```
+4444444444444444444444444444444444444444444444444444444444444444444444444444444+

                                  $$$$$  $$$$   $$$
                                  $      $      $  $
                                  $      $      $  $
                                  $$$$   $      $  $
                                  $      $  $$  $  $
                                  $      $   $  $  $
                                  $$$$$  $$$$   $$$

                           USER=EGO  QUEUE=LPT  DEVICE=@LPA
                SEQ=151  QPRI=127  LPP=63  CPL=80  COPIES=1  LIMIT=153

                             CREATED:  31-AUG-77  11:16:42
                            ENQUEUED:  21-DEC-77  13:38:30
                            PRINTING:  21-DEC-77  13:38:34

                               PATH=:UDD:EGO:DOCUMENT.LS

     $$$$    $$$    $$$   $   $  $   $  $$$$$  $   $  $$$$$        $      $$$
     $   $  $   $  $   $  $   $  $$ $$  $      $$  $    $          $     $   $
     $   $  $   $  $      $   $  $ $ $  $      $ $ $    $          $     $
     $   $  $   $  $      $   $  $ $ $  $$$$   $  $$    $          $      $$$
     $   $  $   $  $      $   $  $   $  $      $   $    $          $         $
     $   $  $   $  $   $  $   $  $   $  $      $   $    $     $$   $     $   $
     $$$$    $$$    $$$    $$$   $   $  $$$$$  $   $    $     $$   $$$$$  $$$

+4444444444444444444444444444444444444444444444444444444444444444444444444444444+


                                  AOS XLPT REV 01.00
```

CHAPTER 1 - Introduction 1-1

1.1 Address Structure 1-1
1.2 Available Registers 1-1
1.3 Data Address Formation 1-3
1.3.1 General Addressing 1-3
1.3.2 Register Addressing 1-4
1.3.3 Local Variable Addressing 1-4
1.3.4 Argument Addressing 1-5
1.3.5 Data Indirection 1-5
1.3.6 Character String Addressing 1-5
1.3.7 Bit Addressing 1-6
1.4 Procedure Addressing 1-7
1.4.1 Procedure Indirection 1-7
1.4.2 Gate Array 1-8
1.5 Stack Structure 1-9
1.6 Opcode Format 1-11

CHAPTER 2 - Data Types and Formats 2-1

2.1 Basic Allowable Types 2-1
2.1.1 Floating Point 2-1
2.1.2 Fixed 2-1
2.1.3 Logical 2-1
2.1.4 Unsigned 2-1
2.1.5 Character 2-2
2.1.6 Commercial 2-2
2.2 Rounding 2-2

CHAPTER 3 - Instruction Set 3-1

3.1 Introduction 3-1
3.2 Signed Fixed Point 3-1
3.2.1 16 Bit Fixed Point 3-1
3.2.1.1 <REF>/<REF> 3-2
3.2.1.2 <REF>/<IMMED> 3-3
3.2.2 32 bit Fixed Point 3-4
3.2.2.1 <REF>/<REF> 3-4
3.2.2.2 <REF>/<IMMED> 3-5
3.3 Unsigned 3-6
3.3.1 Unsigned 8 Bits 3-6
3.3.1.1 <REF>/<REF> 3-6
3.3.1.2 <REF>/<IMMED> 3-7
3.3.2 Unsigned 32 Bits 3-9
3.3.2.1 <REF>/<REF> 3-9
3.3.2.2 <REF>/<IMMED> 3-10
3.4 Floating Point 3-11
3.4.1 Single Precision Floating Point 3-12
3.4.2 Double Precision Floating Point 3-14
3.5 Character 3-15
3.6 Bit 3-18
3.6.1 Single Bits 3-18
3.6.2 Multi-bit 3-19
3.6.3 Bit Numeric 3-19
3.7 Commercial 3-20
3.8 Stack Manipulation 3-22
3.9 Jumps 3-23
3.9.1 Entry and Exit 3-23
3.9.2 Vanilla Jumps 3-25

3.9.3 Dispatches 3-25
3.10 Conversion 3-27
3.11 Reserved Instructions 3-28
3.12 System control 3-28
3.13 Input/Output 3-28
3.14 Miscellaneous 3-28

CHAPTER 4 - Interrupts and Traps 4-1

4.1 General 4-1
4.2 Procedure Traps 4-2
4.3 Process Traps 4-2
4.4 System Traps 4-2

CHAPTER 5 - Protection 5-1

5.1 General 5-1
5.2 Ring Maximization 5-1
5.3 Determination of the Current Ring of Execution . . 5-2
5.4 Stacks 5-2

CHAPTER 6 - Memory Management 6-1

CHAPTER 7 - I/O System 7-1

7.1 Organization 7-1
7.2 Objectives for the EGO-1 I/O system 7-3

CHAPTER 8 - Availability/Reliability/Maintainability . . 8-1

8.1 Overview 8-1
8.2 EGO Diagnostic Control Processor Objectives . . . 8-1

CHAPTER 9 - Measurement and Debug Aids 9-1

This is not the end.
It is not even the beginning of the end.
But it is the end of the beginning.

Winston Churchill

Data General Corporation
Company Confidential

PREFACE

Data General, to accomodate its present customer's growth requirements and expand its sales base, must develop a medium scale architecture for near term use. This architecture will alleviate the logical address space limitation evidenced in the Eclipse line, and will provide a contemporary architectural foundation for a new line of small and medium scale systems.

Architectural Objectives.

1. This medium range architecture will allow for implementations from low cost silicon to high performance multi-unit processors.

2. It will have a large logical address space - somewhere in the range from 8 megabytes (2**23) to 4 billion bytes (2**32).

3. The design will be extensible for future enhancement both in the instruction set and the architectural organization.

4. There will be upward and downward compatibility at the object code level. This will enable us to provide a single code generator and run time library for each language, and to provide program transportability in a network environment.

5. The basic orientation of the machine is for user programming in high level languages (COBOL, FORTRAN, RPG, etc.)

6. SPL will be an integral part of the machine environment. All software should be implementable in SPL.

7. There will be no architectural limit on file storage capacities.

Initial Implementation Objectives.

1. The machine will be released to manufacturing engineering in 14 months and be deliverable within 20 months of project startup. This implies a straightforward implementation.

2. Through use of good engineering practice, the implementation will strive for intrinsic reliability to provide good

availibility and low MTTR. Self diagnostic capabilities will be provided.

3. Risk will be minimized by utilization of existing circuit design (E/500 FPU), packaging schemes (E/250), and mature technologies (TTL) where possible.

4. New circuit designs

	Max Boards	Max Cost
CPU (including cache)	4	$
Address Translation Unit	1	
Console Controller (w/Micro-Nova)	1	
SC Memory Controller (1 per 2 Mb)	1	
256K Byte SC Memory w/ERCC	1	

Existing circuit designs requiring modification:

High Speed Channel
E/500 FPU

5. Eclipse compatibility will be provided at the user level by a processor mode, thus enabling per process selection of Eclipse emulation. Performance in this mode will be maximized subject to the overall time constraints on the project. (Target improvement is 25% faster than the E/500). Emulation of the E/500 map is under investigation. This capability, which would provide operating system transportability from the Eclipse, will be included if possible. It is anticipated that the Eclipse compatibility will not be included in later implementations.

6. The I/O bus will be compatible with the Nova and Eclipse I/O bus, data channel bus, and high speed channels.

7. The I/O bandwidth will have the same magnitude as the memory bandwidth.

8. The data paths for fixed point arithmetic will be 32 bits wide.

9. Hardware features (especially accelerators) that are applicable to each market segment will be modularized so that the machine can be economically configured for various functions.

10. As much software as possible will be written in SPL. The only constraint on this objective will be the availibitiy of the SPL compiler and debugger.

Relationship To Present Activities.

1. The product will provide a graceful upward growth path from the Eclipse E/500 series in terms of immediate performance improvements and long term conversion to the faster, higher capability native mode.

2. Although the initial implementation will provide a high end for Data General's product line, it is anticipated that future implementations will be less expensive, and thus, when the FHP arrives, this architecture will provide a high level language compatible lower end for the product line.

Data General Corporation
Company Confidential

CHAPTER 1 - Introduction

1.1 Address Structure

The EGO architecture supports a process wide, two dimensional address space of 512 million bytes. This total space is divided into 128 segments, each containing up to 4 million bytes. A segment can contain either procedure or data.

The basic addressing granularity is to the byte. 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).

1.2 Available Registers

The processor contains the following classes of registers available to the programmer for use with the standard instruction set:

* Base Registers (BR) - The 8 base registers are 32 bits wide and contain a byte pointer. Bits 0-2 of this pointer represent a ring number, bits 3-9 a segment number, and bits 10-31 a byte offset. Base register 0 is the Program Counter (PC), BR1 the Frame Pointer (FP) , and BR2 the argument Linkage Pointer (LP) . Base register 0, the PC, can only be modified as a result of a branch type instruction. In all other cases, an attempted modification of BR0 is inhibited and signalled as an error condition.

* Index Registers (XR) - Eight 32-bit index registers are provided.

* Accumulators (AC) - There are eight 32-bit accumulators for use in fixed or floating point operations. 32-bit signed or unsigned fixed point numbers or single precision floating point numbers can be moved to the AC's directly. 16-bit signed fixed point values are sign extended to 32-bits on a move to an AC. 8-bit unsigned values are zero

extended to 32-bits on a move to an AC. Double precision floating point values may only be moved to an even-odd pair of AC's providing four double precision floating point accumulators numbered zero two, four and six. Character data types and commercial data types may not be moved to an AC. Bit data types may only be moved to an AC using the bit numeric type moves on a bit field of no more than 32 bits.

The preceding registers are maintained on a per procedure basis. When a new procedure is called, the registers are saved, and their initial values in the new procedure are indeterminate. On a return, the old values of the registers are restored.

In addition to the registers, there is a process wide control register called the Procedure Status Register (PSR), which can only be modified using privileged instructions. This register contains:

A condition register (CR) which describes the condition of the results of all operations performed within the ALU.

Rounding mode bits to define the type of rounding to be performed at the end of floating point arithmetic operations.

Trace bits to define procedure tracing to be performed.

Emulator mode bits defining the instruction set currently being executed.

Procedure trap inhibit bit.

System trap inhibit bit.

Privileged instruction enable bit.

The PSR is saved through traps, but is not saved on a procedure call. When a new procedure segment is invoked, certain of these bits are automatically set from values in the segment descriptor.

1.3 Data Address Formation

An EGU operand reference is self describing and falls into one of four categories: general, register, local variable, and argument. Each of these categories permits indirect addressing, specified when the "@" bit is set. (except index and accumulator register specification in the register address category)

1.3.1 General Addressing

The following formats are used for general address generation:

BR Byte Relative (signed disp)

```
-------------------------------
|1|0|@|  BR |  XR |DISP (7) |
-------------------------------
 0 1 2 3 4 5 6 7 8 9       15
```

BR Byte Relative (signed disp)

```
----------------------------------
|1|1|0|  BR |  XR |@|DISP (14)   |
----------------------------------
 0 1 2 3 4 5 6 7 8 9 10        23
```

BR Byte Relative (positive disp)

```
-----------------------------------------
|1|1|1|  BR |  XR |@|DISP (22)          |
-----------------------------------------
 0 1 2 3 4 5 6 7 8 9 10               31
```

Each of these formats contains a base register field (BR), an index register field (XR), and a displacement field. An effective address is constructed by first summing the offset from the specified BR, the low order 23 bits of the XR and the displacement field sign extended to 23 bits (in the 32 bit form, the displacement field is zero extended to 23 bits). If the XR field is zero, no index register is used in the computation. If the result is greater than 2**22, a segment overflow trap is generated. Otherwise, the result is concatenated with the segment number from the BR to form the effective address. In all cases of byte addressing, bits 0-8 of the specified XR are ignored during the address generation cycle. When PC relative addressing is specified (BR0), the value of the PC used is the address of the first opcode byte.

1.3.2 Register Addressing

The BRs, XRs, and accumulators (ACs) are addressed using the following formats:

```
Base Register          -----------------
                       |0|0|0|1|1|@|  BR |
                       -----------------
                        0 1 2 3 4 5 6 7

Index Register         -----------------
                       |0|0|0|0|0|0|  XR |
                       -----------------
                        0 1 2 3 4 5 6 7

Accumulator            -----------------
                       |0|0|0|0|0|1|  AC |
                       -----------------
                        0 1 2 3 4 5 6 7
```

1.3.3 Local Variable Addressing

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

```
FP Word Positive       -----------------
                       |0|1|@|   FP+     |
                       -----------------
                        0 1 2 3 4 5 6 7
```

The FP+ field is interpreted as a word offset relative to BR1, the frame pointer. Thus, the effective address is formed by shifting the FP+ field left two bits (forming a byte displacement) and adding it to the frame pointer offset.

1.3.4 Argument Addressing

Arguments passed to a subroutine can be addressed using the following abbreviated format:

```
                      ----------------
Linkage Pointer       |0|0|1|1|@|  ARG  |
    Word Positive     ----------------
                       0 1 2 3 4      7
```

The ARG field is interpreted as a word offset relative to BR2, the linkage pointer. Thus, the effective address is formed by shifting the ARG field left two bits (forming a byte displacement) and adding it to the linkage pointer offset.

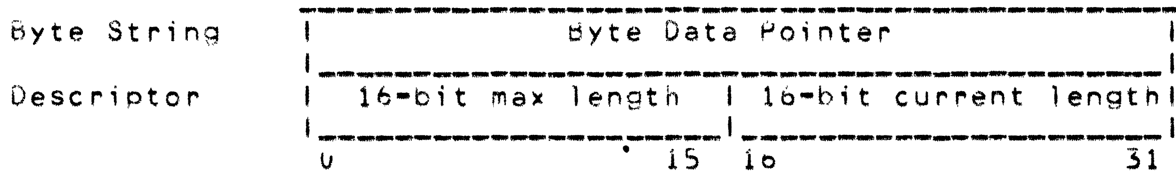
1.3.5 Data Indirection

When indirection is specified, the effective address points to a byte data pointer in memory, used to address the desired operand. The format of that pointer is:

```
              ----------------------------------------
Byte Data     |<Ring>|    <Seg #>    |<Seg. Offset>     |
Pointer       ----------------------------------------
                0    2  3             9 10             31
```

1.3.6 Character String Addressing

To facilitate general purpose string operations, a string addressing descriptor has been defined which contains all necessary information about a character string. The descriptor has the following format:

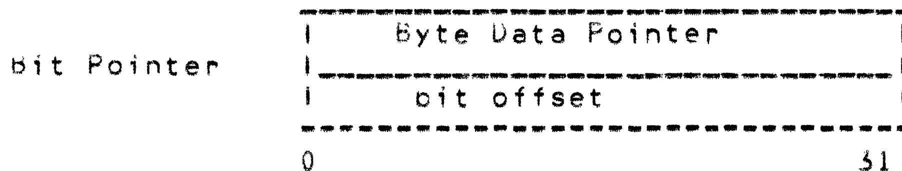


1.3.7 Bit Addressing

It is necessary to build some form of bit addressability upon a native byte addressable structure. The form this support takes is invisible to the address portion of the memory system. The underlying addressing mechanism within the processing unit performs the necessary transformation between bit and byte and the necessary extraction of a bit aligned field from the byte aligned operand.

A bit address is produced in one of two manners: with or without indirection. When indirection is not specified, the contents of the displacement field and index register (if indexing is specified) are added together to form a bit offset relative to the byte pointer contained in the specified base register.

If indirection is specified, the displacement and index register are interpreted as byte offsets as in a regular data address. The byte address generated points to a descriptor with the following format:



The first word is a byte data pointer. The second word is a bit offset relative to the byte specified in the paired pointer. the bit offset relative to the byte pointed to by this pointer.

1.4 Procedure Addressing

The following formats are used to generate a procedure address:

```
PC byte Relative         -----------------
                         |0|   OFFSET    |
                         -----------------
                          0 1 2 3 4 5 6 7

BR Byte Relative         ------------------------------
   (signed disp)         |1|0|@|  BR |  XR |DISP (7) |
                         ------------------------------
                          0 1 2 3 4 5 6 7 8 9        15

BR Byte Relative         ---------------------------------------
   (signed disp)         |1|1|0|  BR |  XR |@|DISP (14)     |
                         ---------------------------------------
                          0 1 2 3 4 5 6 7 8 9 10        23

BR Byte Relative         --------------------------------------------------------
   (positive disp)       |1|1|1|  BR |  XR |@|DISP (22)                         |
                         --------------------------------------------------------
                          0 1 2 3 4 5 6 7 8 9 10                         31
```

All forms except the 8 bit form are equivalent to a data address formation. The evaluation of displacement fields, base register, and indexing are the same. The 8 bit form has an implied base of BR0, the PC, and has a signed byte offset relative to that base. The value of the PC used is the address of the first opcode byte.

1.4.1 Procedure Indirection

When indirection is specified, a procedure pointer is fetched from memory. This pointer is used to address the target of the instruction. The format of a procedure pointer reached by indirection is:

Mode bits (0-2) define the format of the pointer. The following encodings have been defined:

```
000 - Present Segment, Absolute Offset
001 - Present Segment, PC Relative Offset
010 - &lt;SEGMENT #&gt;, Absolute Offset
011 - &lt;SEGMENT #&gt;, Gate #
100
 .  - Reserved
 .
111
```

Procedure pointers allow for inter and intra segment transfers.

1.4.2 Gate Array

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 8 of the target procedure segment, and has the following format:

11:3:37
31/Aug/77
Rev. 1

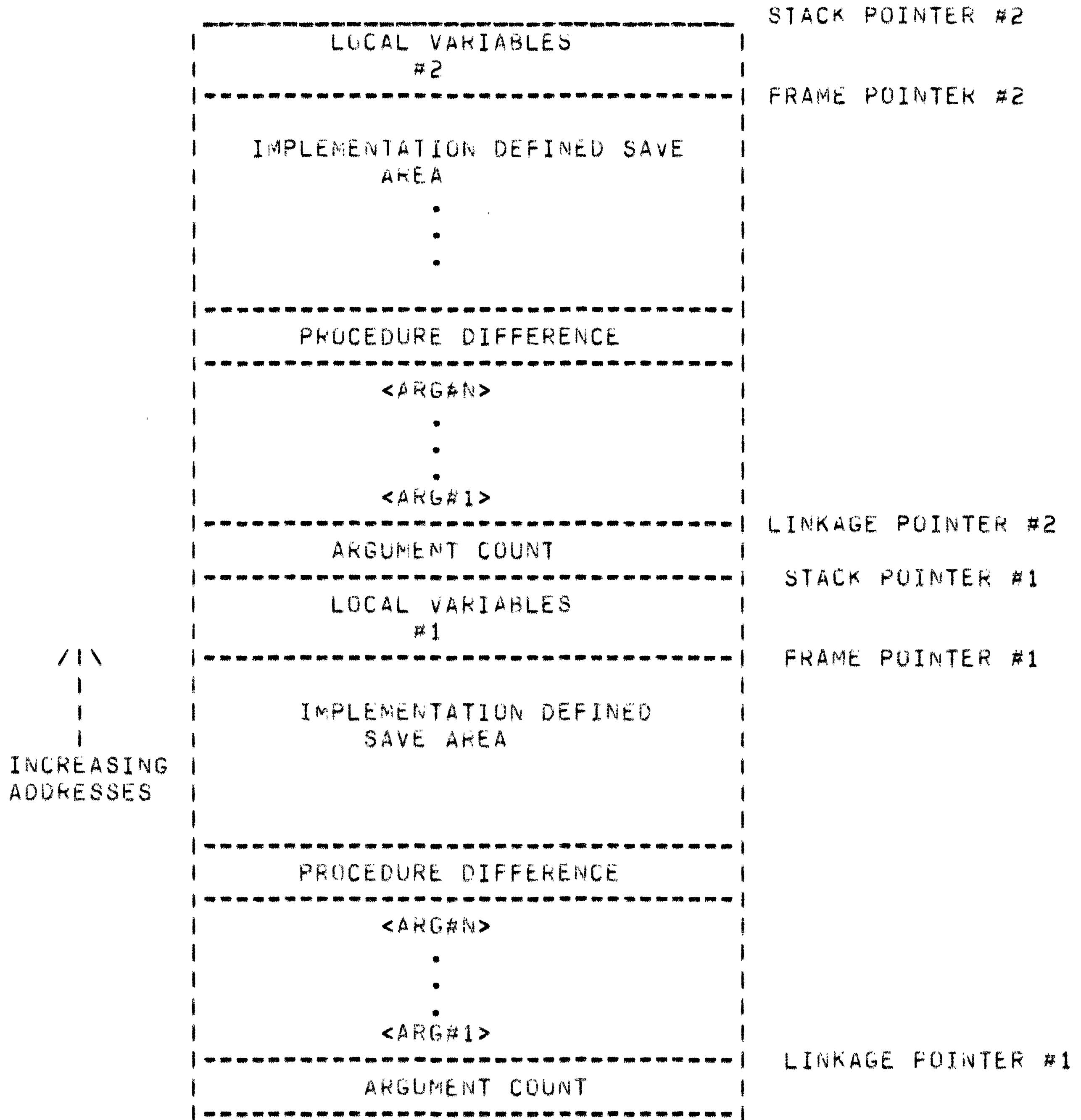
```
----------------------------------
|            | Absolute Offset  |    Gate # N
----------------------------------
 0          9  10               31
               .
               .
               .
----------------------------------
|            | Absolute Offset  |    GATE # 0
----------------------------------
 0          9  10               31


----------------------------------
|            | Max Gate Number  |
----------------------------------
 0          9  10               31
```

Before the gate entry is fetched, the gate number is compared to the max gate number contained in word 8 of the segment. If within this bound, the referenced offset is used as the target of the instruction. If it is not within bound, an error condition is signalled. The first 8 words of each procedure segment are reserved for interrupt and trap vectors.

1.5 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). The present values of the stack pointer, the program counter (BR0), the frame pointer (BR1), and the linkage pointer (BR2), are saved on the stack. 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 restores the stack to its previous state. The old values of PC, FP, LP, SP and all registers are restored to their value prior to the call.

When a subroutine is called, the values of all registers are not propagated across the call.

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.

1.6 Opcode Format

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

```
 ---------------
 | field| Code |
 ---------------
 0 1 2 3 4 5 6 7
```

where "field" is 0 through 14, defining 240 instructions.

```
-----------------------------------
|1 1 1 1|    Code                 |
-----------------------------------
 0 1 2 3 4                      15
```

Defining 4096 instructions for this format, or a total of 4336 instructions.

--End of Chapter--

CHAPTER 2 - Data Types and Formats

2.1 Basic Allowable Types

Throughout this discussion, a single precision word will be considered to have 32 bits. A sixteen-bit entity will be termed a "half word". Eight-bits constitute a byte or character. This section reviews the types supported by the architecture.

2.1.1 Floating Point

Real numbers will be represented in standard Data General (and IBM) format. Both single precision and double precision will be suported.

Variable length formats such as those used in PL/1 will not be precluded, but direct support will not be available.

2.1.2 Fixed

Fixed point numbers are supported in 2's complement integer representation. Direct support for half word and single precision is provided.

2.1.3 Logical

Logical values occupy a one bit container and have the value zero or one. Bit testing is specified within the architecture. 8 or 32 bit unsigned values treated as logical are considered a string of one bit logicals.

2.1.4 Unsigned

As of this writing, Rick Miller is still intent on playing out his option.

In addition, 32-bit and 8-bit unsigned quantities are supported.

2.1.5 Character

Provision is made for character (8-bit) and character string manipulation. This is distinct from commercial string types. When reference is made to character operations and data, the ASCII representation is used except as specifically noted.

2.1.6 Commercial

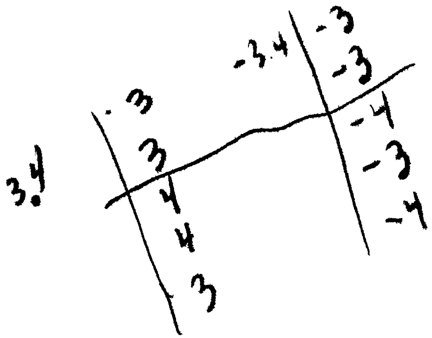
The architecture provides direct support for the COBOL data types. In support of ANSI '74 COBOL standard X3.23, we will provide 18 digits of precision. Numeric display types that are supported are unsigned, trailing sign, leading sign, trailing overpunch, and leading overpunch. Support is also provided for packed decimal, signed and unsigned binary byte strings.

2.2 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.

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--

Data General Corporation
Company Confidential

CHAPTER 3 - Instruction Set

3.1 Introduction

This chapter presents the details of the instruction set for the EGO processor architecture. No op-code assignments have been made as of this writing, but they will appear in subsequent versions of this document. The general form of an instruction is:

```
<op code> {<operand> ... <operand>}
```

where operand is a data or procedure reference as described previously, or an immediate value.

3.2 Signed Fixed Point

There is a complete instruction set to directly manipulate 2's complement fixed point integer operands with 16 and 32 bits of precision.

The possible exception conditions during fixed point arithmetic are: overflow and divide by zero. Potentially every fixed point operation alters the condition register.

3.2.1 16 Bit Fixed Point

3.2.1.1 <REF>/<REF>

* <ADD-16> <REF1> <REF2>
  Add the contents of <REF1> to the contents of <REF2> and move the results to <REF2>.

  CONDITION CODE: The N and Z bits are updated to reflect the results.

* <SUBTRACT-16> <REF1> <REF2>
  Subtract the contents of <REF1> from the contents of <REF2> and move the results to <REF2>.

  CONDITION CODE: The N and Z bits are updated to reflect the results.

* <RSUB-16> <REF1> <REF2>
  Subtract the contents of <REF2> from the contents of <REF1> and move the results to <REF2>.

  CONDITUN CODE: The N and Z bits are updated to reflect the results.

* <MULTIPLY-16> <REF1> <REF2>
  Multiply the contents of <REF1> by the contents of <REF2> and move the least significant 16N and Z bits are updated to reflect the results.
  E: The

* <DIVIDE 16> <REF1> <REF2> and the 16 bits quotient moved to <REF2>. Thlne remainder is not maintained.2

  CONDITION CODE: The N and Z bits are updated to reflect the results.
  bits before the divide is initiated.

* <REMAIN-16> <REF1> <REF2>
  The contents of <REF2> are divided by the contents <REF1> . The 16 bit remainder is moved to <REF2> . The sign of the remainder is the sign of the dividend.
  CONDITION CODE: The N and Z bits are updated to reflect the results.

* <MOVE-16> <REF1> <REF2>

* <COMPARE-16> <REF1> <REF2>
  Only sets condition register.

* <COMPARE-WITHIN-LIMITS-16> <REF1> <REF2>
  <REF2> is a reference to a 32-bit entity. The first 16 bits represent a signed lower limit; the next 16 bits represent a signed upper limit.

* <SHIFT-ARITHMETIC-16> <REF1> <REF2> <REF1> is both source and destination for a 16-bit shift. <REF2> is a pointer to an 8-bit signed shift counter; + = left, - = right.

* <ABSOLUTE-VALUE-16> <REF1> <REF2>

* <NEGATE-16> <REF1> <REF2>

3.2.1.2 <REF>/<IMMED>

The following instructions specify a 8 bit immediate as reference #1. This immediate is sign extended to 16 bits before the operation proceeds.

* <ADD-I-16> <IMMED> <REF2>

* <SUBTRACT-I-16> <IMMED> <REF2>

* <MULTIPLY-I-16> <IMMED> <REF2>

* <DIVIDE-I-16> <IMMED> <REF2>

* <REMAIN-I-16> <IMMED> <REF2>

* <MOVE-I-16> <IMMED> <REF2>

* <COMPARE-I-16> <IMMED> <REF2>

* <COMPARE-WITHIN-LIMITS-I-16> <REF> <IMMED> <IMMED>
The first 16-bit immediate represents a signed lower limit; the next 16-bit immediate represents a signed upper limit.

* <SHIFT-ARITHMETIC-I-16> <REF> <IMMED> <REF> is both source and destination for a 16-bit shift. <IMMED> is an 8-bit signed shift counter; + = left, - = right.

As an optimization, the following instructions have an implied constant of 0 or 1:

* <COMPARE-16-0> <REF> An implied constant of 0.

* <INCREMENT-16> <REF> An implied constant of 1.

* <DECREMENT-16> <REF> An implied constant of 1.

* <CLEAR-16> <REF>
Move zero's.

3.2.2 32 Bit Fixed Point

3.2.2.1 <REF>/<REF>

For every fixed point operation with 16 bits of precision, there exists an equivalent operation for 32 bits.

* <ADD-32> <REF1> <REF2>

* <SUBTRACT-32> <REF1> <REF2>

* <MULTIPLY-32> <REF1> <REF2>
Produces a 32-bit result.

* <DIVIDE-32> <REF1> <REF2>
The dividend is 32 bits.

* <MOVE-32> <REF1> <REF2>

* <COMPARE-32> <REF1> <REF2>

* <REMAIN-32> <REF1> <REF2>
  The dividend is 64 bits. If the dividend is an AC, it must be a double precision even/odd AC pair.

* <ABSOLUTE-VALUE-32> <REF1> <REF2>

* <NEGATE-32> <REF1> <REF2>

* <COMPARE-WITHIN-LIMITS-32> <REF1> <REF2>
  <REF2> is a reference to a 64-bit entity. The first 32 bits represent a signed lower limit; the next 32 bits represent a signed upper limit.

* <SHIFT-ARITHMETIC-32> <REF1> <REF2>

  <REF1> is both source and destination for a 32-bit shift. <REF2> is a pointer to an 8-bit signed shift counter; + = left, - = right.

3.2.2.2 <REF>/<IMMED>

The following instructions specify an 8 bit constant as reference #1. This constant is sign extended to 32 bits before the operation proceeds.

* <ADD-I-32> <IMMED> <REF2>

* <SUBTRACT-I-32> <IMMED> <REF2>

* <MULTIPLY-I-32> <IMMED> <REF2>
  Produces a 32 bit result.

* <DIVIDE-I-32> <IMMED> <REF2>
  The dividend is 32 bits.

* <REMAIN-I-32> <IMMED> <REF2>

* <MOVE-I-32> <IMMED> <REF2>

* <COMPARE-I-32> <IMMED> <REF2>

* <COMPARE-WITHIN-LIMITS-I-32> <REF> <IMMED> <IMMED>
  The first 32-bit constant represents a signed lower limit; the next 32-bit constant represents a signed upper limit.

* <SHIFT-ARITHMETIC-I-32> <REF> <IMMED>

  <REF> is both source and destination for a 32-bit shift. <IMMED> is an 8-bit signed shift counter; + = left, - = right.

  As an optimization, the following instructions have an implied constant of 0 or 1:

* <COMPARE-32-0> <REF>

* <INCREMENT-32> <REF>

* <DECREMENT-32> <REF>

* <CLEAR-32> <REF>

3.3 Unsigned

Unsigned operands contain values that are always positive or zero. EGO supports two unsigned precision, 8 and 32 bits.

3.3.1 Unsigned 8 Bits

The possible exception conditions are: overflow and divide by zero. Alterations, if any, to the condition are specified for each instruction.

3.3.1.1 <REF>/<REF>

* <ADD-8> <REF1> <REF2>

* <SUBTRACT-8> <REF1> <REF2>

* <MULTIPLY-8> <REF1> <REF2>

  Produces a result with 8 bits of precision.

* <DIVIDE-8> <REF1> <REF2>

* <REMAIN-8> <REF1> <REF2>

* <MOVE-8> <REF1> <REF2>

* <COMPARE-8> <REF1> <REF2>

* <COMPARE-WITHIN-LIMITS-8> <REF1> <REF2>
  <REF2> is a reference to a 16-bit entity. The first 8 bits represent a lower limit; the next 8 bits represent an upper limit.

* <SHIFT-LOGICAL-8> <REF1> <REF2> <REF1> is both source and destination for an 8-bit shift. <REF2> is an 8-bit signed shift counter; + = left, - = right.

* <AND-8> <REF1> <REF2>

* <IOR-8> <REF1> <REF2>

* <XOR-8> <REF1> <REF2>

* <SET-DIFF-8> <REF1> <REF2>

  <REF2> becomes <REF1> AND NOT <REF2>.

* <COMPLEMENT-8> <REF1> <REF2>

* <MASK-MERGE-8> <REF1> <REF2> <REF3>
  <REF2> = (<REF1> AND <REF3>) OR (<REF2> AND NOT <REF3>).

3.3.1.2 <REF>/<IMMED>

The following instructions specify an 8 bit constant as

reference #1. This constant is an unsigned 8 bit operand:

* <ADD-I-8> <IMMED> <REF2>

* <SUBTRACT-I-8> <IMMED> <REF2>

* <MULTIPLY-I-8> <IMMED> <REF2>

* <DIVIDE-I-8> <IMMED> <REF2>

* <MOVE-I-8> <IMMED> <REF2>

* <REMAIN-I-8> <IMMED> <REF2>

* <COMPARE-I-8> <IMMED> <REF2>

* <SHIFT-LOGICAL-I-8> <IMMED> <REF2>
  <REF2> is both source and destination for an 8-bit shift. <IMMED> is an 8-bit signed shift counter; + = left, - = right.

* <COMPARE-WITHIN-LIMITS-I-8> <REF> <IMMED> <IMMED>
  The first 8-bit constant represents an unsigned lower limit; the next 8-bit constant represents an unsigned upper limit.

* <AND-I-8> <IMMED-8> <REF2>

* <IOR-I-8> <IMMED-8> <REF2>

* <XOR-I-8> <IMMED-8> <REF2>

* <SET-DIFF-I-8> <IMMED-8> <REF2>
  <REF2> becomes <IMMED-8> AND NOT <REF2>.

* <MASK-MERGE-I-8> <REF1> <REF2> <IMMED>
  <REF2> = (<REF1> AND <IMMED>) OR (<REF2> AND NOT <IMMED>).

As an optimization, the following instructions have an implied constant of 0 or 1:

* <COMPARE-8-0> <REF>
  Compare to 0.

* <INCREMENT-8> <REF>
  Add 1.

* <DECREMENT-8> <REF>
  Subtract 1.

* <CLEAR-8> <REF>
  Move zero's.

3.3.2 Unsigned 32 Bits

3.3.2.1 <REF>/<REF>

* <ADD-U-32> <REF1> <REF2>

* <SUBTRACT-U-32> <REF1> <REF2>

* <MULTIPLY-U-32> <REF1> <REF2>

  Produces a result with 32 bits of precision.

* <DIVIDE-U-32> REF1> <REF2>

* <REMAIN-U-32> <REF1> <REF2>

* <MOVE-U-32> <REF1> <REF2>

* <COMPARE-U-32> <REF1> <REF2>

* <COMPARE-WITHIN-LIMITS-U-32> <REF1> <REF2>
  <REF2> is a reference to a 64-bit entity. The first 32 bits represent a lower limit; the next 32 bits represent an upper limit.

* <SHIFT-LOGICAL-32> <REF1> <REF2>

  <REF1> is both source and destination for a 32-bit shift. <REF2> is an 8-bit signed shift counter; + = left, - = right.

* <AND-32> <REF1> <REF2>

* <IOR-32> <REF1> <REF2>

* <XOR-32> <REF1> <REF2>

* <SET-DIFF-32> <REF1> <REF2>

  <REF2> becomes <REF1> AND NOT <REF2>.

* <COMPLEMENT-32> <REF1> <REF2>

* <MASK-MERGE-32> <REF1> <REF2> <REF3>
  <REF2> = (<REF1> AND <REF3>) OR (<REF2> AND NOT <REF3>).

3.3.2.2 <REF>/<IMMED>

The following instructions specify an 8-bit constant as reference #1. This constant is zero extended to a 32 bit operand:

* <ADD-U-I-32> <IMMED> <REF2>

* <SUBTRACT-U-I-32> <IMMED> <REF2>

* <MULTIPLY-U-I-32> <IMMED> <REF2>

* <DIVIDE-U-I-32> <IMMED> <REF2>

* <MOVE-U-I-32> <IMMED> <REF2>

* <REMAIN-U-I-32> <IMMED> <REF2>

* <COMPARE-U-I-32> <IMMED> <REF2>

  As an optimization, the following instructions have an implied constant of 0 or 1:

* <COMPARE-32-U-0> <REF> Compare to 0

* <INCREMENT-32-U> <REF> Add 1

* <DECREMENT-32-U> <REF> Subtract 1

* <CLEAR-32-U> <REF> Store zero

* <COMPARE-WITHIN-LIMITS-32-U> <REF> <IMMED> <IMMED>
  The first 32-bit constant represents an unsigned lower limit; the next 32-bit constant represents an unsigned upper limit.

* <SHIFT-LOGICAL-I-32> <REF> <IMMED>

  <REF> is both source and destination for a 32-bit shift. <IMMED> is an 8-bit signed shift counter; + = left, - = right.

  The following instructions specify a 32 bit constant as reference one. this constant is an unsigned 32 bit operand.

* <AND-I-32> <IMMED> <REF2>

* <IOR-I-32> <IMMED> <REF2>

* <XOR-I-32> <IMMED> <REF2>

* <SET-DIFF-I-32> <IMMED> <REF2>
  <REF2> becomes <IMMED> AND NOT <REF2>.

* <MASK-MERGE-I-32> <REF1> <REF2> <IMMED>
  <REF2> = (<REF1> AND <IMMED>) OR (<REF2> AND NOT <IMMED>).

3.4 Floating Point

All operations on single precision (32 bit) operands are performed totally in single precision, and all double precision (64 bit) operations are performed totally in double precision. Each operation on single or double precision will potentially set bits in the condition register (CR). Ultimately each operation will respond to overflow/underflow of exponent and there will be a test for divide by zero. See the Data Types and Formats Chapter for rounding information. There is an implicit truncation from double to single when a move of a double precision number is done in single precision mode. Floating point operations assume normalized values.

The following instructions refer to floating point numbers and do not distinguish between single and double precision.

* <ABSOLUTE-VALUE-FP> <REF>

  Set the sign of the value specified by <REF> to positive and leave the result in <REF>

* <NEGATE-FP> <REF>

  Change the sign of the value specified by <REF> leaving the value in <REF>.

* <EXTRACT-EXPONENT> <REF1> <REF2>

  Extract the exponent from the value specified by <REF1> and move it as an unsigned 8-bit quantity referenced by <REF2>.

3.4.1 Single Precision Floating Point

The following are the single precision floating point operations:

* <ADD-SP> <REF1> <REF2>

  Add the single precision value specified by <REF1> to the single precision value specified by <REF2> and move the results to <REF2>.

* <SUBTRACT-SP> <REF1> <REF2>

  Subtract the single precision value specified by <REF1> from the single precision value specified by <REF2> and move the results to <REF2>.

* <MULTIPLY-SP> <REF1> <REF2>

  Multiply the single precision value specified by <REF1> by the single precision value specified by <REF2> and move the single precision results to <REF2>.

* <DIVIDE-SP> <REF1> <REF2>

Divide the single precision value specified by <REF2> by the single prevision value specified by <REF1> and move the single precision results to <REF2>

* <MOVE-SP> <REF1> <REF2>

Move the single precision value specified by <REF1> to <REF2>.

* <COMPARE-SP> <REF1> <REF2>

Compare the single precision value specified by <REF1> with the single precision value specified by <REF2> and set the condition bits in the condition register (CR). The contents of <REF1> and <REF2> are unaltered.

* <NORMALIZE-SP> <REF>

The single precision value specified by <REF> is normalized and returned to <REF>.

* <INTEGERIZE-SP> <REF>

Integerize the single precision value specified by <REF> and move as a single precision value to <REF>.

* <COMPARE-ZERO-SP> <REF>

Compare the single precision value specified by <REF> to true zero and set the condition bits in the condition register (CR).

* <SCALE-SP> <REF1> <REF2>

Scale the single precision value specified by <REF1> by a factor indicated in the signed 8-bit quanitity specified by <REF2> and place the result in <REF1>.

* <HALVE-SP> <REF>

The single precision value specified by <REF> is divided by 2.0 and returned to <REF>.

3.4.2 Double Precision Floating Point

The following are the double precision floating point operations:

* <ADD-DP> <REF1> <REF2>

  Add the double precision value specified by <REF1> to the double precision value specified by <REF2> and move the results to <REF2>.

* <SUBTRACT-DP> <REF1> <REF2>

  Subtract the double precision value specified by <REF1> from the double precision value specified by <REF2> and move the results to <REF2>.

* <MULTIPLY-DP> <REF1> <REF2>

  Multiply the double precision value specified by <REF1> by the double precision value specified by <REF2> and move the double precision results to <REF2>.

* <DIVIDE-DP> <REF1> <REF2>

  Divide the double precision value specified by <REF2> by the double precision value specified by <REF1> and move the double precision results to <REF2>

* <MOVE-DP> <REF1> <REF2>

  Move the double precision value specified by <REF1> to <REF2>.

* <COMPARE-DP> <REF1> <REF2>

  Compare the double precision value specified by <REF1> with the double precision value specified by <REF2> and set the condition bits in the condition register (CR).The contents of <REF1> and <REF2> are unaltered.

* <NORMALIZE-DP> <REF>

  The double precision value specified by <REF> is normalized and returned to <REF>.

* <INTEGERIZE-DP> <REF>

Integerize the double precision value specified by <REF> and move as a double precision value to <REF>.

* <COMPARE-ZERO-DP> <REF>

Compare the double precision value specified by <REF> to true zero and set the condition bits in the condition register (CR).

* <SCALE-DP> <REF1> <REF2>

Scale the double precision value specified by <REF1> by a factor indicated in the signed 8-bit quanitity specified by <REF2> and place the result in <REF1>.

* <HALVE-DP> <REF>

The double precision value specified by <REF> is divided by 2.0 and returned to <REF>.

3.5 Character

The character 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 operations to occur.

Within this section a <STR-REF> will be an address of a string address descriptor which is described in the introduction chapter. <STR-REF>.PTR will correspond to the strings byte pointer. <STR-REF>.MAX will correspond to the string maximum length. <STR-REF>.CUR will correspond to the string-current-length. In scans and translates, single byte reference and table references are data byte pointers. The string descriptor always points to the beginning (left-most) byte of the string. Scans or moves down will have to add the current length to the byte data pointer to get the starting byte data pointer. In scans there are condition codes for failure due to current length being zero and character not found and for a successful scan. If character is not found then scans will set the index to be the current length plus 1.

The following instructions have been defined:

* <MOVE-BYTES-UP> <STR-REF1> <STR-REF2>

* <MOVE-BYTES-DOWN> <STR-REF1> <STR-REF2>

  Move bytes up or down from the reference in <STR-REF1>to the reference 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-BYTES-FILL-RIGHT-DOWN> <FILLER-BYTE> <STR-REF1> <STR-REF2>

* <MOVE-BYTES-FILL-LEFT-UP> <FILLER-BYTE> <STR-REF1> <STR-REF2>

  Move bytes from <STR-REF1> to <STR-REF2> with left or right justification with up or down movement using the filler byte to fill the remainder of the desired string.

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

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

  Scan a string referenced in <STR-REF2> up or down for the byte referenced by <REF1>. Set <REF3> be the index to the next byte position within the string. <REF3> is a signed 16-bit integer.

* <SCAN-NOT-BYTE-UP> <REF1> <STR-REF2> <REF3>

* <SCAN-NOT-BYTE-DOWN> <REF1> <STR-REF2> <REF3>

  Scan a string referenced in <STR-REF2> up or down for the first character not equal to the byte referenced by <REF1>. Set <REF3>.PTR to be the signed 16-bit index to the next byte position within the string.

* <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.

* <MOVE-TRANSLATED-STRING-UP> <REF1> <STR-REF2> <STR-REF3>

* <MOVE-TRANSLATED-STRING-DOWN> <REF1> <STR-REF2> <STR-REF3>

Move translated bytes using a 256-byte translation table referenced by <REF1> up or down 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.

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

Move a string referenced in <STR-REF2> to <STR-REF3> using each byte as an index into a 256-bit table referenced by <REF1> until the indexed bit is on. The move count is limited by MIN(<STR-REF2>.CUR, <STR-REF3>.MAX). Set <STR-REF3>.CUR to the number of characters moved.

3.6 Bit

The bit instructions fall into three classes of operations; single bit instructions , multi-bit string instructions and bit numerics.

3.6.1 Single Bits

The following instructions are 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 bit in the condition register (CR). Set the referenced bit.

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

  Test the bit referenced by <BIT-REF> and set the appropriate bit in the condition register (CR). Clear the referenced bit.

The following instructions are not indivisible.

* <TEST-BIT> <BIT-REF>

  Test the bit referenced by <BIT-REF> and set the appropriate bit in the condition register (CR).

* <SET-BIT> <BIT-REF>

  Set the bit referenced by <BIT-REF>.

* <CLEAR-BIT> <BIT-REF>

Clear the bit referenced by <BIT-REF>.

3.6.2 Multi-bit

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

Scan for first 1 in a bit string specified by <BIT-REF> with a length in <REF2>. Set <REF3> to be the bit offset to this bit.

* <COUNT-BITS> <BIT-REF> <REF2> <REF3>

Count the bits set in a bit string referenced by <BIT-REF> with a bit length in <REF2>. <REF3> is updated with the resulting count.

3.6.3 Bit Numeric

For the bit numeric move operations, a count is always specified in the instruction format as:

```
 -------------------------
|00|S |    LENGTH         |
 ---|---------------------
 0 1|2 3                 7
    |_______ SIGNED/UNSIGNED
```

Movement is always to a 32-bit destination. If the S bit is set, the sign bit is replicated to fill a 32-bit destination. Otherwise, the remaining bits are cleared to zero.

* <MOVE-FROM-BIT> <BIT-REF> <REF2> <REF3>

  Move a bit numeric specified by <BIT-REF> with count descriptor in <REF2> to a 32-bit destination referenced in <REF3>.

* <MOVE-TO-BIT> <BIT-REF> <REF2> <REF3>

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

3.7 Commercial

The following instructions support commercial arithmetics and editing. Floating point manipulations are not directly supported by the set. Explicit conversions must be used in these cases. It should be noted that packed decimal is in IBM format and is byte aligned. Thus, in some cases, the nible string must be zero extended for proper alignment.

For each referenced argument, there is an in-line attribute byte. This byte has the format

```
 _____________________________
|DATA|     LENGTH            |
|TYPE|                       |
|____|_______________________|
|0  2|3                     7|
```

where length is the length of the referenced numeric string in bytes, and data type denotes one of the following 8 types:

000 - Unpacked Decimal - Low Order Sign/Overpunch

001 - Unpacked Decimal - High Order Sign/Overpunch
010 - Unpacked Decimal - Trailing Sign
011 - Unpacked Decimal - Leading Sign
100 - Unpacked Decimal - Unsigned
101 - Packed Decimal - (IBM Format)
110 - Binary Integer - Signed
111 - Binary Integer - Unsigned

All digits referenced are checked for validity. If an invalid digit is referenced, the operation is terminated and error is signaled.

For data type 5, packed decimal, there must be an integral number of bytes including sign. Thus, in some cases, a higher order zero digit must be supplied.

There is one set of numerics, with two forms: A op B --> B and A op B --> C. These instructions directly implement COBOL references for the cases in which ON SIZE is not specified. These instructions perform the specified operation and store the results in the destination right justified zero filled. If significance remains after the storage in the intermediate results, an overflow condition occurs.

In the following instructions, the precision of the result is explicitly designated. The byte pointed to by the address is the most significant (Left-Most) byte of the numeric string.

Instruction	Operands
<ADD-DECIMAL>	<CREF1> <CREF2>
<ADD-DECIMAL-GIVING>	<CREF1> <CREF2> <CREF3>
<SUBTRACT-DECIMAL>	<CREF1> <CREF2>
<SUBTRACT-DECIMAL-GIVING>	<CREF1> <CREF2> <CREF3>
<MULTIPLY-DECIMAL>	<CREF1> <CREF2>
<MULTIPLY-DECIMAL-GIVING>	<CREF1> <CREF2> <CREF3>
<DIVIDE-DECIMAL>	<CREF1> <CREF2>
<DIVIDE-DECIMAL-GIVING>	<CREF1> <CREF2> <CREF3>
<REMAIN-DECIMAL-GIVING>	<CREF1> <CREF2> <CREF3>

The sign of the remainer is the sign of the dividend.

Instruction	Operands
<COMPARE-DECIMAL>	<CREF1> <CREF2>
<COMPARE-ZERO-DECIMAL>	<CREF1> An implied comparand of 0 equal in length to reference 1.
<MOVE-DECIMAL>	<CREF1> <CREF2>
<SCALE-LEFT>	<CREF1> <IMMED><REF2>

Scale (shift) left <REF1> the number of positions specified by the <IMMED> and store the results in <REF2>. Vacated positions are zero.

<SCALE-RIGHT> <CREF1> <IMMED><REF2>

Scale (shift) right <REF1> the number of positions specified by the <IMMED> and store the results in <REF2> Vacated positions are zero filled.

<ROUND> <CREF1> <CREF2>

The least significant digit position of <REF1> is examined. If 5 or greater, ten is added to <REF1> beginning at this position and the sum is moved to <REF2>. If the digit is less than 5, <REF1> is moved to <REF2>. Note: Reference 1 may have more precision then indicated by Attribute 1. Only one digit, however, is considered when ROUNDing is performed.

<EDIT> <CREF1><REF2> <IMMED><REF3>

3.8 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 plus the 32-bit unsigned 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 a one, two, four or eight to its current value.

* <POP-8> <REF>

Data General Corporation
Company Confidential

* <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 a one, two, four or eight from its current value.

* <MOVE-SP> <REF>

Move the current value of the stack pointer (SP) to the specified destination.

3.9 Jumps

3.9.1 Entry and Exit

In the following instructions, <PREF> refers to a procedure reference as define in the introduction 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 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) should be that following the corresponding <PUSH-PC> instruction.

```
|   *   <CALL-PDIFF>      <PREF>  <IMMED>         <#ARGCONST>
|       <REF1>...<REFN>
|
|   *   <CALL-PDIFF-PACKET>       <PREF>  <IMMED>         <REF>
|
|   *   <CALL>            <PREF>  <#ARGCONST>     <REF1>...<REFN>
|
|   *   <CALL-PACKET>     <PREF>  <REF>
```

These call operators are used to install a new stack environment then branch to a subroutine. The first two types of calls allow the setting of a procedure level difference on the stack while building the new frame. This difference is defined by the inline 8-bit unsigned integer referred to as <IMMED>. <PREF> is the specifier of the subroutine. <#ARGCONST> is an 8-bit unsigned integer representing the number of argument references which are to follow. <REF> and <REF1>...<REFN> are the references to a single argument or a list of n arguments. (n being the value of <#ARGCONST>. The call will build a packet of arguments which can be referenced by BR2. This packet has the following format:

```
                  <REFN>
                     .
                     .
                     .
BR2------->       <REF1>
               <#ARGCONST>
```

Callers wishing to build their own parameter packets may use the packet type call opcodes instead. In this case, <REF> is the address of the packet which is identical to that described above. This call places <REF> in BR2.

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

* <RETURN>

Return from procedure or trap handler thru the PC contained within the current frame.

* <RETURN-ARG> <IMMED>

  Functions like <RETURN> except the PC value is the value of the argument specified by the immediate.

3.9.2 Vanilla Jumps

The following instructions are specified:

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

  Jump to <PREF> if the bits condition register (CR) matches the <COND> field.

* <JUMP-GT> <PREF>
* <JUMP-LT> <PREF>
* <JUMP-NE> <PREF>
* <JUMP-EQ> <PREF>
* <JUMP-GE> <PREF>
* <JUMP-LE> <PREF>
* <JUMP> <PREF>

  A jump to <PREF> is unconditional or occurs based on the state of the N and Z bits of the condition register (CR).

3.9.3 Dispatches

All dispatch instrucrtions use a table of the following format:

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

Where the lower bound and upper bound are both signed 16-bit integers. All dispatches through a table validate the index as lower bound <=index <=upper bound, select the indexed <Procedure Pointer>. If the index is not within the bound range, the PC will be set to the next instruction following the dispatch and no branch will occur. Within the discussion of DISPATCH all references labeled <DREF> will be pointers to dispatch tables 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-IMMEDIATE> <REF>
  <INLINE-DISPATCH-TABLE>

* <DISPATCH-IMMEDIATE-PUSHPC> <REF>
  <INLINE-DISPATCH-TABLE>

* <DISPATCH> <REF> <DREF>

* <DISPATCH-PUSHPC> <REF> <DREF> par() Dispatch through an inline dispatch table, or a dispatch table referenced by <DREF> in the last two cases, based on a signed 16-bit index pointed to by <REF>. The PC which is pushed in the <PUSH-PC> type is the PC at the next instruction.

* <DISPATCH-CALL> <XREF> <DREF> <#ARGCONST>
  <REF1...REFN>

* <DISPATCH-CALL-PACKET> <XREF> <DREF> <REF>

  The dispatch with call operations are identical to the call operations defined in the jumps section of the instruction chapter except that the <PREF> is selected from a dispatch table specified by <DREF>. The index is a signed

16-bit value specified by <XREF>. A dispatch failure will cause this instruction to NOP and the next instruction to be executed.

3.10 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.

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

* <PURGE-ATU>
* <LOAD-PHYSICAL>
* <STORE-PHYSICAL>
* <LOAD-PSR>
* <STORE-PSR>

3.13 Input/Output

* <IO-IN>
* <IO-OUT>

Refer to I/O System chapter.

3.14 Miscellaneous

The following instructions have been specified:

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

  Move the effective address of <REF1> to the location specified by <REF2>.

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

  Move <REF3> bytes for <REF1> to <REF2>.

* <COPY-IMMED> <IMMED> <REF2> <REF3>

  Move <REF2> copys of the byte <IMMED> to the area referenced by <REF3>.

* <LOAD-CONDITION-REG> <REF>

  Load the condition register with the value specified by <REF>.

* <STORE-CONDITION-REG> <REF>

  Store the condition register at <REF>.

--End of Chapter--

CHAPTER 4 - Interrupts and Traps

4.1 General

All events in an EGO 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 system. 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. System traps 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 a single argument 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.

The number of different types of state blocks and their exact formats are implementation dependent. Note that certain types of returns have the potential for protection violations and thus must be protected against execution by non-secure procedure segments.

4.2 Procedure Traps

When a procedure trap is taken, the state is pushed on the current stack and the pointer to the handler is found in word zero of the current procedure segment. Only the low order 22 bits of the word are significant, and these are the address within the current segment of the trap handler. If these bits are zero, there is no trap handler, and a process trap is generated.

Procedure trapping can be disabled by setting the appropriate bit in the PSR. When this bit is set, all procedure traps are ignored.

4.3 Process Traps

There are three types of process traps - reserved instructions, non-primitive I/O, and faults. All of these are handled in the ring 0 procedure segment (segment 2) and the state is pushed on the ring 0 stack of the current process (segment 1). For faults, the pointer to the handler is found in word one of the procedure segment, and has the same format as the procedure trap handler pointer. (Note that this segment can also generate procedure traps and thus must have a procedure trap handler address in word zero.) Dispatching for reserved instructions is described in the instruction set chapter, and for non-primitve I/O traps in the I/O system chapter.

4.4 System Traps

System traps are also handled in the ring 0 procedure segment (segment 2). In this case, however, the pointer is not to a trap handler, but rather to a table containing one entry for for each possible system trap. Each entry is one word long and has the same format as the procedure trap handler pointer. This pointer is kept in a register loaded in an implementation defined fashion. When a system trap occurs, the value of the trap is used as an index into the table to find the address of the correct handler.

Stack handling for system traps is also slightly different than for other traps. There is a system stack segment (segment 0)

used for handling system traps. When a system trap occurs, the state is pushed on the current stack. If the current stack is segment 0 (indicating nested system traps), the trap handler is then invoked. Otherwise, the current stack pointer is written back into the current TCB and also placed in word zero of the system stack segment. An indicator of this special type of state block is pushed on the system stack segment, and the stack and frame pointers are set to point to segment 0 just beyond this block.

During the resolution of any system trap, if the trap handler encounters a situation which makes it desire to change the current process after dismissing the trap (e.g. a time slice has expired, or an I/O event has completed on a high priority process), it zeroes the stack pointer in word 0 of the system stack segment. When the return instruction encounters this special state block, it checks if the stack pointer in word 0 is still there. If so, it restores the current state from the stack pointed to by that stack pointer. If not, it branches to the system rescheduling routine pointed to by word three of the ring 0 procedure segment.

System traps can be disabled by setting the appropriate bit in the PSR. When this bit is set, system traps will be queued, and will be generated when the bit is cleared. This is analogous to interrupt enable/disable on the Eclipse.

Due to the nature of system traps and the fault type of process traps, page faults can not be taken while resolving them. Thus, all of the handler pointers and the trap handlers themselves must be resident in system memory.

--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

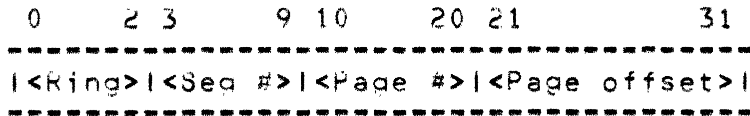
Every ring has its own stack segment with a format as described in the Introduction. When a ring crossing is detected during execution of a call instruction, the stack segment number for the new ring is fetched from the Task Control Block. Arguments and the procedure state block are pushed onto the new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

--End of Chapter--

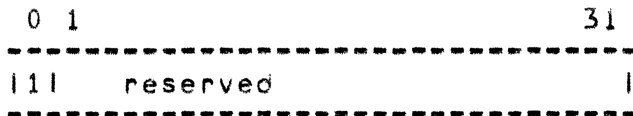
CHAPTER 6 - Memory Management

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

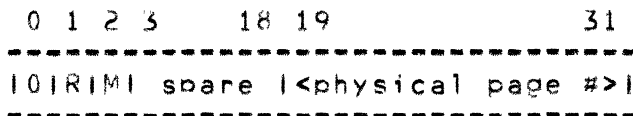
For purposes of memory management, the logical address described in 1.2.2 is further subdivided such that each segment consists of 2K pages, each page containing 2K bytes:



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:



Invalid (unallocated) page



Resident page

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:

```
 0 1 7 8                       31
----------------------------------
|0|   |<physical page table ptr>|
----------------------------------
```

PTP for a resident page table page

```
 0 1                           31
----------------------------------
|1|  reserved                   |
----------------------------------
```

PTP for a non-resident 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, EGO 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--

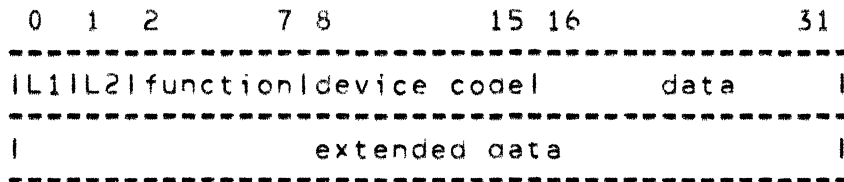
CHAPTER 7 - I/O System

7.1 Organization

Information is transferred between an EGO processor and its I/O system by means of directives - packets of self-describing data and control. The medium for this transfer is physical memory; therefore directives are referred to by physical address. There exist two directive types: primitive, and non-primitive. Primitive directives either represent a Nova/Eclipse programmed I/O instruction, or specify the location in physical memory of a non-primitive directive whose execution is desired. Non-primitive directives represent high-order I/O functions, for example, read next sequential record from file "FOO".

Every EGO implementation will possess hardware capable of executing all primitive I/O instructions. A specific non-primitive directive may or may not be executable by a particular I/O system implementation. If it is not, the I/O system initiates a process trap in some CPU in order that the directive be emulated; the choice of CPU depends on the implementation's resource sharing algorithm. Thus each non-primitive directive requires either an I/O processor capable of executing it or appropriate code in an emulator.

Primitive I/O directives are two 32-bit words in length, and have the following format:



Each CPU has a unique pair of words in physical memory called the primitive I/O words (PIOW) into which it places primitive directives. This location is specified by the contents of the

Data General Corporation
Company Confidential

primitive I/O pointer (PIOP), which is provided to each CPU and I/O converter/processor at system initialization. The location of the PIOW can be changed if necessitated by failure in primary memory.

Two I/O instructions are provided in the EGO kernel instruction set:

<I/O IN><function><device code><destination operand reference>

<I/O OUT><function><device code><source operand reference>

Execution of one of these instructions requires an EGO CPU to first ensure that the previous directive placed in the PIOW has been accepted by the I/O system. Bit L1 of the PIOW is the semaphore which controls this process. The function and device code fields in the instruction are then stored into the appropriate field of the PIOW. If the instruction is I/O OUT, the source operand is fetched and placed in the PIOW data field, and a "new directive" command is sent to the I/O system. The CPU is then free to continue execution. If the instruction is I/O IN, the "new directive" command is immediately sent to the I/O system, which obtains the required datum, stores it in the data field of the PIOW, and sets the L2 bit, which functions as a data-in semaphore. The CPU meanwhile loops on L2 until it is set by the I/O system, indicating the presence of the required datum in the PIOW data field. L2 is then reset, and the datum is stored into the destination operand specified in the instruction.

The first 16 primitive I/O functions are mapped directly from the Nova/Eclipse programmed I/O instruction set. For reference, the format of a Nova/Eclipse programmed I/O instruction is shown below:

```
 0 2 34 5     9 10        15
----------------------------
|011|AC|funtion|device code|
----------------------------
```

Bit 7 of this instruction implies the direction of the transfer - this is subsumed by the EGO I/O instruction opcode. The remaining four bits of the function define the first 16 primitive I/O functions. The skip instructions translate into input instructions; the I/O system simply stores SELB or SELD into the low order bit of the PIOW data field, which the CPU then tests.

Special Nova/Eclipse I/O instructions, i.e. those with device code 0, 1, 2, 3, or 77(octal), are emulated directly without the use of the PIOW.

The seventeenth EGO primitive I/O function specifies the physical address of a packet describing a non-primitive I/O directive. This address is placed in the extended data field of the PIOW; an I/O OUT instruction transfers the information to the I/O system. The remaining 47 functions are free for later definition. An obvious use will be the transfer of 32-bit data to and from a new family of device controllers through the extended data field of the PIOW. Microcode implementing these new I/O functions can be distributed (via floppy disk) if and when the new controllers become available.

The definition of non-primitive I/O directives can be made non-architectural, if we are willing to pay the penalty of carrying around an emulator or I/O processor driver for each set of non-primitive directives defined. Alternatively, we can architecturally define hierarchical layers of non-primitive directives. We defer this decision until an operating system is defined, at which point substantially more information will be available on which the decision can be based.

7.2 Objectives for the EGO-1 I/O system

The following objectives have been established for first EGO implementation. These are not architectural in nature, but are listed in this document for completeness.

* The EGO-1 standard I/O bus will be identical to the Nova/Eclipse standard I/O bus as specified in "The Interface Designer's Guide for the Nova and Eclipse" chapter 4 and appendix D. Thus any existing controllers conforming to this specification will be directly compatible with EGO-1.

* The EGO-1 high-speed channel will be identical to the Eclipse high-speed channel as defined in its specification. Thus high-speed channel controllers will be directly compatible with EGO-1.

* Standard data channel and high-speed channel transfers between controllers and EGO-1's system cache will not degrade CPU performance except when simultaneous cache faults occur. EGO-1 will be capable of sustaining I/O

rates of 15 million bytes per second with peaks approaching 20 million bytes per second.

* EGO-1's I/O converter will perform all primitive I/O directives and initiate a process trap in the CPU on receiving any non-primitive I/O directive; in response to this trap, the CPU will emulate the non-primitive directive on behalf of the I/O converter. I/O system intelligence may be increased by replacing the I/O converter with an I/O processor capable of executing some or all non-primitive I/O directives.

--End of Chapter--

CHAPTER 8 - Availability/Reliability/Maintainability

8.1 Overview

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

8.2 EGO 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 EGO console functions executed through a teletype interface. In addition, it will improve system maintainability by proviiding a software diagnostic capability external to the EGO processor system and independent of its correct operation. Availa- bility 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 EGO 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 EGO 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--

--End of Chapter--

0 3-2 SUBTRACT-16
0 3-2 RSUB-16
0 3-2 MULTIPLY-16
0 3-2 REFIDE-16
0 3-2 REMAIN-16
0 3-3 MOVE-16
0 3-3 COMPARE-16
0 3-3 COMPARE-WITHIN-LIMITS-16
0 3-3 SHIFT-ARITHMETIC-16>
0 3-3 ABSOLUTE-VALUE-16
0 3-3 NEGATE-16
0 3-3 ADD-I-16
0 3-3 SUBTRACT-I-16
0 3-3 MULTIPLY-I-16
0 3-3 DIVIDE-I-16
0 3-3 REMAIN-I-16
0 3-3 MOVE-I-16
0 3-3 COMPARE-I-16
0 3-4 COMPARE-WITHIN-LIMITS-I-16
0 3-4 SHIFT-ARITHMETIC-I-16
0 3-4 COMPARE-16-0
0 3-4 INCREMENT-16
0 3-4 DECREMENT-16
0 3-4 CLEAR-16
0 3-4 ADD-32
0 3-4 SUBTRACT-32
0 3-4 MULTIPLY-32
0 3-4 DIVIDE-32
0 3-4 MOVE-32
0 3-5 COMPARE-32
0 3-5 REMAIN-32
0 3-5 ABSOLUTE-VALUE-32
0 3-5 NEGATE-32
0 3-5 COMPARE-WITHIN-LIMITS-32
0 3-5 SHIFT-ARITHMETIC-32
0 3-5 ADD-I-32
0 3-5 SUBTRACT-I-32
0 3-5 MULTIPLY-I-32
0 3-5 DIVIDE-I-32
0 3-5 REMAIN-I-32
0 3-5 MOVE-I-32
0 3-6 COMPARE-I-32
0 3-6 COMPARE-WITHIN-LIMITS-I-32
0 3-6 SHIFT-ARITHMETIC-I-32
0 3-6 COMPARE-32-0
0 3-6 INCREMENT-32
0 3-6 DECREMENT-32
0 3-6 CLEAR-32
0 3-7 ADD-8
0 3-7 SUBTRACT-8
0 3-7 MULTIPLY-8
0 3-7 DIVIDE-8
0 3-7 REMAIN-8
0 3-7 MOVE-8
0 3-7 COMPARE-8
0 3-7 COMPARE-WITHIN-LIMITS-8
0 3-7 SHIFT-LOGICAL-8
0 3-7 AND-8
0 3-7 IOR-8
0 3-7 XOR-8
0 3-7 SET-DIFF-8
0 3-7 COMPLEMENT-8
0 3-7 MASK-MERGE-8

0 3-8 ADD-I-8
0 3-8 SUBTRACT-I-8
0 3-8 MULTIPLY-I-8
0 3-8 DIVIDE-I-8
0 3-8 MOVE-I-8
0 3-8 REMAIN-I-8
0 3-8 COMPARE-I-8
0 3-8 SHIFT-LOGICAL-I-8
0 3-8 COMPARE-WITHIN-LIMITS-I-8
0 3-8 AND-I-8
0 3-8 IOR-I-8
0 3-8 XOR-I-8
0 3-8 SET-DIFF-I-8
0 3-8 MASK-MERGE-I-8
0 3-8 COMPARE-8-0
0 3-9 INCREMENT-8
0 3-9 DECREMENT-8
0 3-9 CLEAR-8
0 3-9 ADD-U-32
0 3-9 SUBTRACT-U-32
0 3-9 MULTIPLY-U-32
0 3-9 DIVIDE-U-32
0 3-9 REMAIN-U-32
0 3-9 MOVE-U-32
0 3-9 COMPARE-U-32
0 3-9 COMPARE-WITHIN-LIMITS-U-32
0 3-9 SHIFT-LOGICAL-32
0 3-10 AND-32
0 3-10 IOR-32
0 3-10 XOR-32
0 3-10 SET-DIFF-32
0 3-10 COMPLEMENT-32
0 3-10 MASK-MERGE-32
0 3-10 ADD-U-I-32
0 3-10 SUBTRACT-U-I-32
0 3-10 MULTIPLY-U-I-32
0 3-10 DIVIDE-U-I-32
0 3-10 MOVE-U-I-32
0 3-10 REMAIN-U-I-32
0 3-10 COMPARE-U-I-32
0 3-10 COMPARE-32-0
0 3-10 INCREMENT-32-U
0 3-11 DECREMENT-32-U
0 3-11 CLEAR-32-U
0 3-11 COMPARE-WITHIN-LIMITS-32-U
0 3-11 SHIFT-LOGICAL-I-32
0 3-11 AND-I-32
0 3-11 IOR-I-32
0 3-11 XOR-I-32
0 3-11 SET-DIFF-I-32
0 3-11 MASK-MERGE-I-32
0 3-12 ABSOLUTE-VALUE-FP
0 3-12 NEGATE-FP
0 3-12 EXTRACT-EXPONENT
0 3-12 ADD-SP
0 3-12 SUBTRACT-SP
0 3-12 MULTIPLY-SP
0 3-13 DIVIDE-SP
0 3-13 MOVE-SP
0 3-13 COMPARE-SP
0 3-13 NORMALIZE-SP
0 3-13 INTEGERIZE-SP
0 3-13 COMPARE-ZERO-SP

0 3-13 SCALE-SP
0 3-13 HALVE-SP
0 3-14 ADD-DP
0 3-14 SUBTRACT-SP
0 3-14 MULTIPLY-DP
0 3-14 DIVIDE-DP
0 3-14 MOVE-DP
0 3-14 COMPARE-DP
0 3-14 NORMALIZE-DP
0 3-15 INTEGERIZE-DP
0 3-15 COMPARE-ZERO-DP
0 3-15 SCALE-DP
0 3-15 HALVE-DP
0 3-16 MOVE-BYTES-UP
0 3-16 MOVE-BYTES-DOWN
0 3-16 MOVE-BYTES-FILL-RIIGHT-DOWN
0 3-16 MOVE-BYTES-FILL-LEFT-UP
0 3-16 SCAN-BYTE-UP
0 3-16 SCAN-BYTE-DOWN
0 3-16 SCAN-NOT-BYTE-UP
0 3-16 SCAN-NOT-BYTE-DOWN
0 3-16 COMPARE-STRINGS
0 3-16 SUBSTRING
0 3-17 SCAN-SUBSTRING-UP
0 3-17 SCAN-SUBSTRING-DOWN
0 3-17 MOVE-TRANSLATED-STRING-UP
0 3-17 MOVE-TRANSLATED-STRING-DOWN
0 3-17 CHARACTER-SCAN-UNTIL-TRUE
0 3-17 CHARACTER-MOVE-UNTIL-TRUE
0 3-18 TEST-AND-SET-BIT
0 3-18 TEST-AND-CLEAR-BIT
0 3-18 TEST-BIT
0 3-18 SET-BIT
0 3-18 CLEAR-BIT
0 3-19 FIND-LEADING-BIT
0 3-19 COUNT-BITS
0 3-20 MOVE-FROM-BIT
0 3-20 MOVE-TO-BIT
0 3-21 ADD-DECIMAL
0 3-21 ADD-DECIMAL-GIVING
0 3-21 SUBTRACT-DECIMAL
0 3-21 SUBTRACT-DECIMAL-GIVING
0 3-21 MULTIPLY-DECIMAL
0 3-21 MULTIPLY-DECIMAL-GIVING
0 3-21 DIVIDE-DECIMAL
0 3-21 DIVIDE-DECIMAL-GIVING
0 3-21 REMAIN-DECIMAL-GIVING
0 3-21 COMPARE-DECIMAL
0 3-21 COMPARE-ZERO-DECIMAL
0 3-21 MOVE-DECIMAL
0 3-21 SCALE-LEFT
0 3-22 SCALE-RIGHT
0 3-22 ROUND
0 3-22 EDIT
0 3-22 MODIFY-STACK-POINTER
0 3-22 PUSH-8
0 3-22 PUSH-16
0 3-22 PUSH-32
0 3-22 PUSH-64
0 3-22 POP-8
0 3-23 POP-16
0 3-23 POP-32
0 3-23 POP-64

0 3-23 MOVE-SP
0 3-23 PUSH-PC
0 3-23 POP-PC
0 3-24 CALL-PDIFF
0 3-24 CALL-PDIFF-PACKET
0 3-24 CALL
0 3-24 CALL-PACKET
0 3-24 RETURN
0 3-25 RETURN-ARG
0 3-25 JUMP-ON-CONDITION
0 3-25 JUMP-GT
0 3-25 JUMP-LT
0 3-25 JUMP-NE
0 3-25 JUMP-EQ
0 3-25 JUMP-GE
0 3-25 JUMP-LE
0 3-25 JUMP
0 3-26 DISPATCH-IMMEDIATE
0 3-26 DISPATCH-IMMEDIATE-PUSHPC
0 3-26 DISPATCH
0 3-26 DISPATCH-PUSHPC
0 3-26 DISPATCH-CALL
0 3-26 DISPATCH-CALL-PACKET
0 3-27 CONVERT-INTEGER-TO-SP
0 3-27 CONVERT-INTEGER-TO-DP
0 3-27 CONVERT-SP-TO-INTEGER
0 3-27 CONVERT-DP-TO-INTEGER
0 3-27 CONVERT-SP-TO-DP
0 3-27 CONVERT-CHARACTER-TO-DP
0 3-27 CONVERT-DP-TO-CHARACTER
0 3-28 PURGE-ATU
0 3-28 LOAD-PHYSICAL
0 3-28 STORE-PHYSICAL
0 3-28 LOAD-PSR
0 3-28 STORE-PSR
0 3-28 IO-IN
0 3-28 IO-OUT
0 3-29 LOAD-EFFECTIVE-ADDRESS
0 3-29 COPY
0 3-29 COPY-IMMED
0 3-29 LOAD-CONDITION-REG
0 3-29 STORE-CONDITION-REG