FHP Program Architecture

DOCUMENT NUMBER: 90-000005-01

AUTHORS: Various

DATE: November 6, 1979

ABSTRACT: This document defines those FHP architectural abstractions visible to the functioning program.

KEYWORDS: FHP architecture

Data General Corporation
Company Confidential

FHP Program Architecture

DOCUMENT NUMBER: 90-000005-01

AUTHORS: Various

DATE: November 6, 1979

ABSTRACT: This document defines those FHP architectural abstractions visible to the functioning program.

KEYWORDS: FHP architecture

Data General Corporation
Company Confidential

Contents

Chapter 1--Introduction 1-1

1.1 Objects 1-1
1.1.1 Designation 1-1
1.1.2 Logical Address 1-1
1.1.3 Accessing and Protection 1-1
1.1.4 Procedure Objects 1-2
1.1.5 Objects of Extended Type 1-2
1.1.6 Object Management 1-2
1.2 Procedure Objects 1-2
1.2.1 Instructions 1-3
1.2.2 Procedures 1-3
1.3 Object Protection 1-3
1.3.1 Subjects 1-4

Chapter 2--Objects 2-1

2.1 Object Designation 2-1
2.2 Logical Addresses 2-1
2.3 Object Protection 2-1
2.4 Object Accessing Functions 2-2
2.4.1 Write_object 2-2
2.4.2 Read_object 2-3
2.4.3 Fetch_from_object 2-4

Chapter 3--Procedure Object Structure 3-1

3.1 Introduction 3-1
3.2 Procedure Object Header 3-1
3.3 Entry Descriptor 3-3
3.4 Arg_info_array 3-4
3.5 Procedure Environment Descriptor 3-8
3.6 Instructions 3-10
3.6.1 Opcode 3-10
3.6.2 Operands 3-10
3.6.2.1 Name Syllables 3-11
3.6.2.2 Literals 3-11
3.6.2.3 Relative Branches 3-11
3.6.2.4 Absolute Branches 3-11
3.7 Pointers 3-12
3.8 Associative Addressing 3-14

3.8.1 The Associated Address Table 3-14
3.8.2 get_associated_address 3-15
3.9 Referencing Static Data 3-16

Chapter 4--Architectural Base Registers 4-1

4.1.1 Frame Pointer 4-1
4.1.2 Static Data Pointer 4-1
4.1.3 Procedure Base Pointer 4-1

Chapter 5--Name Table and Name Resolution 5-1

5.1 Name Table Entries and Name Table Fields 5-1
5.2 Address Resolution 5-5
5.2.1 Resolution of a Name Table Entry 5-6
5.2.2 BASE Field 5-6
5.3 DISPLACEMENT Field 5-6
5.4 INDEX_NAME Field 5-7
5.5 IES Field 5-7
5.6 LENGTH Field 5-7
5.7 Name Table Entry Evaluation 5-8

Chapter 6--Tracing Facility 6-1

6.1 Trace Data Structures 6-1
6.1.1 Trace Pointer 6-2
6.1.2 Trace Table Header 6-2
6.1.3 Trace Tables 6-4
6.1.4 Class-Specific Trace Event Tables 6-5
6.1.4.1 S-Op Trace Event Table 6-5
6.1.4.2 Name Resolve/Evaluate Trace Event Table . . . 6-6
6.1.4.3 Procedure Transition Trace Event Table . . . 6-6
6.1.4.4 Data Store/Fetch Trace Event Table 6-8
6.2 Considerations For SPRINT 6-9
6.3 Access to Macrostate 6-10
6.4 Debugger/Namespace Functions 6-10
6.4.1 Get_Current_FP (current_fp) 6-11
6.4.2 Get_Previous_FP (fp, prev_fp, err) 6-11
6.4.3 Get_Successor_FP (fp,succ_fp,err) 6-12
6.4.4 Get_State (fp, state_ptr, err) 6-12
6.4.5 Set_State (fp, state_ptr, err) 6-12
6.4.6 Swap_trace_pointers (new_trace_ptr, old_trace_ptr) 6-13
6.5 Macro State Definition 6-13

Chapter 7--Exception Conditions, Faults, and SIGNALs 7-1

7.1 Introduction 7-1
7.2 Conditions 7-1
7.2.1 ILLEGAL_S_OP 7-1
7.2.2 ILLEGAL_OPERAND_SYLLABLE_SIZE 7-2
7.2.3 NAME_OUT_OF_RANGE 7-2
7.2.4 ILLEGAL_EAR_TRAP 7-2
7.2.5 INCONSISTENT_NAME_TABLE_ENTRY 7-3
7.2.6 INVALID_POINTER 7-3
7.2.7 INVALID_ENTRY_DESCRIPTOR 7-4
7.2.8 INVALID_S_INTERPRETER 7-4
7.2.9 INACTIVE_S_INTERPRETER 7-4
7.2.10 S_OP_TRACE 7-5
7.2.11 NAME_RESOLVE/EVAL_TRACE 7-5
7.2.12 PROCEDURE_CALL_TRACE 7-5
7.2.13 PROCEDURE_RETURN_TRACE 7-6
7.2.14 PROCEDURE_LEAVE_TRACE 7-6
7.2.15 PROCEDURE_REENTER_TRACE 7-7
7.2.16 DATA_FETCH_TRACE 7-7
7.2.17 DATA_STORE_TRACE 7-8
7.2.18 NONRESOLVABLE_POINTER 7-8
7.2.19 STATIC_DATA 7-9

Chapter 8--Call and Return 8-1

8.1 Call 8-1
8.1.1 Phase one - Complete_CALLERs_Activation 8-1
8.1.2 Phase two - Locate_TARGET_Environment 8-2
8.1.3 Phase three - Build_TARGET_Activation 8-4
8.2 Return 8-5
8.3 Nonlocal Goto 8-5
8.3.1 Restrictions on Nonlocal Goto 8-5
8.3.2 Semantics of Nonlocal Goto 8-6

Chapter 9--S-Languages 9-1

9.1 Branches 9-1
9.1.1 Relative Branches 9-1
9.1.2 Offset Branch 9-1
9.2 Exception Conditions 9-2
9.2.1 Protection Exceptions 9-2
9.2.2 Namespace Errors 9-2
9.3 Long Instructions 9-2
9.4 Overlapping Operands 9-2
9.5 Definition Format 9-3

9.6 Invalid S-Ops 9-3

Chapter 10--SPL S-Language 10-1

10.1 Data Types and Their Representation 10-1
10.1.1 Integers 10-1
10.1.1.1 Unsigned Integers 10-1
10.1.1.2 Signed Integers 10-1
10.1.2 Bit String 10-2
10.1.3 Pointers 10-2
10.1.4 Typed Operators 10-2
10.2 SPL Exceptions 10-3
10.3 SPL S-Language Operation Definitions 10-3
10.3.1 Control Instructions 10-3
10.3.1.1 BRANCH IF INTEGER ZERO 10-3
10.3.1.2 BRANCH IF INTEGER NOT ZERO 10-4
10.3.1.3 BRANCH IF INTEGER LESS THAN ZERO 10-4
10.3.1.4 BRANCH IF INTEGER LESS THAN OR EQUAL TO ZERO . 10-4
10.3.1.5 BRANCH IF INTEGER GREATER THAN ZERO 10-5
10.3.1.6 BRANCH IF INTEGER GREATER THAN OR EQUAL TO ZERO10-5
10.3.1.7 BRANCH IF INTEGER EQUAL 10-5
10.3.1.8 BRANCH IF INTEGER NOT EQUAL 10-6
10.3.1.9 BRANCH IF INTEGER LESS THAN 10-5
10.3.1.10 BRANCH IF INTEGER LESS THAN OR EQUAL . . . 10-6
10.3.1.11 BRANCH IF BIT STRING EQUAL TO ZERO 10-7
10.3.1.12 BRANCH IF BIT STRING NOT EQUAL TO ZERO . . 10-7
10.3.1.13 BRANCH IF BIT STRINGS EQUAL 10-8
10.3.1.14 BRANCH IF BIT STRINGS NOT EQUAL 10-8
10.3.1.15 BRANCH IF BIT STRING LESS THAN 10-8
10.3.1.16 BRANCH IF BIT STRING LESS THAN OR EQUAL . . 10-9
10.3.1.17 BRANCH IF IN BOUNDS 10-9
10.3.1.18 BRANCH IF NOT IN BOUNDS 10-10
10.3.1.19 BRANCH IF SUBSET 10-10
10.3.1.20 BRANCH IF NOT SUBSET 10-11
10.3.1.21 BRANCH IF POINTER EQUAL 10-11
10.3.1.22 BRANCH IF POINTER NOT EQUAL 10-12
10.3.1.23 FIND FIRST ONE ELSE BRANCH 10-12
10.3.1.24 FIND NEXT ONE AND BRANCH 10-13
10.3.1.25 FIND PREVIOUS ONE AND BRANCH 10-14
10.3.1.26 FIND LAST ONE ELSE BRANCH 10-15
10.3.1.27 LOOP DOWN TO ZERO 10-16
10.3.1.28 LOOP UP 10-16
10.3.1.29 LOOP DOWN 10-16
10.3.1.30 BRANCH IF NULL POINTER 10-17
10.3.1.31 BRANCH IF NOT NULL POINTER 10-17
10.3.1.32 SELF RELATIVE BRANCH 10-18
10.3.1.33 PROCEDURE OBJECT RELATIVE BRANCH 10-18

10.3.2 Integer Arithmetic and Bit String Instructions . 10-18
10.3.2.1 CLEAR INTEGER 10-18
10.3.2.2 CLEAR BIT STRING 10-19
10.3.2.3 SET INTEGER 10-19
10.3.2.4 SET BIT STRING 10-19
10.3.2.5 SET TO ONE 10-19
10.3.2.6 COMPLEMENT WITH ONE OPERAND 10-20
10.3.2.7 COMPLEMENT 10-20
10.3.2.8 AND WITH TWO OPERANDS 10-20
10.3.2.9 AND 10-20
10.3.2.10 OR WITH TWO OPERANDS 10-21
10.3.2.11 OR 10-21
10.3.2.12 AND COMPLEMENT WITH TWO OPERANDS 10-21
10.3.2.13 AND COMPLEMENT 10-22
10.3.2.14 EXCLUSIVE OR 10-22
10.3.2.15 NEGATE INTEGER WITH ONE OPERAND 10-22
10.3.2.16 NEGATE INTEGER 10-23
10.3.2.17 ABSOLUTE VALUE INTEGER 10-23
10.3.2.18 INCREMENT INTEGER WITH ONE OPERAND 10-23
10.3.2.19 INCREMENT INTEGER 10-24
10.3.2.20 DECREMENT INTEGER WITH ONE OPERAND 10-24
10.3.2.21 DECREMENT INTEGER 10-24
10.3.2.22 ADD INTEGER WITH TWO OPERANDS 10-25
10.3.2.23 ADD INTEGER 10-25
10.3.2.24 SUBTRACT INTEGER WITH TWO OPERANDS 10-25
10.3.2.25 SUBTRACT INTEGER 10-26
10.3.2.26 MULTIPLY INTEGER WITH TWO OPERANDS 10-26
10.3.2.27 MULTIPLY INTEGER 10-26
10.3.2.28 DIVIDE INTEGER 10-27
10.3.2.29 REMAINDER INTEGER 10-27
10.3.3 Miscellaneous Instructions 10-28
10.3.3.1 MOVE BIT STRING 10-28
10.3.3.2 INTEGER MOVE 10-28
10.3.3.3 POINTER MOVE 10-29
10.3.3.4 CREATE POINTER 10-29
10.3.3.5 CREATE GENERAL POINTER 10-29
10.3.3.6 STORE NULL POINTER 10-30
10.3.3.7 RESERVE 10-30
10.3.3.8 RELEASE 10-30
10.3.4 Architectural Instructions 10-31
10.3.4.1 CALL 10-31
10.3.4.2 RETURN 10-31
10.3.4.3 NOP 10-32

Chapter 11--Fortran S-Language 11-1

11.1 Data Types and Their Representation 11-1

29

8

3

11.1.1 LOGICAL 11-1
11.1.2 INTEGER 11-2
11.1.3 POINTER 11-3
11.1.4 FLOATING POINT 11-3
11.1.5 CHARACTER STRING 11-4
11.2 FORTRAN Exceptions 11-5
11.3 FORTRAN S-Language Operation Definitions 11-7
11.3.1 Control Instructions 11-7
11.3.1.1 BRANCH IF ZERO 11-7
11.3.1.2 BRANCH IF NOT ZERO 11-7
11.3.1.3 BRANCH IF LESS THAN ZERO 11-8
11.3.1.4 BRANCH IF LESS THAN OR EQUAL TO ZERO . . . 11-8
11.3.1.5 BRANCH IF GREATER THAN ZERO 11-8
11.3.1.6 BRANCH IF GREATER THAN OR EQUAL TO ZERO . . 11-9
11.3.1.7 BRANCH IF INTEGER EQUAL 11-9
11.3.1.8 BRANCH IF INTEGER NOT EQUAL 11-9
11.3.1.9 BRANCH IF INTEGER LESS THAN 11-10
11.3.1.10 BRANCH IF INTEGER LESS THAN OR EQUAL . . . 11-10
11.3.1.11 BRANCH IF FLOATING EQUAL 11-10
11.3.1.12 BRANCH IF FLOATING NOT EQUAL 11-11
11.3.1.13 BRANCH IF FLOATING LESS THAN 11-11
11.3.1.14 BRANCH IF FLOATING LESS THAN OR EQUAL . . . 11-11
11.3.1.15 BRANCH IF CHARACTER EQUAL 11-12
11.3.1.16 BRANCH IF CHARACTER NOT EQUAL 11-12
11.3.1.17 BRANCH IF CHARACTER LESS THAN 11-13
11.3.1.18 BRANCH IF CHARACTER LESS THAN OR EQUAL . . 11-13
11.3.1.19 LOOP DOWN TO ZERO 11-14
11.3.1.20 LOOP UP 11-14
11.3.1.21 ADD AND BRANCH IF LESS THAN OR EQUAL . . . 11-15
11.3.1.22 ADD AND BRANCH IF GREATER 11-15
11.3.1.23 SELF RELATIVE BRANCH 11-16
11.3.1.24 PROCEDURE OBJECT RELATIVE BRANCH 11-16
11.3.2 Integer Arithmetic and Logical Instructions . . . 11-16
11.3.2.1 CLEAR 11-16
11.3.2.2 SET 11-17
11.3.2.3 SET TO ONE 11-17
11.3.2.4 INTEGER MOVE 11-17
11.3.2.5 COMPLEMENT 11-18
11.3.2.6 AND 11-18
11.3.2.7 OR 11-18
11.3.2.8 EXCLUSIVE OR 11-19
11.3.2.9 EQUIVALENCE 11-19
11.3.2.10 NEGATE INTEGER 11-20
11.3.2.11 ABSOLUTE INTEGER 11-20
11.3.2.12 INCREMENT INTEGER WITH 2 OPERANDS 11-20
11.3.2.13 INCREMENT INTEGER 11-21
11.3.2.14 DECREMENT INTEGER WITH 2 OPERANDS 11-21
11.3.2.15 DECREMENT INTEGER 11-22

Data General Corporation
Company Confidential

11.3.2.16 ADD INTEGER WITH 2 OPERANDS 11-22
11.3.2.17 ADD INTEGER 11-22
11.3.2.18 SUBTRACT INTEGER WITH 2 OPERANDS 11-23
11.3.2.19 SUBTRACT INTEGER 11-23
11.3.2.20 MULTIPLY INTEGER WITH 2 OPERANDS 11-23
11.3.2.21 MULTIPLY INTEGER 11-24
11.3.2.22 DIVIDE INTEGER WITH 2 OPERANDS 11-24
11.3.2.23 DIVIDE INTEGER 11-25
11.3.2.24 REMAINDER INTEGER 11-25
11.3.3 Floating Point Arithmetic Instructions 11-26
11.3.3.1 NEGATE FLOATING 11-26
11.3.3.2 ABSOLUTE FLOATING 11-26
11.3.3.3 MOVE FLOATING 11-27
11.3.3.4 ADD FLOATING WITH 2 OPERANDS 11-27
11.3.3.5 ADD FLOATING 11-28
11.3.3.6 SUBTRACT FLOATING WITH 2 OPERANDS 11-28
11.3.3.7 SUBTRACT FLOATING 11-28
11.3.3.8 MULTIPLY FLOATING WITH 2 OPERANDS 11-29
11.3.3.9 MULTIPLY FLOATING 11-29
11.3.3.10 DIVIDE FLOATING 11-30
11.3.4 Character Instructions 11-30
11.3.4.1 MOVE CHARACTERS 11-30
11.3.4.2 MOVE SPACES 11-31
11.3.5 Miscellaneous Instructions 11-31
11.3.5.1 SET EXCEPTION RESPONSE 11-31
11.3.5.2 GET EXCEPTION RESPONSE 11-32
11.3.5.3 MOVE POINTER 11-32
11.3.5.4 CREATE POINTER 11-32
11.3.6 Conversion Instructions 11-33
11.3.6.1 CONVERT INTEGER TO FLOATING 11-33
11.3.6.2 CONVERT FLOATING TO INTEGER 11-33
11.3.7 Input/Output Assist Instructions 11-34
11.3.7.1 CONVERT INTEGER TO CHARACTER STRING 11-34
11.3.7.2 CONVERT CHARACTER STRING TO INTEGER 11-34
11.3.7.3 CONVERT FLOATING TO CHARACTER STRING . . . 11-35
11.3.8 Architectural Instructions 11-35
11.3.8.1 CALL 11-36
11.3.8.2 RETURN 11-36
11.3.8.3 NOP 11-36

Chapter 12--COBOL S-Language 12-1

12.1 Data Types and Their Representation 12-1
12.1.1 Integer 12-1
12.1.2 Floating Point 12-1
12.1.3 Decimal 12-2
12.1.3.1 Packed Decimal 12-2

12.1.3.2 Character Decimal 12-2
12.1.3.2.1 Unsigned 12-3
12.1.3.2.2 Separate Sign Leading/Trailing 12-3
12.1.3.2.3 Overpunched Sign Leading/Trailing 12-3
12.1.4 Character Strings 12-4
12.1.5 Translate Table 12-4
12.1.6 Boolean Vector 12-4
12.1.7 Pointers 12-4
12.2 COBOL Exceptions 12-4
12.2.1 Error Handling 12-5
12.2.2 Exceptions 12-5
12.3 COBOL Operation Definitions 12-8
12.3.1 Control Instructions 12-8
12.3.1.1 BRANCH IF ZERO 12-8
12.3.1.2 BRANCH IF NOT ZERO 12-8
12.3.1.3 BRANCH IF GREATER THAN ZERO 12-9
12.3.1.4 BRANCH IF GREATER THAN OR EQUAL TO ZERO . . 12-9
12.3.1.5 BRANCH IF LESS THAN ZERO 12-9
12.3.1.6 BRANCH IF LESS THAN OR EQUAL TO ZERO . . . 12-10
12.3.1.7 BRANCH IF INTEGER EQUAL 12-10
12.3.1.8 BRANCH IF INTEGER NOT EQUAL 12-10
12.3.1.9 BRANCH IF INTEGER LESS THAN 12-11
12.3.1.10 BRANCH IF INTEGER LESS THAN OR EQUAL . . . 12-11
12.3.1.11 BRANCH IF FLOATING EQUAL 12-12
12.3.1.12 BRANCH IF FLOATING NOT EQUAL 12-12
12.3.1.13 BRANCH IF FLOATING LESS THAN 12-12
12.3.1.14 BRANCH IF FLOATING LESS THAN OR EQUAL . . 12-13
12.3.1.15 BRANCH IF DECIMAL EQUAL 12-13
12.3.1.16 BRANCH IF DECIMAL NOT EQUAL 12-13
12.3.1.17 BRANCH IF DECIMAL LESS THAN 12-14
12.3.1.18 BRANCH IF DECIMAL LESS THAN OR EQUAL . . . 12-14
12.3.1.19 BRANCH IF DECIMAL EQUAL ZERO 12-15
12.3.1.20 BRANCH IF DECIMAL NOT EQUAL ZERO 12-15
12.3.1.21 BRANCH IF DECIMAL GREATER THAN ZERO . . . 12-15
12.3.1.22 BRANCH IF DECIMAL GREATER THAN OR EQUAL TO ZERO 12-
12.3.1.23 BRANCH IF DECIMAL LESS THAN ZERO 12-16
12.3.1.24 BRANCH IF DECIMAL LESS THAN OR EQUAL TO ZERO 12-16
12.3.1.25 BRANCH IF CHARACTER EQUAL 12-17
12.3.1.26 BRANCH IF CHARACTER NOT EQUAL 12-17
12.3.1.27 BRANCH IF CHARACTER LESS THAN 12-18
12.3.1.28 BRANCH IF CHARACTER LESS THAN OR EQUAL . . 12-18
12.3.1.29 BRANCH IF CHARACTERS TRANSLATED EQUAL . . . 12-19
12.3.1.30 BRANCH IF CHARACTERS TRANSLATED LESS THAN . 12-19
12.3.1.31 BRANCH IF CHARACTERS TRANSLATED NOT EQUAL . 12-20
12.3.1.32 BRANCH IF CHARACTERS TRANSLATED LESS THAN OR EQUAL
12.3.1.33 BRANCH IF IN SET 12-21
12.3.1.34 BRANCH IF NOT IN SET 12-21
12.3.1.35 BRANCH IF CHARACTERS SPACES 12-22

Data General Corporation
Company Confidential

12.3.1.36 BRANCH IF CHARACTERS NOT SPACES 12-22
12.3.1.37 LOOP DOWN TO ZERO 12-23
12.3.1.38 PERFORM 12-23
12.3.1.39 PERFORM END 12-23
12.3.1.40 SELF RELATIVE BRANCH 12-24
12.3.1.41 PROCEDURE OBJECT RELATIVE BRANCH 12-24
12.3.2 Integer Arithmetic Instructions 12-24
12.3.2.1 MOVE INTEGER 12-24
12.3.2.2 SET TO ONE 12-25
12.3.2.3 ABSOLUTE INTEGER 12-25
12.3.2.4 NEGATE INTEGER 12-25
12.3.2.5 INCREMENT INTEGER WITH ONE OPERAND 12-26
12.3.2.6 DECREMENT INTEGER WITH ONE OPERAND 12-26
12.3.2.7 ADD INTEGER WITH 2 OPERANDS 12-26
12.3.2.8 ADD INTEGER 12-27
12.3.2.9 SUBTRACT INTEGER WITH 2 OPERANDS 12-27
12.3.2.10 SUBTRACT INTEGER 12-27
12.3.2.11 MULTIPLY INTEGER WITH 2 OPERANDS 12-28
12.3.2.12 MULTIPLY INTEGER 12-28
12.3.2.13 DIVIDE INTEGER 12-28
12.3.2.14 SCALE INTEGER BY 10 12-29
12.3.2.15 SCALE INTEGER WITH ROUNDING 12-29
12.3.3 Floating Point Arithmetic Instructions 12-30
12.3.3.1 MOVE FLOATING 12-30
12.3.3.2 ADD FLOATING WITH 2 OPERANDS 12-30
12.3.3.3 ADD FLOATING 12-30
12.3.3.4 SUBTRACT FLOATING WITH 2 OPERANDS 12-31
12.3.3.5 SUBTRACT FLOATING 12-31
12.3.3.6 FLOATING NEGATE 12-32
12.3.3.7 MULTIPLY FLOATING WITH 2 OPERANDS 12-32
12.3.3.8 MULTIPLY FLOATING 12-32
12.3.3.9 DIVIDE FLOATING 12-33
12.3.4 Decimal Arithmetic Instructions 12-33
12.3.4.1 CLEAR DECIMAL 12-33
12.3.4.2 SET DECIMAL TO ONE 12-33
12.3.4.3 MOVE DECIMAL 12-34
12.3.4.4 NEGATE DECIMAL 12-34
12.3.4.5 INCREMENT DECIMAL WITH ONE OPERAND 12-34
12.3.4.6 DECREMENT DECIMAL WITH ONE OPERAND 12-35
12.3.4.7 ADD DECIMAL WITH 2 OPERANDS 12-35
12.3.4.8 ADD DECIMAL 12-35
12.3.4.9 SUBTRACT DECIMAL WITH 2 OPERANDS 12-36
12.3.4.10 SUBTRACT DECIMAL 12-36
12.3.4.11 MULTIPLY DECIMAL WITH 2 OPERANDS 12-36
12.3.4.12 MULTIPLY DECIMAL 12-37
12.3.4.13 DIVIDE DECIMAL 12-37
12.3.4.14 SCALE DECIMAL 12-38
12.3.4.15 SCALE DECIMAL WITH ROUNDING 12-38

Data General Corporation
Company Confidential

12.3.5 String Instructions 12-38
12.3.5.1 MOVE FROM SUBSTRING 12-39
12.3.5.2 MOVE TO SUBSTRING 12-39
12.3.5.3 MOVE SUBSTRING 12-40
12.3.5.4 SCAN SUBSTRING 12-40
12.3.5.5 MOVE CHARACTERS 12-41
12.3.5.6 MOVE SPACES 12-41
12.3.5.7 MOVE CHARACTERS TRANSLATED 12-42
12.3.5.8 EDIT CHARACTERS 12-42
12.3.5.9 EDIT NUMERIC 12-43
12.3.6 Miscellaneous Instructions 12-45
12.3.6.1 INITIALIZE 12-46
12.3.6.2 CLEAR 12-46
12.3.6.3 CONVERT INTEGER TO FLOATING 12-46
12.3.6.4 CONVERT FLOATING TO INTEGER 12-47
12.3.6.5 CONVERT INTEGER TO DECIMAL 12-47
12.3.6.6 CONVERT DECIMAL TO INTEGER 12-47
12.3.6.7 CONVERT FLOATING TO DECIMAL 12-48
12.3.6.8 CONVERT DECIMAL TO FLOATING 12-48
12.3.7 Size Error Instructions 12-48
12.3.7.1 CLEAR SIZE ERROR 12-49
12.3.7.2 SET SIZE ERROR 12-49
12.3.7.3 CHECK MOVE INTEGER 12-49
12.3.7.4 CHECK MOVE DECIMAL 12-50
12.3.7.5 BRANCH IF NO SIZE ERROR 12-50
12.3.8 Architectural Instructions 12-50
12.3.8.1 CALL 12-50
12.3.8.2 RETURN 12-51
12.3.8.3 NOP 12-51

Data General Corporation
Company Confidential

Chapter 1
Introduction

1.1 Objects

Objects are the basic units of storage in the FHP system. Each object is distinct from all other objects and is individually addressed and protected. The primary use of an object is as a container of data. Each object can contain from 0 to 2**32-1 bits where each bit is directly addressable.

1.1.1 Designation

Each object is identified by a unique 80 bit name or unique identifier (UID). UID's are unique across all FHP systems for all time; they are never reused. 32 of the 80 UID bits designate a Logical Allocation Unit (LAU), the movable unit of storage in the FHP system. This 32 bit LAU identifier is also unique across all FHP systems. The remaining 48 bits of the UID designate the object serial number (OSN). OSN's are unique within each LAU.

1.1.2 Logical Address

The directly addressable virtual memory of an FHP system allows 2**80 objects where each object is directly addressable to each of 2**32-1 bits. A logical address is composed of a combined UID, bit offset pair of 112 bits. This constitutes a single, fairly large address space which is used by all FHP systems; there is only one address space for all systems for all time. Each time the virtual memory is accessed four items must be provided: object name in the form of UID, offset of the first bit to be referenced, number of bits to be referenced and function which is usually read or write.

1.1.3 Accessing and Protection

Objects are used for two primary purposes: to store data for future processing and to store instructions. There are three fundamental operations that can be performed on an object: read bits, write bits and fetch instruction bits. Each function must be explicitly allowed for each object. This is done by attaching

read, write or execution permission to each object. A procedure object (PO), one which may serve as an instruction source, must posses the execute or E attribute. A data object may posses read (R) or write (W) attributes to allow reading or writing.

1.1.4 Procedure Objects

A procedure object is used as a source of instructions. Instructions are retrieved from a procedure object, interpreted (bit settings decoded to derive semantic intent) and appropriate action taken by an entity called an S-Interpreter. Programs stored in procedure objects control actions taken by an FHP system via S-Interpreters. Multiple, distinct instruction sets are possible giving rise to an S-Interpreter for each. Each distinct instruction set is called an S-Language.

1.1.5 Objects of Extended Type

Objects used for storing data and instructions are primitive type objects; only the three primitive operations mentioned before are possible for them. FHP allows for the construction of objects of extended type (ETO) so that data constructs of a more abstract nature can be referenced. ETO's are referenced by their UID's but arbitrary functions as well as datum designations can be defined for each type.

An ETO is the instance of a given type while an associated extended type manager (ETM) is a set of procedures which implements the extended operations of the type.

1.1.6 Object Management

In addition to the access functions, a set of functions are provided which allow for the management of objects. These include the ability to create, delete and control access to objects of both primitive and extended types.

1.2 Procedure Objects

A procedure object is like a data object object except that it can serve as an instruction source. In order to make this possible, procedure objects adhere to a specific internal structure. This structure facilitates orderly entry into the PO to the desired set of instructions, the establishment of the correct

S-Interpreter.

1.2.1 Instructions

Each instruction is composed of an 8 bit operation code (opcode) and a number of operand specifications. The semantics of each opcode is known to the appropriate S-Interpreter. Operand specifications (called syllables) follow the opcode in contiguous storage. Each syllable is 8, 12 or 16 bits long and is either a literal (the operand in the syllable itself) or is an index into a table which will completely specify the location, size and representation of the operand. The table is called the Name Table (NT) while the indexing syllable is called a Name. The Name Table defines the basic mechanism for total name resolution including scalar and array access, pre- and post-indexing and indirection.

1.2.2 Procedures

When a procedure is invoked care must be taken to insure that the correct environment is provided. This is done via a structure called the procedure environment descriptor (PED) which locates the procedure's S-Interpreter and name table. A PED can be shared by all procedures with a common environment. Each legitimate entry associated with a given PED has its own entry descriptor (ED) which locates where in the PO instruction execution is to begin. A single PO may contain multiple PED's each of which may contain multiple ED's.

Those entries which are callable from outside of the PO are defined in a gate list at the beginning of the PO. A gate is essentially an indirection to an ED somewhere in the procedure object. The size of the gate list is specified in a structure called the procedure object header and is located at the beginning of the object.

1.3 Object Protection

Access permissions are attached to each object via an Access Control List or ACL. An ACL exists for each object and lists access rights to the object for each subject. For objects of extended type ACL entries are listed in terms of subjects and the specific extended operations.

1.3.1 Subjects

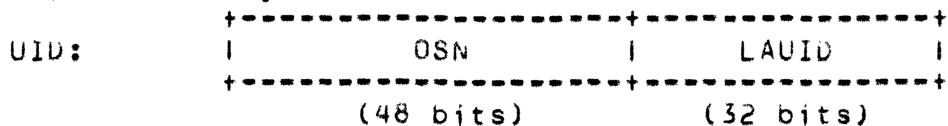
The subject is the basic unit of authority and accountability in the FHP system. At any point in time instruction execution is bound to one subject. Hence object access rights are computed in terms of the current subject. A subject is composed of four seperate authority elements: principal, process, domain and tag. The principal component is an external authority and in the simplest case is the name of the user using the system. The process component identifies the current process. The domain component of the subject allows users and system designers to tie access privileges to the procedure that is being executed. The tag component allows users to construct arbitrary protection structures.

Chapter 2
Objects

All program addressable data reside in objects. An object is a named container of data in which each datum is physically represented by a number of bits. An object can contain up to (2**32)-1 bits.

2.1 Object Designation

An object is named by an 80-bit number known as a Unique Identifier, or UID. The UID is the only name by which an object is known and is unique in both space and time -- UIDs are not reusable. In order to assure uniqueness across all FHP systems without requiring total communication between systems, a UID has two primary components: a Logical Allocation Unit Identifier (LAUID), and an Object Serial Number (OSN).



```
                +-------------------+---------------+
UID:            |        OSN        |     LAUID     |
                +-------------------+---------------+
                     (48 bits)         (32 bits)
```

A null UID consists of all zeros.

2.2 Logical Addresses

A logical address denotes a bit position in a large, two-dimensional virtual memory. This virtual memory is composed of up to 2**80 objects, each of which may hold up to (2**32)-1 bits of data. A particular object is selected by specifying its UID. Selecting a particular bit within that object is accomplished by specifying an offset into that object. An offset is an integer in the range 0 . . (2**32)-2. A logical address is a 112-bit pair (UID,OFFSET). All accesses to virtual memory involve presenting a logical starting address, a count of bits to be transferred, and the direction of transfer (read or write).

2.3 Object Protection

Objects are protected information containers. Any attempt to access their contents or attributes is mediated. The functions specified in this chapter require specific primitive access permis-

sions for their successful completion. If the subject on whose behalf the function is being performed does not have the required access permission, the function is aborted with a protection violation before any observation or modification of the object's contents or attributes is made. (Subjects are discussed in Chapter xxx.) If a function is aborted because of a protection violation and the subject does not have any primitive access permission to the object, all the invoking subject can determine is that the object does not exist or it (the subject) does not have any primitive access permission to the object. If the function is aborted because the subject does not have the specific primitive access required by the function, but it does have some primitive access permission to the object, the subject is told that the object does exist and it lacks the required primitive access permission to perform the function.

2.4 Object Accessing Functions

The most common (and fundamental) operations on objects are writing and reading the bits they contain and fetching bits from them for i-stream interpretation. These operations are invoked by s-interpreters as part of the process of interpreting s-ops. They are described here in an abstract manner.

Type Declarations

```
TYPE offset_type IS 0 .. (2 ** 32)-2;
TYPE object_size_type IS 0 .. (2 ** 32)-1;
TYPE object_identifier_type IS
RECORD
       OSN:      0 .. (2 ** 48)-1,
       LAUID:    0 .. (2 ** 32)-1
END RECORD              BOUND 80;
```

2.4.1 Write_object

Syntax/form

```
PROCEDURE write_object(
```

```
    READ ONLY object_identifier:            object_identifier_type,
    READ ONLY bit_offset:                   offset_type,
    READ ONLY bit_string:                   bit_string_type);

    PROPERTIES INTERFACE;

END PROCEDURE write_object;
```

Semantics/meaning

The bit string is written into the object, starting at bit offset.

Errors

The following errors are detected. If any occur, the function is terminated before any changes are made.

> If the object does not exist or the subject currently bound to the process executing the function does not have write access permission to the object, a write check fault is signalled.
>
> If (bit offset + LENGTH(bit string)) is greater than the object's size, an extent check fault is signalled.

2.4.2 Read_object

Syntax/form

```
PROCEDURE read_object(

    READ ONLY object_identifier:            object_identifier_type,
    READ ONLY bit_offset:                   offset_type,
    READ ONLY bit_length:                   object_size_type,

    WRITE ONLY bit_string:                  bit_string_type);

    PROPERTIES INTERFACE;

END PROCEDURE read_object;
```

Semantics/meaning

The bits from bit offset to (bit offset + bit length - 1) are returned.

Errors

The following errors are detected. If any occur, the function is terminated before any changes are made.

If the object does not exist or the subject currently bound to the process executing the function does not have read access permission to the object, a read check fault is signalled.

If (bit offset + bit length) is greater than the object's size, an extent check fault is signalled.

2.4.3 Fetch_from_object

Syntax/form

```
PROCEDURE fetch_from_object(

    READ ONLY object_identifier:        object_identifier_type,
    READ ONLY bit_offset:               offset_type,
    READ ONLY bit_length:               object_size_type,

    WRITE ONLY bit_string:              bit_string_type);

    PROPERTIES INTERFACE;

END PROCEDURE fetch_from_object;
```

Semantics/meaning

The bits from bit offset to (bit offset + bit length - 1) are fetched so that they can be interpreted as i-stream. (Read_object is used to fetch name table entries.)

Errors

The following errors are detected. If any occur, the function is terminated before any changes are made.

> If the object does not exist or the subject currently bound to the process executing the function does not have execute access permission to the object, an execution check fault is signalled.
>
> If (bit offset + bit length) is greater than the object's size, an extent check fault is signalled.

--End of Chapter--

Data General Corporation
Company Confidential

Chapter 3
Procedure Object Structure

3.1 Introduction

Procedures are realized through Procedure Objects. A procedure object consists of a Procedure Object Header, a gate list, S-instructions, name tables, and program data. Procedures callable from outside of a given procedure object are called external entry points. External entry points are called by specifying the logical address of an element within the gate list of the target procedure's procedure object (a gate).

The procedure object header begins at offset zero of a procedure object. The gate limit field of the procedure object header denotes the number of elements in the gate list.

The gate list immediately follows the procedure object header and is aligned 0 MOD (128). The gates are each 128 bits and denote procedures which may be entered from inside or outside this procedure object. These elements may be pointers to other gates or they may be entry descriptors. An entry descriptor indicates that the realization of this entry point resides in this procedure object, and locates the information necessary to invoke the procedure.

The remaining items that may be contained in a procedure object may appear anywhere within the object. In addition to a gate list, a procedure object may contain internal entry descriptors, procedure environment descriptors, name tables, S-instructions, and literal data.

3.2 Procedure Object Header

The procedure object header contains information pertinent to the entire procedure object.

procedure object header

```
+-----------------+-------+-------+-------------+-------------+
| procedure_object | flags | gate  | reserved_for | reserved -  |
|   _info_offset   |  fmt  | limit | _binder      |     mbz     |
+0--------------31+32---47+48---63+64---------95+96--------127+
```

* procedure_object_info_offset (32 bits) - This offset field contains the location--relative to the start of this procedure object--of the procedure_object_info_type which locates the arg_info_array (see discussion below).

* flags_and_format (16 bits) - These bits define the flags and format field (see below).

* gate_limit (16 bits) - This field contains an unsigned integer value specifying the number of gates in the gate list.

* reserved (32 bits) - This field is ignored; it is available for use by compilers, binders, etc.

* reserved - mbz (32 bits) - This field is reserved for future use -- Each bit must be zero (0).

flags and format field

```
+---------------------------+---+-------------------------+
|       version number      |NR |     format_code = 65    |
|                           |= 1|                         |
+32-----------------------39+40-+41---------------------47+
```

* version (8 bits) - This field contains the integer value of the procedure object structure version number.

* NR (1 bit) - This bit must be set (bit = 1), indicating this is a nonresolvable pointer.

* format code (7 bits) - This field contains an integer value that is the format code. For procedure object headers, this value is sixty-five (65) [decimal].

3.3 Entry Descriptor

The Entry Descriptor contains information pertinent to a particular procedure. The entry descriptor denotes an external entry point if it lies within the gate list; otherwise, it denotes a procedure callable only from a procedure within the containing procedure object. Entry descriptors are stored within procedure objects such that the address of bit 0 is 0 mod(128).

entry descriptor

```
+--------------+-------+-------+-------------+--------------+
| initial_PC_  | flags | resrvd| procedure_  | initial_frame |
|    offset    |  and  |  mbz  | environment_ |    _size     |
|              | format|       |    offset   |              |
+0----------31+32---47+48---63+64---------95+96---------127+
```

* initial_PC_offset (32 bits) - The offset, relative to the Initial PBP, of the first S-Instruction for this procedure (see the Procedure Environment Descriptor discussion below).

* flags and formats (16 bits) - The format code and flags field (see below).

* reserved, must be zero (16 bits)

* procedure_environment_offset (32 bits) - The location--relative to the start of this procedure object--of a Procedure Environment Descriptor defining the execution environment and S-Interpreter required for this procedure.

* initial_frame_size (32 bits) - The number of bits required for this procedure's local automatic data.

```
| entry descriptor -- flags and format
|
| +---+---+---+---+---------+---+-------------------------------+
| |   |   |   |   | version |NR |       format code = 64        |
| |  ||  ||  ||  ||         | =1|                               |
| +32|+33|+34|+35|+36------39+40-+41--------------------------47+
|     |   |   |   |
|     |   |   |   +-> Reserved
|     |   |   |
|     |   |   +-> do_not_check_access flag
|     |   |
|     |   +-> SEPP flag
|     |
|     +-> access_info_present flag
|
```

* access_info_present (1 bit) - If set, this bit indicates that an access_modes array is present for this gate (see below). This flag is examined only for entry descriptors which appear in gate lists.

* SEPP flag (1 bit) - If set indicates the SEB must be changed upon procedure entry.

* do_not_check_access flag (1 bit) - If set bypass access checking. See Section 3.4.

* reserved (1 bit) - The semantics of these bits are defined by the operating system and S-Languages.

* version (4 bits) - This is an integer in the range of 0 .. 15 that identifies the version of the structure.

* format code (7 bits) - This is an integer value in the range of 0..127. For entry descriptors, this value is sixty-four (64) [decimal].

3.4 Arg_info_array

The procedure object header locates a record called the procedure object info type (via the procedure_object_info_offset field). This record contains the argument information array offset (arg_info_array_offset). This offset locates an array of entries which parallels the gate list and specifies information required to validate access to arguments.

10:44:37
9/Nov/79
Rev. 1

procedure object info type

```
+-----------------------------+
|    arg_info_array_offset    |
+0--------------------------31+
```

* arg_info_array_offset (32 bits) - The location--relative to the start of this procedure object--of the arg_info_array (see below).

Each entry point to a domain may specify information describing the access the caller must have to the actual parameters he is passing. Since every gate in the gate list is a potential domain entry point, every gate potentially specifies this information.

Argument access information is specified via a vector, called the arg_info_array, which parallels the gate list and is located by the arg_info_array_offset field of the procedure_object_info_type. An external entry point's arg_info_array element is accessed by using the gate number as a subscript into the arg_info_array. (The gate number is the offset portion of the procedure entry descriptor's logical address, divided by 128.)

Data General Corporation
Company Confidential

arg_info_array_element (long variant)

```
| +-+-------------+---------------+-----------------------------+
| |0| required_#_ | num_formals   |  access_modes_array_offset  |
| | | arguments   |               |                             |
| +0+1----------15+16-----------31-32---------------------------63+
|
```

arg_info_array_element (short variant)

```
+-+-------------+-----------------------------------------------+
|1| num_formals |   access_modes_array                          |
| |             |      (array of up to 12 access_modes_types)   |
+0+1----------15+16-------------------------------------------63+
```

access_modes_type (primitive access variant)

```
+---+---+---+---+
| 0 | E | R | W |
|   |   |   |   |
+0--+1--+2--+3--+
```

access_modes_type

```
+---+-----------+
| 1 | reserved  |
|   |  (MBZ)    |
+0--+1---------3+
```

The fields of an arg_info_array_element have the following meanings:

* Bit 0 -- This bit identifies format of this arg_info_array_element. A value of zero indicates this is a long variant element, a value of one indicates this is a short variant element.

* number_of_required_args (15 bits, long variant) -- Minimum argument count required. See text.

* num_formals (16 bits in long variant, 15 bits in short variant) This field specifies the number of actual parameters this procedure expects as an unsigned integer. A long variant may specify from zero to 65,535 actuals; a short variant may specify from zero to twelve actuals.

* access_modes_array_offset (32 bits, long variant) -- This field locates an array of access_modes_type describing the access the caller is required to have to each of the

actuals he is passing. The access_modes_array_offset field is an offset, relative to the start of the procedure object.

* access_modes_array (48 bits, short variant) -- This field contains access information for up to twelve actual arguments, specified as an array of access_modes_type.

The fields of an access_modes_type have the following meanings:

* Bit 0 (1 bit) - This bit identifies the structure as specifying primitive access (value = 0) or extended access (value = 1).

* E, R, W (3 bits, primitive variant) - These bits specify that the caller must have execute, read, and write access to this actual (respectively).

* ignored (3 bits, extended variant) - these bits are reserved.

If the access_info_present flag of this procedure's entry descriptor is not set, the arg_info_array and access_modes_array are ignored. However, if this flag is set, the number of actuals being passed is verified as being greater than or equal to the number of arguments required by the arg_info_array_element. For the short format this is num_formals, and for the long format required_#_arguments. If the test fails, INVALID_NUMBER_OF_ACTUALS is signalled.

The caller's access to the first min(num_formals, actuals_supplied) arguments is verified to include the procedure's access_modes_type array. Access_modes_type array elements apply to argument pointers in order of increasing negative displacement from FP (e.g., access_modes_array [1] corresponds to the argument pointer at FP@[-1], etc.). An access_modes_type may specify primitive execute, read, and write access (access_modes_type primitive variant).

For the remainder of the actuals supplied, a check is performed to verify that the caller has R,W access to the actual.

Any access check that fails causes TROJAN_HORSE_ARGUMENT to be signalled.

10:44:37
9/Nov/79
Rev. 1

3.5 Procedure Environment Descriptor

A Procedure Environment Descriptor (PED) identifies the instances of static data and procedure storage and the Name Table shared by activations of a (set of) source language procedure(s). A single PED is shared by all procedures which share that portion of the environment. The PED also specifies the S-Interpreter required for execution of those procedures.

Procedure Environment Descriptor

```
| +--------------+-------+-------+----------------------------+
| | reserved_for  | flags |largest|     reserved -            |
| |   _compiler   |  "k"  | name  |              must be zero |
| +0-----------31-32---47+48---63-64-------------------------127+
| |    S-Interpreter Pointer (SIP)                             |
  |                                                            |
  +128-------------------------------------------------------255+
  |    Name Table Pointer (NTP)                                |
  |                                                            |
  +256-------------------------------------------------------383+
| |    Static Data Area Pointer (SDAP)                         |
| |                                                            |
  +384-------------------------------------------------------511+
  |    Initial Procedure Base Pointer (PBP)                    |
  |                                                            |
  +512-------------------------------------------------------639+
  |    S-Interpreter Environment Prototype Pointer (SEPP)      |
  |                                                            |
  +640_______________________________________________________767+
```

* reserved_for_compiler (32 bits) - These bits are uninterpreted. This field is available for use by compilers and debuggers. For example, they might define structures here to locate symbol tables and schemata.

* flags and "k" (16 bits) - These bits are the flags and format field (see below).

* largest name (16 bits) - This field specifies an unsigned integer value in the range of 0 .. ((2 ** 16) -1). This value represents size of Name Table in multiples of 64 bits.

* reserved (64 bits) - These bits are reserved for future use -- Each bit must be zero.

* SIP (128 bits) - This field is a pointer identifying the S-Interpreter to be used in interpreting this procedure's instruction stream.

* NTP (128 bits) - This field specifies a pointer locating the start of this procedure's name table.

* SDAP (128 bits) - This field specifies a pointer which locates the procedure's static data area. This pointer can be null, indicating this procedure requires no static data.

* Initial PBP (128 bits) - This field specifies the location of the procedure storage implementing the procedure. This pointer must be object relative. All offsets relative to the PBP ABR defined in the preceeding sections are relative to this pointer.

* SEPP (128 bits) - This field specifies a pointer which locates the procedure's S-Interpreter Environment Block (SEB).

flags and "k" field

```
+-----+---------------------------+---+------------------------------+
| "k" |          reserved         |NR |       format code = 66       |
|     |                           | =1|                              |
+32-33+34---------------------39+40-+41----------------------------47+
```

* "k" - encoded operand syllable size :
  00 -> k = 8
  01 -> k = 12
  10 -> k = 16
  11 -> reserved

* reserved (6 bits) - These bits are reserved for system use.

* format code (7 bits) - This field contains an integer value in the range of 0..127 which is the format code. For procedure environment descriptors, this code is sixty-six (66) (decimal).

10:44:37
9/Nov/79
Rev. 1

3.6 Instructions

An instruction must convey to the processor information regarding transformation of data or alteration of the target machine's state. This information includes the following:

* the operation to be performed,

* the number of operands,

* the location, size, and representation (type) of operands and

* the direction of data flow.

Location, length, and representation information for operands are grouped in a table called the Name Table (NT). A process' current name table is located by its current Name Table Pointer (NTP). The way source references are encoded in the NT is illustrated in the the Name Table chapter. Resolving NT references to logical addresses is discussed in the Chapter on Name Resolution.

3.6.1 Opcode

Each instruction begins with an 8-bit operation specification (the opcode). The opcode implies the number of operands, which operands are sources/sinks, and the operation to be performed. In many cases, the type and containerization of operands are also conveyed by the opcode.

The S-Language may define up to 256 valid opcodes. An attempt to execute an S-Instruction with a reserved opcode causes an INVALID_S_OP condition to be signalled.

3.6.2 Operands

Each opcode may be followed by some number of operand specifications (represented by "syllables"). Each operand syllable is "k" bits long, where "k" may be eight, twelve or sixteen bits. SPEINI does not support k=12. The value of k is obtained from the Procedure Environment Descriptor of the current procedure. The FHP Architecture dictates the kinds of operand specifications which may appear; the S-Machine defines instruction syntax and defines the order and intermixing of operand syllables.

10:44:37
9/Nov/79
Rev. 1

Valid operand syllables are:

* Name syllable--an operand reference name.

* Literal syllable--an immediate syllable representing a literal value.

3.6.2.1 Name Syllables

All variable references are made via Name syllables. A Name is zero-extended on the left to sixteen bits and used as an index into the current Name Table (see Name Table discussion).

3.6.2.2 Literals

Literal operands have an implicit length of "k" bits and no type; they may be manipulated by the interpreter in any desired manner.

3.6.2.3 Relative Branches

A relative branch address is specified via a literal syllable. In this case, the literal syllable is interpreted as a signed offset relative to the start of the branch instruction (note that the current PC always points to the instruction being interpreted: the current instruction). The relative offset is right justified and sign-extended on the left to 32 bits, multiplied by the greatest common divisor (GCD) of 8 and k, (i.e., 8 for k=8 or 16, and 4 for k=12), and added (modulo 2**32) to the offset portion of the current PC. This value replaces the current PC offset.

3.6.2.4 Absolute Branches

An absolute intra-procedure object branch is made by evaluating an offset name syllable. The result of this evaluation is interpreted as an unsigned, bit-granular offset value. The offset value is extended/truncated to 32 bits as dictated by the offset's Name Table Entry. This result is added to PBP.offset and replaces the offset portion of the current PC.

3.7 Pointers

A pointer represents or implies a logical address as an offset into a specific object (a <UID, offset>). In the simplest case, the pointer directly represents both the object and the offset (this is called a resolvable pointer). Resolution of this pointer type requires no intervention.

Alternatively, a pointer may require intervention for its conversion to a logical address, or the pointer may represent a structure more complex than a simple logical address. Such pointers are termed nonresolvable pointers and any attempt to apply pointer resolution to them results in a nonresolvable pointer fault being signalled. A return from a nonresolvable pointer fault causes the pointer to be refetched and reinterpreted. Fault bits are not ignored, and the nonresolvable pointer fault may recur. The object may be implied as the object containing the pointer structure (called intra_object pointers or object relative pointers) or explicitly mentioned as a UID (called inter_object pointers or UID pointers).

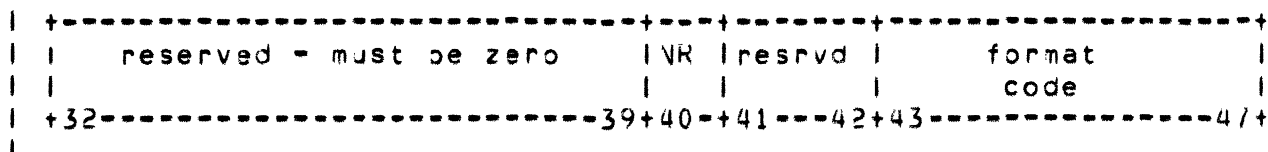
When a non-resolvable pointer is used in conjunction with either an inter- or intra-object format an associative pointer is indicated. When resolve attempts to indirect through an associative pointer, the current stack's Associative Address Table is searched for an entry whose tag matches the address specified by the associative pointer. For an absolute associative pointer, this address is determined from the UID and offset portions of the pointer. For an object relative associative pointer only the offset portion is relevant; the UID used is that of the object containing the pointer. If an association is found in the AAT the resulting address is used by the resolve function. Otherwise a NO_ASSOCIATED_ADDRESS condition is signalled.

General Format

```
+---------------+-------+--------------------------------------+
|  offset       | flags | format = 0,2: undefined and ignored  |
|               | format| format = 1:   unique identifier      |
+0-----------31+32---47+48---------------------------------127+
```

10:44:37
9/Nov/79
Rev. 1

Flags and Format Field



offset (0..31)
an offset relative to the start of the object specified by the format code

Flags and Format field (bits 32..47):

reserved - (32..39)
these bits are not defined at present -- Each bit must be zero (0).

NR (40)

A value of zero indicates this structure is a resolvable pointer.

A value of one indicates that the resolution of the pointer to a logical address requires intervention. In the case where NR = 1 in conjunction with pointer formats 1 or 2 an associative pointer is indicated.

resrvd (41..42) These bits are reserved, and must be zero, or results are unpredictable.

format code (43..47)
This format code identifies the type of pointer. This type is expressed by a integer value in the range of (0 .. 31). The current codes are:

Integer value 0 - null pointer.
All other fields of this pointer are ignored. When this format is encountered during pointer resolve, a null logical address is produced (a UID of zero). Any attempt to reference thru a null logical address results in a null pointer fault being signalled.

Integer value 1 - UID pointer
The offset is relative to the start of the object

10:44:37
9/Nov/79
Rev. 1

whose UID is contained in bits 48..127 of this structure.

Integer value 2 - object-relative pointer.
The offset is relative to the start of the object containing this datum.

Integer values in the range (3 .. 31) - Reserved for future expansion.
Specification of one of these values causes __an invalid_pointer_fault.__

3.8 Associative Addressing

A facility for associating one address with another is provided using special types of pointers. The translation between an address and its associated address is stored in a table located by the process' current stack header. If the resolve function finds no translation for the address in an associative pointer then it signals no_associative_address. The no_associated_address condition handler can then create and return an association.

3.8.1 The Associated Address Table

The table used to contain the current set of associations is the Associated Address Table (AAT). Each user stack contains a pointer to an AAT in its stack header. The current version number of the AAT is 1. It is AAT is organized as follows:

```
TYPE Associated_Address_Table IS
  RECORD
     version:          version_type BOUND 32,
     size_of_table:          AAT_index_type BOUND 32,
     padding:          VOID BOUND 64,
     current_adr:          POINTER BOUND 128,
     entry:          ARRAY [0..size_of_table-1] OF
                Associated_Address_Table_Entry
   END RECORD BOUND;


TYPE Associated_Address_Table_Entry IS
  RECORD
     tag:          AATE_tag,
     associated_address:      POINTER
  END RECORD;
```

10:44:37
9/Nov/79
Rev. 1

```
  TYPE AATE_tag IS
    RECORD
        offset:                    0..(2^32)-1,
       flags:        VOID BOUND 16,
       UID:        0..(2^80)-1
    END RECORD
```

Note: associated_address can be any format of pointer (including null).

3.8.2 get_associated_address

This function is (logically) invoked by resolve to find the associated address for a faulting associative pointer (for the current stack environment). The existence of this function is architectural, its implementation is not.

Syntax/form

```
PROCEDURE get_associated_address (
     READ ONLY  address:           POINTER,
     WRITE ONLY associated_address:    POINTER,
     WRITE ONLY association_exists:    BOOLEAN);
     PROPERTIES CONTRACT;
END PROCEDURE get_associated_address;
```

Semantics

The AAT associated with the caller's stack is searched. If a translation exists for the address specified, association_exists is set to TRUE and the associated address is returned in associated_address. If no translation exists for the given address, association_exists is set to FALSE and the null pointer is returned in associated_address.

Operation

An address, ADDR, is derived by resolving the pointer in the address argument. An index is derived as:

```
       input1 [0:6] := 0
       input1 [7:31] := ADDR.offset [0:24]
```

```
input2 [0:6] := 0
input2 [7:15] := ADDR.UID [39:47]
input2 [16:31] := ADDR.UID [0:15]

hash := MOD ( XOR ( input1, input2), AAT.size_of_table)
```

Note: all values are to be treated as unsigned integers.
X [i:j] means bits i through j of X, where bit number
0 is the left most bit of X.

The AAT associated with the current stack is searched circularly starting at the derived index until either a matching entry is found, in which case association_exists is set to TRUE and the associated address in the table is returned, or a null entry is found in the AAT (or all entries have been examined), in which case association_exists is set to FALSE and the null pointer is returned. The match is performed by comparing the UID and offset field of the tag component of an AAT entry with the UID and offset field of the address specified in the call. If both of these fields match, then the correct entry has been located.

An entry with a 0 UID and (2^32-1) offset in the tag field is a null entry.

3.9 Referencing Static Data

During architectural call the associated address mechanism is used to obtain a pointer to the impure copy of the target procedure's static data. This is accomplished by having compilers place an associative pointer to the procedure's static data prototype in the PED.SDAP. When architectural call enters a new procedure, it resolves this pointer and places the result in the SDP ABR. Resolve invokes the get_associated_address function. If a matching entry is found, then a copy of the procedure's static data has been made for this stack and is located by the associated address in the matching entry. In this case, the associated address is returned to resolve which. If no copy has been made, a no_associated_address condition is signalled.

The kernel creates a zero entry AAT for each user stack. The initial_no_associated_address_handler pointer in the target domain's domain object is copied into the no_associated_address_handler_ptr in the new domain_environment_frame. At this point the domain is on its own.

The (non-kernel supplied) no_associated_address handler must be capable of creating an associated data region, allocating a larger AAT, and copying static data all without taking a no_associated_address fault. [Note: if a static data prototype requires active initialization, the initialization procedure must not itself require active static data initialization or call the procedure whose's static is being initialized.] As stated earlier, the no_associated_address_handler_ptr is copied into the domain_environment_frame so that the full signal mechanism can be avoided. Without this provision, a signaller could not use any static data.

--End of Chapter--

Chapter 4
Architectural Base Registers

Three architectural base registers (ABR's) facilitate addressing. These registers contain logical addresses which locate portions of data space. ABR's are maintained as part of the process' macro state as described in Section 6.5.

4.1.1 Frame Pointer

The Frame Pointer (FP) points to the base of the current procedure activation record. An activation record is created and FP established when a new procedure is entered via the CALL instruction. Negative addressing relative to FP locates procedure parameter pointers. Positive addressing relative to FP locates local data used by the procedure for the current activation. The initial_frame_size in the procedure's entry descriptor (ED) defines the amount of local data allocated to the procedure.

4.1.2 Static Data Pointer

The Static Data Pointer (SDP) is established upon procedure entry. It locates data used by the procedure during a previous activation. If there was no previous activation then procedure entry attempts to establish SDP using the static_prototype_pointer field in the Procedure Environment Descriptor (PED) by signalling a STATIC_DATA fault.

4.1.3 Procedure Base Pointer

The Procedure Base Pointer (PBP) is established upon procedure entry by using the Procedure_Base_Pointer field in the PED. It is useful for addressing program constants and initial values.

--End of Chapter--

Chapter 5
Name Table and Name Resolution

The Name Table is used for all variable references. A Name syllable is used as an index into the current Name Table to locate a Name Table entry. A Name Table entry represents an operand's location, size, and data representation. Reference structure (indirection, pre and post indexing) is reflected in the Name Table; exactly one operand syllable per operand is required in any instruction.

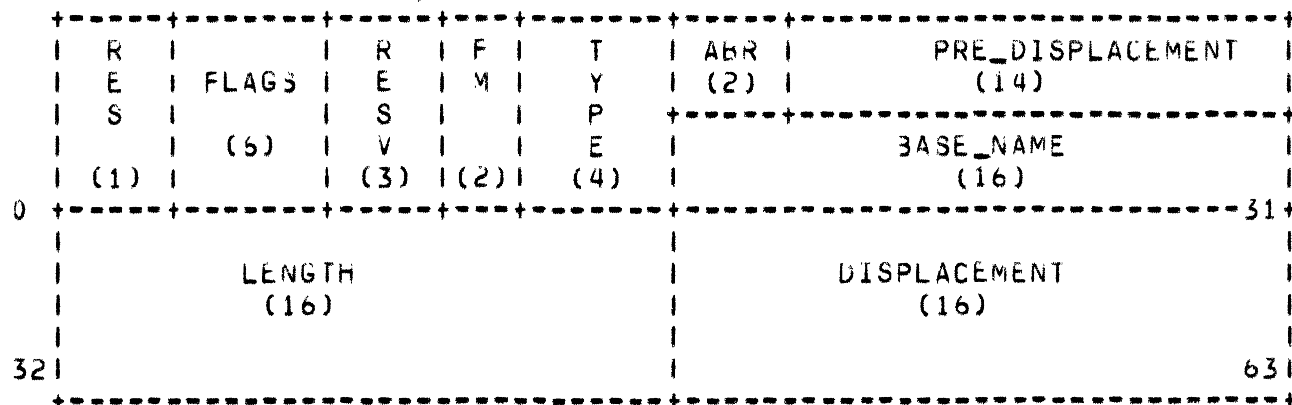
Operand syllables appear in three sizes: 8, 12, and 16 bits. The current operand syllable size (denoted by "k"), the location of the current Name Table, and the largest name represented in the current Name Table (denoted by "N"), are specified in the Procedure Environment Descriptor associated with each procedure using that Name Table. These parameters remain fixed until the Procedure Environment Descriptor changes. The actual number of Name Table Entries is always less than or equal to 2**16. Only names zero through MIN(N,2**k-1) are accessible via Name Syllables in the I-Stream: this defines the I-Stream Name Scope. Any attempt to reference a name larger than the largest name represented in current Name Table generates a NAME_OUT_OF_RANGE exception condition.

5.1 Name Table Entries and Name Table Fields

A Name Table Entry (NTE) always "resolves" to a <logical address, length, fetch mode, type> 4-tuple or "evaluates" to a <value, length, type> triple. The logical address is a <UID, offset> while the length is a 32-bit unsigned literal denoting a true length (i.e., a length of zero indicates no bits are to be processed). Unless otherwise specified, an NTE field is 16 bits long.

Name Table Entries may appear in two sizes: a 64-bit short NTE and a 128-bit long NTE. The short NTE is exactly the first 64 bits of a long NTE. The short format provides information required to resolve all operand references except array references and data values requiring a displacement of more than 16 bits. The extension (i.e., final 64 bits of a long NTE) provides an additional 16 bits of displacement and information required to access arrays. The format of the Name Table Entry is illustrated below:

Short Name Table Entry



TRAP/FLAGS[*]/RESERVED : 1=ASSERTED

```
[0]: RESERVED-MUST BE ZERO
[1]: LONG NTE
[2]: LENGTH_IS_NAME              FM[**]=FETCH_MODE:
[3]: BASE_IS_NAME
[4]: BASE_INDIRECT                 [00]: Right justify, zero fill
[5]: VECTOR                        [01]: Right justify, sign fill
[6]: IES_IS_A_NAME                 [10]: Left justify, zero fill
[7]: RESERVED-MUST BE ZERO         [11]: Left justify,
[8]: RESERVED-MUST BE ZERO                    ASCII space fill
[9]: RESERVED-MUST BE ZERO
```

Name Table Extension

```
  +-----------------------------+------------------------------+
  |                             |                              |
  |                             |                              |
  |        DISPLACEMENT         |          INDEX_NAME          |
  |            (16)             |             (16)             |
  |                             |                              |
64+---------------------------79+80--------------------------95+
  |                             |                              |
  |                             |                              |
  |          RESERVED           |  INTER_ELEMENT_SPACING (IES) |
  |            (16)             |             (16)             |
  |                             |                              |
96+--------------------------111+112------------------------127+
```

BASE Field (16..31)

Each address calculation begins with a UID, offset called the Base. The source of this UID, offset is specified by the BASE field of the NTE. The BASE Field may contain two subfields:

16:9:14
22/Oct/79
Rev. 1

ABR Field (16..17)
The two bit Architectural Base Register Field (ABR) specifies one of three architectural base registers whose contents are used to begin the logical address calculation. An ABR value of "11" is reserved.

PRE-DISPLACEMENT Field (18..31)
The 14 bit PRE-DISPLACEMENT Field represents a 128-bit granular two's complement value used in conjunction with ABR indirection, or zero.

The BASE Field may also represent the name of another NTE, in which case it is a single 16 bit field with no subfields.

All address calculations are base/displacement calculations. Two displacements may be included in the NTE, one called DISPLACEMENT and one called INDEX_NAME.

DISPLACEMENT Field (48..63)
The displacement field is specified as a signed literal to be added to the offset portion of Base. It may be specified as a 16-bit literal in a short NTE or as a 32-bit literal, where the high order 16 bits are obtained from the NTE Extension (bits 64..79).

INDEX_NAME Field (80..95)
The second displacement field, called the INDEX_NAME field, names an NTE whose resolution and subsequent evaluation yields a value to be used as a subscript.

IES Field (96..111)
The Inter-element Spacing (IES) is an unsigned integer value denoting the granularity of the subscript. It indicates the difference between the starting bit addresses of two successive vector elements. It is specified as a 16 bit immediate value, or the name of an unsigned integer valued datum.

LENGTH Field (32..47)
This field denotes the true number of bits which represent this operand value, i.e., the number of bits which must be fetched or stored. A length of zero indicates no bits are fetched or stored, and so forth. It is specified as a 16 bit immediate value, or the name of an integer valued datum.

TYPE Field (12..15)

The TYPE field is a 4-bit literal field implying an S-Interpreter-defined data representation. It is meaningful only to the S-Interpreter.

FETCH_MODE Field (FM) (10..11)

The two bit FETCH_MODE Field specifies the manner in which data is aligned and padded/truncated upon fetch/store (i.e., the containerization of a value accessed within an uncontainerized, bit-addressable virtual memory).

00 right justify and zero fill/truncate on the left

01 right justify and sign fill/truncate on the left

10 left justify and zero fill/truncate on the right

11 left justify and ASCII space fill/truncate on the right

A FETCH_MODE of left justify, ASCII space fill specified for an item whose length is not divisible by eight is undefined and results are unpredictable.

FLAGS Field (1..6)

The FLAGS field of the NTE controls the way the other fields are resolved, determining in particular whether a field is a value or a Name (which denotes another NTE).

1) LONG_NTE - indicates whether this is a short (64 bit) NTE or a long (128) bit NTE.

2) LENGTH_IS_A_NAME - indicates whether the length field is an immediate value (=0) or a name (=1).

3) BASE_IS_A_NAME - indicates whether the BASE field specifies an ABR (=0) or a name (=1). This field is used in conjunction with the BASE_INDIRECT flag in directing BASE address formation.

=Ø ABR
=1 BASE_NAME

4) BASE_INDIRECT - when BASE_IS_A_NAME=0, this flag indicates if pre-displacement is required (=1) or not (=0). If predisplacement is not required, the PRE_DISPLACEMENT field must be zero, or results are unpredictable.

16:9:14
22/Oct/79
Rev. 1

When BASE_IS_A_NAME=1, this flag indicates whether the resolved base name is to be used as an address (=0), or used to fetch a pointer (=1).

5) VECTOR - indicates whether this NTE describes a VECTOR reference (=1) or not (=0). If this flag = 1, LONG must be asserted.

6) IES_IS_A_NAME - indicates whether the IES is an immediate (=0) or a name (=1). This flag is meaningful only if VECTOR = 1.

7) The remaining flag bits are reserved, and must be zero.

5.2 Address Resolution

A name obtained from the I-Stream is used as an index into the Name Table. Ultimately, this name yields, through resolution of the structure which starts at the designated Name Table Entry, the length of the operand, its type, its fetch mode, and its starting logical address. Note that calculation of this address may require the resolution of other Name Table Entries.

One Name Table Entry may describe:

* a local scalar reference (i.e., a scalar reference relative to an Architectural Base Register),

* a scalar parameter or linked reference (i.e., a scalar reference relative to an argument pointer or linkage pointer referenced by predisplacement of an ABR),

* a vector reference requiring the resolution of an index name,

* a pointer dereference (i.e., a record or record item whose location is obtained by resolving a named base pointer), or

* a "based vector" (i.e., a vector whose base address is obtained by resolving a named base pointer).

* most combinations of the above

16:9:14
22/Oct/79
Rev. 1

5.2.1 Resolution of a Name Table Entry

A Name Table Entry is resolved by resolving each of its fields, and then combining certain of these resolutions. The operand name is used as an index into the current Name Table to fetch a 64-bit NTE. If an extension is required (indicated by the LONG flag being asserted), it is fetched from the next contiguous 64 bits.

5.2.2 BASE Field

The BASE Field specifies the BASE portion of a base/displacement logical address calculation.

All arithmetic on offsets is 32 bit unsigned arithmetic (modulo 2**32). Negative two-s complement values are treated as large, positive numbers and wrap around zero. Assertion of the BASE_IS_A_NAME Flag indicates the entire BASE Field names a variable reference (in data space). The Base Name is resolved as any other name (involving recursion of the Name Resolve algorithm). If the BASE_INDIRECT Flag is NOT asserted, the specified Base Name is resolved to a logical address only. This logical address is used as the Base by the referencing NTE. This mode is useful in the resolution of multidimensional arrays. If the BASE_INDIRECT Flag IS asserted, the specified BASE_NAME is resolved and evaluated (i.e., the resolved name is used to retrieve a pointer value from data space). The fetch of this pointer value is controlled by its NTE. The UID, offset of the retrieved pointer value is then used as the Base by the referencing NTE.

Note that Base Name may denote a structure of NTEs. The above discussion of BASE_INDIRECT applies only to the result of resolution/evaluation of the entire structure. The result of resolution/evaluation of a named base is assumed to be pointer-valued.

5.3 DISPLACEMENT Field

The DISPLACEMENT Field is always a literal in the NTE and is always bit granular. If the LONG Flag is NOT asserted, this is a short NTE and the 16-bit DISPLACEMENT Field is right justified and sign filled on the left to 32 bits. If the LONG Flag IS asserted, this is a long NTE, and the 16-bit DISPLACEMENT Field of the short NTE is right justified and extended to 32 bits by concatenating on the left the 16-bit DISPLACEMENT Field of the NTE Extension.

The resolved 32-bit displacement is added to the offset portion of the Base.

5.4 INDEX_NAME Field

The INDEX_NAME Field appears in the NTE Extension and is resolved only if the VECTOR Flag is asserted. An NTE whose VECTOR Flag is asserted and whose LONG Flag is not asserted is an INCONSISTENT NAME TABLE ENTRY and generates an exception condition.

The INDEX_NAME Field always names an operand reference specifying an element-granular subscript. The index name is resolved and evaluated. The least significant 32 bits of the returned value are used as the subscript. Note that index name may denote a structure of NTEs, whose resolution and evaluation yields the desired subscript value. Alignment is dictated by the named reference's NTE.

The INDEX_NAME Field is assumed to name an integer valued subscript reference. Specification of other than an integer valued subscript is undefined, and results are unpredictable.

5.5 IES Field

The IES Field is an unsigned integer denoting the granularity of the subscript referred to by index name. The IES Field may specify a 16 bit literal (IES_IS_A_NAME not asserted) or the name of an unsigned integer-valued reference (IES_IS_A_NAME asserted). If IES_IS_A_NAME, IES name is resolved and evaluated; the least significant 32 bits of the returned value are taken as the true IES of the vector operand. A literal IES is right justified and zero filled on the left to 32 bits. The retrieved subscript is multiplied (modulo 2**32) by IES, yielding a 32-bit, bit-granular index value. This index value is added to the offset portion of the Base Pointer. <u>Results of interpreting an IES Name which denotes other than an unsigned integer valued reference are unpredictable.</u>

IES appears in the NTE Extension and is processed only when both VECTOR and LONG Flags are asserted.

5.6 LENGTH Field

The LENGTH Field may specify a 16 bit literal (LENGTH_IS_A_NAME not asserted) or the name of an integer-valued

reference (LENGTH_IS_A_NAME asserted). If LENGTH_IS_A_NAME, length name is resolved and evaluated; the least significant 32 bits of the returned value are taken as the true length of the referenced operand. If this value is not an unsigned integer, the results are undefined and unpredictable. A literal length is right justified and zero filled on the left to 32 bits. Lengths (either literal or named) are always treated as unsigned quantities.

5.7 Name Table Entry Evaluation

At this point, the NTE has been resolved to a logical address, fetch mode, and length. The TYPE Field is treated as a 4-bit literal and made available to the instruction semantic. The resolved NTE is used by the instruction semantic to evaluate the operand reference (i.e., fetch source operands, store destination operands).

--End of Chapter--

Chapter 6
Tracing Facility

The tracing facility provides support for debugging and performance evaluation without modification or recompilation of procedure objects. Tracing is controlled by a three level data structure in a process' data space on a per stack basis. Since there is at least one stack per process-per domain, tracing is at least a per process-per domain phenomenon.

Tracable events include:

* S-Op Execution

* Name References

* Procedure Call/Return

* Nonlocal Goto

* Data References

When a traced event occurs, a signal is generated naming the event's class. A handler is invoked on the traced stack to deal with the event (e.g., inform the user via his console, update a trace data base). A return by the handler continues the traced operation. The handler receives a parameter indicating the specific event which occurred and a pointer to the environment (i.e., stack frame) in which the traced event occurred. The handler has no special privileges beyond those of the invoking domain. The handler may access trace state via the macrostate accessing interfaces defined later in this chapter. When the trace condition handler and the signaller return, trace data structures are reevaluated and the appropriate trace modes are enabled before resuming the traced event.

6.1 Trace Data Structures

Tracing is controlled by multi-level data structures. Events are signalled when a search of a particular trace structure indicates a traced event is occurring.

In the following structures, all offsets are 32 bit unsigned integers denoting the location of a bit relative to the start of an

object (i.e., object relative offsets).

6.1.1 Trace Pointer

The trace data structure for a particular stack is located via the Trace Pointer, a 128-bit general pointer residing internal to the system. A null trace pointer indicates no tracing is being done for that stack.

A new trace environment is established via a call provided by the kernel operating system. Trace state is reevaluated on each call and return. An existing trace environment is modified by nullifying the current trace pointer, modifying the trace data structures, and establishing a new trace pointer locating the modified structure.

The data structure located by the trace pointer must reside in a single object. This object need not be dedicated to tracing, however. Modifying the trace tables associated with an active stack produces unpredictable results.

6.1.2 Trace Table Header

The trace pointer locates a Trace Table Header, the first level of the trace data structure. Five classes of trace events are defined.

Trace Table Header

```
                         0                 31 32 33          47
                         +-------------------+--+-------------+
Version                  | version (=0)      |1 |classes (5) |     0..47
                         +-------------------+--+-------------+
S-Op Fetch               | trace table offset|0 |entry count |    48..95
                         +-------------------+--+-------------+
Name Resolve/Eval        | trace table offset|0 |entry count |    96..143
                         +-------------------+--+-------------+
Procedure Transit.       | trace table offset|0 |entry count |   144..191
                         +-------------------+--+-------------+
Data Store               | trace table offset|0 |entry count |   192..239
                         +-------------------+--+-------------+
Data Fetch               | trace table offset|0 |entry count |   241..287
                         +-------------------+--+-------------+
```

* Version -- Identifies the version of this trace table structure (version field) and the number of trace classes

defined (classes field) (i.e., the number of trace table header entries).

2) Trace S-Op Fetch -- trace the fetching of S-Ops at specified logical addresses. Each S-Op to be traced is specified by the logical address of the left-most (i.e., low address) bit of its op-code.

3) Trace Name Resolution/Evaluation -- trace the resolution/evaluation of name table entries of specified name tables. Each name table entry to be traced is specified by the logical address of its name table (i.e., the name table pointer locating its name table) and the name corresponding to the traced NTE.

4) Trace Procedure Transitions (Entry and Exit) -- trace the entry into and exit from specified procedures. Each traced procedure is specified by the logical address of its entry descriptor. Procedure transition tracing occurs at two points, procedure entry and procedure exit.

5) Trace Data Store -- trace the storing of any bit of data within a range of offsets within a particular object. A trace event is signalled whenever any bit within a traced range is stored, whether the operand being stored is contained entirely within the traced range, or overlaps the traced range.

6) Trace Data Fetch -- trace the fetching of any bit of data within a range of offsets within a particular object. A trace event is signalled whenever any bit within the traced range is fetched, whether the fetched operand is contained entirely within the traced range, or overlaps it.

Each trace table header entry has three fields: a 32-bit trace table offset, a disable flag bit, and a 15-bit entry count. The trace table offset is the (object-relative) offset of a trace table, which describes the events within this class to be signalled. The entry count is an unsigned integer indicating the number of entries in the trace table.

A disable flag value of one indicates this class of events is not being traced. No trace table exists for this class. The remaining fields of this trace table header entry are ignored.

A disable flag value of zero and zero entry count indicates all events of this class are to be traced. The trace table is not searched; rather a signal is generated unconditionally. The trace table offset field is ignored.

A disable flag value of zero and non-zero entry count indicates specific events are to be traced. The trace table offset locates a trace table for this event class. The entry count indicates the number of entries in the trace table.

Traceable events may occur during certain phases of S-instruction execution. Whenever an event of a particular class may occur, the trace table for this class is searched. If the table contains the presently occurring event, a trace signal is generated. If the presently occurring event is not in the table, the interpretation continues. A return from the trace signal continues the interrupted S-op. Since the trace signal handler may have modified the faulting procedure's procedure object, data, and/or state, all encached information must be rederived following return from a trace signal. Knowledge of the occurence of a trace signal is lost to the interpreter following return from the signal handler.

6.1.3 Trace Tables

A trace table is a two level table. The first level, located by a trace table header entry, is the trace table. The second level is called the trace event table. All entries for a particular table occupy logically contiguous storage. Tables are located by their left most or lowest address bit. All offsets are unsigned integers denoting bit granular, object relative locations.

Trace Table Entry

```
0             127 128     159 160          175
+---------------+----------+---------------+
|   Pointer     |  offset  |  entry count  |
+---------------+----------+---------------+
```

The pointer identifies an entity containing traced events (e.g., a procedure or a name table). The 32-bit offset locates the trace event table within the object containing the trace table. The trace event table defines a set of specific events within the traced entity which are to generate signals. The 16-bit entry count is an unsigned integer denoting the number of specific events defined for this class within this entity (i.e., the number of entries in this trace event table).

A particular UID pointer may be included in at most one trace table entry for a single event class for a particular stack. A zero entry count indicates all events of this class for this entity are to be traced. The trace event table is not searched. The offset field is ignored. A non-zero entry count indicates the number of trace event table entries defined within this class for this entity.

The trace table entries within a particular class are ordered by increasing value of their UID pointer fields (with these fields taken as 128 bit integers). Search of these tables terminates when a UID pointer field greater than a UID pointer to the potentially traceable event (also taken as a 128 bit integer) is found.

6.1.4 Class-Specific Trace Event Tables

Since the trace event table defines specific events, its format depends upon its event class.

6.1.4.1 S-Op Trace Event Table

S-Ops to be traced are specified on a per-procedure basis. The UID pointer in the trace table entry is a pointer to the entry descriptor of a procedure containing traced S-Ops. Fetching of S-Ops associated with this entry descriptor are the potentially tracable events. The trace event table for this class is a table of 32-bit PC offsets locating the S-Ops to be traced. Note that the S-Ops associated with an entry descriptor are contained in the same object as the entry descriptor.

S-Op Trace Event Table Entry

At S-Op fetch time, the S-Op Trace Table containing a UID pointer pointing to the current procedure entry descriptor is found. The S-Op trace event table entries associated with this trace table entry are ordered by increasing PC offset value. Search of this table terminates when a trace event table entry is found containing a PC offset value greater than the current PC offset. At S-Op fetch time, if a search of this table finds an entry matching the fetch PC offset, an S-Op trace event is

signalled. Otherwise, the S-Op is fetched and interpreted. When the signal handler returns, the S-Op is refetched and interpreted.

6.1.4.2 Name Resolve/Evaluate Trace Event Table

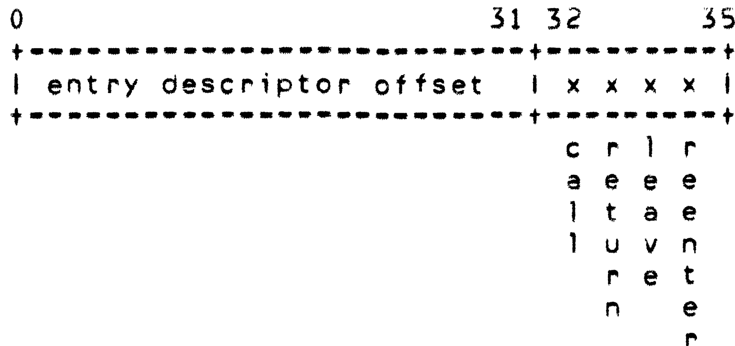
Names to be traced are specified per-name table. The resolution or evaluation of names associated with the traced name table are the potentially tracable events. The UID pointer in the trace table entry is a name table pointer. The trace event table is a bit table (i.e., an array of bits), with one bit for each 64 bit name table entry or name table extension in the specified name table. The trace table's entry count must be zero (i.e., trace all name resolves/evaluates), or equal to the largest name in the corresponding name table. (A procedure's Procedure Environment Descriptor specifies both the name table pointer and largest name). At name resolution or evaluation time, if name tracing is being performed, the name resolve/eval trace table entry whose UID pointer points to the current name table is found. The name resolve/eval trace event table associated with this trace table entry is indexed by the name to be resolved prior to initiating the resolution. If the bit thus located is one, or no trace table entry with the proper UID pointer is found, a name resolve/eval trace event is signalled. Otherwise, the name is resolved or evaluated. When the trace event signal handler returns, the resolution or evaluation is completed.

6.1.4.3 Procedure Transition Trace Event Table

Calls, returns, and nonlocal gotos are tracable events. Procedures to be traced are specified on a procedure object basis. The UID pointer in the trace table entry is a pointer to the procedure object header (note: its offset is ignored). Each trace event table entry specifies a 32-bit offset (relative to the start of the traced procedure object) of an entry descriptor to be traced, and four trace event flags:

9:43:30
5/Nov/79
Rev. 1

Procedure Transition Trace Event Table



In the following descriptions, procedure "A" calls procedure "B", who calls procedure "C". Procedure "B" is being traced.

* call - signal procedure call trace whenever this procedure is called. Tracing is controlled by B's stack's (domain's) trace pointer. The signal occurs within B's domain and on B's stack after the new environment has been established (including the new stack frame and static data area). The signal occurs as the call completes before fetching the first S-op of B.

* return - signal procedure return trace whenever this procedure's activation record is to be deleted. The signal occurs just before the activation record is actually deleted. Tracing is controlled by B's stack's (domain's) trace pointer. The signal occurs within B's domain and on B's stack before any environment is destroyed (i.e., as soon as the return is recognized). A return from the signal handler continues deletion without further signalling for this activation record, including invocation of any cleanup handlers. Note, one signal occurs per activation record per traced stack. This signal may occur when the traced procedure returns, or if a nonlocal goto is passing control to an activation previous to the traced procedure's in the dynamic call history.

* leave - signal procedure leave trace whenever this procedure calls another procedure. Tracing is controlled by B's stack's (domain's) trace pointer. The signal occurs within B's domain on B's stack after parameter pointers and state have been pushed, but before the target environment has been built. A return from the signal handler continues this call without further signalling.

9:43:30
5/Nov/79
Rev. 1

* reenter - signal procedure reenter trace whenever this procedure is reentered via a return. Tracing is controlled by B's stack's (domain's) trace pointer. The signal occurs in B's domain on B's stack upon fetch of the first S-Op at B's return PC after the environment has been reestablished. This signal occurs if C returns to B, or any procedure following B in the dynamic call history performs a nonlocal go to B.

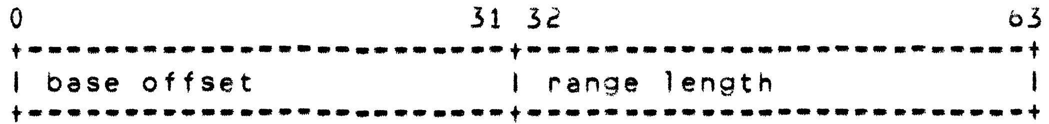
If procedure transition tracing is being performed, the procedure transition trace tables are searched to determine if a signal is to be generated. The procedure transition trace table is searched for an entry whose UID pointer points to the procedure object header of the current procedure object. If none is found, the current procedure transition is not being traced. If one is found, the associated trace event table is searched for an entry equal to the offset of the procedure entry descriptor in question (this offset is relative to the start of the containing procedure object). If none is found, this procedure transition is not being traced. If one is found, and if its flags indicate the transition occurring is being traced, the signal is generated as indicated in the preceding discussion.

Multiple signals of this class may occur if (e.g.) B's stack specifies leave tracing, and C's stack specifies call tracing. In such a case, two distinct signals are generated, the first one on B's stack, and the second one on C's stack.

6.1.4.4 Data Store/Fetch Trace Event Table

The accessing of operand values in data space are the potentially traceable events. Trace event tables for data store and fetch tracing are identical. Areas to be traced are denoted by their containing object (specified as the UID pointer in a trace table entry) and the start and length of each traced area within that object. (The offset of the UID pointer must be zero, or results are unpredictable). Each trace event table entry delimits each range as a 32-bit object relative base offset and the number of bits in the range (range length).

Data Store/Fetch Trace Event Table Entry



```
0                          31 32                        63
+----------------------------+----------------------------+
| base offset                | range length               |
+----------------------------+----------------------------+
```

Trace event table entries are ordered by increasing base offset value. The range length value is a 32-bit unsigned integer. Note that range length is not the offset of the end of the traced range.

If either class of data tracing is being performed, whenever virtual memory is accessed to read or write an operand or portion of an operand, the data fetch or store trace table (respectively) is searched for an entry whose UID pointer pointing to the start of the object containing that operand. If none is found, this data access is not being traced. Otherwise, the trace event table associated with the found trace table entry is searched for delimited area containing any part of the (portion of the) operand being accessed. If none is found, this operand access is not being traced. Otherwise, the first one found generates a signal (a data store trace event or a data fetch event).

6.2 Considerations For SPRINT

Whenever a new stack is entered (e.g., when a domain boundary is crossed), the new trace pointer is examined. If this pointer is non-null, the new stack has tracing enabled. The stack's trace table header is located and each defined entry examined. Tracing for each class is enabled according to the class' trace table header entry. Reserved entries are not examined.

Trace tables and trace event tables are defined and ordered to minimize searching. When a process changes procedure objects, the procedure transition trace table is searched for an entry corresponding to that procedure object. If none is found, this class of tracing is disabled until the procedure object is again changed. Likewise, when a new name table is specified, the name resolve/eval trace table is searched for an entry corresponding to the new name table pointer. If none is found, name tracing is disabled until a new name table is defined. The S-Op trace table is searched at procedure entry/reentry time. If no entry matching the destination procedure's entry descriptor is found, S-Op tracing is disabled until the next procedure entry/reentry.

Since data references are more random than procedure object references, data store/fetch trace tables must be searched for each store or fetch while a process is running on a traced stack (domain). The trace table is searched for an entry corresponding to the operand's data object. If one is found, its trace event table is searched until a range is found containing any bit of the operand, or a base offset is found beyond the end of the operand. Searching terminates as soon as one of these conditions is met: If an operand falls in several traced ranges, only the first one found generates a signal. Only one signal is generated per operand regardless of the number of memory references required to access the entire operand.

6.3 Access to Macrostate

In order for the native mode debugger to function it must be able to access the macrostate of a process. The following functions constitute the total guaranteed interface available to a native mode debugger. Implementation restrictions are expected in certain implementations. These functions supplement the Debugger/Kernel interface functions.

6.4 Debugger/Namespace Functions

The functions which are described are implemented as Unprivileged S-ops and/or software. The debugger can thus execute them while in any domain.

In the following interface descriptions, pointer arguments are expected to be 128 bit pointers. The error argument is expected to be an unsigned integer capable of expressing values in the range 0..65535 (i.e., a 16 bit unsigned integer). Any other arguments produce unpredictable results.

Namespace and protection execution exceptions are signalled. Improperly sized operands are treated as for SPL Pointer instructions (see the SPL S-language definition in Chapter12.) The following conditions are detected and returned as a value of the error argument:

0 - No errors.

1 - fp argument is not a Frame Pointer.

2 - There is no previous or successor Stack Frame.

3 - The new_trace_pointer does not locate a Trace Table Header.

Except as indicated, the state of writable arguments other than "err", and the results of executing one of the following interfaces, are undefined and unpredictable if a nonzero error argument is returned.

6.4.1 Get_Current_FP (current_fp)

```
PROCEDURE get_current_fp
       (WRITE ONLY      current_fp:       POINTER TO
```

This function returns the value of the current frame pointer (FP) base register to the caller in the argument "current_fp". The "current_fp" argument is expected to be of length 128 to hold a full UID pointer.

This function is intended for use to begin traversing the users stack. Note that this function returns the FP of the debugger procedure's activation record.

6.4.2 Get_Previous_FP (fp, prev_fp, err)

```
PROCEDURE get_previous_fp
       (READ ONLY       fp:          POINTER TO stac
        WRITE ONLY      prev_fp:  POINTER TO stack_frame,
        WRITE ONLY      err:          0..65535);
```

This function returns the value of the frame pointer (FP) base register for the predecessor of "fp" in "prev_fp". "fp" is used only as a source and is expected to contain a pointer representing some activation's frame pointer. The frame pointer value for the predecessor frame is stored in the "prev_fp" argument. The "prev_fp" argument is expected to be of length 128 to hold a full UID pointer.

9:43:30
5/Nov/79
Rev. 1

6.4.3 Get_Successor_FP (fp,succ_fp,err)

```
PROCEDURE get_successor_fp
       (READ ONLY        fp:         POINTER TO stac
        WRITE ONLY       succ_fp:  POINTER TO stack_frame,
        WRITE ONLY       err:         0..65535);
```

This function returns the value of the frame pointer (FP) base register for the successor of "fp" in "succ_fp". "fp" is used only as a source and is expected to contain a pointer which is the frame pointer of some frame. The frame pointer value for the successor frame is stored in the "succ_fp" argument. The "succ_fp" argument is expected to be of length 128 to hold a full UID pointer.

6.4.4 Get_State (fp, state_ptr, err)

```
PROCEDURE get_state
       (READ ONLY        fp:         POINTER TO stac
        READ ONLY        state_ptr:       POINTER TO
                         readable_macro_state,
        WRITE ONLY       err:         0..65535);
```

This function returns the "Readable Macro State" of the frame whose frame pointer value is given by "fp". The state is dumped as a record whose format is defined by "readable_macro_state. State is dumped beginning at the location given by "state_ptr", which is expected to contain a pointer.

6.4.5 Set_State (fp, state_ptr, err)

```
PROCEDURE set_state
       (READ ONLY        fp:         POINTER TO stac
        READ ONLY        state_ptr:       POINTER TO
                         writable_macro_state,
        WRITE ONLY       err:         0..65535);
```

This function loads the "Writable Macro State" of the frame whose frame pointer value is given by "fp". The state is loaded from the record whose format is defined by "writable_macro_state". The Writable Macro State is loaded beginning from the location given by

"state_ptr", which is expected to contain a pointer.

If a nonzero "err" is returned, state is not modified.

6.4.6 Swap_trace_pointers (new_trace_ptr, old_trace_ptr)

```
PROCEDURE swap_trace_pointers
      (READ ONLY          new_trace_pointer:        PJ
                        trace_table_header,
       WRITE ONLY         old_trace_pointer:         P
                        trace_table_header,
       WRITE ONLY         err:          0..65535);
```

This function sets the value of the current trace pointer to the value given by the argument "new_trace_ptr". It returns the value of the old trace pointer in argument "old_trace_ptr". "new_trace_ptr" and "old_trace_ptr" are expected to be of length 128 to contain a full UID pointers.

Upon loading the new trace pointer, the current stack's previous trace state is lost. The trace tables located by the new trace pointer are evaluated and provide the new definition for the current stack's trace state. The appropriate trace modes are enabled and tracing commences as soon as the next S-Op is fetched. (N.B., if a Trace Table indicates "trace every occurance", recursive signalling may result.)

If a nonzero "err" is returned, the current stack's trace pointer is set to null (all tracing disabled).

6.5 Macro State Definition

The following describe the MACRO STATE records which are used for the "Set_State" and "Get_State" functions.

```
TYPE readable_macro_state IS
 RECORD
   FP:  POINTER, %frame pointer (ABR)
   SEP: POINTER, %S-environment pointer (ABR)
   SDP: POINTER, %static data pointer (ABR)
   PBP: POINTER, %procedure base pointer (ABR)
   EDP: POINTER, %entry descriptor pointer
   FHP: POINTER, %frame header pointer
         PC:     offset_type, %current pc int
```

```
    STO: offset_type  %stack top offset
  END RECORD;


 TYPE writable_macro_state IS
  RECORD
    EDP: POINTER, %entry descriptor pointer
            PC:     offset_type, %current pc int
    STO: offset_type  %stack top offset
  END RECORD;
```

--End of Chapter--

9:43:30
5/Nov/79
Rev. 1

Chapter 7
Exception Conditions, Faults, and SIGNALs

7.1 Introduction

Exception conditions are error conditions detected by the system. Trace conditions and nonresolvable pointer faults are escapes which causes software to be invoked (e.g., to resolve a nonresolvable pointer, instantiate a static data area, or log an event). Each condition is detected by the system and passed on to a SIGNALLER. The SIGNALLER is invoked with a condition code and a pointer. The semantics of this pointer depend upon the nature of the condition being signalled. Prior to invocation of the SIGNALLER, all macro state is stored where it is accessible to both the SIGNALLER and the handler.

The SIGNALLER is responsible for locating a software handler for a particular condition. The SIGNALLER scans the current stack for a handler defined to handle that condition. If one is found, it is CALLed. If none exists on the current stack, the condition may be passed on to the stack which called the current stack.

7.2 Conditions

The following sections describe the conditions generated and the information presented to the SIGNALLER. For each condition, the value of the pointer parameter passed to the signal handler is given along with the semantics for returning from each fault. The signal handler must request explicitly that the macro state of an activation be modified.

7.2.1 ILLEGALLS_OP

unrecoverable class

POINTER TO: UNDEFINED

Semantics:

An attempt has been made to fetch and execute an S-Op which is not defined in the current S-Language. The signal occurs on the current stack. The POP and PC in macro state

locate the faulting S-Op. A return (re)executes the S-Op located by the PC in macrostate. Such a return may cause recursive faulting.

7.2.2 ILLEGAL_OPERAND_SYLLABLE_SIZE

unrecoverable class

POINTER TO: UNDEFINED

Semantics:

An attempt has been made to invoke a procedure which specifies a value for operand syllable size which is not eight, twelve, or sixteen (eight or sixteen in SPRINT). The POP and PC in macro state locate the faulting S-Op. Macro state also contains a pointer to the entry descriptor which was called. This entry descriptor locates the faulty Procedure Environment Descriptor. The signal occurs on the called stack. A return from handling this fault causes unpredictable results.

7.2.3 NAME_OUT_OF_RANGE

unrecoverable class

POINTER TO: UNDEFINED

Semantics:

An attempt has been made to resolve or evaluate a name which is larger than the largest name in the current name table. The POP and PC in macro state locate the S-Op which specifies the illegal name. The signal occurs on the current stack. A return from handling this fault reexecutes the S-Op located by the PC in macro state. Such a return may cause recursive faulting.

7.2.4 ILLEGAL_EAR_TRAP

unrecoverable class

POINTER TO: faulting name table entry

Semantics:

An attempt has been made to resolve or evaluate a name table entry whose EAR Trap is set, and no EAR handler exists for the current S-Language. The signal occurs on the current stack. The POP and PC locate the S-Op which generated the illegal_EAR exception. The handler is passed a pointer to the offending name table entry. A return from handling this fault reexecutes the S-Op located by the PC in macro state. Note, such a return may cause recursive faulting.

7.2.5 INCONSISTENT_NAME_TABLE_ENTRY

unrecoverable class

POINTER TO: faulting NTE

Semantics

An attempt has been made to resolve or evaluate a name table entry whose specification is not consistent. The signal occurs on the current stack. The POP and PC locate the S-Op which generated the illegal_nte exception. The handler is passed a pointer to the inconsistently specified NTE. A return from handling this fault reexecutes the S-Op located by the PC in macro state. Such a return may cause recursive faulting.

7.2.6 INVALID_POINTER

unrecoverable class

POINTER TO: data

Semantics:

A pointer resolve has been attempted on a pointer whose NR bit is not asserted, but whose format field does not specify a valid resolvable pointer format. The signaller info pointer locates the invalid pointer in data space. A return from the signal handler reattempts the fetch and resolve of the faulting pointer from data space.

7.2.7 INVALID_ENTRY_DESCRIPTOR

unrecoverable class

POINTER TO: invalid entry descriptor

Semantics:

Architectural call found a structure other than an Entry Descriptor, when it expected an Entry Descriptor. The signaller info pointer locates the invalid structure. A return from this signal handler reevaluates the structure located by signaller info pointer.

7.2.8 INVALID_S_INTERPRETER

unrecoverable class

POINTER TO: S-Interpreter_pointer field of a PED.

Semantics:

A call has been made to a procedure whose Procedure Environment Descriptor S-interpreter pointer specifies an object which is not an S-Interpreter. A return from this handler reevaluates the S-Interpreter Pointer field of the target PED.

7.2.9 INACTIVE_S_INTERPRETER

unrecoverable class

POINTER TO: S-Interpreter_pointer field of a PED.

Semantics:

A call has been made to a procedure whose Procedure Environment Descriptor S-interpreter pointer specifies an object which is an S-Interpreter, but is not active. In BIOS, all S-interpreters must be activated at IPL time. A return from this handler reevaluates the S-Interpreter Pointer field of the target PED.

7.2.10 S_OP_TRACE

trace class

POINTER TO:traced S-Op

Semantics:

An attempt has been made to fetch and execute an S-Op whose logical address is contained in the current stack's S-Op trace table. The signal occurs on the current stack. The handler is passed a pointer to the traced S-Op (the same as the current PC stored on macro state). The signal occurs following completion of all previous S-Ops, but prior to interpretation of the traced S-Op. A return refetches the traced S-Op from virtual memory and continues dispatching and interpretation without recursive faulting. Process data space may be modified by the handler.

7.2.11 NAME_RESOLVE/EVAL_TRACE

trace class

POINTER TO: name table entry

Semantics:

An attempt has been made to resolve or evaluate a name that is reflected in the current stack's Name Resolve/Eval Trace Table. The signal occurs on the current stack before the name is actually resolved. The handler is passed a pointer to the traced name's name table entry. A return from the handler refetches the traced NTE from virtual memory and continues its resolution or evaluation without recursive faulting. Process data space may be modified by the handler.

7.2.12 PROCEDURE_CALL_TRACE

trace class

POINTER TO: entry descriptor

Semantics:

An attempt has been made to call a procedure whose entry

Data General Corporation
Company Confidential

descriptor is reflected in the current stack's Procedure Transition Trace Table with the call tracing flag set. The signal occurs on the current stack. The handler is passed a pointer to the traced procedure's entry descriptor. A return from the handler continues the call without recursive faulting. See the Tracing Facility chapter for details on the semantics of this fault. Process data space may be modified by the handler.

7.2.13 PROCEDURE_RETURN_TRACE

trace class

POINTER TO: entry descriptor

Semantics:

An attempt has been made to 1) return from or 2) execute a nonlocal goto through a procedure whose entry descriptor is reflected in the current stack's Procedure Transition Trace Table with the return tracing flag set. The signal occurs on the stack containing the activation record to be deleted (this becomes the current stack). This fault occurs prior to deletion of a traced procedure's activation record and after any cleanup handlers for the traced procedure have returned. The fault occurs while the activation record is entirely intact. The handler is passed a pointer to the traced entry descriptor. A return from the handler continues deletion of the activation record without recursive faulting. Process data space may be modified by the handler.

7.2.14 PROCEDURE_LEAVE_TRACE

trace class

POINTER TO: entry descriptor

Semantics:

An attempt has been made to 1) call out from or 2) execute a nonlocal goto out of a procedure whose entry descriptor is reflected in the current stack's Procedure Transition Trace Table with the leave tracing flag set. The signal

occurs on the current stack as soon as the call or nonlocal goto S-Op is recognized and prior to its dispatch and interpretation. (The current stack is the one from which the call or nonlocal goto was fetched). The handler is passed a pointer to the traced entry descriptor. The current PC stored in macro state is the address of the call or nonlocal goto S-Op. A return from the handler refetches the S-Op pointed to by the current PC from virtual memory without recursive faulting. Process data space may be modified by the handler.

7.2.15 PROCEDURE_REENTER_TRACE

trace class

POINTER TO: entry descriptor

Semantics:

An attempt has been made to return to a procedure whose entry descriptor is reflected in the current stack's Procedure Transition Trace Table with the reenter tracing flag set. This attempt occurs as a result of a return to such a procedure, or a nonlocal go to such a procedure. In this context, the current stack is the one contaiing the activation record which is gaining control. The handler is passed a pointer to the entry descriptor. The signal occurs upon fetch of the S-Op located by the traced procedure's current PC (i.e., the first S-Op at the return point). The macro state stored is that of the activation being returned to. The signal appears as if it originated within the traced procedure at the point to which the return would have passed control. A return from the handler refetches the first S-Op and continues interpretation. Process data space may be modified by the handler.

7.2.16 DATA_FETCH_TRACE

trace class

POINTER TO: data

Semantics:

An attempt has been made to fetch data from an area of logical memory delimited in the current stack's Data Fetch Trace Table. The signal occurs on the current stack before the data is actually fetched. The handler receives a pointer to the first portion of data to be fetched which lies within the delimited range. Note that the actual pointer may vary in a model dependent manner. A return from the handler reattempts the fetch and continues execution of the S-Op without recursive faulting. The handler may modify data space, including the delimited portion of logical memory.

7.2.17 DATA_STORE_TRACE

trace class

POINTER TO: data

Semantics:

An attempt has been made to store into an area delimited in the current stack's Data Store Trace Table. No data is stored. The signal occurs on the current stack at the time of the store. The handler receives a pointer to the first portion of data to be stored which lies within the delimited range. Note that the actual pointer may vary in a model dependent manner. A return from the handler completes the store without recursive faulting. Note that, if an operand being stored spans a Data Store Trace delimited range, it may have been partially modified in logical memory at the time the trace fault is taken. Actual behavior is model dependent. The handler may modify process data space. Modifications to the delimited portion of logical memory are undefined and unpredictable.

7.2.18 NONRESOLVABLE_POINTER

nonresolvable pointer fault

POINTER TO: data

Semantics:

A pointer resolve has been attempted on a pointer whose NR bit is asserted. The signaller information pointer locates the faulting pointer in data space. A return from the handler reattempts the fetch and resolve of the faulting pointer from data space. If the NR bit is still asserted, the fault will be signalled again.

7.2.19 STATIC_DATA

static data fault

POINTER TO: UNDEFINED

Semantics:

Architectural call was unable to locate an instantiated static data area for the procedure being called. Architecural call invokes the static data fault handler through a pointer in the stack header (see the Kernel Design Specification). The macrostate of the faulting activation is that of the called procedure. However, the SDP is undefined. The PED of the called procedure is located via the Entry Descriptor in the faulting activation's macrostate. The static data fault handler provides architectural call with an association between the called procedure's static data prototype pointer and the instantiated static data area. See the Kernel Design Specification for details of how this association is communicated.

--End of Chapter--

Chapter 8
Call and Return

8.1 Call

CALL is invoked via an s-language specific CALL S-Op. It is assumed an S-Interpreter is active on some stack, denoted the CALLER's STACK. The procedure to be invoked is denoted the TARGET PROCEDURE. The address of the TARGET PROCEDURE's ENTRY DESCRIPTOR (ED) is denoted the TARGET ADDRESS.

CALL is partitioned into three phases. The first completes the caller's activation, stores state to be reestablished upon return, pushes parameter pointers. The second phase locates the TARGET ED and builds the TARGET environment, crossing domain boundaries if necessary. The final phase builds the TARGET activation and invokes the S-Interpreter.

8.1.1 Phase one - Complete_CALLERs_Activation

The first phase completes the CALLER's activation, constructs parameter pointers to the actuals, and checks if the CALLER is being traced. The description begins on the CALLER's STACK within the CALLER's activation with the dispatch of a CALL S-Op:

1) If the current S-Machine defines any S-Machine-specific state, it is stored on the CALLER's STACK. The manner in which this state is stored is left to the S-Language to define. When the CALLER's STACK again becomes current (via a RETURN or NONLOCAL GOTO), that state is reloaded by the S-Machine.

2) Allocate and initialize a frame header on top of the current stack for the new activation. The new frame header is threaded onto the call history of the process following the current frame header.

3) The current activation's frame header is completed.

4) Current Macrostate is stored internal to the system. Hence, macrostate is never directly accessible to the program. It may be interrogated or modified only indirectly, via a set of interfaces described in the

Chapter 7.

5) Parameter pointers are constructed for each of the actuals. Space is reserved at the top of the current stack for these. (Note that parameter pointers are linkage pointers, and as such are subject to the restrictions placed on linkage pointers by the Namespace architecture.)

6) The current stack's trace tables are examined to determine if LEAVE tracing is being performed for the current procedure. If so, Procedure_Leave_Trace is signalled on the current stack. A return from this signal continues the algorithm from this point. Note that macrostate and/or the parameter pointers may have been modified by the signal handler. Calls are provided to explicitly request storing or modifiying macro state.

[Note: tracing must be signalled on the traced stack, i.e., the current stack. The handler must be able to modify the caller's macro state and the parameter pointers being passed to the target procedure. The order of events presented here is sufficient, but not necessary to guarantee this capability. This note also applies to CALL tracing, during the last phase of architectural call.]

7) The frame header now contains information describing the CALLER's activation record. The current FP is adjusted to point to the top of the area reserved for parameter pointers. At this point, the current FP and SP point to the same bit and a zero length local data section is allocated at the top of the CALLER's stack.

8.1.2 Phase two - Locate_TARGET_Environment

In essence, if target_pointer locates an ED which is contained within the current Procedure Object, and target ED shares the same PED as the target ED (i.e., the PED_offset fields of the EDs are identical), the calling and target activations share the same Procedure Environment and the remainder of this phase is bypassed.

If target_pointer does not point to an ED contained within the current procedure object, kernal microcode is invoked to find the TARGET PROCEDURE's ED, crossing domains if necessary and passing parameter pointers across domain boundaries. The Namespace microcode invokes the kernel microcode from the CALLER's stack, and receives control again on the TARGET stack. It is assumed that, if

stacks changed, the kernel microcode has built and threaded a frame header and copied the parameter pointers (with appropriate checking) onto the TARGET stack. Trace state for the old stack is saved, and trace state for the TARGET stack is evaluated.

If the TARGET ED does not share the current activation's PED, a new environment is built. Note that, if target_pointer points outside the current procedure object, the current and TARGET EDs cannot share the same PED.

This phase occurs in neither the CALLER's not the TARGET's activation, but in that "GREY" area in between. When Namespace again resumes execution, FP points to the base of the invoked procedure's activation (following the frame header and parameter pointers). In addition, the SP (or STP) points to the first bit available at the top of the current stack (following the parameter pointers). At this point, there is no local storage allocated for the TARGET procedure.

1) If target_pointer points to an ED within the current PU, and the PED associated with the target ED is the same as the PED associated with the current ED, the only part of the process' environment which is changing is its local automatic data (including its parameter pointers). Only this state need be flushed from the Name Cache. Proceed to phase three. Otherwise:

    1) If target_pointer does not point to an ED contained within the current procedure object, invoke kernel microcode to follow the target pointer until an ED is found. Details of this algorithm are beyond the scope of this specification. Control returns to Namespace on the TARGET stack (which is now the current stack). The current ED is the TARGET ED. The TARGET PED is located via TARGET ED's address and the PED_offset field of the TARGET ED.

    2) Since a new environment is being established, the Name Cache must be flushed.

    3) Load the current PBP and NTP from the appropriate fields of the TARGET PED. If the PBP field of the TARGET PED is not an object relative pointer, signal Invalid_Pointer.

    4) Locate an instance of Static Data associated with the SDPP of the TARGET PED for the current stack. If none has been instantiated, invoke the static

data fault handler to instantiate one. Load the SDP with a pointer to the static data area.

5) Load the LARGEST_NAME and CURRENT_SY_LABLE_SIZE from the TARGET PED.

6) Locate the TARGET S-Interpreter, via the SIP field of the TARGET PED. If this SIP field does not denote an object of type S_INTERPRETER, signal Invalid_S_Interpreter. In SPRINT, if it denotes an inactive S-Interpreter object, signal Inactive_S_Interpreter.

8.1.3 Phase three - Build_TARGET_Activation

This ends the second phase of architectural call. The final phase takes place on the TARGET stack and within the TARGET PROCEDURE's new activation. At this point, the TARGET PROCEDURE has logically been invoked. All that remains is to complete the new activation's environment and macrostate.

1) Allocate a local storage area at the top of the stack by adding the INITIAL_FRAME_SIZE field of the TARGET ED to the offset field of SP. Check for extent violation.

2) The TARGET PROCEDURE begins execution at the address denoted by the sum of the PBP offset plus the initial_PC_offset field of the TARGET ED (this offset is within the same object denoted by PBP). SET the PC.

3) The current stack's trace tables are examined to determine if CALL tracing is being performed for the current procedure. If so, Procedure_Call_Trace is signalled on the current stack. A return from this signal continues the algorithm from this point. Note that macrostate and/or the parameter pointers may have been modified by the signal handler. Calls are provided for explicitly requesting the storing or modification of macrostate.

4) Invoke the S-Interpreter.

5) Invoking a new S-interpreter may require the construction of initial S-Machine state. The S-Machine may require knowlege of whether SIP has changed as a result of this call.

9:53:4
5/Nov/79
Rev. 1

6) Begin interpreting S-Ops.

8.2 Return

Return terminates the lifetime of the current activation, and makes the caller's activation current (the caller's activation is the activation which immediately preceeds the current one in the process' dynamic call history). From the Namespace Point of view, this is a relatively simple operation.

Functionally, architectural return performs the following:

1) Check the current activation for return tracing. If the current trace table indicates the current procedure is being return traced, generate a returnable PROCEDURE_RETURN_TRACE signal. Continue when the signaller returns.

2) Locate the caller's macrostate.

3) Delete the current activation record from the current stack.

4) Reload the caller's macrostate. This makes the caller's activation current. The Protection level of the system may be invoked at this point if the return changes stacks.

5) Check if the current trace table indicates the current procedure is being reenter traced. If so, generate a PROCEDURE_REENTER_TRACE returnable signal. Continue when the signaller returns.

6) Continue interpreting S-Ops.

8.3 Nonlocal Goto

Nonlocal goto causes control to pass a specified S-Op within a specified activation (other than the current one).

8.3.1 Restrictions on Nonlocal Goto

A Nonlocal Goto S-Op must feature an opcode unique from all local (intra-activation) gotos. The nonlocal goto instruction

requires two operands: the destination activation (i.e., the FP of the destination activation), and a PC offset. _The specified PC offset must locate an S-Op within the procedure associated with the destination activation, or results are unpredictable._

Architectural return is a special case of nonlocal goto, where the destination activation and PC offset are implied from the calling activation's macrostate.

8.3.2 Semantics of Nonlocal Goto

Functionally, nonlocal goto traverses the dynamic call history of the process backwards in time, issuing a return for each activation encountered until the destination is reached. When the destination activation becomes current, a local goto to the specified PC offset is issued.

1) Check the current activation for return tracing. If the current trace table indicates the current procedure is being return traced, generate a returnable PROCEDURE_RETURN_TRACE signal. Continue when the signaller returns.

2) Locate the caller's macrostate.

3) Delete the current activation record. This involves intervention by KOS to determine if (e.g.) cleanup processing is required.

4) Reload the caller's macrostate. This makes the caller's activation current. The Protection level of the system may be invoked at this point if the return changes stacks.

5) If the current activation is the destination activation, proceed with the next step. Otherwise, repeat the first step through this step.

   At this point, the destination activation is again current.

6) Reload the current PC offset with the PC offset specified in the nonlocal goto S-Op. This PC offset is assumed to locate a valid S-Op within the current procedure.

7) Check if the current trace table indicates the current procedure is being reenter traced. If so, generate a PROCEDURE_REENTER_TRACE returnable signal. Continue when

the signaller returns.

8) Continue interpreting S-Ops.

The results of executing a nonlocal goto to a nonexistant activation results in an signal issued by the Protection level of the system. The results of executing a nonlocal goto to an activation whose macrostate has been incorrectly modified are unpredictable.

--End of Chapter--

9:53:4
5/Nov/79
Rev. 1

Chapter 9
S-Languages

Information in this chapter applies globally to all S-language specifications.

9.1 Branches

9.1.1 Relative Branches

A relative branch instruction specifies as an operand a "relative offset" syllable. This syllable is a k-bit literal denoting a signed, offset relative to the current PC. The PC is defined to be pointing to the beginning of the instruction containing the relative branch. The value of "k" determines the granularity of the offset. The k-bit relative offset syllable is sign-extended to 32 bits then multiplied by the greatest common divisor of 8 and k [GCD(8,k)] to obtain a bit-granular offset. This value is then added (2's complement addition) to the offset portion of the current PC. Interpretation resumes at this new I-stream location.

Note that a relative branch is always intra-Procedure Object, since only the offset of the PC is modified.

9.1.2 Offset Branch

An offset branch is an intra-Procedure Object branch specified relative to the beginning of the current Procedure Object. An offset name operand is evaluated to yield an unsigned integer offset of less than or equal to 32 bits in length. This value represents a bit granular offset. The branch is performed by the replacement of the offset portion of the PC with this value. As such, the value is an absolute branch within the current procedure object. Interpretation continues at the new I-stream location. The result of not using an unsigned integer type for an offset branch is unpredictable.

9.2 Exception Conditions

This section deals with special and exception conditions which may be encountered during s-language execution.

9.2.1 Protection Exceptions

Several exception conditions of an architectural nature may arise as a result of s-language execution. They typically involve protection or addressing violations as a result of attempted improper use of data, instructions or environment. A list and description of these exceptions can be found in Chapter xxx.

9.2.2 Namespace Errors

Any instruction with at least one operand which is a name may generate one or more name space errors. Errors occur when the name table or associated information is invalid or unreadable. Namespace errors are enumerated in the "Namespace" portion of the architecture document.

9.3 Long Instructions

An instruction is divided into discrete units of operation. At the completion of a unit of operation, any external interrupts will be honored if present. At the completion of interrupt handling, the instruction will be continued at the point of interruption with the next unit of operation. The unit of operation for any specific s-instruction may vary from model to model as a function of convenient implementation.

9.4 Overlapping Operands

It is important to note the effect of memory to memory instructions whose source and destination fields overlap. In the interest of simplicity and generality, the operation of any instruction in which the destination field overlaps any source field is specified to be undefined and may vary from model to model as a function of convenient implementation. The operands in those instructions in which a source is also a destination, such as the "add a to a" instruction, are not considered to be overlapping.

9.5 Definition Format

S-language operands can syntactically be one of two forms. They can be names or indices into the Name Table or they can be inline literals. These inline literals are used as relative offsets to the current value of the program counter in conditional and unconditional branches. As such, literals are signed integer values of k bits in length where k is established by the size of the associated name table and may be either eight, twelve or sixteen bits.

The definition the functions "resolved" and "evaluated" refer to the transformation of an operand name to the address it specifies, and of an operand name to the value located at the address it specifies, respectively.

9.6 Invalid S-Ops

| Opcodes 0 and 255 are invalid s-ops in all s-languages.
| Opcode 127 is reserved for diagnostic purposes in all s-languages
| and is treated as an invalid s-op.

--End of Chapter--

Chapter 10
SPL S-Language

10.1 Data Types and Their Representation

The SPL s-language defines six data types to support all of SPL's source data types and data structures. The following is an enumeration of each of the defined data types and its representation.

10.1.1 Integers

There are signed and unsigned integers. The two forms are distinguished by the encoding of the FETCH_MODE field in the associated Name Table Entry. Internally all integers are identically represented as fixed length signed quantities with all transformations of length and/or signedness occurring in the fetch and store steps of an instruction.

10.1.1.1 Unsigned Integers

Unsigned integers are used to represent SPL enumerations (including ASCII and BOOLEAN) and intervals where the lower bound is non-negative. Additionally, unsigned integers will be used to represent the current length of strings.

The length of an unsigned integer may be from one to thirty-two bits and thus may represent any value in the range 0 to (2**32)-1. Unsigned integers are addressed at their left-most end (low address, MSB). When fetched, unsigned integer values are internally right adjusted and zero filled. When stored, unsigned values are truncated on the left. If the truncated part, if any, is not all zeroes, then an integer range exception is signalled. If the length of a unsigned integer is not between 1 and 32, any use of the value will yield unpredictable results.

10.1.1.2 Signed Integers

Signed integers are used to represent all SPL intervals where the lower bound is negative. They are also used to represent literal values in the instruction stream.

Signed integers may be from one to thirty-two bits in length including sign bit and can thus represent any number in the range -(2**31) to +(2**31)-1. Signed integers are addressed at their left-most end (low address, MSB). When fetched, they are internally right adjusted and sign filled. When stored, they are truncated on the left. If all truncated bits, if any, are not identical to the high order bit of the result, then an integer range exception is signalled. If the length of a signed integer is not between 1 and 32, any use of the value will yield unpredictable results.

10.1.2 Bit String

Bit strings are used to represent SPL set values. Bit string operations are sometimes used on boolean values and may be used on arrays and strings of boolean values.

Bit strings are addressed at the left-most end (low address). When fetched, they are internally left adjusted and zero filled when necessary. On store, they are right truncated as necessary. Bit strings may have a minimum length of zero bits and a maximum length of (2**32)-1 bits.

10.1.3 Pointers

Pointers are used for indirect addressing. As such, they are used in the SPL s-language by the pointer instructions and by the RESERVE instruction which returns an address in the current frame.

The definition of pointer formats and their architectural usage can be found in Chapter 3.

10.1.4 Typed Operators

All SPL s-language operators are typed. As a part of its semantics, each s-language operator specifies an algorithm to be performed and the expected data type of each and every input and output operand. However, no explicit type checking is performed on either input or output operands. If operands are incorrectly specified by either the compiler or programmer, the algorithm will be executed treating the input and output data as being of the expected data types and unpredictable results will be generated with possibly no exceptions arising. Thus, unless explicitly specified otherwise, the results of every SPL s-language operator are unpredictable and may vary from model to model if the input

and/or output operands are not of the expected type.

10.2 SPL Exceptions

A number of various exception conditions resulting from execution of SPL s-instructions including the improper specification or use of data or instructions cause the generation of a program exception. The following is a list and description of the program exceptions specified for the SPL s-language. In each case the exception is signalled.

0 - Integer Range Exception

This exception occurs whenever the result of an integer arithmetic operation is outside the range of representable values for the specified destination operand size.

1 - Integer Divide by Zero Exception

The value of the divisor in a QUOTIENT INTEGER or REMAINDER INTEGER instruction is integer zero.

10.3 SPL S-Language Operation Definitions

SPL s-language instructions are defined in detail in the following sections. The operations are classified as Control Instructions, Integer Arithmetic and Bit String Instructions, Miscellaneous Instructions, and Architectural Instructions.

10.3.1 Control Instructions

10.3.1.1 BRANCH IF INTEGER ZERO

Opcode: 64

Syntax: IBZ n1, lit

n1: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.2 BRANCH IF INTEGER NOT ZERO

Opcode: 65

Syntax: IBNZ n1, lit

n1: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is not equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.3 BRANCH IF INTEGER LESS THAN ZERO

Opcode: 66

Syntax: IBLZ n1, lit

n1: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.4 BRANCH IF INTEGER LESS THAN OR EQUAL TO ZERO

Opcode: 69

Syntax: IBLEZ n1, lit

n1: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than or equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.5 BRANCH IF INTEGER GREATER THAN ZERO

Opcode: 68

Syntax: IBGZ n1, lit

n1: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.6 BRANCH IF INTEGER GREATER THAN OR EQUAL TO ZERO

Opcode: 67

Syntax: IBGEZ n1, lit

n1: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than or equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.7 BRANCH IF INTEGER EQUAL

Opcode: 72

Syntax: IBE n1, n2, lit

n1: source of type integer

n2: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are equal, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.8 BRANCH IF INTEGER NOT EQUAL

Opcode: 73

Syntax: IBNE n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.9 BRANCH IF INTEGER LESS THAN

Opcode: 74

Syntax: IBL n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is less than n2, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.10 BRANCH IF INTEGER LESS THAN OR EQUAL

Opcode: 77

Syntax: IBLE n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: The evaluated n1 is compared to the evaluated n2. If it is less than or equal to n2, then the current value of pc is updated by the value of the inline literal. Otherwise, execution continues inline.

10.3.1.11 BRANCH IF BIT STRING EQUAL TO ZERO

Opcode: 80

Syntax: SBZ n1, lit

n1: source of type bit string
lit: inline literal

Description: Evaluated n1 is compared to zero. If every bit is equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.12 BRANCH IF BIT STRING NOT EQUAL TO ZERO

Opcode: 81

Syntax: SBNZ n1, lit

n1: source of type bit string
lit: inline literal

Description: The evaluated n1 is compared to zero. If any bit is not equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

Data General Corporation
Company Confidential

10.3.1.13 BRANCH IF BIT STRINGS EQUAL

Opcode: 88

Syntax: SBE n1, n2, lit

n1: source of type bit string
n2: source of type bit string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If the lengths of the operands are equal and the values of the resultant strings are equal, the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.14 BRANCH IF BIT STRINGS NOT EQUAL

Opcode: 89

Syntax: SBNE n1, n2, lit

n1: source of type bit string
n2: source of type bit string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If the lengths of the operands are equal and the values of the resultant strings are equal, execution continues inline. Otherwise, the current value of pc is updated by using the inline literal.

10.3.1.15 BRANCH IF BIT STRING LESS THAN

Opcode: 90

Syntax: SBL n1, n2, lit

n1: source of type bit string
n2: source of type bit string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If the lengths of the operands differ, the shorter operand is padded with zeroes on the right to the length of the longer. If the resultant n1 is less than the resultant n2, the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

Notes: The lengths of n1 and n2 need not be equal.

10.3.1.16 BRANCH IF BIT STRING LESS THAN OR EQUAL

Opcode: 93

Syntax: SBLE n1, n2, lit

n1: source of type bit string
n2: source of type bit string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If the lengths of the operands differ, the shorter operand is padded with zeroes on the right to the length of the longer. If the resultant n1 is less than or equal to the resultant n2, the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

Notes: The length of n1 need not be equal to the length of n2.

10.3.1.17 BRANCH IF IN BOUNDS

Opcode: 112

Syntax: BINBND n1, n2, n3, lit

n1: source of type integer
n2: source of type integer
n3: source of type integer
lit: inline literal

Description: Evaluated n2 is compared to evaluated n1 and to evaluated n3. If it is greater than or equal to n1 and less than or equal to n3, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.18 BRANCH IF NOT IN BOUNDS

Opcode: 113

Syntax: BNINBND n1, n2, n3, lit

n1: source of type integer
n2: source of type integer
n3: source of type integer
lit: inline literal

Description: Evaluated n2 is compared to evaluated n1 and to evaluated n3. If it is less than n1 or greater than n3, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.19 BRANCH IF SUBSET

Opcode: 114

Syntax: BSUB n1, n2, lit

n1: source of type bit string
n2: source of type bit string
lit: inline literal

Description: This instruction tests whether or not the set named by n1 is a subset of the set named by n2. Evaluated n2 is logically complemented and then logically and'ed with evaluated n1. If the result is zero, then n1 is a subset of n2 and the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

Notes: The length of n1 need not be equal to the length of n2.

10.3.1.20 BRANCH IF NOT SUBSET

Opcode: 115

Syntax: BNSUB n1, n2, lit

n1: source of type bit string
n2: source of type bit string
lit: inline literal

Description: This instruction tests whether or not the set named by n1 is a subset of the set named by n2. Evaluated n2 is logically complemented and then logically and'ed with evaluated n1. If the result is non-zero, then n1 is not a subset of n2 and the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

Notes: The length of n1 need not be equal to the length of n2.

10.3.1.21 BRANCH IF POINTER EQUAL

Opcode: 116

Syntax: PBE n1, n2, lit

n1: source operand of type pointer
n2: source operand of type pointer
lit: inline literal

Description: This instruction tests whether or not the logical address represented by the pointer named by n1 is the same as that of the pointer named by n2. If the logical addresses are the same, the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

Notes: If the "Pointer Fault" bit is asserted, a "Pointer Fault" is signalled. All other trap bits are ignored.

10.3.1.22 BRANCH IF POINTER NOT EQUAL

Opcode: 117

Syntax: PBNE n1, n2, lit

n1: source operand of type pointer
n2: source operand of type pointer
lit: inline literal

Description: This instruction tests whether or not the logical address represented by the pointer named by n1 is the same as that of the pointer named by n2. If the logical addresses are not the same, the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

Notes: If the "Pointer Fault" bit is asserted, a "Pointer Fault" is signalled. All other trap bits are ignored.

10.3.1.23 FIND FIRST ONE ELSE BRANCH

Opcode: 118

Syntax: FFO n1, n2, lit

n1: source of type bit string
n2: source/destination of type integer
lit: inline literal

Description: This instruction is used to perform a forward scan through all or a portion of the bit string named by n1 searching for the first occurrence of a bit set to one. Evaluated n2 is used as an initial index into the bit string named by n1 and indicates the starting bit position for the search. Zero indicates the first bit, (length(n1)-1) indicates the

last bit.

The bit string is scanned in a forward (ascending address) direction until a bit set to one is encountered or the end of the bit string is reached. If a bit set to one is encountered, then the search is successful and the instruction is completed by updating evaluated n2 to point to the one bit found. Execution continues with the next inline instruction.

If the end of the bit string is reached and no bit set to one has been found, then the search is unsuccessful and the instruction is completed by updating the current value of pc by using the inline literal.

Exceptions: 0 - integer range exception

10.3.1.24 FIND NEXT ONE AND BRANCH

Opcode: 119

Syntax: FNO n1, n2, lit

n1: source of type bit string
n2: source/destination of type integer
lit: inline literal

Description: This instruction is used to perform a forward scan through a portion of the bit string named by n1 searching for the next occurrence of a bit set to one. Evaluated n2 is used as an initial index into the bit string named by n1 and the search is initiated at the bit position following the one indexed by evaluated n2. Zero indicates the first bit, (length(n1)-1) indicates the last bit.

The bit string is scanned in a forward (ascending address) direction until a bit set to one is encountered or the end of the bit string is reached. If a bit set to one is encountered, then the search is successful and the instruction is completed by updating

evaluated n2 to point to the one bit found and then performing a branch by updating the current value of pc by using the inline literal.

If the end of the bit string is reached and no bit set to one has been found, then the search is unsuccessful and the instruction is complete. Execution then continues with the next inline instruction.

Exceptions: 0 - integer range exception

10.3.1.25 FIND PREVIOUS ONE AND BRANCH

Opcode: 120

Syntax: FPO n1, n2, lit

n1: source of type bit string
n2: source/destination of type integer
lit: inline literal

Description: This instruction is used to perform a backward scan through a portion of the bit string named by n1 searching for the previous occurrence of a bit set to one. Evaluated n2 is used as an initial index into the bit string named by n1 and the search is initiated at the bit position preceding the one pointed to by evaluated n2. Zero indicates the first bit, (length(n1)-1) indicates the last bit.

The bit string is scanned in a backward (descending address) direction until a bit set to one is encountered or the start of the bit string is reached. If a bit set to one is encountered, then the search is successful and the instruction is completed by updating evaluated n2 to point to the one bit found and then performing a branch by updating the current value of pc by using the inline literal.

Data General Corporation
Company Confidential

If the start of the bit string is reached and no bit set to one has been found, then the search is unsuccessful and the instruction is complete. Execution then continues with the next inline instruction.

Exceptions: 0 - integer range exception

10.3.1.26 FIND LAST ONE ELSE BRANCH

Opcode: 121

Syntax: FLO n1, n2, lit

n1: source of type bit string
n2: source/destination of type integer
lit: inline literal

Description: This instruction is used to perform a backward scan through all or a portion of the bit string named by n1 searching for the first occurrence of a bit set to one. Evaluated n2 is used as an initial index into the bit string named by n1 and indicates the starting bit position for the search. Zero indicates the first bit, (length(n1)-1) indicates the last bit.

The bit string is scanned in a backward (descending address) direction until a bit set to one is encountered or the start of the bit string is reached. If a bit set to one is encountered, then the search is successful and the instruction is completed by updating evaluated n2 to point to the one bit found. Execution continues with the next inline instruction.

If the start of the bit string is reached and no bit set to one has been found, then the search is unsuccessful and the instruction is completed by updating the current value of pc by using the inline literal.

Exceptions: 0 - integer range exception

10.3.1.27 LOOP DOWN TO ZERO

Opcode: 124

Syntax: LPDNZ n1, lit

n1: source/destination of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than zero, then it is decremented by one and the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

10.3.1.28 LOOP UP

Opcode: 122

Syntax: LPUP n1, n2, lit

n1: source of type integer
n2: source/destination of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is less than evaluated n2, then it is incremented by one and a branch is taken by updating the current value of pc by using the inline literal. Otherwise, execution continues inline.

Exceptions: 0 - integer range exception

10.3.1.29 LOOP DOWN

Opcode: 123

Syntax: LPDN n1, n2, lit

n1: source of type integer
n2: source/destination of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is greater than evaluated n2, then it is decremented by one and a branch is taken by updating the current value of pc by using the inline literal. Otherwise, execution continues inline.

Exceptions: 0 - integer range exception

10.3.1.30 BRANCH IF NULL POINTER

Opcode: 98

Syntax: BNPTR n1, lit

n1: source of type pointer lit: inline literal

Description: Evaluated n1 is examined to determine if it is a null pointer. If it is, then a relative branch is taken by updating the offset portion of current pc by using the inline literal. Otherwise, execution continues inline.

10.3.1.31 BRANCH IF NOT NULL POINTER

Opcode: 99

Syntax: BNNPTR n1, lit

n1: source of type pointer

lit: inline literal

Description: Evaluated n1 is examined to determine if it is a null pointer. If it is not, then a relative branch is taken by updating the offset portion of current pc by using the inline literal. Otherwise, execution contin-

ues inline.

10.3.1.32 SELF RELATIVE BRANCH

Opcode: 97

Syntax: BREL lit

lit: inline literal

Description: The offset portion of pc is updated by using the inline literal.

10.3.1.33 PROCEDURE OBJECT RELATIVE BRANCH

Opcode: 96

Syntax: BR n1

n1: source of type integer

Description: The current pc offset is replaced by eval(n1) + PbP.offset.

10.3.2 Integer Arithmetic and Bit String Instructions

10.3.2.1 CLEAR INTEGER

Opcode: 16

Syntax: ICLR n1

n1: destination of type integer

Description: Each bit of evaluated n1 is set to zero.

10.3.2.2 CLEAR BIT STRING

Opcode: 18

Syntax: BCLR n1

n1: destination of type bit string

Description: Each bit of evaluated n1 is set to zero.

10.3.2.3 SET INTEGER

Opcode: 17

Syntax: ISET n1

n1: destination of type integer

Description: Each bit of evaluated n1 is set to one.

10.3.2.4 SET BIT STRING

Opcode: 19

Syntax: SET n1

n1: destination of type bit string

Description: Each bit of evaluated n1 is set to one.

10.3.2.5 SET TO ONE

Opcode: 20

Syntax: SETONE n1

n1: destination of type integer

Description: Evaluated n1 is assigned the integer value 1.

Exceptions: 0 - integer range exception

10.3.2.6 COMPLEMENT WITH ONE OPERAND

Opcode: 21

Syntax: CMPL1 n1

n1: source/destination of type bit string

Description: Evaluated n1 is logically complemented.

10.3.2.7 COMPLEMENT

Opcode: 22

Syntax: CMPL n1, n2

n1: source of type bit string
n2: destination of type bit string

Description: Evaluated n1 is logically complemented and the result stored at resolved n2.

10.3.2.8 AND WITH TWO OPERANDS

Opcode: 23

Syntax: AND2 n1, n2

n1: source of type bit string
n2: source/destination of type bit string

Description: Evaluated n1 is logically ANDed with evaluated n2 and the result is stored at resolved n2.

10.3.2.9 AND

Opcode: 24

Syntax: AND n1, n2, n3

n1: source of type bit string
n2: source of type bit string
n3: destination of type string

Description: Evaluated n1 is logically ANDed with evaluated n2 and the result is stored at resolved n3.

10.3.2.10 OR WITH TWO OPERANDS

Opcode: 25

Syntax: OR2 n1, n2

n1: source of type bit string
n2: source/destination of type bit string

Description: Evaluated n1 is logically ORed with evaluated n2 and the result is stored at resolved n2.

10.3.2.11 OR

Opcode: 26

Syntax: OR n1, n2, n3

n1: source of type bit string
n2: source of type bit string
n3: destination of type bit string

Description: Evaluated n1 is logically ORed with evaluated n2 and the result is stored at resolved n3.

10.3.2.12 AND COMPLEMENT WITH TWO OPERANDS

Opcode: 27

Syntax: ANDNOT2 n1, n2

n1: source of type bit string
n2: source/destination of type bit string

Description: Evaluated n2 is logically ANDed with the logical complement of evaluated n1. The result is stored at resolved n2.

10.3.2.13 AND COMPLEMENT

Opcode: 28

Syntax: ANDNOT n1, n2, n3

n1: source of type bit string
n2: source of type bit string
n3: destination of type bit string

Description: Evaluated n2 is logically ANDed with the logical complement of evaluated n1. The result is stored at resolved n3.

10.3.2.14 EXCLUSIVE OR

Opcode: 29

Syntax: XOR n1, n2, n3

n1: source of type bit string
n2: source of type bit string
n3: destination of type bit string

Description: Evaluated n1 is logically exclusive OR'ed with evaluated n2. The result is stored at resolved n3.

10.3.2.15 NEGATE INTEGER WITH ONE OPERAND

Opcode: 30

Syntax: INEG1 n1

n1: source/destination of type integer

Description: Evaluated n1 is subtracted from zero and the result is stored at resolved n1.

Exceptions: 0 - integer range exception

10.3.2.16 NEGATE INTEGER

Opcode: 31

Syntax: INEG n1, n2

n1: source of type integer
n2: destination of type integer

Description: Evaluated n1 is subtracted from zero and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

10.3.2.17 ABSOLUTE VALUE INTEGER

Opcode: 32

Syntax: IABS n1, n2

n1: source of type integer
n2: destination of type integer

Description: The absolute value of evaluated n1 is stored at resolved n2.

Exceptions: 0 - integer range exception

10.3.2.18 INCREMENT INTEGER WITH ONE OPERAND

Opcode: 33

Syntax: IINC1 n1

n1: source/destination of type integer

Description: Evaluated n1 is incremented by one and the result is stored at resolved n1.

Exceptions: 0 - integer range exception

10.3.2.19 INCREMENT INTEGER

Opcode: 34

Syntax: IINC n1, n2

n1: source of type integer
n2: destination of type integer

Description: Evaluated n1 is incremented by one and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

10.3.2.20 DECREMENT INTEGER WITH ONE OPERAND

Opcode: 35

Syntax: IDEC1 n1

n1: source/destination of type integer

Description: Evaluated n1 is decremented by one and the result is stored at resolved n1.

Exceptions: 0 - integer range exception

10.3.2.21 DECREMENT INTEGER

Opcode: 36

Syntax: IDEC n1, n2

n1: source of type integer
n2: destination of type integer

Description: Evaluated n1 is decremented by one and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

10.3.2.22 ADD INTEGER WITH TWO OPERANDS

Opcode: 37

Syntax: IADD2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

10.3.2.23 ADD INTEGER

Opcode: 38

Syntax: IADD n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

10.3.2.24 SUBTRACT INTEGER WITH TWO OPERANDS

Opcode: 39

Syntax: ISUB2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Description: Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

10.3.2.25 SUBTRACT INTEGER

Opcode: 40

Syntax: ISUB n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Description: Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

10.3.2.26 MULTIPLY INTEGER WITH TWO OPERANDS

Opcode: 41

Syntax: IMUL2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Description: Evaluated n1 is multiplied by evaluated n2 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

10.3.2.27 MULTIPLY INTEGER

Data General Corporation
Company Confidential

Opcode: 42

Syntax: IMUL n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Description: Evaluated n1 is multiplied by evaluated n2 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

10.3.2.28 DIVIDE INTEGER

Opcode: 44

Syntax: IDIV n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := TRUNC (n2/n1)

Description: Evaluated n2 is divided by evaluated n1 and the quotient result is stored at resolved n3. The operation is performed using an integer arithmetic algorithm such that the dividend n2 and the discarded remainder have the same sign.

Exceptions: 0 - integer range exception
1 - integer divide by zero exception

10.3.2.29 REMAINDER INTEGER

Opcode: 45

Syntax: IREM n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := n2 - TRUNC (n2/n1) * n1

Description: Evaluated n2 is divided by evaluated n1 and the remainder result is stored at resolved n3. The operation is performed using an integer arithmetic algorithm such that the result remainder n3 has the same sign as the dividend.

Exceptions: 0 - integer range exception
1 - integer divide by zero exception

Notes: If the value of evaluated n1 is 1 or -1, then the result of this operation will be zero.

10.3.3 Miscellaneous Instructions

10.3.3.1 MOVE BIT STRING

Opcode: 49

Syntax: SMOV n1, n2

n1: source of type bit string
n2: destination of type bit string

Description: Evaluated n1 is moved to resolved n2.

Notes: If the lengths are not equal the results are unpredictable.

10.3.3.2 INTEGER MOVE

Opcode: 48

Syntax: IMOV n1, n2

n1: source of type integer
n2: destination of type integer

Data General Corporation
Company Confidential

Description: Evaluated n1 is stored at resolved n2. The lengths of n1 and n2 need not be equal.

Exceptions: 0 - integer range exception

10.3.3.3 POINTER MOVE

Opcode: 50

Syntax: PMOV n1, n2

n1: source of type pointer
n2: destination of type pointer

Description: Evaluated n1 is stored at resolved n2. All pointer trap bits except the "Pointer Fault" are copied. If the "Pointer Fault" bit is asserted a "Pointer Fault" is signalled.

10.3.3.4 CREATE POINTER

Opcode: 51

Syntax: PTR n1, n2

n1: source of any type
n2: destination of type pointer

Description: Create a pointer from resolved n1 and store it at resolved n2. If resolved n1 has the same UID as resolved n2, store an intra-object pointer, otherwise store a full UID pointer.

10.3.3.5 CREATE GENERAL POINTER

Opcode: 52

Syntax: GPTR n1, n2

n1: source of any type
n2: destination of type pointer

Description: Create a pointer from resolved n1 and store it at resolved n2. Always store a full UID pointer.

10.3.3.6 STORE NULL POINTER

Opcode: 53

Syntax: NULLPTR n1

n1: destination of type pointer

Description: A null format pointer is stored at resolved n1.

10.3.3.7 RESERVE

Opcode: 54

Syntax: RSRV n1, n2

n1: source operand of type integer
n2: destination operand of type pointer

Description: This instruction increments the internal SP (Stack Pointer) by the value of evaluated n1. The previous setting of SP is stored in resolved n2.

Notes: This instruction must generate an extent check if the value of SP "passes through" zero during the increment or is set greater than the object extent.

10.3.3.8 RELEASE

Opcode: 55

Syntax: RLSE n1

n1: source operand of type integer

Description: This instruction releases an area from the stack by replacing the offset of the internal Stack Pointer (SP) by the value of evaluated n1. The UID portion of evaluated n1 is ignored.

Notes: This instruction must generate an extent fault if the value of SP "passes through" FP during the decrement.

10.3.4 Architectural Instructions

10.3.4.1 CALL

Opcode: 2

Syntax: CALL n1, lit, [n3, n4, ...]

n1: source of type pointer
lit: inline literal
[n3, n4,...] : source of any type

Description: This instruction implements Call. Resolved n1 is the target of the call. lit is the number of parameters of the call. The remaining operands are the parameters of the call.

Exceptions: See Chapter 8.

10.3.4.2 RETURN

Opcode: 3

Syntax: RTN

Description: This instruction implements Return.

Exceptions: See Chapter 8.

10.3.4.3 NOP

Opcode: 1

Syntax: NOP

Description: This instruction performs no operation.

--End of Chapter--

Chapter 11
Fortran S-Language

11.1 Data Types and Their Representation

All addresses are bit granular (address at the leftmost bit), locating containerized data. All lengths are bit lengths. All operand lengths are specified in the length field of the operand's NTE. In some cases, however, the opcode may imply a length. Data representation is often implied in the opcode. No type checking is performed; the type field of the NTE is ignored. In case there is conflict between the type or length the opcode specifies and the type or length the NTE specifies, the results are unpredictable.

11.1.1 LOGICAL

A LOGICAL datum is represented as an 8, 16, or 32 bit containerized bit vector. All bits in the container are significant. The representation for TRUE is all bits set (one). The representation for FALSE is all bits reset (zero). If any bit within the container is different from any of the other bits, the value of the logical is undefined. The results of any operation on an undefined logical value are unpredictable.

A LOGICAL datum is addressed at the low-address end, right-justified and "sign" extended.

8 bit logical

representation in HEX	logical value
FF	TRUE
00	FALSE
<all others>	<undefined>

16 bit logical

representation in HEX	logical value
FFFF	TRUE
0000	FALSE
<all others>	<undefined>

32 bit logical

representation in HEX	logical value
FFFFFFFF	TRUE
00000000	FALSE
<all others>	<undefined>

11.1.2 INTEGER

Integer data are represented in 2's complement notation in container sizes of 8, 16 and 32 bits. Integers are addressed from the leftmost (low-address) bit. Integer operations may occur on mixed sized operands. INTEGER data is addressed at its leftmost (i.e. low-address) end, right-justified and sign-filled on the left on fetch, truncated on the left on store. An attempt to store a result with more significance than the result container can hold generates an overflow condition. Overflows are detected on completion of an operation that overflows the container and prior to storing the result. All integer operations are 2's complement operations.

32 Bit Integer

representation in HEX	value in decimal	
7FFFFFFF	2**31 - 1	(most positive)
00000000	0	
80000000	- 2**31	(most negative)

16:19:10
11/Oct/79
Rev. 1

16 Bit Integer

representation in HEX	value in decimal
7FFF	2**15 - 1 (most positive)
0000	0
8000	- 2**15 (most negative)

8 Bit Integer

representation in HEX	value in decimal
7F	2**7 - 1 (most positive)
00	0
80	- 2**7 (most negative)

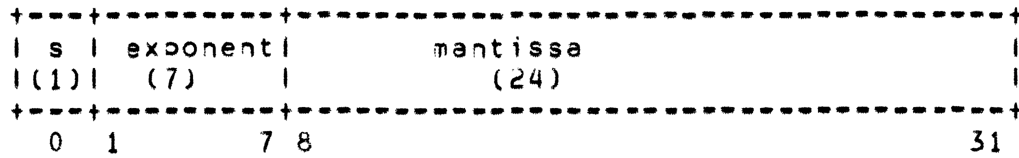
11.1.3 POINTER

A POINTER is a datum which contains or implies a logical address. FORTRAN supports the two types of pointer format: general and intra-object. The instructions which manipulate pointers will produce general object format pointers when the UID of the target container is not that of the pointer value. Intra-object pointers will be produced otherwise. The definition of pointer formats and their architectural usage can be found in Chapter 3.

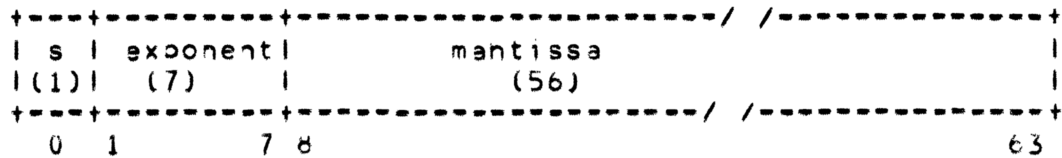
11.1.4 FLOATING POINT

All FLOATING POINT data are in "IBM format" binary floating point and occupy either 32 or 64 bits. In either case, the format is compatible, with variation in only the mantissa section (i.e., an algebraic sign bit followed by a 7-bit, excess-64 hexadecimal exponent, followed by a 24 or 56 bit hexadecimal normalized mantissa).

FLOATING POINT data is addressed at the leftmost (low-address) end, left-justified and zero-filled on the right when fetched.



or



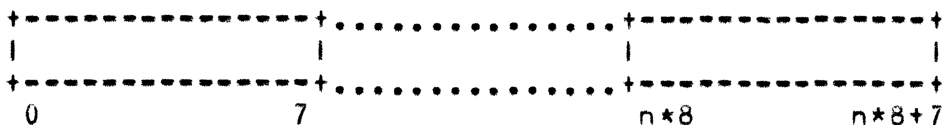
An unnormalized floating point number, a negative zero, or a non-zero exponent with zero mantissa, is considered an illegal floating point number. The use of an unnormalized floating point number yields unpredictable results. A legal zero has all 32 or 64 bits zero.

All floating point values are subject to rounding and/or truncation. A mode within the state of the process determines the selection of rounding or truncation when floating values are generated or moved. The rounding/truncation mode is selected by the "SET EXCEPTION RESPONSE" s-op. Truncation is performed by default.

11.1.5 CHARACTER STRING

Character string is a datum of length of multiples of 8-bit bytes. The minimum length is 0 bytes, and the maximum is 2**29 - 1 bytes.

If the length is not a multiple of 8 bits, the operation and result will be unpredictable. A character datum is always left-justified, blank filled with ASCII blanks.



11.2 FORTRAN Exceptions

A number of various exceptional conditions resulting from execution of FORTRAN s-instructions, including the improper specification or use of data or instructions, cause the generation of a program exception. The following is a list and description of the program exceptions specified for the FORTRAN s-language. Included is a list of the possible response actions for each of the exceptional conditions. In each case the default response is signal.

0 - Integer Range Exception

This exception occurs whenever the result of an integer arithmetic operation is outside the range of representable values for the specified destination operand size. The defined response actions for this exception are the following:

- 00 - signal
- 01 - force largest magnitude, properly signed value
- 10 - reserved
- 11 - rightmost bits of result are stored, left truncation occurs

1 - Integer Divide by Zero Exception

The value of the divisor in a DIVIDE INTEGER or REMAINDER INTEGER instruction is integer zero. The defined responses to this exception are the following:

- 00 - signal
- 01 - force largest magnitude, properly signed value
- 10 - reserved
- 11 - the dividend is stored

2 - Floating Point Overflow

The exponent result of a floating point operation exceeds

the positive representable range of floating point exponent numbers and the result mantissa is not zero. The defined responses to this exception are the following:

00 - signal

01 - force largest magnitude, properly signed value

10 - reserved

11 - rightmost bits are stored, left truncation occurs

3 - Floating Point Underflow

The exponent result of a floating point operation exceeds the negative representable range of floating point exponent numbers and the result mantissa is not zero. The defined responses to this exception are the following:

00 - signal

01 - force the smallest magnitude, properly signed value

10 - force zero

11 - rightmost bits are stored, left truncation occurs

4 - Floating Point Divide by Zero

The value of the divisor in a floating point divide instruction is zero. The defined response actions are the following:

00 - signal

01 - force largest magnitude, properly signed value

10 - reserved

11 - the dividend is stored

11.3 FORTRAN S-Language Operation Definitions

Fortran s-language instructions are defined in detail in the following sections. The operations are classified as Control Instructions, Arithmetic and Logical Instructions, Floating Point Arithmetic Instructions, Character Instructions, Miscellaneous Instructions, Conversion Instructions and Input/Output Assist Instructions.

11.3.1 Control Instructions

11.3.1.1 BRANCH IF ZERO

Opcode: 64

Syntax: BZ n1, lit

n1: source of type integer, floating or logical
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.2 BRANCH IF NOT ZERO

Opcode: 65

Syntax: BNZ n1, lit

n1: source of type integer, floating or logical
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is not equal to zero, then the current value of pc is updated by using the inline literal.

Otherwise, execution continues inline.

11.3.1.3 BRANCH IF LESS THAN ZERO

Opcode: 66

Syntax: BLZ n1, lit

n1: source of type integer or floating
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.4 BRANCH IF LESS THAN OR EQUAL TO ZERO

Opcode: 69

Syntax: BLEZ n1, lit

n1: source of type integer or floating
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than or equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.5 BRANCH IF GREATER THAN ZERO

Opcode: 68

Syntax: BGZ n1, lit

n1: source of type integer or floating
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than zero, then the current value of

pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.6 BRANCH IF GREATER THAN OR EQUAL TO ZERO

Opcode: 67

Syntax: BGEZ n1, lit

n1: source of type integer or floating
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than or equal to zero, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.7 BRANCH IF INTEGER EQUAL

Opcode: 72

Syntax: IBE n1, n2, lit

n1: source of type integer or logical
n2: source of type integer or logical
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are equal, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.8 BRANCH IF INTEGER NOT EQUAL

Opcode: 73

Syntax: IBNE n1, n2, lit

n1: source of type integer or logical
n2: source of type integer or logical
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.9 BRANCH IF INTEGER LESS THAN

Opcode: 74

Syntax: IBL n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is less than n2, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.10 BRANCH IF INTEGER LESS THAN OR EQUAL

Opcode: 77

Syntax: IBLE n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: The evaluated n1 is compared to the evaluated n2. If it is less than or equal to n2, then the current value of pc is updated by the value of the inline literal. Otherwise, execution continues inline.

11.3.1.11 BRANCH IF FLOATING EQUAL

Opcode: 82

Syntax: FBE n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If n1 is equal to n2 then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.12 BRANCH IF FLOATING NOT EQUAL

Opcode: 83

Syntax: FBNE n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.13 BRANCH IF FLOATING LESS THAN

Opcode: 84

Syntax: FBL n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is less than n2, then the current value of pc is updated by using the inline literal. Otherwise, execution continues inline.

11.3.1.14 BRANCH IF FLOATING LESS THAN OR EQUAL

Data General Corporation
Company Confidential

Opcode: 85

Syntax: FBLE n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is less than or equal to n2, then the current value of pc is updated by the value of the inline literal. Otherwise, execution continues inline.

11.3.1.15 BRANCH IF CHARACTER EQUAL

Opcode: 86

Syntax: CBE n1, n2, lit

n1: source of type character string
n2: source of type character string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are equal, then a relative branch is taken by updating the offset portion of current pc by the value of the inline literal. Otherwise execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string.

11.3.1.16 BRANCH IF CHARACTER NOT EQUAL

Opcode: 87

Syntax: CBNE n1, n2, lit

n1: source of type character string
n2: source of type character string

lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then a relative branch is taken by updating the offset portion of current pc by the value of the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string.

11.3.1.17 BRANCH IF CHARACTER LESS THAN

Opcode: 91

Syntax: CBL n1, n2, lit

n1: source of type character string
n2: source of type character string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is less than evaluated n2, then a relative branch is taken by updating the offset portion of current pc by the value of the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string.

11.3.1.18 BRANCH IF CHARACTER LESS THAN OR EQUAL

Opcode: 92

Syntax: CBLE n1, n2, lit

n1: source of type character string
n2: source of type character string

lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If it is less than or equal to evaluated n2, then a relative branch is taken by updating the offset portion of current pc by the value of the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string.

11.3.1.19 LOOP DOWN TO ZERO

Opcode: 124

Syntax: LPDNZ n1, lit

n1: source/destination of type integer
lit: inline literal

Description: Evaluated n1 is decremented by 1 and stored at resolved n1. If the resulting value is greater than zero, then the current value of pc is updated by the value of the inline literal. Otherwise, execution continues inline.

11.3.1.20 LOOP UP

Opcode: 122

Syntax: LPUP n1, n2, lit

n1: source of type integer
n2: source/destination of type integer
lit: inline literal

Description: The evaluated n2 is compared to evaluated n1. If it is less than n1, then it is incremented by one and the current value of the pc is updated by the value of the inline literal.

Otherwise, execution continues inline.

11.3.1.21 ADD AND BRANCH IF LESS THAN OR EQUAL

Opcode: 119

Syntax: ADDBLE n1, n2, n3, lit

n1: source of type integer
n2: source/destination of type integer
n3: source of type integer
lit: literal, signed relative offset of PC

Description: Evaluated n1 is added to evaluated n2 and stored at resolved n2. The result is checked against evaluated n3. If the new n2 is less than or equal to n3, then the current value of pc is updated by the value of the inline literal. Otherwise, execution continues inline.

11.3.1.22 ADD AND BRANCH IF GREATER

Opcode: 118

Syntax: ADDBG n1, n2, n3, lit

n1: source of type integer
n2: source/destination of type integer
n3: source of type integer
lit: literal, signed relative offset of PC

Description: Evaluated n1 is added to evaluated n2 and stored at resolved n2. The new result is checked against evaluated n3. If the new n2 is greater than n3 then the current value of pc is updated by the value of the inline literal. Otherwise, execution continues inline.

Data General Corporation
Company Confidential

11.3.1.23 SELF RELATIVE BRANCH

Opcode: 97

Syntax: BREL lit

lit: inline literal

Semantics:

Description: The current value of pc is updated by the value of the inline literal.

11.3.1.24 PROCEDURE OBJECT RELATIVE BRANCH

Opcode: 96

Syntax: BR n1

n1: source of type integer

Description: The current value of PC offset is replaced by eval(n1) + PBP.offset.

11.3.2 Integer Arithmetic and Logical Instructions

11.3.2.1 CLEAR

Opcode: 18

Syntax: CLR n1

n1: source/destination of any type

Description: Each bit of evaluated n1 is set to zero.

11.3.2.2 SET

Opcode: 19

Syntax: SET n1

n1: source/destination of type integer or logical

Description: Each bit of evaluated n1 is set to one.

11.3.2.3 SET TO ONE

Opcode: 20

Syntax: SETONE n1

n1: source/destination of type integer

Semantics: n1 := 1

Description: Evaluated n1 is set to integer 1.

11.3.2.4 INTEGER MOVE

Opcode: 48

Syntax: IMOV n1, n2

n1: source of type logical or integer
n2: destination of type logical or integer

Semantics: n2 := n1

Description: Evaluated n1 is stored at resolved n2.

Exceptions: 0 - integer range exception

Notes: An integer range exception can occur when moving integers of different container sizes. It cannot occur with logical to logical moves.

11.3.2.5 COMPLEMENT

Opcode: 22

Syntax: CMPL n1, n2

n1: source of type logical
n2: destination of type logical

Semantics: n2 := NOT n1

Description: Evaluated n1 is logically complemented and the result is stored at resolved n2.

11.3.2.6 AND

Opcode: 24

Syntax: AND n1, n2, n3

n1: source of type logical or integer
n2: source of type logical or integer
n3: destination of type logical or integer

Semantics: n3 := n1 AND n2

Description: Evaluated n1 is logically AND'ed with evaluated n2 and the result is stored at resolved n3. Each bit in n1 is bitwise AND'ed with the positionally corresponding bit in n2. The corresponding bit of n3 is set to the result.

11.3.2.7 OR

Opcode: 26

Syntax: OR n1, n2, n3

n1: source of type logical or integer
n2: source of type logical or integer
n3: destination of type logical or integer

Semantics: n3 := n1 OR n2

Description: Evaluated n1 is logically inclusively OR'ed with evaluated n2 and the result is stored at resolved n3. Each bit in n1 is bitwise OR'ed with the positionally corresponding bit in n2. The corresponding bit of n3 is set to the result.

11.3.2.8 EXCLUSIVE OR

Opcode: 29

Syntax: XOR n1, n2, n3

n1: source of type logical or integer
n2: source of type logical or integer
n3: destination of type logical or integer

Semantics: n3 := n1 XOR n2

Description: Evaluated n1 is logically exclusive OR'ed with evaluated n2 and the result is stored at resolved n3. Each bit in n1 is bitwise XOR'ed with the positionally corresponding bit in n2. The corresponding bit of n3 is set to the result.

11.3.2.9 EQUIVALENCE

Opcode: 16

Syntax: EQV n1, n2, n3

n1: source of type logical or integer
n2: source of type logical or integer
n3: destination of type logical or integer

Description: Evaluated n1 is logically exclusive NOR'ed with evaluated n2 and the result is stored at resolved n3. Each bit in n1 is bitwise XOR'ed with the positionally corresponding bit in n2. The resulting bit is complemented. The corresponding bit of n3 is

set to this result.

11.3.2.10 NEGATE INTEGER

Opcode: 31

Syntax: INEG n1, n2

n1: source of type integer
n2: destination of type integer

Semantics: n2 := - n1

Description: Evaluated n1 is "twos complemented" and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.2.11 ABSOLUTE INTEGER

Opcode: 32

Syntax: IABS n1, n2

Semantics: n2 := |n1|

n1: source of type integer
n2: destination of type integer

Description: The absolute value of evaluated n1 is computed and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.2.12 INCREMENT INTEGER WITH 2 OPERANDS

Opcode: 34

Syntax: IINC n1, n2

n1: source of type integer

n2: destination of type integer

Semantics: n2 := n1 + 1

Description: Evaluated n1 is incremented by one and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.2.13 INCREMENT INTEGER

Opcode: 33

Syntax: IINC1 n1

n1: source/destination of type integer

Semantics: n1 := n1 + 1

Description: Evaluated n1 is incremented by one and the result is stored at resolved n1.

Exceptions: 0 - integer range exception

11.3.2.14 DECREMENT INTEGER WITH 2 OPERANDS

Opcode: 36

Syntax: IDEC n1, n2

n1: source of type integer
n2: destination of type integer

Semantics: n2 := n1 - 1

Description: Evaluated n1 is decremented by one and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.2.15 DECREMENT INTEGER

Opcode: 35

Syntax: IDEC1 n1

n1: source/destination of type integer

Semantics: n1 := n1 - 1

Description: Evaluated n1 is decremented by one and the result is stored at resolved n1.

Exceptions: 0 - integer range exception

11.3.2.16 ADD INTEGER WITH 2 OPERANDS

Opcode: 37

Syntax: IADD2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Semantics: n2 := n2 + n1

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.2.17 ADD INTEGER

Opcode: 38

Syntax: IADD n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := n2 + n1

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

11.3.2.18 SUBTRACT INTEGER WITH 2 OPERANDS

Opcode: 39

Syntax: ISUB2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Semantics: n2 := n2 - n1

Description: Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.2.19 SUBTRACT INTEGER

Opcode: 40

Syntax: ISUB n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := n2 - n1

Description: Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

11.3.2.20 MULTIPLY INTEGER WITH 2 OPERANDS

Opcode: 41

Syntax: IMUL2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Semantics: n2 := n2 * n1

Description: Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.2.21 MULTIPLY INTEGER

Opcode: 42

Syntax: IMUL n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := n2 * n1

Description: Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

11.3.2.22 DIVIDE INTEGER WITH 2 OPERANDS

Opcode: 43

Syntax: IDIV2 n1, n2

n1: source operand of type integer
n2: source/destination of type integer

Semantics: n2 := TRUNC(n2/n1)

Description: Evaluated n2 is divided by evaluated n1 and the quotient result is stored at resolved n2. The division is done using an integer arithmetic algorithm and in such a way that the

dividend n2 and the discarded remainder have the same signs.

Exceptions: 0 - integer range exception
1 - integer divide by zero exception

11.3.2.23 DIVIDE INTEGER

Opcode: 44

Syntax: IDIV n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := TRUNC(n2 / n1)

Description: Evaluated n2 is divided by evaluated n1 and the quotient result is stored at resolved n3. The division is done using an integer arithmetic algorithm and in such a way that the dividend n2 and the discarded remainder have the same signs.

Exceptions: 0 - integer range exception
1 - integer divide by zero exception

11.3.2.24 REMAINDER INTEGER

Opcode: 45

Syntax: IREM n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := n2 - n1 * TRUNC(n2/n1)

Description: Evaluated n2 is divided by evaluated n1 and the remainder result is stored at resolved n3. The division is done using an integer arithmetic algorithm which produces a remain-

der with the same sign as that of the dividend n2 and whose absolute value is less than that of the divisor n1.

Exceptions: 0 - integer range exception
1 - integer divide by zero exception

Notes: If the divisor evaluated n1 is 1 or -1, then the result of this instruction is zero.

11.3.3 Floating Point Arithmetic Instructions

11.3.3.1 NEGATE FLOATING

Opcode: 04

Syntax: FNEG n1, n2

n1: source of type floating point
n2: destination of type floating point

Semantics: n2 := - n1

Description: Evaluated n1 is negated and the result is stored at resolved n2.

11.3.3.2 ABSOLUTE FLOATING

Opcode: 05

Syntax: FABS n1, n2

n1: source of type floating point
n2: destination of type floating point

Semantics: n2 := |n1|

Description: The absolute value of evaluated n1 is computed and the result is stored at resolved n2.

11.3.3.3 MOVE FLOATING

Opcode: 06

Syntax: FMOV n1, n2

n1: source of type floating point
n2: destination of type floating point

Semantics: n2 := n1

Description: Evaluated n1 is stored at resolved n2.

Notes: The lengths of the floating point operands may differ. The round/truncate mode effects the mantissa when a 64 bit floating point is moved to a 32 bit floating point. When a 32 bit floating point is moved to a 64 bit floating point, the rightmost 32 bits of the mantissa are zeroed. For operands of equal length a bit copy is performed.

11.3.3.4 ADD FLOATING WITH 2 OPERANDS

Opcode: 07

Syntax: FADD2 n1, n2

n1: source of type floating point
n2: source/destination of type floating point

Semantics: n2 := n2 + n1

Description: Evaluated n1 is added to evaluated n2 using a floating point algorithm and the result is stored at resolved n2.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

11.3.3.5 ADD FLOATING

Opcode: 08

Syntax: FADD n1, n2, n3

n1: source of type floating point
n2: source of type floating point
n3: destination of type floating point

Semantics: n3 := n2 + n1

Description: Evaluated n1 is added to evaluated n2 using a floating point algorithm and the result is stored at resolved n3.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

11.3.3.6 SUBTRACT FLOATING WITH 2 OPERANDS

Opcode: 09

Syntax: FSUB2 n1, n2

n1: source of type floating point
n2: source/destination of type floating point

Semantics: n2 := n2 - n1

Description: Evaluated n1 is subtracted from evaluated n2 using a floating point algorithm and the result is stored at resolved n2.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

11.3.3.7 SUBTRACT FLOATING

Opcode: 10

Syntax: FSUB n1, n2, n3

n1: source of type floating point
n2: source of type floating point
n3: destination of type floating point

Semantics: n3 := n2 - n1

Description: Evaluated n1 is subtracted from evaluated n2 using a floating point algorithm and the result is stored at resolved n3.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

11.3.3.8 MULTIPLY FLOATING WITH 2 OPERANDS

Opcode: 11

Syntax: FMUL2 n1, n2

n1: source of type floating point
n2: source/destination of type floating point

Semantics: n2 := n2 * n1

Description: Evaluated n2 is multiplied by evaluated n1 using a floating point algorithm and the result is stored at resolved n2.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

11.3.3.9 MULTIPLY FLOATING

Opcode: 12

Syntax: FMUL n1, n2, n3

n1: source of type floating point
n2: source of type floating point
n3: destination of type floating point

Semantics: n3 := n2 * n1

Description: Evaluated n2 is multiplied by evaluated n1 using a floating point algorithm and the result is stored at resolved n3.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

11.3.3.10 DIVIDE FLOATING

Opcode: 14

Syntax: FDIV n1, n2, n3

n1: source of type floating point
n2: source of type floating point
n3: destination of type floating point

Semantics: n3 := n2 / n1

Description: Evaluated n2 is divided by evaluated n1 using a floating point algorithm and the result is stored at resolved n3.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception
4 - floating point divide by zero exception

11.3.4 Character Instructions

11.3.4.1 MOVE CHARACTERS

Opcode: 100

Syntax: CMOV n1, n2

n1: source of type character
n2: destination of type character

Semantics: n2 := n1

Description: Evaluated n1 is stored at resolved n2. If the length of n1 is greater than the length of n2, then the move is truncated on the right. If the length of n1 is less than the length of n2, then evaluated n2 is ASCII blank filled on the right after the string named by n1 is exhausted.

11.3.4.2 MOVE SPACES

Opcode: 101

Syntax: CMOVS n1

n1: destination of type character

Semantics: n1 := <ASCII blanks>

Description: Evaluated n1 is filled with ASCII blanks.

11.3.5 Miscellaneous Instructions

11.3.5.1 SET EXCEPTION RESPONSE

Opcode: 125

Syntax: SER n1, n2

n1: source of type integer
n2: source of type integer

Description: Set the response of an exception. Evaluated n1 is the exception number. Evaluated n2 is the scenario number for the response of this exception.

Exceptions: 0 - integer range exception

Notes: This instruction is also used to set the floating point truncation/rounding mode.

11.3.5.2 GET EXCEPTION RESPONSE

Opcode: 126

Syntax: GER n1, n2

n1: source of type integer
n2: destination of type integer

Description: Evaluated n1 is used as an index into the current exception responses. The scenario for the exception is stored at resolved n2.

Exceptions: 0 - integer range exception

11.3.5.3 MOVE POINTER

Opcode: 50

Syntax: PTR n1, n2

n1: source of type pointer
n2: destination of type pointer

Semantics: n2 := n1

Description: Evaluated n1 is stored at resolved n2.

11.3.5.4 CREATE POINTER

Opcode: 51

Syntax: ADDR n1, n2

n1: source of any type
n2: destination of type pointer

Description: Create a pointer from resolved n1 and store it at resolved n2. If resolved n1 has the

same UID as resolved n2, store an intra-object format pointer, otherwise store a general pointer.

11.3.6 Conversion Instructions

11.3.6.1 CONVERT INTEGER TO FLOATING

Opcode: 102

Syntax: IFCVT n1, n2

n1: source of type integer
n2: destination of type floating point

Semantics: n2 := FLOAT(n1)

Description: Evaluated n1 is converted to a floating point number and the result is stored at resolved n2. The Round/Truncate mode setting determines whether the resulting floating point number is rounded or truncated prior to storing the result.

11.3.6.2 CONVERT FLOATING TO INTEGER

Opcode: 103

Syntax: FICVT n1, n2

n1: source of type floating point
n2: destination of type integer

Semantics: n2 := INT(n1)

Description: Evaluated n1 is converted to an integer and the result is stored at resolved n2. The Rounding /Truncation mode determines whether the result is rounded or truncated before conversion.

Exceptions: 0 - integer range exception

11.3.7 Input/Output Assist Instructions

11.3.7.1 CONVERT INTEGER TO CHARACTER STRING

Opcode: 104

Syntax: ICCVT n1, n2

n1: source operand of type integer
n2: destination of type character string

Description: Evaluated n1 is converted to a decimal character string and stored in the resolved n2. The character string is right justified, with blank fill on the left. If the integer is negative, the first (leftmost) non-blank character is a minus sign ('-'). If the character string is too small to hold the full integer value, the entire character string is filled with the asterisk character ('*').

11.3.7.2 CONVERT CHARACTER STRING TO INTEGER

Opcode: 105

Syntax: CICVT n1, n2

n1: source operand of type character string
n2: destination of type integer

Description: This instruction converts the ASCII character string given by evaluated n1 into an integer value which is stored at resolved n2. The character string is interpreted as a decimal number. The character string may contain only characters in the range "0" - "9" with the exception of blanks and [optionally] a leading sign. The length of the character

string determines the number of characters translated.

Exceptions: 0 - integer range exception

Notes: An integer range exception is generated if the integer container at resolved n2 cannot hold the integer representation of character string n1. Blanks preceeding a sign are ignored, all other blanks are treated as numeric zeroes. If non-numeric characters occur (excluding exceptions above) the value of the destination integer is set to 0.

11.3.7.3 CONVERT FLOATING TO CHARACTER STRING

Opcode: 106

Syntax: FCCVT n1, n2, n3

n1: source operand of type floating
n2: source operand of type integer
n3: destination of type character string

Description: This instruction converts the floating point number given by evaluated n1 into an ASCII character string at resolved n3. "Character rounding" is performed on the conversion. The decimal point is placed in the character string at the n2+1 th character position from the right. The length of the character string is determined by its name table entry. If the character string is insufficient in size, or there are too few characters on the left of the decimal digit to represent the floating number, asterisks ('*') are placed in the entire character string.

11.3.8 Architectural Instructions

11.3.8.1 CALL

Opcode: 02

Syntax: CALL n1, lit, [n3, n4, ...]

n1: source of type pointer
lit: inline literal
[n3, n4,...] : source of any type

Description: This instruction implements Call. Resolved n1 is the target of the call. lit is the number of parameters of the call. The remaining operands are the parameters of the call.

Exceptions: See Chapter 8.

11.3.8.2 RETURN

Opcode: 03

Syntax: RTN

Description: This instruction implements Return.

Exceptions: See Chapter 8.

11.3.8.3 NOP

Opcode: 01

Syntax: NOP

Description: This instruction performs no operation.

--End of Chapter--

12.1 Data Types and Their Representation

Data representation is often implied by the opcode. Excepting decimal s-ops, no type checking is performed. If there is a conflict between the opcode and the NTE type or length, or if the NTE specifies an invalid length for a data type, the results are unpredictable.

Truncation is implied in all numeric calculations and conversion instructions unless otherwise stated.

12.1.1 Integer

COMPUTATIONAL data is represented as signed, two's complement binary integers, sized 8, 16, 24, 32, 40, 48, 56, and 64 bits. With the exception of offset names, unsigned integers are represented as positive 2's complement numbers (i.e. with leading zeros). Integer operations may occur on mixed sized operands. Integer data is right justified and sign filled on the left when fetched, and truncated on the left when stored. An attempt to store a result with more significance than the result container can hold generates an integer range exception.

12.1.2 Floating Point

All floating point data are in "IBM format" binary floating point and occupy either 32 or 64 bits. The format consists of an algebraic sign bit followed by a 7 bit, excess 64 hexadecimal exponent, followed by a 24 or 56 bit hexadecimal normalized mantissa. COMPUTATIONAL-1 data is represented as 32 bit floating point. COMPUTATIONAL-2 data is represented as 64 bit floating point. The interpreter handles operations on mixes of the two precisions by converting operands to 64 bit floating. Floating point data is left justified and zero filled on the right when fetched. All operations are done using a 64 bit algorithm, with right truncation of results on store back (if necessary). The use of a "dirty" zero or an unnormalized number gives unpredictable results (a "clean" zero has all bits zero).

12.1.3 Decimal

The name table type field is used to encode the various decimal data types. The encoding of the four bit type field is as follows:

- 0000: Character decimal, unsigned.
- 0001: Character decimal, sign is separate and leading.
- 0010: Character decimal, sign is separate and trailing.
- 0011: Character decimal, sign is leading overpunched.
- 0100: Character decimal, sign is trailing overpunched.
- 0101: Packed decimal, unsigned.
- 0110: Packed decimal, signed.

The decimal operations are defined to accept any combination of these decimal formats. Decimal operands have lengths of 1 to 18 decimal digits plus sign, for both packed and unpacked numbers. Decimal data is 0-digit filled on the left when fetched, and left truncated when stored (exclusive of the sign).

A negative decimal zero may be generated if the source operand for a MOVE instruction is negative zero, or if a negative result requires truncation when stored. In no other case will negative zero be generated. A negative zero source operand will be interpreted as positive zero.

12.1.3.1 Packed Decimal

PACKED-DECIMAL data is represented as either signed or unsigned packed decimal numbers. It is not checked for validity. All signed packed decimal numbers have their trailing 4 bits as sign, hexadecimal "C" for plus and "D" for minus, and all other characters will be treated as plus. All unsigned packed decimal numbers have their last 4 bits disregarded (the trailing 4 bits are always treated as positive).

12.1.3.2 Character Decimal

Numeric USAGE IS DISPLAY data is represented as character decimal. It is not checked for validity. There is an instruction

to check the range and do a MOVE if in range. Valid character encodings for numeric USAGE IS DISPLAY are digit values "0" through "9" and the valid sign encodings. In all cases, the space character is a valid substitute for the character zero. Operations on any other encodings yield unpredictable results.

12.1.3.2.1 Unsigned

This representation is generated when a data item is described by a PICTURE which does not contain the operational sign designator S. The numeric value is represented as a string of the characters '0'..'9'. There is no sign character.

12.1.3.2.2 Separate Sign Leading/Trailing

The sign is either '+' or '-' and is the first/last character position of the data field. The digit values are as above, '0'..'9'.

12.1.3.2.3 Overpunched Sign Leading/Trailing

In this representation, the first/last character position of the data field implies both a digit and an operational sign. This encoding is presented herewith along with the EBCDIC encoding for comparison. Note that an ASCII/EBCDIC character translation on a character decimal field performs the correct mapping.

Decimal Digit	ASCII +	ASCII -	ASCII unsigned	EBCDIC +	EBCDIC -	EBCDIC unsigned
0	7B	7D	30	C0	D0	F0
1	41	4A	31	C1	D1	F1
2	42	4B	32	C2	D2	F2
3	43	4C	33	C3	D3	F3
4	44	4D	34	C4	D4	F4
5	45	4E	35	C5	D5	F5
6	46	4F	36	C6	D6	F6
7	47	50	37	C7	D7	F7
8	48	51	38	C8	D8	F8
9	49	52	39	C9	D9	F9
SPACE	2B	2D	20	50	60	40

(all table entries in hexadecimal)

17:18:7
11/Oct/79
Rev. 1

12.1.4 Character Strings

Any operand whose length is a multiple of 8 bits (but not zero) is of type character string (in addition to any other type it may have). Interpretation as a character string is determined by the opcode. For ALL character operands, the FIU and type fields in NTE are overridden. This is not considered a type conflict. The minimum length is 1 byte, and the maximum is 2**29 - 1 bytes. Character data is left justified and ASCII blank filled when fetched, and right truncated when stored. Although translate s-ops are provided, EBCDIC is not directly supported by this s-language.

12.1.5 Translate Table

Character translation instructions specify an operand which is a translate table. The resolved operand name is then the starting address of an array of bytes. A character is translated by interpreting it as an unsigned, byte relative index into the translate table. The byte at this location is the translated character. Index "0" corresponds to the first byte of the translate table. A translate table consists of 256 bytes.

12.1.6 Boolean Vector

Set membership instructions specify an operand which describes a set. The resolved operand name is then the starting address of an array of bits. A character is tested for set membership by interpreting it as an unsigned, bit relative index into the set vector. A one bit at this location indicates that the character is in the set. A zero bit at this location indicates that the character is not in the set. A set vector consists of 256 bits corresponding to character codes zero through 255.

12.1.7 Pointers

Pointers are used in the COBOL s-language by the Architectural instructions. The definition of pointer formats and their architectural usage can be found in Chapter 3.

12.2 COBOL Exceptions

12.2.1 Error Handling

Many arithmetic and conversion operations can produce results which are too long to fit in the specified destination. For COBOL, division by zero, overflow and underflow fall into this category. Explicit size error handling, when specified by the programmer, mandates that the destination field remain unchanged when a size error occurs.

The COBOL S-language has a 1-bit size error flag called SIZE_ERROR which can be set by any arithmetic instruction or by the check move instructions.

Note that when storing a signed value (be it decimal, floating or integer) into a decimal datum whose type field indicates "unsigned", only the magnitude of the value is stored. No size error occurs from the sign change.

12.2.2 Exceptions

A number of conditions, including the improper specification or use of data or instructions, cause the signalling of a program exception. The following is a description of the program exceptions specified for this s-language. Included is a list of the possible response actions for each of the exception conditions. The default response is given by "00".

0 - Integer Range Exception

The result of an integer operation is outside the range of representable values for the destination operand size. The defined response actions for this exception are the following:

00 - Set SIZE_ERROR. Store an unpredictable result.

01 - reserved

10 - reserved

11 - reserved

1 - Integer Divide by Zero Exception

The value of the divisor in an integer quotient or integer

remainder instruction is integer zero. The defined responses to this exception are the following:

00 - Set SIZE_ERROR. Store an unpredictable result.

01 - reserved

10 - reserved

11 - reserved

2 - Floating Point Overflow

The exponent result of a floating point operation exceeds the positive representable range of floating point exponent numbers and the result mantissa is not zero. The defined responses to this exception are the following:

00 - Set SIZE_ERROR. Store an unpredictable result.

01 - reserved

10 - reserved

11 - reserved

3 - Floating Point Underflow

The exponent result of a floating point operation exceeds the negative representable range of floating point exponent numbers and the result mantissa is not zero. The defined responses to this exception are the following:

00 - Set SIZE_ERROR. Store an unpredictable result.

01 - reserved

10 - reserved

11 - reserved

Data General Corporation
Company Confidential

4 - Floating Point Divide by Zero

The value of the divisor in a floating point divide instruction is zero. The defined response actions are the following:

00 - Set SIZE_ERROR. Store an unpredictable result.

01 - reserved

10 - reserved

11 - reserved

5 - Decimal Overflow Exception

The result of a decimal operation exceeds 18 decimal digits. The defined response actions for this exception are the following:

00 - Set SIZE_ERROR. The result is right truncated to 18 digits. The number of truncated digits is stored in the decimal name. That result is stored in the destination left truncated if the destination size is less than 18 digits.

01 - reserved

10 - reserved

11 - reserved

6 - Decimal Divide by Zero Exception

The value of the divisor in a decimal divide instruction is zero. The defined response actions are the following:

00 - Set SIZE_ERROR. Store an unpredictable result.

01 - reserved

10 - reserved

11 - reserved

12.3 COBOL Operation Definitions

The COBOL s-language instructions are defined in detail in the following sections. The operations are classified as Control instructions, Integer Arithmetic instructions, Floating Point Arithmetic instructions, Decimal Arithmetic instructions, String instructions, Size Error instructions, and Architectural instructions.

12.3.1 Control Instructions

12.3.1.1 BRANCH IF ZERO

Opcode:	64
Syntax:	BZ n1, lit
	n1: source of type integer or floating point lit: inline literal
Description:	Evaluated n1 is compared to zero. If it is equal to zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.2 BRANCH IF NOT ZERO

Opcode:	65
Syntax:	BNZ n1, lit
	n1: source of type integer or floating point lit: inline literal
Description:	Evaluated n1 is compared to zero. If it is not equal to zero, then a relative branch is taken by updating the offset portion of

current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.3 BRANCH IF GREATER THAN ZERO

| Opcode: 68

Syntax: BGZ n1, lit

n1: source of type integer or floating point
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.4 BRANCH IF GREATER THAN OR EQUAL TO ZERO

| Opcode: 67

Syntax: BGEZ n1, lit

n1: source of type integer or floating point
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than or equal to zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.5 BRANCH IF LESS THAN ZERO

| Opcode: 66

Syntax: BLZ n1, lit

n1: source of type integer or floating point
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.6 BRANCH IF LESS THAN OR EQUAL TO ZERO

| Opcode: 69

Syntax: BLEZ n1, lit

n1: source of type integer or floating point
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than or equal to zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.7 BRANCH IF INTEGER EQUAL

| Opcode: 72

Syntax: IBE n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise execution continues inline.

12.3.1.8 BRANCH IF INTEGER NOT EQUAL

| Opcode: 73

Syntax: IBNE n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.9 BRANCH IF INTEGER LESS THAN

Opcode: 74

Syntax: IBL n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: If evaluated n1 is less than evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.10 BRANCH IF INTEGER LESS THAN OR EQUAL

Opcode: 77

Syntax: IBLE n1, n2, lit

n1: source of type integer
n2: source of type integer
lit: inline literal

Description: If evaluated n1 is less than or equal to evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

Data General Corporation
Company Confidential

12.3.1.11 BRANCH IF FLOATING EQUAL

Opcode: 82

Syntax: FBE n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.12 BRANCH IF FLOATING NOT EQUAL

Opcode: 83

Syntax: FBNE n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.13 BRANCH IF FLOATING LESS THAN

Opcode: 84

Syntax: FBL n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: If evaluated n1 is less than evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.14 BRANCH IF FLOATING LESS THAN OR EQUAL

| Opcode: 85

Syntax: FBLE n1, n2, lit

n1: source of type floating point
n2: source of type floating point
lit: inline literal

Description: If evaluated n1 is less than or equal to evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.15 BRANCH IF DECIMAL EQUAL

| Opcode: 86

Syntax: DBE n1, n2, lit

n1: source of type decimal
n2: source of type decimal
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise execution continues inline.

12.3.1.16 BRANCH IF DECIMAL NOT EQUAL

| Opcode: 87

Data General Corporation
Company Confidential

Syntax: DBNE n1, n2, lit

n1: source of type decimal
n2: source of type decimal
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.17 BRANCH IF DECIMAL LESS THAN

| Opcode: 88

Syntax: DBL n1, n2, lit

n1: source of type decimal
n2: source of type decimal
lit: inline literal

Semantics: If evaluated n1 is less than evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.18 BRANCH IF DECIMAL LESS THAN OR EQUAL

| Opcode: 89

Syntax: DBLE n1, n2, lit

n1: source of type decimal
n2: source of type decimal
lit: inline literal

Description: If evaluated n1 is less than or equal to evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.19 BRANCH IF DECIMAL EQUAL ZERO

Opcode:	90
Syntax:	DBZ n1, lit n1: source of type decimal lit: inline literal
Description:	Evaluated n1 is compared to zero. If it is equal to zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.20 BRANCH IF DECIMAL NOT EQUAL ZERO

Opcode:	91
Syntax:	DBNZ n1, lit n1: source of type decimal lit: inline literal
Description:	Evaluated n1 is compared to zero. If it is not equal to zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.21 BRANCH IF DECIMAL GREATER THAN ZERO

Opcode:	92
Syntax:	DBGZ n1, lit n1: source of type decimal lit: inline literal
Description:	Evaluated n1 is compared to zero. If it is greater than zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal.

Otherwise, execution continues inline.

12.3.1.22 BRANCH IF DECIMAL GREATER THAN OR EQUAL TO ZERO

| Opcode: 93

Syntax: DBGEZ n1, lit

n1: source of type decimal
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than or equal to zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.23 BRANCH IF DECIMAL LESS THAN ZERO

| Opcode: 94

Syntax: DBLZ n1, lit

n1: source of type decimal
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.24 BRANCH IF DECIMAL LESS THAN OR EQUAL TO ZERO

| Opcode: 95

Syntax: DBLEZ n1, lit

n1: source of type decimal
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is less than or equal to zero, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.25 BRANCH IF CHARACTER EQUAL

Opcode: 96

Syntax: CBE n1, n2, lit

n1: source of type character string
n2: source of type character string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string.

12.3.1.26 BRANCH IF CHARACTER NOT EQUAL

Opcode: 97

Syntax: CBNE n1, n2, lit

n1: source of type character string
n2: source of type character string
lit: inline literal

Description: Evaluated n1 is compared to evaluated n2. If they are not equal, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string.

12.3.1.27 BRANCH IF CHARACTER LESS THAN

| Opcode: 98

Syntax: CBL n1, n2, lit

n1: source of type character string
n2: source of type character string
lit: inline literal

Description: If evaluated n1 is less than evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string.

12.3.1.28 BRANCH IF CHARACTER LESS THAN OR EQUAL

| Opcode: 99

Syntax: CBLE n1, n2, lit

n1: source of type character string
n2: source of type character string
lit: inline literal

Description: If evaluated n1 is less than or equal to evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter

Data General Corporation
Company Confidential

string is ASCII blank filled on the right to the length of the longer string.

12.3.1.29 BRANCH IF CHARACTERS TRANSLATED EQUAL

Opcode: 100

Syntax: CTBE n1, n2, n3, lit

n1: source of type character string
n2: source of type character string
n3: source of type translate table
lit: inline literal

Description: The character strings named by n1 and n2 are translated through the table named by n3. If translated evaluated n1 is equal to translated evaluated n2, then a relative branch is taken by updating the offset portion of current PC using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string. The blank fill occurs before translation.

12.3.1.30 BRANCH IF CHARACTERS TRANSLATED LESS THAN

Opcode: 101

Syntax: CTBL n1, n2, n3, lit

n1: source of type character string
n2: source of type character string
n3: source of type translate table
lit: inline literal

Description: The character strings named by n1 and n2 are translated through the table named by n3. If translated evaluated n1 is less than translated evaluated n2, then a relative branch is taken by updating the offset portion of

current pc, using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string. The blank fill occurs before translation.

12.3.1.31 BRANCH IF CHARACTERS TRANSLATED NOT EQUAL

Opcode: 102

Syntax: CTBNE n1, n2, n3, lit

n1: source of type character string
n2: source of type character string
n3: source of type translate table
lit: inline literal

Description: The character strings named by n1 and n2 are translated through the table named by n3. If translated evaluated n1 is not equal to translated evaluated n2, then a relative branch is taken by updating the offset portion of current PC using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string. The blank fill occurs before translation.

12.3.1.32 BRANCH IF CHARACTERS TRANSLATED LESS THAN OR EQUAL

Opcode: 103

Syntax: CTBLE n1, n2, n3, lit

n1: source of type character string
n2: source of type character string
n3: source of type translate table

lit: inline literal

Description: The character strings named by n1 and n2 are translated through the table named by n3. If translated evaluated n1 is less than or equal to translated evaluated n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

Notes: If the character strings named by n1 and n2 are of different lengths, then the shorter string is ASCII blank filled on the right to the length of the longer string. The blank fill occurs before translation.

12.3.1.33 BRANCH IF IN SET

| Opcode: 104

Syntax: CBINSET n1, n2, n3, lit

n1: source of type character string
n2: source of type boolean vector
n3: destination of type integer
lit: inline literal

Description: The string named by n1 is scanned. If all characters in evaluated n1 are in the set named by n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline. The index of the leftmost character not in the set is stored at resolved n3. Resolved n3 is unchanged if all characters are in the set.

12.3.1.34 BRANCH IF NOT IN SET

| Opcode: 105

Syntax: CBNINSET n1, n2, n3, lit

n1: source of type character string
n2: source of type boolean vector
n3: destination of type integer
lit: inline literal

Description: The string named by n1 is scanned. If any character in evaluated n1 is not in the set named by n2, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline. The index of the leftmost character not in the set is stored at resolved n3. Resolved n3 is unchanged if all characters are in the set.

12.3.1.35 BRANCH IF CHARACTERS SPACES

| Opcode: 106

Syntax: CBS n1, lit

n1: source of type character string
lit: inline literal

Description: The string named by n1 is scanned. If all characters of evaluated n1 are ASCII spaces, then a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.1.36 BRANCH IF CHARACTERS NOT SPACES

| Opcode: 107

Syntax: CBNS n1, lit

n1: source of type character string
lit: inline literal

Description: The string named by n1 is scanned. If any character of evaluated n1 is not an ASCII space, then a relative branch is taken by updating the offset portion of current pc,

using the inline literal. Otherwise, execution continues inline.

12.3.1.37 LOOP DOWN TO ZERO

Opcode: 108

Syntax: LPDNZ n1, lit

n1: source of type integer
lit: inline literal

Description: Evaluated n1 is compared to zero. If it is greater than zero, it is decremented by one and stored at resolved n1; the offset portion of current pc is updated using the inline literal. Otherwise, execution continues inline.

12.3.1.38 PERFORM

Opcode: 109

Syntax: PFM n1, n2

n1: destination of type unsigned integer
n2: source of type unsigned integer

Description: The PC offset of the next inline instruction is stored at resolved n1 and a branch is taken by replacing the current PC offset with eval(n2) + PBP.offset.

12.3.1.39 PERFORM END

Opcode: 110

Syntax: PFMEND n1

n1: source/destination of type unsigned integer

| Description: Evaluated n1 is compared to zero. If is is not equal to zero, then the current PC offset is replaced by evaluated n1 and integer zero is stored at resolved n1. Otherwise, execution continues inline.

12.3.1.40 SELF RELATIVE BRANCH

| Opcode: 111

Syntax: BREL lit

lit: inline literal

Description: An unconditional branch is taken by updating the offset portion of current pc, using the inline literal.

12.3.1.41 PROCEDURE OBJECT RELATIVE BRANCH

| Opcode: 112

Syntax: BR n1

n1: source of type unsigned integer

| Description: A branch is taken by replacing the current PC offset with PBP.offset + eval(n1).

12.3.2 Integer Arithmetic Instructions

12.3.2.1 MOVE INTEGER

| Opcode: 113

Syntax: IMOV n1, n2

n1: source of type integer
n2: destination of type integer

Description: Evaluated n1 is stored at resolved n2.

Exceptions: 0 - integer range exception

12.3.2.2 SET TO ONE

Opcode: 114

Syntax: ISETONE n1

n1: destination of type integer

Description: Resolved n1 is set to integer one.

12.3.2.3 ABSOLUTE INTEGER

Opcode: 115

Syntax: IABS n1, n2

n1: source of type integer
n2: destination of type integer

Description: The absolute value of evaluated n1 is computed and stored at resolved n2.

Exceptions: 0 - integer range exception

12.3.2.4 NEGATE INTEGER

Opcode: 116

Syntax: INEG n1, n2

n1: source of type integer
n2: destination of type integer

Description: Evaluated n1 is negated and stored at resolved n2.

Exceptions: 0 - integer range exception

12.3.2.5 INCREMENT INTEGER WITH ONE OPERAND

| Opcode: 117

Syntax: IINC1 n1

n1: source/destination of type integer

Description: Evaluated n1 is incremented by one and the result is stored at resolved n1.

Exceptions: 0 - integer range exception

12.3.2.6 DECREMENT INTEGER WITH ONE OPERAND

| Opcode: 118

Syntax: IDEC1 n1

n1: source/destination of type integer

Description: Evaluated n1 is decremented by one and the result is stored at resolved n1.

Exceptions: 0 - integer range exception

12.3.2.7 ADD INTEGER WITH 2 OPERANDS

| Opcode: 119

Syntax: IADD2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

Data General Corporation
Company Confidential

12.3.2.8 ADD INTEGER

Opcode:	120
Syntax:	IADD n1, n2, n3
	n1: source of type integer n2: source of type integer n3: destination of type integer
Description:	Evaluated n1 is added to evaluated n2 and the result is stored at resolved n3.
Exceptions:	0 - integer range exception

12.3.2.9 SUBTRACT INTEGER WITH 2 OPERANDS

Opcode:	121
Syntax:	ISUB2 n1, n2
	n1: source of type integer n2: source/destination of type integer
Description:	Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n2.
Exceptions:	0 - integer range exception

12.3.2.10 SUBTRACT INTEGER

Opcode:	122
Syntax:	ISUB n1, n2, n3
	n1: source of type integer n2: source of type integer n3: destination of type integer
Description:	Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

12.3.2.11 MULTIPLY INTEGER WITH 2 OPERANDS

| Opcode: 123

Syntax: IMUL2 n1, n2

n1: source of type integer
n2: source/destination of type integer

Description: Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

12.3.2.12 MULTIPLY INTEGER

| Opcode: 124

Syntax: IMUL n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Description: Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n3.

Exceptions: 0 - integer range exception

12.3.2.13 DIVIDE INTEGER

| Opcode: 125

Syntax: IDIV n1, n2, n3

n1: source of type integer
n2: source of type integer
n3: destination of type integer

Semantics: n3 := TRUNCATE(n2 / n1)

Description: Evaluated n2 is divided by evaluated n1 and the quotient is stored at resolved n3. The discarded remainder has the same sign as the dividend n2.

Exceptions: 0 - integer range exception
1 - integer divide by zero

12.3.2.14 SCALE INTEGER BY 10

Opcode: 126

Syntax: ISCALE lit, n1, n2

lit: literal, signed scale factor
n1: source of type integer
n2: destination of type integer

Semantics: n2 := n1 * 10 ** lit

Description: Evaluated n1 is multiplied by 10 to the power of the integer literal and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

12.3.2.15 SCALE INTEGER WITH ROUNDING

Opcode: 128

Syntax: IRND lit, n1, n2

lit: literal, signed scale factor
n1: source of type integer
n2: destination of type integer

Description: Evaluated n1 is multiplied by 10 to the power of the integer literal and the result is stored at resolved n2. Rounding occurs (the result is incremented by one) if the scale factor is negative and the leftmost trunctaed bit is a one.

Exceptions: 0 - integer range exception

12.3.3 Floating Point Arithmetic Instructions

12.3.3.1 MOVE FLOATING

Opcode: 129

Syntax: FMOV n1, n2

n1: source of type floating point
n2: destination of type floating point

Description: Evaluated n1 is stored at resolved n2.

12.3.3.2 ADD FLOATING WITH 2 OPERANDS

Opcode: 130

Syntax: FADD2 n1, n2

n1: source of type floating point
n2: source/destination of type floating point

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n2.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

12.3.3.3 ADD FLOATING

Opcode: 131

Syntax: FADD n1, n2, n3

n1: source of type floating point
n2: source of type floating point

n3: destination of type floating point

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n3.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

12.3.3.4 SUBTRACT FLOATING WITH 2 OPERANDS

| Opcode: 132

Syntax: FSUB2 n1, n2

n1: source of type floating point
n2: source/destination of type floating point

Description: Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n2.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

12.3.3.5 SUBTRACT FLOATING

| Opcode: 133

Syntax: FSUB n1, n2, n3

n1: source of type floating point
n2: source of type floating point
n3: destination of type floating point

Description: Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n3.

Exceptions: 2 - floating point overflow exception
3 - floating point underflow exception

Data General Corporation
Company Confidential

12.3.3.6 FLOATING NEGATE

Opcode:	134
Syntax:	FNEG n1,n2 n1: source of type floating point n2: destination of type floating point
Description:	Evaluated n1 is negated and stored at resolved n2.

12.3.3.7 MULTIPLY FLOATING WITH 2 OPERANDS

Opcode:	135
Syntax:	FMUL2 n1, n2 n1: source of type floating point n2: source/destination of type floating point
Description:	Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n2.
Exceptions:	2 - floating point overflow exception 3 - floating point underflow exception

12.3.3.8 MULTIPLY FLOATING

Opcode:	136
Syntax:	FMUL n1, n2, n3 n1: source of type floating point n2: source of type floating point n3: destination of type floating point
Description:	Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n3.
Exceptions:	2 - floating point overflow exception 3 - floating point underflow exception

12.3.3.9 DIVIDE FLOATING

Opcode:	137
Syntax:	FDIV n1, n2, n3
	n1: source of type floating point n2: source of type floating point n3: destination of type floating point
Semantics:	n3 := n2 / n1
Description:	Evaluated n2 is divided by evaluated n1 and the result is stored at resolved n3.
Exceptions:	2 - floating point overflow exception 3 - floating point underflow exception 4 - floating point divide by zero exception

12.3.4 Decimal Arithmetic Instructions

12.3.4.1 CLEAR DECIMAL

Opcode:	138
Syntax:	DCLR n1
	n1: destination of type decimal
Description:	Evaluated n1 is assigned the value of decimal zero.

12.3.4.2 SET DECIMAL TO ONE

Opcode:	139
Syntax:	DSETONE n1
	n1: destination of type decimal

Description: Evaluated n1 is set to decimal one.

12.3.4.3 MOVE DECIMAL

Opcode: 140

Syntax: DMOV n1, n2

n1: source of type decimal
n2: destination of type decimal

Description: Evaluated n1 is stored at resolved n2.

12.3.4.4 NEGATE DECIMAL

Opcode: 141

Syntax: DNEG n1, n2

n1: source of type decimal
n2: destination of type decimal

Description: Evaluated n1 is negated and stored at resolved n2.

12.3.4.5 INCREMENT DECIMAL WITH ONE OPERAND

Opcode: 142

Syntax: DINC1 n1

n1: source/destination of type decimal

Description: Evaluated n1 is incremented by one and the result is stored at resolved n1.

Exceptions: 5 - decimal overflow exception

12.3.4.6 DECREMENT DECIMAL WITH ONE OPERAND

| Opcode: 143

Syntax: DDEC1 n1

n1: source/destination of type decimal

Description: Evaluated n1 is decremented by one and the result is stored at resolved n1.

Exceptions: 5 - decimal overflow exception

12.3.4.7 ADD DECIMAL WITH 2 OPERANDS

| Opcode: 144

Syntax: DADD2 n1, n2

n1: source of type decimal
n2: source/destination of type decimal

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n2.

Exceptions: 5 - decimal overflow exception

12.3.4.8 ADD DECIMAL

| Opcode: 145

Syntax: DADD n1, n2, n3

n1: source of type decimal
n2: source of type decimal
n3: destination of type decimal

Description: Evaluated n1 is added to evaluated n2 and the result is stored at resolved n3.

Exceptions: 5 - decimal overflow exception

12.3.4.9 SUBTRACT DECIMAL WITH 2 OPERANDS

Opcode:	146
Syntax:	DSUB2 n1, n2
	n1: source of type decimal n2: source/destination of type decimal
Description:	Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n2.
Exceptions:	5 - decimal overflow exception

12.3.4.10 SUBTRACT DECIMAL

Opcode:	147
Syntax:	DSUB n1, n2, n3
	n1: source of type decimal n2: source of type decimal n3: destination of type decimal
Description:	Evaluated n1 is subtracted from evaluated n2 and the result is stored at resolved n3.
Exceptions:	5 - decimal overflow exception

12.3.4.11 MULTIPLY DECIMAL WITH 2 OPERANDS

Opcode:	148
Syntax:	DMUL2 n1, n2
	n1: source of type decimal n2: source/destination of type decimal
Description:	Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n2.
Exceptions:	5 - decimal overflow exception

12.3.4.12 MULTIPLY DECIMAL

| Opcode: 149

Syntax: DMUL n1, n2, n3

n1: source of type decimal
n2: source of type decimal
n3: destination of type decimal

Description: Evaluated n2 is multiplied by evaluated n1 and the result is stored at resolved n3.

Exceptions: 5 - decimal overflow exception

12.3.4.13 DIVIDE DECIMAL

| Opcode: 150

Syntax: DDIV n1, n2, n3

n1: source of type decimal
n2: source of type decimal
n3: destination of type decimal

Semantics: n3 := n2 / n1

Description: Evaluated n2 is divided by evaluated n1 and the result is stored at resolved n3. Resolved n3 has an assumed decimal point which is given by the length of n2. The number of integral digits stored in the destination is equal to the number of digits in resolved n2. The fractional part of the result is right truncated and stored in the remaining portion of the destination. Results are unpredictable if resolved n2 is longer then the destination.

Exceptions: 6 - decimal divide by zero exception

12.3.4.14 SCALE DECIMAL

Opcode:	151
Syntax:	DSCALE lit, n1, n2 lit: literal, signed scale factor n1: source of type decimal n2: destination of type decimal
Semantics:	n2 := n1 * 10 ** lit
Description:	Evaluated n1 is multiplied by 10 to the power of the integer literal and the result is stored at resolved n2.
Exceptions:	5 - decimal overflow exception

12.3.4.15 SCALE DECIMAL WITH ROUNDING

Opcode:	152
Syntax:	DRND lit, n1, n2 lit: literal, signed scale factor n1: source of type decimal n2: destination of type decimal
Description:	Evaluated n1 is multiplied by 10 to the power of the integer literal, lit, and the result is stored at resolved n2. Rounding occurs (the result is incremented by one) if the scale factor is negative and the leftmost truncated decimal digit is five or larger.
Exceptions:	5 - decimal overflow exception

12.3.5 String Instructions

The following instructions support the COBOL STRING, UNSTRING and INSPECT statements. The data named by <start>, <end>, and <index> are binary integers which "index" either the source or the destination. The value one designates the first character position, two the second, etc. (i.e. the strings are entity

indexed). These instructions give unpredictable results if:

1) start > end
2) start < 1
3) "end" indexes past the last character position
4) any source string is null

12.3.5.1 MOVE FROM SUBSTRING

Opcode:	153
Syntax:	CMOVFMSUB n1, n2, n3, n4
	n1: source of type character string n2: source of type integer, start position n3: source of type integer, end position n4: destination of type character string
Description:	The characters in source string n1 from start position through end position are moved to the destination string n4. If the number of characters to be moved exceeds the destination string size then right truncation takes place. The delimited source substring is ASCII space filled to the length of the destination string.

12.3.5.2 MOVE TO SUBSTRING

Opcode:	154
Syntax:	CMOVTOSUB n1, n2, n3, n4
	n1: source of type character string n2: destination of type character string n3: source of type integer, start position n4: source of type integer, end position
Description:	All the characters in source string n1 are moved to the destination string n2 starting at "start" position and ending at "end" position. If the number of characters to be moved exceeds the space in the delimited destination, right truncation takes place.

The source string is ASCII space filled to the length of the delimited destination string.

12.3.5.3 MOVE SUBSTRING

Opcode: 155

Syntax: CMOVSUB n1, n2, n3, n4, n5, n6

n1: source character string
n2: source string start position, integer
n3: source string end position, integer
n4: destination character string
n5: start position in destinition string, integer
n6: end position in destinition string, integer

Description: Characters in source string designated by n1 from start position through end position are copied to that portion of the destination string designated by n4 from destination start position through destination end position. If the number of characters to be moved exceeds the space in the delimited destination, right truncation takes place. The delimited source substring is ASCII space filled to the length of the delimited destination string.

12.3.5.4 SCAN SUBSTRING

Opcode: 156

Syntax: SCAN n1, n2, n3, n4, n5, lit

n1: source of type character string
n2: source of type integer, start position
n3: source of type integer, end position
n4: source of type character string, delimiter
n5: destination of type signed integer, index

lit: inline literal

Description: The characters in "source" n1 from "start" through "end" are scanned for character(s) in "delimiter". If the delimiter is not found, "index" is unchanged and a relative branch is taken by updating the offset portion of current pc by the value of the inline literal. If a delimiter is found, "index" is set to the character position of the start of the delimiter and execution continues inline.

Notes: If the delimiter contains more than one character, all the characters must occur in sequence in "source" n1.

12.3.5.5 MOVE CHARACTERS

| Opcode: 157

Syntax: CMOV n1, n2

n1: source of type character string
n2: destination of type character string

Description: Evaluated n1 is stored at resolved n2. If the length of n1 is greater than that of n2, the move is right truncated. If the length of n1 is less than that of n2, the move is ASCII spaces filled at the right.

12.3.5.6 MOVE SPACES

| Opcode: 158

Syntax: CMOVS n1

n1: destination of type character string

Description: Each byte of evaluated n1 is assigned the value ASCII space.

12.3.5.7 MOVE CHARACTERS TRANSLATED

| Opcode: 159

Syntax: CTMOV n1, n2, n3

n1: source of type character string
n2: source of type translate table
n3: destination of type character string

Description: Each character of evaluated n1 is translated and stored into resolved n3. If the length of n1 is greater than that of n3, the move is right truncated. If the length of n1 is less than that of n3, the move is filled at the right with translated ASCII spaces.

12.3.5.8 EDIT CHARACTERS

| Opcode: 160

Syntax: EDITC n1, n2, n3

n1: source of type character string
n2: source of type character string
n3: destination of type character string

Description: Source string n1 is edited into the destination under control of the mask n2. The mask is processed a byte at a time. The edit terminates when the end of the mask or destination is reached. The interpretation of each mask byte (mb) is:

mb = 0; ignored
0 < mb < 16; copy mb characters from source to destination
mb = 16; ignored
16 < mb < 32; skip over mb-16 source characters
All others: The mask character is copied literally into the destination.

Note: If the destination is too short, the result is right truncated.

12.3.5.9 EDIT NUMERIC

Opcode: 161

Syntax: EDITNUM n1, n2, n3

n1: source of type decimal
n2: source of type character string
n3: destination of type character string

Description: The source is a decimal number which is edited into the destination according to the mask n2. The first byte of the mask controls zero suppression, sign type, sign placement and currency symbol placement. The rest of the mask contains literal characters and control characters for building the destination string.

Pass 1

The source is fetched starting with the most significant digit and ASCII characters are stored in the destination under control of the mask. Source characters in excess of those required by the mask are ignored. A mask which requires more digits than are in the source yields unpredictable results.

Character code 0 in the mask causes the next source digit to be passed into the destination as an ASCII character.

Character code 1 in the mask is a termination symbol for both floating insertion and zero suppression. It has no effect on the destination in pass 1.

Character code 2 skips the next character in the destination with no modification.

Character code 3 is reserved for future use.

All other characters in the mask are passed literally into the destination.

Data General Corporation
Company Confidential

Pass 2

The first two bits of the first byte of the mask contain the command controlling leading zero suppression. Code 0 causes no zero suppression. Code 1 is reserved for future use and is ignored. Code 2 selects space as the fill character and code 3 selects "*" as the fill character. If one of the forms of zero suppression is selected, all characters to the left of the leading nonzero digit and to the left of the termination symbol are replaced with the fill character.

Pass 3

The 3rd pass inserts sign and the currency symbol. There are 3 sets of commands for pass 3.

Bits 2 and 3 of the first byte of the mask control the sign position. Code 0 omits any sign. Code 1 causes the sign to be placed in the first character of the destination (leading sign). Code 2 causes the sign to be placed in the last one or two characters of the destination depending on sign type selected. Code 3 causes the sign to be placed in the character before the leading nonzero digit and before the termination symbol (floating sign).

Bits 4 and 5 of the first byte of the mask control the type of sign to be used. Code 0 inserts "-" if the source is negative and "+" otherwise. Code 1 inserts "-" if the source is negative and a space otherwise. Codes 2 and 3 are valid only for trailing sign position. Code 2 inserts "CR" if the source is negative and two spaces otherwise. Code 3 inserts "DB" if the source is negative and two spaces otherwise.

Bits 6 and 7 of the first byte of the mask control the placement of the currency symbol. Code zero causes no currency symbol to be used. Code 1 causes the currency symbol to be placed in the first character of the

destination. Code 2 causes the currency symbol to be placed in the second character of the destintion. Code 3 causes the currency symbol to replace the character before the leading nonzero digit or before the termination symbol, whichever is leftmost (floating currency).

Illegal Combinations

The following combinations are invalid. Their use yields unpredictable results.

* Leading sign with "CR" or "DB".
* Floating sign with "CR" or "DB".
* Leading sign and currency as first character.
* Zero suppression and no termination symbol.
* Termination symbol in mask and no zero suppression.
* Floating character and no zero suppression.
* Two or more termination symbols.
* Floating insertion into the same position as a fixed insertion.
* Two floating characters.
* Floating insertion and no position available in front of the first non-zero digit.

12.3.6 Miscellaneous Instructions

12.3.6.1 INITIALIZE

| Opcode: 162

Syntax: INIT lit1,lit2,n1

lit1: literal
lit2: literal
n1: decimal overflow name of type integer

Description: The low order eight bits of literal lit1 are designated the currency character. n1 becomes the overflow name used by the decimal arithmetic instructions. lit2 is ignored.

12.3.6.2 CLEAR

| Opcode: 163

Syntax: CLR n1

n1: destination of type integer or floating point

Description: Each bit of evaluated n1 is set to zero.

12.3.6.3 CONVERT INTEGER TO FLOATING

| Opcode: 164

Syntax: IFCVT n1, n2

n1: source of type integer
n2: destination of type floating point

Description: Evaluated n1 is converted to a floating point number and the result is stored at resolved n2.

Notes: The converted result will be right truncated if there are more significant bits in integer n1 than the mantissa of the floating point allows.

12.3.6.4 CONVERT FLOATING TO INTEGER

| Opcode: 165

Syntax: FICVT n1, n2

n1: source of type floating point
n2: destination of type integer

Description: Evaluated n1 is converted to an integer value and the result is stored at resolved n2. The fractional part of evaluated n1 is discarded.

Exceptions: 0 - integer range exception

12.3.6.5 CONVERT INTEGER TO DECIMAL

| Opcode: 166

Syntax: IDCVT n1, n2

n1: source of type integer
n2: destination of type decimal

Description: Evaluated n1 is converted to a decimal number and the result is stored at resolved n2.

Exceptions: 5 - decimal overflow exception

12.3.6.6 CONVERT DECIMAL TO INTEGER

| Opcode: 167

Syntax: DICVT n1, n2

n1: source of type decimal
n2: destination of type integer

Description: Evaluated n1 is converted to an integer value and the result is stored at resolved n2.

Exceptions: 0 - integer range exception

12.3.6.7 CONVERT FLOATING TO DECIMAL

| Opcode: 168

Syntax: FDCVT n1, n2

n1: source of type floating point
n2: destination of type decimal

Description: Evaluated n1 is converted to a decimal value and the result is stored at resolved n2.

Exceptions: 5 - decimal overflow exception

Notes: The conversion is done with truncation of fraction.

12.3.6.8 CONVERT DECIMAL TO FLOATING

| Opcode: 169

Syntax: DFCVT n1, n2

n1: source of type decimal
n2: destination of type floating point

Description: Evaluated n1 is converted to a floating point value and the result is stored at resolved n2.

Notes: The conversion is done with right truncation if there are not enough mantissa digits to represent the magnitude of the decimal number.

12.3.7 Size Error Instructions

12.3.7.1 CLEAR SIZE ERROR

| Opcode: 170

Syntax: CLRSZ

Description: This operation sets SIZE_ERROR to zero.

12.3.7.2 SET SIZE ERROR

| Opcode: 171

Syntax: SETSZ

Description: This operation sets SIZE_ERROR to one.

12.3.7.3 CHECK MOVE INTEGER

| Opcode: 172

Syntax: ICKMOV n1, n2, n3

n1: source of type integer
n2: source of type integer, limit
n3: destination of type integer

Description: If the source fits in the destination, the the source is assigned to the destination as in the IMOV instruction. If the source does not fit in the destination SIZE_ERROR is set to one. The source fits in the destination if the magnitude of source does not exceed limit (see note).

Notes: The integer limit is often equal to the value of decimal numbers of a number of 9's, to check if the number n1 will be in range when translated to a decimal number. Left truncation will occur if "limit" exceeds the range of the destination.

17:18:7
11/Oct/79
Rev. 1

12.3.7.4 CHECK MOVE DECIMAL

Opcode:	173
Syntax:	DCKMOV n1, n2 n1: source of type decimal n2: destination of type decimal
Description:	If the source fits in the destination, then the source is assigned to the destination. If the source does not fit in the destination then SIZE_ERROR is set to one. Note that the condition "fits in the destination" may depend on the number of significant decimal digits in the source, and not just its length!

12.3.7.5 BRANCH IF NO SIZE ERROR

Opcode:	174
Syntax:	BNSE lit lit: inline literal
Description:	If SIZE_ERROR is 0, a relative branch is taken by updating the offset portion of current pc, using the inline literal. Otherwise, execution continues inline.

12.3.8 Architectural Instructions

12.3.8.1 CALL

Opcode:	2
Syntax:	CALL n1, lit, [n3, n4, ...] n1: source of type pointer

lit: inline literal
[n3, n4,...] : source of any type

Description: This instruction implements Call. Resolved n1 is the target of the call. lit is the number of parameters of the call. The remaining operands are the parameters of the call.

Exceptions: See Chapter 8.

12.3.8.2 RETURN

| Opcode: 3

Syntax: RTN

Description: This instruction implements Return.

Exceptions: See Chapter 8.

12.3.8.3 NOP

| Opcode: 1

Syntax: NOP

Description: This instruction performs no operation.

--End of Chapter--

Index

activation 8-5
activation record 8-6

call 8-1
Call and Return 8-1

nonlocal goto 8-5

procedure reenter trace 8-5,
8-6
procedure return trace 8-5,
8-6

return 8-1, 8-5