AD-A158 120	ADVANCED INSTRUCT	AVIONICS CO ION SET ARCH H L GREENSP	DMPUTER ARCI	HITECTURE VOL SANDERS ASSO 19 85	LUME 2 CIATES INC	1/4
	AFWAL-TR	-85-1041-VOL	-2 F33615-7	^{79-C-1935}	F/G 9/2	
PREVIOUS PARAMAN IN	ine o entre	AND COMMENT	ALE SELECTION			NESSIN ACCOUNT
Bernard allerer Bernard allere		10000000000000000000000000000000000000		All Constants		
	BUNGENCE STRATEGICS MEMOURA ME					
	nitäken Nitäälä Erivent		The second secon		Entransistante Alternationale Altern	
		Manager and American America American American A		MARCHARD	Balance and a second se	
						6



AD-A158 120

AFWAL-TR-85-1041

ADVANCED AVIONICS COMPUTER ARCHITECTURE

VOLUME II - INSTRUCTION SET ARCHITECTURE SPECIFICATION

LAWRENCE GREENSPAN RONALD SINGLETARY

SANDERS ASSOCIATES, INC. 95 CANAL STREET NASHUA, NEW HAMPSHIRE 03061-2034

MAY 1985

FINAL REPORT FOR PERIOD MAY 1980 - NOVEMBER 1984

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

OTIC FILE COPY

AVIONICS LABORATORY AIR FORCE WRIGHT AERONAUTICAL LABORATORIES AIR FORCE SYSTEMS COMMAND WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433



85

2

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign mations.

This technical report has been reviewed and is approved for publication.

Vince

GUY A. **VI**NCE Project Engineer, Information Processing Technology Branch Avionics Laboratory

FOR THE COMMANDER

RAYMOND D. BELLEM, COL, USAF Deputy Chief System Avienics Division

JERRY L. COVERT, Acting Chief Information Processing Technology Branch Avionics Laboratory

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT;? W-PAFB, OH 45433 to help us maintain a current mailing list".

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

	REPORT DOCUME	ENTATION PAGE	E			
1a REPORT SECURITY CLASSIFICATION		16. RESTRICTIVE MARKINGS				
UNCLASSIFIED						
28 SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT				
N/A		Approved for	r public re	lease; dist:	ribution	
2b. DECLASSIFICATION/DOWNGRADING SCHET	DOLE .	unlimited.				
4. PERFORMING ORGANIZATION REPORT NUM	BER(S)	5. MONITORING OR	GANIZATION R	PORT NUMBER(S	3)	
			10/1 11-1			
6. NAME OF PERFORMING ORGANIZATION	55. OFFICE SYMBOL	7. NAME OF MONIT	ORING ORGAN			
	(If applicable)	Avionics La	aboratory (AFWAL/AAAT)		
SANDERS	l	AF Wright A	Aeronautica	1 Laborator:	ies (AFSC)	
6c. ADDRESS (City, State and ZIP Code)		7b. ADDRESS (City,	State and ZIP Cod	le)		
95 Canal Street						
Nashua NH 03061		LIDATE ON 45	433-6543			
BA NAME OF FUNDING/SPONSORING	80. OFFICE SYMBOL	9 PROCUBEMENT I	NSTRUMENT ID	ENTIFICATION N	UMBER	
ORGANIZATION	(If applicable)					
Avionics Laboratory	AFWAL/AAAT	F33615-79-0	-1935			
8c. ADDRESS (City, State and ZIP Code)		10. SOURCE OF FUR	DING NOS.			
		PROGRAM	PROJECT	TASK	WORK UNIT	
LIDAER OIL (5422 6542	· · ·	ELEMENT NO.	NO.	NO.	NU.	
WPAFB UN 43433-0343						
Advanced Avionics Computer Arch	*See Back	62204F	2003	04	19	
12. PERSONAL AUTHOR(S)			<u></u>	L		
Lawrence Greenspan, Ronald S:	Ingletary					
134. TYPE OF REPORT 13b. TIME (OVERED	14. DATE OF REPOR	AT (Yr., Mo., Dey,	15. PAGE C	OUNT	
FINAL FROM 5	<u>/80</u>	1985 May		305		
16. SUPPLEMENTARY NOTATION				and a state of the		
	THE SUBJECT THENE IC					
EVELD GROUP SUB GB	High Toyol Ja	on tinue on reverse if he	the second second	I'Y OY DIOCK NUMDE	"	
69 02	Semantic Rap	Reduction.				
	Language-Dire	cted Architect	ure . Toy.	en)		
19. ABSTRACT (Continue on reverse if necessary an	d identify by block number	r)	70			
This exploratory development	program was orig	inally aimed a	t developi	ng a compute	er with	
features to specifically supp	ort the JOVIAL (J73) programmi	ing languag	e with cons:	iderations	
to Ada. Later, the program wa	as redirected to	modify the in	nstruction	set archited	cture (ISA)	
to more fully support Ada and	increase perfor	mance.				
	Married and south south and and and	- and it was not a second and				
The new ISA supports most of	the standard fun	ctions found i	In most ISA	, but gives	addition-	
al supports in: the Ada pack	age concept, pro	cessing arrays	s and recor	ds, unconsti	rained	
arrays, dynamic storage alloca	ation, detecting	dangling refe	erences, de	tecting und	erined	
Variables, Ada-like exception	nandling, case	instructions,	IOT-LOOP 1	nstructions,	Ada like	
type, Keyward -	asking instructi	ons and IEEE-8	scanuaru 11	oating point	L UALA	
cipes. negwords:	•					
	-	 Be Block search and Base search pro- 	and a set of the set o			
20. DISTRIBUTION/AVAILABILITY OF ABSTRA	CT	21. ABSTRACT SEC	URITY CLASSIFI	CATION		
UNCLASSIFIED/UNLIMITED LO SAME AS RPT.		UNCLASSIFI	ED			
224. NAME OF RESPONSIBLE INDIVIDUAL		226. TELEPHONE N	UMBER	22c. OFFICE SYM	BOL	
Guy Vince		57706	140)	AFWAL /AAA'	r-2	
				1		

UNCLASSIFIED SECURITY CLASSIFICATION OF THIS PAGE 11. Title (Cont'd) Volume II Instruction Set Architecture Specification 18. Non-Von Neumann Architecture; Object Oriented Architecture, Capability Based Addressing. UNCLASSIFIED SECURITY CLASSIFICATION OF THIS PAGE

PREFACE

The contents of the document are technically accurate, and no sensitive items, detrimental ideas, or deleterious information are contained therein. Furthermore, the views expressed in the document are those of the author(s) and do not necessarily reflect the views of the Avionics Laboratory, the Air Force Systems Command, the United States Air Force, or the Department of Defense.

Accession For NTIS GRA&I DITC TAB Unannounced Justification By___ Distribution/ Availability Codes Lette and or Lako C. Dist

TABLE OF CONTENTS

	Page
SECTION 1	- INTRODUCTION
1.1	ISA Summary
	and a second
SECTION 2	- STORAGE OBJECTS
2.1	Package Object (PO)
2.2	Activation Record (AR)
2 3	Task Object (TO) 2-2
2.5	Data Object (DO) 2-3
2.4	Implementation 2-3
2.5	
2.5.1	Loading of Packages
2.5.2	Data Templates
CECETON 2	
SECTION 3	= DAIA FORMAIS
3.1	16-Dit value Data (V16)
3.2	32-Dit value Data (V32)
3.3	64-bit Value Data (V64)
3.4	96-bit Pointer (PTR)
3.5	96-bit Formal Reference Parameter (FRP)
3.6	Variable Size Record (REC)
3.7	Variable Size Array Header
3.7.1	Lower Bound and Upper Bound (LB/UB, LB, UB)
3.7.2	Multi-dimension Span (SPAN)
3.7.3	Separate Array Value Offset (AVO)
3.7.4	Dynamic Array Value Address (AVA)
3 7 5	Total Record Size (TRS) 3-16
3 8	Data Object Descriptor (DOD) 3-17
201	Constrained DOD
3.0.1	Constrained DOD
3.0.2	Unconstrained DOD
SECTION 4	- INSTRUCTION FORMATS. 4-1
A 1	Operation Code (OPCODE)
4 2	Operand Formate (FMT)
4.2	
4.2.1	
4.2.2	Formats
4.2.3	Register
4.2.4	Immediate
4.2.5	Stack
4.3	New Operand Specifier (NOS)4-9
4.4	Operand Qualifiers4-9
4.4.1	Bit Position (BPOS)4-9

1 1 2	Pacard Component Offset (PCO) 4-10
4.4.2	
4.4.3	Array Subscript (SOB)
4.4.4	Array Slice Index (SLICE)4-11
. 4.4.5	Index Constraint (IDXCON)4-12
4.4.6	Base Relative Offset (BRO)4-13
4 4 7	Array Size (ASIZ) 4-13
3.3./	
CRCETON E	
SECTION 5	
5.1	Data Movement
5.1.1	Move (MOV)
5.1.2	Move Array (MOVARR)
5.1.3	Move Array Slice (MOVSL)
514	Load Array Base Address (LDBA) 5-8
5 1 5	Move Pointer (MOVPTP) 5-9
5.1.5	MOVE FOILDET (MOVETR) = 10
5.1.6	Set Underine (SETUND)
5.1.7	Purge Stack (PURGE)
5.1.8	Swap Stack (SWAP)
5.1.9	Clear Temporaries Mask (CLRMSK)
5.2	Arithmetic
5 2 1	Add Integer (ADDI2 ADDI3) 5-16
5.2.1	Add Integer (ADD12/ADD13/
5.2.2	Add Floating Fornt (Abbr 2, Abbr 3)
5.2.3	Subtract Integer (SUBI2, SUBI3)
5.2.4	Subtract Floating Point (SUBF2, SUBF3)
5.2.5	Multiply Integer (MULI2, MULI3)
5.2.6	Multiply Floating Point (MULF2, MULF3)
5 2 7	Divide Integer (DIVI2 DIVI3) 5-28
5 2 9	Divide Floating Point (DIVE2 DIVE2) 5-20
5.2.0	$\frac{1}{2} \frac{1}{2} \frac{1}$
5.2.9	Remainder Integer (REMI2, REMI3)
5.2.10	Remainder Floating Point (REMF2, REMF3)
5.2.11	Modulus Integer (MODI2, MODI3)5-36
5.2.12	Modulus Floating Point (MODF2, MODF3)
5.2.13	Negate Integer (NEGI1, NEGI2)
5 2 14	Negate Floating Point (NEGFL NEGF2) 5-42
5 2 15	Absolute Integer (ABSI) ABSI2) 5-44
5.2.15	Absolute Integer (Absil, Absil, Absil, Absil, Absolute Planting Daipt (Apsil, Absil, Absil, Absolute Planting Daipt (Apsil, Absil, Absolute Planting, Baipt (Apsil, Absolute Absolute Planting, Baipt (Apsil, Absolute Abs
5.2.10	Absolute Floating Point (Absri, Absr2)
5.2.17	Square Root Integer (SQRTI1, SQRTI2)5-48
5.2.18	Square Root Floating Point (SQRTF1, SQRTF2)5-50
5.2.19	Round to Nearest (RNDN)
5.2.20	Round to Zero (RNDZ)
5.2.21	Round to Plus Infinity (RNDP)
5 2 22	Round to Minus Infinity (PNDM) 5-55
5 2 22	Convert Integer to Floating Point (CONVIE) 5-56
5.2.23	Convert Integer to Floating Foint (CONVIF)
5.2.24	convert rloating point to integer (CONVFI)
5.3	Log1ca1
5.3.1	And (AND2, AND3)5-59
5.3.2	And Array (ANDA2, ANDA3)
5.3.3	And Slice (ANDS2, ANDS3)
5.5.5	

ę

vi

5 3 4	0~ (0P2 0P3) 5-65
5.5.4	O(0, Z) = O(0, Z)
5.3.5	Or Array (ORA2, ORA3)
5.3.6	Or Slice (ORS2, ORS3)
537	Exclusive Or (EXOR2, EXOR3), 5-71
5 2 0	Evelucine Or Laner (Even) Trons?
5.3.8	Exclusive of Array (EXORA2, EXORAS)
5.3.9	Exclusive Or Slice (EXORS2, EXORS3)
5.3.10	Equivalence $(EO2, EO3)$.
5 2 11	= -70
5.3.11	Equivalence Array (EQA2, EQA3)
5.3.12	Equivalence Slice (EQS2, EQS3)
5.3.13	Not (NOT), NOT2)
E 2 14	
2.2.14	NOL AITAY (NOTAL, NOTAZ)
5.3.15	Not Slice (NOTS1, NOTS2)
5.3.16	Set (SET)
5 3 17	Set Array (SETA) 5-90
5.3.17	\mathcal{G}
5.3.18	Set Slice (SETS)
5.3.19	Clear (CLR)
5 3 20	Clear Array (CLRA) 5-93
5.3.20	
5.3.21	Clear Slice (CLRS)
5.4	Branch
5.4.1	If (IF)
5 4 2	If F_{r} (IF-) F_{r}
5.4.2	11 Equal (1F-)
5.4.3	If Not Equal (IF $< >$)
5.4.4	If Less than Integer (IFI<)
545	If Less than Floating Point (IFF<) 5-101
5.4.0	If dead the theory to the state of the state
5.4.0	II Greater than integer (IFI>)
5.4.7	If Greater than Floating Point (IFF>)
5.4.8	If Greater than or Equal to Integer (IFI>=)
E 4 0	If Greater than or Equal to Floating Daint (IPPL-) 5 105
3.4.3	II Greater than or Equal to Floating Point (IFF>=)5-105
5.4.10	If Less than or Equal to Integer (IFI<=)
5.4.11	If Less than or Equal to Floating Point (IFF<=)
5 4 12	If Defined (IFD) 5-108
5.4.12	
5.4.13	If in Range Integer (IFIRNG)
5.4.14	If in Range Floating Point (IFFRNG)
5.4.15	GO TO (GOTO)
5 4 16	
5.4.10	Case (CASE)
5.4.17	Set Loop Control Variable (SETLCV)
5.4.18	LOOP Up (LOOPUP)
5 4 19	LOOD DOWN (LOOPDN) 5-115
J. 1. 1. J	
and the second se	
SECTION 6	- SUBPROGRAMS
6.1	Call Subprogram (CALL)
6.2	Parameter Association 6-7
6.2	
6.2.1	Passing via Register File
6.2.2	Passing via Memory Transfer
6.2.3	Load RO Reference Parameter (LDRO)
6 2 4	Load 50 Perference Parameter (LDCO)
0.2.4	Load wo Reference Fardmeter (LDWO)
6.2.5	Load RW Reference Parameter (LDRW)
6.2.6	Clear Valid Parameter Mask (CLRVPM)
627	Bind Parameters (BIND)
6.2.8	Return from Subprogram (RETSUB)6-18

TE CREAK SEL A.

SECTION 7 7.1 7.2 7.3	- PACKAGES
7.4	Allocate Package Storage (ALLOCP)
SECTION 8	- DYNAMIC STORAGE ALLOCATION/DEALLOCATION
8.1	Create Data Object (CRDO)
8.2	Create Unchecked Data Object (CRUNDO)
8.3	Destroy Data Object (DSTROY)
SECTION 9	- TASKS
9.1	Task Scheduling
9.2	Task Switching
9.3	Exception Modes
9.4	Tasking Instructions
9.4.1	Create Task Object (CRTO)
9.4.2	Activate Task (ACTV)
9.4.3	End Elaboration (NELAB)
9.4.4	End Activation (NACTV)
9.4.5	End Elaboration & Activation (NELACT)
9.4.6	Evaluate Allocated Task Object (EVALTO)
9.4.7	Call Entry (CALEN)
9.4.8	Call Entry Conditionally (CALENC)
9.4.9	Call Entry with Time Out (CALENT)
9.4.10	Accept Entry (ACCEPT)
9.4.11	End Rendezvous (ENDRNV)
9.4.12	Delay (DELAY)
9.4.13	Select Accept (SACCPT)
9.4.14	Wait (WAIT)
9.4.15	Select Delay (SDELAY)
9.4.16	Select Else (SELSE)
9.4.17	Select Terminate (STERM)
9.4.18	Return from Task (RETTSK)
9.4.19	Schedule Task (SCHDL)
9.4.20	Set Task Duration (SETDUR)
9.4.21	Abort Task (ABORT)9-46
SECTION 10	- POINTERS
10.1	Assign Pointer to Global Data (ASNPGD)
10.2	Assign Pointer to External VGD (ASNPXV)
10.3	Assign Pointer to External CGD (ASNPXC)
10.4	Assign Pointer to External Program (ASNPXP)
10.5	Restrict Access Rights (RSTRCT)10-11
SECTION 11	- FYCEPTIONS
SECTION II	Doice (DAIGE) 11.7
11 2	Accert Range Integer (ASRTRI) 11_9

......

.

A. Examples of ArraysB. Task Dependencies

A-1 B-1

1 INTRODUCTION

This document describes the Instruction Set Architecture (ISA) of a computer known as the High Level Language Machine (HLLM) that has special features to support the programming languages, Ada and JOVIAL. The HLLM is intended for embedded applications. The objectives of this ISA are to provide high performance support for the frequently used "low level" features of the HOLS (integer and floating point arithmetic, logical and relational operations, looping, processing of arrays, calling subprograms) while reducing the run time software overhead associated with the advanced features of the HOLS (Ada packages, tasking, dynamic storage allocation for unconstrained arrays and evaluation of allocators, exception handling, etc.). Low level support is accomplished by including powerful and versatile addressing modes with multiple operand instructions in which the operands can be in memory, registers, or on an expression stack. The register file can also be used as a medium for passing parameters during subprogram or task entry calls. Support for the advanced features is embodied in powerful instructions in which microcode and hardware will replace the software routines otherwise required.

An attempt has been made to write the ISA in sufficient detail to satisfy both the compiler designer and the system implementor. Each section has an introduction followed by a detailed functional description; in many cases, the relationship between the feature being described and a language construct is indicated. For instructions, the legal formats and use of each operand is specified; then, a detailed description of the function of the instruction is provided with a list of applicable An attempt was made to logically arrange the exceptions. sections of the ISA. First to be described are the four storage objects into which all data and programs are organized. Next, the various data types and data descriptors are described. Finally, the instruction formats and functional operation of the instructions are described. The instructions are divided into basic instructions, subprograms, packages, dynamic ten groups: storage allocation, tasks, pointers, exceptions, traps, input-Features and examples of the use of output, and attributes. instructions whose descriptions are too lengthy or detailed for inclusion in the sections are relegated to appendixes.

ISA Summary. Memory is logically partitioned into four 1.1 types of storage objects. Package objects, which are non-nested or nested in other packages, support the Ada package construct. Activation record objects, which are allocated whenever a subprogram is called or a task object is created, support recursion and reentrancy. Task objects, together with a task scheduler and twenty specialized instructions, support Ada tasking. Data objects, which are explicitly allocated during run time, support the evaluation of allocators in Ada. (Task, as well as data objects, can be dynamically created.) Data templates for activation records, variable global data, and data objects allow the compiler to assign initial values. (Declared data that is not initialized is marked as undefined and cannot be read until a value is assigned at run time.) There are three distinct physical memories which are simultaneously accessible: instruction memory, data template memory, and data value memory. Packages are loaded into instruction memory and data template memory, which are read-only at run time. A package contains global data and the automatic data of subprograms and tasks as well as the instructions of subprograms and tasks that are contained in the main (outermost) package and in all nested Data value memory is allocated at run time for packages. activation records of subprograms and tasks, data objects, and (See Section 2.) unconstrained arrays.

Data formats exist that accommodate Booleans, characters, 16, 32, and 64-bit mask data, 16 and 32-bit signed integers, 32-bit single precision and 64-bit double precision IEEE floating point numbers, 96-bit pointers, and several array and record Pointers contain access rights which prevent descriptors. illegal access to or modification of data, illegal subprogram or task entry calls, etc. Array and record processing are supported. Array headers that contain descriptors specifying bounds and spans for each dimension permit the machine to automatically check subscripts vs bounds and to compute the address of an array component. The machine also automatically computes the size of unconstrained arrays whose bounds are not known until run time. (See Section 3.)

Instruction formats include memory, stack, register, and immediate addressing. Each activation record has a 16-word deep stack for expression evaluation. Sixteen general purpose registers and sixteen registers dedicated to passing parameters are provided. Compact formats are available which utilize short address offset and immediate value fields so that two or three operands can be specified in a single instruction word. In addition to operations on individual array and record components, block moves and logical operations on whole arrays and slices are supported. Array base addresses can be extracted from array headers and loaded into registers. This allows base plus offset addressing of array components and slices, a convenient addressing mechanism when offsets are known at compile time or can be easily computed at run time. Operand qualifiers provide additional information about operands; they include array subscripts, slice indexes, array size, record component offsets, etc. (See Section 4.)

Basic instructions have one, two, or three operands (destination, source-destination, and source-source-destination, respectively). This category comprises (1) data movement which includes moving whole arrays, slices, and records, (2) arithmetic operations which include IEEE standard floating point instructions and square root, remainder, modulus, rounding, and type conversion instructions, (3) logical operations on scalar Booleans and mask data and on arrays and slices of Booleans and masks, and (4) branch operations that include the full complement of relational instructions, range check instructions, a case instruction, and loop control instructions. (See Section 5.)

Subprogram support includes calling and returning from subprograms with parameter passing by value or reference using memory or registers as the medium; control can be exercised over the rights which a called subprogram has to the actual parameters. (See Section 6.)

Operations on packages include (1) loading a package from an external package representation, (2) creating a package object by allocating space in data value memory for the package variable global data and administrative data and returning a pointer to the package, and (3) elaborating a package object by executing subprogram #0 of the created package. (See Section 7.)

Data objects are allocated space in data value memory at run time. The type definition of a data object is specified in its data template. Storage is reclaimed (data object destroyed) when the storage object in which the data object's access type is declared is destroyed. This storage object is designated in the instruction that creates the data object. Although storage is normally reclaimed in the above manner, data objects can be abnormally destroyed by a DESTROY DATA OBJECT instruction if Ada unchecked storage deallocation was programmed. (Any dangling references resulting from such destruction will be detected by means of a "unique name" if access is attempted.) An important use of data objects is for I/O buffering. (See Section 8.)

A task object comprises a task program, an activation record, and associated attributes such as number of entries, identification of dependent and MASTER storage objects, etc. Instructions are available which create and activate tasks, evaluate task allocators, control task rendezvous, and terminate tasks. A hardware task scheduler and clock manager are provided. Different exception modes allow errors to be handled differently when a task (or subprogram) is being elaborated, a task is being activated, a task is in rendezvous, or a task is running normally. (See Section 9.)

Instructions are provided that assign values to pointers for data entities located in the global storage area of the local package or in the global storage area of an external package; in addition, pointers can be assigned for subprograms in external packages. Pointers contain the physical address of the data entity or the subprogram identification and the access rights to the data entity or subprogram. (Access rights to a subprogram control whether the subprogram can be called.) For security, the only operations permitted on pointers are to assign values to (The machine nulls all pointers when them and move them. packages are loaded.) Access rights can be restricted but never Linking of separately compiled packages expanded. is accomplished by moving a pointer to a data entity or a subprogram located in one package into the variable global data area of another package. (See Section 10.)

The Ada exception handling mechanism is fully supported. Normally, the local exception handler is entered, if one is present; if not, the machine traces dynamic links (calling chain) until one is found (or the main program or a task object is reached - terminating the search for a handler). Exceptions in tasks cause them to be completed and exceptions in main programs cause them to be abandoned. All predefined exceptions of Ada are detected by hardware and user - defined exceptions are supported with raise and range checking instructions. (See Section 11.)

Several important Ada-defined attributes are supported with instructions that extract and return the attributes. These include array bounds, length, and size, image and value, and the tasking attributes of count (number of tasks queued on an entry) and callable (that returns a Boolean specifying whether a task is callable, i.e., not completed or terminated). (See Section 12.)

Input/output is supported with six instructions that specify the type of operation (READ, WRITE, GET, PUT, SEND CONTROL, and RECEIVE CONTROL), the logical name of the I/O device, and a pointer to the data object or address of the activation record that serves as an input or output buffer. Direct memory access (DMA) data transfer of 32-bit words or transfer of a single 32-bit word takes place between an I/O card and the buffer under control of the I/O card. During input operations, a 4-bit tag is read from the buffer's data template and attached to a 32-bit data word as each word is written into the buffer (in data value memory). During output operations, the tags are stripped before the data (32-bit words) are transferred to the I/O card. I/O cards are responsible for any packing and unpacking of data

values that may be required because of different word sizes in the device and the HLLM. I/O cards provide the interface between the HLLM and a device. They receive control information from the HLLM and send I/O operation status to the HLLM. I/O cards support direct, sequential, and text I/O. When packages are loaded from the User Console, a special interface card is required that converts between the User Console data format and the 36-bit word format of the HLLM (see Section 14).

A trapping mechanism is provided to permit program traces during debugging. Traps can be programmed to occur after every instruction, every successful branch, every unsuccessful branch, every call, every exception, or whenever a special trace trap instruction is executed (see Section 13).

いたいたいになった。 御書 たいたい たいたい たません 御殿 きいたい ひょうけん 日本 見たす はない はん 大手 たんち 御殿 コンド・

2 STORAGE OBJECTS

This ISA defines four types of storage objects: package object (PO), activation record (AR), task object (TO), and data object Instructions exist which explicitly create each type of (DO). storage object. Storage objects contain information of two user-specified types: administrative data and data. Administrative data comprises ancillary information required for the performance of functions called out in the ISA Specification. It is created by the machine when a storage is created and may be updated from time to time by the machine for the lifetime of the storage object (see Appendix A for a complete description of administrative data). User-specified information is acquired directly from the program at the time of a storage object's creation and includes instructions and/or data which are logically related.

2.1 <u>Package Object (PO)</u>. A package object is the central storage for groups of logically related subprograms, data, task programs, and nested packages. The external representation of a package is shown in Figure 2-1. The entire activity of the machine is determined directly or indirectly by the package object.

A package object contains a header, variable global data (VGD), constant global data (CGD), automatic data templates (ADT) for a number of subprograms and task programs, a nested package area, and instructions for subprograms and task programs in the package (including instructions in nested packages). The package header contains a 1 word package descriptor cell (PKG) and a 5 word descriptor cell for each component in the package (subprograms, task programs and nested packages; see Figure 2-2). The first four bits of the package header descriptor cell is the DESCription tag; the following four bits is the extend tag, PKG. The next eight bits specify how many 5-word package component descriptor cells for subprograms, tasks, and nested packages are The remaining 20 bits specify included in this package header. the storage size, in number of words, occupied by the variable global data. Component descriptor cells for subprogram and task components of packages are very similar as seen by the words at offsets of -1..-5 and -6..-10 in Figure 2-2. (The only differences exist in the words at offsets of -2 and -7; a subprogram requires an exception mode subfield and a formal parameter mask while a task program requires priority level and number of entries.) The meaning and use of the subfields shown in Figure 2-2 are described in Sections 6 and 9, on subprograms and tasks. Note that when the external package is loaded into the machine, all offset values in the package header are. converted to absolute addresses. The description of nested

packages also requires a 5-word descriptor cell per nested package. The first word is the same as the package descriptor cell except that the extended tag identifies a nested package (NPKG) cell. Again, all offsets in this descriptor cell are converted to absolute addresses when the external package is loaded (See Section 2.5.1).

The variable global data and constant global data of a package are directly accessible to any subprogram or task program defined in the package header. Up to 2^{20} halfwords may be addressed in VGD and CGD areas. VGD may be read and written to but CGD can only be read. Each subprogram and task program can contain data with initial values preset by the compiler. Tag and initial values of data are stored in the automatic data template for the corresponding subprogram or task program.

2.2 Activation Record (AR). When a subprogram is called or a task object is created, storage is allocated for an activation record and administrative data. The activation record contains an automatic data area, a stack area, and a separate array value area (see Figure 2-3). The automatic data has a one to one correspondence with initial values in the automatic data template of the subprogram/task program (see Section 2.5.2 for details on data templates). The stack area contains a sixteen word stack that may be used to evaluate expressions with two or more terms. provides efficient memory The separate array value area utilization for arrays defined with one initial value for all of its components. The size of these arrays must be known (constrained) at compile time to compute the appropriate offset to the separate values (see Section 3.7.3). Unconstrained arrays with separate values are also handled by the machine but are not permitted in activation records (see Section 3.8.2).

Task Object (TO). Ada defines tasks as entities whose 2.3 execution can proceed in parallel, independently, except at points where they synchronize (rendezvous). Some tasks have entries which permit rendezvous with other tasks which issue entry calls. A task accepts a call of one of its entries by executing an accept instruction for the entry. Some calls have provide a controlled parameters which environment for communicating values between tasks. The actions performed when . an entry of a task is called are similar to those performed when a subprogram is called except that when the called task reaches a return instruction, both the calling task and the task containing the called entry will resume execution in parallel (see Section 9). When a task object is created, storage is allocated for an activation record (automatic data area, stack area, and separate array value area) and administrative data. Each task can be considered to be executed by a logical processor of its own.

Parallel tasks on the HLLM are implemented with interleaved execution on a single physical processor. Each task object is assigned a priority level by the compiler which determines the relative percentage of CPU time allocated to it.

2.4 Data Object (DO). Data objects, which are explicitly allocated during run time, support the evaluation of allocators in Ada. In addition, data objects (as well as activation records) are used as I/O buffer storage in the HLLM. Each data object includes an administrative data area, a description of the data object (which is usually an array or record or a composite of them), and space for data values (see Figure 2-4). Arrays and arrays of records, whose sizes are dynamically specified at run time, can only appear in data object descriptions. Data object descriptions can only appear in the constant global data area of packages.

2.5 <u>Implementation</u>. The HLLM memory system is divided into three distinct sections: instruction memory (IM), data template memory (DTM), and data value memory (DVM). These storage sections can be simultaneously accessed, a feature required to support the HLLM's pipeline architecture. Package objects are loaded into IM and DTM as discussed in Section 2.5.1. Space in DVM is allocated at run time for activation records, data objects, and unconstrained arrays. Since an activation record is allocated each time a subprogram is called or a task object is created, all subprograms and task programs are recursive and reentrant. The implementation scheme for controlling access to data in DTM and DVM is discussed in Section 2.5.2.

Loading of Packages. The external representations of 2.5.1 packages are loaded into the HLLM via a user console interface A bootstrap loader on the user console interface card loads a package (called the loader-linker) that contains card. first programs that will subsequently load and link other related packages. Headers of packages and all data are loaded into the data template memory while all instructions in the packages are loaded in the instruction memory. Offset fields in headers of non-nested (library) packages are converted to absolute addresses by the bootstrap loader prior to loading the headers. When the loader-linker package has been loaded, the bootstrap loader, via commands from the User Console, allocates storage in data value memory for the package's administrative data and variable global data and then elaborates the packge by invoking its subprogram 0. The loader-linker, in turn, initiates the loading of other packages and then creates, elaborates, and links them (see Section 7).

2.5.2 Data Templates. A data template corresponds to a declarative part in Ada. Data templates are used for the automatic data of subprograms and task programs, the variable global data of packages, and for data objects. Their use allows the compiler to assign initial values to any or all data entities, including arrays. (Any declared data that is not assigned an initial value is marked as undefined by the compiler.) Data templates are loaded into data template memory (DTM) as part of a package object. Templates contain read-only data comprising tags and initial values of data and descriptors of arrays and records.

When memory space is allocated in DVM at run time for an activation record, variable global data, or a data object, the size of the allocation is equal to that of the corresponding data template plus space for administrative data. (Certain array space which is allocated in DVM does not appear in the template, as described in Section 3.7). Initial values of data entities When a value is first assigned to a data are read from DTM. entity, it is written with its tag into DVM. Since a template is never disturbed, it can be reused, e.g., the template of an activation record can be used for any number of subprogram invocations. An initial value in DTM and the corresponding variable value in DVM are located at the same offset from different base addresses. A special control bit called the "residency" bit selects DTM or DVM, depending on whether the addressed data entity contains an initial value or a modified value ("0" selects DTM, "1" selects DVM). A residency bit corresponds to each word in DVM. Hence, its address is the same as that of the data word in DVM. As part initialization, all residency bits are cleared. As part of power-up Thereafter, whenever a storage object is deallocated, the memory manager clears all residency bits for the deallocated block before attaching the storage to the free list. When a data entity is first assigned a value, the following steps occur:

- Residency bit is read and found to be "0" (data in DTM).
- tag and initial value are read from template (in DTM) at address = base of template + OFFSET.
- tag and new (computed) value are written into DVM at address = base of activation record (or variable global data or data object) + same OFFSET.
- residency bit is set to "1" so that data is henceforth referenced in DVM.

When a value is first assigned to a data entity that occupies two or three words (64-bit mask data, double precision floating point number, or a pointer), the residency bit for each of the words is set to 1. Residency bits for words in DVM that correspond to descriptors always remain "0". (The single exception to this occurs when the descriptor is for an unconstrained array - see Appendix B.) Residency bits corresponding to administrative data of storage objects start out as "0s" and are set to "1s" as the administrative information is written. Administrative data has no initial values and, therefore, no data template.

Residency bits may be stored in a fast 2-port RAM organized as 1bit x N words, where N is the number of words in DVM. Two ports are required to allow the memory manager to clear the residency bits of a deallocated block of DVM while, simultaneously, other residency bits are being accessed during normal memory reference operations.

PACKAGE HEADER	
VARIABLE GLOBAL DATA (VGD)	
CONSTANT GLOBAL DATA (CGD)	
AUTOMATIC DATA TEMPLATES (ADT) FOR SUBPROGRAMS AND TASK PRO-	THIS PART OF THE EXTERNAL PACKAGE IS LOADED INTO THE DATA TEMPLATE
NESTED PACKAGE AREA INCLUDING NESTED PACKAGE HEADERS, GLOBAL DATA, AND AUTOMATIC DATA TEMPLATES FOR SUBPROGRAMS AND TASK PROGRAMS IN EACH NESTED PACKAGE	MEMORY
INSTRUCTIONS FOR SUBPROGRAMS AND TASK PRO- GRAMS IN THE NON-NESTED PACKAGE	THIS PART OF THE EXTERNAL PACKAGE IS LOADED INTO THE
INSTRUCTIONS FOR SUBPROGRAMS AND TASK PRO- GRAMS IN NESTED PACKAGES	INSTRUCTION MEMORY

and a second second

ø

FIGURE 2-1 EXTERNAL PACKAGE REPRESENTATION.

PACKAGE	
COMPONENT	
OFFSET	4 32
-N	CONTI OFFSET TO NESTED PACKAGE DATA TEMPLATES
	4 32
-(N-1)	CONT OFFSET TO NESTED PACKAGE INSTRUCTIONS
	4 32
-(N-2)	CONT OFFSET TO NESTED PACKAGE CONSTANT GLOBAL DATA
	4 32
-(N-3)	CONT OFFSET TO NESTED PACKAGE HEADER
	4 4 8 20
-(N-4)	DESC PPGM # COMPONENTS VARIABLE GLOBAL DATA SIZE
	4 32
-10	CONT OFFSET TO PROGRAM# AUTOMATIC DATA TEMPLATE
	4 32
-9	CONT OFFSET TO PROGRAM# 1 LAST INSTRUCTION
-8	CONT OFFSET TO PROGRAM# 1. INSTRUCTIONS
Ŭ	$\frac{1}{4} \frac{1}{4} \frac{1}$
-7	
-0.	DESCITION ACTIVATION RECORD SIZE
	4 20
e .	
-5	ICONTI OFFSET TO PROGRAM# U AUTOMATIC DATA TEMPLATEI
-4	ICONTI OFFSET TO PROGRAM# U LAST INSTRUCTION
	4 32
-3	CONTI OFFSET TO PROGRAM# 0 INSTRUCTIONS
	4 4 4 4 16
-2	CONTI I N.D. EXCPTI FORMAL PARAMETER MASK
	4 4 28
-1	DESCISPGMI ACTIVATION RECORD SIZE
	4 4 8 20
0	DESC PKG # COMPONENTS VARIABLE GLOBAL DATA SIZE
PKG - PACKAG	GE DESCRIPTOR EXCPT- EXCEPTION MODE
SPGM- SUBPRO	OGRAM DESCRIPTOR N.D NESTING DEPTH
TPGM- TASK	PROGRAM DESCRIPTOR PRLVL- PRIORITY LEVEL
PPGM- PACKA	GE PROGRAM N/5 - # OF 5-WORD HEADER
DESCR	IPTOR COMPONENTS

FIGURE 2-2 EXTERNAL PACKAGE HEADER.



and the first of a set of the set

FIGURE 2-3 ACTIVATION RECORD OBJECT.

1	ADMINIS	TRATIVE DATA
DESC	DOD	SIZE OF DOD
AVO	ARRAY	VALUE OFFSET
DESC	LB/UB	LOWER UPPER BOUND BOUND
V32D	. I	NITIAL VALUE
 SEP (OV] IND 	ARATE A ER 267 IRECTLY	RRAY VALUE AREA MILLION WORDS ADDRESSABLE)

FIGURE 2-4 DATA OBJECT - EXAMPLE OF ARRAY.

3 DATA FORMATS

The ISA supports 16-bit and 32-bit integers, 32-bit and 64-bit floating point data, 8-bit characters, 16-bit, 32-bit, and 64-bit mask data, 1-bit Booleans, 96-bit pointers, 96-bit formal reference parameters, and variable size records and arrays. A data cell is defined as a unit of addressable data that contains a tag per word and a value part. Each data cell except records and arrays has a 4-bit tag that identifies the bit size of the value field and specifies whether the value is defined or undefined. Tags specifying value sizes greater than 32 bits require each additional 32-bit value to follow consecutively. The tag identifier of each additional 32-bit value is the CONTinued taq. Variable size records and arrays have a 4-bit tag that identifies an extension to the tag description and the next 4-bit field specifies the type of description in the remaining 28 bits. Descriptor words and initial values of components are combined to completely define records and arrays.

3.1 <u>16-Bit Value Data (V16)</u>. Two 16-bit value fields are packed in 32 bits. The 4-bit tag, V16, specifies whether each of the value fields is defined or not. If only one 16-bit value is required, bit 33, which is the undefined bit of value field(2), must be set to 1 (undefined) and value field(2) must contain the value "one". This field corresponds to undeclared data and can never be addressed. Undefined data (declared but not assigned an initial value by the compiler) cannot be read until the program assigns a value, changing the UNDEFINED bit to 0. The manner in which each value is designated as defined, undefined, or undeclared is shown below. Here, XXXX represents a defined hexadecimal value.

MSE 35 32 31	LSB MS 16 19	SB LSB
V16	VALUE FIELD 2	VALUE FIELD 1
TAGID V16DD=0	VALUE(2) XXXX	VALUE(1) XXXX
V16DU=1	XXXX	0000=> UNDEFINED 0001=> UNDECLARED
V16UD=2	0000=> UNDEFINED 0001=> UNDECLARED	XXXX
V16UU=3	0000=> UNDEFINED 0001=> UNDECLARED	0000=> UNDEFINED 0001=> UNDECLARED

Defined 16-bit value fields may represent a 16-bit signed (2's complement) integer, an 8-bit character, a 16-bit mask, or a 1-bit Boolean. The data type represented by the value field is determined by the instruction. Examples of 16-bit integers and 8-bit characters are shown in Tables 3.1 and 3.2, respectively.

 Integer 	16-bit value field
32,767	7FFF
16,384	l 4000 l
4,096	1000
2	I 0002 I
1	0001
1 0	I 0000 I
-1	FFFF
-2	FFFE
1 -4,096	F000 .
-16,384	I C000 I
-32,767	I 8001 I
1	

Table 3.1 16-bit integer numbers.

Table 3.2 8-bit characters in the 16-bit value field.

Character	 16-bit value field
"A"	0041
"B"	0042
"C"	0043
"a"	0061
"b"	0062
space	0020
"Z"	005A

The 16-bit mask data represents 16 binary digits that may be set or cleared collectively or on an individual bit basis. Four examples of 16-bit mask data are shown in table 3.3.

Table 3.3 10-DIC mask de	ata.
--------------------------	------

Mask#	16-bit value field
1	0000
2	0001
3	A5C3
4	E976

The binary mask values "zero" and "one" (masks #1 and #2 in Table 3.3) can also represent Boolean values FALSE and TRUE, respectively.

3.2 <u>32-bit Value Data (V32)</u>. The 4-bit tag, V32, specifies whether the 32-bit value field is defined or not. Again, XXXXXXXX represents a defined hexadecimal value.

35	MSB 31	alerer das	and the second second	LSB 0
V32			VALUE FIELD.	•
TAGID V32D=4		1	VALUE XXXXXXXXX	
V32U=5	5		UNDEFINED	

Defined 32-bit value fields may represent a signed 32-bit integer, 32-bit mask data, or a 32-bit floating point number in the IEEE standard format. Examples of 32-bit integers are shown in table 3.4.

Table 3.4 32-bit integer numbers.

Integer	32-bit value field
2.147.483.647	7FFF FFFF
1,073,741,824	4000 0000
2	0000 0002
0	0000 0000
-2	FFFF FFFE
-1,073,741,824	C000 0000
-2,147,483,647	8000 0001

The format of the 32-bit mask data is the same as the 16-bit mask data except that the size of the mask is 32 bits. Floating point numbers are represented in the IEEE floating point standard format for single precision (32-bits) and double precision (64-bits) numbers. Single precision floating point contains a 1-bit sign (0=plus, 1=minus), an 8-bit exponent, and a 23-bit fraction.

35	MSB 31 30	23 22		LSB 0
V32	S EXPONE	NT	FRACTION	

Exponents are represented in excess 127. Fractions are represented in unsigned binary for numbers in the range $0..1-2^{-23}$. For exponents in the range +1..+254, the number represented is:

(-1) sign * 2(exponent-127) * (1 + fraction).

These numbers, represented with full precision, are called normalized. For an exponent of 0, the number represented is:

(-1) sign * 2^{-126} * (0 + fraction).

一日 いいいい ない 日本 ひとう ちょうちょう

■「これになる」■「ことなるなどの」 ペイン マイン ■ たらせのせる

In this case, the number is represented with less than full precision and is called denormalized. Examples of 32-bit floating point numbers are shown in table 3.5

 Decimal Number 	Sign	Exponent	Fraction
0.5x2 ¹²⁷	0	FD	000000
0.625x2 ⁴	. 0	82	200000
0.5x2 ¹	0	7F	000000
0.5×2^{-1}	0	7D	000000
$.25x2^{-127}$	0	00	200000
-1.0x2 ⁰	1	75	000000
-0.7500001x2 ⁴	1	82	400002

Table 3.5. 32-bit floating point numbers.

3.3 <u>64-bit Value Data (V64)</u>. The 4 bit tag, V64, specifies whether the 64-bit value field is defined or not. Defined 64-bit data may represent 64-bit mask data or a double precision floating point number. The 64-bit mask data contains 64 bits and occupies two words. The tag of the second word is CONTinued (tag ID = C). Double precision floating point requires two 32-bit value fields to represent a 1-bit sign, an 11-bit exponent, and a 52-bit fraction.



The 64-bit floating point exponent is represented in excess 1023. For exponents in the range +1..+2046, the number represented is:

(-1) sign $\star 2$ (exponent-1023) $\star (1 +$ fraction).

For an exponent of 0, the number represented is:

(-1)sign * 2⁻¹⁰²² * (0 + fraction).

3.4 96-bit Pointer (PTR). Three 32-bit value fields are required to represent pointers to whole storage objects, data entities in the global storage area of packages, and non-nested subprograms. The value field of the first word of the pointer contains a 1-bit UNIQUE NAME flag, a 3-bit ENTITY (ENT) subfield, a 4-bit RIGHTS subfield, and a 24-bit subfield whose contents depend on the pointed-to entity specified by the ENT subfield (see Table 3.6). If ENT designates a program in an external package, the 24-bit subfield contains the offset, in number of words, from the address of the package header to the program (subprogram or task program) component in the header (see Figure If ENT designates a data object, the UNIQUE NAME flag 2-2).controls whether the 24-bit subfield contains a UNIQUE NAME. If the flag =1, a UNIQUE NAME is present; if the flag =0, the 24-bit subfield is ignored. The value fields of the second and third words of the pointer contain, respectively, the ABSOLUTE ADDRESS of the ENTITY TEMPLATE and the ABSOLUTE ADDRESS of the ENTITY in data value memory (the latter not used when the pointed-to entity is a data entity in the constant global data area of a package.) There is a special use of the pointer data type that does not involve pointing to an entity. If ENT designates "package load addresses", (1) the subfields that occupy bits 0..27 in word 1 of the pointer are ignored, (2) the value in the second word of the pointer is the absolute address of a package in data template memory, and (3) the value in word 3 of the pointer is the absolute address in instruction memory of the first instruction of the first program contained in the package. This special pointer is returned when the instruction, ALLOCATE PACKAGE STORAGE is executed during loading of package (see Sections 7 and 7.4).

35 32	31 30 28 27 24 23	0
PTR	ENT RIGHTS VALUE DEPENDS ON ENTITY	
TAGID PTRD=8 PTRU=9 UNI NAM	QUE E FLAG	
35 32	31	0
CONT	ABSOLUTE ADDRESS OF ENTITY TEMPLATE	
35 32	31	0

3-6

and the second second

Additional information on pointers is found in Section 10.

Table 3.6 Pointed-to Entities.

ENTITY CODE	ENTITY
000	PACKAGE OBJECT
001	TASK OBJECT
010	DATA OBJECT
011	DATA ENTITY IN VGD
100	DATA ENTITY IN CGD
101	PROGRAM IN EXTERNAL PACKAGE
110	PACKAGE LOAD ADDRESSES

In the above table, VGD = variable global data area and CGD = constant global data area.

3.5 <u>96-bit Formal Reference Parameter (FRP)</u>. A formal reference parameter contains a path to the actual parameter and specifies the rights which the called subprogram has to the actual parameter. Formal reference parameters have the same forme as a pointer to a data entity. The ABSOLUTE ADDRESS of the ENTITY TEMPLATE is the base address of the caller's data template (or of an enclosing scope's template or the package variable or constant global data) plus the offset to the actual parameter. The ABSOLUTE ADDRESS of the ENTITY is the base address of the caller's activation record (or of an enclosing scope's activation record or the package variable global data) plus the same offset. The FRP is assigned only during parameter binding.

35 32	31	30 2	8 27	24 2	3	•	 0
FRP	-	ENT	RIC	SHTS		der The	
TAGID FRPD=A FRPU=B				Rea Wri	d = te =	XXX1 XX10	

35	32	31		0
cc	NT	1	ABSOLUTE ADDRESS OF ENTITY TEMPLATE	
35	32	31		0
1		T T		

ABSOLUTE ADDRESS OF ENTITY

CONT

n	the	FRP,	the	UNIQU	E NAME	flag	and	the	UNIQUE	NAME	field	are
n+	1100	he	Tho	ENT fi	ald can	ACCIMA	only	r the	follo	wind	values	

ENT	meaning
111	actual parameter in caller's activation or an enclosing program's activation
011	actual parameter in variable global data area of enclosing package
100	actual parameter in constant global data area of enclosing package

See Sections 6.3.1 and 6.3.2 for information on the use of FRPs.

3.6 <u>Variable Size Record (REC)</u>. The record descriptor (REC) specifies the number of record components and the number of words required to describe the record. The tag ID of the record belongs to the extended tag group called DESCriptor tags. The next four bits identify the extended tag as a record. The value field is divided into an 8-bit subfield specifying number of record components (#C) and a 20-bit subfield specifying the size, in number of words (#W), of the total record description. Immediately following the record descriptor are #W-1 words that

define the components, including initial values. An initial value is any legal value or UNDEFINED, preset by the compiler. The entire record description would be located in the data template of an activation record or a data object.

35 32	31 28 2	7 20	19			0
DESC	REC #	COMPONENTS	 # 1	WORDS IN	RECORD	DESC
TAGID DESC=D	EXTENDED REC=0		<u>,</u>	-		

Figure 3-1 shows an example of the record format with three initialized components including a 16-bit value, a 32-bit value, and a 96-bit pointer. Note that the second value field of V16(bits 16..31) is marked as undeclared. This corresponds to the Ada construct in Figure 3.2 which declares one integer (KEY). The pointer has been initialized to NULL.

35 32 31 2	28 27 20 19	0
DESC REC	#C = 3	#W = 6
35 32 31	<u> </u>	$\frac{0}{100} = 5$
141000 1	value - 1	value
35 32 31		0
V32D	Value = 1,073,741,8	24
35 32 31		0
PTRU	Null	
35 32 31		0
CONT	Null	
35 32 31		0
CONT	Null	

Figure 3-1 Three component record format.

Figure 3-2 Ada record construct.

Variable Size Array Header. 3.7 Arrays are defined as collections of homogeneous data entities. An array header describes the dimensions of the array (lower bound, upper bound, and span for each dimension), specifies initial value(s) of array components, and designates, explicitly or implicitly, the location of the array component values. Array headers facilitate automatic subscript vs bounds checking and array component address calculation from the subscripts. (See Section 4.4.3 on Array Subscripts.) In the header, the bounds in any dimension can be designated as being unconstrained (not known at compile Then, the values are supplied at run time, when storage time). is allocated for the array. Array headers with constrained bounds (values preset by compiler) may be located in the data template of activation records or data objects. Headers with unconstrained bounds can only be located in the templates of data In addition to dimension information, the header objects. contains one or more initial component values that were defined in the source program and preset by the compiler. Components that are not assigned initial values must be marked as UNDEFINED by the compiler.

defined Constrained arrays may be with component values immediately following the header or with component values separated from the header by the array value offset. In the former case, the location of the array component values is. implicit (at the end of the header); each component can have a separate initial value. These initial values are located in the data template of the array; hence, as new values are assigned, they are written to corresponding locations in data value memory. In the latter case, the location of the array component values is explicitly given by an array value offset to an area of storage designated for separate array values (at the end of an activation record or data object). These values do not have a corresponding template. Hence, the components cannot have individual initial However, the header contains one initial value (or values. UNDEFINED) that applies to all components. Unconstrained arrays must be of the class with separate array component values.
3.7.1 Lower Bound and Upper Bound (LB/UB,LB,UB). Array headers with immediate values must start with one of the following descriptors:

35 32	31 28	27	16	15	0
DESC	LB/UB	LOWER BO	UND (LB)	UPPER BOUND()	UB)
TAGID DESC=D	EXTENDE LB/UB=3	<u>3D</u>			
	LB=> rang LB=8	> 2's comp Je=> -2 ¹¹ . 300=> Unco	lement. .+(2 ¹¹ -1). nstrained.	UB=> 2's comp range=> -2^{15} . UB=8000=> Unce	lement .+ (215_ onstrain
35 32	31 28	27		and the second second	. 0
DESC	LB		LOWER	R BOUND (LB)	
TAGID DESC=D	EXTENI LB=1	DED	LB=> 2's range=> LB= 8000	s complement. -2 ²⁷ +(2 ²⁷ -1). 0000=> Unconstrained	d.
35 32	31 28	27			. 0
DESC	UB		UPPER	R BOUND (UB)	
TAGID DESC=D	EXTENI UB=2	DED	UB=> 2's range=> UB=80000	s complement. -2 ²⁷ +(2 ²⁷ -1). 000=> Unconstrained	

Note that the LB/UB descriptor format combines the lower bound and the upper bound into one word for describing small to medium dimensions. Larger dimensions require two consecutive words containing a 28-bit lower bound value and a 28-bit upper bound value. Multiple dimensions are indicated by successive lower and upper bound values.

3-11

Labit Alexander Service

3.7.2 <u>Multi-dimension Span (SPAN)</u>. Arrays with two or more dimensions require an additional descriptor word for each dimension over the first. This descriptor is called the SPAN of the nested dimension. For any dimension, i,

 $SPAN_i = Length_{i-1} * SPAN_{i-1}$

where i-l represents the number of the immediately nested dimension and length = upper bound - lower bound + 1. For the innermost dimension (dimension 1), SPAN is replaced by the array component size, expressed in number of half-words (V16=> 1 halfword, V32=> 2 halfwords, etc.). Hence, for an n dimensional array,

SPAN2 = Length1 * Component Size
SPAN3 = Length2 * SPAN2
SPAN4 = Length3 * SPAN3

 $SPAN_n = Length_{n-1} * SPAN_{n-1}$

and the total array size is

 $SIZE = Length_n * SPAN_n$.

The advantage of including SPANs in the header is a simplification of the component address computation. (See Section 4.4.3.)

35 32 31 28 27	0
DESC SPAN	SPAN _i VALUE FIELD
TAGID EXTENDED	Range=> 02 ²⁸ -2 halfwords.
	SPAN=FFFFFFF=> Unconstrained.

When the last dimension (dimension 1) is specified, the components are listed with their initial values. The component size is derived from the first component tag following the dimension 1 descriptor.

/UB LB	3 = -2			
			<u>$UB_3 = -1$</u>	
PAN	SPAN3	= 6 hal	fwords	<u> </u>
UB I	$LB_2 = 0$		UB ₂ = 2	1
PAN	SPAN2	= 2 hal	fwords	<u> </u>
/UB	$LB_1 = 1$		<u>UB1 = 2</u>	
VALUE	= 2		VALUE = 1	<u> </u>
VALUE	= 4		VALUE = 3	
VALUE	= 6		VALUE = 5	1
VALUE	= 8		VALUE = 7	
VALUE	= 10		VALUE = 9	1
	PAN /UB PAN /UB /UB VALUE VALUE VALUE VALUE	PAN SPAN3 $/UB $ $LB_2 = 0$ PAN SPAN2 $/UB $ $LB_1 = 1$ $/UB $ $LB_1 = 1$ $VALUE = 2$ $VALUE = 4$ $VALUE = 4$ $VALUE = 6$ $VALUE = 8$ $VALUE = 10$	PAN SPAN3 = 6 hal: /UB LB2 = 0 PAN SPAN2 = 2 hal /UB LB1 = 1 /VALUE = 6 I /VALUE = 8 I /VALUE = 10 I	PAN SPAN3 = 6 halfwords /UB $LB_2 = 0$ $UB_2 = 2$ PAN SPAN2 = 2 halfwords /UB $LB_1 = 1$ $UB_1 = 2$ /VALUE = 2 VALUE = 1 VALUE = 4 VALUE = 3 VALUE = 6 VALUE = 5 VALUE = 8 VALUE = 7 VALUE = 10 VALUE = 9

1.1.1.1.1.1.1.

5. 5. S. S. S.

Figure 3-3 3-dimensional array of 16-bit values.

Figure 3-4 Ada array construct.

ote that the last component of ARR_DESC (-1,2,2) is not nitialized by the Ada construct. Therefore, the compiler sets he undefined bit for value field(2) in the tag and assigns all eros to value field(2). .

.

3.7.3 <u>Separate Array Value Offset (AVO)</u>. Headers of constrained arrays with separate values must start with the Array Value Offset.(AVO) descriptor. The AVO contains a 28-bit self-relative offset (in words) to the start of the separate array value area at the end of an activation record or data object.

35 32	31 28 27	·····	0
DESC	AVO	RELATIVE OFFSET	
TAGID DESC=D	EXTENDED AVO=7	Offset to separate values. Range => 32 ²⁸ -1 words.	

Following the AVO descriptor are the lower bound, upper bound, and SPAN of each dimension in the array and a single component descriptor (tag and initial value) that applies to all the array components in the separate array value area.

A description of a 1. dimension array with a lower bound of 1 and an upper bound of 128 is shown in figure 3-5. The start of the separate values is 48 words from the AVO descriptor and each component is initialized to the value 16.

35	32	31	28	27				0
DES	SC	AV	0		RELATIV	E OFFSET =	= 48	
35	32	31		27	1	6 15		0
DES	SC	LB/	UB		LB=1		UB=128	
35	32	31						0
V:	32D	1			VALUE =	16		

Figure 3-5 Separate constrained array header.

3.7.4 Dynamic Array Value Address (AVA). Array headers with unconstrained bounds must start with the array value address (AVA) descriptor. AVA contains the 32-bit address of dynamically allocated storage for separate array values. Initially, the machine sets the undefined bit in all AVAs to "1" (UNDEFINED). When the template of a data object is an unconstrained array (or is a record with one or more unconstrained array components), the array bounds are supplied by an operand qualifier (see Section 4.4.5) in the instruction, CREATE DATA OBJECT. Then, the size of the array(s) can be >mputed and storage can be allocated for the data object. Since ne location(s) of the separate array values becomes known, the achine loads a valid address into each AVA and changes the idefined bit to "0" (DEFINED).

35	31	<u> </u>
AVA		ARRAY VALUE ADDRESS
TAGID AVAD=E		32-bit absolute address of dynamically allocated array
AVAU=F		storage. Absolute addresses are assigned by the machine only.

pllowing the array value address descriptor are the lower bound, oper bound, and SPAN of each dimension in the array and a omponent descriptor (tag and initial value) that applies to all he array components in the dynamically allocated separate array alue storage area. As mentioned earlier, unconstrained array aders can only occur in data object templates.

.7.5 Total Record Size (TRS). The total record size escriptor is used in an array header when the total size of a scord which is an array component is not given by the "number of ords in record description" field in the REC descriptor. This ase arises when an array with separate values has a record omponent and the record contains a component which is an array ith separate values. Each array may be unconstrained or onstrained with separate values. TRS contains the correct array omponent size which is used in array component address TRS precedes the REC descriptor of the record that omputations. ontains a component which is an array with separate values. The alue of the TRS descriptor is a 28-bit field representing total ecord size, in number or words. Appendix B gives four examples f arrays with record components, when the record contains an rray component. These examples illustrate when TRS is and is ot required.

36 32 31 28 27	
DESC TRS	28-BIT TOTAL RECORD SIZE -
TAGID EXTENDED DESC=D TRS=8	NUMBER OF WORDS IN TOTAL RECORD.
	TRS=FFFFFFF=> UNCONSTRAINED.

3.8 Data Object Descriptor (DOD). A Data Object Descriptor specifies the fixed storage size (known at compile time) of a dynamically allocated data object. A DOD identifies the associated data template as that of a data object. These templates are contained in the constant global data area of packages (see Section 2.4). The DOD contains a 28-bit value field representing size, in number of words, as shown below:



Data objects can be any data type including records and arrays, but excluding pointers and formal reference parameters. The type is specified in the description (template) that follows the DOD. Data objects can be constrained or unconstrained. Unconstrained data objects are unconstrained arrays or contain records with one or more components that are unconstrained arrays.

3.8.1 <u>Constrained DOD</u>. The DOD of a constrained data object specifies the total size of data value memory to be allocated dynamically, in a CREATE DATA OBJECT or CREATE UNCHECKED DATA OBJECT instruction. This includes the size of the data template plus the size of separate array values, if the data object is (or contains) an array with separate values.

Since the storage size of an 3.8.2 Unconstrained DOD. unconstrained data object is not known at compile time, the size field in the DOD specifies only the size of the unconstrained array or unconstrained record description (data template). Bounds of unconstrained arrays are supplied in the instruction, CREATE DATA OBJECT or CREATE UNCHECKED DATA OBJECT. These instructions first allocate storage in data value memory of a size equal to that specified in the DOD. Then, lower and upper bounds and computed quantities that depend on the values of the bounds (SPAN and Total Record Size, if a record) are written into data value memory at locations having the same relative offsets as their counterparts in the template in data template memory. The total size of the unconstrained data object, including separate array values, can then be computed and storage allocated for it. The addresses (in the allocated storage) of the separate array values for each unconstrained array can then be written into the Array Value Address descriptors, completing the operation. This process is illustrated in Example 4 of Appendix B. An example of an unconstrained data object description is shown in Figure 3-6. Here, the symbol < > stands for "unconstrained" value (= 800...0 hey).

35	32 31	28	27					0
	ESC DO	DI				1;	2	
35 A	32 31				DON'T C	ARE		0
		•						· · ·
35 DI	32 31 ESC LB/	28 UB	27	< >	16	15	< >	0
35	32 31	28	27					0
DI	ESC TR	<u>s</u>				< >		
35 .	32 31	28	27		20 19	_		0
DI	ESC RE	C		4			8	
35	32 31	28	27				. ·	0
	ESC AV	0 1				/		
35	<u>32 31</u>	28	27		16	15	<u>_</u>	0
וע ו				<u>4</u>			40	
35 1 V	32 31					0		.0
					· ·			
35	32 31		_		16	15		0
V10	6DD			15			-3	I
35	32 31							0
A	VAU	_			DON'T (CARE		
35	32 31	28	27		16	15		0
	ESC LB/	UB		< >		l	< >	
35	32 31							0
V:	32D			-2,	147,483,0	647		

Figure 3-6 Unconstrained data object description.

The data object description in Figure 3-6 defines an unconstrained array of unconstrained records. The record descriptor defines four record components including an array with 48 separate 32-bit values, two 16-bit values, and an unconstrained array with a 32-bit component.

4 INSTRUCTION FORMATS

The instruction set supports 1-operand, 2-operand and 3-operand instructions. Twenty-eight operand formats specify 1-operand, 2operand, and 3-operand combinations of memory, register, immediate, and stack references. Instructions vary in size from one word (36 bits) to N words, depending on the number of operands.

4.1 <u>Operation Code (OPCODE)</u>. The operation code consists of the eight most significant bits (bits 35..28) of the first instruction word. The OPCODE identifies the action to be taken, the number of operands involved, and the value representation of the operands.

4.2 <u>Operand Formats (FMT)</u>. Immediately following the OPCODE is a 4-bit format (FMT) field (bits 27..24) specifying one of fourteen memory reference formats or format extend. Additional non-memory reference formats are obtained by extending the format field by 4 bits (bits 23..20) when FMT = Extend.

4.2.1 <u>Memory</u>. Reference to data in memory requires a 4-bit address space (ADS) specifier and a 20-bit cell offset (CO). The address space field specifies the nesting depth of the addressed data (in the local activation record, an activation record of an enclosing subprogram or task program, or the constant or variable global data area of the package).

lesting Depth	Location of Addressed Data
15	constant global data
0	variable global data
1	non-nested subprogram or task program
214	nested subprograms or task programs

The nesting depth of addressed data must be less than or equal to the current nesting depth or 15. A nesting depth in the range 0..14 designates a display register pair that contains the base address of the activation record (or variable global data area) and the base address of the activation record's data template (or the variable global data area's data template). Nesting depth 15 designates the display register that contains the base address of the constant global data in data template memory. The 20-bit cell offset is a halfword offset relative to the base addresses mentioned above. Hence, the machine adds the CO to the base address of an activation record or data template to locate a data cell. The residency bit determines which absolute address (in data value memory or data template memory) is used. The range of the CO value for memory references is 32 to 1,048,575 halfwords. (Offsets 0 to 31 are reserved for register addresses.)



Data cells that are located within the first 1,024 halfwords of the activation record (or global data area) can be referenced with a compacted (shorter) cell offset format. Compact formats allow the specification of multiple operands in a single word. Two memory operands residing in the same activation record (or global area) can be referenced with the 4-bit ADS field and two 10-bit cell offsets. The range of the short cell offset values for memory reference is 32 to 1,023 halfwords.

35	28	27	24	23	20	19	_	10	9		0
		 MM		AD	S	 CELL 	OF	FSET2	CELL	OFFSET	
8-bit opera	tior	n				10-b	it	cell of	fset2.		<u> </u>
code	4-1 spe	bit ecif	forn ier	natl		Rang	e=>	2 ⁵ 2 ¹	0-1.		
for 2 memory operands (See Table 4.1)								10-b offs	it cell etl.		
							Ra	ange=>	2 ⁵ 2 ¹⁰)-1.	
			4-b:	it a	ddre	ess			-	·	

4.2.2 Formats. Table 4.1 shows the fourteen memory format codes. Here, MEM=Memory, IMM= Immediate, and STK=Stack operand. Note that the table includes several 2-operand and 3-operand formats. For example, if FMT code=9, the 36-bit instruction would specify OPCODE, FMT, ADS, and a full length CO field and would imply the presence of two other operands on the stack. (Stack operands are zero-address.) Normally, in multiple operand instructions, the operand specified last (memory operand, in this example) is the destination and the other operands (on the stack, in this example) are sources.

 FORMAT CODES (FMT)	MEMORY RI OPERAND LOCATIONS	EFERENCE (ADS) FORMATS FIELD SIZES
0 = M 1 = MM 2 = IM 3 = MI 4 = SM 5 = MS 6 = MMS 7 = MSM 8 = SMM 9 = SSM 10 = MSS 11 = SMS 12 = MIS 13 = IMS 14 = - 15 = Extend	Memory Mem-Mem Imm-Mem Mem-Imm Stk-Mem Mem-Stk Mem-Stk-Mem Stk-Mem-Mem Stk-Stk-Mem Mem-Stk-Stk Stk-Mem-Stk Mem-Imm-Stk Imm-Mem-Stk Reserved	C0 = 20 bits Each C0 = 10 bits Imm = 10 bits, CO = 10 bits C0 = 10 bits, Imm = 10 bits C0 = 20 bits C0 = 20 bits Each C0 = 10 bits Each C0 = 10 bits Each C0 = 10 bits C0 = 20 bits C0 = 10 bits, Imm = 10 bits Imm = 10 bits, C0 = 10 bits

Table 4.1 Memory Formats.

Format code 14 is reserved for future use. Format code 15 extends the format field by an additional 4 bits to specify combinations of operands on the stack, in registers, immediate values, and base-relative values. The extended format codes are shown in Table 4.2.

· ·	I NON-MEMC	DRY REFERENCE FORMATS
EXTENDED	OPERAND	
FORMAT CODES	LOCATIONS	FIELD SIZES
EXTENDED FORMAT CODES 1 = SS 1 = SS 2 = I 3 = II 4 = SI 5 = ISS 6 = SIS 7 = ISS 8 = RRR 9 = IRR 10 = RIR 11 = B 12 = BI	LOCATIONSLOCATIONSStackStk-StkImmediateImm-ImmStk-ImmStk-Imm-StkImm-Stk-StkImm-Stk-StkReg-Reg-RegImm-Reg-RegReg-Imm-RegBase RegBase-Imm	FIELD SIZES Imm = 20 bits Each Imm = 10 bits Imm = 20 bits Brach Reg = 5 bits Imm=10 bits, Each Reg= 5 bits B-Reg = 5 bits, Imm = 10 bits
13 = BM	Base-Memory	B-Reg=5 bits, ADS=4 bits,
		CO=10 bits
14 -	Reserved	
15 -	Reserved	

Table 4.2 Non-Memory Formats.

REGISTER. Thirty-two 36-bit registers comprise the 4.2.3 register file which is divided into two groups: registers 0 through 15 and registers 16 through 31. The former group comprises general purpose registers (of concern here) while the latter group is dedicated to passing parameters (see Section 6.3.1). Register 0 contains two 16-bit control fields, leaving registers 1 through 15 for general purpose use. Bits 0..15 of register 0 is called the Temporaries Mask. (Bits 16..31 of register 0 comprise the Valid Parameter Mask for control of parameters.) Each bit in the Temporaries Mask corresponds to a general purpose register in the following way: bit 0 corresponds to register 0 (itself); bit 1 corresponds to register 1; ...bit 15 corresponds to register 15. Whenever a general purpose register is written into, the corresponding bit in the Temporaries Mask is automatically set to "1". If an attempt is made to read a register when the corresponding bit in the Temporaries Mask is not "1", a PROGRAM ERROR is raised. The Mask has no control over writing to registers. In presence of a task switch, the contents of those registers corresponding to "ls" in the Temporaries Mask, including register 0, are automatically saved in the current task object's administrative data area; the values are restored when the task is again scheduled to run. The Temporaries Mask is automatically cleared for the called subprogram when the CALL SUBPROGRAM instruction is executed. The instruction, CLEAR TEMPORARIES MASK, is provided to allow (There is no need to save/restore compiler optimization. registers when the data they contain is garbage.)

Cell offsets in the range 1..31 address registers (ADS is ignored). Therefore, the cell offset in activation records starts at 32 halfwords. Note that the control register (register 0) is not generally addressable; this register is conditioned automatically by the machine and is modifiable by special instructions. A consequence of memory mapped register addressing is that all formats in Table 4.1 are usable for registers ("REG" replaces "MEM"). In addition, extended formats 8 through 13 utilize the general purpose registers (see Table 4.2). An instruction using format 8 (RRR) is shown below:

35	28	27	21	23	20	19		15	14	10	9	5	4	0
OPCODE		EX	T	RF	RR	-	-		RE	G3	REG	2	REG1	

Here, two source operands may be located in registers 1 and 2 and register 3 may be the result destination. Extended formats 9 and 10 are similar, with a 10-bit immediate value replacing one register operand.

When writing to a register, the 4-bit tag as well as the value field are loaded. Data types V16, V32, V64, and pointers can be loaded into general purpose registers. (Formal reference parameters can be loaded into parameter registers 16 through 31). A sixteen bit value (V16) is loaded into bits 0..15 of a general register. A further feature permits array base addresses to be loaded into general purpose registers. The undefined bit in the base address tag identifies the array as having immediate values or separate values. Array value and template base addresses are loaded into registers by the instruction, LOAD ARRAY BASE ADDRESS (see Section 5.1.4). Two consecutive general purpose registers, addressed by CO and CO+1, are loaded as shown below:

(a) Arrays with separate values

Register Address	< Tag	REGISTER> Value Field
CO (114)	F	Address of 1st component in array value space in data value memory
CO + 1	CONT	Address of component tag/initial value in array header in template

(b) Arrays with immediate values

Register Address	< <u>Tag</u>	Value Field
CO (114)	E	Address of 1st component in array value space in data value memory
CO + 1	CONT	Address of 1st component tag/ initial value in array header in template

Note that the tag used for the base address in data value memory is the same as that used for Array Value Address (AVA). This does not cause a problem since array headers are not permitted in

registers. Similarly, base addresses are only permitted to reside in registers. An array component value is addressed by adding an offset to the base address of the array values. The base address is contained in the first register of each pair. In case b (arrays with immediate values), the same offset added to the base address of the component description in the header yields the address of the tag/initial value of the component. This base address is contained in the second register of the pair. (The residency bit automatically selects DTM or DVM as the source of the value.) In case a (arrays with separate values), base address in the second register of the pair addresses the single tag/initial value for the entire array. Instruction formats are available for base plus offset addressing, where the offset can be an immediate value or a memory operand (see extended format codes 12 and 13 in Table 4.2). The extended format code 12 (Base-Immediate) specifies a base register pair and a 10-bit immediate displacement (in halfwords) to the addressed array component.

35	28	27	24	23	20	19	15	14	10	9	0
OPCODE		 E:	хт	 B: 	t	B.	REG	 -		 IMMEDI 	IATE VALUE

Extended format code 13 (Base-Memory) specifies a base register, an ADS field, and a 10-bit cell offset. If CO is in the range 1..31, it specifies a register which contains the 32-bit offset and ADS is ignored; if CO is in the range $32..2^{10}-1$, then ADS and CO specify the memory location that contains the offset value.

35	28 2	7 24	23 20	19 15	13 10	9	0
OPCODE		EXT	BM	 B.REG 	- ADS	 CELL OFFSET 	

If the 10-bit immediate value (extended format 12) or the 10-bit CO (extended format 13) is too small, then extended format code 11 (base) that designates a base register is used with the operand qualifier, Base Relative Offset (BRO). BRO specifies an offset from the array base address using a full length format (see Section 4.4.6). Shown below is the base register (B) format:

35	28	27	24	23	20	19	15	14		0
OPCODE			KT		в	В.	REG		-	

rray component addresses can be derived by two methods. In the irst, the instruction addresses the array header and includes ubscript operand qualifiers (one per dimension). The machine, ising the information in the array header, automatically checks he subscripts vs bounds for each dimension and computes the irray component address using the subscripts and spans (see Section 4.4.3). The second method of addressing an array component involves use of base plus offset addressing, as lescribed earlier. Then, subscripts for each dimension must be :hecked by program, using the instruction, ASSERT RANGE, unless :he Ada index check is suppressed. Having base addresses in registers greatly speeds up array processing when frequent accesses to array components is required. Note that there will be times when the compiler cannot compute the array component offset, namely, when subscript values are not known at compile ime or when array bounds are unconstrained.

Instructions are available which move and perform logical operations on whole arrays and slices. For a whole array, these instructions either address the array header and the array size is automatically computed by the machine or address a base register and the pre-computed array size is gotten from the operand qualifier, ASIZ (see Section 4.4.7). Alternatively, the compact formats, BI and BM, may be used which (only for instructions that address whole arrays) are interpreted as specifying base register and array <u>size</u> (in halfwords). Note that extended formats B, BI, and BM (extended format codes 11, 12, and 13) may only be used when addressing arrays.

for a slice, the instructions either address the array header and include two slice index operand qualifiers (see Section 4.1.1) in the instruction stream (machine computes the location and size of the slice and checks the slice indexes vs bounds) or address the array base, pre-computed slice size, and the offset to the start of the slice. In the latter case, extended format 11 (base) is used with the operand qualifiers, Alternatively, BRO and ASIZ. the compact formats, BI or BM, may be used with the single operand qualifier, ASIZ. (Here, BI and BM are interpreted the same as when an array component is addressed, i.e., as a base register and offset.)

1.2.4 Immediate. Several forms of immediate addressing (operand value present in instruction) are provided. A full size immediate operand is 20 bits which may be a single operand in an instruction word (extended format I) or combined with one or two stack operands (extended formats SI, IS, SIS, and ISS). Several compact formats are available in which the immediate operand size is 10 bits (formats IM, MI, and IMS and extended formats II, IRR, RIR, and BI). Tables 4.1 and 4.2 list all these formats.

4.2.5 <u>Stack</u>. An expression stack of depth = 16 words is included in each activation record for use by subprograms and task programs. Although the purpose of the stack is to evaluate non-trivial arithmetic expressions, any V(16), V(32), and V(64) data type can be placed on the stack. V(16) types are unpacked on the stack (right justified). Instructions can combine stack, memory, and immediate operands as shown in Tables 4.1 and 4.2. Stack overflow (push beyond sixteenth value) and underflow (pop under first value) are detected and cause a PROGRAM_ERROR exception to be raised.

4.3 <u>New Operand Specifier (NOS)</u>. When an instruction has multiple operands, the first instruction word that specifies the first operand contains the OPCODE (bits 28 to 35). Each consecutive instruction word that specifies another operand contains the new operand specifier (code = FF) in place of the OPCODE. The number of new operand specifiers in an instruction depends on the OPCODE (instruction type) and FMT (full length or compact operands).

4.4 <u>Operand Qualifiers</u>. Operand qualifiers provide additional information about operands which cannot (for lack of room) be coded into operand specifiers in instructions. The type of an operand qualifier is coded into bits 28 to 35. The function and location in an instruction of each type of operand qualifier is shown below.

4.4.1 Bit Position (BPOS). This operand qualifier may be used in the MOVE instruction or any logical instruction that operates on V16, V32, or V64 mask data. A 6-bit field selects a bit in the mask operand addressed in the instruction. The selected bit takes part in the operation. BPOS follows the operand specifier that addresses the mask data in the instruction stream.

35	28 27 24 23		<u>65</u> 0
BPOS	FMT	-	BIT SEL
<u>INST-ID</u> 0000000	I GNORE		BIT SELECTOR FIELD

4.2 <u>Record Component Offset (RCO)</u>. An RCO operand alifier specifies, in number of half words, the offset from an dressed record descriptor (REC) to the desired component of the cord. RCO is a 20-bit immediate value. It is present in the struction stream following the operand specifier that addresses C; if the record containing the desired component is itself a mponent of an array, RCO follows the array component offset hen base register - offset addressing is used) or follows array bscript operand qualifiers (when the machine computes the array mponent address). The examples in Appendix x show how the chine uses RCO in address computations.



00000000

4.3 <u>Array Subscript (SUB)</u>. An array subscript operand alifier addresses an array component index in a particular mension of the array. To compute the offset to an array mponent, the machine requires n subscripts, where n = number of mensions in the array. A subscript is a signed integer that is instrained to be within the index bounds specified in the array ader for the particular dimension. Subscripts are present in the instruction stream, following the operand specifier that dresses the header, in the order of descending dimension mber.

SUB	FMT ADS	CELL OFFSET	

INST.ID 00000000

IT may specify memory, register, stack, or immediate format memory or register shown above); two subscripts may be combined a compact format.

le following general formulas show how the machine computes the ldress of a component in an i-dimension array using subscripts.

a) <u>Arrays with Separate Component Values</u>. Address of component i = array header address + AVO + (SUB_i - LB_i) * SPAN_i + (SUB_{i-1} - LB_{i-1}) * SPAN_{i-1} + ... + (SUB₂ - LB₂) * SPAN₂ + (SUB₁ - LB₁) * component size

here

AVO = Array Value Offset,

LB = Lower Bound,

SPAN₂ = Length₁ * component size,

 $SPAN_j = SPAN_{j-1} * Length_{j-1} (j = 3...i),$

ind

Length; = Upper Bound; - Lower Bound; + 1.

'or unconstrained arrays, Array Value Address (AVA) replaces 'array header address + AVO". (The value of AVA becomes known when storage is allocated for an unconstrained data object during the instruction, CREATE DATA OBJECT or CREATE UNCHECKED DATA OBJECT).

b) Arrays with Immediate Values.

The array component offset computation is the same for arrays with immediate values. However, the base address of array values is:

Array header address + size of header.

1.4.4 <u>Array Slice Index (SLICE)</u>. Array slice indexes are operand qualifiers (always present in pairs) that specify the index values of the first (lower index) and last (upper index) component in a slice in any array dimension.



FMT may specify memory, register, stack, or immediate format (memory or register shown above); the two slice indexes may be combined in a compact format. SLICE indexes are present in the instruction stream in the order "lower index, upper index" following the operand specifier that addresses the array header.

The offset to the first component of a slice is computed from subscripts, lower bounds, and SPANs as shown in Section 4.4.3 except that the slice's lower index is used in place of the highest dimension subscript. If the slice is in dimension i, the offset to the first slice component is:

 $(lower index - LB_i) * SPAN_i$

- + $(SUB_{i-1} LB_{i-1}) * SPAN_{i-1}$
- + ...
- + $(SUB_2 LB_2) * SPAN_2$
- + (SUB₁ LB₁) * component size.

The length of the slice is:

upper index - lower index +1.

Each slice index in constrained to be within the bounds of the dimension in which the slice is located.

4.4.5 Index Constraint (IDXCON). When lower and/or upper array bounds are unconstrained in a data object template, IDXCON operand qualifiers supply the values of the bounds, constraining the array. These operand qualifiers are present in the instruction stream (part of the instruction, CREATE DATA OBJECT) and appear in the same order as unconstrained bound descriptors in the data object template.

35		28	27	24	23	20	19		0
	IDXCON		 FM 	т	AD	S		CELL OFFSET	
<u>.</u>									·
	INST.ID 0000000	0							

FMT may specify any memory, register, stack, or immediate format (memory or register shown above). Note, however, that the 20-bit immediate extended format does not support the full size lower or upper bound descriptor value (28 bits). Compact formats may be used, as appropriate.

4.4.6 <u>Base Relative Offset (BRO)</u>. This operand qualifier supplies the array component offset (or the offset to the first component of a slice) relative to an array base address in a register. It is used when the offset value is too large to be accommodated in extended format BI (Base-Immediate) or BM (Base-Memory).



FMT may specify a memory, stack, or immediate value format (memory shown above).

4.4.7 <u>Array Size (ASIZ)</u>. The ASIZ operand qualifier is used with instructions that operate on whole arrays and slices. It specifies the size of the array or slice, in halfwords.

35	28 27	24 23	20 19		0
ASIZ	F	AT ADS		CELL OFFSET	
INST.ID					

FMT may specify a memory, stack; or immediate value format (memory shown above).

5 BASIC INSTRUCTIONS

The following pages describe the basic set of HLLM instructions including data movement, arithmetic, logical, and branching operations. Instructions that support special features of Ada, e.g., tasking, packages, exceptions, etc., are covered in other sections.

In the description of the instructions that follows, the only operand formats shown are for full length operands. This is for convenience; any appropriate compact format can be used by the compiler (see Tables 4.1 and 4.2). Of course, any memory format designates a register if the cell offset is in the range 1..31. In the description of the operands, S means Source and D means Destination. Enclosed in parentheses following each Format (FMT) alternative is the format code., e.g., memory (0), immediate (EXT, 2), where EXT refers to the extended codes in Table 4.2.

5.1 Data Movement

These instructions correspond to simple assignment statements in Ada, including assignment of whole arrays and records; also included in this group are instructions that change the representation of data during assignment, set data to "undefined", manipulate the stack (swap, purge), and clear the Temporaries Mask.

5.1.1 MOVE

Format: 00_H, S, D,

Mnemonic: MOV

Operands:

S: <u>Source to Be Moved</u> FMT: <u>immediate (EXT,2)</u>, memory (0), or stack (EXT,0)

D: <u>Destination of Move Operation</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The operand specified by S (immediate value, any data entity except a whole array or slice, or a data object) is copied to the destination location specified by D. Table 5.1 shows the legal combinations of source and destination operands. If the source or destination operand is a pointer to a data entity in global storage (ENT=011 or 100) or to a data object (ENT=010), the pointed-to entity, not the pointer takes part in the operation.

Exceptions: PROGRAM ERROR

source	destination
 immediate value scalar in memory or register) or stack array or record component 	 conforming scalar in memory (or register) or stack conforming array or record component
• whole record	• conforming whole record
 data object of any type except array 	 conforming data object conforming data entity in an activation record or in global storage
 any data entity except a pointer or an array in an activation record or in global storage 	• conforming data object

Table 5.1

5.1.2 MOVE ARRAY

Format: 01_H, S, D

Mnemonic: MOVARR

Operands:

S: <u>Source Array to Be Moved</u> FMT: <u>memory (0) or array base register (EXT,11)</u>

> If an array base register is specified, the array size must be provided by the operand qualifier, ASIZ; alternatively, the compact format BI or BM, may be used.

D: <u>Destination Array</u> FMT: <u>memory (0) or array base register (EXT,11)</u>

> If an array base register is specified, the array size must be provided by the operand qualifies, ASIZ; alternatively, the compact format BI or BM may be used.

Function:

The array operand specified by S is copied to the conforming array specified by D. Table 5.2 shows the allowable addressing combinations. If the source or destination operand is a pointer to a data entity in global storage (ENT=011 or 100) that is an array or is a pointer to a data object (ENT=010) of type array, the pointed-to array, not the pointer, takes part in the operation. Arrays in constant global storage must have immediate values. When an array is addressed via its header in memory, the machine computes the array size as the product of the highest numbered (outermost) dimension SPAN and Length; the machine also locates the first array component.

Exceptions: PROGRAM_ERROR

source	destination
 array addressed via an array header in memory array addressed via an array base register and an array size qualifier 	• any conforming array,
 array addressed via a pointer to an array header in the global storage of a package 	similarly addressed
 array addressed via a printer to a data object of type array 	

Table 5.2

•

5.1.3 MOVE ARRAY SLICE

- Format: $02_{\rm H}$, S, D,
- Mnemonic: MOVSL

Operands:

S:	Source Array Containing Slice to Be Moved
FMT:	memory (0) or array base register (EXT,11)
Memory:	upper and lower array slice index (SLICE) operand qualifiers and the necessary array
	subscripts (SUB) are present in instruction

- Base Req: base register offset (BRO) and slice size (ASIZ) operand qualifiers are present in instructions; alternatively, the compact format BI or BM may be used to provide base register and offset to start of slice with the operand qualifier, ASIZ, providing slice size.
- D: Destination Slice
 - FMT: memory (0) or array base register (EXT,11) Memory: upper and lower array slice index (SLICE) operand qualifiers and the necessary array subscripts (SUB) are present in instruction
 - Base Req: base register offset (BRO) and slice size (ASIZ) operand qualifiers are present in instructions; alternatively, the compact format BI or BM may be used to provide base register and offset to start of slice with the operand qualifier, ASIZ, providing slice size.

Function:

The array slice specified by S is copied to the conforming slice specified by D. Table 5.3 shows the allowable addressing combinations. If the source or destination operand is a pointer to a data entity in global storage (ENT=011 or 100) that is an array or is a pointer to a data object (ENT=010) of type array, the pointed-to array slice, not the pointer, takes part in the operation. Slices of arrays in constant global storage must have immediate values. When an array slice in a multi-dimensional array is addressed through the array header in memory, the machine computes the address of the slice from subscripts, lower slice index, lower bounds, SPANs, and array component size (see Section 4.4.4). It also computes the size of the slice as shown below:

source	destination	
 slice addressed via an array header in memory, array subscripts, and upper and lower slice index qualifiers 		
 slice addressed via an array base register and base register offset and size qualifiers 	 any conforming array slice, similarly addressed 	
 slice addressed via a pointer to an array header in the global storage of a package, array subscripts, and upper and lower slice index qualifiers 		
 slice addressed via a pointer to a data object of the type array, array subscripts, and upper and lower slice index qualifiers 		

Table 5.3

Size=(Upper Index - Lower Index +1) * Component Size

Exceptions: PROGRAM_ERROR CONSTRAINT_ERROR

5.1.4 LOAD ARRAY BASE ADDRESS

Format: $03_{\rm H}$, S, D

Inemonic: LDBA

Operands:

S: <u>Array Header</u> FMT: <u>memory (0)</u>

D: <u>Destination Array Base Register</u> array base register (EXT,11); register #1..14

Function:

The cell offset (CO) specified by S is added to the contents of each register of the display register pair corresponding to the nesting depth (ADS) specified by S to form the address of the array header in data template memory (DTM) and the corresponding The machine determines address in data value memory (DVM). whether the array has separate values (first word in header is AVO or DOD followed by AVA) or immediate values (first word in If the array has separate values the header is array bounds). address of the first component value in DVM and the address of the tag/initial value in DTM (for all components are found and loaded into the base registers specified by D and D+1. The tags written into these registers are F(AVA with UNDEFINED flag = 1)and CONT, respectively. If the array has immediate values, the address of the first component value in DVM and the address of the first component taq/initial value in DTM are found and loaded into the base registers specified by D and D+1. The tags written into these registers are E(AVA with UNDEFINED flag=0) and CONT, respectively. "AVA" provides a unique identification for array base registers.

Exceptions: PROGRAM ERROR 5.1.5 MOVE POINTER

Format: O4_H, S, D

Mnemonic: MOVPTR

Operands:

S: <u>Pointer to Be Moved</u> FMT: <u>memory (0)</u>

D: <u>Destination Pointer</u> FMT: <u>memory (0)</u>

Function:

The pointer operand addressed by S is copied to the destination location addressed by D. In every case when a pointer is referenced except this instruction, the pointed-to entity takes part in the operation.

Exceptions: PROGRAM_ERROR

1.6 SET UNDEFINE

ormat: 05_H, D

memonic: SETUND

perands:

): <u>Data to Be Set to Undefined</u> FMT: <u>memory (0)</u>

inction:

he data entity addressed by D is set to undefined by writing a l" into the "undefined bit" in the tag. For type V16, the ddressed 16-bit field is also set to $000_{\rm H}$ (see Section 3.1). If he data entity is a pointer, writing a "1" into the undefined it causes the pointer value to be NULL. If D addresses an array r a record, each component is set to undefined.

xceptions: None

5.1.7 PURGE STACK

Format: 06_H

Inemonic: PURGE

Operands: None

Function:

The local stack is pop'd (contents lost) until the stack pointer points to the top-of-stack.

Exceptions: None 8 SWAP STACK

at: 07_H

ionic: SWAP

ands:

le

:tion:

contents of the top two locations on the local stack are langed.

4

.

.

.

eptions:

5.1.9 CLEAR TEMPORARIES MASK

Format: 08_H

Mnemonic: CLRMSK

Operands: None

Function:

Each bit in the Temporaries Mask (bits 0..15 of register 0) is cleared.

. .

and the second sec

•

.

10

Exceptions: None

5.2 Arithmetic

The basic principle of operations involving numeric operands is that, except where otherwise specified, the result is computed as if correct to infinite precision. This infinite precision result is then rounded, if necessary, to the precision of the result operand. The arithmetic operations and floating-point formats are based on the proposed IEEE floating-point standard [IEEE 81], modified as described below (unless otherwise noted, section numbers refer to [IEEE 81]).

Formats (Para 3): The ISA does not support representations of infinity. The only not-a-number (NaN) which can be represented in "Undefined". The representations specified for infinity and other NaNs must not be used.

Default Rounding Mode (Para 4.1): Unless otherwise specified in the ISA, round-to-nearest is the default rounding mode for all operations.

Directed Rounding Modes (Para 4.2): The default rounding mode for any instruction can be overridden by preceding that instruction by one of the rounding instructions (Round Toward Zero, Round Toward Minus Infinity, Round Toward Plus Infinity, round to Nearest).

Rounding Precision (Para 4.3): Rounding is always to the precision of the result operand, regardless of whether that operand is integer or floating or whether the result precision is less than or greater than other operands.

Operations (Para 5): Operations are defined only for combinations of the same operand type. Exceptions are the CONVERT instructions.) Instructions may have different precision operands, however, (V16, V32, V64). Some operations required by the standard (e.g. round to integer) require software implementations. Comparison testing is by predicates (Para 5.6.2) rather than condition codes; it is not possible for the relation "unordered" to occur, given the modifications herein described, so no "unordered" predicate test is provided.

Infinity Arithmetic (Para 6.1) is not supported.

Operations with NaNs (Para 6.2) are not supported.

[IEEE 81] "A Proposed Standard for Binary Floating-Point Arithmetic", <u>Computer</u>, March 1981, pp. 51-62
Sign Bit (Para 6.3): The sign of zero in an integer type is considered to be "+". If -0 is required to be delivered as a result of integer type, the sign bit is ignored.

Normalizing Mode (Para 7.1) is the only mode supported; warning mode is not supported.

Exceptions (Para 8): The only exception which is raised when operands have valid numeric values (i.e., not "undefined") is NUMERIC ERROR. NUMERIC ERROR is raised for

- 1. DIVIDE, MODULUS, OR REMAINDER where the divisor is zero.
- 2. Square root of a negative number (square root of -0 results in -0 and does not raise an exception).
- 3. Overflow, i.e.; the rounded result's magnitude is too large to represent in the result format.

Underflow (Para 8.4) and inexact (Para 8.5) are not supported; the rounded result is always delivered (in order to directly support Ada arithmetic rules). In no case is any result (including infinity or NaN) provided when an exception is raised.

Traps (Para 9): Traps, as defined in the standard, are not provided. Exceptions raised are handled in the manner specified in Section 11 of the ISA. In particular, exception handling cannot be disable, no information (other than the exception type) is delivered to the exception handler, and the handler cannot return control to the point at which the exception occurred.

In the integer arithmetic instructions, the precision of 10-bit and a 20-bit immediate operands is taken to be V16 and V32, respectively.

[IEEE 81] "A Proposed Standard for Binary Floating-Point Arithmetic", <u>Computer</u>, March 1981, pp. 51-62 5.2.1 ADD INTEGER

Format: 09_H, S, D

Mnemonic: ADDI2

Operands:

S: <u>First Addend</u> FMT: <u>immediate</u> (EXT,2), memory (0), or stack (EXT,0)

D: <u>Second Addend and Sum</u> FMT: <u>memory (0) or stack EXT,0)</u>

Function:

The binary addition of the integers specified by S and D is performed. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The result (sum) is checked for overflow (magnitude of result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

The source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

5.2.1 (CONT) ADD INTEGER

Format: OAH, S1, S2, D

Operands:

S1: FMT:	First Addend immediate (EXT,2), memory (0), or stack (EXT,0)	
S2: FMT:	Second Addend immediate (EXT,2), memory (0), or stack (EXT,0)	
D: FMT:	Sum memory (0) or stack (EXT,0)	

Functions:

The binary addition of the integers specified by S1 and S2 is performed. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. the result (sum) is checked for overflow (magnitude of result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

Each source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

5.2.2 ADD FLOATING POINT

Format: OB_H, S, D

Mnemonic: ADDF2

Operands:

S: <u>First Addend</u> FMT: <u>memory (0) or stack (EXT,0)</u>

D: <u>Second Addend and Sum</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The binary floating point addition of the floating point numbers addressed by S and D is performed. Source and destination operands may have different precisions (V32 or V64). The precision of the operation is sufficient to accommodate the largest magnitude result (sum). The fractional part of the is rounded, if necessary, the precision of the result to destination fraction. The result is then checked for exponent overflow (magnitude larger than precision of destination exponent A NUMERIC ERROR exception is raised in the presence of allows). overflow, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

5.2.2 (CONT) ADD FLOATING POINT

Format: OCH, S1, S2, D

Mnemonic: ADDF3

Operands:

S1:	First Adde	nd			
FMT:	memory	τ (0)	or	stack	(EXT,0)

S2: <u>Second Addend</u> FMT: <u>memory (0)</u> or stack (EXT,0)

D: <u>Sum</u> FMT: memory (0) or stack (EXT, 0)

Function:

The binary floating point addition of the floating point numbers addressed by S1 and S2 is performed. Source and destination operands may have different precisions (V32 or V64). The precision of the operation is sufficient to accommodate the largest magnitude result (sum). The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (magnitude larger than precision of destination exponent allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array record.

5.2.3 SUBTRACT INTEGER

ODH, S, D Format:

Mnemonic SUBI2

Operands:

S:

Subtrahend

immediate (EXT,2), memory (0), or stack (EXT,0) FMT:

Minuend and Difference D: memory (0) or stack (EXT,0) FMT:

Function:

The binary subtraction of the integer specified by S from the integer addressed by D is performed. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The result (difference) is checked for overflow (magnitude of result larger than precision of destination allows). A NUMERIC_ERROR exception is raised in the presence of overflow, else the result value is stored in the destination location.

The source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

5.2.3 (CONT) SUBTRACT INTEGER

Format: OE_H, S1, S2, D

Mnemonic: SUBI3

Operands:

S1: <u>Minuend</u> FMT: imm

immediate (EXT,2,) memory (0), or stack (EXT,0)

S2: <u>Subtrahend</u> FMT: <u>immediate</u> (EXT,2), memory (0), or stack (EXT,0)

D: Difference

Function:

The binary subtraction of the integer specified by S2 from the integer specified by S1 is performed. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The result (difference) is checked for overflow (magnitude of result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result value is stored in the destination location.

Each source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type inter), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

2.4 SUBTRACT FLOATING POINT

rmat: OFH, S, D

emonic SUBF2

erands:

: <u>Subtrahend</u> FMT: <u>memory</u> (0) or stack (EXT.0)

FMT: Minuend and Difference memory (0) or stack (EXT,0)

nction:

binary floating point subtraction of the floating point mber addressed by S from the floating point number addressed by is performed. Source and destination operands may have fferent precisions (V32 or V64). The precision of the berations is sufficient to accommodate the largest magnitude sult (difference). The fractional part of the result is bunded, if necessary, to the precision of the destination raction. The result is then checked for exponent overflow lagnitude larger than precision of destination exponent allows). NUMERIC ERROR exception is raised in the presence of overflow, lse the result value is stored in the destination location.

ich operand may be a directly addressed floating point number, i indirectly addressed number in global storage or a data object type floating point), or a floating point component of an ray or record.

CCEPTIONS: PROGRAM_ERROR NUMERIC_ERROR 5.2.4 (CONT) SUBTRACT FLOATING POINT

Format: 10_H, S1,S2, D

Mnemonic SUBF3

Operands:

Sl: <u>Minuend</u> FMT: <u>memory</u> (0), or stack (EXT,0)

S2: <u>Subtrahend</u> FMT: <u>memory</u> (0) or stack (EXT,0)

D: <u>Difference</u> FMT: <u>memory</u> (0) or stack (EXT,0)

Function:

The binary floating point subtraction of the floating point number addressed by S2 from the floating point number addressed by S1 is performed. Source and destination operands may have different precisions (V32 or V64). The precision of the operation is sufficient to accommodate the largest magnitude result (difference). The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (magnitude larger than precision of destination exponent allows). A NUMERIC ERROR EXCEPTION is raised in the presence of overflow, else the result is stored in the destination location.

Each operand may be a directly addressed floating point OP number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

5 MULTIPLY INTEGER

at: $1l_H$, S, D

onic MULI2

ands:

Multiplier

T: immediate (EXT,2), memory (0), or stack (EXT,0)

T: <u>Multiplicand and Product</u> memory (0) or stack (EXT,0)

tion:

binary multiplication of the integers specified by S and D is ormed. Source and destination operands may have different isions (V16 or V32); the precision of the operation is V32. result (product) is checked for overflow (magnitude of result er than precision of destination allows). A NUMERIC_ERROR ption is raised in the presence of overflow, else the result tored in the destination location.

source operand may be an immediate value, a directly ressed integer, an indirectly addressed integer (via a pointer in integer in global storage or a data object of type rger), or an integer component of an array or record. The ination operand may be any of these except an immediate le.

Ptions: GRAM_ERROR MERIC_ERROR 2.5 (CONT) MULTIPLY INTEGER

rmat: $12_{\rm H}$, S1, S2, D

emonic: MULI3

erands:

: Multiplicand

FMT: immediate (EXT,2); memory (0), or stack (EXT,0)

2: <u>Multiplier</u> FMT: <u>immediate (EXt,2), memory (0) or stack (EXT,0)</u>

: <u>Product</u> FMT: <u>memory</u> (0) or stack (EXT,0)

nction:

le binary multiplication of the integers specified by Sl and S2 performed. Source and destination operands may have different 'ecisions (V16 or V32); the precision of the operation is V32. le result (product) is checked for overflow (magnitude of result irger than precision of destination allows). A NUMERIC ERROR ception is raised in the presence of overflow, else the result is stored in the destination location.

ich source operand may be an immediate value, a directly Idressed integer, an indirectly addressed integer (via a pointer) an integer in global storage or a data object of type iteger), or an integer component of an array or record. The istination operand may be any of these except an immediate ilue.

CCEPTIONS: PROGRAM_ERROR JUMERIC_ERROR

MULTIPLY FLOATING POINT

t: 13_H, S, D

nic SULF2

nds:

•

Multiplier memory (0), or stack (EXT,0)

Multiplicand and Product memory (0) or stack (EXT,0)

ion:

inary floating point multiplication of the floating point rs addressed by S and D is performed. Source and nation operands may have different precisions (V32 or V64). recision of the operation is sufficient to accommodate the st magnitude result (product). The fractional part of the t is rounded, if necessary, to the precision of the nation fraction. The result is then checked for exponent low (magnitude larger than precision of destination exponent s). A NUMERIC ERROR exception is raised in the presence of low, else the result is stored in the destination location.

operand may be a directly addressed floating point number, directly addressed number (via a pointer to a floating point ir in global storage or a data object of type floating), or a floating point component of an array or records.

TRAM_ERROR

5.2.6 (CONT) MULTIPLY FLOATING POINT

Format: 14_H, S1, S2, D

Mnemonic MULF3

Operands:

S:	Multiplicand			
FMT:	memory (0)	or	stack	(EXT,0)

S2: <u>Multiplier</u> FMT: <u>memory</u> (0) or stack (EXT,0)

D: <u>Product</u> FMT: <u>memory</u> (0) or stack (EXT,0)

Function:

The binary floating point multiplication of the floating point numbers addressed by S1 and S2 is performed. Source and destination operands may have different precisions (V32 or V64). The precisions of the operation is sufficient to accommodate the largest magnitude result (product). The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (magnitude larger than precision of destination exponent allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or to a data object of type floating point), or a floating point component of an array or record.

5.2.7 DIVIDE INTEGER

Format: 15_H, S, D

Mnemonic DIVI2

Operands:

S:

Division

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

D: <u>Dividend and Quotient</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The binary division of the integer addressed by D by the integer specified by S is performed. With integer arithmetic, the result (quotient) will be non-zero if the magnitude of the dividend is greater than the magnitude of the divisor. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The result is rounded, if necessary, using "round-toward-zero" as the default rounding rule (rather than the standard "round-to-nearest" rule). A NUMERIC ERROR exception is raised if the divisor is zero, else the result is stored in the destination location.

The source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

5.2.7 (CONT) DIVIDE INTEGER

Format: 16_H, S1, S2, D

Mnemonic DIVI3

Operands:

S: Dividend

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

S2: Divisor

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

D: <u>Quotient</u> FMT: <u>memory</u> (0) or stack (EXT,0)

Function:

The binary division of the integer specified by S1 by the integer specified by S2 is performed. With integer arithmetic, the result (quotient) will be non-zero if the magnitude of the dividend is greater than the magnitude of the divisor. Source. and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The result is rounded, if necessary, using "round toward zero" as the default rounding rule (rather than the standard "round-to-nearest" rule). The result is then checked for overflow (magnitude of result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow or if the divisor is zero, else the result is stored in the destination location.

Each source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

5.2.8 DIVIDE FLOATING POINT

Format: 17_H, S, D

Mnemonic DIVF2

Operands:

S: <u>Divisor</u>

FMT: memory (0) or stack (EXT,0)

D: <u>Dividend and Quotient</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The binary floating point division of the floating point number addressed by D by the floating point number addressed by S is performed. Source and destination operands may have different precisions (V32 or V64). The precision of the operation is sufficient to accommodate the largest magnitude result (quotient) representable in an intermediate format compatible with the accuracy rules specified in the 1981 IEEE proposed floating point arithmetic standard. The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (magnitude larger than precision of destination exponent allows). A NUMERIC ERROF exception is raised in the presence of overflow or if the divisor is zero, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly address number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or records.

5.2.8 (CONT) DIVIDE FLOATING POINT

Format:	18 _H , S1, S2, D
Mnemonic	DIVF3
Operands: Sl: FMT:	Dividend memory (0), or stack (EXT,0)
S2: FMT:	Divisor memory (0) or stack (EXT,0)
D •	Quotient

FMT: memory (0) or stack (EXT,0)

Function:

The binary floating point division of the floating point number addressed by S1 by the floating point number addressed by S2 is performed. Source and destination operands may have different precisions (V32 or V64). The precision of the operation is sufficient to accommodate the largest magnitude result (quotient) representable in an intermediate format compatible with the accuracy rules specified in the 1981 IEEE proposed floating point arithmetic standard.

The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (magnitude larger than precision of destination exponent allows). A NUMERIC_ERROR exception is raised in the presence of overflow or if the divisor is zero, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

i.2.9 REMAINDER INTEGER

Format: 19_H, S, D

Inemonic REMI2

)perands:

S: Divisor

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

D: <u>Dividend and Remainder</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The binary division of the integer addressed by D by the integer addressed by S is performed. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The quotient is rounded toward zero, leaving only the integer part of the quotient, called Q. The remainder, R, is computed as

R=Dividend - Q*Divisor

When R is non-zero, its sign is the same as the sign of the dividend. A NUMERIC ERROR exception is raised if the divisor is zero, else the result (remainder) is stored in the destination location.

The source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

5.2.9 (CONT) REMAINDER INTEGER

Format: 1A_H, S1, S2, D

Mnemonic REMI3

Operands:

S1: Dividend

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

- S2: <u>Divisor</u> FMT: <u>immediate (EXT,2), memory (0), or stack (EXT,0)</u>
- D: <u>Remainder</u> FMT: <u>memory</u> (0) or stack (EXT,0)

Function:

The binary division of the integer specified by Sl by the integer specified by S2 is performed. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The quotient is rounded toward zero (leaving the integer part of the quotient called Q). The remainder, R, is computed as

R=Dividend - Q*Divisor

When R is non-zero, its sign is the same as the sign of the dividend. The result (remainder) is then checked for overflow (magnitude of result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow or if the divisor is zero, else the result is stored in the destination lcoation.

Each source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be any of these except an immediate value.

2.10 REMAINDER FLOATING POINT

rmat: 1B_H, S, D

emonic REMF2

erands:

: <u>Divisor</u>

FMT: memory (0), or stack (EXT,0)

: <u>Dividend and Remainder</u> FMT: <u>memory (0) or stack (EXT,0)</u>

nction:

e binary floating point division of the floating point number dressed by D by the floating point number addressed by S is rformed. Source and destination operands may have different ecisions (V32 or V64). The precision of the operation is ifficient to accommodate the whole number (integer) part of the lotient plus the extra bits required for rounding. The quotient rounded to the nearest integer; the fractional part is scarded. If the integer part of the quotient is called q, the emainder, R, is computed as

R=Dividend - Q*Divisor

te remainder is rounded, if necessary, toward nearest (unless a receding rounding instruction specifies otherwise). A NUMERIC ROR exception is raised if the divisor is zero, else the result remainder) is stored in the destination location.

ich operand may be a directly addressed floating point number, i indirectly addressed number (via a pointer to a floating point umber in global storage or a data object of type floating pint), or a floating point component of an array or record.

CCEPTIONS: PROGRAM_ERROR JUMERIC_ERROR 5.2.10 (CONT) REMAINDER FLOATING POINT

Format: 1C_H, S1, S2, D

Inemonic REMF3

Dperands: S: <u>Dividend</u> FMT: <u>memory</u> (0) or stack (EXT,0)

S2: <u>Divisor</u> FMT: <u>memory (0) or stack (EXT,0)</u> D: Remainder

FMT: memory (0) or stack (EXT,0)

Function:

The binary floating point division of the floating point number addressed by S1 by the floating point number addressed by S2 is performed. Source and destination operands may have different precisions (V32 or V64). The precision of the operation is sufficient to accommodate the whole number (integer) part of the quotient plus the extra bits required for rounding. The quotient is rounded to the nearest integer; the fractional part is discarded. If the integer part of the quotient is called Q, the remainder, R, is computed as

R=Dividend - Q*Divisor.

The remainder is rounded, if necessary, toward nearest (unless a preceding rounding instruction specified otherwise). The result (remainder) is then checked for exponent overflow (magnitude of result larger than precision of destination exponent allows). A NUMERIC ERROR exception is raised in the presence of overflow or if the divisor is zero, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

11 MODULUS INTEGER

at: 1DH, S, D

nonic MODI2

ands:

Divisor

immediate (EXT,2), memory (0), or stack (EXT,0)

AT: <u>Dividend and Modulus</u> MT: <u>memory (0) or stack (EXT,0)</u>

ction:

binary division of the integer addressed by D by the integer ressed by S is performed. Source and destination operands may a different precisions (V16 or V32); the precision of the ration is V32. The quotient is rounded toward minus infinity, ving only the integer part of the quotient, called Q. The alus, M, is computed as

M=Dividend - Q*Divisor.

n M is non-zero, its sign is same as the sign of the divisor. UMERIC ERROR exception is raised if the divisor is zero, else result (modulus) is stored in the destination location.

source operand may be an immediate value, a directly ressed integer, an indirectly addressed integer (via a pointer an integer in global storage or a data object of type eger), or an integer component of an array or record. The tination operand may be any of these except an immediate ue.

a global souther of the second second second

eptions: OGRAM_ERROR MERIC_ERROR

2.11 (CONT) MODULUS INTEGER

rmat: 1E_H, S1, S2, D

emonic MODI3

il: <u>Dividend</u>

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

32: Divisor

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

D: <u>Modulus</u> FMT: <u>memory (0) or stack (EXT,0)</u>

unction

he binary division of the integer specified by Sl by the integer pecified by S2 is performed. Source and destination operands ay have different precisions (V16 and V32); the precision of the peration is V32. The quotient is rounded toward minus infinity, eaving only the integer part of the quotient, called Q. The odules, M is computed as

M=Dividend - Q*Divisor.

hen M is non-zero, its sign is the same as the sign of the ivisor. The result (modulus) is then checked for overflow magnitude of result larger than precision of destination llows). A NUMERIC ERROR exception is raised in the presence of verflow or if the divisor is zero, else the result is stored in he destination location.

ach source operand may be an immediate value, a directly ddressed integer, an indirectly addressed integer (via a pointer o an integer in global storage or a data object of type nteger), or an integer component of an array or record. The estination operand may be any of these except an immediate alue.

2 MODULUS FLOATING POINT

t: $1F_{H}$, S, D

nic: MODF2

nds:

: <u>Divisor</u> memory (0) or stack (EXT,0)

Dividend and Modulus memory (0) or stack (EXT,0)

ion:

binary floating point division of the floating point number essed by D by the floating point number addressed by S is brmed. Source and destination operands may have different sions (V32 or V64). The precision of the operation is leient to accommodate the whole number (integer) part of the lent plus the extra bits required for rounding. The quotient counded toward minus infinity; the fractional part is arded. If the integer part of the quotient is called Q, the lus M, is computed as

M=Dividend - Q*Divisor.

nodulus is rounded, if necessary, toward nearest (unless a eding rounding instruction specifies otherwise). A NUMERIC R exception is raised if the divisor is zero, else the result ulus) is stored in the destination location.

operand may be a directly addressed floating point number, ndirectly addressed number (via a pointer to a floating point er in global storage or a data object of type floating t), or a floating point component of an array or record.

ptions: GRAM_ERROR ERIC_ERROR .12 (CONT) MODULUS FLOATING POINT

mat: 20_H, S1, S2, D

monic MODF3

rands:

MT: <u>Dividend</u> MT: <u>memory</u> (0) or stack (EXT,0)

: <u>Divisor</u> MT: <u>memory</u> (0) or stack (EXT,0)

MT: <u>Modulus</u> MT: <u>memory</u> (0) or stack (EXT,0)

ction:

binary floating point division of the floating point number lressed by Sl by the floating point number addressed by S2 is formed. Source and destination operands may have different cisions (V32 or V64). The precision of the operation is fficient to accommodate the whole number (integer) part of the ptient plus the extra bits required for rounding. The quotient rounded toward minus infinity; the fractional part is scarded. If the integer part of the quotient is called Q, the lulus, M, is computed as

M=Dividend - Q*Divisor.

> modulus is rounded, if necessary, toward nearest (unless a >ceding rounding instruction specified otherwise). The result >dulus) is then checked for exponent overflow (magnitude of sult larger than precision of destination exponent allows). A MERIC_ERROR exception is raised in the presence of overflow or the divisor is zero, else the result is stored in the stination location.

ch operand may be a directly addressed floating point number, indirectly addressed number (via a pointer to a floating point nber in global storage or a data object of type floating int), or a floating point component of an array or record.

NEGATE INTEGER

21_H, D :

ic: NEGI1

ds:

Integer to Be Negated and Negated Integer memory (0) or stack (EXT,0)

on:

gative of the integer (V16 or V32) addressed by D is stored destination location.

erand may be a directly addressed integer, an indirectly sed integer (via a pointer to an integer in global storage ata object of type _ integer), or an integer component of an or record.

.

...

ions: AM ERROR ICERROR

UNCLA	SSIFIE	D AF	IAL-TR	-85-10	41-YOL	-2 F3	3615-7	9-C-19	35	F/G	9/2	NL	
	ш_,	And Andrewson an	ALL AND	EN-ZOSTEN	in Contraction Sector Contraction Biggint Constants Big	Constant Constant Constant Sectors Sec	in a second seco	Electronic and a second	BOAUSTICKU Marvistikus Marvistikus	BULLEUCH AND ASSOCIATION	E.E. Bitac Antoine Manacescree Ma		i i i i i i i i i i i i i i i i i i i
il I ^{nte} Tra Gagan werea Fr	11 10 10 10 10 10 10 10 10 10 10 10 10 1	L. Bruchenen	Baser competences	North States	ไปมีสุขจากแห								
									i Tanana Marana Marana Marana				
igitasacusuan immatri	a. Museessan generation		Britshill Description Loose Cores		<u> andra an</u>			DANS CONTRACTOR		And a second sec		e anteresta e ant	
							ngesiRetenen	in 177 Sector States Sector States Sector States	Spires.70000	EXCLUSION LEVEL DE LEVEL MARINE DE LEVEL MARIN			MERCARNA ESSENSE MARCHIN Frank
	BOOMENIA, BOOMEN		Binorys	- Beneditar - Beneditar - Beneditar - Beneditar - Beneditar	Electron and a second	New Street	10 10 10 10 10 10 10 10 10 10 10 10 10 1		ERV2894				



5.2.13 (CONT) NEGATE INTEGER

Format: 22_H, S, D

Mnemonic: NEGI2

Operands:

S: Integer to Be Negated FMT: memory (0) or stack (EXT,0)

D: <u>Negated Integer</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The negative of the integer addressed by S is taken. Source and destination operands may have different precisions (V16 or V32). The result is checked for overflow (magnitude or result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

Each operand may be a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR

5.2.14 NEGATE FLOATING POINT

Format: 23_H, D

Mnemonic: NEGF1

Operands:

1000

におからのはな。 「いっていた」「「いっていた」「「「いっていた」」ののグログでは、「「いった」」というので、「「いったい」」ので、「いったい」ので、「いったい」ので、「いった」、」、「いった」、」、「いった」、」、」、」、」、」、」、」、」、

D: Floating Point Number to Be Negated and Negated Number FMT: memory (0) or stack (EXT,0)

Function:

The negative of the floating point number (V32 or V64) addressed by D is stored in the destination location.

The operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

AND A REAL ADD. A DWD WHEN THE

5.2.14 (CONT) NEGATE FLOATING POINT

Format: 24_H, S, D

Mnemonic: NEGF2

Operands:

S:	Floating	Point	Number	to	Be	Negated	
FMT:	memor	(0) cv	or stad	ck	EX:	r.0)	

D: <u>Negated Number</u> FMT: <u>memory (0)</u> or stack (EXT,0)

Function:

The negative of the floating point number addressed by S is taken. Source and destination operands may have different precisions (V32 or V64). The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (magnitude of result larger than precision of destination exponent allows). A NUMERIC_ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

5.2.15 ABSOLUTE INTEGER

Format: 25_H, D

Mnemonic: ABSI1

Operands:

D: Integer and Absolute Value of Integer FMT: memory (0) or stack (EXT,0)

Function:

The absolute value of the integer (V16 or V32) addressed by D is stored in the destination location.

The operand may be a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record.

5.2.15 (CONT) ABSOLUTE INTEGER

Format: 26H, S, D

Mnemonic: ABSI2

Operands:

S: FMT:	Integer memory (0)	or	stack	(EXT,0)
D:	Absolute Value	of	Intege	er
FMT •	memory (0)	or	stack	$(\mathbf{E}\mathbf{X}\mathbf{T},0)$

Function:

The absolute value of the integer addressed by S is taken. Source and destination operands may may have different precision (V16 or V32). The result is checked for overflow (result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

Each operand may be a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR

5.2.16 ABSOLUTE FLOATING POINT

Format: 27_H, D

Mnemonic: ABSF1

Operands:

D: Floating Point Number and Absolute Value of Number FMT: memory (0) or stack (EXT,0)

Function:

The absolute value of the floating point number (V32 or V64) addressed by D is stored in the destination location.

The operand may be directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or records.

5.2.16 (CONT) ABSOLUTE FLOATING POINT

Format: 28_H, S, D

Mnemonic: ABSF2

Operands:

S:

Floating Point Number memory (0) or stack (EXT,0) FMT:

Absolute Value of Floating Point Number D: FMT: memory (0) or stack (EXT,0)

Function:

The absolute value of the floating point number addressed by S is taken. Source and destination operands may have different precisions (V32 or V64). The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (result larger than precision of destination exponent allows). A NUMERIC_ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

5.2.17 SQUARE ROOT INTEGER

Format: 29_H, D

Mnemonic: SQRTI1

Operands:

D: Integer and Square Root of Integer FMT: memory (0) or stack (EXT,0)

Function:

The square root of the integer (V16 or V32) addressed by D is taken. The result is rounded, if necessary, toward zero. A NUMERIC ERROR exception is raised if the integer addressed by D is negative, else the result is stored in the destination location.

The operand may be a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record.

5 12 10 Tab

tin besternette of the
5.2.17 (CONT) SQUARE ROOT INTEGER

Format: 2A_H, S, D

Mnemonic: SQRT12

Operands:

S: Integer FMT: memory (0) or stack (EXT,0)

D: Square Root of Integer FMT: memory (0) or stack (EXT,0)

Function:

The square root of the integer addressed by S is taken. Source and destination operands may have different precisions (V16 or V32); the precision of the operation is V32. The result is rounded, if necessary, toward zero. The result is then checked for overflow (result larger than precision of destination allows). A NUMERIC ERROR exception is raised in the presence of overflow or if the integer addressed by S is negative, else the result is stored in the destination location.

Each source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR

5-49

5.2.18 SQUARE ROOT FLOATING POINT

Format: 2B_H, D

Mnemonic: SQRTF1

Operands:

D: Floating Point Number and Square Root of Number FMT: memory (0) or stack (EXT,0)

Function:

The square root of the floating point number (V32 or V64) addressed by D is taken. The fractional part of the result is rounded, if necessary. A NUMERIC ERROR exception is raised if the floating point number addressed by D is negative, else the result is stored in the destination location.

The operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR 5.2.18 (CONT) SQUARE ROOT FLOATING POINT

Format: 2C_H, S, D

Mnemonic: SQRTF2

Operands:

S:	Floating Point	Number	
FMT:	memory (0)	or stack	(EXT,0

D: Square Root of Floating Point Number FMT: memory (0) or stack (EXT,0)

Function:

The square root of the floating point number addressed by S is taken. Source and destination operands may have different precisions (V32 or V64). The precision of the operation is sufficient to accommodate the largest result representable in an intermediate format compatible with the accuracy rules specified in the 1981 IEEE proposed floating point arithmetic standard. The fractional part of the result is rounded, if necessary, to the precision of the destination fraction. The result is then checked for exponent overflow (magnitude larger than precision of destination exponent allows). A NUMERIC ERROR exception is raised in the presence of overflow or if the floating point number addressed by S is negative, else the result is stored in the destination location.

Each operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR 5.2.19 ROUND TO NEAREST

Format: 2D_H,

Mnemonic: RNDN

Operands: None

Function:

The next instruction is executed with its rounding rule, for the result of the operation, if any, replaced by ROUND TO NEAREST. This instruction and the following instruction are executed as an inseparable couplet.

Exceptions: None 5.2.20 ROUND TO ZERO

Format: 2E_H

Mnemonic: RNDZ

Operands: None

Function:

The next instruction is executed with its rounding rule, if any, for the result of the operation replaced by ROUND TO ZERO. This instruction and the following instruction are executed as an inseparable couplet.

Exceptions: None

1.

5.2.21 ROUND TO PLUS INFINITY

Format: $2F_{\rm H}$

Mnemonic: RNDP

Operands: None

Function:

The next instruction is executed with its rounding rule, if any, for the result of the operation replaced by ROUND TO PLUS INFINITY. This instruction and the following instruction are executed as an inseparable couple.

Exceptions: None

5.2.22 ROUND TO MINUS INFINITY

Format: 30_H

Mnemonic: RNDM

Operands: None

Function:

The next instruction is executed with its rounding rule, if any, for the result of the operation replaced by ROUND TO MINUS INFINITY. This instruction and the following instruction are executed as an inseparable couplet.

It was the trained a way to share and the first

the second second second second

Exceptions: None 5.2.23 CONVERT INTEGER TO FLOATING POINT

Format: 31_H, S, D

Mnemonic: CONVIF

Operands:

S: Integer FMT: memory (0) or stack (EXT,0)

D: <u>Floating Point Number</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The type conversion of the integer (V16 or V32) addressed by S to the floating point number format (V16 or V64) addressed by D is performed. The fractional part of the result is rounded, if necessary, to the precision of the destination fraction and the result (floating point number) is stored in the destination location.

The source operand may be a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. The destination operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record.

5.2.24 CONVERT FLOATING POINT TO INTEGER

Format: 32_H, S, D Mnemonic: CONVFI

Integer

Operands:

S:	Floating Poi	nt	Nu	nber	
FMT:	memory (0)	or	stack	(EXT,0)

D:

FMT: memory (0) or stack (EXT,0)

Function:

The type conversion of the floating point number (V32 or V64) addressed by S to the integer format (V16 or V32) addressed by D is performed. The resulting integer is rounded to the nearest integer. Note that the result will be zero if the magnitude of the floating point number is less than 0.5 in value. The result is checked for overflow (magnitude of result larger than precision of destination integer allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the result is stored in the destination location.

The source operand may be a directly addressed floating point number, an indirectly addressed number (via a pointer to a floating point number in global storage or a data object of type floating point), or a floating point component of an array or record. The destination operand may be a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR

5.3 LOGICAL

The logical instructions fully support Ada by including all the well known logical operations on Booleans, mask data, and arrays and slices of Booleans and masks. Additional logical instructions set and clear Booleans, mask data, and arrays and slices of Booleans and masks.

5.3.1 AND

Format: 33_H, S, D

Mnemonic: AND2

Operands:

- S: FMT:
-

D:

Second Logical Operand and Result memory (0) or stack (EXT.0)

First Logical Operand

Function:

FMT:

The AND operation between the operands specified by S and D is performed. When corresponding bits of the operands are both 1 s, the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

immediate (EXT,2), memory (0) or stack (EXT,0)

The source operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Both operands must be Booleans (V16) or mask data of the same precision If the operand qualifier, BIT POSITION, is (V16, V32, or V64). present, only the selected bit position (same for both operands) takes part in the operation. All unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16-bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.1 (CONT) AND

Format: 34_{H} , S1, S2, D

Mnemonic: AND3

Operands:

S1: FMT:	First Logical Operand immediate (EXT,2),	memory	(0),	or	stack	(EXT,0)
S2: FMT:	Second Logical Operand immediate (EXT,2),	memory	(0),	or	stack	(EXT,0)

D: <u>Result</u> FMT: <u>memory</u> (0) or stack (EXT,0)

Function:

The AND operation between the operands specified by S1 and S2 is performed. When corresponding bits of the source operands are both 1 s, the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

Each source operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Source and destination operands must be Booleans (V16) or mask data of the same If the operand qualifier, BIT precision (V16, V32, or V64). POSITION, is present, only the selected bit position (same for each operand) takes part in the operation. all unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16-bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.2 AND ARRAY

Format: 35_H, S, D

Mnemonic: ANDA2

Operands:

S:	First Array of	Logicals	
FMT:	memory (0)	or base register	(EXT,11)

D: Second Array of Logicals and Result Array FMT: memory (0) or base register (EXT,11)

Function:

The AND operation between each pair of corresponding components of the arrays addressed by S and D is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction. Alternatively, the compact format, BI(EXT, 12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). For each pair of corrresponding array components, when corresponding bits are both 1s, the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.2 (CONT) AND ARRAY

Format: 36_H, S1, S2, D

Mnemonic: ANDA3

Operands:

S1:	First Array of Logicals	
FMT:	memory (0) or base register (EXT,11)	
S2:	Second Array of Logicals	
FMT:	memory (0) or base register (EXT,11)	

D: <u>Result Array</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The AND operation between each pair of corresponding components of the arrays addressed by S1 and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and all arrays must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then the machine computes the array size as the product of the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT, 11 or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required Alternatively, the compact format, in the instruction. BI(EXT,12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). For each pair of corresponding array components, when corresponding bits are both 1 s, the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.3 AND SLICE

Format: 37_H, S D

Mnemonic: ANDS2

Operands:

51:	First Array	of	Logicals	s and Resu	lt Array
FMT:	memory	(0)	or base	register	(EXT,11)

D: <u>Second Array of Logicals and Result Array</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The AND operation between each pair of corresponding components in slices of the arrays addressed by S and D is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimension of the arrays. The number of dimensions and slice lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXT, 11 or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice size. Alternatively, the compact format, BI(EXT, 12) or BM(EXT, 13), may be used with the single operand qualifier, ASIZ, as explained in Section 4.2.3 (page 4-8). When corresponding bits in the slices are both 1 s, the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination array slice location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.3 (CONT) AND SLICE

Format: 38_H, S1, S2, D

Mnemonic: ANDS3

Operands:

S1:	First Array	of	Log	gicals	5	
FMT:	memory	(0)	or	base	register	(EXT,11)

S2: <u>Second Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

D: <u>Result Array</u> FMT: <u>memory (0)</u> or base register (EXT,11)

Function:

The AND operation between each pair of corresponding components in slices of the arrays addressed by S1 and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and all arrays must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimension of the arrays. The number of dimension and slice lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXt, 11 or FMT=0 and the cell offset designates a base register - with an Ava Tag), the operand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice Alternatively, the compact format BI(EXT,12) or BM size. (EXT, 13), may be used with the single operand qualifier, ASIZ, as explained in Section 4.2.3 (page 4-8). When corresponding bits in the slices are both 1 s, the result bit is set to 1), else the result bit is set to 0. The result is stored in the destination array slice location.

When addressed through a header, each array of logicals operands may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.4 OR

39_H, S, D Format:

Mnemonic: OR2

Operands:

S1:

First Logical Operand immediate (EXT,2), memory, (0) or stack (EXT,0) FMT:

D: Second Logical Operand and Result FMT: memory (0) or stack (EXT,0)

Function:

The inclusive OR operation between the operands specified by S and D is performed. When either (both) of a pair of corresponding bits of the operand is 1 (are 1 s), the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

The source operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Both operands must be Booleans (V16) or mask data of the same precision (V16, V32, or V64). If the operand qualifier, BIT POSITION, is present, only the selected bit position (same for both operands) takes part in the operation. All unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16 bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.4 (CONT) OR

Format: 3A_H, S1, S2, D

Mnemonic: OR3

Operands:

- S1: <u>First Logical Operand</u> FMT: <u>immediate (EXT,2)</u>, memory, (0) or stack (EXT,0)
- S2: <u>Second Logical Operand</u> FMT: <u>immediate (EXT,2)</u>, memory (0) or stack (EXT,0)

D: <u>Result</u> memory (0) or stack (EXT,0)

Function:

The inclusive OR operation between the operands specified by Sl and S2 is performed. When either (both) of a pair of corresponding bits of the source operands is 1 (are 1 s), the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

Each sourco operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Source and destination operands must be Booleans (V16) or mask data of the same precision (V16, V32, or V64). If the operand qualifier, BIT POSITION, is present, only the selected bit position (same for each operand) takes part in the operation. All unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16 bit operation). The machine cannot differentiate Booleans from masks since both have Differentiation occurs in the use of the result., V16 tags. e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.5 OR ARRAY

3B_{H.} S, D Format:

Mnemonic: ORA2

Operands:

S:

First Array of Logicals memory (0) or base register (EXT,11) FMT:

Second Array of Logicals and Result Array **D**: memory (0) or base register (EXT,11) FMT:

Function:

The inclusive OR operation between each pair of corresponding components of the arrays addressed by S and D is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT,11) or FMT=0 and - with an AVA tag), the cell offset designates a base register the operand qualifier, ARRAY SIZE (ASIZ), is required in the Alternatively, the compact format, BI (EXT,12) or instruction. BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). corresponding For each pair of array components, when corresponding bits are both 1 s, or when either of the bits is 1, the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.5 (CONT) OR ARRAY

Format: 3C_H, S1, S2, D

Mnemonic: ORA3

Operands:

S1:	First Array of Logicals
FMT:	memory (0) or base register (EXT,11)
S2:	Second Array of Logicals
FMT:	memory (0) or base register (EXT,11)
D.	Pecult Array

FMT: memory (0) or base register (EXT,11)

Function:

The inclusive OR operation between each pair of corresponding components of the arrays addressed by S1 and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT,11) or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the Alternatively, the compact format, BI (EXT,12) or instruction. BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). corresponding each pair of array components, For when corresponding bits are both 1 s, or when either bits is 1, the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.6 OR SLICE

Format: 3_{DH}, S, D

Mnemonic: ORS2

Operands:

S: FMT:

: First Array of Logicals memory (0) or base register (EXT,11)

D: <u>Second Array of Logicals and Result Array</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

アンビビジン 内容的ない たたい 日本語 いたので ことに 法意義 マン・インド

The inclusive OR operation between each pair of corresponding components of the arrays addressed by S and D is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimension of the arrays. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice size. Alternatively, the compact format, BI (EXT, 12) or BM (EXT, 13), may be used with the single operand qualifier, ASIZ, as explained in section 4.2.3 (page 4-8). When corresponding bits in the slices are both 1 s, or when either bit is a 1, the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination array slice location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.6 (CONT) OR SLICE

Format: $3E_{H}$, S1, S2, D

Mnemonic: ORS3

Operands:

S1: FMT:	First Array of Logicals memory (0) or base register (EXT,11)	
S2: FMT:	Second Array of Logicals memory (0) or base register (EXT,11)	
D: FMT:	Result Array memory (0) or base register (EXT,11)	

Function:

The AND operation between each pair of corresponding components in slices of the arrays addressed by S1 and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimension of the arrays. The number of dimensions and slice lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXt, 11 or FMT=0 and the - with an AVA tag), the cell offset designates a base register operand qualifiers, BASE RELATIVE OFFSET (BRO) AND array size (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice size. Alternatively, the compact format, BI (EXT, 12) or BM (EXT, 13), may be used with the single operand qualifier, ASIZ, as explained in section 4.2.3 (page 4-8). When corresponding bits in the slices are both 1 s, or when either bit is a 1, the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination array slice location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.7 EXCLUSIVE OR

Format: 3F_H, S, D

Mnemonic: EXOR2

Operands:

S:

FMT:

First Logical Operand immediate (EXT,2), memory (0) or stack (EXT,0)

D: <u>Second Logical Operand and Result</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The EXCLUSIVE OR operation between the operands specified by S and D is performed. When corresponding bits of the operands are complements of one another (1 and 0 or 0 and 1), the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

The source operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Both operands must be Booleans (V16) or mask data of the same precision (V16, V32, or V64). If the operand qualifier, BIT POSITION, is present, only the selected bit position (same for both operands) takes part in the operation. All unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16 bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.7 (CONT) EXCLUSIVE OR

Format: 40_H, S1,S2, D

Mnemonic: EXOR3

Operands:

S1: FMT:	First Logical Operand immediate (EXT,2),	memory	(0)	or	stack	(EXT,0)
S2: FMT:	Second Logical Operand immediate (EXT,2),	memory	(0)	or	stack	(EXT,0)

D:

FMT: Result

memory (0) or stack (EXT,0)

Function:

The EXCLUSIVE OR operation between the operands specified by S1 and S2 is performed. When corresponding bits of the operands are complements of one another (1 and 0 or 0 and 1), the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

Each source operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Source and Destination operands must be Booleans (V16) or mask data of the same precision (V16, V32, or V64). If the operand qualifier, BIT POSITION, is present, only the selected bit position (same for both operands) takes part in the operation. All unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16 bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

EXCLUSIVE OR ARRAY 5.3.8

Format: 41_H, S, D

Mnemonic: EXORA2

Operands:

5	:		
	-	-	

First Array of Logicals memory (0) or base register (EXT,11) FMT:

Second Array of Logicals and Result Array D: FMT: memory (0) or base register (EXT,11)

Function:

The EXCLUSIVE OR operation between each pair of corresponding components of the arrays addressed by S and D is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction. Alternatively, the compact format, BI(EXT, 12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). corrresponding array components, when For each pair of corresponding bits are complements of one another (0 and 1 or 1 and 0), the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.8 (CONT) EXCLUSIVE OR ARRAY

Format: $42_{\rm H}$, S1, S2, D

Mnemonic: EXORA3

Operands:

Sl:	First Array	of	Logicals	
FMT:	memory	(0)	or base register	(EXT,11)

S2: <u>Second Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

D: <u>Result Array</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The EXCLUSIVE OR operation between each pair of corresponding components of the arrays addressed by S1 and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the Alternatively, the compact format, BI(EXT, 12) or instruction. BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). corrresponding array components, when For each pair of corresponding bits are complements of one another (0 and 1 or 1 and 0), the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.9 EXCLUSIVE OR SLICE

Format: 4_{3H}, S, D

Mnemonic: EXORS2

Operands:

S:	First Array of Logicals
FMT:	memory (0) or base register (EXT,11)
D:	Second Array of Logicals and Result Array
FMT:	memory (0) or base register (EXT,11)

Function:

The EXCLUSIVE OR operation between each pair of corresponding components in slices of the arrays addressed by S and D is The arrays must have Boolean components or mask performed. components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimensions of the arrays. The number of dimensions and slice lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ) are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice size. Alternatively, the compact format, BI(EXT, 12) or BM(EXT, 13), may be used with the single operand qualifier, ASIZ as explained in When corresponding bits in the slices section 4.2.3 (page 4-8). are complements of one another (0 and 1 or 1 and 0), the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination array slice location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.9 (CONT) EXCLUSIVE OR SLICE

Format: 44_H, S1, S2, D

Mnemonic: EXORS3

Operands:

S1: 1	First Array	ot	Loc	licals	5	
FMT:	memory	(0)	or	base	register	(EXT,11)

S2: <u>Second Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

D: <u>Result Array</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The EXCLUSIVE OR operation between each pair of corresponding components in slices of the arrays addressed by Sl and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimensions of the arrays. The number of dimensions and slice lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ) are required in the instruction. BRO gives the offset from the array base address register) to the start of the slice an (contained in the base and ASIZ gives the slice the size. Alternatively, compact format, BI(EXT, 12) or BM(EXT,13), may be used with the single operand qualifier, ASIZ as explained in Section 4.2.3 (page 4-8). When corresponding bits in the slices are complements of one another (0 and 1 or 1 and 0), the result bit is set to 1, else the result bit is set to The result is stored in the destination array slice location. 0.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via pointer to an array in global storage or a data object of type array), or a component of a record. Exceptions: PROGRAM ERROR

5.3.10 EQUIVALENCE

Format: 45_H, S, D

Mnemonic: EQ2

Operands:

S: FMT:

First Logical Operand immediate (EXT,2), memory (0), or stack (EXT,0)

D: <u>Second Logical Operand and Result</u> FMT: <u>memory (0) or stack (EXT,0)</u>

Function:

The EQUIVALENCE operation between the operands specified by S and D is performed. When corresponding bits of the operands are equal (both 0 or both 1), the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

The source operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Both operands must be Booleans (V16) or mask data of the same precision (V16, V32, or V64). If the operand qualifier, BIT POSITION, is present, only the selected bit position (same for both operands) takes part in the operation. All unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.10 (CONT) EQUIVALENCE

Format: 46_H, S1, S2, D

Mnemonic: EQ3

Operands:

S1: <u>First Logical Operand</u> FMT: <u>immediate (EXT,2)</u>, memory (0), or stack (EXT,0)

S2: <u>Second Logical Operand</u> FMT: <u>immediate (EXT,2), memory (0), or stack (EXT,0)</u>

D: Result

Function:

The EQUIVALENCE operation between the operands specified by Sl and S2 is performed. When corresponding bits of the operands are equal (both 0 or both 1), the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination location.

Each source operand may be an immediate value, interpreted as a Boolean (V16) or a mask (V16) - depending on its use, a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. The destination operand may be any of these except an immediate value. Both operands must be Booleans (V16) or mask data of the same precision (V16, V32, or V64). If the operand qualifier, BIT POSITION, is present, only the selected bit position (same for both operands) takes part in the operation. All unselected bits of the destination operand are unchanged. Note that the operation performed on Booleans is exactly the same as the operation performed on V16 masks (a 16bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.11 EQUIVALENCE ARRAY

Format: 47_H, S, D

Mnemonic: EQA2

Operands:

5:	First Array of	f_Logicals	
FMT:	memory (0) or base register	(EXT, 11)

D: <u>Second Array of Logicals and Result Array</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The EQUIVALENCE operation between each pair of corresponding components of the arrays addressed by S1 and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the Alternatively, the compact format, BI(EXT,12) or instruction. BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). For each pair of corrresponding array components, when corresponding bits are equal (both 0 or both 1) the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.11 (CONT) EQUIVALENCE ARRAY

Format: 48_H, S1, S2, D

Mnemonic: EQA3

Operands:

S1: FMT:	First Array of Logicals memory (0) or base register (EXT,11)	
S2: FMT:	Second Array of Logicals memory (0) or base register (EXT,11)	

D: <u>Result Array</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The EQUIVALENCE operation between each pair of corresponding components of the arrays addressed by Sl and S2 is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). of corrresponding array components, when For each pair corresponding bits are equal (both 0 or both 1) the result bit is set to 1, else the result bit is set to 0. The components of the result array are stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.12 EOUIVALENCE SLICE

Format: 49_H, S, D

Mnemonic: EQS2

Operands:

S:

First Array of Logicals memory (0) or base register (EXT,11) FMT:

D: Second Array of Logicals and Result Array memory (0) or base register (EXT,11) FMT:

Function:

The EQUIVALENCE operation between each pair of corresponding components of the arrays addressed by S and D is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimension of the arrays. The number of dimensions and slice lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice size. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used with the single operand qualifier ASIZ, as explained in Section 4.2.3 (page 4-8). When corrresponding bits in the slices are equal (both 0 or both 1) the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.12 (CONT) EQUIVALENCE SLICE

Format: 4A_H, S1, S2, D

Mnemonic: EQS3

Operands:

S:	First Array of Logicals
FMT:	memory (0) or base register (EXT,11)
S2:	Second Array of Logicals
FMT:	memory (0) or base register (EXT,11)
ס:	Result Array
FMT:	memory (0) or base register (EXT,11)

Function:

The EQUIVALENCE operation between each pair of corresponding components in slices of the arrays addressed by S and D is performed. The arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimension of the arrays. The number of dimensions and slice. lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case; ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice compact Alternatively, the format, BI(EXT, 12)size. or BM(EXT,13), may be used with the single operand qualifier ASIZ, as explained in Section 4.2.3 (page 4-8). When corrresponding bits in the slices are equal (both 0 or both 1) the result bit is set to 1, else the result bit is set to 0. The result is stored in the destination array location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.13 NOT

Format: 4B_H, D

Mnemonic: NOT1

Operands:

D: Logical Operand and Result FMT: memory (0) or stack (EXT,0)

Function:

The NOT operation is performed on the operand addressed by D. The bit (or bits) in the operand is (are) complemented (0 to 1 or 1 to 0) and the result is stored in the destination location.

The operand may be a directly addressed Boolean (V16) or mask (V16, V32, or V64) an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. If the operand qualifier, BIT POSITION, is present, only the selected bit position is affected. All unselected bits of the operand are unchanged. Note that the operation performed on a Boolean is exactly the same as the operation performed on a V16 mask (a 16 bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.13 (CONT) NOT

Format: 4C_H, S, D

Mnemonic: NOT2

Operands:

S: <u>Logical Operand</u> FMT: memory (0) or stack (EXT,0)

D: <u>Result</u> FMT: <u>memory</u> (0) or stack (EXT,0)

Function:

The NOT operation is performed on the operand addressed by S. The bit (or bits) in the operand is (are) complemented (0 to 1 or 1 to 0) and the result is stored in the destination location.

Each operand may be a directly addressed Boolean or mask, an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. Source and destination operands must both be Booleans (V16) or mask data of the same precision (V16, V32, or V64). If the operand qualifier, BIT POSITION, is present, only the selected bit position is affected. All unselected bits of the destination operand are unchanged. Note that the operation performed on a Boolean is exactly the same as the operation performed on a V16 mask (a 16 bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.
5.3.14 NOT ARRAY

Format: 4D_H, D

Mnemonic: NOTAl

Operands:

D: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The NOT operation is performed on the components of the array addressed by D. The array must have Boolean components or mask components. When the array is addressed through its header (FMT=0), the machine computes the array size as the product of the outermost (highest dimensioned)length and SPAN (length and component size if the number of dimensions is 1). When the array is addressed through a base register (FMT=EXT, 11 or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ) is required in the instruction. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). Each bit in each component is complemented (0 to 1 or 1 to 0) and the result array is stored in the destination array location.

When addressed through a header, the array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.14 (CONT) NOT ARRAY

Format: 4E_H, S, D

Mnemonic: NOTA2

Operands:

S: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

D: <u>Result Array</u> memory (0) or base register (EXT,11)

Function:

The NOT operation is performed on the components of the array addressed by S. Source and destination arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions and equal lengths for corresponding dimensions. The number of dimensions and lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). Then, the machine computes the array size as the product of the the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When an array is addressed through a base register (FMT=EXT, \cdot 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). Each bit in each component of the source array is complemented (0 to 1 or 1 to 0) and the result is stored in the destination array location. precision

When addressed through a header, the array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.15 NOT SLICE

Format: 4FH, D

Mnemonic: NOTS1

Operands:

D: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The NOT operation is performed on the components in a slice of the array addressed by D. the array must have boolean components or mask components. When the array is addressed through its header, ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction. The machine computes the address of the beginning of the slice and the slice size. When the array is addressed through a base register (FMT=EXT,11 or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice size. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used with the single operand qualifier, ASIZ, as explained in Section 4.2.3 (page 4-8). Each bit in each component of the slice is complemented (0 to 1 or 1 to 0) and the result is stored in the destination array slice location.

When addressed through a header, the array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.15 (CONT) NOT ARRAY

Format: 50_H, S, D

Mnemonic: NOTS2

Operands:

S: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

D: Result Array

FMT: memory (0) or base register (EXT,11) Function:

The NOT operation is performed on the components in a slice of the array addressed by S. Source and destination arrays must have Boolean components or mask components of the same precision (V16, V32, or V64) and must have the same number of dimensions; the slice lengths must be the same but the slices need not be in the same dimension of the arrays. The number of dimensions and slice lengths are checked by the machine only when the arrays are addressed through their headers (FMT=0). In this case, ARRAY SUBSCRIPT (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction for each array operand. The machine computes the address of the beginning of each slice and the slice size. When an array is addressed through a base register (FMT=EXT, 11 or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice and ASIZ gives the slice size. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used with the single operand qualifier, ASIZ, as explained in Section 4.2.3 (page 4-Each bit in each component of the source array slice is 8). complemented (0 to 1 or 1 to)) and the result is stored in the destination array slice location.

When addressed through a header, each array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.16 SET

Format: 51_H, D

Mnemonic: SET

Operands:

D: Logical Operand and Result FMT: memory (0) or stack (EXT,11)

Function:

The SET operation is performed on the operand addressed by D. The bit (bits) in the operand is (are) set to 1 and the result is stored in the destination location

The operand may be a directly addressed Boolean (V16) or mask (V16, V32, or V64), an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record. If the operand qualifier, BIT POSITION, is present, only the selected bit position is affected. A11 unselected bits of the operand are unchanged. Note that the operation performed on a Boolean is exactly the same as the operation performed on a V16 mask (a 16 bit operation). The machine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g, the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.17 SET ARRAY

Format: 52_H, D

Mnemonic: SETA

Operands:

D: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The SET operation is performed on the components of the array addressed by D. The array must have Boolean components or mask components. When the array is addressed through its headers (FMT=0) the machine computes the array size as the product of the the outermost (highest dimensioned) length and SPAN (length and component size if the number of dimensions is 1). When the array is addressed through a base register (FMT=EXT, 11 or FMT = 0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). Each bit in each component is set to 1 and the result array is stored in the destination array location.

When addressed through a header, the array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

5.3.18 SET.SLICE

Format: 53_H D

Mnemonic: SETS

Operands:

D: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The SET operation is performed on the components in a slice of the arrays addressed by The arrays must have Boolean D. components or mask components. When the array is addressed through its header, ARRAY SUBSCRIPT, (if required) and upper and lower ARRAY SLICE INDEX operand qualifiers are present in the instruction. The machine computes the address of the beginning of each slice and the slice size. When the array is addressed through a base register (FMT=EXT, 11 or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE (ASIZ), are required in the instruction. BRO gives the offset from the array base address (contained in the base register) to the start of the slice. Alternatively, the compact format, BI(EXT, 12) or BM(EXT, 13), may be used with the single operand qualifier, ASIZ, as explained in Section 4.2.3 (page 4-8). Each bit in each component of the slice is set to 1 and the result is stored in the destination slice location.

When addressed through a header, the array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record.

i.3.19 CLEAR

'ormat: 54_H, D

inemonic: CLR

)perands:

D: Logical Operand and Results FMT: memory (0) or stack (EXT,11)

function:

The CLEAR operation is performed on the operand addressed by D. The bits (bits) in the operand is (are) reset to 0 and the result is stored in the destination location.

The operand may be a directly addressed Boolean (V16) or mask (V16, V32, or V64) an indirectly addressed Boolean or mask (via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an If the operand qualifier, BIT POSITION, is array or record. present, only the selected bit position is affected. A11 unselected bits of the operand are unchanged. Note that the operation performed on a Boolean is exactly the same as the operation performed on a V16 mask (a 16 bit operation). The nachine cannot differentiate Booleans from masks since both have V16 tags. Differentiation occurs in the use of the result, e.g., the IF instruction tests a Boolean but when the operand qualifier, BIT POSITION, is present, it tests the selected bit in a 16-bit mask.

5.3.20 CLEAR ARRAY

Format: 55_H D

Mnemonic: CLRA

Operands:

D: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

Function:

The CLEAR operation is performed on the components of the array addressed by D. The array must have Boolean components or mask components. When the array is addressed through its header (FMT=0), the machine computes the array size as the product of the outermost (highest dimensioned)length and SPAN (length and component size if the number of dimensions is 1). When the array is addressed through a base register (FMT=EXT, 11 or FMT=0 and the cell offset designates a base register - with an AVA tag), the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction. Alternatively, the compact format, BI(EXT,12) or BM(EXT,13), may be used as explained in Section 4.2.3 (page 4-8). Each bit in each component is reset to 0 and the result array is stored in the destination array location.

When addressed through a header, the array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record. Exceptions:

PROGRAM ERROR

.3.21 CLEAR SLICE

ormat: 56_H D

nemonic: CLRS

perands:

D: <u>Array of Logicals</u> FMT: <u>memory (0) or base register (EXT,11)</u>

unction:

he CLEAR operation is performed on the components in a slice of he array of the array addressed by D. The array must have loolean components or mask components. When the array is ddressed through its header ARRAY SUBSCRIPT (if required) and pper and lower ARRAY SLICE INDEX operand qualifiers are present The machine computes the address of the n the instruction. eqinning of the slice and the slice size. When the array is ddressed through a base register (FMT=EXT, 11 or FMT=0 and the ell offset designates a base register - with an AVA tag), the perand qualifiers, BASE RELATIVE OFFSET (BRO) and ARRAY SIZE ASIZ), are required in the instruction. BRO gives the offset (contained in the base register) to rom the array base address he start of the slice. Alternatively, the compact format, SI(EXT, 12) or BM(EXT, 13), may be used with the single operand Jualifier, ASIZ, as explained in Section 4.2.3 (page 4-8). Each bit in each component of the slice is set to 0 and the result is stored in the destination array location.

When addressed through a header, the array of logicals operand may be directly addressed, indirectly addressed (via a pointer to an array in global storage or a data object of type array), or a component of a record. Exceptions PROGRAM ERROR

5.4 Branch

The branch instructions support the Ada IF statement (if Booleanexpression then, else), the CASE statement, the LOOP statement (for-loop), and the GOTO statement. Note that Ada WHILE LOOP statements are supported by the HLLM IF and GOTO instructions as shown below:

> LABEL A: IF (RELATIONAL), LABEL B SEQUENCE OF INSTRUCTIONS GOTO LABEL A

LABEL B:

5.4.1 IF

Format: 57_H S, D

Mnemonic: IF

Operands:

S:

FMT: memory (0) or stack (EXT,0)

Logical Operand

D: Label

FMT: immediate (EXT,2), interpreted as a label operand

Function:

The state of the operand addressed by S is tested; if 1 (true), no action is taken but if 0 (false), a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). The source operand must be a Boolean (V16) or mask data (V16, V32, or V64). If mask data, the operand qualifier, BIT POSITION, must be present in the instruction to select the bit of the mask to be tested.

The source operand may be a directly addressed Boolean or a mask, an indirectly addressed Boolean or mask via a pointer to a Boolean or mask in global storage or a data object of type Boolean or mask), or a Boolean or mask component of an array or record

5.4.2 IF EQUAL

Format: 58_H S1, S2, D

Mnemonic: IF=

Operands:

S1: FMT: <u>Comparand 1</u> immediate (EXT,2), memory (0), stack (EXT,0), or base register (EXT,11)

S2: Comparand 2

Label

FMT: immediate (EXT,2), memory (0), stack (EXT,0), or base register (EXT,11)

D: FMT:

immediate (EXT, 2), interpreted as a label operand

Function:

The operands specified by Sl and S2 are compared for equality. If the values are equal, no action is taken but if the values are not equal, a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). The source operands may be numeric or logical (V16, V32, or V64), pointers (PTR), components of arrays or records, or whole arrays or records. Immediate source operands are interpreted as having a V32 tag with sign extend. Equal pointers are both undefined (null) or both defined with identical absolute address and unique names (checked if unique name flags = 1). Equal arrays have the same number of dimensions, the same lengths for corresponding dimensions, and equal corresponding components. Equal records have equal corresponding components.

Note that in this instruction (and in the IF NOT EQUAL instruction), when S1 and S2 address pointers, the pointers, not the pointed to data entities, are compared for equality. Also note, as usual, when arrays are addressed via base registers, the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction; alternatively, the compact format, BI(EXT,12) or BM (EXT,13), may be used as explained in Section 4.2.3 (page 4-8).

EXCEPTIONS: PROGRAM ERROR 5.4.3 IF NOT EQUAL

Format: 59_H S1, S2, D

Mnemonic: IF <>

Operands:

- S1: <u>Comparand 1</u> FMT: immediate (EXT,2), memory (0), stack (EXT,0), or base register (EXT,11)
- S2: <u>Comparand 2</u> FMT: <u>immediate (EXT,2), memory (0), stack (EXT,0),</u> or base register (EXT,11)

D: Label

FMT: immediate (EXT,2), interpreted as a label operand

Function:

The operands specified by S1 and S2 are compared for inequality. If the values are not equal, no action is taken but if the values are equal, a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). The source operands may be numeric or logical (V16, V32, or V64), pointers (PTR), components of arrays or records, or whole arrays or records. Immediate source operands are interpreted as having a V32 tag with sign extend. Source operands are not equal if they do not meet the equality definitions specified in the IF EQUAL instruction.

Note that in this instruction (and in the IF EQUAL instruction), when S1 and S2 address pointers, the pointers, not the pointed-to data entities, are compared for equality. Also note, as usual, when arrays are addressed via base registers, the operand qualifier, ARRAY SIZE (ASIZ), is required in the instruction; alternatively, the compact format, BI(EXT,12) or BM (EXT,13), may be used as explained in Section 4.2.3 (page 4-8).

EXCEPTIONS: PROGRAM ERROR

5.4.4 IF LESS THAN INTEGER

Format: 5AH S1, S2, D

Mnemonic: IFI<

Operands:

- S1: <u>Comparand 1</u> FMT: immediate (EXT,2), memory (0), stack (EXT,0), or base register (EXT,11)
- S2: <u>Comparand 2</u> FMT: immediate (EXT,2), memory (0), stack (EXT,0), or base register (EXT,11)

D: Label

FMT: immediate (EXT,2), interpreted as a label operand

Function:

The operands specified by Sl and S2 are compared; if the former value (Sl) is less than the latter value (S2) no action is taken, else a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). Both source operands must be integers (V16 or V32) or single dimension arrays of integers. If arrays, lower bounds and lengths need not match. The value of the array operand addressed by Sl is less than the value of the array operand addressed by S2 if the former value lexicographically precedes the latter value, using the collating sequence of the component type. Formally stated, let k be the largest integer such that

k<=length of Sl array operand

and

k<=length of S2 array operand

and let the first k components of the Sl and S2 array operands be equal (k>=0). Then array operand Sl is less than array operand S2 if and only if k is less than the length of array operand S2 and either

- 1. k=length of array operand S1, or
- 2. k=length of array operand Sl and the k+lst component of array operand Sl is less than the k+lst component of array operand S2.

Note that a source operand may be an integer component of an array or record. As usual, when an array component is addressed via a base register, the operand qualifier, BASE RELATIVE OFFSET (BRO) is required in the instruction to locate the component. When a whole array of integers is addressed via a base register, the operand qualifier, ARRAY SIZE (ASIZ) is required in the instruction. In both cases, the compact format, BI(EXT,12) OR BM(EXT,13) may be used as explained in Section 4.2.3 9 (page 4-8). Note, also, that immediate source operands are interpreted as having a V32 tag with sign extend.

5.4.5 IF LESS THAN FLOATING POINT

Format: 5BH S1, S2, D

Mnemonic: IFF<

Operands:

S1: <u>Comparand 1</u> FMT: <u>memory (0)</u> or stack (EXT,0)

S2: <u>Comparand 2</u> FMT: <u>memory (0) or stack (EXT,0)</u>

D: <u>Label</u> FMT: <u>immediate (EXT,2)</u>, interpreted as a label operand

Function:

The operands specified by S1 and S2 are compared; if the former value (S1) is less than the latter value (S2), no action is taken, else a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). Both source operands must be floating point numbers (V32 or V64).

5.4.6 IF GREATER THAN INTEGER

Format: 5CH S1, S2, D

Mnemonic: IFI<

Operands:

S1: <u>Comparand</u> FMT: immediate (EXT,2), memory (0), stack (EXT,0), or base register (EXT,11)

S2: <u>Comparand 2</u> FMT: immediate (EXT,2), memory (0), stack EXT,0), or base register (EXT,11)

D: Label FMT: immediate (EXT,2), interpreted as a label operand

Function:

The operands specified by Sl and S2 are compared; if the former value (Sl) is greater than the latter value (S2) no action is taken, else a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). Both source operands must be integers (V16 or V32) or single dimension arrays of integers. If arrays, lower bounds and lengths need not match. The value of the array operand addressed by Sl is less than the value of the array operand addressed by S2 if the latter value is less than the former value as formally defined in the IF LESS THAN INTEGER instruction (Section 5.4.4.).

Note that a source operand may be an integer component of an array or record. As usual, when an array component is addressed via a base register, the operand qualifier, BASE RELATIVE OFFSET (BRO) is required in the instruction to locate the component. When a whole array of integers is addressed via a base register, the operand qualifier, ARRAY SIZE (ASIZ) is required in the instruction. In both cases, the compact format, BI(EXT,12) OR BM(EXT,13) may be used as explained in Section 4.2.3 9 (page 4-8). Note, also, that immediate source operands are interpreted as having a V32 tag with sign extend.

5.4.7 IF GREATER THAN FLOATING POINT

Format: 5D_H S1, S2, D

Mnemonic: IFF>

Operands:

S1: <u>Comparand 1</u> FMT: <u>memory (0) or stack (EXT,0)</u>

S2: <u>Comparand 2</u> FMT: <u>memory (0) or stack (EXT,0)</u>

D: Label

FMT: immediate (EXT,2), interpreted as a label operand

Function:

The operands specified by S1 and S2 are compared; if the former value (S1) is greater than the latter value (S2), no action is taken, else a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). Both source operands must be floating point numbers (V32 or V64).

12 mg bully high hugh see

.4.8 IF GREATER THAN OR EQUAL TO INTEGER

ormat: 5E_H S1, S2, D

nemonic: IFI>=

perands:

- S1: <u>Comparand 1</u> FMT: immediate (EXT,2), memory (0), stack EXT,0), or base register (EXT,11)
- S2: <u>Comparand 2</u> FMT: <u>immediate (EXT,2), memory (0), stack (EXT,0),</u> or base register (EXT,11)

D: Label FMT: immediate (EXT,2), interpreted as a label operand

unction:

The operands specified by S1 and S2 are compared; if the former value (S1) is greater than or equal to the latter value (S2), no action is taken, else a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). Both source operands must be integers (V16 or V32) or single dimension arrays of integers. If arrays, lower bounds and lengths need not match. The value of the array operand specified by S1 is greater than or equal to the value of the array operand specified by S2 if the former value is not less than the latter value as formally defined in the IF LESS THAN INTEGER instruction (Section 5.4.4).

Note that a source operand may be an integer component of an array or record. As usual, when an array component is addressed via a base register, the operand qualifier, BASE RELATIVE OFFSET (BRO) is required in the instruction to locate the component. When a whole array of integers is addressed via a base register, the operand qualifier, ARRAY SIZE (ASIZ) is required in the instruction. In both cases, the compact format, BI(EXT,12) OR BM(EXT,13) may be used as explained in Section 4.2.3 9 (page 4-3). Note, also, that immediate source operands are interpreted as having a V32 tag with sign extend.

5.4.9 IF GREATER THAN OR EQUAL TO FLOATING POINT

Format: 5FH S1, S2, D

Mnemonic: IFF>=

Operands:

S1: <u>Comparand 1</u> FMT: <u>memory (0) or stack (EXT,0)</u>

S2: <u>Comparand 2</u> FMT: <u>memory (0) or stack (EXT,0)</u>

D: <u>Label</u> FMT: immediate (EXT,2), interpreted as a label operand

Function:

The operands specified by S1 and S2 are compared; if the former value (S1) is greater than or equal to the latter value (S2), no action is taken, else a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). Both source operands must be floating point numbers (V32 or V64).

4.10 IF LESS THAN OR EQUAL TO INTEGER

rmat: 60_H S1, S2, D

emonic: IFI<=

erands:

- 1: <u>Comparand 1</u> FMT: immediate (EXT,2), memory (0), stack EXT,0), or base register (EXT,11)

): <u>Label</u> FMT: immediate (EXT,2), interpreted as a label operand

inction:

he operands specified by Sl and S2 are compared; if the former slue (S1) is greater than or equal to the latter value (S2), no tion is taken, else a branch is made to the address of the irrent instruction plus the value of the label operand displacement, in words). Both source operands must be integers /16 or V32) or single dimension arrays of integers. If arrays, ower bounds and lengths need not match. The value of the array perand specified by Sl is less than or equal to the value of the rray operand specified by S2 if the latter value is not less han the former value as formally defined in the IF LESS THAN NTEGER instruction (Section 5.4.4).

bte that a source operand may be an integer component of an rray or record. As usual, when an array component is addressed ia a base register, the operand qualifier, BASE RELATIVE OFFSET BRO) is required in the instruction to locate the component. hen a whole array of integers is addressed via a base register, he operand qualifier, ARRAY SIZE (ASIZ) is required in the nstruction. In both cases, the compact format, BI(EXT,12) OR M(EXT,13) may be used as explained in Section 4.2.3 9 (page 4-). Note, also, that immediate source operands are interpreted s having a V32 tag with sign extend.

5.4.11 IF LESS THAN OR EQUAL TO FLOATING POINT

Format: 61_H S1, S2, D

Mnemonic: IFF<=

Operands:

S1: <u>Comparand 1</u> FMT: <u>memory (0) or stack (EXT,0)</u>

S2: <u>Comparand 2</u> FMT: <u>memory (0) or stack (EXT,0)</u>

D: Label FMT: immediate (EXT,2), interpreted as a label operand

Function:

The operands specified by S1 and S2 are compared; if the former value (S1) is less than or equal to the latter value (S2), no action is taken, else a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). Both source operands must be floating point numbers (V32 or V64).

.12 IF DEFINED

nat: 62_H S, D

monic: IFD

rands:

MT: <u>Data Entity to Be Tested</u> MT: <u>memory (0), stack (EXT,0), or base register</u>

(EXT,11)

Label

MT: immediate (EXT,2), interpreted as a label operand

ction:

undefined bit in the tag of the operand addressed by S is ted; if 0 (data defined), no action is taken but if 1 (data efined), a branch is made to the address of the current truction plus the value of the label operand (displacement, in ds). The source operand may be any data type including a nter, formal reference parameter, an array or record ponent, or a whole array or record. A pointer is defined if undefined flag=0; however, if the pointed -to entity is a a object (ENT=010) with a true (1) unique name flag, the nter is defined only if the addressed data object still sts, i.e., if the pointer is not a "dangling reference". A he array or record is defined only if every component is ined.

e that if S addressed an array component via a base register, operand qualifier, BASE RELATIVE OFFSET (BRO) is required in instruction to locate the component. If S addresses a whole 'ay via a base register, the operand qualifier, ARRAY SIZE 'IZ) is required in the instruction. In both cases, the 'pact format, BI(EXT,12) or BM(EXT,13) may be used as explained Section 4.2.3 (page 4-8).

eptions:

5.4.13 IF IN RANGER INTEGER

Format: 63_H S1, S2, S3, D

Mnemonic: IFIRNG

Operands:

S1:	Test Integer			
FMT:	memory (0)	or	stack	(EXT,0)

S2:	Upper Limit Integer	
FMT:	immediate (EXT,2), memory (0), or stack	(EXT,0)

- S3: Lower Limit Integer FMT: immediate (EXT,2), memory (0), or stack (EXT,0)
- D: Label FMT: immediate (EXT,2), interpreted as a label operand

Function:

An in-range test is performed on the operand addressed by S1. If that operand is less than or equal to the operand addressed by S2 and greater than or equal to the operand addressed by S3, no action is taken; otherwise, a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). All source operands must be integers (V16 or V32). Any source operand may be an integer component of an array or record. Immediate source operands are interpreted as having as V32 tag with sign extend.

5.4.14 IF IN RANGE FLOATING POINT

Format: 64_H S1, S2, S3, D

Mnemonic: IFFRNG

Operands:

51:	Test Floating		Poin	t Numb	umbers	
FMT:	memory (0)	or	stack	(EXT,0)	

- S2: Upper Limit Floating Point Number FMT: memory (0) or stack (EXT,0)
- S3: Lower Limit Floating Point Number FMT: memory (0) or stack (EXT,0)

D: Label

FMT: immediate (EXT, 2), interpreted as a label operand

Function:

An in-range test is performed on the operand addressed by S1. If that operand is less than or equal to the operand addressed by S2 and greater than or equal to the operand addressed by S3, no action is taken; otherwise, a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words). All source operands must be floating point numbers (V32 or V64). Any source operand may be a floating point component of an array or record.

5.4.15 GOTO

Format: 65_H, D

Mnemonic: GOTO

Operands:

D: Label

FMT: immediate (EXT,2), interpreted as a label operand

Function:

A branch is made to the address of the current instruction plus the value of the label operand (displacement, in words).

Exceptions: NONE 5.4.16 CASE

Format: $66_{\rm H}$ S1, S2, S² D₀, . . . D_n

Mnemonic: CASE

Operands:

S1: <u>Case Selector</u> FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

S2: Lower Limit of Case Selector Range FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

S3: Upper Limit of Label Subscripts FMT: immediate (EXT,2)

D: <u>lst Label</u> FMT: immediate (EXT,2), interpreted as a label operand

•

D_n <u>nth Label (n=S3)</u> FMT: <u>immediate (EXT,2)</u>, interpreted as a label operand

Function:

The operand specified by S2 (lower limit of case selector range) is subtracted from operand S1 (case selector) to produce an unbiased (base 0) label subscript, called k. If k is in the range 0..S3, where S3 is an immediate operand, a branch is made to the address of the current instruction plus the value of the label operand, D_k . If k is greater S3, a branch is made to the address of the current instruction plus the value of the label operand, D_n , where n=S3 (value of upper limit of label subscripts).

All source operands must be integers (V16 or V32). Immediate source operands are interpreted as having a V32 tag with sign extend. Source operands, S1 and S2, may be immediate values, directly addressed integers, indirectly addressed integers (via pointers to integers in global storage or data objects of type integer), or integer components of an array or record.

5.4.17 SET LOOP CONTROL VARIABLE

Format: 67_H, S, D

Label

Mnemonic: SETLCV

Operands:

S: <u>Initial Value of Loop Control Variable</u> FMT: <u>immediate (EXT,2), memory (0), or stack (EXT,0)</u>

D:

FMT: immediate (EXT,2), interpreted as a label operand

Function:

The operand specified by S is used as the initial value of the loop control variable in the instruction, LOOP UP or LOOP DOWN. Hence, this instruction and LOOP UP or LOOP DOWN form an inseparable execution couplet. The source operand must be an integer (V16 or V32). The instruction at an address equal to the current instruction address plus the value of the label operand (displacement, in words) is next executed and must be LOOP UP or LOOP DOWN.

The source operand may be an immediate value, a directly addressed integer, an indirectly addressed integer (via a pointer to an integer in global storage or a data object of type integer), or an integer component of an array or record. An immediate source operand is interpreted as having a V32 tag with sign extend.

Land and the second

PROGRAM ERROR

5.4.18 LOOP UP

Format: 68_H S1, S2, S3, D

Mnemonic: LOOPUP

Operands:

51:	Loop Control Variable			
FMT:	memory	(0) or stack	(EXT,0)	

S2: Increment Amount

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

S3: Upper Limit

FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

D: Label

immediate (EXT,2), interpreted as a label operand

Function:

FMT:

This instruction, compiled from the Ada FOR LOOP iteration If this instruction is the scheme, controls program looping. branch target of a SET LOOP CONTROL VARIABLE instruction, the operand addressed by S1 (loop control variable) is set to the value of the operand specified by S in the SET LOOP CONTROL VARIABLE instruction. Since these are integer operands with V16 or V32 tags, a check is made for overflow (magnitude of operand S from SET LOOP CONTROL VARIABLE instruction larger than precision of operand S1 in LOOP UP instruction allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the When this instruction is not the branch instruction proceeds. target of a SET LOOP CONTROL VARIABLE instruction, the loop control variable is incremented by the value of the operand specified by S2 (increment amount). The incremented value is checked for overflow (magnitude larger than precision of loop control variable allows; a NUMERIC_ERROR exception is raised in the presence of overflow else the instruction proceeds. The incremented (or preset) value of the loop control variable is next checked against the operand specified by S3 (upper limit of loop control variable). If less than or equal to the limit value, a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words); if greater than the limit value, no further action is taken.

All source operands must be integers. Source operands specified by S2 and S3 may be immediate values, directly addressed integers, indirectly addressed integers (via pointers to integers in global storage or data objects of type integer), or integer components of an array or records. Source operands S1 may be any of these except an immediate value. An immediate source operand is interpreted as having a V32 tag with sign extend.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR 5.4.19 LOOP DOWN

Format: 69_H S1, S2, S3, D

Mnemonic: LOOPDN

Operands:

- Sl: <u>Loop Control Variable</u> FMT: <u>memory (0) or stack (EXT,0)</u>
- S2: Decrement Amount

Label

FMT: immediate (EXT, 2), memory (0), or stack (EXT, 0)

S3: Lower Limit

FMT: immediate (EXT, 2), memory (0), or stack (EXT, 0)

D:

FMT: immediate (EXT,2), interpreted as a label operand

Function:

This instruction, compiled from the Ada FOR LOOP iteration scheme, CONTROLS PROGRAM LOOPING. If this instruction is the branch target of a SET LOOP CONTROL VARIABLE) instruction, the operand addressed by S1 (loop control variable) is set to the value of the operand specified by S in the SET LOOP CONTROL VARIABLE instruction. Since there are integer operands with V16 or V32 tags, a check is made for overflow (magnitude of operand S from SET LOOP CONTROL VARIABLE instruction larger than precision of operand S1 in LOOP UP instruction allows). A NUMERIC ERROR exception is raised in the presence of overflow, else the instruction proceeds. When this instruction is not the branch target of a SET LOOP CONTROL VARIABLE instruction, the loop control variable is decremented by the value of the operand specified by S2 (decrement amount). The decremented value is checked for overflow (magnitude larger than precision of loop control variable allows); a NUMERIC ERROR exception is raised in the presence of overflow, else the instruction proceeds. The decremented (or preset) value of the loop control variable is next checked against the operand specified by S3 (lower limit of loop control variable). If greater than or equal to the limit value, a branch is made to the address of the current instruction plus the value of the label operand (displacement, in words); if less than the limit value, no further action is taken.

All source operands must be integers. Source operands specified by S2 and S3 may be immediate values, directly addressed integers, indirectly addressed integers (via pointers to integers in global storage or data objects of type integer), or integer components of an array or record. Source operand S1 may be any of these except an immediate value. An immediate source operand is interpreted as having a V32 tag with sign extend.

Exceptions: PROGRAM_ERROR NUMERIC_ERROR

6 SUBPROGRAMS

Any visible subprogram in the local package, i.e., enclosing, immediately enclosed, sibling, or self (as determined by the compiler) can be called. In addition, any non-nested subprogram in an external package can be called. (Nested subprograms in external packages cannot be called because their environments are not visible.) The instructions which call a subprogram, pass parameters, and return to the calling environment are described in this section. CALL SUBPROGRAM.

mat: 6A_H, S1, S2,...

monic: CALL

rands:

: <u>Subprogram Identification</u> MT: immediate (EXT,2) or memory (0) Immediate: Sl specifies an offset to a subprogram component in the local package header. Memory: Sl addresses a pointer to a program (subprogram) in a external package.

The second second

The side shirt that friends and

ote: If no parameters are passed via memory transfer, hen no additional operands are present in this nstruction.

MT: <u>Actual Parameters</u> MT: <u>immediate (EXT,2)</u> or memory (0)

lote: Any number of parameters may be passed via memory ransfer. Any two may be combined in a 2-operand compact format. This instruction calls a visible subprogram in the local package (offset to subprogram component in local package header given by immediate value of operand S1) or calls a non-nested subprogram in an external package (offset to subprogram component in external package header contained in the pointer addressed by S1). The pointer must have READ authority for the called subprogram.

The CALL instruction is processed only up to the actual parameter operands; thus, the value in the program counter addresses the word following operand S1 (subprogram identification). The value in the program counter is saved in the administrative data area of the caller and becomes the return address when the number of memory parameters is zero. If, however, one or more parameters are passed via memory transfer, they are bound during execution of the BIND PARAMETERS instruction (first instruction of the called subprogram) which requires access to both actual and formal parameters. BIND PARAMETERS completes the processing of the operands in the CALL instruction and the last value in the program counter (return address) is saved in the administrative data area of the caller. (In addition to saving the return address, the following quantities are saved in the caller's administrative data area: address of first instruction of calling subprogram, address of last instruction of calling subprogram, caller's stack index, all general purpose registers that correspond to "1s" in the caller's Temporaries Mask, caller's nesting depth, and caller's exception mode.) See. Section 6.2.2 for more details on passing parameters via memory.

Parameters that are passed via registers are loaded into parameter registers (16..31) at some points during the caller's execution and no further action is required. BIND PARAMETERS is not present in the called subprogram if all parameters are passed via register. Parameter placement in registers is checked by the machine. A PROGRAM ERROR exception is raised if the "1s" in the formal parameter mask retrieved from the called subprogram component in the package header (set by the compiler) do not match "1s" in the Valid Parameter Mask (set by the machine as the actual parameters are loaded into registers). See Section 6.2.1 for more details on passing parameters via registers.

When a subprogram in the local package is called, the offset to the subprogram component in the local package header, given by operand S1, is an immediate value. The base address of the local package header is the value in display register 0 that addresses the base of the package variable global data template -1. ("One" btracted because the header is displaced by 1 word from the ble global data template - see Figure 2-1). When a ogram in an external package is called, the offset to the ogram component in the external package header is retrieved word 1 of the pointer to the subprogram, addressed by nd S1. The base address of the external package header is alue retrieved from word 2 of the pointer. In both cases, ffset is subtracted from the base address of the header to ce the subprogram component address. This is the start of a d packet of information pertinent to the called subprogram Figure 2.2). The following information is extracted:

- size of activation record, in words (28 bits) The size includes immediate and separate array values and the stack. This value is passed to the memory manager which allocates space for the activation record plus the fixed size administrative data area and returns the base address of the activation record. The base address is loaded in one of the pair of local display registers. (See discussion below on adjusting display registers.)
- nesting depth of called subprogram (4 bits) The nesting depth is used in the determination of which display registers are saved. (See discussion below).
- address of first and last instruction of called subprogram (each 32 bits) - Program control is transferred to the first instruction; instruction addressing and branching is confined to be within the specified limits.
- address of automatic data template (32 bits) This address is loaded in one of the pair of local display registers (see discussion below on adjusting display registers).
- formal parameter mask (16 bits) The "1s" in the formal parameter mask must match "1s" in the Valid Parameter Mask register and the former is loaded into the Valid Parameter Mask register.
- exception mode (4 bits) This field specifies the initial exception mode (ELABORATION or NORMAL) of the called subprogram. The called subprogram enters the ELABORATION exception mode if, at the Ada program level, the subprogram has a declarative part that requires creation of objects and/or subcomputations for initialization of declared objects; the NORMAL exception mode is entered otherwise.

t, the stack index is set to "0", the Temporaries Mask is ared for the called subprogram, and display registers are usted. When the called subprogram is in the local package, rules for adjusting display registers depend on the nesting ths of the called and calling subprograms as follows:

- (a) nesting depth (ND) of called subprogram = ND of calling subprogram +1 - The base address of the called subprogram's automatic data template is loaded into one of the pair of the display registers corresponding to the ND of the called subprogram.
- (b) NDs of called and calling subprograms are equal The contents of the local display register pair (of the calling subprogram) are saved in the calling subprogram's administrative data area and the base address of the called subprogram's data template is loaded into one register of that display register pair.
- (c) ND of called subprogram is less than ND of calling subprogram - The contents of the local display register pair (of the calling subprogram) and of each display register pair corresponding to NDs less than that of the calling subprogram but greater than and equal to that of the called subprogram are saved in the administrative data area of the calling subprogram. The base address of the called subprogram's data template does not have to be loaded into one of the registers of the display register pair corresponding to the nesting depth of the called subprogram since it is already resident.

In the called subprogram is in an external package, all display pister pairs corresponding to NDs less than and equal to the ND the calling subprogram (including ND = 0 and 15 for global a) are saved in the caller's administrative data area. The is addresses of the external package header in data template wory and the external package administrative data in data value wory are retrieved from words 2 and 3 of the pointer to the program; these addresses are incremented by "1" so that they int to the variable global data in data template and data value wory, respectively. They are then loaded into display register if 0 (corresponding to ND = 0). Next, the size of the variable and offset of -1 from the package header descriptor (PKG) an offset of -1 from the base address of the variable and data template (see Figure 2.2). The base address of the istant global data is computed as the sum of the base address the variable global data's template and the size of the istant global data. This value is loaded into display register
rresponding to ND = 15). Finally, the base address of the subprogram's and its automatic data template is loaded ne of the pair of display registers corresponding to ND =

bove operations take place concurrently with storage tion. When space for the called subprogram's activation and administrative data has been allocated, the base s of the activation record is loaded into the second er of the display register pair that corresponds to the g depth of the called subprogram. (Recall that for both and external subprogram calls, the base address of the subprogram's automatic data template is already in place other register of the display register pair). In addition usting the display register, the following information is n into the called subprogram's administrative data area:

lynamic link to base of calling subprogram's dministrative data.

tatic Save Flag which is set to "0".

ate of the Static Save Flag designates whether the static lents of the machine state (priority level, addresses of and last subprogram instructions, nesting depth of ogram, and display registers of nesting depths < = nesting of subprogram) need to be saved in the subprogram's istrative data area when the subprogram is executing in ice of a task switch. A "0" means the information must be and a "1" means it does not have to be saved because it is ly in the administrative data area (previously saved). Note the dynamic components of the machine state (registers sponding to "1s" in the Temporaries and Valid Parameter , stack index, exception mode, and execution resumption is) are always saved when tasks are switched.

ions: XAM_ERROR AGE_ERROR

6-6

Parameter Association. Actual parameters may be passed value or by reference via the register file or via memoryory transfer. Performance advantages accrue from using the ister file. If the number of words taken by the parameters eeds sixteen, however, some parameters must be passed by ory-memory transfer.

Passing via Register File. Registers 16 through 31 of .1 register file are dedicated to passing parameters. Bits .31 of register 0 comprise the Valid Parameter Mask. Each bit this mask corresponds to a parameter register in the following : bit 16 corresponds to register 16; bit 17 corresponds to lister 17; ...; bit 31 corresponds to register 31. The bits in . Valid Parameter Mask specify which registers are to be saved the current task is (asynchronously) switched out. These The Valid Parameter Mask also isters hold valid parameters. cifies which registers hold passed parameters when a program is entered.

en a parameter is to be passed, it is loaded into one of the sisters (range 16..31) and the corresponding bit in the Valid ameter Mask is automatically set to 1. As with the general pose registers (registers 0..15), a PROGRAM_ERROR is raised if attempt is made to read a parameter register when the responding bit in the Valid Parameter Mask is not "1". If ssing a parameter by value, any instruction may be used (e.g., /E) with the destination being the register. If passing by ference, three instructions (executed by the caller) are silable:

> LOAD RO REFERENCE PARAMETER LOAD WO REFERENCE PARAMETER LOAD RW REFERENCE PARAMETER.

> > 6-7

These instructions load a Formal Reference Parameter (FRP) into a register (See Section 3.5 for a description of the format of an FRP). RO designates that the called subprogram has read-only authority to the actual parameter, WO designates write-only authority, and RW designates that the subprogram has both read and write authorities to the actual parameter. As with pointers, initial values are not permitted in FRPs; hence, FRPs are set to UNDEFINED by the machine when the containing package is loaded. An FRP contains a path to the actual parameter (absolute addresses of parameter in data template memory and in data value memory) and specifies the rights which the called subprogram has to the actual parameter. The subprogram uses the FRP like a pointer to the actual parameter, i.e., it references the actual parameter indirectly via the FRP. An actual parameter could be (In these cases, an array, a slice, or a component thereof. additional descriptors and/or operand qualifiers follow.) Hence, arrays, slices, and components' can be passed by reference. Ιf the actual parameter is a pointer, the called subprogram, executing the appropriate instruction, can indirectly reference any of the entities which a pointer can point to (See Table 3.6). For example, if the entity is a task object, the subprogram can call an entry of the task by referencing the pointer parameter Similarly, if the pointed-to entity indirectly through an FRP. is a non-nested subprogram in an external package, the called subprogram can call the external subprogram by referencing the pointer parameter indirectly through an FRP. When any entity is thus accessed through an FRP-pointer pair, the subprogram's rights to the target entity are the most restrictive rights present in the FRP-pointer. (As discussed earlier, the rights of the called subprogram to an actual parameter can be controlled.)

If the source operand of a Load Reference Parameter instruction (an actual parameter) is an FRP, all fields of the actual FRP are moved to the FRP in the register, in a one-to-one correspondence. This eliminates a level of indirection when the subprogram references the FRP. The effect is that of passing the actual FRP by value. The use of registers 16..31 to pass parameters may be summarized as follows:

- Whenever a parameter is loaded into a register, a corresponding bit is automatically set in the Valid Parameter Mask. Parameters may be values (pass by value) or FRPs (pass by reference). Only registers corresponding to "1s" in the Valid Parameter Mask may be read.
- 2. In presence of a task switch, those parameters in registers corresponding to "1s" in the Valid Parameter Mask, including register 0, are automatically saved in the administrative data area of the current task object or subprogram, depending on which is executing; the parameters are restored when the task program or subprogram is again scheduled to run.
- 3. During the CALL instruction, the following steps relating to passing parameters via registers are taken:
 - The Formal Parameter Mask for the called subprogram (read from the called subprogram component in the package header) has a "1" corresponding to each formal parameter. Corresponding "1s" must be present in the Valid Parameter Mask register to indicate that expected actual parameters were indeed passed. If an expected "1" is missing in the Valid Parameter Mask register, a PROGRAM ERROR exception is raised. Note that additional "1s" may be present in the Valid Parameter Mask register; these would correspond to parameters that were passed to the caller. They present no problem since the Valid Parameter Mask is regenerated at each CALL.
 - The Formal Parameter Mask (retrieved from the called subprogram's header) is loaded into the Valid Parameter Mask register.
- 4. During execution of the RETURN FROM SUBPROGRAM or END RENDEZVOUS instruction, the Valid Parameter Mask is cleared.

Finally, an instruction, CLEAR VALID PARAMETER MASK, is provided to allow compiler optimization. (There is no reason, during task switching, to save/restore parameter registers that contain data which is no longer needed.)

Passing via Memory Transfer. As with registers, 6.2.2 parameters can be passed by value or by reference. When passing by value, the actual parameter(s) addressed in the CALL instruction (immediate or memory operands) are retrieved and stored in the formal parameter location(s) specified by the BIND PARAMETERS instruction. This instruction must be the first in The tags of corresponding actual and the called subprogram. formal parameters must match according to the rules of the MOVE, MOVE POINTER, MOVE ARRAY, or MOVE ARRAY SLICE instruction. When passing a parameter by reference, the CALL instruction addresses the actual parameter (cannot be an immediate operand) and the BIND PARAMETERS instruction in the called subprogram addresses and FRP (which must be in the called subprogram's activation record). Then, as discussed in Section 6.2.1, the following values are stored in the FRP:

- Absolute address of actual parameter in data template memory.
- Absolute address of actual parameter in data value memory.

The comments in Section 6.2.1 on FRPs applies here, as well, except for the method of restricting rights to the actual parameters. When reference parameters are passed via registers, the instructions that load the registers restrict rights to readonly, write-only, or read-write. However, when passing reference parameters via memory transfer, the compiler stores the rights (READ, WRITE, or READ and WRITE) directly into the FRP.

Note that pass by value parameters and pass by reference parameters with read-only authority support Ada in parameters, pass by reference parameters with write-only authority support Ada out parameters, and pass by reference parameters with readwrite authority support Ada in-out parameters. 6.2.3 LOAD RO REFERENCE PARAMETER.

Format: 6B_H, S, D

Mnemonic: LDRO

Operands:

S: <u>Data Entity (Actual Parameter)</u> FMT: <u>memory (0)</u>

D: <u>Register Address</u> FMT: <u>memory (0)</u>

Function:

This instruction loads registers addressed by D, D+1, and D+2 with a formal reference parameter that points to the actual parameter addressed by S. The cell offset in D can only designate registers 16..29. The rights given to the formal reference parameter are READ only. The following values are loaded into the formal reference parameter:

(a) when ADS of data entity is not 0 or 15

- WORD1 ENT <= 110 (actual parameter in an activation).
 - RIGHTS <= READ.
- WORD2 Absolute address of actual parameter in the automatic data template corresponding to the nesting depth (ND) specified by ADS (base address of template in display register #ND + cell offset specified by S).
- WORD3 Absolute address of actual parameter in activation record corresponding to nesting depth (ND) specified by ADS (base address of activation record in display register #ND + cell offset specified by S).

(b) when ADS of data entity = 0

WORD1 - ENT <= 011 (actual parameter in variable global data area).

- RIGHTS <= READ.

- WORD2 Absolute address of actual parameter in the variable global data template of the enclosing package (base address of template in display register #0 + cell offset specified by S).
- WORD3 Absolute address of actual parameter in the variable global data area of the enclosing package (base address of data values in display register #0 + cell offset specified by S).
- c) when ADS of data entity = 15
 - WORD1 ENT <= 100 (actual parameter in constant global data area).
 - RIGHTS <= READ.
 - WORD2 Absolute address of actual parameter in the constant global data area of the enclosing package (base address in display register #15 + cell offset specified by S).

WORD3 - Not used.

Exceptions: PROGRAM_ERROR CONSTRAINT_ERROR 6.2.4 LOAD WO REFERENCE PARAMETER.

Format: 6C_H, S, D

Mnemonic: LDWO

Operands:

S: <u>Data Entity (Actual Parameter)</u> FMT: <u>memory (0)</u>

D: <u>Register Address</u> FMT: <u>memory (0)</u>

Function:

This instruction loads registers addressed by D, D+1, and D+2 with a formal reference parameter that points to the actual parameter addressed by S. The cell offset in D can only designate a register in the range 16..29. The rights given to the formal reference parameter are WRITE only. The detail functions performed by this instruction are the same as described for LOAD RO REFERENCE PARAMETER except that the assigned rights are WRITE ONLY.

Exceptions: PROGRAM ERROR CONSTRAINT ERROR 2.5 LOAD RW REFERENCE PARAMETER.

rmat: 6D_H, S, D

emonic: LDRW

erands:

:	Data Entit	y (Actual	Parameter)	
FMT:	memory	(0)		

: <u>Register Address</u> FMT: <u>memory (0)</u>

nction:

is instruction loads registers addressed by D, D+1, and D+2 th a formal reference parameter that points to the actual rameter addressed by S. The cell offset in D can only signate a register in the range 16..29. The rights given to e formal reference parameter are READ and WRITE. The detail nctions performed by this instruction are the same as described r LOAD RO REFERENCE PARAMETER except that the assigned rights e READ and WRITE.

ceptions: ROGRAM_ERROR ONSTRAINT_ERROR

CLEAR VALID PARAMETER MASK. 5.2.6

recall & multiplesses

and the state of t

6EH Format:

Inemonic: CLRVPM

Operands: None

Function: The Valid Parameter Mask, bits 16..31 of register 0, is cleared.

the statement of anti-tweeters of the sector and the

.

and the set the set of the set

.

Cale Contract - Charles and the

THE REPORT OF THE PARTY

Exceptions: None

BIND PARAMETERS.

it: 6F_H, D1, D2, D3,...

onic: BIND

inds:

1:

Formal Parameter memory (0)

:ion:

is the first instruction of a called subprogram (or an T body) when parameters are to be passed via memory-memory Any number of formal memory parameters can be sfer. ified as operands. Formal and actual parameters have a one-ne correspondence. It is the responsibility of the compiler reck that the number of formals and actuals match. Before ievina the actual parameters, the caller's execution nption address (saved program counter value) must be read the caller's administrative data area at a location which is (fixed) offset relative to the dynamic link to the own er's administrative data area. This is the address in the SUBPROGRAM instruction of one the CALL ENTRY (or ructions) where execution was halted (at the address of the and which is the first actual parameter). To compute the lute addresses of the actual parameters (each as a cell et + base address of activation or package global data), the er's display register environment, saved in the caller's nistrative data area, is required. A display register is ssed using a known offset from the dynamic link, indexed by actual parameter's nesting depth (given by ADS). One-by-one, al parameters are retrieved and stored in corresponding al parameters, addressed by Di. All formal parameters must n the local activation record of the called subprogram (or er task). Parameters that are passed by value (Di addresses ta cell which is not a formal reference parameter) must obey rules of the MOVE, MOVE POINTER, MOVE ARRAY or MOVE ARRAY E instruction. If an array component, array, or slice is ed by value, the actual parameter expands to a group of rmation (descriptors and/or operand qualifiers) that defines array or component. The compiler must ensure that sufficient e has been allocated in the activation record of the called rogram (or server task). Parameters that are passed by

eference (Di addresses a formal reference parameter) are not hysically moved; rather, the absolute addresses of an actual arameter in data template memory and in data value memory are cored in words 2 and 3 of the formal reference parameter. These idresses are derived from the cell offset and display register air of the actual parameter. When all parameters have been assed (new instruction detected in place of another actual arameter), the address of this next instruction is stored in the aller's execution resumption address (return address for abprogram calls) in the caller's administrative data area.

LA DANCE - LAN LAND -

kceptions: PROGRAM_ERROR

...

RETURN FROM SUBPROGRAM.

70_H

ic: RETSUB

ds:

on:

nstruction completes the execution of the subprogram. If ependent tasks are extant, the activation record and strative data storage cannot be immediately reclaimed. The of the subprogram (same as the state of the task to which bprogram is dynamically linked, through any number of) is changed from RUNNING to SUSPENDED. The only ents of the machine state that need to be saved in the gram's administrative data area are priority level and ion resumption address. The latter is the address of this FROM SUBPROGRAM instruction. The task scheduler schedules r task to run. At some later time, when the last dependent s TERMINATED, the state of the subprogram is changed to and the subprogram is put on the ready queue corresponding saved priority level. When the subprogram is scheduled to he RETURN FROM SUBPROGRAM instruction is again executed, ime without any non-terminated dependent tasks. Storage w be reclaimed for any data objects that designated this gram in the CREATE DATA OBJECT or CREATE UNCHECKED DATA instruction. (Designation of the subprogram means, at the rogram level, that the data object's access type was ed in the subprogram.) The dynamic link to the strative data area of the calling subprogram or task is ved from the called subprogram's administrative data area; alled subprogram's activation record/administrative data e are then deallocated. . Finally, the dynamic link is used rieve the machine state (dynamic and static components) of lling subprogram, completing the return.

.

ions:

7 PACKAGES

ickage objects and the loading of packages were introduced Section 2.1 and 2.5.1: Packages reside in three 1 mories: (1) data template memory, which contains headers, instant global data, variable global data template (initial ilues), and automatic data templates of subprograms and programs, (2) data value memory, which contains isk iministrative data and modified values of variable global which and (3) instruction memory, contains ata. structions of subprograms and task programs contained in he package (see Figure 2-1). Note that variable global ata requires a template because subprograms that declare ested packages support reentrancy and recursion. Nonested (library) packages may contain nested packages that ere declared in the Ada package body or in subprograms or asks that were themselves declared in the non-nested The location in data template memory and ackage. package hstruction memory of each nested (its eader/variable global data template, constant global data, utomatic data templates, and instructions) is specified in 5-word component of the non-nested package's header (see iqure 2-2). In the external representation of a non-nested ackage header, all locations are specified as offsets from he base of the non-nested package in data template memory nd from the base of instructions in instruction memory. hen a non-nested package is loaded, these offsets are onverted to absolute addresses.

ackages are compiled to machine code in the following eneral way:

- (a) Data declared in an Ada package specification is placed in the package's global data area (constants in the constant global data area and variables with initial values in the variable global data area).
- (b) Data declared in an Ada package body is located in the global data area if any subprograms or task programs in the package body reference the data. Data declared in the package body that is only referenced by subprograms 0 (that elaborates the declarative part of the package) is located in the automatic data template of subprogram 0. It is the responsibility of the compiler to enforce visibility rules.

- (c) Packages declared in the specification part of an Ada non-nested package are merged with the nonnested package, i.e., data is merged with the nonnested package's global data and subprograms and task programs are included with those of the nonnested package. This has the effect of creating a single larger package.
- (d) Packages declared in the body of a non-nested package are not merged as in (c); rather, they are nested packages, created in subprogram 0 of the non-nested package and elaborated via calls to subprogram 0 of the nested packages. Each nested package becomes dependent on the non-nested package that created it.
- (e) Tasks and subprograms declared in an Ada package are defined in the package header, each as a 5-word component of the header, as shown in Figure 2-2. Tasks declared in the Ada package body are created and activated (see Sections 9.4.1, 9.4.2, and 9.4.6) in subprogram 0 of the package; tasks declared in the Ada package specification may be created/activated in subprogram 0 or in an external program. (Tasks created in subprogram 0 become dependent on the enclosing non-nested package.) The instructions and automatic data of the task are included in the programs and subprograms package, as shown in Figure 2-1.
- (f) The sequence of statements, if any, in an Ada package body compile to instructions in subprogram 0.

that packages that are declared in tasks or subprograms nested in these programs and, therefore, are created and prated within them and become dependent on them (package al and administrative data areas not reclaimed until the is TERMINATED or subprogram returns). In general, a ed package depends on the environment in which it is ared (created, at the machine level). The instructions automatic data of programs in a nested package declared task or subprogram are located in the enclosing noned package (see Figure 2-1). Non-nested (library) ages do not depend on any objects and their global age lives "forever." Finally, note that some Ada ages may not include a body. These packages contain declared data entities which, at the machine level, ar in the global data area of the package. No further

boration of such a package is required, i.e., there is no 1 to subprogram 0.

all nested including -nested packages, package ponents, are loaded via a User Console interface card IC) that contains a bootstrap loader (see Section 2.5.1). first package loaded, called the loader-linker, contains grams which load other packages and link programs and The loader-linker is loaded under a in those packages. itrol of the bootstrap loader microcode which relies on mands and responses from the User Conscle. The loading cedure commences when the User Console sends an "initiate d" command to the UCIC. This immediately puts the HLLM the User console (highest) priority level. The UCIC then ids a request to the User Console for the storage size of . : package to be loaded. The User Console responds by iding the sizes required in data template and instruction nories. The UCIC, which contains a 2K x 36 bit buffer arge enough to accommodate any package header), sends a juest to the User Console for the package header. At the ne time, it requests storage allocation for data and Note that the UCIC structions from the memory manager. st buffer individual 16-bit or 32-bit words from the User isole before storing standard HLLM 36-bit words in the 2K ffer (nine 16-bit words form four 36-bit words or nine 32t words form eight 36-bit words). When the header has en loaded into the 2K buffer and the base addresses in ta template and instruction memories are known (returned om the memory manager in the form of a pointer cell -- see ction 3.4), the UCIC converts all address offsets in the n-nested package header to absolute addresses. Next, it ansfers the contents of the 2K buffer to the allocated prage in data template memory in the HLLM. At this point, owing the size of the header, the UCIC modifies word 2 of e pointer (address of the package in data template memory) subtracting the header size; the result is the address of e base of the header, i.e., the package descriptor, PKG ee Figure 2-2). This address is needed later, when the The UCIC then requests the next data ckage is created. ock (equal or less than 2K x 36 bits in size) from the er Console. When stored in the buffer, the UCIC scans the ta, nulling all pointers (setting UNDEFINED bits to 1) and addresses (AVAs) and formal tting all array value ference parameters (FRP's) to UNDEFINED. The buffer ntents are then transferred to the HLLM. This process is ntinued until the data template memory load is complete. e UCIC then requests a block of instructions from the User nsole. When loaded, the UCIC transfers the contents of e 2K buffer to the instruction memory in the HLLM. When e entire non-nested package (loader-linker) is resident in e HLLM, the UCIC sends a "load complete" response to the er Console. The User Console then, normally, sends a

58 120	AD		AVION ION SE	IICS CO T ARCH	DMPUTE	R ARCH	ITECTU ANDERS	RE YOU	UME 2	INC	3,	·4
SSIFIE	D AF	AL-TR	-85-10	41-YOL	-2 F3	3615-7	9-0-19	35	F/G :	9/2	NL	
81_1				BUS OFFS		<u>Bassa</u>		AND AND A COMPANY		Baranan Baranan		
- Millions	in the second se								Management of	Station Station Station		
Beeren	in the second se		jir. Bostanos		En se						MERCEN PERSONAL PERSO	
		a antesia Barran Constantino C	Nazyoveni Nazyoveni Nazyoveni Nazyoveni Nazyoveni Nazyoveni		FREE SIGN		<u>BUSHIR</u>				<u>1978</u> 9223	ANG SAN
Bootenana Er.		is Barrawana	Estational									in manual Line Montainin M
	Harris Harris									a fi Summer and a A file and	A DIT States and Boundaryout	ETT
i Contrata Bantonia					<u>183778</u> 4	Beneralise Beneralise	ingenes T			MELLIONI MELLIONI MELLIONI MELLIONI	- 3000 3000	*
			Sa 120 ADVANCED SSIFIED INSTRUCT Image: Structure Image: Structure Image: Structure Image: Structure	Structure APPANCED AVION SSIFIED INSTRUCTURE Image: Structure Image: Structure Image: Structure Image: Structu	58 120 ADVANCED AVION SET ARCE INSTRUCTION SET ARCE INSTRUCTION SET ARCE INSTRUCTI	58 120 ADVANCED AVIONICS COMPUTE INSTRUCTION SET ARCHITEC. NASHUA NH L GREENSPAN ET SIFIED IMA INTRUCTION SET ARCHITEC. IMA INTRUCTION SET ARCHITEC. NASHUA NH L GREENSPAN ET IMA INTRUCTION SET ARCHITEC. Image: Imag	58 120 ADVANCED AVIONICS COMPUTER ARCHITEC. (U) S SSIFIED INSTRUCTION SET ARCHITEC. (U) S SSIFIED INSTRUCTION SET ARCHITEC. (U) S Image: Signal and the set of	58 120 ADVANCED AVIONICS COMPUTER ARCHITECTURINSTRUCTION SET ARCHITEC. (U) SANDERS NASHUA NH L GREENSPAN ET AL. MAY 85 SSIFIED SSIFIED AFWAL-TR-85-1041-VOL-2 F33615-79-C-19 Image: Strate in the strate	58 120 ADVANCED AVIONICS COMPUTER ARCHITECTURE YOU SANDERS ASSOCIATION SET ARCHITEC. (U) SANDERS ASSOCIATION SET ARCHITECTURE (U) SANDERS ASSOCIATION (U) SANDERS AS	58 120 ADVANCED AVIONICS COMPUTER ARCHITECTURE VOLUME 2 INSTRUCTION SET ARCHITEC. (U) SANDERS ASSOCIATES NASHUA NH L GREENSPAN ET AL. MAY 85 SIFIED F/G Image: Signa in the set of the set	58 120 ADVANCED AVIONICS COMPUTER ARCHITECTURE YOLUME 2 INSTRUCTION SET ARCHITEC. (U) SANDERS ASSOCIATES INC MASHUA NH L GREENSPAR ET AL. MAY 85 AFRAL-TR-85-1041-VOL-2 F33615-79-C-1935 F/G 9/2 Image: Sife of the structure of the structu	58 120 ADVANCED AVIONICS COMPUTER ARCHITECTURE VOLUME 2 INSTRUCTION SET ARCHITEC. (U) SANDERS ASSOCIATES INC INSTRUCTION SET ARCHITEC. (U) SANDERS ASSOCIATES INC INSTRUCTION SET ARCHITEC. (U) SANDERS ASSOCIATES INC INSTRUCTION SET ARCHITECTURE VOLUME 2 INC 3. SSIFIED APARL-TR-85-1041-VOL-2 F33615-79-C-1935 F/G 9/2 NL Image: Solution Set architecture volume 2 AFARL TR-85-1041-VOL-2 F33615-79-C-1935 F/G 9/2 NL Image: Solution Set architecture volume 2 AFARL TR-85-1041-VOL-2 F33615-79-C-1935 F/G 9/2 NL Image: Solution Set architecture volume 2 AFARL TR-85-1041-VOL-2 F33615-79-C-1935 F/G 9/2 NL Image: Solution Set architecture volume 2 AFARL TR-85-1041-VOL-2 F33615-79-C-1935 Image: Solution Set architecture volume 2 F/G 9/2 Image: Sol



command to the UCIC which directs it to create the loaderlinker package in data value memory (simulating a CREATE NON-NESTED PACKAGE OBJECT instruction). The UCIC extracts the size of the variable global data from the package descriptor (PKG) and requests the memory manager to allocate storage in data value memory for the variable global data and fixed size administrative data (a single allocation). The memory manager returns the address of the variable global data and the UCIC sends a "package created" response to the User Console. The User Console next sends a command to elaborate the created package. The UCIC interprets this command as a request to invoke subprogram 0. The following preliminary steps are taken:

- Extract machine state information from the subprogram 0 component in the package header (a 5-word packet of information pertinent to subprogram 0, as shown in Figure 2-2). Note that the Formal Parameter Mask contains all zeros since no parameters are passed, the exception mode is ELABORATION, and the nesting depth is "1".
- Set stack index to "0" and clear Temporaries Mask.
- Load display register 0 and 15 for package global data.
- Request memory manager to allocate storage for subprogram 0's activation record and fixed size administrative data. When the base address of the activation record is known, proceed to next step.
- Load display register pair corresponding to nesting depth = 1 (local display registers for subprogram 0). These registers are loaded with the base addresses of the subprogram's activation record and automatic data template.
- The dynamic link and Status Save Flag are ignored (need not be saved in subprogram 0's administrative data area as in a normal call).

In the above, the last operation performed is to load the first instruction of subprogram 0 (gotten from the subprogram 0 component in the package header) into the program counter, starting the execution of subprogram 0. When the loader-linker package is being elaborated,

subprogram 0 must create and activate at least one task (which is put on a ready queue that corresponds to the task's priority level). All created/activated tasks are placed on ready queues; none gets scheduled to run because subprogram 0 executes at the User Console priority level highest in the system. The last instruction of subprogram O is RETURN FROM PACKAGE ELABORATION. This instruction deallocates subprogram 0's activation record/administrative data storage. It then "wipes out" the User console priority and invokes the task scheduler which schedules a task in the loader-linker package to run. This task will be the one at the head of the highest priority ready queue. (Note that there is no requirement to create/activate tasks during execution of subprogram 0 in subsequently loaded non-nested packages; when RETURN FROM PACKAGE ELABORATION is executed, previously activated tasks in other packages will compete to be scheduled to run).

Task programs and/or subprograms in the loader-linker package load other non-nested packages (which may contain nested packages) and link data and subprograms in different packages. The latter is accomplished by assigning values to pointers to data and/or non-nested subprograms and moving the the pointers to variable global data areas of Loading of any non-nested package appropriate packages. other than the loader-linker is accomplished by a program in the loader-linker package. First, the instruction, ALLOCATE PACKAGE STORAGE, is executed which requests the memory manager to allocate storage for the package in data template and instruction memories. The required storage sizes are operands of this instruction. When storage allocation is base addresses complete, the in data template and instruction memories are returned in the form of a pointer. The loader-linker program then executes an INITIATE LOAD instruction which passes a "load command" together with the pointer and the package number (of the non-nested package to be loaded) to the UCIC. ... The UCIC sends a request to the User Console to load the header of the package identified by the package number. Loading proceeds, as described earlier, under control of the User Console. When the non-nested package is loaded, the UCIC returns a "load complete" signal to the HLLM. This completes the INITIATE LOAD instruction. The loader-linker program next executes a CREATE NON-NESTED PACKAGE instruction which, after extracting the size of the package variable global data from the package descriptor, requests storage allocation in data value memory for the variable global data and fixed size administrative data (a single allocation). When the base address is returned from · manager, pointer tó the memory a

the package is generated. This pointer contains the absolute addresses of the variable global data in data template and data value memories. To elaborate the package, a call to its subprogram 0 must be programmed. First, a pointer to subprogram 0 is assigned (see Section 10.4); then, the CALL SUBPROGRAM instruction is executed in which operand S1 (see Section 6.1) addresses a pointer to a subprogram in an external package (subprogram 0 in the nonnested package being elaborated). Tasks created in subprogram 0 are made dependent on the non-nested package. (These tasks were declared in this package in the Ada program.) The last instruction in subprogram 0 is RETURN ELABORATION which deallocates the activation FROM PACKAGE record/administrative data, removes the User Console priority, and invokes the task scheduler which schedules a task to run.

Nested packages, declared in Ada tasks, subprograms, or nonnested package bodies must also be created and elaborated. This is accomplished within the respective task program, subprogram, or non-nested package (more precisely, within subprogram 0 of the non-nested package) that declared the nested package. The process of creating and elaborating a nested package is similar to that for non-nested packages, However, the CREATE NESTED PACKAGE described earlier. instruction, in addition to allocating storage for the variable global data and administrative data and returning a pointer to the created package, must convert all offsets in the nested package header to absolute addresses (see Section 7.3). The created nested package is made dependent on the task, subprogram, or non-nested package that created it. Dependency of a nested package means that reclamation of its variable global data and administrative data storage in data value memory is delayed until the destruction of the object that created the nested package. When this happens, any dependent data objects (whose access types were declared in the nested package) are also destroyed. The nested package is elaborated, as before, by calling its subprogram 0 (if it exists). When the nested package is declared in the body of a non-nested package, subprogram 0 of the non-nested package calls subprogram 0 of the nested package. Note that the User Console priority is in effect even while subprogram 0 of the nested package is executing. The completion of subprogram 0 of any nested package is marked by the normal RETURN FROM SUBPROGRAM instruction.

Tasks that are created and activated in subprogram 0 of any nested package will not get a chance to be scheduled to run until the instruction, RETURN FROM PACKAGE ELABORATION, is

executed in subprogram 0 of the non-nested package. (As stated earlier, this instruction removes the User Console priority from the system; it represents the completion of the elaboration of the enclosing non-nested package.) A task created and activated in subprogram 0 of a nested package is made dependent on the task, subprogram, or nonnested package in which the nested package was created. (Nested packages cannot be masters.)

7.1 INITIATE LOAD.

Format: 71_H, S1, S2, S3

Mnemonic: INTLD

Operands:

S1: <u>Pointer Containing Package Load Addresses</u> FMT: <u>memory (0)</u>

S2: <u>Package Number</u> FMT: <u>immediate</u> (EXT,2)

S3: <u>Delay Amount</u> FMT: <u>memory (0) or immediate (EXT,2)</u>

Function:

This instruction initiates loading of a non-nested package. A "load command" is sent to the User Console via the User Console Interface Card. Accompanying this command are the pointer addressed by Sl and the package identification number, equal to the immediate value of operand S2. Words 2 and 3 of the pointer contain the absolute (base) addresses of the storage allocated in data template and instruction memories for the package load. Simultaneously with emitting the "load command", timing of the delay specified by S3 begins. (See Section 9.4.12 for an explanation of how the delay amount is expressed.) The INITIATE LOAD instruction is not completed until a "load complete" message is received from the user Console Interface Card or the delay times out. In the latter case, a PROGRAM_ERROR exception is raised.

Exceptions: PROGRAM_ERROR 7.2 CREATE NON-NESTED PACKAGE OBJECT.

Format: 72_H, S, D

Mnemonic: CRPO

Operands:

:	Pointer	to	Loaded	Package
FMT:	memo	ry	(0)	1212010

D: <u>Pointer to Created Package</u> FMT: <u>memory (0)</u>

Function:

This instruction requests allocation of storage in data value memory for the package variable global data and fixed size administrative data. The size of the variable global data, in words, is extracted from the package descriptor (PKG) addressed by the contents of word 2 of the pointer addressed by S. When storage has been allocated, the absolute address of the variable global data in data value memory is stored in word 3 of the pointer to the created package and the absolute address of the variable global data in data template memory is stored in word 2. (The former address is the address of the allocated storage + size of administrative data and the latter address is the address of the package header + 1.) The rights assigned to the pointer are READ and WRITE. The ENTITY (ENT) field is set to 000.

Exceptions: STORAGE ERROR

7.3 CREATE NESTED PACKAGE OBJECT.

Format: 73_H, S1, S2, S3, D

Mnemonic: CRNPO

Operands:

FMT:

S1:	Pointer to	Containing	Non-Nested	Package
FMT:	memory	(0)		

S2: Offset to Nested Package Component FMT: immediate (EXT,2)

S3: Creator

immediate (EXT,2) S3 specifies, via nesting depth (ADS), the task, subprogram, or non-nested package on which the created package depends.

D: <u>Pointer to Created Package</u> FMT: <u>memory (0)</u>

Function:

This instruction allocates storage in data value memory for the nested package's variable global data and administrative data. To extract the variable global data size, the package program descriptor (PPGM) in the nested package component in the containing non-nested package header must be addressed The base address of the non-nested (see Figure 2-2). package header is retrieved from word 2 of the pointer addressed by S1. The offset, given by S2, subtracted from this base is the address of the package program descriptor that contains the size, in words, of the variable global data of the nested package. Storage allocation in data value memory for the variable global data and fixed size administrative data is requested. While the memory manager is allocating storage, the offsets in the nested package header are converted to absolute addresses using the absolute addresses in the 5-word nested package program component. The word at an offset of -1 from the package program descriptor contains the absolute (base) address of the nested package header. Each 5-word program component in the nested header contains an offset to the program's automatic data template. Each offset is added to the base address of the nested package header; the resulting absolute addresses are written into the components of the nested package header. Similarly, the word at an offset of -3 from the package program descriptor contains the absolute (base) address of the nested package instructions. This is added to the offset to the first instruction and the offset to the

last instruction in each program component of the nested package header; again, the resulting absolute addresses are written into the components of the nested package header.

When the address of the allocated storage is returned by the memory manager, the pointer to the created nested package, addressed by D, can be formed. The absolute address of the variable global data in data value memory is stored in word 3 of the pointer and the absolute address of the variable global data in data template memory is stored in word 2. (The former address is the address of the allocated storage + size of administrative data and the latter address is the address of the nested package header + 1). The rights assigned to the pointer are READ and WRITE. The ENTITY (ENT) field is set to 000.

S3 specifies the task or subprogram in the non-nested package or the non-nested package itself on which the created nested package depends. This dependency is specified by the nesting depth (or address space, ADS) of the task; subprogram, or non-nested package. Nesting depth designates the display register pair which contains the base addresses of the task, subprogram, or non-nested package in data template memory and data value memory. (For tasks and subprograms, this would be the addresses of the activation record and corresponding automatic data template; for the non-nested package that corresponds to a nesting depth of 0, this would be the addresses of the variable global data and variable global data template).

Exceptions: STORAGE ERROR

7.4 ALLOCATE PACKAGE STORAGE.

Format: 74_{H} , S1, S2, D

Mnemonic: ALLOCP

Operands:

- S1: Size of Allocation in Data Template Memory FMT: memory (0) or immediate (EXT,2)
- S2: Size of Allocation in Instruction Memory FMT: memory (0) or immediate (EXT,2)
- D: <u>Pointer to Allocated Package</u> FMT: <u>memory (0)</u>

Function:

This instruction requests allocation of storage in data template memory and instruction memory for the non-nested package to be subsequently loaded. S1 and S2 address or directly specify the sizes, in words, required for storage allocation of the package in data template memory and instruction memory, respectively. Note that immediate values restrict the allocation size to 220 word whereas memory operands have an upper limit of 232 words. The base addresses in data template memory and instruction memory returned by the memory manager are stored in words 2 and 3, respectively, of the pointer addressed by D. The ENTITY (ENT) field is set to 110; other fields in word 1 of the pointer are ignored.

Exception: STORAGE_ERROR

7.5 RETURN FROM PACKAGE ELABORATION.

Format: 75_H

Mnemonic: RETPE

Operands: None

Function:

This instruction marks the completion of subprogram 0 of a non-nested package. The activation record and administrative data of the subprogram are deallocated immediately since no objects are dependent on it. The User Console priority is removed from the system and the task scheduler is invoked.

Exceptions: None

8 DYNAMIC STORAGE ALLOCATION/DEALLOCATION

Data objects are allocated space in data value memory at run time. Their creation corresponds to the evaluation of allocators in Ada when the objects created are any type other than a task. (Evaluation of an allocator that creates a task object is supported with the EVALUATE ALLOCATED TASK OBJECT instruction, described in Section 9.4.6.) Access values, returned when an allocator is evaluated in Ada, are represented in the HLLM by pointers. Data templates for data objects are placed in the enclosing package's constant global data area.

Deallocation of space in data value memory normally takes place when the subprogram, task, or package in which the access type was declared is destroyed (per the Ada CONTROLLED pragma). The data object is said to be dependent on the subprogram, task object, or package object. Data objects can be explicitly destroyed (storage in data value memory reclaimed) when the instantiated generic library procedure, UNCHECKED DEALLOCATION, is called at the Ada Dangling references caused bv level. program UNCHECKED DEALLOCATION are detected in the HLLM. This is accomplished by creating a data object with the instruction, CREATE UNCHECKED DATA OBJECT, that assigns a 24-bit unique name to the data object, stores it into the pointer, and sets the unique name flag (see pointer format in Section 3.4). A unique name will not be reassigned until 224 different names have been assigned to data objects that are to be explicitly destroyed. When assigned, a unique name is stored in a system-wide Unique Name Table; when the pointedto data object is destroyed (via the DESTROY DATA OBJECT instruction), its unique name is deleted from the table, Any reference via a never, in principle, to reappear. pointer to a data object in which the unique name flag is set requires a check for the existence of the unique name in If the unique name is not in the table, a the table. CONSTRAINT ERROR exception is raised.

The template of a data object can specify any supported data type except pointers and formal reference parameters; arrays and records should be most common. Arrays may be constrained or unconstrained (see Appendix B). Data objects may be used as I/O buffers.

8.1 CREATE DATA OBJECT.

Format: 76_H, S1, S2, D

Mnemonic: CRDO

Operands:

S1: <u>Data Object Type (Data Template)</u> FMT: <u>memory (0)</u>

S2: Object on which Data Object Depends FMT: immediate (EXT,2) or memory (0)

Immediate: S2 specifies, via address space (ADS), the subprogram, task object, or enclosing package object on which the data object depends.

- Memory: S2 addresses a pointer to a subprogram or task object in an external package or to an external package object on which the data object depends.
- D: <u>Pointer to Created Data Object</u> FMT: <u>memory (0)</u>

Function:

This instruction allocates storage in data value memory for the data object and returns a pointer to the data object. Sl is the address of the data object template in the constant global data area of the enclosing package; hence, the address space (ADS) must be equal to 15. The first word in the template is a Data Object Descriptor (see Section 3.8) which specifies the total size of storage in data value memory to be allocated for the data object when the data object is not an unconstrained array or a record with one or more unconstrained array components. If the data object is an unconstrained array or a record with unconstrained array components, index constraints (which supply the array bounds) follow the CREATE DATA OBJECT instruction in the The order of unconstrained bounds in instruction stream. the data object description corresponds to the order of the index constraints (see Section 4.4.5) in the instruction stream. When array bounds are unconstrained, the total size of the data object in data value memory is computed as described in Section 3.8.2 and illustrated in example 4 of When storage has been allocated (two Appendix B. allocations for unconstrained arrays), the pointer addressed by D is assigned values as follows:

Word 1 unique name flag <= 0 ENT <= 010 RIGHTS <= read, write. Word 2

Receives absolute address of data object template (address of Data Object Descriptor).

Word 3

Receives absolute address of data object values (address in data value memory that corresponds to the Data Object Descriptor in data template memory).

S2 specifies the subprogram (via its activation record), task object, or package object on which the lifetime of the data object depends. It will be called a "dependee" in this instruction. If S2 is an immediate operand, its value is the address space (nesting depth) of the dependee. The display register pair corresponding to this nesting depth contains the dependee's absolute (base) addresses in data value and data template memories. If S2 addresses a pointer to the dependee, the base addresses are contained in the pointer. Having the address of the administrative data of the dependee, a doubly linked list of dependent data objects originating at the dependee can be maintained as follows:

- (a) In the dependee's administrative data is a link to the former most recent data object. This link is moved to the created data object and the address of the administrative data of the created data object replaces the link in the dependee.
- (b) The address of the created data object is also stored in the administrative data area of the former most recent data object and the latter's address is stored in the administrative data area of the created data object.
- (c) If there is no "former most recent data object," the link in the dependee will contain a NULL. Then, the address of the created data object replaces the NULL link and the address of the dependee is stored in the administrative data area of the created data object.

When a dependee is destroyed (storage in data value memory reclaimed), all data objects in the linked list are deallocated in data value memory. Note that the links are in place to support explicit destruction (via the DESTROY DATA OBJECT instruction) of a data object at any location in the linked list.

Exceptions: STORAGE ERROR

8.2 CREATE UNCHECKED DATA · OBJECT.

Format: 77_H, S1, S2, D

Mnemonic: CRUNDO

Operands:

S1: <u>Data Object Type (Data Template)</u> FMT: <u>memory (0)</u>

S2: Object on which Data Object Depends

FMT: immediate (EXT,2) or memory (0) Immediate: S2 specifies, via address space (ADS), the subprogram, task object, or enclosing package object on which the data object depends. Memory: S2 address a pointer to a subprogram or task object in an external package or to an external package object on which the data object depends.

D: <u>Pointer to Created Data Object</u> FMT: <u>memory (0)</u>

Function:

This instruction performs the same function as CREATE DATA OBJECT and, in addition, assigns a unique name to the data object for the purpose of detecting dangling references that may result if the created data object is explicitly destroyed (allowed when the UNCHECKED DEALLOCATION library procedure is called). Prior to assigning values to word 1 of the pointer addressed by D, a request is made to the memory manager to assign a unique name to the data object. Then, word 1 of the pointer is assigned values as follows:

> unique name flag <= 1 ENT <= 010 RIGHTS <= read, write, destroy bits 0..23 <= unique name.

Exceptions: STORAGE ERROR

8.3 DESTROY DATA OBJECT.

Format: 78_H, S

Mnemonic: DSTROY

Operands:

S: <u>Pointer to Data Object to be Destroyed</u> FMT: <u>memory (0)</u>

Function:

The data object pointed-to by the pointer addressed by S is destroyed, i.e., storage occupied in data value memory (values and administrative data) is reclaimed. The pointer must have DESTROY authority. The data object is removed from the list of data objects dependent on some task object, subprogram, or package object. The unique name assigned to the data object (found in word 1 of the pointer) is deleted from the unique name table.

Exceptions: CONSTRAINT ERROR

9. Tasks

The support for tasking provided in the HLLM is firmly based on the concept of tasking described in Section 9 of MIL-STD-1815A. When a task object is created, it can spawn a dynamically linked chain of activation records through successive subprogram calls. Any subprogram can create new tasks which spawn other dynamically linked chains of activation records, and so forth. In addition, any task can create other tasks each of which may parent a dynamic chain. The resulting structure is a cactus stack of task and subprogram activations. The rules of tasking specify certain lifetime dependencies of tasks on other tasks and subprograms within the cactus stack and on library packages. The manner in which the HLLM supports task dependencies is described in Appendix C.

To assist in the understanding of the tasking instructions, the following important terms are defined in accordance with their use in this ISA:

- TASK OBJECT A task object is a created storage object comprising the entries, data values (template and activation record), and instructions of the task unit.
- TASK PROGRAM A task program is the instructions compiled from the task body statements.
- COMPLETED TASK A COMPLETED task is one that has executed all its instructions (end of task program reached) or one in which a TASKING ERROR was raised during its activation. Although no instructions are executed in a COMPLETED task, the task's local data is still accessible to other tasks and subprograms.
- SUSPENDED TASK A SUSPENDED task is one that has been activated but is neither executing instructions nor ready to be scheduled; it is temporarily blocked on an entry queue or at an accept, delay, or wait instruction. SUSPENDED tasks are normally resumed when the suspending condition is removed.
- ABNORMAL TASK An ABNORMAL task is one that has been aborted. Although an ABNORMAL task is allowed to execute until it reaches a synchronization point, it will immediately become a COMPLETED task in the HLLM and will be TERMINATED when dependency conditions permit.

- TERMINATED TASK TERMINATED tasks are destroyed with storage reclaimed in the HLLM. COMPLETED tasks with all termination conditions met are TERMINATED.
- CUSTOMER TASK A customer task is one that partakes in a rendezvous by calling an entry of another (server) task.
- SERVER TASK A server task is one that partakes in a rendezvous by accepting an entry call of another (customer) task.
- DEPENDENT TASK A dependent task is one that affects the lifetime (activation to termination) of another task designated as its master and, under certain conditions, whose own lifetime is affected by the state of the master and its dependent tasks. In the latter case, the master could be a subprogram or library package as well as a task.
- MASTER A master may be a task, a subprogram (which includes a block statement in the HLLM implementation), or a library package. A completed master is not destroyed until all its dependent tasks are TERMINATED.
- ACCEPT BODY An ACCEPT body is a group of instructions that immediately follows an ACCEPT ENTRY or SELECT ACCEPT instruction and is terminated with the END RENDEZVOUS instruction. The ACCEPT body corresponds to the sequence of statements that follows an ACCEPT statement in Ada.

Refer to the Glossary and to MIL-STD-1815A for further definitions and explanations.

the law way is a strange of the low the second of the second second second second second second second second s

9.1 Task Scheduling. The HLLM task scheduler deals with two categories of tasks:

1.RUNNING TASK- A RUNNING task is one that has been activated (created and started) and is executing on a processor.

2.READY TASK - A READY task is one that has been activated (created and started), is neither SUSPENDED nor COMPLETED, and is waiting on a ready queue to be scheduled for execution.

When tasks compete for access to a processor, the scheduling algorithm ensures that no READY task is waiting to be scheduled while a lower priority task is RUNNING. Each task waiting to be scheduled is put on a ready queue corresponding to the task's priority level. When two or more tasks of the same priority are ready to be scheduled, the order of scheduling is FIFO. In this case, when the scheduler is invoked, the task at the head of the highest priority ready queue is scheduled for a specified execution time period and the executing task is moved to the tail of the ready queue. When the time quantum expires, an interrupt is generated that again invokes the scheduler. If an executing task becomes SUSPENDED or COMPLETED or if a higher priority task becomes READY before this interrupt occurs, a new task is scheduled, resetting the execution time quantum. A different time quantum may be specified for each ready queue. One choice is "infinite" duration. If a task scheduled with "infinite" duration is at the highest priority level, only suspension or completion of the task will cause it to relinquish the processor. (However, if a lower priority task is scheduled with an "infinite" time quantum, it could get "bumped" if a SUSPENDED higher priority task becomes READY.) Tasks not assigned any priority are relegated to the lowest priority level (priority 0). When a task is activated or when some event removes a task from a state of suspension, the task becomes READY and is appended to the tail of its ready queue. If, however, it is the only task at the highest priority level, it will be scheduled to run with a fresh execution time quantum. Tasks in rendezvous assume a priority which is the higher of that of the customer and server tasks. Since a customer task in rendezvous is SUSPENDED, its priority level need not be modified if lower than that of the server task. However, if the server has the lower priority, its priority is raised to that of the customer for the duration of the rendezvous ACCEPT body.
9.2 Task Switching. Tasks exist in one of five states: RUNNING, READY, COMPLETED, SUSPENDED, or TERMINATED. A scheduling decision is made whenever any task changes state. Following is a list of actions that cause a change of state, hence invocation of the task scheduler:

- (a) a task is activated (through execution of the END ACTIVATION or END ELABORATION AND ACTIVATION instruction) and becomes READY or RUNNING.
- (b) a task is suspended (when, for example, a task calls an entry or tries to accept an entry call when there is no caller) and becomes SUSPENDED (another task is scheduled to run).
- (c) a task is removed from the state of suspension (when, for example, a delay expires or a rendezvous is completed) and becomes READY or RUNNING.
- (d) a task is completed for any reason and becomes COMPLETED (another task is scheduled to run).
- (e) a task is directly terminated for any reason and becomes TERMINATED (another task is scheduled to run).

Note: A task may be directly terminated only when (1) it becomes COMPLETED and all its dependent tasks, if any, are already TERMINATED or (2) it executes a SELECT TERMINATE instruction and its master is in a COMPLETED state and all tasks that depend on that master are TERMINATED or waiting at SELECT TERMINATE instructions.

(f) an interrupt from the clock manager signals that an execution time quantum has expired (another task is scheduled to run).

In the discussion that follows, when reference is made to a "task or subprogram", the subprogram belongs to the task to which it is dynamically linked (through any number of levels); hence, the subprogram executes at that task's priority level. Priority level is a static component of the machine state for subprograms as well as tasks. Its value must be known when tasks are switched and when a rendezvous takes place (server's priority level can be affected by customer's level). The task scheduler keeps track of the address of the activation record and the priority level of the current task or subprogram. During task switching, the scheduler manages the change of environments, i.e., it changes the state of tasks, manages delay and entry queues, and saves/restores the states of switched-out and switched-in tasks and subprograms. In particular, if a task program is executing when the scheduler switches tasks, only the dynamic components of the switched-out task's machine state (stack index, exception mode, registers corresponding to "ls" in the Temporaries and Valid Parameter Masks, and execution resumption address) are saved in the task's administrative data area. The static components of the state (nesting depth of task program, priority level, display registers of nesting depths < = nesting depth of task program, and addresses of first and last instructions of the task program) are not saved since they are stored in the task's administrative data area when the task is created. If a subprogram is executing when tasks are switched, the above listed static components of the subprogram's machine state are saved in the administrative data area of the subprogram if the Static Save Flag (see Section 6.1) is "0". The Static Save Flag is set to "1" when these components are first saved and, thereafter, the static components are not saved in the presence of a task switch. The above listed dynamic components are always saved for the switched-out subprogram. Further, all components of the machine state (static and dynamic) are restored for the switched-in task or subprogram.

less dies an Cal

9.3 Exception Modes. Tasks and subprograms exist in one of four exception modes, managed by the machine. Errors are handled differently in each mode as described below:

ELABORATION mode - This corresponds to the elaboration of a declarative part in Ada. An error causes termination of all created but not yet activated subcomponent tasks and abandonment of the executing task or subprogram. If the executing program is a task, it is marked as COMPLETED and a TASKING ERROR exception is raised at the point of its activation. If the executing program is a subprogram, the same error exception is raised at the point of call. (If the subprogram is a main program, it is abandoned without error propagation).

ACTIVATION mode - This corresponds to the parallel activation of tasks that are subcomponents of declared objects in Ada. (In the HLLM, tasks are activated sequentially.) An error causes the task being activated to be marked as COMPLETED. Activation of other subcomponent tasks (successfully or not) is not affected. When each of these tasks has been activated (or marked as COMPLETED), a TASKING_ERROR is raised in the environment of the task or subprogram whose declarative part is being elaborated, i.e., the task or subprogram that is executing the ACTIVATE TASK instructions.

ACCEPT BODY mode - This corresponds to the sequence of statements following the ACCEPT statement in Ada. While executing an ACCEPT body, an error that is not handled by a local exception handler causes the following to occur:

- (a) ACCEPT body is abandoned.
- (b) the same exception is raised again in the server task's environment at a point immediately following the ACCEPT ENTRY or SELECT ACCEPT instruction.
- (c) the customer task is marked with TASKING ERROR pending and its state is changed from SUSPENDED (in rendezvous) to READY; when it next becomes RUNNING, a TASKING_ERROR is raised at the point of entry call.

If the ACCEPT body is executing when the server task is destroyed (aborted), the server task is marked as COMPLETED and the customer task is marked with TASKING ERROR pending (raised when the customer task is again RUNNING, as described in (c) above).

NORMAL mode - This mode corresponds to the statement between "begin" and "end" in Ada. The handling of exceptions in the NORMAL mode is described in Section 11 on Exceptions and, in a few special cases, is included in the description of an instruction. Error modes are entered as the result of executing certain instructions, as explained below:

- (a) When a subprogram or task program executes a CALL instruction, the called subprogram enters the ELABORATION mode or NORMAL mode, depending on whether or not the called subprogram has a declarative part that requires creation of objects and/or subcomputations for initialization of declared objects. Within the area of the package header that corresponds to the subprogram is a bit that specifies which of these two modes is entered.
- (b) When a subprogram or task program executes an END ELABORATE instruction, that subprogram or task enters the ACTIVATION mode.
- (c) When a subprogram or task program executes an ACTIVATE TASK instruction, the named task enters the ELABORATION mode.
- (d) When a subprogram or task program executes an EVALUATE ALLOCATED TASK instruction, the named task, when created, enters the ELABORATION mode.
- (e) When a subprogram or task program executes an END ACTIVATE or END ELABORATE AND ACTIVATE instruction, that subprogram or task enters the NORMAL mode.
- (f) When a server task program executes an ACCEPT ENTRY or SELECT ACCEPT instruction and a customer task is queued on the accepted entry, the exception mode is changed to ACCEPT BODY for the duration of the ACCEPT body.

3.4 Tasking Instructions. The group of instructions described on the following pages supports Ada tasking as outlined in Sections 9.0 through 9.4. These instructions implement task creation, activation, rendezvous, and termination.

And Date of the second s

9.4.1 CREATE TASK OBJECT.

Format: 79_H, S1, S2, D

Mnemonic: CRTO

Operands:

Sl: Tas	k Program Identification
FMT:	immediate (EXT, 2) or memory (0)
Immediate:	Sl specifies an offset to a task program component in the local package header.
Memory:	Sl addresses a pointer to a program (task program) in an external package.

S2: Master

FMT:	immediate (EXT,2) or memory (0)
Immediate:	S2 specifies a local master via its nesting depth (ADS).
Memory:	S2 addresses a pointer (no specific authority required) to an external package master (library package).

D: FMT:

Pointer to Created Task Object memory (0) D addresses a pointer in the lo

D addresses a pointer in the local package that is assigned to point to the created task object (ENT = 100).

inction: is instruction creates (but does not activate) a task object. he offset to the task program component (S1) is an immediate the pointer to the external task alue or is contained in (this pointer must have READ authority for the task cogram. The offset is subtracted from the base address of the cogram.) ader of the local or external package, the former derived from isplay register 0, the latter retrieved from the pointer to the ask program. The result is the address of a 5-word packet of iformation pertinent to the task, from which the size of the ask's activation record, the absolute address of the task's stomatic data template, the task's nesting depth, priority evel, and number of entries, and the addresses of the first and ast instructions of the task program are retrieved. Space for ne activation record and the administrative data area are llocated in data value memory. The size of administrative data rea is a fixed value + 16*number of entries, thus allowing a aximum of 16 customer tasks to be queued on each entry. All the lantities retrieved from the package header (except the first vo listed) are now saved in the created task's administrative ita area for subsequent easy access. In addition, the following isplay registers are saved, completing the state of the created ask:

(a) when the task program is in the local package

- all display registers corresponding to nesting depths
 < nesting depth of created task.
- local display register pair corresponding to created task's nesting depth (address of task's activation record and address of task's automatic data template).
- (b) when the task program is in an external package (hence, at nesting depth = 1)
 - display register 0 of external package (addresses contained in display register pair gotten from words 2 and 3 of the pointer to the task program).

- display register 15 of external package (address contained in display register = contents of word 2 of the pointer + size of variable global data + 1).
- local display register pair corresponding to nesting depth = 1 (address of task's activation record and address of task's automatic data template).

S2 specifies the package, task, or subprogram that is the created task's master. It may be any enclosing activation, the enclosing package, or an external (library) package. The number of entries, N_E, restricts any rendezvous with this task to entry numbers $0..N_E-1$. The pointer addressed by D is given READ, WRITE, and DESTROY authorities. The absolute addresses of the task's automatic data template and activation record are loaded into words 2 and 3 of the pointer, respectively. This is now the pointer to the created task object (ENT = 001).

an exception raised during this instruction (the If is ELABORATION mode is extant), creation of this task is abandoned and all tasks previously created but not yet activated during this elaboration are terminated. Hence, a chain of tasks must be maintained as tasks are successively created. When a task is created, a link to the previously created task (null, if no previous task) is retrieved from a location, L, in the administrative data area of the task or subprogram being elaborated and stored in the created task's administrative data area. A link to the newly created task is stored in L. This process is repeated for all created tasks (illustrated below for This three tasks).



Any instruction executed in the ELABORATION exception mode has access to L so that, in the event of an error, all linked tasks can be located and terminated. Further, an error in the ELABORATION mode causes a return to the point of call if a subprogram is executing or completion of the task and a return to its point of activation if a task program is executing. In the former case, the exception is raised in the caller's environment while in the latter case, a TASKING ERROR pending condition is set at the point of activation of the task. (For continuity of discussion, see the description of errors in the ACTIVATE TASK instruction.)

Award a break of the share of

Exceptions: PROGRAM_ERROR STORAGE_ERROR

9.4.2 ACTIVATE TASK.

Format: 7A_H, D

Mnemonic: ACTV

Operands:

D: <u>Pointer to Task Being Activated</u> FMT: <u>memory (0)</u> D addresses a pointer to the task being activated.

Function:

This instruction causes the activation of a previously created task by executing what corresponds to the declarative part of the task body in Ada. Prior to transferring program control to the created task's task program for activation, the following administrative steps are performed:

• A link to the administrative data area of the activating current task or subprogram is stored in the administrative data area of the created task; this allows a return to the environment of the activating task or subprogram when the created task has been successfully activated or if an error occurs during its activation.

• If a task is executing (activating the pointed-to task), the executing task's dynamic machine state components are saved in the executing task's administrative data area. If a subprogram is executing, the dynamic components are saved in the subprogram's administrative data area; the static components of the machine state are saved only if the Static Save Flag is "0" (see Section 9.2). These actions permit proper resumption of execution of the activating task or subprogram when the created task has been successfully activated or if an error occurs during its activation.

The initial machine state for the task to be activated is next This involves retrieving information stored in the established. pointed-to task's administrative data area when the task was created (see CREATE TASK OBJECT instruction. Section 9.4.1). Included are the task priority level, nesting depth, display registers corresponding to nesting depths < = nesting depth of task to be activated, and the addresses of the first and last instructions of the task program. Further, the Temporaries and Valid Parameter Masks (register 0) are cleared, the stack index is set to zero, and the exception mode of the task is set to ELABORATION. This completes the transfer of program control to the task to be activated. The task program starts with a group of instructions that activates (elaborates the declarative part of) the created task. The instruction, END ACTIVATION or END ELABORATION AND ACTIVATION, returns control to the activating task or subprogram.

If a return to the activating task or subprogram is made because an error occurred during the activation of the created task (or if an error occurs during the execution of the ACTIVATE TASK instruction prior to transfer of control to the task program for activation), the created task becomes COMPLETED and TASKING ERROR pending is set in the activating task or subprogram. When all the created subcomponent tasks have been activated, the END ACTIVATION instruction is executed. The ensuing action depends on whether a task or subprogram is executing and is described in the END ACTIVATION instruction (see Section 9.4.4).

Exceptions: PROGRAM ERROR CONSTRAINT ERROR

END ELABORATION. 9.4.3 .

7BH Format:

Mnemonic: NELAB

Operands: None

Function:

This instruction marks the end of elaboration of the declarative part of a task or subprogram. The exception mode is changed from ELABORATION to ACTIVATION. Subcomponent tasks created during the elaboration (with no errors) are now ready to be activated.

EN EL COLOR

.

North States

...

Exceptions: None

9.4.4 END ACTIVATION.

Format: 7C_H

Mnemonic: NACTV

Operands: None

Function:

This instruction marks the end of a series of one or more ACTIVATE TASK instructions. If TASKING ERROR pending in the executing subprogram or task is not true (no errors), the instruction proceeds as follows:

(a) If a subprogram is executing, its exception mode is changed from ACTIVATION to NORMAL and execution continues.

(b) If a task program is executing, the task has been properly activated. The exception mode is changed from ACTIVATION to NORMAL and the dynamic components of the machine state (see Section 9.2) are saved in the activated task's administrative data area, allowing proper resumption of execution when the activated task is scheduled to run. Prior to invoking the scheduler, a few additional administrative functions are performed to return control to the task or subprogram that activated this task. The link to the administrative data area of the task or subprogram that activated this task (saved in the activated task's administrative data area by the ACTIVATE TASK or the EVALUATE ALLOCATED TASK instruction) is retrieved and used, in turn, to retrieve the machine state (dynamic and static components) of the task or subprogram (see Section 9.2). The scheduler is then invoked.

Note: The scheduler will assign the READY state to the activated task and append it to the tail of the ready queue that corresponds to its priority level. On the queue, the task is identified by the address of its administrative data area.

If TASKING_ERROR pending is true (an error occurred during the activation of a subcomponent task), then a TASKING_ERROR is raised if a subprogram is executing. If a task is executing (call it task A), it is COMPLETED since an error occurred during

the elaboration of its declarative part. A return is made to task A's point of activation in another task program or subprogram (call it program B). The return is effected by restoring the state of program B, known via the link to its administrative data area (that had been saved in the administrative data area of task A when task A was activated). TASKING ERROR pending is set in program B; then, other subcomponent tasks of program B, if any, are activated followed by END ACTIVATION, etc.

and the second of the second second

MIZ SHILL -

Exceptions: PROGRAM ERROR

9.4.5 END ELABORATION AND ACTIVATION.

Format: 7D_H

Mnemonic: NELACT

Operands: None

Function:

This instruction is used at the end of the elaboration of the declarative part of a task or subprogram when no subcomponent tasks were created, i.e., when there is no activation of subcomponent tasks. Hence, the end of elaboration and activation are coincident. The instruction proceeds as follows:

(a) If a subprogram is executing, the exception mode is changed from ELABORATION to NORMAL and execution continues.

(b) If a task is executing, it has been properly activated. The exception mode is changed from ELABORATION to NORMAL and the dynamic components of the machine state (see Section 9.2) are saved in the activated task's administrative data area, allowing proper resumption of execution when the activated task is scheduled to run. Prior to invoking the scheduler, a few additional administrative functions are performed to return control to the task or subprogram that activated this task. The link to the administrative data area of the task or subprogram that activated this task (saved in this task's administrative data area by the ACTIVATE TASK or EVALUATE ALLOCATED TASK instruction) is retrieved and used, in turn, to retrieve the machine state (dynamic and static components) of the task or subprogram (see Section 9.2).

Note: The scheduler will assign the READY state to the activated task and append it to the tail of the ready queue that corresponds to its priority level. On the queue, the task is identified by the address of its administrative data area.

Exceptions: PROGRAM ERROR

9-19

· .

9.4.6 EVALUATE ALLOCATED TASK OBJECT.

Format: $7E_H$, S1, S2, D

Mnemonic: EVALTO

Operands:

S1: Task Program Identification

FMT: immediate (EXT,2) or memory (0)

Immediate: Sl specifies an offset to a task program component in the local package header. Memory: Sl addresses a pointer to a program (task program) in an external package.

S2: Master

- FMT: immediate (EXT,2) or memory (0) Immediate: S2 specifies a local master via its nesting depth (ADS).
- Memory: S2 addresses a pointer (no specific authority required) to an external package master (library package).

D: <u>Pointer to Created Task Object</u> FMT: <u>memory (0)</u> D addresses a pointer in the local package that is assigned to point to the created task object

Function:

This instruction creates and causes the activation of a task object. The offset to the task program component (S1) is an immediate value or is contained in the pointer to the external task program. (This pointer must have READ authority for the task program.) The offset is subtracted from the base address of the header of the local or external package, the former derived from display register 0, the latter retrieved from the pointer to the task program. The result is the address of a 5-word packet of information pertinent to the task, from which the size of the task's activation record, the absolute address of the task's automatic data template, the addresses of the first and last instructions of the task program, and the task's nesting depth, priority level, and number of entries are retrieved. Space for activation record and the administrative data area are the allocated in data value memory. The size of the administrative . data area is a fixed value + 16* number of entries, thus allowing a maximum of 16 customer tasks to be queued on each entry. All the quantities retrieved from the package header (except the now two listed) are saved first in the created task's administrative data area for easy access when the task is scheduled. In addition, the display register environment of the

created task's task program is established and saved as described in detail for the CREATE TASK OBJECT instruction (see Section 9.4.1). S2 specifies the package, task, or subprogram that is the created task's master. It may be any enclosing activation, the enclosing package, or an external (library) package. The number of entries, N_E, restricts any rendezvous with this task to entry numbers $0..N_{E}$ -1. The pointer addressed by D is given READ, WRITE, and DESTROY authorities. The absolute addresses of the task's automatic data template and activation record are loaded into words 2 and 3 of the pointer, respectively. This is now the pointer to the created task object (ENT = 101).

Following these actions associated with creating the task, the task is activated by executing what corresponds to the declarative part of the task body in Ada. Prior to transferring program control to the created task's task program for activation, the following administrative steps are performed:

• A link to the administrative data area of the activating task or subprogram is stored in the administrative data area of the created task; this allows a return to the environment of the activating task or subprogram when the created task has been successfully activated or if an error occurs during its activation.

• If a task is executing, the dynamic machine state components are saved in the executing task's administrative data area. If a subprogram is executing, the dynamic components of the machine state are saved in the subprogram's administrative data area; the static components are saved only if the Static Save Flag is "0" (see Section 9.2). These actions permit proper resumption of execution of the activating task or subprogram when the created task has been successfully activated.

The initial machine state of the task to be activated is next established. Since the quantities retrieved from the package header are available (i.e., do not have to be read from the administrative data area as in the ACTIVATE TASK instruction) and the display register environment was established earlier, all that remains to be done to complete the transfer of program control to the task being activated is to clear the Temporaries and Valid Parameter Masks (register 0), set the stack index to zero, and set the execution mode of the created task to ELABORATION. The task program starts with a group of instructions that activates (elaborates the declarative part of) The instruction, END ACTIVATION or END the created task. ELABORATION and ACTIVATION, returns control to the activating task or subprogram.

Response to an error depends on the exception mode existing when the EVALUATE ALLOCATED TASK instruction is executed and on whether this instruction is contained in a task program or a subprogram.

(a) ELABORATION mode, subprogram executing

If an error occurs while the task is being created, creation is abandoned and all tasks previously created during this elaboration but not yet activated are terminated. Hence, this instruction (as any instruction executing in the ELABORATION mode) has access to the chain of tasks created but not activated, as described in the CREATE TASK OBJECT instruction. The subprogram returns to its point of call where the exception is raised.

If the task was successfully created but a return from its activation is made with a TASKING_ERROR pending, the created task becomes COMPLETED, all tasks previously created during this elaboration but not yet activated are terminated as discussed above, and the subprogram returns with a TASKING_ERROR raised at the point of call.

(b) ELABORATION mode, task executing Handling of errors is the same as (a) except that the executing task is marked as COMPLETED and a TASKING_ERROR pending condition is returned to the executing task's point of activation.

(c) NORMAL mode Response to errors again depends on whether a subprogram or task program is executing as described in Section 11 on Exceptions.

Exceptions: PROGRAM_ERROR STORAGE_ERROR

9-22

9.4.7 CALL ENTRY.

Format: 7F_H, S1, S2, S3,...

Mnemonic:

Operands:

- S1: <u>Pointer to Server Task</u> FMT: <u>memory (0)</u> S1 addresses a pointer to the server task.
- S2: <u>Entry Number of Server Task</u> FMT: <u>memory (0) or immediate (EXT,2)</u>

Note: If no parameters are passed via memory transfer, then no additional operands are present in this instruction.

S3,...: Actual Parameters FMT: memory (0) or immediate (EXT,2)

Note: Any number of parameters may be passed via memory transfer. Any two may be combined in a 2-operand compact format.

Function:

This instruction calls the entry, given by S2, of the server task identified by the pointer addressed by S1. The pointer must have READ authority for the server task. The number of entries, N_E, is retrieved from the server task's administrative data area and the following condition must be met (else a CONSTRAINT_ERROR is raised):

 $0 \le S_2 \le N_E - 1$.

As with the CALL instruction, CALL ENTRY is processed only up to the actual parameter operands; thus, the execution resumption address (value in the program counter) that is saved in the customer task's administrative data area when the customer is SUSPENDED addresses the word following operand S2 (entry number of server task). Parameters that are to be passed by memory are bound during execution of the BIND PARAMETERS instruction (first instruction of the ACCEPT body) which requires access to both actual and formal parameters. BIND PARAMETERS completes the processing of the operands in the CALL ENTRY instruction. arameters that are passed by registers are loaded into parameter egisters (16..31) at some points during execution of the Istomer task program. During CALL ENTRY, the registers becified in the Valid Parameter Mask (see Section 6.2.1) are aved (as part of the dynamic machine state) in the customer isk's administrative data area. These values are restored in he registers by the server task when the instruction, ACCEPT VTRY or SELECT ACCEPT, is executed provided that a customer task s queued on the accepted entry.

ext, actions (a) and (b), actions (a) and (c), or action (d) akes place with the assistance of the task scheduler:

a) The customer (executing) task's state is changed from RUNNING > SUSPENDED and the customer task is placed at the tail of the ntry queue for entry S2 of the pointed-to task. All dynamic omponents of the machine state are saved in the administrative ata area of the queued customer task. (Note that if a ubprogram had executed the CALL ENTRY instruction, the static omponents of the state would also be saved if the Static Save lag is "zero".)

b) If the server task is SUSPENDED and marked as waiting for a all of entry S2 (it had executed an ACCEPT ENTRY instruction but o task had called that entry or it had executed a WAIT nstruction after being marked during a SELECT ACCEPT instruction s waiting for a call of entry S2), its state is changed to READY nd it is placed at the tail of the ready queue corresponding to he higher priority of the customer and server tasks. If an Ada elective Wait statement had been programmed, several SELECT CCEPT instructions could have been executed prior to WAIT. In rder for the server to resume execution at the proper SELECT CCEPT instruction (when the server is next scheduled to run), he SELECT ACCEPT instruction must save its own address in a edicated location in the server's administrative data area. ach such location corresponds to the accepted entry number. During execution of ACCEPT ENTRY, the address of the ACCEPT NTRY instruction is also saved in a location corresponding to he entry number.) Then, during execution of CALL ENTRY, after t is determined that the server task is marked as waiting for a all of this entry (S2), the address of SELECT ACCEPT (or ACCEPT NTRY) that corresponds to S2 is transferred to the location that ontains the "execution resumption address" of the server task. his ensures that when the server task is scheduled to run, it the proper SELECT ACCEPT (or ACCEPT ill resume execution at NTRY) instruction and a rendezvous will successfully begin. The erver task ceases to be marked as waiting for any entry to be alled or delay to expire. Then, all subsequent calls are placed n the proper entry queue without changing the server task's xecution resumption address.

Note: When the server task is scheduled to run, the rendezvous will proceed (parameters passed to ACCEPT body which is then executed).

(c) If the server is READY and is marked as waiting for a call of entry S2 (it had executed a SELECT ACCEPT instruction but no customer task had called the entry), the address of SELECT ACCEPT is moved to the location that contains the server's execution resumption address as described above and the server task ceases to be marked as waiting for any entry to be called or delay to expire. The server task is moved to the ready queue corresponding to the customer task's priority if that priority level is higher than the server task's level. The rendezvous proceeds as described in the Note under (b) above.

(d) If the server task is not SUSPENDED on entry S2 nor READY and marked as waiting for a call of entry S2, only the actions described in (a) above take place.

If the server task is COMPLETED when one of its entries is called, a TASKING ERROR is raised in the customer task at the point of call. If customer tasks are SUSPENDED on entry queues of a server task which becomes COMPLETED before accepting any call, the task scheduler removes the customer tasks from the In each of these queues and changes their state to READY. customer tasks, TASKING ERROR pending is set. (The exception is raised as each task is scheduled to run.) If an exception is raised while the ACCEPT body is executing, the local exception handler is entered. If, however, no local handler is defined for the ACCEPT body, the exception is raised in the server task following the ACCEPT ENTRY or SELECT ACCEPT instruction. Further, the scheduler is invoked which changes the customer task's state to READY. TASKING ERROR pending is set. If the customer task is SUSPENDED in rendezvous (ACCEPT body executing) when the server task becomes abnormally COMPLETED, the scheduler changes the state of the customer task to READY and TASKING_ERROR pending is set in the customer task. If the customer task is SUSPENDED on an entry queue when it becomes ABNORMAL, the scheduler removes it from the queue and immediately changes its state to COMPLETED. If, however, the customer task becomes ABNORMAL during a rendezvous, the rendezvous is finished and then the scheduler changes the customer task's state to COMPLETED.

Exceptions: PROGRAM_ERROR CONSTRAINT_ERROR TASKING_ERROR

9.4.8 CALL ENTRY CONDITIONALLY.

Format: 80_H, S1, S2, S3, S4,...

Mnemonic: CALENC

Operands:

S1: <u>Pointer to Server Task</u> FMT: memory (0) S1 addresses a pointer to the server task.

S2: <u>Entry Number of Server Task</u> FMT: <u>memory (0) or immediate (EXT,2)</u>

S3: Label

FMT: immediate (EXT,2), interpreted as a label operand

Note: If no parameters are passed via memory transfer, then no additional operands are present in this instruction.

S4,...: Actual Parameters FMT: memory (0) or immediate (EXT,2)

Note: Any number of parameters may be passed via memory transfer. Any two may be combined in a 2-operand compact format.

Function:

This instruction attempts to call the entry, given by S2, of the server task identified by the pointer addressed by S1. The pointer must have READ authority for the server task. The number of entries, N_E , is retrieved from the server task's administrative data area and the following condition must be met (else a CONSTRAINT ERROR is raised):

 $0 <= S2 <= N_E - 1$.

As with the CALL instruction, when parameters are passed via memory, CALL ENTRY CONDITIONALLY is processed only up to the actual parameter operands; thus, the execution resumption address (value in the program counter) that is saved in the customer task's administrative data area when the customer is SUSPENDED addresses the word following operand S3 (label). Parameters that are to be passed by memory are bound during execution of the BIND PARAMETERS instruction (first instruction of the ACCEPT body) which requires access to both actual and formal parameters. BIND PARAMETERS completes the processing of the operands of the CALL ENTRY CONDITIONALLY instruction.

Parameters that are passed by registers are loaded into parameter registers (16..31) at some points during execution of the customer task program. During CALL ENTRY CONDITIONALLY, the registers specified in the Valid Parameter Mask (See Section 6.2.1) are saved (as part of the dynamic machine state) in the customer task's administrative data area. These values are restored in the registers by the server task when the instruction, ACCEPT ENTRY or SELECT ACCEPT, is executed provided that a customer task is queued on the accepted entry.

Program control is immediately transferred to the absolute address of the first instruction of the customer task program + the label offset, S3, if either of the following conditions is true:

(a) One or more customer tasks are already queued (waiting for service) at entry S2 of the server task.

(b) The server task is not marked as waiting for a call of entry S2.

If neither of the above conditions is true, a rendezvous is possible. The task scheduler is invoked and actions (a) and (b) or (a) and (c) take place:

(a) The customer (executing) task's state is changed from RUNNING to SUSPENDED and the customer task is placed at the head of the entry queue for entry S2 of the pointed-to server. All dynamic components of the machine state are saved in the administrative data area of the queued customer task. (Note that if a subprogram had executed the CALL ENTRY CONDITIONALLY instruction, the static components of the state would also be saved if the Static Save Flag is "zero".)

(b) If the server task is SUSPENDED and marked as waiting for a call of entry S2 (it had executed an ACCEPT ENTRY instruction but no task had called that entry or it had executed a WAIT instruction after being marked during a SELECT ACCEPT instruction as waiting for a call of entry S2), its state is changed to READY and it is placed at the tail of the ready queue corresponding to the higher priority of the customer and server tasks. As described for the CALL ENTRY instruction, the address of the

SELECT ACCEPT (or ACCEPT ENTRY) instruction corresponding to entry S2 is transferred to the location in the administrative data area of the server task that contains the "execution resumption address" of the server. The server task, when scheduled to run, will resume execution at the proper SELECT ACCEPT (or ACCEPT ENTRY) instruction. The server task ceases to be marked as waiting for any entry to be called or delay to expire.

Note: When the server task is scheduled to run, the rendezvous will proceed (parameters passed to ACCEPT body which is then executed).

(c) If the server task is READY and is marked as waiting for a call of entry S2 (it had executed a SELECT ACCEPT instruction but no customer task had called the entry), the address of SELECT ACCEPT of entry S2 is moved to the location that contains the server's execution resumption address as described above and the server task ceases to be marked as waiting for any entry to be called or delay to expire. The server task is moved to the ready queue corresponding to the customer task's priority if that priority level is higher than the server task's level. The rendezvous proceeds as described in the Note under (b) above.

If the server task is COMPLETED when one of its entries is called, a TASKING ERROR is raised in the customer task at the point of call. If an exception is raised while the ACCEPT body is executing, the local exception handler is entered. If, however, no local handler is defined for the ACCEPT body, the exception is raised in the server task following the ACCEPT ENTRY or SELECT ACCEPT instruction. Further, the scheduler is invoked which changes the customer task's state to READY. TASKING ERROR pending is set. If the customer task is SUSPENDED in rendezvous (ACCEPT body executing) when the server task becomes abnormally COMPLETED, the scheduler changes the state of the customer task to READY and TASKING ERROR pending is set in the customer task. If the customer task becomes ABNORMAL during a rendezvous, the rendezvous is finished and then the scheduler changes the customer task's state to COMPLETED.

Exceptions: PROGRAM_ERROR CONSTRAINT_ERROR TASKING_ERROR

9.4.9 CALL ENTRY WITH TIMEOUT

Format: 81_H, S1, S2, S3, S4, S5,...

Mnemonic: CALENT

Operands:

S1: FMT: Pointer to Server Task memory (0) Sl addresses a pointer to the server task.

S2: Entry Number of Server Task FMT: memory (0) or immediate (EXT,2)

S3: Label FMT: immediate (EXT,2), interpreted as a label operand

S4: Delay Amount FMT: memory (0) or immediate (EXT,2)

Note: If no parameters are passed via memory transfer, then no additional operands are present in this instruction.

S5,...: Actual Parameters FMT: memory (0) or immediate (EXT,2)

Note: Any number of parameters may be passed via memory transfer. Any two may be combined in a 2-operand compact format.

Function:

This instruction attempts to call the entry, given by S2, of the server task identified by the pointer addressed by S1. The pointer must have READ authority for the server task. The number of entries, N_E , is retrieved from the server task's administrative data area and the following condition must be met (else a CONSTRAINT ERROR is raised):

$0 \le S2 \le N_E - 1$.

As with the CALL instruction, CALL ENTRY WITH TIMEOUT is processed only up to the actual parameter operands; thus, the execution resumption address (value in the program counter) that is saved in the customer task's administrative data area when the customer is SUSPENDED addresses the word following operand S4 (delay amount). Parameters that are to be passed by memory are bound during execution of the BIND PARAMETERS instruction (first instruction of the ACCEPT body) which requires access to both actual and formal parameters. BIND PARAMETERS completes the processing of the operands of the CALL ENTRY WITH TIMEOUT instruction.

Parameters that are passed by registers are loaded into parameter registers (16..31) at some points during execution of the customer task program. During CALL ENTRY WITH TIMEOUT, the registers specified in the Valid Parameter Mask (See Section 6.2.1) are saved (as part of the dynamic machine state) in the customer task's administrative data area. These values are restored in the registers by the server task when the instruction, ACCEPT ENTRY or SELECT ACCEPT, is executed provided that a customer task is queued on the accepted entry.

The delay amount, S4, is specified in units of 50 micro-seconds. If the delay is zero or negative, execution of this instruction is the same as CALL ENTRY CONDITIONALLY. If the delay is positive, the task scheduler changes the state of this (customer) task to SUSPENDED and puts it on the entry queue for entry S2 of the pointed-to server; timing of the delay begins. All dynamic components of the machine state are saved in the administrative data area of the queued customer task. (Note that if a subprogram had executed the CALL ENTRY WITH TIMEOUT instruction, (Note that if a the static components of the state would also be saved if the Static Save Flag is "zero".) If the pointed-to server task is not marked as waiting for a call of entry S2 and does not execute a SELECT ACCEPT or ACCEPT ENTRY instruction before the delay expires and/or if other customer tasks are queued on entry S2 up to the time the delay expires, then no rendezvous takes place.

The scheduler changes the state of the customer task to READY and removes it from the entry queue. The execution resumption address stored in the administrative data area of the customer task becomes "address of first instruction of customer task + label offset, S3".

When conditions do permit a rendezvous, the task scheduler is invoked and the actions in (a), (b), or (c) take place:

(a) If the server task is SUSPENDED and marked as waiting for a call of entry S2 (it had executed an ACCEPT ENTRY instruction but no task had called that entry or it had executed a WAIT instruction after being marked during a SELECT ACCEPT instruction as waiting for a call of entry S2), its state is changed to READY and it is placed at the tail of the ready queue corresponding to the higher priority of the customer and server tasks. As described for the CALL ENTRY instruction, the address of the SELECT ACCEPT (or ACCEPT ENTRY) instruction corresponding to entry S2 is transferred to the location in the administrative data area of the server task that contains the "execution resumption address" of the server. The server task, when scheduled to run, will resume execution at the proper SELECT ACCEPT (or ACCEPT ENTRY) instruction. The server task ceases to be marked as waiting for any entry to be called or delay to expire.

Note: When the server task is scheduled to run, the rendezvous will proceed (parameters passed to ACCEPT body which is then executed).

(b) If the server task is READY and is marked as waiting for a call of entry S2 (it had executed a SELECT ACCEPT instruction but no customer task had called the entry), the address of SELECT ACCEPT of entry S2 is moved to the location that contains the server's execution resumption address as described above and the server task ceases to be marked as waiting for any entry to be called or delay to expire. The server task is moved to the ready queue corresponding to the customer task's priority if that priority level is higher than the server task's level. The rendezvous proceeds as described in the Note under (a) above.

(c) If the server task executes a SELECT ACCEPT or ACCEPT ENTRY instruction for entry S2 while the delay is being timed, the delay is reset and a rendezvous ensues as described in the Note under (a).

If the server task is COMPLETED or becomes COMPLETED before the delay expires when one of its entries is called, a TASKING_ERROR is raised in the customer task at the point of call. If an exception is raised while the ACCEPT body is executing, the local exception handler is entered. If, however, no local handler is

defined for the ACCEPT body, the exception is raised in the server task following the ACCEPT ENTRY or SELECT ACCEPT instruction. Further, the scheduler is invoked which changes the customer task's state to READY. TASKING ERROR pending is set. If the customer task is SUSPENDED in rendezvous (ACCEPT body executing) when the server task becomes abnormally COMPLETED, the scheduler changes the state of the customer task to READY and TASKING ERROR pending is set in the customer task. If the ABNORMAL during a rendezvous, customer task becomes the rendezvous is finished and then the scheduler changes the customer task's state to COMPLETED.

Exceptions: PROGRAM_ERROR CONSTRAINT_ERROR TASKING_ERROR

9.4.10 ACCEPT ENTRY.

Format: $82_{\rm H}$, S1, S2

Mnemonic: ACCEPT

Operands:

S1:	Entry Numbe			
FMT:	memory	(0) or	immediate	(EXT,2)

S2: Formal Parameter Mask FMT: immediate (EXT,2)

Function:

This instruction attempts to accept, on behalf of the executing server task, a customer task's call of the entry given by Sl. If one or more customer tasks are SUSPENDED on the entry queue for entry Sl, the customer task at the head of the queue is taken off the queue (still SUSPENDED) and a rendezvous takes place. Values in parameter registers saved in the administrative data area of the customer task during CALL ENTRY, CALL ENTRY CONDITIONALLY, or CALL ENTRY WITH TIMEOUT are now restored in the registers. Only those registers, if any, designated by the Valid Parameter Mask are restored. As in the CALL SUBPROGRAM instruction, the "ls" in the Formal Parameter Mask (operand S2) must be matched by "ls" in the Valid Parameter Mask, else a PROGRAM ERROR exception is raised. The instruction following ACCEPT ENTRY is the start of the ACCEPT body; if parameters are passed via memory, this instruction is BIND PARAMETERS. The exception mode of the server task is changed from NORMAL to ACCEPT BODY. The instruction, END RENDEZVOUS, marks the end of the ACCEPT body. If no customer task is present on the entry queue for entry Sl, the server task is marked as waiting for a call to entry Sl and the address of the ACCEPT ENTRY instruction is stored in a location in the administrative data area of the server task corresponding to entry Sl (to be moved to the location that contains the "execution resumption address" of the server by the first CALL ENTRY, CALL ENTRY CONDITIONALLY, or CALL ENTRY WITH TIMEOUT instruction that calls entry Sl of this server task). The dynamic components of the machine state are saved in the administrative data area of the server task. The scheduler then changes the state of the server task to SUSPENDED.

Exceptions: PROGRAM_ERROR STORAGE_ERROR 9.4.11 END RENDEZVOUS.

Format: 83_H, D

Mnemonic: ENDRNV

Operands:

D: Label Offset FMT: immediat

immediate (EXT,2), interpreted as a label operand.

Function:

This instruction marks the end of the ACCEPT body and the end of the rendezvous. The Valid Parameter Mask is cleared, the priority level of the server task is lowered to its prerendezvous value (if it had been raised to the customer's level during rendezvous), the exception mode of the server task is changed from ACCEPT BODY to NORMAL, and the "execution resumption address" of the server task (value in program counter) is changed to "address of first instruction in server task program + label offset" (done to handle the case when the ACCEPT body follows a SELECT ACCEPT instruction and other SELECT alternatives must be skipped over). Then, the task scheduler is invoked which changes the state of the customer task from SUSPENDED to READY and places it on its ready queue. A task is scheduled to run (server task unless another task, e.g., the customer, is extant on a higher priority ready queue).

Exceptions: None .12 DELAY.

mat: 84H, S

monic: DELAY

rands:

MT: <u>Delay Amount</u> memory (0) or immediate (EXT,2)

ction:

s instruction delays execution of the RUNNING task by an ount specified by operand S. The delay amount is expressed in ober of seconds, up to the maximum representable by the chine. The quantization used is 50 micro-seconds. (In one r, 1,728*10⁶ "ticks" would occur, counting off a delay of 400.00000 seconds.) Floating point is needed to represent tiples of .00005 seconds. If S is an immediate operand, lays in units of whole seconds are represented. Negative delay ues are interpreted as zero delay. If the delay is negative zero, this instruction is a NO-OP. If the delay is positive, task is marked as waiting for a delay to expire. The task heduler is invoked and the task's state is changed to SPENDED. When the delay expires, the scheduler changes the sk's state to READY and puts it at the tail of its ready queue. task ceases to be marked as waiting for a delay to expire.

ceptions: ROGRAM ERROR .4.13 SELECT ACCEPT.

ormat: 85_H, S1, S2, S3

nemonic: SACCPT

perands:

- Sl: <u>Entry Number of Server Task</u> FMT: <u>memory (0) or immediate (EXT,2)</u>
- S2: Formal Parameter Mask FMT: immediate (EXT,2)

S3: Label Offset FMT: immediate (EXT,2), interpreted as a label operand.

Function:

This instruction executes an open ACCEPT alternative of the Ada SELECT statement. If one or more customer tasks are SUSPENDED on an entry queue for entry S1, the customer task at the head of the queue is removed from the queue (still SUSPENDED) and a rendezvous takes place. The server task ceases to be marked as waiting for any entry calls. Values in parameter registers saved in the administrative data area of the customer task during CALL ENTRY, CALL ENTRY CONDITIONALLY, or CALL ENTRY WITH TIMEOUT are now restored in the registers. Only those registers, if any, designated by the Valid Parameter Mask are restored. As in the CALL SUBPROGRAM instruction, the "ls" in the Formal Parameter Mask (operand S2) must be matched by "ls" in the Valid Parameter Mask, else a PROGRAM ERROR exception is raised. The instruction following SELECT ACCEPT is the start of the ACCEPT body; if parameters are passed via memory, this instruction is BIND PARAMETERS. The exception mode of the server task is changed from NORMAL to ACCEPT BODY. The instruction, END RENDEZVOUS, marks the end of the ACCEPT body. no customer task is present on the entry queue for entry S1, server task is marked as waiting for a call to entry S1, the iress of the SELECT ACCEPT instruction is stored in a location the administrative data area of the server task corresponding entry S1 (to be moved to the location that contains the cecution resumption address" of the server by the first CALL TRY, CALL ENTRY CONDITIONALLY, or CALL ENTRY WITH TIMEOUT struction that calls entry S1 of this server task). Then, ogram control is transferred to the address of the first struction of the server task program + label offset. (In this se, other SELECT alternatives should be evaluated or, if none i there are no instructions corresponding to an ELSE part, the IT instruction should be executed.)

ceptions: ROGRAM_ERROR FORAGE_ERROR .4.14 WAIT.

ormat: 86_H

nemonic: WAIT

perands: None

unction:

his instruction is executed when one or more open ACCEPT Iternatives were selected by the executing (server) task but no ustomer tasks were queued on entries and no other open SELECT Iternatives or an ELSE part were present. The server task, reviously marked as waiting for a call of an entry corresponding o each ACCEPT alternative, is now SUSPENDED by the task cheduler. When one of the entries is called, the scheduler changes the server task's state to READY, the task ceases to be larked as waiting for any entry call, and a rendezvous with the caller (customer task) ensues as soon as the server task is cheduled to run. If the server task is not marked as waiting for any entry call, the PROGRAM_ERROR exception is raised.

and the second second

h of the states of the

The second second

Exceptions: PROGRAM_ERROR
9.4.15 SELECT DELAY.

Format: 87_H, S

Mnemonic: SDELAY

Operands:

S: <u>Delay Amount</u> FMT: <u>memory (0) or immediate (EXT,2)</u>

Function:

This instruction executes an open DELAY alternative of the Ada SELECT statement. Execution of the RUNNING task is delayed by an amount specified by operand S. The delay amount is expressed in number of seconds, up to the maximum representable by the machine. The quantization used is 50 micro-seconds. (In one day, 1,728*10⁶ "ticks" would occur, counting off a delay of 86,400.00000 seconds.) Floating point is needed to represent multiples of .00005 seconds. If S is an immediate operand, delays in units of whole seconds are represented. Negative delay values are interpreted as zero delay. If the delay is negative or zero, the instruction immediately following SELECT DELAY is executed. If the delay is positive, the task is marked as waiting for a delay to expire and the dynamic components of the machine state are saved in the administrative data area of the task. The task scheduler is then invoked and the task's state is The task may also have been marked as changed to SUSPENDED. waiting for a call of one or more of is entries. The task scheduler is again invoked when an entry of this task is called or the delay expires, whichever occurs first. Then, the state of the task is changed to READY, the task ceases to be marked as waiting for any delay expirations or entry calls, and, when the task is scheduled to run, the machine state is restored and the task either enters a rendezvous with a caller (customer) or continues execution at the instruction immediately following SELECT DELAY. (A GOTO instruction can be used at the end of the sequence of instructions following SELECT DELAY to skip over other SELECT alternatives.)

Exceptions: PROGRAM ERROR 9.4.16 SELECT ELSE.

Format: 88_H

Mnemonic: SELSE

Operands: None

Function:

This instruction is executed when one or more open ACCEPT alternatives were selected but no callers were queued on entries and no other SELECT alternatives were present. The task ceases to be marked as waiting for any entry call. The instructions corresponding to the ELSE sequence of statements in Ada are next executed.

S such a storter state will safe

semplate for and sales front program and the second

DRUG REPAIR STREET ADDORE THE

when to Winter when the ten when it when

Exceptions: None 9.4.17 SELECT TERMINATE.

Format: 89_H

Mnemonic: STERM

Operands: None

Function:

This instruction executes an open TERMINATE alternative of the Ada SELECT statement. If the termination conditions as described in Appendix C are met, the task becomes TERMINATED. Storage for the task object's activation record is reclaimed. Further. storage is reclaimed for any data object that designated this task in a CREATE DATA OBJECT instruction. (Designation of the task in the CREATE DATA OBJECT instruction means, at the Ada program level, that the data object's access type was declared in the task program.) If any customer tasks are queued on entries of this server task, the scheduler removes them from the entry queues, changes their state from SUSPENDED to READY, and sets TASKING ERROR pending in each. If the termination conditions are not met, the task is marked as potentially terminated and the task scheduler is invoked which changes the state of the task to SUSPENDED. This (server) task may also be marked as waiting for entry calls (if it had previously executed SELECT-ACCEPT instructions with no queued customer tasks). Then, if a call to one of the marked (ACCEPTed) entries arrives before the termination conditions are met, the scheduler changes the task's state from SUSPENDED to READY, the task ceases to be marked as potentially terminated and waiting for any entry call, and a rendezvous ensues as soon as this server task is scheduled to If the termination conditions are met before a call to a run. marked entry arrives, the task is TERMINATED by the task scheduler and any customer task queued on an unmarked (not ACCEPTed) entry of this server task is removed from the gueue with its state changed to READY and TASKING ERROR pending set.

Exceptions: None

9-42

9.4.18 RETURN from TASK.

Format: 8A_H

Mnemonic: RETTSK

Operands: None

Function:

This instruction signals the normal completion of a task program. The task becomes COMPLETED but must wait to be TERMINATED until each of its dependent tasks, if any, becomes TERMINATED. When TERMINATED, storage for the task object's activation record is reclaimed. Further, storage is reclaimed for any data object that designated this task in a CREATE DATA OBJECT instruction. (Designation of the task in the CREATE DATA OBJECT instruction means, at the Ada program level, that the data object's access type was declared in the task program.) The task scheduler is invoked to schedule another task.

when a to start had at manage to set their to

.

Exceptions: None

9.4.19 SCHEDULE TASK.

Format: 8BH, D

Mnemonic: SCHDL

Operands:

D: Pointer to Task

FMT:

memory (0) D addresses a pointer to the next task to be scheduled.

Function:

The pointer, which must have WRITE authority, designates the next task to be scheduled. This task is placed at the head of the highest priority ready queue. Individual task priority, if any, that was assigned by the compiler and written in the package header for the task, is ignored. If the designated task is not READY, a PROGRAM_ERROR exception is raised.

Comment:

This instruction, which explicitly schedules a task regardless of task priority, is meant to be part of a user scheduler task that replaces the standard Ada-specific microcode scheduler. This microcode reduces to a single function: whenever a task changes state and the microcode is entered (meaning that a scheduling decision must be made, per Section 9.2), the user scheduler task is always scheduled for execution; the state of the scheduler is changed from SUSPENDED to RUNNING. To exit, the user scheduler executes a DELAY instruction, suspending itself for an "infinite" duration. (As indicated above, this state is overridden when the microcode schedules the scheduler task.)

Exceptions: PROGRAM_ERROR 9.4.20 SET TASK DURATION.

Format: 8CH, S1, S2

Mnemonic: SETDUR

Operands:

S1: <u>Time Quantum</u>

FMT:	memory (0) or immediate (EXT,2)
Immediate:	Sl specifies a time quantum in units of
	50 micro-seconds.
Memory:	S1 addresses an integer that is
and a figure	interpreted as the time quantum in
	units of 50 micro-seconds.

S2:

52: FI 10	STILY Level Of Ready Quede
FMT:	memory (0) or immediate (EXT,2)
Immediate:	S2 specifies a ready queue by priority level.
Memory:	S2 addresses an integer that is interpreted as a ready queue priority level.

Function:

The operand designated by S1 is a positive integer representing the assigned time quantum for execution of tasks on the ready queue identified by S2 via priority level. The operand designated by S2 is an integer of value >=0. If an activated task is not explicitly assigned a task duration, infinite duration is assumed when the task is scheduled to run. Tasks scheduled to run with this time quantum relinquish the processor only when they become SUSPENDED or COMPLETED or when the state of a higher priority task changes from SUSPENDED to READY.

Exceptions: PROGRAM ERROR

9-45

,4.21 ABORT TASK.

ormat: 8DH, D

nemonic: ABORT

perands:

D: <u>Pointer to Task to Be Aborted</u> FMT: <u>memory (0)</u>

unction:

perand D addresses the pointer to the task to be aborted; the ointer must have DESTROY authority. If the task's state is EADY, it is changed to COMPLETED. (However, if the task has een created but not yet activated, it is TERMINATED.) The ask's state is also changed to COMPLETED if the task is USPENDED on an ACCEPT ENTRY, SELECT ACCEPT, DELAY, or SELECT ELAY instruction. Further, if the task is a customer SUSPENDED n an entry queue, it is removed from the queue and its state is hanged to COMPLETED. If the task is a customer in rendezvous, ts state is changed to ABNORMAL and the rendezvous goes to ompletion; then, the task's state is changed to COMPLETED. In 11 cases described, when a task is aborted, every dependent task ecomes COMPLETED or ABNORMAL, the latter only if the task is a ustomer in rendezvous. COMPLETED tasks immediately become 'ERMINATED when all dependent tasks, if any, are TERMINATED.

f a customer calls an entry of an aborted (COMPLETED) task, a 'ASKING ERROR exception is raised at the point of call. If a 'ustomer is SUSPENDED on an entry queue or is SUSPENDED in

9-46

rendezvous when the server task is aborted, the customer's state is changed to READY, the customer is placed on its ready queue, and TASKING_ERROR pending is set.

Exceptions: PROGRAM_ERROR

10 POINTERS

A pointer can designate a storage object, a data entity in the variable or constant global area of a package, or a subprogram. Read, write, and destroy authorities for the pointed-to entity are specified in the pointer. A pointer to a storage object is returned by each instruction that creates a storage object. The base addresses of the storage object in data template memory and in data value memory are contained in the pointer. Table 10.1 shows the contents of the pointers returned by each "create" instruction.

Table 10.1 Pointers to storage object.

Instruction	Addresses	R	ight	s
CREATE TASK OBJECT	1 2.absolute address of task's automatic data template 3.absolute address of task's AR	R,	W,	D
CREATE PACKAGE OBJECT	1 2.absolute address of VGD template 3.absolute address of VGD	R,	₩.	
CREATE DATA OBJECT	1 2.absolute address of DO template 3.absolute address of DO	R,	W	
 CREATE UNCHECKED DATA OBJECT 	1.unique name 2.absolute address of DO template 3.absolute address of DO	R,	W,	D

|<---->|

In Table 10.1, AR = Activation Record, VGD = Variable Global Data, DO = Data Object, R = Read authority, W = Write authority, and D = Destroy authority. Note, in Table 10.1, that CREATE PACKAGE OBJECT represents the two instructions, CREATE NON-NESTED PACKAGE OBJECT and CREATE NESTED PACKAGE OBJECT; both instructions return the same pointer format.

pointer to a task object, the absolute address of the vation record in data value memory can be used to access es in the adjacent task administrative data. (Base address dministrative data = absolute address of activation record In a pointer to a package object, the absolute address of variable global data template can be used to access values in adjacent package header. (Base address of header = absolute ess of variable global data template -1.) The absolute ess of the variable global data in data value memory can be access values in the adjacent package administrative to (Base address of administrative data = absolute address of able global data -1.) Finally, in a pointer to a data ct, the absolute address of the data object in data value ry can also be used to access values in the adjacent data ct administrative data. (Base address of administrative data solute address of data object -1.)

ecked storage deallocation, programmed at the Ada Level, ws explicit deallocation of dynamically allocated data Execution of the instruction, DESTROY DATA OBJECT, cts. d leave dangling references (pointers to objects that no er exist). To detect dangling references, data objects can reated with the CREATE UNCHECKED DATA OBJECT instruction that qns a 24-bit unique name to the data object, stores it into pointer, and sets the unique name flag (see pointer format in ion 3.4). A unique name will not be reassigned until 224 erent names have been assigned to data objects that are to be icitly destroyed. (Note that the normal procedure for roying a data object is to wait for the destruction of the age object in which the Ada access type was declared.) When ique name is assigned, it is stored in a system-wide Unique Table; when the pointed-to data object is destroyed, its jue name is deleted from the table, never, in principle, to pear. Any reference via a pointer to a data object in which unique name flag is set requires a check for the existence of unique name in the table. If the unique name is not in the .e, a CONSTRAINT ERROR is raised.

ructions are provided which assign values to pointers to data ties in the variable and constant global data of local and rnal packages and to non-nested subprograms in external tages. These pointers support the Ada context clause TH/USE). Table 10.2 shows their contents.

Table 10.2 Explicitly assigned pointers.

|<---->|

ADDRESSES	RIGHTS
<pre>1 2.absolute address of data entity in VGD (or.CGD) template 3.absolute address of data entity in VGD (not used for CGD)</pre>	 R, W (see note)
<pre>1 2.absolute address of data entity in VGD template of external package 3.absolute address of data entity in VGD of external package</pre>	R, W (see note)
1 2.absolute address of data entity in CGD template of external package 3	R
<pre>1.offset to program component in external package header 2.absolute address of external package header 3.absolute address of external package adminis- trative data</pre>	R
	ADDRESSES 1 2.absolute address of data entity in VGD (or. CGD) template 3.absolute address of data entity in VGD (not used for CGD) 1 2.absolute address of data entity in VGD template of external package 3.absolute address of data entity in VGD of external package 1 2.absolute address of data entity in CGD template of external package 3 1.offset to program component in external package header 2.absolute address of external package header 3.absolute address of external package header 3.absolute address of external package adminis- trative data

In Table 10.2, VGD = Variable Global Data, CGD = Constant Global Data, R = Read authority, and W = Write authority.

Note: The rights indicated may be further restricted by certain conditions existing when a particular instruction that assigns a value to a pointer is executed (see Sections 10.1 and 10.2).

In a pointer to an external program (subprogram or task program in an external package), the offset to the program component in the package header gives the relative location, in number of ls, of a five-word packet of information pertinent to the gram.

h a data entity in the variable global data area of a local or ernal package is referenced via a pointer, the residency bit ects the address in data template memory (pointer word 2) or data value memory (word 3). When a data entity in the stant global data area is referenced, the address in template ory (pointer word 2) is always used.

nters can be moved to any visible location in the local kage and to the global data area of an external package (to ieve linking). Pointers can be passed as parameters and can a their rights restricted. For security, initial values of nters, preset by the compiler, are not permitted. Only the hine can load values into pointers. When packages are loaded, nter values are set to NULL (undefined bit = 1 interpreted as L). NULL pointers designate no entity.

D and WRITE authorities for data entities simply allow the nted-to data to be examined and modified, respectively. ever, when these authorities appear in a pointer to a task ect or a subprogram, their meaning depends on the particular truction in which the pointer is an operand and is described ividually for each such instruction. DESTROY authority allows explicit destruction of certain storage objects. Present in operand of a DESTROY DATA OBJECT pointer which is an truction, it permits the destruction of the pointed-to data Present in a pointer which is an operand of an ABORT ect. K instruction, it permits the destruction of the TASK OBJECT suming dependency conditions permit the COMPLETED task to Note that packages and activation records ome TERMINATED). never destroyed explicitly.

Section 3.4 for a description of the pointer format.

10 - 4

1 ASSIGN POINTER TO GLOBAL DATA.

mat: 8E_H, S1, D

emonic: ASNGPD

A REAL PROPERTY AND A REAL

erands:

l: <u>Data Entity in Global Data Area</u> FMT: <u>memory (0)</u>

: <u>Assigned Pointer</u> FMT: <u>memory (0)</u>

nction:

is instruction generates a pointer to a data entity located in e variable or constant global data area of the enclosing ocal) package. Sl is the address of the data entity. The dress space, ADS, must be 0 or 15, designating either display gister 0 (that contains the base addresses of the package riable global data and its template) or display register 15 hat contains the base address of the package constant global ta in template memory). The absolute addresses of the data tity are computed as follows:

(a) ADS = 0

- Address in data template memory = base address of variable global data template + cell offset.
- Address in data value memory = base address of variable global data + cell offset.

(b) ADS = 15

• Address in data template memory = base address of constant global data + cell offset.

le following values are assigned to the pointer addressed by D:

(a) ADS = 0

• WORD 1 - ENT <= 011 (data entity in variable global data area).

- RIGHTS <= READ, WRITE (see Note).

• WORD 2 - Absolute address of data entity in variable global data template.

• WORD 3 - Absolute address of data entity in variable global data area in data value memory.

ADS = 15

• WORD 1 - ENT <= 100 (data entity in constant global data area).

- RIGHTS <= READ.

 WORD 2 - Absolute address of data entity in constant global data area.

• WORD 3 - Not used.

If S1 addresses a pointer or a formal reference parameter, the values in the three words of the pointer or formal reference parameter are copied into the pointer addressed by D. Hence, the rights to a data entity in the variable global data area may be restricted (i.e., not READ and WRITE). If a formal reference parameter is addressed by S1, it must have an ENT field or 011 or 100 (global data).

a pointer to a data entity in the variable global data area efferenced (ENT = 011), the residency bit selects the absolute ess in word 2 or word 3 of the pointer. When the data entity is the constant global data (ENT = 100), the absolute address ord 2 is always used.

STIONS: SRAM ERROR 10.2 ASSIGN POINTER TO EXTERNAL VGD.

Format: 8FH, S1, S2, D

Mnemonic: ASNPXV

Operands:

- Sl: <u>Pointer to External Package</u> FMT: <u>memory (0)</u>
- S2: Offset to Data Entity in Variable Global Data FMT: immediate (EXT,2)
- D: <u>Assigned Pointer</u> memory (0)

Function:

This instruction generates a pointer to a data entity located in the variable global data area of an external package. Sl is the address of a pointer to the external package. S2 is the offset to the data entity in question in the variable global data area of this package. The absolute addresses of the data entity are computed as follows:

- o Address in data template memory = base of variable global data template (retrieved from word 2 of the pointer to the external package) + cell offset (operand S2).
- o Address in data value memory = base address of variable
 global data (retrieved from word 3 of the pointer to the
 external package) + cell offset (operand S2).

The following values are assigned to the pointer addressed by D:

- o WORD 1 ENT <= 011 (data entity in variable global data area).
 - RIGHTS <= READ, WRITE (See Note).
- o WORD 2 Absolute address of data entity in variable global data template.
- o WORD 3 Absolute address of data entity in variable global data area in data value memory.
- Note: The rights to the data entity are READ and WRITE only if the pointer to the package has these rights. Lesser rights in this pointer restrict the rights given to the generated pointer.

When the generated pointer is referenced, the residency bit selects the absolute address in word 2 or word 3.

STATISTICS IN MALE

Exceptions: PROGRAM ERROR

.

10.3 ASSIGN POINTER TO EXTERNAL CGD.

Format: 90_H, S1, S2, D

Mnemonic: ASNPXC

Operands:

Sl: <u>Pointer to External Package</u> FMT: <u>memory (0)</u>

S2: Offset to Data Entity in Constant Global Data FMT: immediate (EXT,2)

D: <u>Assigned Pointer</u> FMT: <u>memory (0)</u>

Function:

This instruction generates a pointer to a data entity located in the constant global data area of an external package. Sl is the address of a pointer to the external package. S2 is the offset to the data entity in question in the constant global data area of this package. The absolute address of the data entity is computed as follows:

• Address in template memory = base address of variable global data template (retrieved from word 2 of the pointer to the external package) + size of variable global data area (retrieved from the package descriptor located at the address in word 2 of the pointer to the external package -1) + cell offset (operand S2).

The following values are assigned to the pointer addressed by D:

- WORD 1 ENT <= 100 (data entity in constant global data area).
 - RIGHTS <= READ.
- WORD 2 Absolute address of data entity in constant global data area.
- WORD 3 Not used.
- Note: The rights in the pointer to the external package must include READ.

Exceptions: PROGRAM_ERROR 10.4 ASSIGN POINTER TO EXTERNAL PROGRAM.

Format: 91_H, S1, S2, D

Mnemonic: ASNPXP

Operands:

- S1: <u>Pointer to External Package</u> FMT: <u>memory (0)</u>
- S2: Offset to Program Component in Package Header FMT: immediate (EXT,2)
- D: <u>Assigned Pointer</u> FMT: <u>memory (0)</u>

Function:

This instruction generates a pointer to a non-nested program (subprogram or task program) in an external package. Sl is the address of a pointer to the external package. S2 is the offset in the external package header to the subprogram or task program component (a 5-word packet of information pertinent to the program). The following values are assigned to the pointer addressed by D:

- WORD 1 ENT <= 101 (program in external package).
 - RIGHTS <= READ.
 - Offset, in number of words, to program component in package header (operand S2).
- WORD 2 Absolute address of external package header (address in word 2 of the pointer to the external package - 1).
- WORD 3 Absolute address of external package administrative data area (address in word 3 of the pointer to the external package -1).
- Note: The rights in the pointer to the external package must include READ.

Exceptions: PROGRAM_ERROR 10.5 RESTRICT ACCESS RIGHTS.

Format: 92_H, S1, D

Mnemonic: RSTRCT

Operands:

S1: <u>Access Rights Restrictions</u> FMT: <u>immediate (EXT,2)</u>

D: <u>Pointer Whose Rights Are Restricted</u> FMT: <u>memory (0)</u>

Function:

This instruction lowers one or more of the authorities of the pointer addressed by D. Sl is a 3-bit immediate operand that controls the rights of the pointer as follows:

BIT 0: 1 - READ authority removed. 0 - no restriction imposed. BIT 1: 1 - WRITE authority removed. 0 - no restriction imposed. BIT 2: 1 - DESTROY authority removed. 0 - no restriction imposed.

Exceptions: None

11 Exceptions

An exception is an indication that an erroneous condition has occurred in the execution of a program. When an exception occurs (is "raised"), normal execution of the program unit in which the exception occurred is abandoned and is replaced by execution of an exception handler (see below). It is not possible to continue or resume execution at the point at which the exception occurred.

Exceptions may be predefined or user-defined. Predefined exceptions (see Table 11.I) are raised automatically by the machine when the corresponding erroneous condition is detected. User-defined exceptions may only be explicitly raised by the RAISE instruction; the RAISE instruction can also be used with predefined exceptions. I/O devices can only raise predefined exceptions.

An exception handler can be defined for an activation record (of a subprogram or task program) by execution of the INITIALIZE HANDLER instruction that specifies the address of the first instruction of the exception handler in the corresponding subprogram or task program.

When an exception is raised during an instruction, execution of that instruction is abandoned after completing any operations required to maintain the integrity of the machine, e.g. linking into a queue). If a handler is defined in the current environment (subprogram or task program), execution continues at the first instruction of the handler; otherwise, the current environment is terminated (after waiting for the termination of any dependent tasks, in the RETURN FROM SUBPROGRAM as instruction) and the exception is "propagated". If the current environment is an activation record resulting from a subprogram call, the exception is raised in the dynamically linked (calling) environment. If the current environment is activation record of a task program (sever task) when an ACCEPT body is executing RENDEZVOUS extant), the exception is raised in the server task and the customer task enters the ready state with TASKING ERROR pending is set. Special cases of exceptions corresponding to the different exception modes are covered in detail in the text.

The exception handler can obtain the exception which occurred by executing the RETRIEVE EXCEPTION instruction. The handler can execute any instruction which is otherwise legal, including raising the same or another exception, or any RETURN instruction. Table 11.I Predefined Exceptions.

- - - - - - -

Number	Name	Raised by Machine When
0	CONSTRAINT_ ERROR	1. operand falls outside range of values specified in ASSERT RANGE INTEGER or ASSERT RANGE FLOATING POINT instruction.
		2. array subscript falls outside bounds of corresponding dimension when an array component or slice is referenced through the array header.
		 3. instruction addresses a data object but its unique name is unknown to the machine (not in unique name table).
		<pre>4. attempt made to reference ar entity via a Null pointer.</pre>
1	NUMERIC_ ERROR	 arithmetic overflow occurs. division by zero is attempted.
		3. taking square root of negative number is attempted.
2	PROGRAM_ ERROR	<pre>1. invalid instruction operation code or format (FMT) detected.</pre>
		2. end of subprogram's or task program's instruction space encountered during fetching of an instruction.
		3. stack overflow/underflow occurs.

Number	Name	Raised by Machine When
2	PROGRAM_ ERROR	4. operand tag not compatible. with instruction.
		5. value of a label operand would cause a branch outside the current subprogram's or task program's instruction space.
125		6. executing WAIT instruction and there are no open SELECT alternatives (task not marked as waiting for any entry call).
		7. program attempts to operate on an entity via a pointer or formal reference parameter and lacks the appropriate authority.
3	STORAGE_ ERROR	<pre>1. insufficient storage is available in data value memory for creation of an activation record and associated administrative data, the variable global data of a package and associated administrative data, (or a data object. and associated administrative data</pre>
		2. insufficient storage is available in data template memory and/or in instruction memory when space is to be allocated for a non-nested package (in the ALLOCATE PACKAGE STORAGE instruction).

Table 11.1 Predefined Exceptions. (continued)

11-3

1.1.1.1.1.1.1.1.

Table 11.1 Predefined Exceptions. (continued)

Number	Name	Raised by Machine When
4	TASKING_ ERROR	Any of several conditions arise during task creation, activation, and rendezvous (see Section 9).
5-7	RESERVED	
8-31	1/0	See Table 11.2, these exceptions lare raised automatically only as la result of performing loperations on I/O devices.

Number	Name	Raised by Device When
- 8	NAME ERROR	<pre>1. a file with the specified file name cannot be created (e.g., because a file with that named already exists).</pre>
		 a file with the specified file name does not exist or access to the file is prohibited.
9	USE ERROR	an operation is incompatible with the properties of the specified file (e.g., an attempt is made to write to a protected file).
10	STATUS ERROR	an operation cannot be performed on a file in its present state (e.g., an attempt is made to read a file which is not open).
11	DATA ERROR	 input data has the undefined value. input data is not of the
12	DEVICE ERROR	required type. an operation cannot be completed because of a mal- function of the underlying
		system (e.g., printer runs out of paper).

Table 11.2 Predefined I/O Exceptions.

11-5

Table 11.2 Predefined	1 1/0	Exceptions.	(continued)
-----------------------	-------	-------------	-------------

A CONTRACTOR OF THE OWNER OWNER OF THE OWNER OWNER

Number	Name	Raised by Device When
13	END ERROR	an attempt is made to read beyond the end of a file.
14	LAYOUT ERROR	an operation is incompatible with the layout of the specified file.
15	MODE ERROR	an attempt is made to read an OUT_FILE or write to an IN_FILE.
16-31	RESERVED	

L.1 RAISE

ormat: 93_H, S

nemonic: RAISE

perands:

S: Exception Number FMT: immediate (EXT,2), memory (0), or stack (EXT,0)

unction:

he exception specified by operand S is raised. operand S must e a position integer (V16 or V32). The correspondence between he exception raised and the value of the integer is shown below:

integer	exception
0-7	machine predefined
8-31	I/O device predefined
31-Limit	user-defined

f S is an immediate value, it is interpreted as having a V32 tag ith sign (zero) extend.

xceptions: PROGRAM_ERROR

ASSERT RANGE INTEGER

t: 94_H, S1, S2, S3

nic: ASRTRI

nds:

1:

12

1.

Upper Limit of Range immediate (EXT,2), memory (0), or stack (EXT,0)

Lower Limit of Range immediate (EXT,2), memory (0), or stack (EXT,0)

Variable to Be Range Checked memory (0) or stack (EXT,0)

ion:

The value of the operand addressed by S3 is greater than or to the value of the operand specified by S2 and less than jual to the value of the operand specified by S1, no action then, else a CONSTRAINT ERROR exceptions is raised. All inds must be integers (V16 or V32). Immediate operands are preted as having a V32 tag with sign extend.

STRAM ERROR

3 ASSERT RANGE FLOATING POINT

mat: 95_H, S1, S2, S3

monic: ASRTRF

erands:

L: Upper Limit of Range FMT: memory (0) or stack (EXT,0)

2: Lower Limit of Range FMT: memory (0) or stack (EXT,0)

3: <u>Variable to Be Range Checked</u> FMT: <u>memory (0) or stack (EXT,0)</u>

nction:

the value of the operand addressed by S3 is greater than or ual to the value of the operand specified by S2 and less than equal to the value of the operand specified by S1, no action taken, else a CONSTRAINT ERROR exceptions is raised. All erands must be floating point numbers (V16 or V64).

Ceptions: ROGRAM_ERROR CONSTRAINT_ERROR

11.4 INITIALIZE HANDLER

Label

Format: 96_H, S

Mnemonic: IHNDLR

Operands:

S: FMT:

immediate (EXT, 2), interpreted as a label operand

Function:

If the operand specified by S is non-zero, the exception handler located at the address of the current instruction plus the value of the label operand is enabled (handler enable bit set to 1 and address of handler written into administrative data area of the local activation). The instruction data at the address of the handler must be RETRIEVE EXCEPTION. If the operand specified by S is zero, the current local exception handler is disabled (handle enable bit reset to 0).

Exceptions: PROGRAM_ERROR CONSTRAINT_ERROR

11.5 RETRIEVE EXCEPTION

Label

Format: 97_H, S, D Mnemonic: RTRVXC

Operands:

S: FMT:

immediate (EXT,2), interpreted as a label operand

D: <u>Exception Number Location</u> FMT: memory (0)

Function:

This is the first instruction of the exception handler. The exception number of the exception that occurred prior to entering this handler (automatically written into the administrative data area of the activation containing the enabled handler) is made available to the program by storing it in the location specified by operand D. Operand D must be an integer (V16 or V32). The handler for subsequent exceptions is set to the address of the currrent instruction plus the value of the label operand and enabled (if the label is no zero) as in the INITIALIZE HANDLER instruction.

Exceptions: PROGRAM ERROR

12 User Console

The User Console is a small computer - based "workstation" used to control all' phases of program debugging, maintenance, and loading of the HLLM. The computer may be a small DEC model such as the PDP11/34a or the PDP11/24. The User Console has two interfaces: a general purpose parallel interface (e.g., DR11-C) with the HLLM (which requires a User Console Interface Card) and a serial modem - controlled interface (e.g., DL11) with VAX mainframe. User Console software is partitioned into three functional area: (1) user interface, (2) HLLM interface, and (3) mainframe interface.

1. User Interface - All interactions between the User Console, the HLLM, and the mainframe are a direct or indirect result of user commands. These include the following categories:

- console control commands (initialize, terminate, show console status, execute commands in file, etc.)
- memory commands (inspection and alteration of HLLM memory words or of an image of the HLLM memory stored in a file.
- storage object related commands (display of administrative data, list of task objects, cell displays, instruction displays, etc.).
- HLLM control commands (set/report machine state, control instruction execution, control trace options, report execution history).
- breakpoint control commands (set, clear, list breakpoints).
- symbolic definition commands (create, delete, list symbols which may be used in command parameters).
- HLLM I/O commmands (create input data files, display output data files, connect files to logical simulated I/O device).
- save/restore commands (save HLLM memory block contents in files, restore HLLM memory blocks from files, compare memory blocks to files).
- commands to effect transfer of files (in either directions) between the User Console and the VAX mainframe.

12-1

2. HLLM Interface - The User Console (PDP11) may issue commands to the HLLM and the HLLM may issue request and responses to the User Console across this interface. The former (user Console command) comprise the following:

- reset
- read status
- read register
- write register
- read block
- write block
- run
- halt
- step
- set breakpoint
- interrupt (raised for a simulated I/O device)
- take input data (for a simulated I/O device)
- send output data (from a simulated I/O device)
- send trace data

The latter (requests and responses to User Console) include the following:

- status
- register data
- memory data
- HLLM output data (from simulated I/O device)
- trace data
- request "send input data"
- request "receive output data"

3. Mainframe (VAX) Interface - user commands effect the transfer of files between the User Console and the VAX mainframe. The user's terminal (on the (User Console) appears as a normal "dumb" terminal to the VAX until file transfers are initiated.

The hardware configuration of the User Console is the following:

- 64K word memory (minimum).
- disk drive, e.g., RLO1 drive.
- general purpose parallel interface, e.g., DR11-C.
- general purpose video terminal, e.g., VT-52 or VT-100.
- serial interface with modem (e.g., DL11-E).

The User Console hardware and software are described in more detail in the following documents:

- Functional Design for an Advanced Avionics Computer Architecture (interim report for period 22 Nov. 1980 to 18 Feb. 1982); 19 March 1982, pages 65-73.
- 2. Theory of Operation HLLM Hardware, 19 Nov., 1982, pages 25-32, pages 73 to 80, and pages 90 to 95.
- 3. HLLM User Console Software Functional Requirements (supplement to third interim report), 19 March 1982.

13 Traps

A trap is an automatically generated entry call to a task called the trap handler. A trap occurs at predetermined times as a result of executing certain instructions. Traps can occur in the following situations (at most one trap is generated per instruction):

trap type

when trap occurs

instruction trace every instruction (excluding those generating other traps)

branch trace every IF instruction in which the branch is actually taken or the GOTO instruction

no branch trace every IF type instruction in which the branch is not taken

instruction

every

call trace

exception trace

every time an exception is raised either by the machine or explicitly by the RAISE instruction

procedure

call

TRACE instruction

every time the TRACE instruction is executed

Each type of trap may be independently enabled or disabled by the CONTROL TRACE instruction. A trap of a particular type is ignored if it has not been enabled. The task entry which is called as a result of a trap is identified to the machine by a call to entry zero of the trap mechanism, a predefined pointer which is available to the machine. The call must have two output parameters: a pointer to the trap handler, and an integer specifying the trap handler task entry number. If no entry has been so identified, all traps are ignored. Traps may also be handled by the User Console in an implementation dependent manner.

13-1

hen a trap occurs, six read-only parameters are passed to the lentified entry of the trap handler task. The first is a pinter with no authority to the package causing the trap. The econd is an integer specifying the subprogram number in which he trap occurred. The third is an integer specifying the ubprogram number in which the trap occurred. The third is an iteger specifying the instruction address of the instruction hich caused the trap. The fourth is an integer specifying the cace type. The fifth is an integer whose value is (1) the umber of the called subprogram for call trace, (2) the number of he exception for an exception trace, (3) the immediate operand ontained (S2) in the TRACE TRAP instruction, or UNDEFINED therwise. The sixth is (1) a pointer with no authority to the ackage containing the subprogram called for a call trace, (2) a ointer with READ authority to the data entity addressed by the RACE TRAP instruction, or a "null" pointer otherwise.

hen a trap occurs, execution of the instruction causing the trap s blocked until the corresponding entry call is processed.

13.1 CONTROL TRACE

Format: 98_H, S1, S2, D

Mnemonic: CTRACE

Operands:

Sl: <u>On/Off Control for all Trace Functions</u> FMT: <u>immediate (EXT,2)</u>,

S2: <u>Trace Functions</u> FMT: <u>immediate (EXT,2), memory (0), or stack (EXT,0)</u>

D: <u>Pointer to Package Being Traced</u> FMT: <u>memory (0)</u>

Function:

The trace function indicated by the operand specified by S2 for the package pointed-to by the pointer addressed by D are turned "on" if S1 is a 1 or "off" if S1 is a 0. The immediate value of operand S1 is interpreted as a Boolean (V16) and the operand specified by S2 is mask data (V16), interpreted as follows:

Bit P	osition in Mask	Trace Function
,	6-15	reserved exception trace
	4	no-branch trace
	3 2	branch trace call trace
	1	instruction trace
	U	explicit TRACE TRA instruction

For each bit (in the range 0..5) in the mask that is a 1, the corresponding trace function is turned on or off by S1. Trace functions corresponding to the bits in the mask which are 0 are unaffected.

Exceptions: PROGRAM ERROR
2 TRACE TRAP

mat: 99_H, S1, S2, S3

monic: TRAP

rands:

.: Trace Trap Control

'MT: immediate (EXT,2), memory (0), or stack (EXT,0)

MT: <u>Type of Trace</u> MT: immediate (EXT,2)

H: Additional Trace Information MT: memory (0)

iction:

explicit tracing is "on" (see CONTROL TRACE instruction) and poperand specified by Sl is 1, a Trace Trap occurs; otherwise, action is taken. The operand specified by Sl is a Boolean 16) and the immediate value of operand S2 is mask data (V16), terpreted as a code to identify the type of trace trace. TRACE AP instructions may be selectively inserted after any struction and identified by S2. Additional information quired on the trace function may be passed to the trap handler sk via the operand specified by S3 (a pointer to a data entity ntaining the information).

ceptions: ROGRAM ERROR

APPENDIX A - EXAMPLES OF ARRAYS

- tes 1: In each example, all values are expressed in decimal unless otherwise indicated.
 - 2: Values in the Data Template Memory (DTM) can only be read.
 - 3: In general, data in data value memory (DVM) can be read and written to. In the following examples, these locations are indicated by 0, 1 in the column under residency bit and by the assignment symbol (=) following the data type in the column under DVM. When data is first written into one of these locations, the residency bit is changed from 0 (data accessed from DTM) to 1 (data accessed from DVM). Certain locations in DVM, however, are never written to. These correspond to descriptors in DTM of constrained arrays and records and to initial values of scalar and record components of arrays with separate values. A "0" in the column under residency bit designates such a location in DVM. Note that the type of descriptor (e.g., LB/UB) and tag (e.g., V32) are indicated for clarity of reading the examples.

: #1

Type REC1 is
 record
 BIG_NUM:LONG_INTEGER:=5000;
 ARR2:array (0..2) of FLOAT: =
 (0..2=> 100.1, 1000.1, 10000.1);
 end record;

ARR1:array (1..2) of REC1;

the type definition of REC₁, the variable name ARR₁ defines by with two REC₁ components. In this example, the array of 5 (ARR₁) has separate values and the array component of the (ARR₂) has immediate values (See Figure A-1).

	DTM	<u>co</u>	DVM	RESIDENCY	BIT
	AV01=8	32	AVO1	0	
	$ LB_1/UB_1=1,2$	34	LB1/UB1	. 0	
rogram	$ REC_1 = 2, 6$	36	REC ₁	0	
#1	V32=5000	38	. V3Ž	. 0	
matic	$LB_2/UB_2=0,2$	40	LB ₂ /UB ₂	0	
ata	V32=100.1	42	V32	0	
	V32=1000.1	44	V32	0	
	V32=10000.1	46	V32	0	
		48	REC1 <	0	
		1 50	V32 =	0,1	
	Subprogram	1 52	LB ₂ /UB ₂	0	
	#1	- 54	V32 =	0,1	
	Activation	56	V32 =	0,1	
	Recordn	1 58	V32 =	. 0,1	
		1 60	RECI	. 0	
		1 62	$V3\bar{2} =$	0,1	
		64	LB ₂ /UB ₂	0	
		1 66	V32 =	0,1	
		1 68	V32 =	0,1	
		70	· V32 =	0,1	
	1	Figur	e A-1		

ference data at a cell offset (CO) of 70 halfwords, AVO1 be addressed first; the address space (ADS) determines the ute base addresses of the containing activation record and emplate. A cell offset of 32 halfwords added to the ate base produces the base address of the ARR1 header (AVO1) the same offset added to the base of the activation record ices the base address of the array in data value memory. The wing additional information is supplied in the instruction im to allow computation of the offset from AVO1 to the red data (at CO=70 halfwords):

 $Subscript_1 (SUB_1) = 2$

Record₁ Component Offset $(RCO_1) = 4$ halfwords

Subscript₂ (SUB₂) = 2

s assumed, in example 1, that the subscripts of both arrays variables (values not known at compile time). Hence, the ine computes the component address as shown:

Data Address = base (ADS) + CO + AVO_1*2

+ (SUB1-LB1)*comp1 size + RCO1

+ Size of ARR₂ header+(SUB₂-LB₂)*comp₂ size.

Data Address = base (ADS) + 32 + 8*2

+ (2-1) * 6 * 2 + 4

+ 2 + (2-0) * 2

= base (ADS) + 48 + 16 + 2 + 4

= base (ADS) + 70 halfwords.

In the above computation, AVO1 is multiplied by 2 to convert to halfwords. Component1 size of ARR1 is converted to halfwords by multiplying the number of words in the record1 description (6) by 2. TRS is not needed since component1 size of ARR1 is correctly specified by the size of the record description given in the REC1 descriptor (6 words). Type REC1 is record BIG_NUM:LONG_INTEGER; ARR2:array (0..2) of FLOAT; end record;

ARR1:array (1..2) of REC1:=
 (1=> (5000, (others=> 100.1))
 2=> (10000, (others=> 1000.1));

e type definition of REC1, the variable name ARR1 defines with two REC1 components. The first REC1 component, is initialized with values 5000 for BIG NUM and 100.1 of ARR2 values. The second REC1 component, ARR1(2), is zed with values 10000 for BIG NUM and 1000.1 for all ues. In this example, the array of records (ARR1) has e values and the array component of the record (ARR2) has values (See Figure A-2).

	DTM	<u>co</u>	DVM	RESIDENCY BIT
rogram #2 matic ata	LB1/UB1=1,2 REC1=2,5 V32=5000 AV01=8 LB2/UB2=0,2 - V32=100.1 REC1=2,5 V32=10,000 AV02=6 LB2/UB2=0,2 V32=1000.1 Subprogram #2 Activation Recordn	32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64	LB1/UB1 REC1 V32 = AVO1 LB2/UB2 V32 REC1 V32 = AVO2 LB2/UB2 V32 = V32 = V32 = V32 = V32 = V32 = V32 = V32 = V32 =	0 0 0,1 0 0 0 0,1 0 0 0 0,1 0,1 0,1 0,1

.

.

Figure A-2





Before showing a second example of a component address computation, some general remarks on addressing of array components will be made. When arrays have immediate values and subscripts are known at compile time, the address of the array component can be computed by the compiler and the component can be directly addressed at run time. Alternatively, the compiler can compute the offset to the component; then, array base address register/offset addressing can be used to improve performance when frequent accesses to components in the array is required. If subscripts are variables and not known at compile time, then either the machine computes the component's address (using subscripts and header information) or the offset to the component is computed at run time, allowing base address register/offset addressing to be used. When arrays have separate values, individual components cannot be directly addressed even if subscripts are known at compile time because the tag/initial value of the separate values is not addressable in instructions. Hence, either the machine computes the component address or base address register/offset addressing is used. (In the latter case, the offset is computed at compile time if the subscripts are known, else at run time.)

In example 2, it is assumed that the subscript for ARR_1 (=2) and the record component offset (=4 halfwords) are known at compile time. Hence, the compiler can <u>directly</u> address AVO_2 using CO=48 (offset from base of activation record). If the subscript for ARR_2 is a variable addressed in the instruction stream, the machine performs the component address computation as follows (assuming the variable subscript also =2, referencing the data at CO=64):

	Base(AVO2)	= base(ADS) + CO.
Data	address	= base(ADS) + CO + AVO_2 *2
		+ (SUB ₂ -LB ₂)*comp ₂ size.
Data	address	= base(ADS) + 48 + 6*2
		+ (2-0)*2
		= base(ADS) + 60 + 4
	post stand	= base(ADS) + 64 halfwords.

Note: In the computation above, AVO₂ is multiplied by 2 to convert to halfwords. Component₂ size of ARR₂ is implied by the V32 component descriptor at CO=64 in Figure A-2. Again, TRS is not needed since the size of the array₁ component is correctly specified in the REC₁ descriptor (5 words).

EXAMPLE #3

Type REC1 is
record
BIG_NUM:LONG_INTEGER:=5000;
ARR2:array (0..2) of FLOAT:=(others=>10.9);
end record;

ARR1:array (1..2) of REC1;

Given the type definition of REC_1 , the variable name ARR₁ defines an array with two components with initial values determined by REC_1 type definition. In this example, both the array of records (ARR₁) and the array component of the record (ARR₂) have separate array values (See Figure A-3).

	DTM	<u>co</u>	DVM	RESIDENCY	BIT
	AV01=8	32	AV01	- 0	
	$ LB_1/UB_1 = 1.2$	34	LB1/UB1	1 0	
Subprogram	TRS=8	1 36	TRS	1 0	
#3	-1 REC1=2,5	38	REC1	1 0	
Automatic	1 V32=5000	40	V32	1 0	
Data	AV02=3	1 42	AV02	1 0	
	$LB_2/UB_2=0.2$	44	LB2/UB2	1 0	
	v32=10.9	46	V32 I	1 0	
		48	REC1 <<-	- 0	
		1 50	V32 =	0,1	
		1 52	AV02	0	
		54	LB2/UB2	0	
		56	V32	0	
		1 58	V32 = <	0,1	
	Subprogram	60	V32 =	0,1	
	#3	1 62	V32 =	0,1	
	Activation	64	REC1	0	
	Record	66	V32 =	0,1	
		68	AV02	0	
		70	LB2/UB2	0	
	A STATE STREET	72	V32	0	
	and a second	74	V32 = <	0,1	
	TO DO DO DO DO	1 76	V32 =	0,1	
		78	V32 =	0.1	

Figure A-3

To reference data at CO=78, AVO_1 must be addressed first; ADS determines the base addresses of the containing activation record and its data template. CO=32 specifies the offset to ARR_1 header (AVO_1) as well as to the base of the array in data value memory. The following additional information is required:

Subscript: $(SUB_1) = 2$

Record₁ Component Offset $(RCO_1) = 4$ halfwords

 $Subscript_2$ (SUB_2) = 2

In this example, it is assumed that the subscripts of both arrays are variables (values computed at run time). The address computation performed by the machine is as follows:

Data Address = base(ADS) + CO + AVO_1*2

+ (SUB1-LB1)*comp1 size + RCO

+ AVO_2 *2 + (SUB₂-LB₂)*comp₂ size

Data Address = base(ADS) + 32 + 8*2

+ (2-1) * 8 * 2 + 4

+ 3*2 + (2-0)*2

= 48 + 16 + 10 + 4

= base(ADS) + 78 halfwords.

Note: In the computation above, AVO1 and TRS are multiplied by 2 to convert to halfwords. The component size of ARR1 is explicitly specified in the Total Record Size (TRS) descriptor (located at CO=36 halfwords). TRS is needed in this example since the size of ARR1 component does not equal the size of the record description (5 words) in REC1 (at CO=38); TRS gives the correct size of ARR1 components (8 words).

EXAMPLE #4

Type REC1 is record BIG_NUM:INTEGER:=5000; ARR2:array (< >) of FLOAT:=10.9; end record;

ARR1:array (< >) of REC1;

Given the type definition of REC₁, the variable name ARR₁ defines an array of records. In this example, the array of records (ARR₁) and the array component of the record (ARR₂) are unconstrained (with dynamic bounds). Hence, in the array₁ header (See Figure A-4), LB₁/UB₁, LB₂/UB₂, and TRS have values to indicate "unconstrained" and AVA is set to undefined. Unconstrained array headers as well as unconstrained record descriptors can only appear in the contents of a Data Object Descriptor (DOD). Data object descriptions are only present in the constant global area of a package.

	DTM		DVM	RESIDENCY	BIT
Constant Global Data	DOD=9 AVA=Undefined LB1/UB1=800H/8000H TRS=FFFFFFFH REC1=2,5 V32=5000 AVO2=3 LB2/UB2=800H/8000H V32=10.9	X+0 2 4 6 8 10 12 14 16	DOD AVA=Y LB1/UB1=1,2 TRS=8 REC1 V32 AVO2 LB2/UB2=0,2 V32	0 0,1 0,1 0 0 0 0 0 0,1 0	
lst allo =9 wo X is 1 st s in D	ocation size ords (DOD)	Y+0 2 4 6 8 10 12 14 16 18	REC1 V32 = AVO2 LB2/UB2 V32 = V32 = V32 = V32 = REC1 V32 =	0 0,1 0 0,1 0,1 0,1 0,1	
2nd all detern time Y is the 2 ¹ alloc	ocation size nined at run = 16 words. the address of nd storage ation.	20 22 24 26 28 30	AVO_2 LB_2/UB_2 V32 V32 = < V32 = V32 =	0 0 0,1 0,1 0,1	

Figure A-4

Note in figure B-4 that the first entry in the header in DTM is the Data Object Descriptor (DOD) which identifies this template as that of a data object with template size = 9 words. Execution of the instruction, CREATE DATA OBJECT, includes computing the size of the outer array (ARR₁) values and allocating storage for the arrays as shown below:

- (a) First Storage Allocation Nine words are allocated in DVM for the header. This is required because of the unconstrained arrays. Lower and upper bounds are extracted from the instruction stream (given by "index constraint" operand qualifiers) and written at locations in DVM corresponding to the bounds designated as unconstrained in the DTM header.
- (b) <u>Array1 Size Computation</u> With the bounds known, the machine computes the size of the arrays as follows:

(1) Size of $ARR_2 = (UB_2 - LB_2 + 1) * comp_2$ size

=3*2=6 half-words(3 words).

(2) Component1 size=size of record description + size of array2 values

= 5 + 3 = 8 words.

A-13

This value (8 words) can now be written in TRS at a location in DVM corresponding to TRS in the DTM header.

(3) Size of array1=(UB1-LB1+1)*component1 size

=2*8=16 words.

4

(c) <u>Second Storage Allocation</u> - Sixteen words are allocated for array₁ values at some address in DVM. This address (Y) is written in AVA at a location in DVM corresponding to AVA in the DTM header.

b reference data at the location of Y+30 halfwords, AVA must be ddressed first; ADS designates the absolute base address of the onstant global data area and the specified cell offset from that ase addresses AVA. The following additional information is equired to specify data located at Y+30 halfwords:

 $Subscript_1 (SUB_1) = 2$

Record component offset $(RCO_1) = 4$ halfwords

 $Subscript_2$ (SUB_2) = 2

The address computation performed by the machine is as follows:

Data Address = Y + (SUB1-LB1)*comp1 size

+ RCO₁ + size of ARR₂ header (halfwords)

+ (SUB₂-LB₂)*comp₂ size

= Y + (2-1) *8*2

+ 4 + 6

+ (2-0)*2

Data Address = Y + 30 halfwords.

Note: In the above computation, the component size of ARR₁ (given by TRS) is multiplied by 2 to convert to halfwords.

Appendix B - Task Dependencies

the state of the s

The rules of Ada specify precise task termination conditions that are fully supported in the HLLM. They are the following:

1. A task becomes TERMINATED when it is COMPLETED and all tasks directly or indirectly dependent on it, if any, are TERMINATED. If non-terminated dependent tasks are extant, the COMPLETED task is marked as waiting for dependent tasks to terminate. Further, a subprogram that is a master cannot complete its execution, i.e., must wait at a RETURN instruction until all its dependent tasks have TERMINATED.

2. A task becomes TERMINATED when it is marked as potentially terminated (task executed SELECT TERMINATE instruction and is SUSPENDED waiting for termination conditions to be fulfilled) and both of the following termination conditions are met:

(a) One of the potentially terminated task's masters (more than one master are possible if some are indirect) has completed execution. That master could be a task in the COMPLETED state or a subprogram waiting at a RETURN. A subprogram master is also considered to have completed execution if it is waiting at a RETURN of an exception handler or if no handler exists to process the exception.

(b) Every task that depends directly or indirectly on that master is SUSPENDED and marked as potentially terminated or is TERMINATED.

Every task is directly dependent on one master and can be indirectly dependent on other masters. A task's direct master is specified in the CREATE TASK OBJECT or EVALUATE ALLOCATED TASK instruction. Indirect masters come into being when the direct master of a created task is a subprogram called by another master (the indirect master of the created task) or when the direct master of a created task is itself a task created by another master (the indirect master).

Example 1

.



.

Termination rule #1 states that if any task is COMPLETED, it is TERMINATED when all its directly and indirectly dependent tasks are TERMINATED. In example 1, if task 2 is COMPLETED, it cannot be TERMINATED until task 3 (a direct dependent) and task 4 (an indirect dependent) are TERMINATED. Note that if task 4 is COMPLETED, it is immediately TERMINATED since it has no dependent tasks.

5

Termination rule #2 states that a potentially terminated task is not TERMINATED until one of its masters is COMPLETED and all the tasks directly or indirectly dependent on that master are potentially terminated or TERMINATED. Stated another way, rule #2 means that when a task (a master) is COMPLETED and all directly and indirectly dependent tasks (beneath the master in the dependency chain) are potentially terminated or TERMINATED, all the dependent tasks linked to the master and the master are TERMINATED. For example, if task 2 is COMPLETED and tasks 3 and 4 are potentially terminated, task 4 can be TERMINATED because one of its masters (task 2, indirectly) is COMPLETED and all of task 2's directly and indirectly dependent tasks are potentially terminated. Task 3 can be TERMINATED because its direct master, task 2, is COMPLETED and all of task 2's directly and indirectly dependent tasks are potentially terminated. Then, task 2 can be TERMINATED because, from rule #1, it is COMPLETED and all of its dependent tasks are TERMINATED. Hence, each master must maintain a count of its dependents and the termination state of each. A doubly linked chain of tasks is necessary for the following reasons:

1. If a task becomes COMPLETED (or, as we will see, if a subprogram executes RETURN), it must check its dependent tasks (down the chain) for potential termination or TERMINATED state.

2. If a task becomes potentially terminated, it must notify all its masters (up the chain) so that the masters can maintain the count and termination state of its dependents. If a potentially terminated task receives an entry call, it must service the call; hence, it ceases to be marked as potentially terminated and all masters (up the chain) are notified.

When a dependency chain is being traversed, a special "restriction" mode is entered that is reset at the end of the chain (designated by a null link). While this mode is in effect (a chain is being traversed), no task is allowed to change state.

B-3

Example 2



2 4 1 By 1 - - - - B

From rule #1, if task 2 in example 2 is COMPLETED, it can be TERMINATED immediately. If task 1 is COMPLETED, it cannot dependent, task 2, becomes be TERMINATED until its TERMINATED. In the latter case, if task 2 becomes potentially terminated when task 1, its direct master, is COMPLETED, rule #2 allows task 2 to be TERMINATED and then rule #1 allows task 1 to be TERMINATED. If subprogram 2 is waiting at a RETURN (execution completed), it must not complete the RETURN instruction until both its dependent tasks are TERMINATED or potentially terminated. As in example 1, the subprogram master must maintain a count of its dependent tasks and the termination state of each. Although tasks 1 and 2 indirectly depend on subprogram 1, subprogram 1 does not affect the termination of the tasks since subprogram 2 will always execute RETURN first.

Example 3



From rule #1, if task 3 in example 3 is COMPLETED, it can be TERMINATED immediately. If the subprogram is waiting at a RETURN, rule #1 prevents it from completing the instruction until its dependent task, task 3, has TERMINATED. However, if task 3 becomes potentially terminated, rule #2 allows it to be TERMINATED and then rule #1 allows the subprogram to complete its RETURN instruction. If task 2 becomes COMPLETED, the subprogram, by definition, must also have (If task 2 is aborted, the subprogram is returned. considered to have completed .its execution.) Then, when task 3 becomes potentially terminated, rule #2 permits task 3 and then task 2 to be TERMINATED. If task 1 becomes COMPLETED, rule #1 prevents it from being TERMINATED until tasks 2 and 3 are TERMINATED. If task 3 is potentially terminated when task 1 (one of task 3's indirect masterS) becomes COMPLETED, rule #2 prevents termination until task 2 becomes potentially terminated. Then, task 3, task 2, and task 1 are TERMINATED.

Example 4

LIBRARY | 1. The Library Package creates the task. PACKAGE | The Task is dependent on the Package (no links directly. required)

TASK

The task, if COMPLETED, can be TERMINATED immediately. If the task becomes potentially terminated, it never terminates because the library package never becomes COMPLETED.

and the second second second second second

CONTRACTORS OF A CONTRACTOR

EN FILMED 9-85 DTIC