REPORT DOCUMENTATION PAGE

AD-A238 078

Public report
needed, and
Headquarters
Management

ponse, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
timate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
via Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENC		E	3. REPORT TYPE AND DATES COVERED
			Final: 09 Jan 1991 to 01 Mar 1993

4. TITLE AND SUBTITLE
Tartan Inc., Tartan Ada Sun/96MC, Version 4.0, Sun3/60 (Host) to Intel ICE960/25
on an SunOS vers 4.0.3 (Target), 90121011.11122

5. FUNDING NUMBERS

DTIC
ELECTE
JUL 02 1991
S C D

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgesellschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

**8. PERFORMING ORGANIZATION
REPORT NUMBER**
IAGB-VSR 081

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

**10. SPONSORING/MONITORING AGENCY
REPORT NUMBER**

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Tartan Inc, Tartan Ada Sun/96MC Version 4.0, Sun 3/60 (Host) to Intel ICE960/25 on an SunOS vers 4.0.3 (Target), ACVC
1.11.

91-03867

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val.
Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

| 17. SECURITY CLASSIFICATION
OF REPORT
UNCLASSIFIED | 18. SECURITY CLASSIFICATION
UNCLASSIFED | 19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on December 10, 1990.

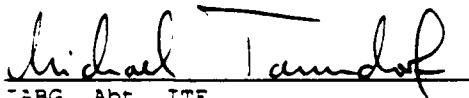Compiler Name and Version: Tartan Ada Sun/960MC version 4.0

Host Computer System: Sun 3/60   SunOS version 4.0.3

Target Computer System: Intel ICE960/25 on an Intel EXV80960MC board

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901210I1.11122 is awarded to Tartan Inc. This certificate expires on 1 March, 1993.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

== based on TEMPLATE Version 90-08-15 ==

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901210I1.11122
Tartan Inc.
Tartan Ada Sun 960MC version 4.0
Sun 3/60   => Intel ICE960/25 on an
SunOS vers 4.0.3   Intel EXV80960MC board

Prepared By:
IABG, ABT. ITE

DECLARATION OF CONFORMANCE

Customer: _____ Tartan, Inc. _____

Certificate Awardee: _____ Tartan, Inc. _____

Ada Validation Facility: ____ IABG _____

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name and Version: ____ Tartan Ada Sun/960MC Version 4.0 ____

Host Compiler System: _____ Sun 3/60 SunOS Version 4.0.3 _____

Target Computer System: _____ Intel ICE960/25 on an Intel EXV80960MC Board ___

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A   ISO 8652-1987 in the implementation listed above.

_____

Customer Signature

Date: __12/14/90__

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada
Validation Procedures [Pro90] against the Ada Standard [Ada83] using the
current Ada Compiler Validation Capability (ACVC). This Validation Summary
Report (VSR) gives an account of the testing of this Ada implementation.
For any technical terms used in this report, the reader is referred to
[Pro90]. A detailed description of the ACVC may be found in the current
ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada
Certification Body may make full and free public disclosure of this report.
In the United States, this is provided in accordance with the "Freedom of
Information Act" (5 U.S.C. #552). The results of this validation apply
only to the computers, operating systems, and compiler versions identified in
this report.

The organizations represented on the signature page of this report do not
represent or warrant that all statements set forth in this report are
accurate and complete, or that the subject implementation has no
nonconformities to the Ada Standard other than those presented. Copies of
this report are available to the public from the AVF which performed this
validation or from:

               National Technical Information Service
               5285 Port Royal Road
               Springfield VA 22161

Questions regarding this report or the validation test results should be
directed to the AVF which performed this validation or to:

               Ada Validation Organization
               Institute for Defense Analyses
               1801 North Beauregard Street
               Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint
Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC
contains a collection of test programs structured into six test classes:
A, B, C, D, E, and L. The first letter of a test name identifies the class
to which it belongs. Class A, C, D, and E tests are executable. Class B
and class L tests are expected to produce errors at compile time and link
time, respectively.

The executable tests are written in a self-checking manner and produce a
PASSED, FAILED, or NOT APPLICABLE message indicating the result when they
are executed. Three Ada library units, the packages REPORT and SPPRT13,
and the procedure CHECK_FILE are used for this purpose. The package REPORT
also provides a set of identity functions used to defeat some compiler
optimizations allowed by the Ada Standard that would circumvent a test
objective. The package SPPRT13 is used by many tests for Chapter 13 of the
Ada Standard. The procedure CHECK_FILE is used to check the contents of
text files written by some of the Class C tests for Chapter 14 of the Ada
Standard. The operation of REPORT and CHECK_FILE is checked by a set of
executable tests. If these units are not operating correctly, validation
testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class
B tests are not executable. Each test in this class is compiled and the
resulting compilation listing is examined to verify that all violations of
the Ada Standard are detected. Some of the class B tests contain legal Ada
code which must not be flagged illegal by the compiler. This behavior is
also verified.

Class L tests check that an Ada implementation correctly detects violation
of the Ada Standard involving multiple, separately compiled units. Errors
are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by
implementation-specific values -- for example, the largest integer. A list
of the values used for this implementation is provided in Appendix A. In
addition to these anticipated test modifications, additional changes may be
required to remove unforeseen conflicts between the tests and implementation-
dependent characteristics. The modifications required for this
implementation are described in Section 2.3.

1-2

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see Section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.

1-3

Customer   An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.

Declaration of A formal statement from a customer assuring that conformity Conformance is realized or attainable on the Ada implementation for which validation status is realized.

Host Computer A computer system where Ada source programs are transformed System   into executable form.

Inapplicable A test that contains one or more test objectives found to be test   irrelevant for the given Ada implementation.

Operating   Software that controls the execution of programs and that System   provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.

Target   A computer system where the executable form of Ada programs Computer are executed. System

Validated Ada The compiler of a validated Ada implementation. Compiler

Validated Ada An Ada implementation that has been validated successfully Implementation either by AVF testing or by registration [Pro90].

Validation The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.

Withdrawn A test found to be incorrect and not used in conformity test   testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO.   The rationale for
withdrawing each test is available from either the AVO or the AVF. The
publication date for this list of withdrawn tests is November 21, 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	B85001L	C83026A	C83041A	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	AD1B08A	BD1B02B	BD1B06A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant
for a given Ada implementation. Reasons for a test's inapplicability may
be supported by documents issued by ISO and the AJPO known as Ada
Commentaries and commonly referenced in the format AI-ddddd.   For this
implementation, the following tests were determined to be inapplicable for
the reasons indicated; references to Ada Commentaries are included as
appropriate.

The following 159 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113C..Y (11 tests)	C35705C..Y (11 tests)
C35706C..Y (11 tests)	C35707C..Y (11 tests)
C35719C..Y (11 tests)	C35802C..Z (12 tests)
C45241C..Y (11 tests)	C45321C..Y (11 tests)
C45401C..Y (11 tests)	C45521C..Z (12 tests)
C45524C..Z (12 tests)	C45621C..Z (12 tests)
C45641C..Y (11 tests)	C46012C..Z (12 tests)

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, there is no such type.

C45536A, C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

C45624A and C45624B are not applicable as MACHINE_OVERFLOWS is TRUE for floating-point types.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C instantiate generic units before their bodies are compiled; this implementation creates a dependence on generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (see 2.3.)

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten type'small; this implementation does not support decimal smalls. (see 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than what the length clause specified, as allowed by AI-00558.

The following 264 tests check for sequential, text, and direct access files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)

CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..E (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B and CE3107A require NAME_ERROR to be raised when an attempt
is made to create a file with an illegal name; this implementation does not
support external files and so raises USE_ERROR. (see 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 114 tests.

The following tests were split into two or more tests because this
implementation did not report the violations of the Ada Standard in the way
expected by the original tests.

B22003A	B24007A	B24009A	B25002B	B32001A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38006A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B45002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B83E01B
B85007C	B85008G	B85008H	B91004A	B91005A	B95003A
B95007B	B95031A	B95074E	BC1002A	BC1109A	BC1109C

. BC1206A   BC2001E   BC3005B   BD2A06A   BD2B03A   BD2D03A
  BD4003A   BD4006A   BD8003A

BC9002B was graded inapplicable by Evaluation and Test Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect, but for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that pragma LIST had effect.

Tests C45524A..N (14 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was within the smallest safe interval by adding the following code after line 141; the modified tests were passed:

  ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package report's body, and thus the packages' calls to function Report.Ident_Int at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by compiling the seperate files in the following order (to allow re-compilation of obsolete units), and all intended errors were then detected by the compiler:

BC3204C: C0, C1, C2, C3M, C4, C5, C6, C3M

BC3205D: D0, D2, D1M

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-ten value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following command was issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the program, this command alone is sufficient:

        adalib960> interface -sys -L=library AD9004A

CE2103A, CE2103B and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described
adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada
implementation system, see:

>   Mr Ron Duursma
>   Director of Ada Products
>   Tartan Inc.
>   300, Oxford Drive,
>   Monroeville, PA 15146,
>   USA.
>   Tel. (412) 856-3600

For a point of contact for sales information about this Ada implementation
system, see:

>   Mr Bill Geese
>   Director of Sales
>   Tartan Inc.
>   300, Oxford Drive,
>   Monroeville, PA 15146,
>   USA.
>   Tel. (412) 856-3600

Testing of this Ada implementation was conducted at the customer's site by
a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test
of the customized test suite in accordance with the Ada Programming
Language Standard, whether the test is applicable or inapplicable;
otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was
obtained that conforms to the Ada Programming Language Standard.

       a) Total Number of Applicable Tests   3629
       b) Total Number of Withdrawn Tests   83
       c) Processed Inapplicable Tests   36
       d) Non-Processed I/O Tests   264
       e) Non-Processed Floating-Point
            Precision Tests   159

       f) Total Number of Inapplicable Tests   459 (c+d+e)

       g) Total Number of Tests for ACVC 1.11 4170 (a+b+f)

The above number of I/O tests were not processed because this implementation
does not support a file system. The above number of floating-point tests were
not processed because they used floating-point precision exceeding that
supported by the implementation. When this compiler was tested, the tests
listed in Section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was
tested, the tests listed in Section 2.1 had been withdrawn because of test
errors. The AVF determined that 459 tests were inapplicable to this
implementation. All inapplicable tests were processed during validation
testing except for 159 executable tests that use floating-point precision
exceeding that supported by the implementation and 264 executable tests that
use file operations not supported by the implementation. In addition, the
modified tests mentioned in Section 2.3 were also processed.

A 1/4" Data Cartridge containing the customized test suite (see Section 1.3)
was taken on-site by the validation team for processing. The contents of the
tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of
tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as
appropriate. The executable images were transferred to the target computer
system by the communications link, an RS232 interface, and run. The results
were captured on the host computer system.

Testing was performed using command scripts provided by the customer and
reviewed by the validation team. See Appendix B for a complete listing of
the processing options for this implementation. It also indicates the
default options. The options invoked explicitly for validation testing
during this test were:

Options used for compiling:

-f forces the compiler to accept an attempt to compile a unit imported
   from another library, which is normally prohibited.

-q quiet, stops output of all compiler phase names. Not documented in
   product version as it is the default setting. Option -v was the
   default setting for the validation run.

-c normally the compiler creates a registered copy of the user's source
   code in the library directory for proper operation of the remake and
   make subcommands to Adalib.

-La forces a compiler to produce a listing even if no errors were found.

No explicit linker Options were used.

Test output, compiler and linker listings, and job logs were captured on a
1/4" Data Cartridge and archived at the AVF. The listings examined on-site
by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
SMAX_IN_LEN	240
SACC_SIZE	32
SALIGNMENT	4
SCOUNT_LAST	2147483646
SDEFAULT_MEM_SIZE	2097152
SDEFAULT_STOR_UNIT	8
SDEFAULT_SYS_NAME	I960MC
SDELTA_DOC	2#1.0#E-31
SENTRY_ADDRESS	SYSTEM.ADDRESS'(16#0000_00C8#)
SENTRY_ADDRESS1	SYSTEM.ADDRESS'(16#0000_00C9#)
SENTRY_ADDRESS2	SYSTEM.ADDRESS'(16#0000_00CA#)
SFIELD_LAST	20
SFILE_TERMINATOR	' '
SFIXED_NAME	NO_SUCH_TYPE
SFLOAT_NAME	EXTENDED_FLOAT
SFORM_STRING	""
SFORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
SGREATER_THAN_DURATION	100_000.0
SGREATER_THAN_DURATION_BASE_LAST	100_000_000_0
SGREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
SGREATER_THAN_FLOAT_SAFE_LARGE	1.0E+38

SGREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                    1.0E+38

SHIGH_PRIORITY   17

SILLEGAL_EXTERNAL_FILE_NAME1
                    ILLEGAL_EXTERNAL_FILE_NAME1

SILLEGAL_EXTERNAL_FILE_NAME2
                    ILLEGAL_EXTERNAL_FILE_NAME2

SINAPPROPRIATE_LINE_LENGTH
                    -1

SINAPPROPRIATE_PAGE_LENGTH
                    -1

SINCLUDE_PRAGMA1   "PRAGMA INCLUDE ("A28006D1.TST")"

SINCLUDE_PRAGMA2   "PRAGMA INCLUDE ("B28006F1.TST")"

SINTEGER_FIRST   -2147483648

SINTEGER_LAST   2147483647

SINTEGER_LAST_PLUS_1   2147483648

SINTERFACE_LANGUAGE   Use_Call

SLESS_THAN_DURATION   -100_000.0

SLESS_THAN_DURATION_BASE_FIRST
                    -100_000_000.0

SLINE_TERMINATOR   ' '

SLOW_PRIORITY   2

SMACHINE_CODE_STATEMENT
                    Two_Format'(MOV,(Reg_Lit,5),(Reg,R5));

SMACHINE_CODE_TYPE   Mnemonic

SMANTISSA_DOC   31

SMAX_DIGITS   18

SMAX_INT   9223372036854775807

SMAX_INT_PLUS_1   · 9223372036854775808

SMIN_INT   -9223372036854775808

A-3

$NAME	BYTE_INTEGER
$NAME_LIST	I960MC
$NAME_SPECIFICATION1	/tlsuna2/acvcl_11/validation x2120a
$NAME_SPECIFICATION2	/tlsuna2/acvcl_11/validation/x2120b
$NAME_SPECIFICATION3	/tlsuna2/acvcl_11/validation/x3119a
$NEG_BASED_INT	16#FFFFFFFFFFFFFFFE#
$NEW_MEM_SIZE	2097152
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	I960MC
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	record Operation: Mnemonic; Operand_1: Operand; Operand_2: Operand; end record;
$RECORD_NAME	Two_Format
$TASK_SIZE	32
$TASK_STORAGE_SIZE	4096
$TICK	0.015625
$VARIABLE_ADDRESS	SYSTEM.ADDRESS'(16#0000_1000#)
$VARIABLE_ADDRESS1	SYSTEM.ADDRESS'(16#0000_1004#)
$VARIABLE_ADDRESS2	SYSTEM.ADDRESS'(16#0000_1008#)
$YOUR_PRAGMA	NO_SUCH_PRAGMA

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

Compilation switches for Tartan Ada Sun 960.

-a	Generate an assembly code file. The assembly code file has an extension .s for a body or .ss for a specification (see Section FILES-LEFT).
-A	Generate an assembly code file with interleaved source code. The assembly code file has an extension .s for a body or .ss for a specification.
-c	Normally, the compiler creates a registered copy of the user's source code in the library directory for proper operation of the remake and make subcommands to AL960. This option suppresses the creation of this copy.
-Cl	Controls the type of calls generated by the compiler through the option supplied. With this option, the compiler generates all long calls in the compiled code. With the default, the compiler generates short calls within application code and long calls from applications to runtime routines.
-Cs	Controls the type of calls generated by the compiler through the option supplied. With this option, the compiler generates all short calls in the compiled code. Inappropriate use of this switch will cause a failure at link time. With the default, the compiler generates short calls within application code and long calls from applications to runtime routines.
-d	When compiling a library unit, determine whether the unit is a refinement of its previous version and, if so, do not make dependent units obsolete. This check is not done by default.
-e=<integer>	Stop compilation and produce a listing after n errors are encountered, where n is in the range 0..255. The default value for n is 255. The -e qualifier cannot be negated.
-f	Forces the compiler to accept an attempt to compile a unit imported from another library, which is normally prohibited.
-g	Compile with debugging information for AdaScope.
-i	Cause compiler to omit data segments with the

text of enumeration literals. This text is
normally produced for exported enumeration
types in order to support the text attributes
('IMAGE, 'VALUE and 'WIDTH). You should use
-i only when you can guarantee that no unit
that will import the enumeration type will
use any of its text attributes. However, if
you are compiling a unit with an enumeration
type that is not .visible to other compilation
units, this option is not needed. The
compiler can recognize when the text
attributes are not used and will not generate
the supporting strings.

-L=[project:]library Select library and/or project for this
compilation. This option takes effect after
all commands from the .adalibrc file have
been executed, thereby possibly overriding
its effects.

-La   Generate a listing, even if no errors were
found. The default is to generate a listing
only if an error is found.

-Ln   Never generate a listing. The default is to
generate a listing only if an error is found.

-Me   When package MACHINE_CODE is used, controls
whether the compiler attempts to alter operand
address modes when those address modes are used
incorrectly. With this option, The compiler
does not attempt to fix any machine code insertion
that has incorrect address modes. An error
message is issued for any machine code insertion
that is incorrect. With the default, the compiler
attempts to generate extra instructions to fix
incorrect address modes in the array aggregates
operand field.

-Mw

The compiler attempts to generate extra
instructions to fix incorrect address modes. A
warning message is issued if such a ''fixup''
is required. With the default, the compiler
attempts to generate extra instructions to fix
incorrect address modes in the array aggregates
operand field.

-Op=n   Control the level of optimization performed
by the compiler, requested by n. The
optimization levels available are:

                  n = 0   Minimum - Performs context
                            determination, constant fold-
                            ing, algebraic manipulation,
                            and short circuit analysis.

n = 1   Low - Performs level 0 op-
timizations plus common sub-
expression elimination and
equivalence propagation within
basic blocks. It also op-
timizes evaluation order.

n = 2   Best tradeoff for space/time -
the default level. Performs
level 1 optimizations plus flow
analysis which is used for
common subexpression elimina-
tion and equivalence propaga-
tion across basic blocks. It
also performs invariant expres-
sion hoisting, dead code
elimination, and assignment
killing. Level 2 also performs
lifetime analysis which is used
to improve register allocation.
It also performs inline expan-
sion of subprogram calls in-
dicated by Pragma INLINE, if
possible.

n = 3   Time - Performs level 2
optimizations plus inline ex-
pansion of subprogram calls
which the optimizer decides are
profitable to expand (from an
execution time perspective).
Other optimizations which im-
prove execution time at a cost
to image size are performed
only at this level.

n = 4   Space - Performs those
optimizations which usually
produce the smallest code,
often at the expense of
speed. This optimization
level may not always produce
the smallest code, however,
another level may produce
smaller code under certain
conditions.

-RS   Causes the compiler to accept non-Ada input,
necessary to repl-ce package SYSTEM. This
qualifier should not be used for compiling
user-defined packages containing illegal
code. Changes of package SYSTEM must fully
conform to the requirements stated in ARM 4-5
13.7 and 13.7.1, and must not change the
given definition of type ADDRESS, in order to
preserve validatability of the Ada system.

-r   For internal use only, this option is used by
         AL960 when it invokes the compiler in
         (re)make mode.

-S[ACDEILORSZ] Suppress the given set of checks:

        A   ACCESS_CHECK
        C   CONSTRAINT_CHECK
        D   DISCRIMINANT_CHECK
        E   ELABORATION_CHECK
        I   INDEX_CHECK
        L   LENGTH_CHECK
        O   OVERFLOW_CHECK
        R   RANGE_CHECK
        S   STORAGE_CHECK
        Z   "ZERO"DIVISION_CHECK

        The -S option has the same effect as an
        equivalent pragma SUPPRESS applied to the
        source file. If the source program also
        contains a pragma SUPPRESS, then a given
        check is suppressed if either the pragma or
        the switch specifies it; that is, the effect
        of a pragma SUPPRESS cannot be negated with
        the command line option. See LRM 11.7 for
        further details. Supplying the -S option
        significantly decreases the size and execu-
        tion time of the compiled code. Examples
        are:

       -SOZ Suppress OVERFLOW_CHECK and
           "ZERO"DIVISION_CHECK.

       -S   Suppress all checks.

       -SC   Suppress CONSTRAINT_ERROR, equiv-
           alent to -SADILR. (Note that -SC
           is upward compatible with Version
           2.0)

-s   Parses a unit and reports syntax errors, then
         stops compilation without entering a unit in
         the library.

-v   Print out compiler phase names. The compiler
         prints out a short description of each
         compilation phase in progress.

b

-w   Suppress warning messages.

-x   Include cross reference information for the
         source in the object file.

In addition, the output from the compiler may be redirected using the
redirection facility including '&' for stderr; for example

 % tada960mc tax_spec.ada >& tax_spec.txt

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

Linker switches for Sun hosted Tartan Ada compilers.

This section describes the command options available to a user who directly
invokes the linker. The option names can be abbreviated to unique prefixes;
the first letter is sufficent for all current option names. The option names
are not case sensitive.

-CONTROL file	The specified file contains linker control commands. Only one such file may be specified, but it can include other files using the CONTROL command. Every invocation of the linker must specify a control file.
-OUTPUT file	The specified file is the name of the first output object file. The module name for this file will be null. Only one output file may be specified in this manner. Additional output files may be specified in the linker control file.
-ALLOCATIONS	Produce a link map showing the section allocations.
-UNUSEDSECTIONS	Produce a link map showing the unused sections.
-SYMBOLS	Produce a link map showing global and external symbols.
-RESOLVEMODULES	This causes the linker to not perform unused section elimination. Specifying this option will generally make your program larger, since unreferenced data within object files will not be eliminated. Refer to Sections RESOLVE_CMD and USE_PROCESSING for infor- mation on the way that unused section elimination works.
-MAP	Produce a link map containing all information except the unused section listings.

Note that several listing options are permitted. This is because link maps
for real systems can become rather large, and writing them consumes a
significant fraction of the total link time. Options specifying the contents
of the link map can be combined, in which case the resulting map will contain
all the information specified by any of the switches. The name of the file
containing the link map is specified by the LIST command in the linker control
file. If your control file does not specify a name and you request a listing,
the listing will be written to the standard output stream.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of this Appendix F, are:

```
package STANDARD is

    . . . . . . . . . .

    type BYTE_INTEGER  is range -128 .. 127;
    type SHORT_INTEGER is range -32768 .. 32767;
    type INTEGER       is range -2147483648 .. 2147483647;
    type LONG_INTEGER  is range -9223372036854775808 .. 9223372036854775807;

    type FLOAT         is digits 6 range
    -2#1.11111111111111111111111#e126 .. 2#1.11111111111111111111111#e126;

    type LONG_FLOAT    is digits 15 range
     -2#1.1111111111111111111111111111111111111111111111111111#e1022 ..
      2#1.1111111111111111111111111111111111111111111111111111#e1022;

    type EXTENDED_FLOAT is digits 18 range
 -2#1.111111111111111111111111111111111111111111111111111111111111111#e16382 ..
  2#1.111111111111111111111111111111111111111111111111111111111111111#e16382;

    type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

    . . . . . . . . . .

    end STANDARD;
```

Chapter 5
Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to the LRM which is *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983) .

5.1. PRAGMAS

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.

- Pragma ELABORATE is supported.

- Pragma INLINE is supported.

- Pragma INTERFACE is supported. A particular Ada calling sequence is associated with a subprogram whose implementation is provided in the form of an object code module. Language_Name may be either Use_Call or Use_Bal as described in Section 5.1.2.2. Any other Language_Name will be accepted, but ignored, and the default, Use_Call will be used.

- Pragma LIST is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.

- Pragma MEMORY_SIZE is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.

- Pragma OPTIMIZE is supported, but on a subprogram basis only. It does not affect code at the block level.

- Pragma PACK is supported.

- Pragma PAGE is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.

- Pragma PRIORITY is supported.

- Pragma STORAGE_UNIT is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.

- Pragma SHARED is not supported. No warning is issued if it is supplied.

- Pragma SUPPRESS is supported.

- Pragma SYSTEM_NAME is accepted but no value other than that specified in Package SYSTEM (Section 5.3) is allowed.

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

5.1.2.1. Pragma LINKAGE_NAME

The pragma LINKAGE_NAME associates an Ada entity with a string that is meaningful externally; e.g., to a linkage editor. It takes the form

pragma LINKAGE_NAME (Ada-simple-name, string-constant)

The *Ada-simple-name* must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation; e.g., a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the *string-constant* to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

This pragma has no effect when applied to a library subprogram or to a *renames* declaration; in the latter case, no warning message is given.

When determining the maximum allowable length for the external linkage name, keep in mind that the compiler will generate names for elaboration flags simply by appending the suffix #GOTO. Therefore, the external linkage name has 5 fewer significant characters than the lower limit of other tools that need to process the name (e.g., 40 in the case of the Tartan Linker).

5.1.2.2. Pragma FOREIGN_BODY

In addition to Pragma INTERFACE, Tartan Ada supplies Pragma FOREIGN_BODY as a way to access subprograms in other languages.

Unlike Pragma INTERFACE, Pragma FOREIGN_BODY allows access to objects and exceptions (in addition to subprograms) to and from other languages.

Some restrictions on Pragma FOREIGN_BODY that are not applicable to Pragma INTERFACE are:

* Pragma FOREIGN_BODY must appear in a non-generic library package.
* All objects, exceptions and subprograms in such a package must be supplied by a foreign object module.
* Types may not be declared in such a package.

Use of the pragma FOREIGN_BODY dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module. In order to successfully link a program including a foreign body, the object module for that body must be provided to the library using the AL960 foreign command described in sections 3.3.3 and 13.5.5. The pragma is of the form:

pragma FOREIGN_BODY (Language_name [, elaboration_routine_name])

The parameter *Language_name* is a string intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). Currently, the programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada compiler. Subprograms called by tasks should be reentrant.

The optional *elaboration_routine_name* string argument is a linkage name identifying a routine to initialize the package. The routine specified as the *elaboration_routine_name*, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object declarations that use an unconstrained type mark, and number declarations. Pragmas may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names.

In the following example, we want to call a function plmn which computes polynomials and is written in C.

```
package MATH_FUNCTIONS is
   pragma FOREIGN_BODY ("C");
   function POLYNOMIAL   (X:INTEGER) return INTEGER;
         --Ada spec matching the C routine
   pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
         --Force compiler to use name "plmn" when referring to this
   -- function
end MATH_FUNCTIONS;

with MATH_FUNCTIONS; use MATH_FUNCTIONS;
procedure MAIN is
   X:INTEGER := POLYNOMIAL(10);
         -- Will generate a call to "plmn"
   begin ...
end MAIN;
```

To compile, link and run the above program, you do the following steps:

1. Compile MATH_FUNCTIONS

2. Compile MAIN

3. Obtain an object module (e.g. math.TOF) containing the compiled code for plmn.

4. Issue the command

     AL960 foreign math_functions math.TOF

5. Issue the command

     AL960 link main

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for MATH_FUNCTIONS.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command AL960 foreign (see Sections 3.3.3 and 13.5.5) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

5.1.2.3. Pragma INTERFACE

The pragma INTERFACE associates a particular Tartan Ada calling sequence with a subprogram whose implementation is provided in the form of an object code module.

The form of the pragma is:

          pragma INTERFACE (*Language_Name, Subprogram_Name*)

Language_Name may be either Use_Call or Use_Bal as described in Section 5.1.2.2. Any other Language_Name will be accepted, but ignored, and the default, Use_Call will be used.

While the BAL calling convention is faster than the standard calling convention, be aware that BAL must be used carefully. In particular, when a routine is called with BAL:

- No new stack frame is allocated. This means that the called routine must not change the stack pointer, or must at least ensure that the stack pointer is restored before the routine returns.

- No new local registers are allocated.

- The called routine must return via a bx (reg) instruction. The BAL instruction will automatically store the return address in register g14.

- If a called routine has more than 12 words worth of parameters, the compiler will store the argument block pointer in g14. Since the BAL instruction will place the return address in g14, the called routine could find that its argument block pointer has been trashed.

Please see Chapter 6 for a complete list of BAL calling convention restrictions.

5.2. IMPLEMENTATION-DEPENDENT ATTRIBUTES

No implementation-dependent attributes are currently supported.

5.3. SPECIFICATION OF THE PACKAGE SYSTEM

The parameter values specified for the 80960MC in package SYSTEM [LRM 13.7.1 and Appendix C] are:

```
package SYSTEM is
    type ADDRESS is new Integer;
    type NAME is (I960MC);

    SYSTEM_NAME  : constant name := I960MC;
    STORAGE_UNIT : constant := 8;
    MEMORY_SIZE  : constant := 2_097_152;

    MAX_INT      : constant := 9_223_372_036_854_775_807;
    MIN_INT      : constant := -MAX_INT - 1;

    MAX_DIGITS   : constant := 18;

    MAX_MANTISSA : constant := 31;
    FINE_DELTA   : constant := 2#1.0#e-31;
    TICK         : constant := 0.015625;
    subtype PRIORITY is INTEGER range 2 .. 17;
    DEFAULT_PRIORITY : constant PRIORITY := PRIORITY'FIRST;
    RUNTIME_ERROR    : exception;
end SYSTEM;
```

5.4. RESTRICTIONS ON REPRESENTATION CLAUSES

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

5.4.1. Basic Restriction

The basic restriction on representation specifications [LRM 13.1] is that they may be given only for types declared in terms of a type definition, excluding a generic_type_definition (LRM 12.1) and a private_type_definition (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; an error message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions cause compilation to stop and a diagnostic message to be issued.

5.4.2. Length Clauses

Length clauses [LRM 13.2] are, in general, supported. For details, refer to the following sections.

5.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.

2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

   - An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example

```
type My_Enum is (A,B);
for My_enum'size use 1;
V,W: My_enum;  -- will occupy two storage
               -- units on the stack
               -- (if allocated at all)
type rec is record
   V,W: My_enum;
end record;
pragma Pack(rec);
O: rec;        -- will occupy one storage unit
```

   - A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.

   - Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational sizes in different contexts.

Note: A size specification cannot be used to force a certain size in value operations of the type; for example

```
type my_int is range 0..65535;
for my_int'size use 16; -- o.k.
A,B: my_int;
   ...A + B... -- this operation will generally be
               -- executed on 32-bit values
```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. Thus, in the example

```
type MY_INT is range 0..2**15-1;
for MY_INT'SIZE use 16; -- (1)
subtype SMALL_MY_INT is MY_INT range 0..255;
type R is record
              ...
              X: SMALL_MY_INT;
              ...
        end record;
```

the component R.X will occupy 16 bits. In the absence of the length clause at (1), R,X may be represented in 8 bits.

Size specifications for access types must coincide with the default size chosen by the compiler for the type.

Size specifications are not supported for floating-point types or task types.

5.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0 (even if 0 is not in the range of the values of the type). For numeric types with negative values the number of bits must account for the sign bit. No skewing of the representation is attempted. Thus

```
type my_int is range 100..101;
```

requires at least 7 bits, although it has only two values, while

```
type my_int is range -101..-100;
```

requires 8 bits to account for the sign bit.

A size specification for a real type does not affect the accuracy of operations on the type. Such influence should be exerted via the accuracy_definition of the type (LRM 3.5.7, 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

5.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy. Any alignment constraints on the component type (see Section 5.4.7) must be met.

The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless pragma PACK is given. This applies even to boolean types or other types that require only a single bit for the representation of all values.

5.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by Pragma PACK.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record or contain relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package SYSTEM for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a STORAGE_ERROR exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. No matter what the requested object size, the allocator must allocate a minimum of 2 words per object. This lower limit is necessary for administrative overhead in the allocator. For example, a request of 5 words results in an allocation of 5 words; a request of 1 word results in an allocation of 2 words.

Furthermore, the allocator must round non-word sized requests up to the nearest word. For example, a request of 11 bytes is rounded up to 12 bytes (3 words).

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, STORAGE_ERROR is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package SYSTEM for the meaning of storage units.

If the storage specified for a task activation (T' Storage_Size) is not a multiple of 4096 (one page), the compiler allocates the next higher multiple of 4096, as permitted by the language.

Any attempt to exceed the activation size during execution causes a STORAGE_ERROR exception to be raised. Unlike collections, there is no extension of task activations.

5.4.2.7. Specification of ' SMALL

Only powers of 2 are allowed for ' SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; the specification of ' SMALL must then be accommodatable within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses [LRM 13.3], the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between INTEGER'FIRST and INTEGER'LAST. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.

- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses [LRM 13.4] is observed.

Static objects may be aligned at powers of 2 up to a page boundary. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

The size specified for each component must be sufficient to allocate all possible values of the component subtype (but not necessarily the component type). The location specified must be compatible with any alignment constraints of the component type; an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, then the discriminants and components without component clauses are allocated after those with component clauses; no attempt is made to utilize gaps left by the user-provided allocation.

5.4.5. Address clauses

Address clauses [LRM 13.5] are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is accepted but meaningless. Please refer to section 8.10 for details on how address clauses relate to linking; refer to section 12.2 for an example.

- Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied to a package only if it is a body stub.

- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt. Refer to section 10.2.7 for more details. A specified address must be an Ada static expression.

- Address clauses specify virtual, not physical, addresses.

- When specifying absolute addresses, please note that the compiler will treat addresses as an INTEGER type. This means that specifications of addresses may raise arithmetic overflow errors; i.e., addresses must be in the range INTEGER'FIRST..INTEGER'LAST. To represent an address greater than INTEGER'LAST, use the negated radix-complement of the desired address. For example, to express address 16#C000_000, specify instead -16#4000_000.

5.4.6. Pragma PACK

Pragma PACK [LRM 13.1] is supported. For details, refer to the following sections.

5.4.6.1. Pragma PACK for Arrays

If pragma PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (Section 5.4.2.3).

If, in addition, a length clause is applied to

1. The array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.

2. The component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if pragma PACK is applied where there was no length clause given for the component type.

5.4.6.2. The Predefined Type String

Package STANDARD applies Pragma PACK to the type string.

However, when applied to character arrays, this pragma cannot be used to achieve denser packing than is the default for the target: 4 characters per 32-bit word.

5.4.6.3. Pragma PACK for Records

If pragma PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. Pragma PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are of non-referable nature. In the absence of pragma PACK, such components generally consume a referable amount of space.

It should be noted that the default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

for TOENTRY **use at** intID;

by associating the interrupt specified by intID with the toentry entry of the task containing this address clause. The interpretation of intID is both machine and compiler dependent.

The Ada runtimes provide interrupts that may be associated with task entries. These interrupts are of type System.Address in the ranges 8..243, 252..255, 264..499, and 508..511.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports UNCHECKED_CONVERSION with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of UNCHECKED_CONVERSION are made inline automatically.

5.8. IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES

Tartan Ada supplies the predefined input/output packages DIRECT_IO, SEQUENTIAL_IO, TEXT_IO, and LOW_LEVEL_IO as required by LRM Chapter 14. However, since 80960MC processor is used in embedded applications lacking both standard I/O devices and file systems, the functionality of DIRECT_IO, SEQUENTIAL_IO, and TEXT_IO is limited.

DIRECT_IO and SEQUENTIAL_IO raise USE_ERROR if a file open or file access is attempted. TEXT_IO is supported to CURRENT_OUTPUT and from CURRENT_INPUT. A routine that takes explicit file names raises USE_ERROR. LOW_LEVEL_IO for 80960MC processor provides an interface by which the user may read and write from memory mapped devices. In both the SEND_CONTROL and RECEIVE_CONTROL procedures, the device parameter specifies a device address while the data parameter is a byte, halfword, word, or doubleword of data transferred.

5.9. OTHER IMPLEMENTATION CHARACTERISTICS

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

5.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the AL960 LINK command) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks [described in LRM 9.4 (6-10)]. Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

5.9.2. Implementation of Generic Units

All instantiations of generic units, except the predefined generic UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada.

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will casue any units that instantiated this generic unit to become obsolete.

5.9.3. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

Attribute	Value
DURATION'DELTA	0.0001 sec
DURATION'SMALL	0.000061 sec
DURATION'FIRST	-86400.0 sec
DURATION'LAST	86400.0 sec

5.9.4. Values of Integer Attributes

Tartan Ada supports the predefined integer type INTEGER. The range bounds of the predefined type INTEGER are:

    INTEGER'FIRST is -2**31
    INTEGER'LAST is 2**31-1

    LONG_INTEGER'FIRST is -2**63
    LONG_INTEGER'LAST is 2**63-1

    SHORT_INTEGER'FIRST is -2**15
    SHORT_INTEGER'LAST is 2**15-1

    BYTE_INTEGER'FIRST is -128
    BYTE_INTEGER'LAST is 127

The range bounds for subtypes declared in package TEXT_IO are:

    COUNT'FIRST is 0
    COUNT'LAST is INTEGER'LAST - 1

    POSITIVE_COUNT'FIRST is 1
    POSITIVE_COUNT'LAST is INTEGER'LAST - 1

    FIELD'FIRST is 0
    FIELD'LAST is 20

The range bounds for subtypes declared in packages DIRECT_IO are:

    COUNT'FIRST is 0
    COUNT'LAST is INTEGER'LAST

    POSITIVE_COUNT'FIRST is 1
    POSITIVE_COUNT'LAST is COUNT'LAST

5.9.5. Ordinal Types

Ordinal types are supported via a separate package, which is included with the standard packages. Package Ordinal_Support provides support for unsigned arithmetic, including functions which convert between Integer and Ordinal types, and a complete set of Ordinal arithmetic operations. The specification of package Ordinal_Support may be found in the appendix.

5.9.6. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types FLOAT, LONG_FLOAT, and EXTENDED_FLOAT.

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON approximately	16#0.1000_000#E-4 9.536743E-07
SMALL approximately	16#0.8000_000#E-21 2.58494E-26
LARGE approximately	16#0.FFFF_F80#E+21 1.93428E+25
SAFE_EMAX	126
SAFE_SMALL approximately	16#0.2000_000#E-31 5.87747E-39
SAFE_LARGE approximately	16#0.3FFF_FE0#E+32 8.50706+37
FIRST approximately	-16#0.7FFF_FFC#E+32 -1.70141E+38
LAST approximately	16#0.7FFF_FFC#E+32 1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	126
MACHINE_EMIN	-126
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for LONG_FLOAT
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON approximately	16#0.4000_0000_0000_000#E-12 8.8817841970013E-16
SMALL approximately	16#0.8000_0000_0000_000#E-51 1.9446922743316E-62
LARGE approximately	16#0.FFFF_FFFF_FFFF_E00#E+51 2.5711008708143E+61
SAFE_EMAX	1022
SAFE_SMALL approximately	16#0.2000_0000_0000_000#E-255 1.1125369292536-308
SAFE_LARGE approximately	16#0.3FFF_FFFF_FFFF_F80#E+256 4.4942328371557E+307
FIRST approximately	-16#0.7FFF_FFFF_FFFF_FE#E+256 -8.988465674312E+307
LAST approximately	16#0.7FFF_FFFF_FFFF_FE0#E+256 8.9884656743115E+307
MACHINE_RADIX	2
MACHINE_MANTISSA	51
MACHINE_EMAX	1022
MACHINE_EMIN	-1022
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for EXTENDED_FLOAT
DIGITS	18
MANTISSA	61
EMAX	244
EPSILON approximately	16#0.1000_0000_0000_0000_0#E-14 8.67361737988403547E-19
SMALL approximately	16#0.8000_0000_0000_0000_0#E-61 1.76868732008334226E-74
LARGE approximately	16#0.FFFF_FFFF_FFFF_FFF8_0#E+61 2.82695530364541493E+73
SAFE_EMAX	16382
SAFE_SMALL approximately	16#0.2000_0000_0000_0000_0#E-4096 1.68105157155604675E-4932
SAFE_LARGE approximately	16#0.3FFF_FFFF_FFFF_FFFF_0#E+4096 2.97432873839307941E+4931
FIRST approximately	-16#0.7FFF_FFFF_FFFF_FFFF_8#E+4096 -5.94865747678615883E+4931
LAST approximately	16#0.7FFF_FFFF_FFFF_FFFF_8#E+4096 5.94865747678615883E+4931
MACHINE_RADIX	2
MACHINE_MANTISSA	63
MACHINE_EMAX	16382
MACHINE_EMIN	-16382
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

5.10. *SUPPORT FOR PACKAGE MACHINE_CODE*

Package MACHINE_CODE provides the programmer with an interface through which to request the genera-
tion of any instruction that is available on the 80960. The Tartan Ada Sun 960 implementation of package
MACHINE_CODE is similar to that described in Section 13.8 of the Ada LRM, with several added features. Refer
to appendix A of this manual for the specification for package MACHINE_CODE.

5.10.1. *Basic Information*

As required by LRM, Section 13.8, a routine which contains machine code inserts may not have any other
kind of statement, and may not contain an exception handler. The only allowed declarative item is a use clause.
Comments and pragmas are allowed as usual.

5.10.2. *Instructions*

A machine code insert has the form TYPE_MARK' RECORD_AGGREGATE, where the type must be one of the
records defined in package MACHINE_CODE. Package MACHINE_CODE defines four types of records. Each
has an opcode and zero to 3 operands. These records are adequate for the expression of all instructions provided
by the 80960.

5.10.3. *Operands*

An operand consists of a record aggregate which holds all the information to specify it to the compiler. All
operands have an address mode and one or more other pieces of information. The operands correspond exactly to
the operands of the instruction being generated.

5.10.3.1. *Address Modes*

Each operand in a machine code insert must have an *Address_Mode_Name*. The address modes provided in
package MACHINE_CODE provide access to all address modes supported by the 80960.

In addition, package MACHINE_CODE supplies the address modes Symbolic_Address and
Symbolic_Value which allow the user to refer to Ada objects by specifying Object'ADDRESS as the value
for the operand. Any Ada object which has the 'ADDRESS attribute may be used in a symbolic operand.
Symbolic_Address should be used when the operand is a true address (that is, a branch target or the source
of an LDA instruction). Symbolic_Value should be used when the operand is actually a value (that is, one of
the source operands of an ADD instruction).

When an Ada object is used as a *source* operand in an instruction (that is, one from which a value is read), the
compiler will generate code which fetches the *value* of the Ada object. When an Ada object is used as the
destination operand of an instruction, the compiler will generate code which uses the *address* of the Ada object as
the destination of the instruction. See section 5.10.10 for further details.

5.10.4. *Examples*

The Tartan Ada Sun 960 implementation of package MACHINE_CODE makes it possible to specify
both simple machine code inserts such as

```
two_format'(MOV, (Reg_Lit, 5), (Reg, R5))
```

and more complex inserts such as

```
three_format'(MULI,
        (Symbolic_Value, Array_Var(X, Y, 27)'ADDRESS),
        (Lit, 123456),
        (Symbolic_Address, Parameter_1'ADDRESS))
```

In the first example, the compiler will emit the instruction mov 5, r5. In the second example, the compiler
will first emit whatever instructions are needed to form the address of Array_Var(X, Y, 27), load the
value found at that address into a register, load 123456 into a register, and then emit the MULI instruction. If
Parameter_1 is not found in a register, the compiler will put the result of the multiplication in a temporary

register and then store it to Parameter_1'ADDRESS. Note that the destination operand of the MULI instruction is given as a Symbolic_Address. This holds true for all destination operands. The various error checks specified in the LRM will be performed on all compiler-generated code unless they are suppressed by the programmer (either through pragma SUPPRESS, or through command qualifiers).

5.10.5. Incorrect Operands

Under some circumstances, the compiler attempts to correct incorrect operands. Three modes of operation are supplied for package MACHINE_CODE: -Fixup=None, -Fixup=Warn and -Fixup=Quiet. These modes of operation determine whether corrections are attempted and how much information about the necessary corrections is provided to the user. -Fixup=Quiet is the default.

In -Fixup=None mode, the specification of incorrect operands for an instruction is considered to be a fatal error. In this mode, the compiler will not generate any extra instructions to help you to make a machine code insertion. Note that it is still legal to use 'ADDRESS constructs as long as the object which is used meets the requirements of the instruction.

In -Fixup=Quiet mode, if you specify incorrect operands for an instruction, the compiler will do its best to fix up the machine code to provide the desired effect. For example, although it is illegal to use a memory address as the destination of an ADD instruction, the compiler will accept it and try to generate correct code. In this case, the compiler will allocate a temporary register to use as the destination of the ADD, and then store from that register to the desired location in memory.

In -Fixup=Warn mode, the compiler will also do its best to correct any incorrect operands for an instruction. However, a warning message is issued stating that the machine code insert required additional machine instructions to make its operands legal.

The compiler will *always* emit the instruction named in the machine code insert – even if it was necessary to fix up all of its operands. In extreme cases this can lead to surprising code sequences. Consider, for example, the machine code insert

```
    Two_Format'(MOV,    (Reg_Ind,   G0),   (Reg_Ind_Disp,  G1,  128))
```

The MOV instruction requires two registers, but both operands are memory addresses. The compiler will generate a code sequence like

```
    ld       (g0), g12
    mov      g12, g13
    st       g13, 128(g1)
```

Note that the MOV instruction is generated even though a LD ST combination would have been sufficient. As a result of always emitting the instruction specified by the programmer, the compiler will never optimize away instructions which it does not understand (such as SENDSERV), unless they are unreachable by ordinary control flow.

5.10.6. Assumptions Made in Correcting Operands

When compiling in /Fixup=[WARN,QUIET] modes, the compiler attempts to emit additional code to move "the right bits" from an incorrect operand to a place which is a legal operand for the requested instruction. The compiler makes certain basic assumptions when performing these fixups. This section explains the assumptions the compiler makes and their implications for the generated code. Note that if you want a correction which is different from that performed by the compiler, you must make explicit machine_code insertions to perform it.

For source operands:

- Symbolic_Address means that the *address* specified by the 'ADDRESS expression is used as the source bits. When the Ada object specified by the 'ADDRESS instruction is bound to a register, this will cause a compile-time error message because it is not possible to "take the address" of a register.

- Symbolic_Value means that the *value* found at the address specified by the 'ADDRESS expression will be used as the source bits. An Ada object which is bound to a register is correct here, because the contents

- Label indicates that the *address* of the label will be used as the source bits.
- Any other non-register means that the *value* found at the address specified by the operand will be used as the source bits.

For destination operands:

- Symbolic_Address means that the desired destination for the operation is the *address* specified by the 'ADDRESS expression. An Ada object which is bound to a register is correct here; a register is a legal destination on the 960.
- Symbolic_Value means that the desired destination for the operations is found by fetching 32 bits from the address specified by the 'ADDRESS expression, and storing the result to the address represented by the fetched bits. This is equivalent to applying one extra indirection to the address used in the Symbolic_Address case.
- All other operands are interpreted as directly specifying the destination for the operation.

Table 5-1 below describes the correction attempted for each possible instruction-operation combination. The actions shown in the table have the following meanings:

Load to Register 1 The operand given represents a memory location, but the instruction requires a register. The operand is used as a source. The compiler will load from the operand to a temporary register.

Load to Register 2 The operand given represents a register, but the instruction requires a memory location. The operand is a destination. The compiler will store the result value to a scratch memory location, and then load it into the specified register.

Store to Memory 1 The operand given represents a register, but the instruction requires a memory location. The operand is a source. The compiler will store the value to a scratch memory location so that it will be in the proper place for the instruction.

Store to Memory 2 The operand given represents a memory location, but the instruction requires a register. The operand is a destination. The compiler will allocate a scratch register, use that as the destination for the instruction, and then store the result value to the specified memory address.

Store to Memory 3 The operand given is not the address of a label. The operand will be stored to a scratch memory location, and then used as the indirect branch target.

Error 1 The only incorrect operand for the source of an LDA is a register. It is not possible to take the address of a register on the 960.

Error 2 The operand must be a Label'Address.

Inst	Opnd1	Opnd2	Opnd3
addo, addi, addc, addr, addrl	Load to Register 1	Load to Register 1	Store to Memory 2
alterbit	Load to Register 1	Load to Register 1	Store to Memory 2
and, andnot	Load to Register 1	Load to Register 1	Store to Memory 2
atadd	Load to Register 1	Load to Register 1	Store to Memory 2
atanr, atanrl	Load to Register 1	Load to Register 1	Store to Memory 2
atmod	Load to Register 1	Load to Register 1	Store to Memory 2

Table 5-1: Machine_Code Fixup Operations

Inst	Opnd1	Opnd2	Opnd3
b	Error 2		
bx	Store to Memory 3		
bal	Error 2		
balx	Store to Memory 3		
bbc, bbs	Load to Register 1	Load to Register 1	Error 2
BRANCH IF	Error 2		
call	Error 2		
calls	Load to Register 1		
callx	Store to Memory 3		
chkbit	Load to Register 1	Load to Register 1	
classr, classrl	Load to Register 1		
clrbit	Load to Register 1	Load to Register 1	Store to Memory 2
cmpi, cmpo	Load to Register 1	Load to Register 1	
cmpdeci, cmpdeco	Load to Register 1	Load to Register 1	Store to Memory 2
cmpinci, cmpinco	Load to Register 1	Load to Register 1	Store to Memory 2
cmpor, cmporl	Load to Register 1	Load to Register 1	
cmpr, cmprl	Load to Register 1	Load to Register 1	
cmpstr	Load to Register 1	Load to Register 1	Load to Register 1
COMPARE AND BRANCH	Load to Register 1	Load to Register 1	Error 2
concmpi, concmpo	Load to Register 1	Load to Register 1	
condrec	Load to Register 1	Load to Register 1	
condwait	Load to Register 1		
cosr, cosrl	Load to Register 1	Store to Memory 2	
cpyrsre, cpysre	Load to Register 1	Load to Register 1	Store to Memory 2
cvtilr	Load to Register 1 (64 bits)	Store to Memory 2	
cvtir	Load to Register 1	Store to Memory 2	
cvtri	Load to Register 1	Store to Memory 2	
cvtril	Load to Register 1	Store to Memory 2 (64 bits)	
cvtzri	Load to Register 1	Store to Memory 2	
cvtzril	Load to Register 1	Store to Memory 2 (64 bits)	
daddc	Load to Register 1	Load to Register 1	Store to Memory 2
divo, divi, divr, divrl	Load to Register 1	Load to Register 1	Store to Memory 2

Table 5-1. Machine Code Fixup Operations

Inst	Opnd1	Opnd2	Opnd3
dmovt	Load to Register 1	Store to Memory 2	
dsubc	Load to Register 1	Load to Register 1	Store to Memory 2
ediv	Load to Register 1	Load to Register 1 (64 Bits)	Store to Memory 2 (64 bits)
emul	Load to Register 1	Load to Register 1	Store to Memory 2 (64 bits)
expr, exprl	Load to Register 1	Store to Memory 2	
extract	Load to Register 1	Load to Register 1	Store to Memory 2
FAULT IF			
fill	Load to Register 1	Load to Register 1	Load to Register 1
flushreg			
fmark			
inspacc	Load to Register 1	Store to Memory 2	
LOAD	Store to Memory 1	Store to Memory 2	
lda	Error 1	Store to Memory 2	
ldphy	Load to Register 1	Store to Memory 2	
ldtime	Store to Memory 2		
logbnr, logbnrl	Load to Register 1	Store to Memory 2	
logepr, logeprl	Load to Register 1	Store to Memory 2	
logr, logrl	Load to Register 1	Store to Memory 2	
mark			
modac	Load to Register 1	Load to Register 1	Store to Memory 2
modi	Load to Register 1	Load to Register 1	Store to Memory 2
modify	Load to Register 1	Load to Register 1	Store to Memory 2
modpc	Load to Register 1	Load to Register 1	Store to Memory 2
modtc	Load to Register 1	Load to Register 1	Store to Memory 2
MOVE	Load to Register 1	Store to Memory 2	
movqstr, movstr	Load to Register 1	Load to Register 1	Load to Register 1
mulo, muli, mulr, mulrl	Load to Register 1	Load to Register 1	Store to Memory 2
nand	Load to Register 1	Load to Register 1	Store to Memory 2
nor	Load to Register 1	Load to Register 1	Store to Memory 2
not	Load to Register 1	Store to Memory 2	
notand	Load to Register 1	Load to Register 1	Store to Memory 2
notbit	Load to Register 1	Load to Register 1	Store to Memory 2

Table 5-1: Machine_Code Fixup Operations

Inst	Opnd1	Opnd2	Opnd3
notor	Load to Register 1	Load to Register 1	Store to Memory 2
or, ornot	Load to Register 1	Load to Register 1	Store to Memory 2
recieve	Load to Register 1	Load to Register 1	
remo, remi, remr, remrl	Load to Register 1	Load to Register 1	Store to Memory 2
resumprcs	Load to Register 1		
ret			
rotate	Load to Register 1	Load to Register 1	Store to Memory 2
roundr, roundrl	Load to Register 1	Store to Memory 2	
saveprcs			
scaler, scalerl	Load to Register 1	Load to Register 1	Store to Memory 2
scanbit	Load to Register 1	Store to Memory 2	
scanbyte	Load to Register 1	Load to Register 1	
schedprcs	Load to Register 1		
send	Load to Register 1	Load to Register 1	Load to Register 1
sendserv	Load to Register 1		
setbit	Load to Register 1	Load to Register 1	Store to Memory 2
SHIFT	Load to Register 1	Load to Register 1	Store to Memory 2
signal	Load to Register 1		
sinr, sinrl	Load to Register 1	Store to Memory 2	
spanbit	Load to Register 1	Store to Memory 2	
sqrtr, sqrtrl	Load to Register 1	Store to Memory 2	
STORE	Load to Register 1	Load To Register 2	
subo, subi, subc, subr, subrl	Load to Register 1	Load to Register 1	Store to Memory 2
syncf			
synld	Load to Register 1	Store to Memory 2	
synmov, synmovl, syn-movq	Load To Register 1	Load to Register 1	
tanr, tanrl	Load to Register 1	Store to Memory 2	
TEST	Store to Memory 2		
wait	Load to Reg 1		
xnor, xor	Load to Register 1	Load to Register 1	Store to Memory 2

Table 5-1: Machine_Code Fixup Operations

5.10.7. Register Usage

Since the compiler may need to allocate registers as temporary storage in machine code routines, there are some restrictions placed on your register usage. The compiler will automatically free all the registers which would be volatile across a call for your use (that is, g0..g7, g13, and g14). If you reference any other register, the compiler will reserve it for your use until the end of the machine code routine. The compiler will *not* save the register automatically. This means that the first reference to a register which is not volatile across calls should be an instruction which saves its value in a safe place. The value of the register should be restored at the end of the machine code routine. This rule will help ensure correct operation of your machine code insert even if it is inline expanded in another routine.

The compiler may need several registers to generate code for operand fixups in machine code inserts. If you use all the registers, fixups will not be possible. If a fixup is needed, the compiler may require up to three registers to guarantee success. In general, when more registers are available to the compiler it is able to generate better code.

5.10.8. Inline Expansion

Routines which contain machine code inserts may be inline expanded into the bodies of other routines. This may happen under programmer control through the use of pragma INLINE, or at Optimization Level 3 when the compiler selects that optimization as an appropriate action for the given situation. The compiler will treat the machine code insert as though it was a call; volatile registers will be saved and restored around it, etc.

5.10.9. Unsafe Assumptions

There are a variety of assumptions which should *not* be made when writing machine code inserts. Violation of these assumptions may result in the generation of code which does not assemble or which may not function correctly.

- Do not assume that a machine code insert routine has its own set of local registers. This may not be true if the routine is inline expanded into another routine. Explicitly save and restore any registers which are not volatile across calls. If you wish to guarantee that a routine will never be inline expanded, you should use an Ada separate body for the routine and make sure that there is no pragma INLINE for it.

- Do not attempt to move multiple Ada objects with a single long instruction such as MOVL or STT. Although the objects may be contiguous under the current circumstances, there is no guarantee that later changes will permit them to remain contiguous. If the objects are parameters, it is virtually certain that they will not be contiguous if the routine is inline expanded into the body of another routine. In the case of locals, globals, and own variables, the compiler does not guarantee that objects which are declared textually "next" to each other will be contiguous in memory. If the source code is changed such that it declares additional objects, this may change the storage allocation such that objects which were previously adjacent are no longer adjacent.

- The compiler *will not* generate call site code for you if you emit a call instruction. You must save and restore any volatile registers which currently have values in them, etc. If the routine you call has out parameters, a large function return result, or an unconstrained result, it is your responsibility to emit the necessary instructions to deal with these constructs as the compiler expects. In other words, when you emit a call, you must follow the linkage conventions of the routine you are calling. For further details on call site code, see Sections 6.4, 6.5 and 6.6.

- Do not assume that the 'ADDRESS on Symbolic_Address or Symbolic_Value operands means that you are getting an ADDRESS to operate on. The Address- or Value-ness of an operand is determined by your choice of Symbolic_Address or Symbolic_Value. This means that to add the *contents* of X to r3, you should write

```
Three_Format' (ADDI, (Symbolic_Value, X'ADDRESS),
                      (Reg, R3), (Reg, R3));
```

```
Three_Format' (ADDI,  (Symbolic_Address, X'ADDRESS),
                      (Reg, R3), (Reg, R3));
```

- The compiler will not prevent you from writing register r3 (which is used to hold the address of the current exception handler). This provides you the opportunity to make a custom exception handler. Be aware, however, that there is considerable danger in doing so. Knowledge of the details on the structure of exception handlers will help; see the *Tartan Ada Runtime Implementor's Guide*.

5.10.10. Limitations

- When specifying absolute addresses in machine_code inserts, please note that the compiler will treat addresses as an INTEGER type. This means that specifications of addresses may raise arithmetic overflow errors; i.e., addresses must be in the range INTEGER'FIRST..INTEGER'LAST. To represent an address greater than INTEGER'LAST, use the negated radix-complement of the desired address. For example, to express address 16#C000_000, specify instead -16#4000_000.

- The current implementation of the compiler is unable to fully support automatic fixup of certain kinds of operands. In particular, the compiler assumes that the size of a data object is the same as the number of bits which is operated on by the instruction chosen in the machine code insert. This means that the insert:

```
Three_Format' (ADDO,  (Symbolic_Value, Byte_Variable'ADDRESS),
                      (Reg, R0), (Reg, R1))
```

will not generate correct code when Byte_Variable is bound to memory. The compiler will assume that Byte_Variable is 32 bits, when in fact it is only 8, and will emit an LD instruction to load the value of Byte_Variable into a register. If, on the other hand, Byte_Variable was bound to a register the insertion will function properly, as no fixup is needed.

- The compiler generates incorrect code when the BAL and BALX instructions are used with symbolic operands which are not of the form Routine'ADDRESS. To get the effect of an unconditional branch, use the B or BX instructions instead.

- Note that the use of X'ADDRESS in a machine code insert *does not* guarantee that X will be bound to memory. This is a result of the use of 'ADDRESS to provide a "typeless" method for naming Ada objects in machine code inserts. For example, it is legal to say to (Symbolc_Value, X'ADDRESS) in an insert even when X is a formal parameter of the machine code routine (and is thus found in a register).

5.10.11. Example

```
package mtest is
 type ary_type is array(1..4) of integer;

 procedure inline_into_me;
end mtest;


with machine_code;
use machine_code;
package body mtest is

 own_var : integer := -1;

 procedure mach_test(x, y, z: in integer; ary: in out ary_type) is
 begin
     -- The next instruction is only OK if this routine is not INLINED.
     -- If the routine is inlined, there is no guarantee that parm X will
     -- be either A) in an even numbered register, or B) "next to" parm
     -- Y.  If the programmer uses an instruction like MOVLp here, he is
     -- assuming too much about the generated code; his program is
     -- erroneous.  On the other hand, the use of x'ADDRESS does guarantee
     -- that the instruction will use X even when this routine is inline
     -- expanded into a caller.
```

```
Two_Format'(MOVL, (Symbolic_Value, x'ADDRESS), (Reg, G6));
Two_Format'(MOV, (Symbolic_Value, x'ADDRESS), (Reg, G6));
Two_Format'(MOV, (Symbolic_Value, y'ADDRESS), (Reg, G7));
Two_Format'(MOV, (Symbolic_Value, z'ADDRESS), (Reg, G8));
Three_Format'(ADDI, (Symbolic_Value, x'ADDRESS), (Reg, G8), (Reg, G1));


Three_Format'(MULI,
              (Reg, G7),
              (Symbolic_Value, y'ADDRESS),
              (Reg, G12));

-- Note the use of a complicated Ada object in this instruction.
Two_Format'(ST,
              (Reg, G12),
              (Symbolic_Address, ary(1)'ADDRESS));

-- In this instruction, note that ary(1)'ADDRESS is NOT kept in a
-- register and is thus NOT a legal source for XORp.  That's OK,
-- because the compiler can fix it up for the user.
Three_Format'(XORi, (Symbolic_Value, ary(1)'ADDRESS),
              (Symbolic_Value, ary(2)'ADDRESS), (Reg, G12));

Two_Format'(ST,
              (Reg, G12),
              (Symbolic_Address, ary(3)'ADDRESS));

Two_Format'(ST,
              (Reg, Gi2),
              (Symbolic_Address, ary(x)'ADDRESS));
Two_Format'(ST,
              (Reg, G0),
              (Symbolic_Address, own_var'ADDRESS));

Two_Format'(LDA,
              (Symbolic_Value, own_var'ADDRESS),
              (Reg, G14));

One_Format'(CALLX, (Symbolic_Address, inline_into_me'ADDRESS));
end mach_test;
pragma inline(mach_test);

procedure mtestl(first, second, third: in integer; fourth: out ary_type) is
begin

-- Note the use of fourth(1)'ADDRESS as the destination of the MOV
-- instruction.  The compiler will understand that the user "really
-- wanted" something moved to fourth(1)'ADDRESS, and will make sure
-- that the bits get there.  The compiler does NOT assume that it
-- knows enough to second guess the user's choice of instructions.
-- we generate the MOV, followed by a store to memory.
Two_Format'(MOV,
              (Symbolic_Value, First'Address),
              (Symbolic_Address, fourth(1)'ADDRESS));
end mtestl;

procedure inline_into_me is
  array1 : ary_type := (1, 2, 3, 4);
begin
  if array1(3) >= 0 then
      -- note that mach_test is inline expanded
      mach_test(22, 41, array1(4), array1);
  else
      -- but mtest is not at Op=2 (No pragma inline)
      mtestl(1, 2, 3, array1);
  end if;
```

```
     end inline_into_me;

   end mtest;
```

Assembly code output:

```
# Generated from USER01:[SMITH]MTEST.ADB;1
# by TARTAN Ada Compiler VMS 80960MC, Version Pre-Release

        .data

        .align 2
ADA.OWN:        .space   4
        .align 2
ADA.GLOBAL:     .space   1
        .globl  xxmtest$inline_into_me$00
        .globl  xxmtest$declare
        .globl  xxmtest$body

.seto   own_var$00,ADA.OWN,0
        .globl  xxmtest$inline_into_me$goto$00
.seto   xxmtest$inline_into_me$goto$00,ADA.GLOBAL,0


        .text

        .align 4

xxmtest$inline_into_me$00:
        mov     0,r3
        st      sp,40(sp)
        lda     40(sp),sp
        st      g12,100(fp)
        lda     .L19,r3

        mov     1,g13    #              line 74

        st      g13,80(fp)
        mov     2,g13
        st      g13,84(fp)
        mov     3,g13
        st      g13,88(fp)
        mov     4,g13
        st      g13,92(fp)
        ldq     80(fp),g4
        stq     g4,64(fp)
        ld      72(fp),g13     #              line 76

        cmpibg  0,g13,.L17
        ld      76(fp),g13     #              line 78

        mov     22,r4
        addc    31,10,r5
        mov     g13,r6
        lda     64(fp),r8
        movl    r4,g6
        mov     r4,g6
        mov     r5,g7
        mov     r6,g8
        addi    r4,g8,g1
        muli    g7,r5,g12
        st      g12,96(fp)
        ld      96(fp),g13
        st      g13,(r8)
        ld      4(r8),r7
        xor     g13,r7,g12
        st      g12,8(r8)
        subi    1,r4,g13       #              line 46

        cmpo    g13,3
```

```
            faultg
            st        g12,-4(r8)[r4*4]
            st        g0,ADA.OWN
            lda       ADA.OWN,g14
            callx     xxmtest$inline_into_me$00
            b         .L19    #                   line 76

.L17:       mov       1,g0    #                   line 81

            mov       2,g1
            mov       3,g2
            lda       64(fp),g3
            bal       mtest1$00
.L19:

            ld        100(fp),g12
            ret

# Total bytes of code in the above routine = 216

            .align 4

mach_test$00:
            mov       0,r3
            st        sp,8(sp)
            addo      8,sp,sp
            st        g12,68(fp)
            lda       .L21,r3

            movl      g0,g6
            mov       g0,g6
            mov       g1,g7
            mov       g2,g8
            addi      g0,g8,g1
            muli      g7,g1,g12
            st        g12,64(fp)
            ld        64(fp),g13
            st        g13,(g3)
           ·ld        4(g3),g5
            xor       g13,g5,g12
            st        g12,8(g3)
            subi      1,g0,g13            #                   line 46

            cmpo      g13,3
            faultg
            st        g12,-4(g3)[g0*4]
            st        g0,ADA.OWN
            lda       ADA.OWN,g14
            callx     xxmtest$inline_into_me$00
.L21:

            ld        68(fp),g12
            ret

# Total bytes of code in the above routine = 124

            .align 4

mtest1$00:
            mov       g14,g4
            mov       0,g14

            mov       g0,g13
            st        g13,(g3)

            bx        (g4)
```

```
# Total bytes of code in the above routine = 20

        .align 4

xxmtest$declare:

        stob    g14,ADA.GLOBAL

        ret

# Total bytes of code in the above routine = 12

        .align 4

xxmtest$body:

        not     0,g13   #                       line 5

        st      g13,ADA.OWN
        mov     1,g13
        stob    g13,ADA.GLOBAL

        ret

# Total bytes of code in the above routine = 28

        .text

        .align 2
        .align 2
# Total bytes of code = 400
# Total bytes of data = 5
```

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1-1