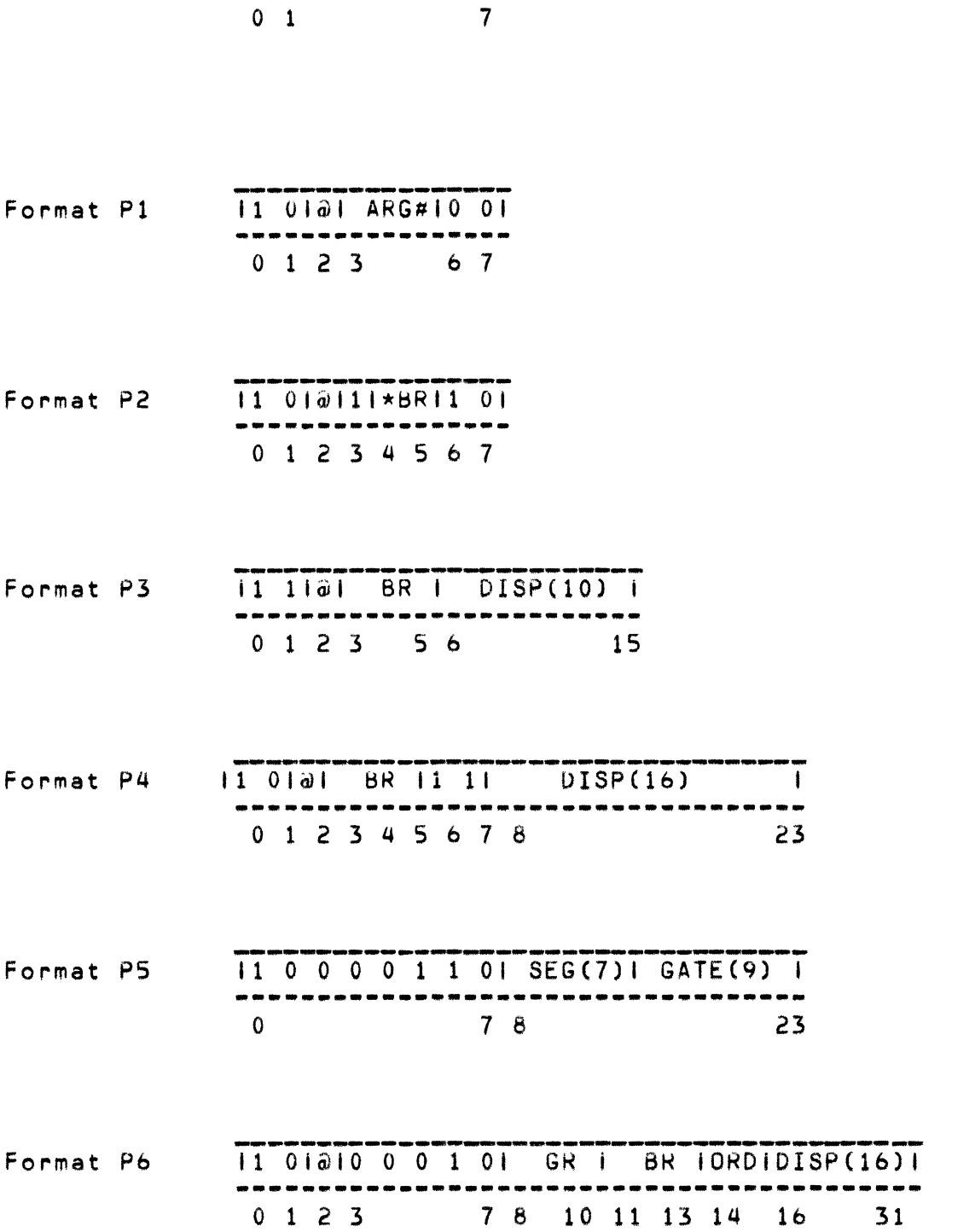
Type	Description
0	Packed decimal (IBM format)
1	Unpacked decimal, low-order sign/overpunch
2	Unpacked decimal, high-order sign/overpunch
3	Unpacked decimal, trailing sign
4	Unpacked decimal, leading sign
5	Unpacked decimal, unsigned
6	Binary integer, signed
7	Binary integer, unsigned

The two data references comprising a decimal operand reference are referred to as a decimal-ref.

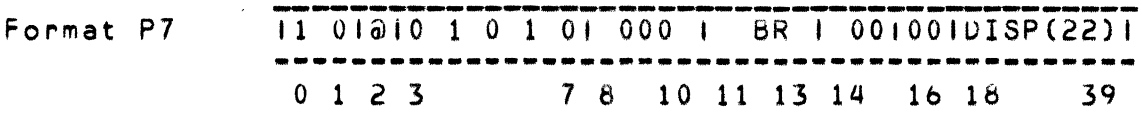
2.5 Procedure Addressing

The following formats are used to generate a procedure address:

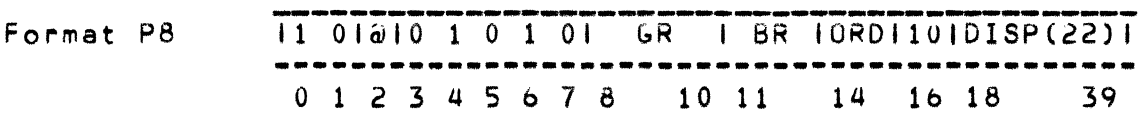
Format P0


0 1 7
Format P1
1 0 @ ARG# 0 0
0 1 2 3 6 7
Format P2
1 0 @ 1 1 *BR 1 0
0 1 2 3 4 5 6 7
Format P3
1 1 @ BR DISP(10)
0 1 2 3 5 6 15
Format P4
1 0 @ BR 1 1 DISP(16)
0 1 2 3 4 5 6 7 8 23
Format P5
1 0 0 0 0 1 1 0 SEG(7) GATE(9)
0 7 8 23
Format P6
1 0 @ 0 0 0 1 0 GR BR IORD DISP(16)
0 1 2 3 7 8 10 11 13 14 16 31

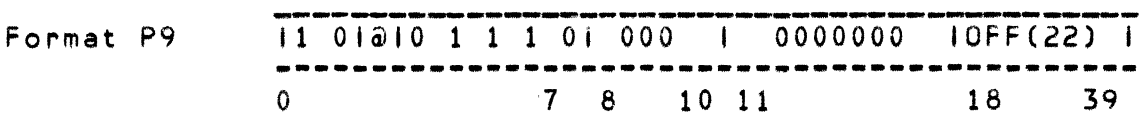
Format P7



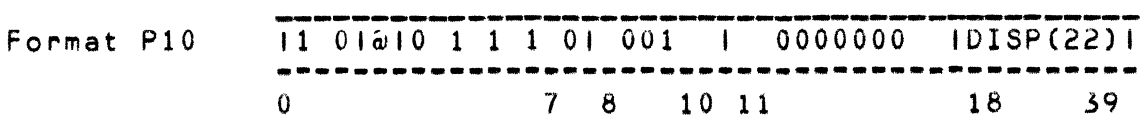
Format P8



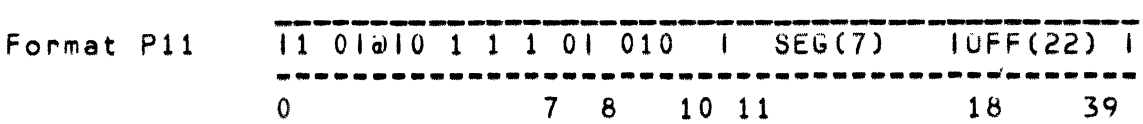
Format P9



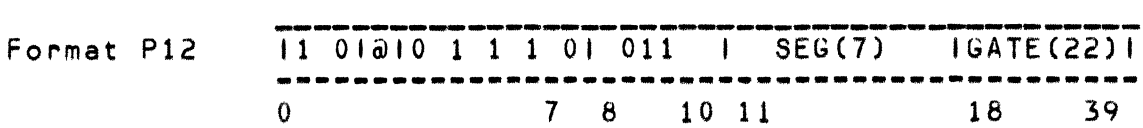
Format P10



Format P11



Format P12



Each procedure address is resolved into an effective byte address consisting of two fields: a 7-bit effective segment, and a 22-bit effective byte offset. This resolution occurs differently

for each format:

Format	Effective Segment	Effective Byte Offset
P0	[PC(3,9)]	[BR0(10,31)] + DISP(7)
P1	[AP(3,9)]	[BR1(10,31)] + (ARG# * 4)
P2	[BR(3,9)]	[BR*(10,31)]
P3	[BR(3,9)]	[BR(10,31)] + DISP(10)
P4	[BR(3,9)]	[BR(10,31)] + DISP(16)
P5	SEG(7)	[GATE(9)]
P6	[BR(3,9)]	[BR(10,31)] + DISP(16) + [GR(10,31)]*(2**ORD)
P7	[BR(3,9)]	[BR(10,31)] + DISP(22)
P8	[BR(3,9)]	[BR(10,31)] + DISP(22) + [GR(10,31)]*(2**ORD)
P9	[PC(3,9)]	OFF(22)
P10	[PC(3,9)]	[PC(10,31) + DISP(22)
P11	SEG(7)	OFF(22)
P12	SEG(7)	[GATE(22)]

where

[BR(X,Y)]	represents bits X through Y of the contents of the specified BR
[PC(X,Y)]	represents bits X through Y of the contents of the program counter (BR0)
[AP(X,Y)]	represents bits X through Y of the contents of the argument pointer (BR2)
[GR(X,Y)]	represents bits X through Y of the contents of the specified GR
DISP(X)	represents the X bits of the specified displacement
SEG(7)	represents the 7 bits of the specified segment field
[GATE(X)]	represents the offset contained in the specified gate entry in the specified segment
ORD	represents the contents of the specified ordinal multiplier
OFF(X)	represents the X bits of the specified offset
+,*	represent twos-complement addition and multiplication respectively

2.5.1 Procedure Indirection

When indirection is specified by a procedure reference, an intermediate procedure address is fetched. An effective byte address is computed as if indirection were not specified, except that any specified indexing (formats P6 and P8) is does not occur. This effective byte address is used to fetch the intermediate procedure address, which has the following format:

The ultimate effective address is obtained from the intermediate address in one of four ways, selected by the MODE field:

MODE	Effective Segment	Effective Byte Offset
0	[PC(3,9)]	FIELD(22) + [GR(10,31)]*(2**ORD)
1	[PC(3,9)]	[PC(10,31)] + FIELD(22) + [GR(10,31)]*(2**ORD)
2	SEG	FIELD(22) + [GR(10,31)]*(2**ORD)
3	SEG	[GATE] + [GR(10,31)]*(2**ORD)

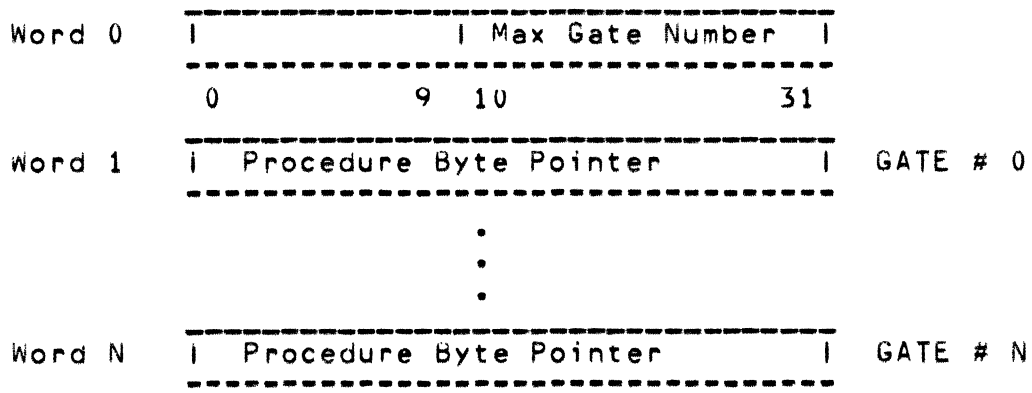
where [GATE] represents the contents of the gate entry specified by FIELD for segment SEG

GR contents are only included in the ultimate effective byte offset computation if the procedure reference specified indexing (formats P6 and P8); this mechanism implements post-indexing for indirect procedure references.

2.5.2 Gates

It is necessary to restrict access to procedure segments that are more privileged than a calling procedure. This is done by allowing control to enter these segments only at specific routine entry points called gates. In this case, the caller, instead of specifying a byte address, specifies a gate number (procedure pointer mode 011). This number is used as an index into a gate array which contains the byte address of the routine to be executed. Gates are numbered starting with 0. The gate array is

located starting at word 0 of the target procedure segment, and has the following format:

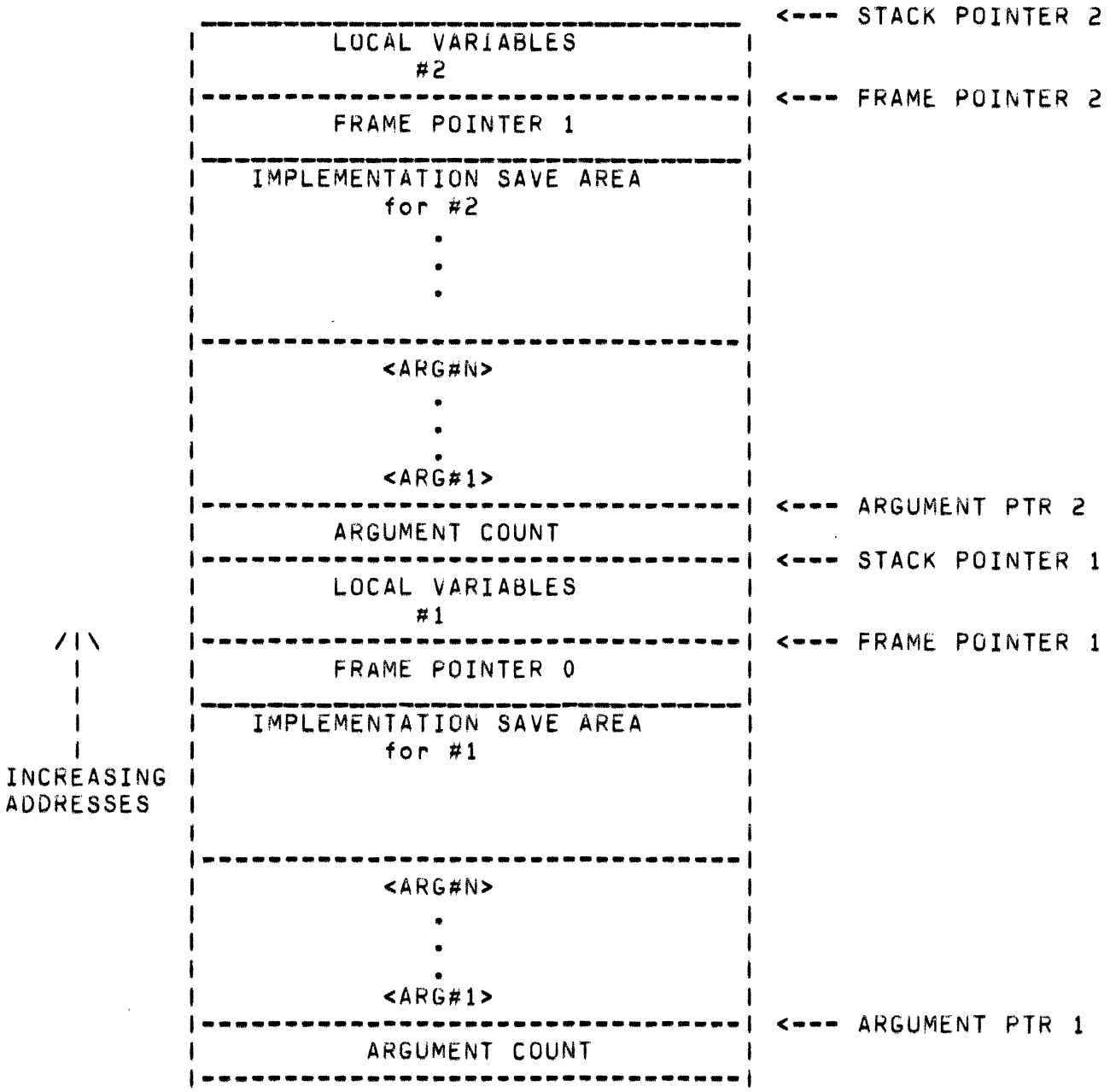


Before the gate entry is fetched, the gate number is compared to the max gate number contained in word 0 of the segment. If the gate number is less than the maximum, the referenced offset is used as the target of the instruction. If it is not within bound, an error condition is signalled. ~~The first 6 words of each procedure segment are reserved for interrupt and trap vectors.~~

Since the contents of a gate ~~entry~~ are interpreted as a Procedure Pointer, a reference to a GATE entry may result in a transfer to another segment.

2.6 Stack Structure

Efficient handling of subroutine call and return, trap processing and space for temporary variables is achieved by support of a stack mechanism. The stack is divided into units called frames. When a subroutine is called or a trap processed, a new frame is created. The structure of the stack at a typical point in time is:



The functioning of the stack is as follows: When a call instruction is issued, an argument packet can be built on the stack. (Alternatively, the argument list can be built in a segment other than the stack segment). Enough information must be saved in the frame's implementation save area to allow a complete restora-

tion of the caller's environment, including all registers. An calling routine uses its implementation save area to save its registers (and other state) on calling another routine. The format in which this information is saved is not specified, with the exception that the caller must save its frame pointer (BR1) in the last location of its implementation save area. In the above figure, routine #1 saves its registers in implementation save area #1 before calling routine #2. If routine #2 calls another routine, it will first save its registers in implementation save area #2. FP and SP are updated to the next available (empty) stack location, and the PC is updated with the starting address of the first instruction to be executed in the called subroutine. Typically, a called subroutine then allocates stack area for local variables with the save instruction.

A return instruction restores a caller's registers (and state) by obtaining them from the caller's implementation save area. In the above figure, when routine #2 executes a return, the original contents of the registers of routine #1 will be restored from implementation save area #1.

When a routine is called, no registers (BRs or XRs) are propagated from caller to callee except GP, (the global pointer) and USR.

Each stack occupies a segment by itself. Thus overflow and underflow are detected by segment boundary faults which (in the case of overflow) can be resolved by the operating system invisibly to the executing procedure.

--End of Chapter--

CHAPTER 3 - Instruction Set

3.1 Introduction

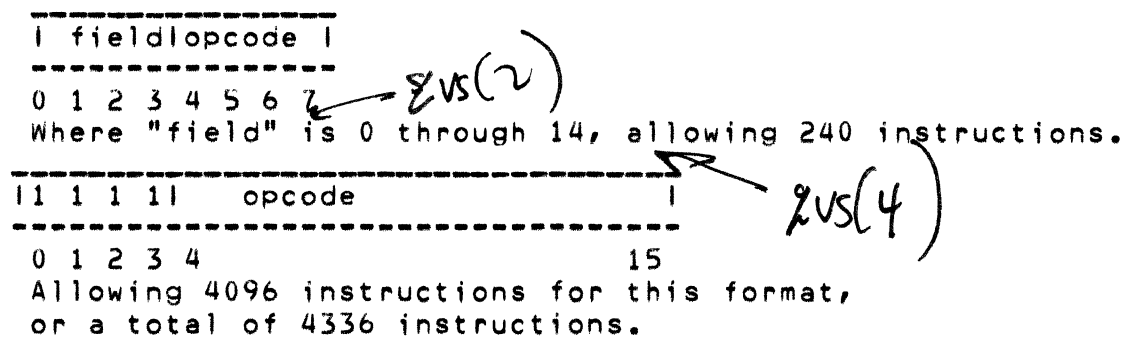
This chapter presents the details of the instruction set for the MTA architecture. The general form of an instruction is:

<op code> {<operand>}*

where operand is a data or procedure reference as described previously.

3.2 Opcode Format

There are two opcode formats - 8 and 16 bits. The encodings are:



3.3 Arithmetic and Related Operations

The table below summarizes the instruction set for seven of the basic data types. Legal operations for a given data type are indicated by an X.

A description of the format and action of each operation follows the table. The operation code for each data type is constructed by appending the type ID found at the top of the column for each data type to the general operation name. For example, <ADD-16> adds two 16 bit fixed point numbers, where <ADD-P> adds two packed decimal numbers.

Operation	Fixed Point (signed) (8,16,32 64 bits)	Floating point (32 and 64 bits)	Packed Decimal
type ID	-8, -16 -32, -64	-S, -D	-P
ADD	X	X	X
SUBTRACT	X	X	X
MULTIPLY	X	X	X
DIVIDE	X	X	
REMAINDER	X		
DIV-R			X
NEGATE	X	X	
SHIFT-A	X	X	X
INCREMENT	X		
DECREMENT	X		
AND	X		
IOR	X		
XOR	X		
COMPLEMENT	X		
MASK-MERGE	X		
SHIFT-L	X		
MOVE	X	X	X
COMPARE	X	X	X
TEST	X	X	X
CLM	X		
CLEAR	X	X	X
NORMALIZE		X	
INTEGERIZE		X	
HALVE		X	
ROUND			X
EDIT			X

Following are the operands and descriptions of each operation. Except for packed decimal operations, each operand is a standard data reference as described in the addressing chapter. Packed decimal operands contain two data references, one specifying the length, and one pointing to the first byte of the decimal string. For more detail, refer to the chapter on addressing forms.

, Certain operations are provided with both implicit and explicit destinations. When the operation has an implicit destination, the result is always placed in the container from which the last operand was extracted. An explicit destination is provided by an additional reference. In operations where this is permitted, the last reference has been placed in parentheses, to indicate that its absence will result in an implicit reference. In opcode terminology, where an operation demands an explicit destination, a "-E" will appear in its suffix.

* <ADD><REF1><REF2>(<REF3>)

* <SUBTRACT><REF1><REF2>(<REF3>)

* <MULTIPLY><REF1><REF2>(<REF3>)

  For all types, multiply returns a result of the same type and size as the inputs.

* <DIVIDE><REF1><REF2>(<REF3>)

  The value specified by <REF2> is divided by the value specified by <REF1> and the result is placed in <REF2>(<REF3>).

* <REMAINDER><REF1><REF2>(<REF3>)

* <DIV-R><REF1><REF2><REF3> - divide with remainder

  The value specified by <REF2> is divided by the value specified by <REF1>. The quotient is placed in <REF2> and the remainder is placed in <REF3>.

* <NEGATE><REF1>(<REF2>)

* <SHIFT-A><REF1><REF2>(<REF3>) - arithmetic shift

  The operand specified by <REF1> is an 8 bit shift count. Positive implies a left shift, negative a right

shift.

* <INCREMENT><REF1>(<REF2>)

* <DECREMENT><REF1>(<REF2>)

* <AND><REF1><REF2>(<REF3>)

* <IOR><REF1><REF2>(<REF3>)

* <XOR><REF1><REF2>(<REF3>)

* <COMPLEMENT><REF1>(<REF2>)

* <MASK-MERGE><REF1><REF2><REF3>(<REF4>)

<REF3> (<REF4>) becomes (<REF1> AND <REF2>) OR (NOT <REF1> AND <REF3>)

* <SHIFT-L><REF1><REF2>(<REF3>) - logical shift

A logical shift of the operand specified by <REF2> is performed. The shift count is an 8 bit quantity specified by <REF1> (positive -> left shift, negative -> right shift).

* <MOVE><REF1><REF2>

* <COMPARE><REF1><REF2>

The only result of this instruction is to set the condition register based on the result of the comparison between the operands specified by the references.

* <TEST><REF1>

The condition register is set based on the result of a comparison between the operand specified by <REF1> and zero.

* <CLM><REF1><REF2>

<REF1> specifies an operand twice the length of the data type specified. The first half is a lower bound and the second half of this operand is an upper bound. The condition register is set based on a comparison between the operand specified by <REF2> and these two signed values.

* <CLEAR><REF1>

The operand specified by <REF1> is set to zero.

* <NORMALIZE><REF1>

* <INTEGERIZE><REF1>

* <HALVE><REF1>

The value specified by <REF1> is divided by 2.0 and returned to the place specified by <REF1>

* <ROUND><REF1><REF2><REF3><REF4><REF5><REF6>

The packed decimal string specified by reference 3, with length specified by reference 2 is shifted(scaled by a factor of 10) as specified by reference 1. The result is moved to the decimal string specified by reference 6, with length specified by reference 5.

A positive count results in a left shift (multiply by 10). A negative count results in a right shift and addition of the rounding factor to the shifted string before the final shift right occurs. That is, after count-1 right shifts, the rounding factor is added to shifted string before the last shift. The last shift results in the rounded shifted string and its carry out being shifted right one additonal time. A count 0 results in a move of reference 2 to reference 6.

The rounding factor is interpreted as a signed byte.

* <EDIT><REF1>...<REFN>

3.4 Strings

The string instructions provided are generally oriented to multi-byte character strings. The compare instructions will set conditions bits in the condition register (CR). Up is defined as an increasing byte address and down as a decreasing byte address. A string length of zero will cause no operation to occur.

Within this section a <STR-REF> is a pointer to a string descriptor as described in the Addressing chapter. <STR-REF>.PTR is the string's byte pointer as specified in the descriptor. <STR-REF>.MAX is the string's maximum length as specified in the descriptor. <STR-REF>.CUR is the string's current length as specified in the descriptor. The string descriptor always points to the first byte of the string. In scans there are condition codes for failure due to current length being zero or negative, character not found and successful scan. If the scan is unsuccessful, the returned index will be set to zero.

The following instructions have been defined:

* <MOVE-STRING><STR-REF1><STR-REF2>

  Move bytes from the string referenced in <STR-REF1> to the string referenced in <STR-REF2> for a count equal to MIN (<STR-REF1>.CUR,<STR-REF2>.MAX). This instruction also updates the value of <STR-REF2>.CUR to the number of bytes moved. <STR-REF2>.MAX is unchanged.

* <MOVE-WITH-FILL><STR-REF1><STR-REF2><REF3>

  Similar to <MOVE-STRING> except that if <STR-REF1>.CUR is less than <STR-REF2>.MAX, the remainder of string two is padded out with the eight bit character specified by <REF3>.

* <COMPARE-STRINGS><STR-REF1><STR-REF2>

  Compare strings referenced by <STR-REF1> and <STR-REF2> setting the condition register (CR).

* <SUBSTRING><STR-REF1><STR-REF2><REF3><REF4>

  Set <STR-REF1> to be a new string descriptor to a substring of the string specified by <STR-REF2> with <REF3> being a 16-bit offset into the string for the start of the substring and <REF4> a 16-bit offset into the string for the end of the substring.

* <SCAN-SUBSTRING-UP><STR-REF1><STR-REF2><REF3>

* <SCAN-SUBSTRING-DOWN><STR-REF1><STR-REF2><REF3>

  Scan a string referenced in <STR-REF2> up or down for the substring referenced in <STR-REF1>.Set <REF3> to be the

index to the leftmost character of the found substring. <REF3> is a signed 16-bit integer.

* <TRANSLATE-STRING><REF1><STR-REF2><STR-REF3>

Move translated bytes using a 256-byte translation table referenced by <REF1> from the string referenced by <STR-REF2> to the string referenced by <STR-REF3> for a count equal to MIN(<STR-REF2>.CUR,<STR-REF3>.MAX). Set <STR-REF3>.CUR accordingly.

* <CHARACTER-SCAN-UNTIL-TRUE><REF1><STR-REF2><REF3>

Scan a string referenced in <STR-REF2> using each byte as an index into a 256-bit table referenced by <REF1> until the indexed bit is on. Set <REF3> to be the 16-bit index to the found byte.

3.5 Bit

The bit instructions fall into three classes of operations; boolean instructions, multi-bit string instructions and bit numerics. Throughout this section, <BIT-REF> is used to represent a bit reference. This reference is described in the chapter on addressing formats, and consists of a pair of data references specifying the base of a bit table, and the offset in that table.

3.5.1 Boolean

The following instruction is indivisible, which means the read/modify/write occurs as one completely contained operation locking out any other asynchronous request until the modification is complete.

* <TEST-AND-SET-BIT><BIT-REF>

Test the bit referenced by <BIT-REF> and set the appropriate condition bits. Set the referenced bit.

The following instructions are not indivisible.

* <TEST-BIT><BIT-REF>

  Test the bit referenced by <BIT-REF> and set the appropriate condition bits.

* <SET-BIT><BIT-REF>

  Set the bit referenced by <BIT-REF>.

* <CLEAR-BIT><BIT-REF>

  Clear the bit referenced by <BIT-REF>.

3.5.2 Multi-bit

* <FIND-LEADING-BIT><REF1><REF2><REF3>

  Scan for first 1 in a 32 bit word specified by <REF1>. Set <REF2> to be the bit offset to this bit.

3.5.3 Bit Numeric

For the bit numeric move operations, an unsigned operation (-U) involves clearing the high order bits of the destination; a signed operation (-S) involves sign extending the most significant bit of the bit field to fill the destination. Movement is always to a 32 bit destination.

* <EXTRACT-U><BIT-REF1><REF2><REF3>

* <EXTRACT-S><BIT-REF1><REF2><REF3>

Move a bit numeric specified by <BIT-REF1> with count contained in the five least significant bits of the byte specified by <REF2> to a 32 bit destination referenced by <REF3>.

* <INSERT-U><REF1><BIT-REF2><REF3>

* <INSERT-S><REF1><BIT-REF2><REF3>

Move a 32 bit source specified by <REF1> to a bit numeric specified by <BIT-REF2> with count in <REF3>.

3.6 Stack Manipulation

The following instructions modify the stack:

* <MODIFY-STACK-POINTER><REF>

Set the value of the stack pointer (SP) to be the current value of SP added to the 16-bit signed integer referenced by <REF>.

* <PUSH-8><REF>

* <PUSH-16><REF>

* <PUSH-32><REF>

* <PUSH-64><REF>

Move one, two, four or eight bytes of data referenced by <REF> To the end of the stack starting at SP. Adjust SP to point to the new end of the stack by adding one, two, four or eight to its current value.

* <POP-8><REF>

* <POP-16><REF>

* <POP-32><REF>

* <POP-64><REF>

Remove the last one, two, four or eight bytes from the end of the stack starting at SP-1 and place them at the reference <REF>. Readjust SP to the new end of stack by subtracting one, two, four or eight from its current value.

* <PUSH-MULTIPLE><REF>

Push multiple registers. <REF> specifies a 16-bit mask used to determine which registers to push. Mask bit 0 represents GR7, bit 1 represents GR6, bit 8 represents BR7, bit 9 represents BR6, etc. If the bit representing a register is set, that register is pushed.

* <POP-MULTIPLE><REF>

Pop multiple registers. <REF> spcifies a 16-bit mask used to determine which registers to pop. The mask is interpreted as in <PUSH-MULTIPLE>.

* <MOVE-TO-SP><REF>

Move the current value of the stack pointer (SP) to the 32 bit operand specified by <REF>.

* <MOVE-FROM-SP><REF>

Move the 32 bit operand specified in <REF> to the stack pointer.

* <RESTORE><REF>

Recover the state from an implementation save area in the current stack. <REF> resolves to a pointer to the frame containing the context to be restored. The stack and frame pointers are set so that the frame specified by <REF> becomes the current frame.

3.7 Transfer of Control

3.7.1 Entry and Exit

In the following instructions, <PREF> refers to a procedure reference as defined in the Addressing chapter.

* <PUSH-PC><PREF>

  Place the PC for the next instruction at the end of the stack starting at SP and branch to <PREF>. This facilitates a quick call to a subroutine which will use the current stack and register environment as its own. SP becomes SP+4.

* <POP-PC>

  Remove the last four bytes from the end of the stack and set the PC to be their value. SP becomes SP-4. This facilitates a quick return from a <PUSH-PC> type call. The next instruction executed (whose address was at the end of stack) will be that following the corresponding <PUSH-PC> instruction.

* <CALL-PACKET><PREF><REF0><REF1>...<REFN>

* <CALL><PREF><REF>

  These call operators branch to a subroutine which uses a new stack environment. <PREF> is the specifier of the address of the subroutine. <REF0> is an 8-bit unsigned integer representing the number of argument references which are to follow. <REF1>...<REFN> are the references to a single argument or a list of n arguments. (N being the value of <REF0>.) The call will build a packet of arguments which can be referenced by the callee using AP. This packet has the following format (PTR is a 32 bit absolute address):

```
                    <#ARG>
AP------->          <PTR1>
                      .
                      .
                    <PTR1>
                    <PTRN>
```

Note: <REF0> is placed one byte before the base register address.

Callers building their own parameter packets use the <CALL> operation. In this case, <REF> is the address of the packet. This address is placed in the callee's AP.

* <RETURN>

Return from procedure or trap handler. The current stack frame is popped and the return information is obtained from the implementation save area.

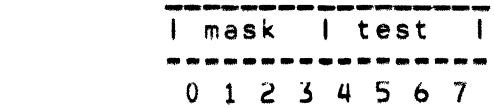
* <RETURN-ABN><PREF><REF2>

Functions like <RETURN> except the address to return to is specified by <PREF>. <REF2> specifies a pointer to the stack frame containing the procedure environment to restore. (This allows a return to a location and a context greater than one procedure level above the current routine.

3.7.2 Vanilla Jumps

* <JUMP-ON-CONDITION><REF1><PREF>

<REF1> specifies an 8-bit field with the following format:



The mask is logically anded with the condition bits in the

USR and the result is compared with the test field. If the two are equal a branch is taken to the address specified by <PREF>. Otherwise, execution continues with the instruction after the JUMP.

The following instructions perform conditional branches based on the settings of the condition code following a <COMPARE>, <TEST>, or <SUBTRACT> instruction. As defined previously, for arithmetic operations CB1 is called the C bit, CB2 is called the N bit, and CB3 is called the Z bit.

* <JUMP-NE><PREF>

Branch to the address specified by <PREF> if the Z bit is one.

* <JUMP-EQ><PREF>

Branch to the address specified by <PREF> if the Z bit is zero.

Arithmetic signed comparisons:

* <JUMP-GT><PREF>

Branch to the address specified by <PREF> if the Z and N bits are zero.

* <JUMP-LT><PREF>

Branch to the address specified by <PREF> if the N bit is one.

* <JUMP-GE><PREF>

Branch to the address specified by <PREF> if the N bit is zero.

* <JUMP-LE><PREF>

Branch to the address specified by <PREF> if the Z bit is one or the N bit is one.

Unsigned comparisons:

* <JUMP-UGT><PREF>

Branch to the address specified by <PREF> if the C and Z bits are zero.

* <JUMP-ULT><PREF>

Branch to the address specified by <PREF> if the C bit is zero.

* <JUMP-UGE><PREF>

Branch to the address specified by <PREF> if the C bit is one.

* <JUMP-ULE><PREF>

Branch to the address specified by <PREF> if the C bit is one or the Z bit is one.

Unconditional branch:

* <JUMP><PREF>

Equivalent to a <JUMP-ON-CONDITION><0><PREF>

3.7.3 Dispatches

All dispatch instructions use a table of the following format:

```
                    <Lower Bound><Upper Bound>
<REF>------->       <Procedure Pointer>
                          .
                          .
                          .
                    <Procedure Pointer>
```

The lower bound and upper bound are both signed 16-bit integers. All dispatches validate the index as lower bound <=index <=upper bound. If the index is not within the bound range, the PC

will be set to the next instruction following the dispatch. Otherwise a branch will be taken through the indexed procedure pointer. In the operation descriptions, <REF0> describes a sixteen bit index, and <REF1> is a pointer to a dispatch table as described above. All dispatch operators will set the condition register to indicate one of three conditions; dispatch index out of range, dispatch index in range but there was no label, or successful dispatch. The dispatch table can have "holes" by setting the value of that position in the table as a 32-bit zero (illegal label within table).

* <DISPATCH><REF0><REF1>

* <DISPATCH-PUSHPC><REF0><REF1>

* <DISPATCH-CALL><REF0><REF1><REF2>

<REF2> is the argument pointer as specified under the description of the <CALL> operation.

3.8 Conversion

* <CONVERT-INTEGER-TO-SP><REF1><REF2>

* <CONVERT-INTEGER-TO-DP><REF1><REF2>

Convert the integer specified by <REF1> to floating point referenced by <REF2>. Conversion of 16-bit integers is to single precision, of 32-bit integers to double precision floating point.

* <CONVERT-SP-TO-INTEGER><REF1><REF2>

* <CONVERT-DP-TO-INTEGER><REF1><REF2>

Convert the floating point number specified by <REF1> to an integer. Conversion is from single precision to 16-bit integer, double precision to 32-bit integer.

* <CONVERT-SP-TO-DP><REF1><REF2>

Convert the single precision number specified by

<REF1> to a double precision number specified by <REF2>.

* <CONVERT-CHARACTER-TO-DP><STR-REF><REF>

Convert the character string to a double precision floating point number.

* <CONVERT-DP-TO-CHARACTER><REF><STR-REF>

Convert a double precision floating point number to a character string.

* <PACK><DEC-REF1><DEC-REF2>

Converts the operand specified by <DEC-REF1> to packed decimal format and places the result in the operand specified by <DEC-REF2>.

* <UNPACK><DEC-REF1><DEC-REF2>

Convert the packed decimal operand specified by <DEC-REF1> to the format and location specified by <DEC-REF2>.

3.9 Loop Control

* <LOOP-8><REF1><REF2><REF3><PREF>

* <LOOP-16><REF1><REF2><REF3><PREF>

* <LOOP-32><REF1><REF2><REF3><PREF>

* <LOOP-64><REF1><REF2><REF3><PREF>

These instructions are used to control iterative loops such as FORTRAN DO statements. <REF1>, <REF2> and <REF3> are fixed point numbers with a container size specified by the instruction suffix. The action of the instruction is:

```
REF1 <- REF1 + REF2
IF (REF2 >= 0 AND REF1 =< REF3) OR
(REF2 < 0 AND REF1 >= REF3)
THEN PC <- PREF
ELSE PC <- PC + 1
```

3.10 Semaphores

The following instructions are used for synchronization of coroutines, etc:

* <LOCK><REF1>

  IF REF1 = 0 THEN CB0 <- 1 ELSE CB0 <- 0; IF REF1 # 0 THEN REF1 <- REF1 + 1;

* <UNLOCK><REF1>

  IF REF1 = 0 THEN CB1 <- 1 ELSE DO CB1 <- 0; REF1 <- REF1 - 1; IF REF1 = 0 THEN CB0 <- 1 ELSE CB0 <- 0; END;

3.11 Reserved Instructions

There is a set of 256 op codes reserved for definition on a per system basis. Execution of any of these instructions causes a process trap (see Interrupts and Traps Chapter) to a software or microcode routine which then executes the instruction.

The format of the specific instructions is determined by the programmer or microcoder who writes the emulator routine. These instructions will typically be used by system programmers for operating system or compiler specific accelerators, and for entry to user written microcode routines.

3.12 System control

* <CONVERT-TO-PHYSICAL><REF1><REF2>

  Generate the logical address defined by <REF1>. Convert this to the corresponding physical address and return this address in the 32 bit operand specified by

<REF2>. If there is no corresponding physical address (i.e. the specified page is not resident) return zero.

* <GET-MACHINE-ID><REF1>

Return machine specific information in the 64 bit area specified by <REF1>. The exact format of this information is to be defined.

* <MOVE-TO-ISR><REF1>

Load the Interrupt Status Register (ISR) from the 32 bit operand specified by <REF1>.

* <MOVE-FROM-ISR><REF1>

Place the current contents of the ISR into the 32 bit operand specified by <REF1>. Unused bits are set to zero.

* <WAIT>

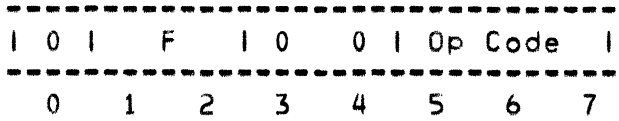
Places the processor in a wait state, from which it can still handle interrupts. (Similar to a JUMP . except that it does not tie up the memory bus.)

* <HALT>

Halts the processor by activating a solenoid which yanks the AC cord out of the rear of the cabinet.

3.13 Input/Output

I/O devices are controlled and monitored by means of 8-bit I/O directives (IODs) having the following format:



The op code field specifies the specific I/O operation to be performed, and the F field specifies control information. A more

detailed treatment of IODs can be found in the I/O System chapter.

An IOD is constructed and emitted by the following instruction:

* <GIOD><REF1><REF2><REF3>

  The operand specified by <REF1> is the eight bit I/O directive. <REF2> specifies an eight bit field which contains the device code of the controller to which the IOD is addressed. The operand specified by <REF3> is a 16-bit field used either to supply data, receive data, or receive status, as specified by the IOD.

3.14 Miscellaneous

* <LOAD-EFFECTIVE-ADDRESS><REF1><REF2>

  Move the effective address of <REF1> to the 32 bit operand specified by <REF2>.

* <COPY><REF1><REF2><REF3>

  Move bytes from the area specified by <REF1> to the area specified by <REF2>. The number of bytes to move is contained in the 16 bit fixed point operand specified by <REF3>. If the move count is positive, bytes will be moved left to right. If it is negative, bytes will be moved right to left.

* <FILL><REF1><REF2><REF3>

  Copy the byte specified by <REF1> starting at the byte specified by <REF2>. The number of copies to make is contained in the 16 bit operand specified by <REF3>.

* <MOVE-TO-USR><REF1>

  The User Status Register (USR) is set from the 16 bit operand specified by <REF1>.

* <MOVE-FROM-USR><REF1>

The current contents of the USR are stored in the sixteen bit operand specified by <REF1>. Unused bits are set to zero.

* <GET-OPERAND><REF1><REF2>

Used to obtain the operands of an instruction that is to be emulated by the software. The exact format and operation is to be defined.

* <EXCHANGE><REF1><REF2><REF3>

<REF1> specifies an eight bit operand containing the number of bytes to exchange between the areas specified by <REF2> and <REF3>. The bytes are exchanged left to right, one byte at a time.

* <EXECUTE><REF1>

Execute the instruction at the address specified by <REF1>. Unless the executed instruction causes a transfer of control, instruction execution continues with the instruction following the EXECUTE.

* <NO-OP>

* <USER-TRAP-x>

"x" is a number between 1 and 256, which provides 256 instructions which are available for user definition. These instructions cause a procedure trap to be taken through a unique gate with an argument containing the value of "x".

--End of Chapter--

CHAPTER 4 - Interrupts and Traps

4.1 General

All events in an MTA machine which require a change in the normal flow of control are handled using a trap mechanism. Traps are divided into three categories - procedure, process, and interrupt. Procedure traps are events which can be handled by a user procedure. These include all instruction exceptional conditions such as fixed and floating point overflow, etc. Process traps are procedure caused events which need system intervention in order to be resolved. These include page faults, page table faults, protection faults, etc. Interrupts are asynchronous events which must be resolved by the operating system, including I/O interrupts, power failure, etc.

All traps appear to the trap handlers like procedure calls. This is done by generating a parameter packet containing arguments and then pushing a state block on a stack. Each trap within a group is assigned a unique value which is passed as the argument to the trap handler. Thus the trap handler can detect the type of trap by accessing the argument and, optionally, dispatch to a unique type handler based on the argument. In addition, all traps are dismissed merely by executing a return instruction, which will continue execution at the point where the trap was taken. This value passing forces only one trap to be generated on each machine cycle, even in a pipelined implementation.

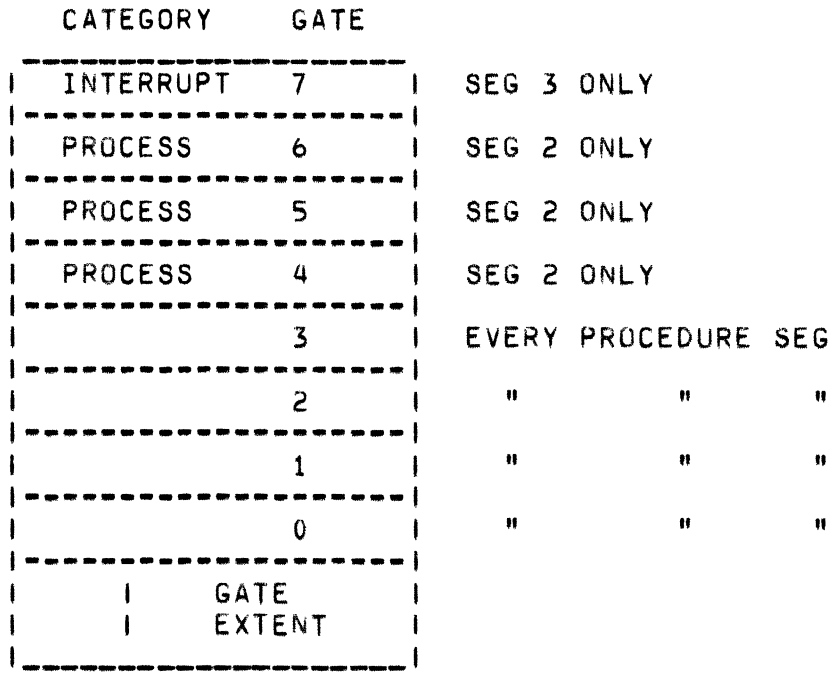
Since traps can be taken at different points in the execution of an instruction, differing amounts of information must be saved in order to continue execution after dismissing the trap. Thus, the state block must be self describing to the extent that the return instruction can determine how to restore from it.

In order to respond to the process and interrupt categories of traps, architectureally defined procedure segements exist. Segment #2 is always assigned to respond to process traps. Segement #3 is always assigned to handle interrupts.

Every procedure segment has 1, 2, or 3 groups of trap pointers. These groups are for the procedure traps, process traps,

and interrupts. Procedure segments that can only handle procedure traps have only the procedure trap pointers. The procedure segment that can handle process traps has the procedure and trap pointers. The procedure segment that handles interrupts has all three pointer groups. Thus, a procedure trap in the procedure segment for the interrupt handler uses the same relative gate entry in its gate array to vector to the procedure trap handler.

The structure of the trap pointers in the procedure segment root is as follows:



The contents of each gate entry contains a procedure pointer (Ref. Addrressing Chapter). Thus, a trap may vector to a handler in the present procedure segment, or specify an entry point in some other procedure segment. The final target address becomes the new value of the PC. Transfer of control to a procedure segment must be cognizant of the predefined gate entries. Thus, user defined gates must be adjusted to reflect the segment called and the fixed number of gate entries.

In essence, a trap results in the implicit execution of a CALL instruction. The target address is a gate in the present segment, the arguments passed are a function of the ultimate ring of execution. As with an explicit CALL, sufficient state information is saved on the state to _____to the point following the

procedure invocation, including the number of arguments pass. Thus, executing a RETURN instruction is used to exit from a trap handler. The point of execution return for the arguments passed to the trap handler are delineated for every trap type.

5 Procedure Traps

Procedure traps are broken down into 4 categories:

	GATE NUMBER
Floating Point Arithmetics	0
Fixed Point Arithmetics	0
Commercial Arithmetic	1
Trace	2
User	3

When a trap is detected, it is classified as one into one of these areas. The target address is the gate number listed in the present procedure segment. The arguments pushed into the stack are:

Further delineation of the trap type (e.g. float exponent overflow, as differentiated from underflow
The value of the PC that points to the instruction causing the trap.

The state block pushed into the stack has a return address pointing to the instruction following the one causing the trap.

5.1 Floating/Fixed Point

The following floating point error conditions are detected:

Exponent overflow
Exponent underflow
Divide by zero

For these classes of traps, the bit field of 32 bits is passed as an argument. Exponent overflow is indicated by 00...1. Expon-

ent underflow by 00...10. Divide byu zero by 00...100.

When exponent overflow occurs, the value returned to the specified destination is_____. When exponent underflow occurs, the value return to the specified destination is_____. When divide by zero occurs, the specified destination remains unchanged.

The following fixed point conditions are detected:

Integer overflow on ADD/SUB
Divide by zero
Integer overflow on MPY
Float to fixed conversion

The classes of traps the bit field passed as an argumment is:

Integer overflow on ADD SUB	00 ...	1000.
Divide by zero	00 ...	10000.
Integer overflow on MPY	00 ...	100000.
Conversion	00 ...	1000000.

5.2 Commercial String

The following commercial and string traps are detected:

Size on for numberic ADD, SUB, MULTIPLY, DIVIDE
Divide by zero
String overflow

For these classes of traps, the bit field produced is:

Size error	0001
Divide by zero	0010
String overflow	0100

5.3 Trace

TBDL

5.4 User Traps

When bits 0-7 of the opcode are 1111 1111 (all 1's), a user define opcode is trap enabled. Gate #3 is transferred through. The argument passed to the trap handler is the second byte of the opcode (Bit 8-15).

6 Process Traps

Process traps are classified into three areas and thus three gate locations (4, 5, 6). A fault is a condition which is a result of specified resource not being in the proper state. The fault handler can optionally supply this resource and successfully restart the instruction. The three types of process traps are:

FAULTS
UNIMPLEMENTED OPCODE
DATA GENERAL RESERVED

The processing of a process trap is handled slightly different than procedure traps. The gate array access is not in the present procedure segment but in segment #2. Segment #2 can only be executed in Ring0. Thus, all arguments are pushed on the Ring 0 stack with a subsequent change of the ring of execution to ring 0.

6.1 Faults

The following faults are defined:

Page
Access violation
Specification exception
Segment bounds check on stack operation
Segment bounds check on non-stack operation
Gate bound check

For page faults, the arguments pushed are:

Address of the PTE
The logical address causing the fault
A field indicating a page fault 00...011
The state of the machine at the time of the fault

For access violations, the arguments pushed are:

A bit field indicating the type of access violation:

00 10	READ
00 100	WRITE
00 1000	EXECUTE
00 10000	CALL

The value of the PC referencing the instruction causing the fault.
The logical address of the target reference that caused the fault.
Machine state at the time of the machine.

For specification exception, the bit field pushed on the stack is:

List possible specification exception
e.g. Write into literal
Write into BR using non-pointer instruction

For segment bounds check on stack operation, the argument are:

A bit field of xxxxx.
The value of the PC referencing the instruction cau the fault.
The logical address of the target.

For segment bounds check on non-stack operation, the arguments pushed are:

A bit field of xxxxx.
The value of the PC referencing the instruction cau the fault.
The logical address of the target.

For gate bounds check, the arguments pushed are:

A bit field of xxxx.
The value of the PC referencing the instruction cau the fault.
The value of the procedure pointer referencing the array.

6.2 Unimplement Opcode

An unimplemented opcode is defined to exist when the control sequences necessary to map the algorithm specified by the opcode on the hardware are absent. To maintain deject code compatibility, these control sequences are implemented in software. When such an opcode is detected, the argument passed to the trap handler is the value of the program counter referencing the instruction. The return address set up by the trap handler references the instruction following the unimplemented opcode.

6.3 Data General Reserved

When bits 0-7 of the opcode are 1111 1110, a reserved Data General trap is enabled. The arguments passed is the second byte of the opcode (bit 8-15).

7 Interrupts

The processing of an interrupt is handled in the following manner. Gate #7 located in segment #3 is used to vector to the interrupt handler. The state of the current process is pushed on its ring 0 stack. The new stack used in segment #4. The new ring of execution is ring 0. The interrupt stack defined is not process specific. If an interrupt occurs while the interrupt stack is being used, the current stack is pushed on the interrupt stack. The state of the process includes ISR.

When an interrupt is processed, the following actions occur:

The present stack is pushed as described above.

The code of the device causing the interrupt is obtained. This code is used to dispatch to a 64 bit entry. The base these tables is a physical address maintained in the interr vector table.
This entry contains a new ISR and a procedure pointer.

The contents of the new ISR are used as follows:

The interrupt mask contained in the ISR is inclusive "ORed" the present interrupt mask. Interrupts are enabled or disa by bit___ of the mask. If the enable bit is 0, interrupts enabled. Is the enable bit is 1, interrupts are disabled. The procedure pointer is evalued and becomes the new value the Program Counter. The interrupt handler is entered at t new value.

--End of Chapter--

CHAPTER 5 - Protection

5.1 General

Segments are the basic unit of protection. Segments are always referenced within a hierarchical domain structure organized into units called rings. There are 8 rings of protection. Ring 0 contains the system security kernel and is the least restricted. Ring 7 is a user domain and is the most restricted. At all times, there is a current ring of execution (CRE), which determines the access allowed to the current procedure.

There are four types of access which can be allowed to a segment. Two are related to data access. Read access allows data within the segment to be fetched. Write access allows modification of data within the segment. The other two apply to procedure transfer. Direct access allows control to be passed to any location within the segment. Gate access allows transfer to the segment only through use of a gate (described in the Introduction).

Whenever access is attempted to a segment, the processor generates an effective ring number (see Ring Maximization), and uses that and the target segment number as indices into a two dimensional access array. This array is associated with the current translation table (see Memory Management) and each entry in it contains a bit for each of the four types of access. If the bit is set, that type of access is allowed from the effective ring to the target segment.

5.2 Ring Maximization

In any hierarchical system, there exists a problem of a higher ring passing as a parameter to a lower ring a pointer to a segment that the higher ring has no access to. To avoid this problem, the architecture provides a technique called ring maximization, which is applied to all data accesses. Every base register and byte data pointer involved in an effective address calculation has a ring

number contained in it. The effective ring used for access checking is the maximum of all these rings and the current ring of execution. In this way, a more privileged ring can make data accesses with the same access limitations as the higher ring on whose behalf it is executing, but a higher ring can not masquerade as a lower (more privileged) ring.

5.3 Determination of the Current Ring of Execution

Every procedure segment has associated with it the mimimum (MINRE) and the maximum (MAXRE) ring in which the procedure is allowed to execute. These are kept in the segment descriptor. Whenever the procedure segment is changed as a result of a call, jump, or return instruction, a new current ring of execution is determined according to the following formula:

CRE <- MAX { MIN(MAXRE,CRE) , MINRE }

5.4 Stacks

Every ring has its own stack segment with a format as described in the Introduction. When a ring crossing is detected during execution of a call instruction, the stack segment number for the new ring is fetched from the Task Control Block. Arguments and the procedure state block are pushed onto the new stack segment.

5.5 BASE REGISTER MODIFICATIONS

Base Registers can only be the destination of instructions which produce pointers as part of the result. Thus the Program Counter can only be altered by instructions which are allowed to change the sequence of execution (e.g., JUMP, RETURN, DISPATCH). BR's 1-7 can only be altered by instructions whose result is a pointer (e.g., Load Effective Address. and Move Stack Pointer).

5.6 PROGRAM COUNTER RELATIVE

A Read or Write access to the present procedure segment is

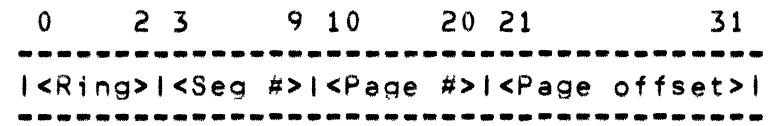
treated as a reference to a data segment. Thus, the normal rules governing access apply. The only exception to this rule is that an operand reference that is specified as a literal is read from the instruction stream ignoring the read access of the present procedure segment.

--End of Chapter--

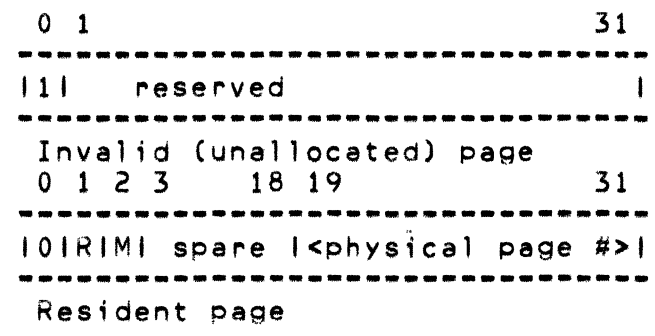
CHAPTER 6 - Memory Management

Since the state of the art in memory management policies for virtual systems continues to advance, it would seem reasonable to encapsulate MTA's memory management algorithms in a module whose internals are not architecturally specified. Thus, the following description of memory management for MTA implementation 1 does not in principal belong in this document; it is provided solely for completeness.

For purposes of memory management, the logical address described in 1.2.2 is further subdivided such that each segment consists of 2K pages, each page containing 2K bytes:



Conversion from logical address to physical address is implemented by constructing a page table for each segment. This table contains one entry for every page in the segment (entries exist for pages beyond the current length of the segment, but are marked invalid). An entry in this table (PTE - for Page Table Entry) has one of the following two formats, depending on the associated page's status:



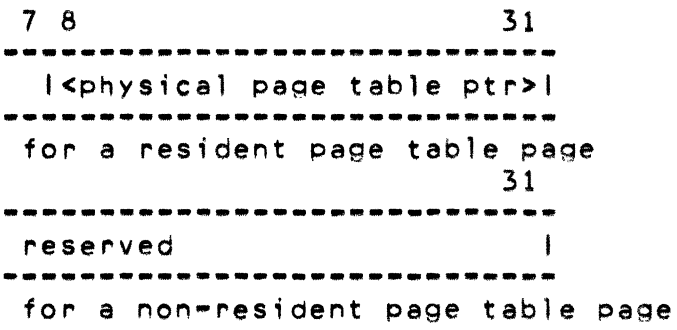
In the PTE for a resident page, the R-bit indicates whether the page has been referenced by a process since the last time the R-bit was reset, and the M-bit indicates whether the page has been modified by a process since the last time the M-bit was reset.

These two bits are required by most useful memory management algorithms. The 16 spare bits in the resident page PTE are available to the memory manager - a typical use might be the W-bit required by the page fault frequency algorithm to mark pages belonging to the active process' working set.

The page table associated with each segment is itself 4 pages in length:

2K pages/segment * 1 PTE/page * 4 bytes/PTE * 1 page/2K bytes = 4

Since we anticipate that most segments will be less than one fourth their maximum length, it is desirable to require only those page table pages containing PTE's for allocated pages of an active segment to be resident in primary memory. This is achieved by associating 4 page table pointers (PTPs) with each of the 128 segments of a process' logical address space. A PTP has one of the following two formats, depending on the status of the associated page table page:



The physical address contained in a PTP for a resident page is a byte pointer to the page table page itself (the low order 11 bits of this pointer are always 0, since page table pages must be aligned on physical page boundaries).

The 512 PTPs associated with a process' 128 segments are grouped in sequence to form the process' translation table - this table defines its process' logical address space. The translation table for the currently active process is pointed at by the current translation table pointer (CTTP), itself a physical address. Naturally, the translation table for the currently active process is resident in primary memory.

Each logical address emitted by the processor is translated to

a physical address by adding bits 3 through 11 of the logical address to CTTP to select a PTP from the current translation table. If bit 0 of this PTP is reset, a boundary fault is initiated; otherwise, bits 12 through 20 of the logical address are added to the pointer in the PTP to select a PTE from the page table. If this PTE is invalid, a boundary fault is initiated. If it is resident, the PTE is updated as required by the memory management algorithm (for example, the R-bit may be set), and the desired physical address is constructed by concatenating bits 19 through 31 of the PTE with bits 21 through 31 of the logical address.

Although this mechanism provides the desired functionality, it is painfully slow, since two memory references are required (one to get the PTP, one to get the PTE). Therefore, MTA implementation 1 will be provided with an associative address translation unit (ATU) which, when provided with bits 3 through 20 of a logical address, either produces the associated PTE (R, M, and physical page number), or initiates an ATU fault. This fault is serviced (in microcode) by obtaining the PTE as described above, loading its contents into the ATU (perhaps overwriting some other PTE in the ATU), and retrying the translation. The ATU also faults if a page whose M-bit is reset is modified. This fault is handled by setting the M-bit of the page's PTE and ATU entry.

Because logical addresses are process specific, the ATU must be purged before a new process is activated. Thus each page a process references in its time slice is guaranteed to generate at least one ATU fault.

In order to support memory management as defined above, the memory management module must be provided with the following:

1. The ability to move operands in primary memory using physical addresses.
2. The ability to load a PTE into the ATU.
3. The ability to purge the ATU.

--End of Chapter--

CHAPTER 7 - I/O System

An MTA processor controls and monitors its I/O devices via I/O directives (IODs), which are identical in functionality to the programmed I/O instructions of the Nova/Eclipse architecture. Upon execution of a GIOD instruction, the processor constructs an IOD and transmits it over the programmed-I/O bus to the specified I/O device's controller. Depending on the IOD contents, this device controller may accept two bytes of data from the processor, transmit two bytes of data to the processor, accept control information from the processor, or transmit its status to the processor.

An MTA processor's programmed-I/O bus is both functionally and physically identical to the Nova/Eclipse I/O bus, as specified in Data General publication 015-000031 (Interface Designer's Reference for the Nova and Eclipse Computers) chapter 3, chapter 4, and appendix D. Any I/O device controller adhering to these specifications can be attached to the I/O bus of any MTA processor. An exception is those controllers which utilize the Increment or Add-to-memory data channel functions; these may not be available on some or all MTA implementations. Extensions to the I/O bus may also be supported by particular implementations, as long as compatibility is not jeopardized. Data parity, control parity, and 8-bit device codes are examples of potential extensions.

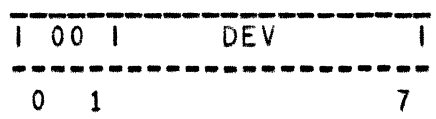
In addition to carrying IODs and data, the I/O bus permits device controllers to asynchronously request processor interrupts and data channel service. The bus contains signals for interrupt priority arbitration (INTA) and priority mask maintenance (MSKO). Unlike the Nova/Eclipse architecture, however, these signals are not explicitly activated by programmed I/O instructions. Instead the processor automatically uses INTA to determine the highest priority device when an interrupt is synchronized and vectors to that device's interrupt handler. The interrupt vector routine installs a new interrupt mask in the ISR, using MSKO to transfer this mask to the I/O controllers (see the chapter entitled "Traps and Interrupts" for a more detailed description of I/O interrupts and the vector routine). In general, any alteration of the interrupt mask field in the ISR will implicitly activate MSKO.

The MTA architecture defines a programmed-I/O instruction with

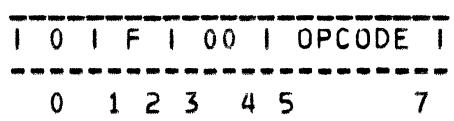
the following format:

GIOD <REF1><REF2><REF3>

where the operand specified by <REF3> is the source destination, the operand specified by <REF2> provides a device code, and the operand specified by <REF1> provides the I/O directive (IOD). The device code operand has the following format:



The 6-bit DEV field identifies the device controller to which the IOD is addressed. The IOD has the following format:



Device code 000000 is reserved for the processor power monitor, and device code 111111 is reserved for processor use.

An IOD specifies two transactions: a data transaction, followed by a control transaction. The OPCODE field controls the data transaction with the following encoding:

OPCODE	Data Transaction
000	none
001	A-register -> <REF3>
010	<REF3> -> A-register
011	B-register -> <REF3>
100	<REF3> -> B-register
101	C-register -> <REF3>
110	<REF3> -> C-register
111	status -> <REF3>

Data transactions 001 through 110 involve three 16-bit virtual device registers associated with the addressed device controller; data is transferred between the 16-bit operand specified by <REF3> and the selected virtual device register over the 16 DATA lines of the programmed-I/O bus. Data transaction 111 is a device status transaction, with status bits BUSY and DONE transferred into the least significant two bits of the 16-bit operand specified by <REF3>; the most significant 14 bits of this operand are undefined. BUSY and DONE are also set into condition code bits CC0 and CC1. BUSY and DONE are transmitted on dedicated lines (SELB and SELD) of the programmed-I/O bus.

The F field controls the control transaction with the following encoding:

F	Control Transaction
00	none
01	START
10	CLEAR
11	PULSE

A controller's reaction to a control transaction is device specific unless the data transaction OPCODE was 111 (status transfer), in which case the control transaction is ignored.

--End of Chapter--

CHAPTER 8 - Availability/Reliability/Maintainability

8.1 Overview

This chapter at present contains theoretical directions which we expect Data General and MTA to be taking. This material is indicative of the techniques we will employ, but is preliminary as an architectural definition.

8.2 MTA Diagnostic Control Processor Objectives

Data General Corporation, and its customers, are becoming increasingly aware that the characteristics of maintainability and availability are vital to future systems sales. To meet the availability and maintainability goals, a soft console will provide all required MTA console functions executed through a teletype interface. In addition, it will improve system maintainability by proviiding a software diagnostic capability external to the MTA processor system and independent of its correct operation. Availability can be enhanced by providing downline system control and the capacity to monitor timing on critical system data paths. Other capabilities that can enhance marketability can be provided nearly free given the above.

The first two objectives effectively define the basic form that the console will take. To interrupt console commands received through the teletype, interface intelligence is required. A software diagnostic capability independent of a working processor system also requires intelligence, plus memory capacity - both RAM and bulk storage. To provide the intelligence needed, a microNOVA will be present on the console board, with a teletype interface and an interface to the MTA System. Basic control software for the console is present in ROM storage, and RAM is present for data and additional console program storage. Bulk diagnostic software is provided by a diskette unit connected to the microNOVA I/O bus via an external cable. The interface to the MTA System allows the microNOVA to force the processor to any microstate, as well as forcing data onto buses, and examining the data on those buses. No part of the actual processor need be working, except the power supplies for the console microNOVA, to perform complete

diagnostics. The microNOVA will also be provided with its own set of self-diagnostic programs, further improving maintainability.

Enhancements to availability are accomplished by:

* Providing a capability for downline control. This is done merely by connecting the teletype interface mentioned to a modem instead of a terminal. Console commands normally received directly from a terminal are then received via a phone line. This allows remote diagnosis of the system before a field engineer arrives at the site.

* Providing the capability to monitor timing on critical system data paths. This does not reduce failure probability but allows imminent failures to be located before they occur by spotting symptoms indicating failure, such as late bits.

* Providing the ability to continuously monitor the power supply. This feature will enable early warning of power supply irregularities, and avoid catastrophic failures or critical data loss.

* Providing the ability to run diagnostic programs from the console at specified hardware breakpoints. This eases software debugging and allows the checking of specified hardware registers or paths in the middle of certain routines.

The diagnosis of intermittent hardware failures is generally difficult. Classically, the simplest way to locate such an intermittent failure is to vary the system characteristics until the failure becomes hard. The diagnostic control processor can facilitate this debugging of intermittent failures by permitting us to vary three key parameters:

* voltage

* temperature

* clock frequency

Within certain rigid limits, these parameters may be varied, under the control of the microNOVA, as an aid to off-line failure analysis.

--End of Chapter--

CHAPTER 9 - Measurement and Debug Aids

--End of Chapter--

APPENDIX I - INDEX

10:57:58
22/Mar
Rev