AD-A220 908

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(2)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) **Ada Compiler Validation Summary Report:** InterACT Corporation, InterACT Ada Mips Cross-Compiler System Release 1.0 MicroVAX 3100 Cluster (Host) to MIPs R2000 in an Integraded Solutions (Target), 891116S1.10233 | | 5. TYPE OF REPORT & PERIOD COVERED 16 Nov. 1989 to 16 Nov. 1990 |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) National Institute of Standards and Technology Gaithersburg, Maryland, USA | | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION AND ADDRESS National Institute of Standards and Technology Gaithersburg, Maryland, USA | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS **Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081** | | 12. REPORT DATE |
| | | 13. NUMBER OF PAGES |
| 14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) National Institute of Standards and Technology Gaithersburg, Maryland, USA | | 15. SECURITY CLASS (of this report) UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A |

16. DISTRIBUTION STATEMENT (of this Report)

**Approved for public release; distribution unlimited.**

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

DTIC ELECTE APR 26 1990 S D D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

**Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO**

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

InterACT Ada Mips Cross-Compiler System Release 1.0, Gaithersburg, MD, MicroVAX 3100 Cluster under VMS 5.2 (Host) to MIPS R2000 in an Integrated Solutions, INC Advantedge 2000 Board (bare machine)(Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

90 04 24 095

AVF Control Number: NIST89ACT575_2_1.10
1 February 1990

# Ada COMPILER
# VALIDATION SUMMARY REPORT:
Certificate Number: 891116S1.10233
InterACT Corporation
InterACT Ada Mips Cross-Compiler System Release 1.0
MicroVAX 3100 Cluster Host and MIPS R2000 in an Integrated
Solutions, INC Advantedge 2000 Board (bare machine)



Completion of On-Site Testing:
16 November 1989

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899



Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: InterACT Ada Mips Cross-Compiler System Release 1.0

Certificate Number: 891116S1.10233

Host: MicroVAX 3100 Cluster under VMS 5.2

Target: MIPS R2000 in an Integrated Solutions, INC
Advantedge 2000 Board (bare machine)

Testing Completed 16 November 1989 Using ACVC 1.10

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

for Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

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

TABLE OF CONTENTS

# 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 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by GEMMA Corporation under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 16 November 1989 at InterACT Corporation, 417 Fifth Avenue, New York, New York, 10016.

## 1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO 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 compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

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

## 1.3 REFERENCES


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

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, Version 2.0, May 1989.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.


## 1.4 DEFINITION OF TERMS

ACVC — The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary — An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard — ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant — The agency requesting validation.

AVF — The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.

AVO — The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

| | |
|---|---|
| Compiler | A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters. |
| Failed test | An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard. |
| Host | The computer on which the compiler resides. |
| Inapplicable test | An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test |
| Passed test | An ACVC test for which a compiler generates the expected result. |
| Target | The computer which executes the code generated by the compiler. |
| Test | A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files. |
| Withdrawn | An ACVC test found to be incorrect and not used to check test conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language. |

## 1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada 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, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

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 every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It 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 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. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

# CHAPTER 2

# CONFIGURATION INFORMATION

## 2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: InterACT Ada Mips Cross-Compiler System Release 1.0

ACVC Version: 1.10

Certificate Number: 891116S1.10233

Host Computer:

- Machine: MicroVAX 3100 Cluster
- Operating System: VMS 5.2
- Memory Size: 32MB

Target Computer:

- Machine: MIPS R2000 in an Integrated Solutions INC Advantedge 2000 Board
- Operating System: bare machine
- Memory Size: 4MB

Communications Network: RS232 Link

The Ada program is compiled on the MicroVAX, the InterACT Embedded Systems Linker is run under VAX/VMS and produces a Mips load module in InterACT's own format.

The execution controllers are a pair of cooperating processes. The Remote Process Administrator (RPA) runs under VAX/VMS, and is a translator/downloader. The Remote Process Monitor (RPM) runs on the target Mips machine. They communicate via a RS2332 link.

The RPA is invoked with a Mips load module as input which is translated into one or more Unix-style (a.out) format files. The RPA then instructs the RPM to download file(s) via a pair of Ethernet server/client processes. It then directs the RPM to start execution of the Ada program. As the Ada program executes, it calls the RPM to perform input/output. When the Ada program finishes its execution, it gives control back to the RPM. The RPA then gives control back to the user in VAX/VMS.

## 2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined type LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language.

While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) All of the default initialization expressions for record components are not evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised when a literal operand in a comparison or membership is outside the range of predefined Integer and in a comparison or membership test that is greater than System.Max_Int. No exception is raised when an integer literal operand in a comparison is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual.(See tests C45524A..Z(26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z (26 tests).)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z (26 tests).)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds

STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

(2) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array objects are declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises a NUMERIC_ERROR when the subtype is declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, the

test results indicate that index subtype checks are made as choices are evaluated. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is supported for functions or procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

(3) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(4) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

(5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs.(See test CA2009F.)

(6) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

(7) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

(8) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

j. Input and output.

(1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types or record types with

discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO can be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO except for text IO standard input and standard output.

# CHAPTER 3

# TEST INFORMATION

## 3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 718 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 8 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

## 3.2 SUMMARY OF TEST RESULTS BY CLASS

| RESULT | TEST CLASS | | | | | | TOTAL |
|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | L | |
| Passed | 123 | 1126 | 1630 | 17 | 13 | 46 | 2955 |
| Inapplicable | 6 | 12 | 685 | 0 | 15 | 0 | 718 |
| Withdrawn | 1 | 2 | 35 | 0 | 6 | 0 | 44 |
| TOTAL | 130 | 1140 | 2350 | 17 | 34 | 46 | 3717 |

## 3.3 SUMMARY OF TEST RESULTS BY CHAPTER

| RESULT | CHAPTER | | | | | | | | | | | | | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
| Passed | 195 | 572 | 532 | 242 | 172 | 99 | 158 | 331 | 135 | 36 | 250 | 157 | 76 | 2955 |
| Inapplicable | 17 | 77 | 148 | 6 | 0 | 0 | 8 | 1 | 2 | 0 | 2 | 212 | 245 | 718 |
| Wdrn | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 35 | 4 | 44 |
| TOTAL | 213 | 650 | 680 | 248 | 172 | 99 | 166 | 334 | 137 | 36 | 253 | 404 | 325 | 3717 |

## 3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

| | | | | | |
|---|---|---|---|---|---|
| A39005G | B97102E | C97116A | BC3009B | CD2A62D | CD2A63A |
| CD2A63B | CD2A63C | CD2A63D | CD2A66A | CD2A66B | CD2A66C |
| CD2A66D | CD2A73A | CD2A73B | CD2A73C | CD2A73D | CD2A76A |
| CD2A76B | CD2A76C | CD2A76D | CD2A81G | CD2A83G | CD2A84M |
| CD2A84N | CD2B15C | CD2D11B | CD5007B | CD50110 | CD7105A |
| CD7203B | CD7204B | CD7205C | CD7205D | CE2107I | CE3111C |
| CE3301A | CE3411B | E28005C | ED7004B | ED7005C | ED7005D |
| ED7006C | ED7006D | | | | |

See Appendix D for the reason that each of these tests was withdrawn.

## 3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 718 tests were inapplicable for the reasons indicated:

a. The following three tests, C24113I..K, are not applicable because the max line length of 126 characters is exceeded.

b. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

| | |
|---|---|
| C24113L..Y (14 tests) | C35705L..Y (14 tests) |
| C35706L..Y (14 tests) | C35707L..Y (14 tests) |
| C35708L..Y (14 tests) | C35802L..Z (15 tests) |
| C45241L..Y (14 tests) | C45321L..Y (14 tests) |
| C45421L..Y (14 tests) | C45521L..Z (15 tests) |
| C45524L..Z (15 tests) | C45621L..Z (15 tests) |
| C45641L..Y (14 tests) | C46012L..Z (15 tests) |

c. The following 170 tests are not applicable because 'SIZE representation clauses are not supported.

| | |
|---|---|
| A39005B | CD1009B |
| CD1009P | CD2A21A..E (5 TESTS) |
| CD2A22A..J (10 TESTS) | CD2A23A..E (5 TESTS) |
| CD2A24A..J (10 TESTS) | CD2A31A..D (4 TESTS) |
| CD2A32A..J (10 TESTS) | CD2A41A..E (5 TESTS) |
| CD2A42A..J (10 TESTS) | CD2A51B..E (4 TESTS) |
| CD2A52A..D (4 TESTS) | CD2A52G..J (4 TESTS) |
| CD2A53A..E (5 TESTS) | CD2A54A..D (4 TESTS) |
| CD2A54G..J (4 TESTS) | CD2A64A..D (4 TESTS) |
| CD2A65A..D (4 TESTS) | CD2A61A..L (12 TESTS) |
| CD2A62A..C (3 TESTS) | CD2A71A..D (4 TESTS) |
| CD2A72A..D (4 TESTS) | CD2A74A..D (4 TESTS) |
| CD2A75A..D (4 TESTS) | CD2A81A..F (6 TESTS) |
| CD2A83A..C (3 TESTS) | CD2A83E..F (2 TESTS) |
| CD2A84B..I (8 TESTS) | CD2A84K..L (2 TESTS) |
| CD2A87A | CD2A91A..E (5 TESTS) |
| CD1C03A | CD1C04A |
| CD1C04C | CD1009A |
| CD1009C..I (7 TESTS) | CD1009O |
| CD1009Q | ED2A26A |
| ED2A56A | ED2A86A. |

d. C35508I, C35508J, C35508M, C35508N, C87B62A, AD1C04D, AD3015C, AD3015F, AD3015H, AD3015K, CD1C04B, CD1C04E, CD3015A,B,D,E (4 TESTS), CD3015G, CD3015I, CD3015J, CD3015L, CD4051A..D (4 TESTS) these 24 tests are not applicable because representation clauses are not supported for derived types.

e. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

f. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

| | | | |
|---|---|---|---|
| C45231B | C45304B | C45502B | C45503B |
| C45504B | C45504E | C45611B | C45613B |
| C45614B | C45631B | C45632B | B52004E |
| C55B07B | B55B09D | B86001V | CD7101E |

g. C45531M..P(4 tests) and C45532M..P(4 tests) aren't applicable because the fixed point definitions are not supported.

h. C4A013B is not applicable because the evaluation of an expression involving 'MACHINE_RADIX applied to the most precise floating-point type would raise an exception; since the expression must be static, it is rejected at compile time.

i. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

| | | | |
|---|---|---|---|
| B52004D | B55B09C | B86001W | C45231C |
| C45304C | C45502C | C45503C | C45504C |
| C45504F | C45611C | C45613C | C45614C |
| C45631C | C45632C | C55B07A | CD7101F |

j. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER or LONG_INTEGER.

k. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

l. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

m. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

n. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

o. CA2009C, CA2009F, BC3204C, and BC3205D are not applicable because this implementation requires that generic bodies be located in the same file or precede the instantiation. In these four tests the Generic bodies are all in separate files and those files come after the instantiation. If either of these two conditions were reversed, the tests would report passed and would then be applicable.

p. The following 21 tests are not applicable because, for this implementation, Address clauses for constants are not supported:

| | | | | |
|---|---|---|---|---|
| CD5011B | CD5011D | CD5011F | CD5011H | |
| CD5011L | CD5011N | CD5011R | CD5012C | |
| CD5012D | CD5012G | CD5013B | CD5012H | |
| CD5012L | CD5013D | CD5013F | CD5013H | |
| CD5013L | CD5013N | CD5013R | CD5014U | CD5014W |

q. The following 245 tests are inapplicable because sequential, text, and direct access files are not supported:

| | |
|---|---|
| CE2102A..C (3 tests) | CE2102G..H (2 tests) |
| CE2102K | CE2102N..Y (12 tests) |
| CE2103A..D (4 tests) | CE2104A..D (4 tests) |
| CE2105A..B (2 tests) | CE2106A..B (2 tests) |
| CE2107A..H (8 tests) | CE2107L |
| CE2108A..B (2 tests) | CE2108C..H (6 tests) |
| CE2109A..C (3 tests) | CE2110A..D (4 tests) |
| CE2111A..I (9 tests) | CE2115A..B (2 tests) |
| CE2201A..C (3 tests) | CE2201F..N (9 tests) |
| CE2204A..D (4 tests) | CE2205A |
| CE2208B | CE2401A..C (3 tests) |
| CE2401E..F (2 tests) | CE2401H..L (5 tests) |
| CE2404A..B (2 tests) | CE2405B |
| CE2406A | CE2407A..B (2 tests) |
| CE2408A..B (2 tests) | CE2409A..B (2 tests) |
| CE2410A..B (2 tests) | CE2411A |
| CE3102A..B (2 tests) | CE3102F..H (3 tests) |
| CE3102J..K (2 tests) | CE3103A |
| CE3104A..C (3 tests) | CE3107A..B (2tests) |
| CE3108A..B (2 tests) | CE3109A |
| CE3110A | CE3111A..B (2 tests) |
| CE3111D..E (2 tests) | CE3112A..D (4 tests) |
| CE3114A..B (2 tests) | CE3115A |
| CE3208A | CE3302A |
| CE3305A | CE3402A |
| CE3402C..D (2 tests) | CE3403A..C (3 tests) |
| CE3403E..F (2 tests) | CE3404B..D (3 tests) |
| CE3405A | CE3405C..D (2 tests) |
| CE3406A..D (4 tests) | CE3407A..C (3 tests) |
| CE3408A..C (3 tests) | CE3409A |
| CE3409C..E (3 tests) | CE3410A |
| CE3410C..E (3 tests) | CE3411A,C (2 tests) |
| CE3412A | CE3413A |
| CE3413C | CE3602A..D (4 tests) |
| CE3603A | CE3604A..B (2 tests) |
| CE3605A..E (5 tests) | CE3606A..B (2 tests) |
| CE3704A..F (6 tests) | CE3704M..O (3 tests) |
| CE3706D | CE3706F..G (2 tests) |
| CE3804A..P (16 tests) | CE3805A..B (2 tests) |
| CE3806A..B (2 tests) | CE3806D..E (2 tests) |
| CE3806G..H (2 tests) | CE3905A..C (3 tests) |
| CE3905L | CE3906A..C (3 tests) |

| | |
|---|---|
| CE3906E..F (2 tests) | EE2201D |
| EE2201E | EE2401D |
| EE2401G | EE3102C |
| EE3203A | EE3301B |
| EE3402B | EE3405B |
| EE3409F | EE3410F |
| EE3412C | |

### 3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 8 tests.

Modification was required for 1 A Class test, AD7006A. The assignment at line 23 was modified via use of a number declaration, as there is no predetermined integer type whose range includes SYSTEM.MEMORY_SIZE.

The following 6 B Class tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B33301B B55A01A BA1101B BC1109A BC1109C BC1109D

Modification was required for 1 C Class test, C87B62B. A length clause specifying a collection size for type JUST_LIKE_LINK was added to prevent CHECK from raising a STORAGE_ERROR.

### 3.7 ADDITIONAL TESTING INFORMATION

#### 3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the InterACT Ada Mips Cross-Compiler System Release 1.0 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

#### 3.7.2 Test Method

Testing of the InterACT Ada Mips Cross-Compiler System Release 1.0 using

ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

| | |
|---|---|
| Host computer: | MicroVAX 3100 Cluster |
| Host operating system: | VMS 5.2 |
| Target computer: | Mips R2000 in an Integrated Solutions, INC. Advantedge 2000 Board |
| Target operating system: | Bare machine |
| Compiler: | InterACT Ada Mips Cross-Compiler System Release 1.0 |

The host and target computers were linked via RS232.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precision was taken on-site by the validation team for processing. This tape could not be read so the prevalidation test suite which was already on disk was used to perform the on site validation. A copy of this test suite was placed on magnetic tape and subsequently compared to the original on site test suite and shown to be the same.

TEST INFORMATION

The test suite resided on disk. The full set of tests was compiled and run on the MicroVAX 3100 cluster and all executable tests were transferred to the Mips R2000 via RS232 and run. Results were printed from the host computer.

The compiler was tested using command scripts provided by InterACT and reviewed by the validation team. The compiler was tested using all default option settings.

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

### 3.7.3 Test Site

Testing was conducted at InterACT Corporation, 417 Fifth Avenue, New York, N Y 10016 and was completed on 16 November 1989.

# APPENDIX A

## DECLARATION OF CONFORMANCE

InterACT has submitted the following Declaration of Conformance concerning the InterACT Ada Mips Cross-Compiler System.

# APPENDIX A

## Declaration of Conformance

**Customer:** InterACT Corporation

**Ada Validation Facility:** National Institute of Standards & Technology

**ACVC Version:** 1.10

**Ada Implementation**

Ada Compiler Name: InterACT Ada MIPS Cross-Compiler System

Version: 1.0

Host Computer System: MicroVAX 3100 Cluster VMS 5.2

Target Computer System: MIPS R2000 in an Integrated Solutions, Inc. Advantedge2000 board (bare machine)

**Customer's Declaration**

I, the undersigned, representing InterACT declare that InterACT has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.

Signature

Edward D. Bright
CEO

1/16/89
Date

## APPENDIX B

## 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 the InterACT Ada Mips Cross-compiler System Release 1.0, as described in this Appendix, are provided by InterACT Corp. 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 Appendix F, are:

```
package STANDARD is

...

 type INTEGER is range -2_147_483_648..2_147_483_647;

 type FLOAT is digits 6,
      range -2#1.0#E126..2#0.111111111111111111111#E126

 type LONG_FLOAT is digits 15
      range 2#1.0#E1024..
       2#0.11111111111111111111111111111111111111111111111111#E1024;

 type DURATION is delta 2**(-14) range -131_072.0..131_071.0;

...

end STANDARD;
```

# Appendix F updated
# Appendix F of the Ada Reference Manual

This appendix describes all implementation-dependent characteristics of the Ada language as implemented by the InterACT Ada Mips Cross-Compiler, including those required in the Appendix F frame of *Ada RM*.

## F.1. Predefined Types in Package STANDARD

This section describes the implementation-dependent predefined types declared in the predefined package STANDARD [*Ada RM Annex C*], and the relevant attributes of these types.

### Integer Types

One predefined integer type is implemented, INTEGER. It has the following attributes:

| | | |
|---|---|---|
| INTEGER'FIRST | = | -2_147_483_648 |
| INTEGER'LAST | = | 2_147_483_647 |
| INTEGER'SIZE | = | 32 |

### Floating Point Types

Two predefined floating point types are implemented, FLOAT and LONG_FLOAT. They have the following attributes:

| | | |
|---|---|---|
| FLOAT'DIGITS | = | 6 |
| FLOAT'FIRST | = | -2#1.0#E126 |
| FLOAT'LAST | = | 2#0.111111111111111111111111#E126 |
| FLOAT'MACHINE_EMAX | = | 128 |
| FLOAT'MACHINE_EMIN | = | -126 |
| FLOAT'MACHINE_MANTISSA | = | 24 |
| FLOAT'MACHINE_OVERFLOWS | = | TRUE |
| FLOAT'MACHINE_RADIX | = | 2 |
| FLOAT'MACHINE_ROUNDS | = | TRUE |
| FLOAT'SAFE_EMAX | = | 126 |
| FLOAT'SAFE_LARGE | = | 2#0.111111111111111111111111#E126 |
| FLOAT'SAFE_SMALL | = | 2#0.1#E-126 |
| FLOAT'SIZE | = | 32 |

| | | |
|---|---|---|
| LONG_FLOAT'DIGITS | = | 15 |
| LONG_FLOAT'FIRST | = | -2#1.0#E1024 |
| LONG_FLOAT'LAST | = | 2#0.11111111111111111111111111111111111111111111111111111# :4 |
| LONG_FLOAT'MACHINE_EMAX | = | 1024 |
| LONG_FLOAT'MACHINE_EMIN | = | -1022 |
| LONG_FLOAT'MACHINE_MANTISSA | = | 53 |
| LONG_FLOAT'MACHINE_OVERFLOWS | = | TRUE |
| LONG_FLOAT'MACHINE_RADIX | = | 2 |
| LONG_FLOAT'MACHINE_ROUNDS | = | TRUE |
| LONG_FLOAT'SAFE_EMAX | = | 1024 |
| LONG_FLOAT'SAFE_LARGE | = | 2#0.11111111111111111111111111111111111111111111111111111# .24 |
| LONG_FLOAT'SAFE_SMALL | = | 2#0.1#E-1024 |
| LONG_FLOAT'SIZE | = | 64 |

**Fixed Point Types**

One kind of anonymous predefined fixed point type is implemented: *fixed*. Note that this name is not defined in package STANDARD, but is used here only for reference.

For objects of *fixed* types, 32 bits are used for the representation of the object.

For *fixed* there is a virtual predefined type for each possible value of *small* [*Ada RM 3.5.9*]. The possible values of *small* are the powers of two that are representable by a LONG_FLOAT value (or if a length clause is used, any number representable by a LONG_FLOAT value).

The lower and upper bounds of these types are:

| | | |
|---|---|---|
| lower bound of *fixed* types | = | -2_147_483_648 * *small* |
| upper bound of *fixed* types | = | 2_147_483_647 * *small* |

A declared fixed point type is represented as that predefined *fixed* type which has the largest value of *small* not greater than the declared delta, and which has the smallest range that includes the declared range constraint.

Any fixed point type T has the following attributes:

| | | |
|---|---|---|
| T'MACHINE_OVERFLOWS | = | TRUE |
| T'MACHINE_ROUNDS | = | FALSE |

**Type DURATION**

The predefined fixed point type DURATION has the following attributes:

| | | |
|---|---|---|
| DURATION'AFT | = | 5 |
| DURATION'DELTA | = | DURATION'SMALL |
| DURATION'FIRST | = | -131_072.0 |
| DURATION'FORE | = | 7 |
| DURATION'LARGE | = | 1.31071999938965E05 |
| DURATION'LAST | = | 131_071.0 |
| DURATION'MANTISSA | = | 31 |
| DURATION'SAFE_LARGE | = | DURATION'LARGE |

| | | |
|---|---|---|
| DURATION'SAFE_SMALL | = | DURATION'SMALL |
| DURATION'SIZE | = | 32 |
| DURATION'SMALL | = | 2**(-14) = 6.10351562500000E-05 |

## F.2. Pragmas

This section lists all language-defined pragmas and any restrictions on their use and effect as compared to the definitions given in *Ada RM*.

### Pragma CONTROLLED

This pragma has no effect, as no automatic storage reclamation is performed before the point allowed by the pragma.

### Pragma ELABORATE

As in *Ada RM*.

### Pragma INLINE

This pragma causes inline expansion to be performed, except in the following cases:

1. The whole body of the subprogram for which inline expansion is wanted has not been seen. This ensures that recursive procedures cannot be inline expanded.

2. The subprogram call appears in an expression on which conformance checks may be applied, i.e., in a subprogram specification, in a discriminant part, or in a formal part of an entry declaration or accept statement.

3. The subprogram is an instantiation of the predefined generic subprograms UNCHECKED_CONVERSION or UNCHECKED_DEALLOCATION. Calls to such subprograms are expanded inline by the compiler automatically.

4. The subprogram is declared in a generic unit. The body of that generic unit is compiled as a secondary unit in the same compilation as a unit containing a call to (an instance of) the subprogram.

5. The subprogram is declared by a renaming declaration.

6. The subprogram is passed as a generic actual parameter.

A warning is given if inline expansion is not achieved.

### Pragma INTERFACE

This pragma is supported for the language names defined by the enumerated type INTERFACE_LANGUAGE in package SYSTEM.

*Language ASSEMBLY*

Ada programs may call assembly language subprograms that have been assembled with the VAX/VMS-hosted InterACT Mips Assembler. The compiler generates a call to the name of the subprogram (in all upper case). If a call to a different external name is desired, use pragma INTERFACE_SPELLING in conjunction with pragma INTERFACE (see Section F.3).

Parameters and results, if any, are passed in the same fashion as for a normal Ada call (see Appendix P).

Assembly subprogram bodies are not elaborated at runtime, and no runtime elaboration check is made when such subprograms are called.

Assembly subprogram bodies may in turn call Ada program units, but must obey all Ada calling and environmental conventions in doing so. Furthermore, Ada dependencies (in the form of context clauses) on the called program units must exist. That is, merely calling Ada program units from an assembly subprogram body will not make those program units visible to the Ada Linker.

A pragma INTERFACE (ASSEMBLY) subprogram may be used as a main program. In this case, the procedure specification for the main program must contain context clauses that will (transitively) name all Ada program units.

If an Ada subprogram declared with pragma INTERFACE (ASSEMBLY) is a library unit, the assembled subprogram body object code module must be put into the program library via the Ada Library Injection Tool (see Chapter 7). The Ada Linker will then automatically include the object code of the body in a link, as it would the object code of a normal Ada body.

If the Ada subprogram is not a library unit, the assembled subprogram body object code module cannot be put into the program library. In this case, the user must direct the Ada Linker to the directory containing the object code module (via the /user_rts qualifier, see Section 5.1), so that the InterACT Mips Embedded Systems Linker can find it.

*Other Languages*

It is possible to use pragma INTERFACE (ASSEMBLY) to call subprograms written in other languages (such as FORTRAN, Pascal, and C) supported by MIPS Computer Systems, Inc. compilers. This is because the object code format and the compiler protocols [*MIPS Appendix D*] used by the Compiler System are the same as those used in the MIPS-supplied compilers.

To do this, compile such subprograms on a MIPS computer system (making sure they are compiled for a big-endian configuration), and then transfer the object files (and any language runtime library object files needed by the subprograms) to VAX/VMS. (Make sure the transfer preserves the binary nature of the files.) Then proceed as with assembly language subprograms.

**Pragma LIST**

As in *Ada RM*.

**Pragma MEMORY_SIZE**

This pragma has no effect. See pragma SYSTEM_NAME.

**Pragma OPTIMIZE**

This pragma has no effect.

**Pragma PACK**

This pragma is accepted for array types whose component type is an integer or enumeration type that may be represented in 32 bits or less. The pragma has the effect that in allocating storage for an object of the array type, the object components are each packed into the next largest $2^n$ bits needed to contain a value of the component type. For example, integer components with the range constraint -8 .. 7 are packed into four bits; boolean components are packed into one bit.

The representation of packed array objects is such that the ordering of bits within words is defined to be big-endian. For example, in a packed array (0..31) of boolean, which occupies a word, component 0 is the most significant bit and component 31 is the least significant bit. Note that this convention differs from the one used in [*MIPS p. 2-6*] for bit-ordering. (The representation ordinarily does not matter, unless assembly language programming or other external interfaces are involved.)

This pragma is also accepted for record types but has no effect. Record representation clauses may be used to "pack" components of a record into any desired number of bits; see Section F.6.

**Pragma PAGE**

As in *Ada RM*.

**Pragma PRIORITY**

As in *Ada RM*. See the *Ada Mips Runtime Executive Programmer's Guide* for how a default priority may be set.

**Pragma SHARED**

This pragma has no effect, in terms of the compiler (and a warning message is issued). However, based on the current method of code generation, the effect of pragma SHARED is automatically achieved for all scalar and access objects.

**Pragma STORAGE_UNIT**

This pragma has no effect. See pragma SYSTEM_NAME.

**Pragma SUPPRESS**

Only the "identifier" argument, which identifies the type of check to be omitted, is allowed. The "[ON => ] name" argument, which isolates the check omission to a specific object, type, or subprogram, is not supported.

Pragma SUPPRESS with all checks other than DIVISION_CHECK results in the corresponding checking code not being generated. The implementation of arithmetic operations is such that, in general, pragma SUPPRESS with DIVISION_CHECK has no effect. In this case, runtime executive customizations may be used to mask the overflow interrupts that are used to implement these checks (see the *Ada Mips Runtime Executive Programmer's Guide* for details).

**Pragma SYSTEM_NAME**

This pragma has no effect. The only possible SYSTEM_NAME is Mips. The compilation of pragma MEMORY_SIZE, pragma STORAGE_UNIT, or this pragma does not cause an implicit recompilation of package SYSTEM.

## F.3. Implementation-dependent Pragmas

### F.3.1. Pragma EXPORT

This pragma is used to define an external name for Ada objects, so that they may be accessed from non-Ada routines. The pragma has the form

pragma EXPORT (*object*_name [,*external_name*_string_literal]);

The pragma must appear immediately after the associated object declaration. If the second argument is omitted, the object name in all upper case is used as the external name. Note that the Mips Assembler is case-sensitive; the second argument must be used if the external name is to be other than all upper case.

The associated object must be declared in a library package (or package nested within a library package), and must not be a statically-valued scalar constant (as such constants are not allocated in memory).

Identical external names should not be put out by multiple uses of the pragma (names can always be made unique by use of the second argument).

As an example of the use of this pragma, the objects in the following Ada library package

```
package GLOBAL is

   ABLE : FLOAT;
   pragma EXPORT (ABLE);

   Baker : STRING(1..8);
   pragma EXPORT (Baker, "Baker");

end GLOBAL;
```

may be accessed in the following assembly language fragment

```
        lw      $8,ABLE         # get value of ABLE

        la      $9,Baker        # get address of Baker
```

### F.3.2. Pragma IMPORT

This pragma is used to associate an Ada object with an object defined and allocated externally to the Ada program.

pragma IMPORT (*object*_name [,*external_name*_string_literal]);

The pragma must appear immediately after the associated object declaration. If the second argument is omitted, the object name in all upper case is used as the external name. Note that the Mips Assembler is case-sensitive; the second argument must be used if the external name is to be other than all upper case.

The associated object must be declared in a library package (or package nested within a library package). The associated object may not have an explicit or implicit initialization.

As an example of the use of this pragma, the objects in the following Ada library package

```
package GLOBAL is

   ABLE : FLOAT;
   pragma IMPORT (ABLE);

   Baker : STRING(1..8);
   pragma IMPORT (Baker, "Baker");

end GLOBAL;
```

are actually defined and allocated in the following assembly language fragment

```
        .globl   ABLE
        .lcomm   ABLE, 4

        .globl   Baker
        .lcomm   Baker, 8
```

### F.3.3. Pragma INTERFACE_SPELLING

This pragma is used to define the external name of a subprogram written in another language, if that external name is different from the subprogram name (if the names are the same, the pragma is not needed). Note that the Mips Assembler is case-sensitive; this pragma must be used if the external name is to be other than all upper case. The pragma has the form

**pragma** INTERFACE_SPELLING (*subprogram*_name, *external_name*_string_literal);

The pragma should appear after the pragma INTERFACE for the subprogram. This pragma is also useful in cases where the desired external name contains characters that are not valid in Ada identifiers.

Example:

```
procedure Connect_Bus (SIGNAL : INTEGER);
pragma INTERFACE (ASSEMBLY, Connect_Bus);
pragma INTERFACE_SPELLING (Connect_Bus, "Connect_Bus");
```

### F.3.4. Pragma SUBPROGRAM_SPELLING

This pragma is used to define the external name of an Ada subprogram. Normally such names are compiler-generated, based on the program library unit number. The pragma has the form

**pragma** SUBPROGRAM_SPELLING (*subprogram*_name [,*external_name*_string_literal]);

The pragma is allowed wherever a pragma INTERFACE would be allowed for the subprogram. If the second argument is omitted, the subprogram name in all upper case is used as the external name. Note that the Mips Assembler is case-sensitive; the second argument must be used if the external name is to be other than all upper case.

This pragma is useful in cases where the subprogram is to be referenced from another language.

## F.4. Implementation-dependent Attributes

None are defined.

## F.5. Package SYSTEM

The specification of package SYSTEM is:

```
package SYSTEM is

   type ADDRESS        is new INTEGER;
   ADDRESS_NULL        : constant ADDRESS := 0;

   type NAME           is (Mips);

   SYSTEM_NAME         : constant NAME := Mips;

   STORAGE_UNIT        : constant := 8;
   MEMORY_SIZE         : constant := 4 * 1024 * 1024 * 1024;

   MIN_INT             : constant := -2_147_483_647-1;
   MAX_INT             : constant := 2_147_483_647;
   MAX_DIGITS          : constant := 15;
   MAX_MANTISSA        : constant := 31;
   FINE_DELTA          : constant := 1.0 / 2.0 ** MAX_MANTISSA;
   TICK                : constant := 1.0;

   subtype PRIORITY    is INTEGER range 0..255;

   type INTERFACE_LANGUAGE is (ASSEMBLY);

   -- these are the possible ADDRESS values for interrupt entries
   MODx                : constant :=  1 * 4;      -- (MOD is reserved word)
   TLBL                : constant :=  2 * 4;
   TLBS                : constant :=  3 * 4;
   AdEL                : constant :=  4 * 4;
   AdES                : constant :=  5 * 4;
   IBE                 : constant :=  6 * 4;
   DBE                 : constant :=  7 * 4;
   Sys                 : constant :=  8 * 4;
   Bp                  : constant :=  9 * 4;
   RI                  : constant := 10 * 4;
   CpU                 : constant := 11 * 4;
   Ovf                 : constant := 12 * 4;
   Reserved13          : constant := 13 * 4;
   Reserved14          : constant := 14 * 4;
   Reserved15          : constant := 15 * 4;
   IP0                 : constant := 2**0 * 1024;
   IP1                 : constant := 2**1 * 1024;
   IP2                 : constant := 2**2 * 1024;
   IP3                 : constant := 2**3 * 1024;
   IP4                 : constant := 2**4 * 1024;
   IP5                 : constant := 2**5 * 1024;

end SYSTEM;
```

## F.6. Representation Clauses

In general, no representation clauses may be given for a derived type. The representation clauses that are accepted for non-derived types are described by the following:

### Length Clause

Three kinds of length clauses are accepted, specifying the number of storage units to be reserved for a collection (attribute designator STORAGE_SIZE), the number of storage units to be reserved for an activation of a task (STORAGE_SIZE), or the *small* for a fixed point type (SMALL). Length clauses specifying object size for a type (SIZE) are not allowed.

### Enumeration Representation Clause

Enumeration representation clauses are accepted.

### Record Representation Clause

Alignment clauses are allowed for values 2 and 4.

In terms of allowable component clauses, record components fall into three classes:

- integer and enumeration types;
- statically-bounded arrays or records composed solely of the above;
- all others.

Components of the "integer/enumeration" class may be given a component clause that specifies a storage place at any bit offset, and for any number of bits, as long as the storage place is large enough to contain the component and does not cross a word (32-bit) boundary. Unsigned representations (for example, an integer with a range of 0..3 being represented in two bits) are allowed, but the component subtype must belong to the predefined integer base type normally associated with that many bits (for example, an integer with a range of 0..2**32-1 being represented in 32 bits is not allowed). Biased representations (for example, an integer with a range of 7..10 being represented in two bits) are not allowed.

Components of the "array/record of integer/enumeration" class may be given a component clause that specifies a storage place at any bit offset, if the size of the array/record is less than a word, or at a word offset otherwise, and for any number of bits, as long as the storage place is large enough to contain the component and none of the individual integer/enumeration elements of the array/record cross a word boundary.

Components of the "all others" class may only be given component clauses that specify a storage place at a word offset, and for the number of bits normally allocated for objects of the underlying base type.

Components that do not have component clauses are allocated in storage places beginning at the next word boundary following the storage place of the last component in the record that has a component clause.

Records with component clauses cannot exceed 1K words (32K bits) in size.

The ordering of bits within storage units is defined to be big-endian. That is, bit 0 is the most significant bit and bit 31 is the least significant bit. Note that this convention differs from the one used in [*MIPS p. 2-6*] for bit-ordering.

### F.7. Implementation-dependent Names for Implementation-dependent Components

None are defined.

### F.8. Address Clauses

Address clauses are allowed for variables (objects that are not constants), and for interrupt entries. Address clauses are not allowed for constant objects, or for subprogram, package, or task units.

Address clauses occurring within generic units are always allowed at that point, but are not allowed when the units are instantiated if they do not conform to the implementation restrictions described here. (Note that the effect of such address clauses may depend on the context in which they are instantiated; for example, whether multiple address clauses specifying the same address are erroneous may depend on whether they are instantiated into library packages or subprograms.)

#### Address Clauses for Variables

Address clauses for variables must be static expressions of type ADDRESS in package SYSTEM.

It is the user's responsibility to reserve space at link time for the object. See the *Mips Embedded Systems Linker Reference Manual* for the means to do this.

Type ADDRESS is a 32-bit signed integer. Thus, addresses in the memory range 16#8000_0000#..16#FFFF_FFFF# (i.e., the upper half of target memory) must be supplied as negative numbers, since the positive (unsigned) interpretations of those addresses are greater than ADDRESS'LAST. Furthermore, addresses in this range must be declared as named numbers, with the named number (rather than a negative numeric literal) being used in the address clause. The hexadecimal address can be retained in the named number declaration, and user computation of the negative equivalent avoided, by use of the technique illustrated in the following example:

```
X : INTEGER;
for X use at 16#7FFF_FFFF#;   -- legal

Y : INTEGER;
for Y use at 16#FFFF_FFFF#;   -- illegal

ADDR_HIGH : constant := 16#FFFF_FFFF# - 2**32;
Y : INTEGER;
for Y use at ADDR_HIGH;  -- legal, equivalent to unsigned 16#FFFF_FFFF#
```

#### Address Clauses for Interrupt Entries

Address clauses for interrupt entries do not use target addresses but rather, the values in the target *Cause* register that correspond to particular interrupts. For convenience these values are defined as named numbers in package SYSTEM, corresponding to the mnemonics used in [*MIPS pp. 5-4, 5-5*].

The following restrictions apply to interrupt entries. The corresponding accept statement must have no formal parameters and must not be part of a select statement. Direct calls to the entry are not allowed. If any exception can be raised from within the accept statement, the accept statement must include an exception handler. The accept statement cannot include another accept statement for the same interrupt entry.

When the accept statement is encountered, the task is suspended. If the specified interrupt occurs, execution of the accept statement begins. When control reaches end of the accept statement, the special interrupt entry processing ends, and the task continues normal execution. Control must again return to the point where the accept statement is encountered in order for the task to be suspended again, awaiting the interrupt.

There are many more details of how interrupt entries interact with the target machine state and with the Runtime Executive. For these details, see the *Ada Mips Runtime Executive Programmer's Guide*.

### F.9. Unchecked Conversion

Unchecked conversion is only allowed between values of the same size. In addition, if UNCHECKED_CONVERSION is instantiated with an array type, that type must be statically constrained. Note also that calls to UNCHECKED_CONVERSION-instantiated functions are always generated as inline calls by the compiler, and cannot be instantiated as library units or used as generic actual parameters.

Unchecked conversion operates on the data for a value, and not on type descriptors or other compiler-generated entities (with the sole exception that records containing discriminant-dependent arrays have compiler-generated extra components representing array type descriptors).

For values of an access type, the data is the address of the designated object; thus, unchecked conversion may be done in either direction between access types and type SYSTEM.ADDRESS (which is derived from INTEGER). The named number SYSTEM.ADDRESS_NULL supplies the type ADDRESS equivalent of the access type literal **null**.

For values of a task type, the data is the address of the task's Task Control Block (see the *Ada Mips Runtime Executive Programmer's Guide*).

### F.10. Input-Output

The predefined library generic packages and packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO are supplied. However, file input-output is not supported except for the standard input and output files. Any attempt to create or open a file will result in USE_ERROR being raised.

TEXT_IO operations to the standard input and output files are implemented as input from or output to some visible device for a given Mips environment. Depending on the environment, this may be a console, simulator files, etc. See the *Ada Mips Runtime Executive Programmer's Guide* for more details. Note that by default, the standard input file is empty.

The range of the type COUNT defined in TEXT_IO is 0 .. INTEGER'LAST.

The predefined library package LOW_LEVEL_IO is empty.

In addition to the predefined library units, a package STRING_OUTPUT is also included in the predefined library. This package supplies a very small subset of TEXT_IO operations to the standard output file. The specification is:

```
package STRING_OUTPUT is

   procedure PUT (ITEM : in STRING);

   procedure PUT_LINE (ITEM : in STRING);

   procedure NEW_LINE;

end STRING_OUTPUT;
```

By using the 'IMAGE attribute function for integer and enumeration types, a fair amount of output can be done using this package instead of TEXT_IO. The advantage of this is that STRING_OUTPUT is smaller than TEXT_IO in terms of object code size, and faster in terms of execution speed.

Use of TEXT_IO in multiprogramming situations (see Chapter 5) may result in unexpected exceptions being raised, due to the shared unit semantics of multiprogramming. In such cases STRING_OUTPUT may be used instead.

## F.11. Other Chapter 13 Areas

The following language features, defined in [*Ada RM 13*], are supported by the compiler:

- representation attributes [*13.7.2, 13.7.3*]
- unchecked storage deallocation [*13.10.1*]

Note that calls to UNCHECKED_DEALLOCATION-instantiated procedures are always generated as inline calls by the compiler, and cannot be instantiated as library units or used as generic actual parameters.

Change of representation [*13.6*] and machine code insertions [*13.8*] are not supported by the compiler.

## F.12. Miscellaneous Implementation-dependent Characteristics

### Uninitialized Variables

There is no check to detect the use of uninitialized variables. The effect of a program that refers to the value of an uninitialized variable is undefined. A compiler cross-reference listing may be of use in finding such variables.

## F.13. Compiler System Capacity Limitations

The following capacity limitations apply to Ada programs in the Compiler System:

- the names of all identifiers, including compilation units, may not exceed the number of characters specified by the INPUT_LINELENGTH component in the compiler configuration file (see Section 4.1.4);
- a sublibrary can contain at most 4096 compilation units (library units or subunits). A program library can contain at most eight levels of sublibraries, but there is no limit to the number of sublibraries at each level. An Ada program can contain at most 32768 compilation units.

The above limitations are diagnosed by the compiler.

# APPENDIX C

## TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

| | |
|---|---|
| $ACC_SIZE<br>An integer literal whose value is the number of bits sufficient to hold any value of an access type. | 32 |
| $BIG_ID1<br>Identifier the size of the maximum input line length with varying last character. | <125*"A">1 |
| $BIG_ID2<br>Identifier the size of the maximum input line length with varying last character. | <125*"A">2 |
| $BIG_ID3<br>Identifier the size of the maximum input line length with varying middle character. | <62*"A">3<63*"A"> |
| $BIG_ID4<br>Identifier the size of the maximum input line length with varying middle character. | <62*"A">4<63*"A"> |
| $BIG_INT_LIT<br>An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length. | <123*"0">298 |
| $BIG_REAL_LIT | <120*"0">69.0E1 |

| | |
|---|---|
| A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length. | |
| $BIG_STRING1<br>A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1. | "<63*"A">" |
| $BIG_STRING2<br>A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1. | "<62*"A">1" |
| $BLANKS<br>A sequence of blanks twenty characters less than the size of the maximum line length. | 106 |
| $COUNT_LAST<br>A universal integer literal whose value is TEXT_IO.COUNT'LAST. | 2_147_483_647 |
| $DEFAULT_MEM_SIZE<br>An integer literal whose value is SYSTEM.MEMORY_SIZE. | 4*1024*1024*1024 |
| $DEFAULT_STOR_UNIT<br>An integer literal whose value is SYSTEM.STORAGE_UNIT. | 8 |
| $DEFAULT_SYS_NAME<br>The value of the constant SYSTEM.SYSTEM_NAME. | Mips |
| $DELTA_DOC<br>A real literal whose value is SYSTEM.FINE_DELTA. | 1.0/2.0**(SYSTEM.MAX_MANTISSA) |
| $FIELD_LAST<br>A universal integer literal whose value is TEXT_IO.FIELD'LAST. | 35 |
| $FIXED_NAME<br>The name of a predefined fixed-point type other than DURATION. | NO_SUCH_FIXED_TYPE |

| | |
|---|---|
| $FLOAT_NAME<br>The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT. | NO_SUCH_FLOAT_TYPE |
| $GREATER_THAN_DURATION<br>A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION. | 131_071.0 |
| $GREATER_THAN_DURATION_BASE_LAST<br>A universal real literal that is greater than DURATION'BASE'LAST. | 131_072.0 |
| $HIGH_PRIORITY<br>An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY. | 255 |
| $ILLEGAL_EXTERNAL_FILE_NAME1<br>An external file name which contains invalid characters. | ILLEGAL_FILE_NAME_1 |
| $ILLEGAL_EXTERNAL_FILE_NAME2<br>An external file name which is too long. | ILLEGAL_FILE_NAME_2 |
| $INTEGER_FIRST<br>A universal integer literal whose value is INTEGER'FIRST. | -2_147_483_648 |
| $INTEGER_LAST<br>A universal integer literal whose value is INTEGER'LAST. | 2_147_483_647 |
| $INTEGER_LAST_PLUS_1<br>A universal integer literal whose value is INTEGER'LAST + 1. | 2_147_483_648 |
| $LESS_THAN_DURATION<br>A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION. | -131_072.0 |
| $LESS_THAN_DURATION_BASE_FIRST<br>A universal real literal that is | -131_073.0 |

less than DURATION'BASE'FIRST.

$LOW_PRIORITY — 0
An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.

$MANTISSA_DOC — 31
An integer literal whose value is SYSTEM.MAX_MANTISSA.

$MAX_DIGITS — 15
Maximum digits supported for floating-point types.

$MAX_IN_LEN — 126
Maximum input line length permitted by the implementation.

$MAX_INT — 2_147_483_647
A universal integer literal whose value is SYSTEM.MAX_INT.

$MAX_INT_PLUS_1 — 2_147_483_648
A universal integer literal whose value is SYSTEM.MAX_INT+1.

$MAX_LEN_INT_BASED_LITERAL — 2:<121*"0">11:
A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.

$MAX_LEN_REAL_BASED_LITERAL — 16:<119*"0">F.E:
A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.

$MAX_STRING_LITERAL — "<124*"A">"
A string literal of size MAX_IN_LEN, including the quote characters.

$MIN_INT — -2_147_483_648
A universal integer literal whose value is SYSTEM.MIN_INT.

$MIN_TASK_SIZE — 32

An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.

| | |
|---|---|
| $NAME<br>A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER. | NO_SUCH_INTEGER_TYPE |
| $NAME_LIST<br>A list of enumeration literals in the type SYSTEM.NAME, separated by commas. | Mips |
| $NEG_BASED_INT<br>A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT. | 16#FFFFFFFE# |
| $NEW_MEM_SIZE<br>An integer literal whose value is a permitted argument for pragma memory_size, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE. | 4*1024*1024*1024 |
| $NEW_STOR_UNIT<br>An integer literal whose value is a permitted argument for pragma storage_unit, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT. | 8 |
| $NEW_SYS_NAME<br>A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value. | Mips |
| $TASK_SIZE | 32 |

An integer literal whose value is the number of bits required to hold a task object which has a single entry with one inout parameter.

$TICK 1.0

A real literal whose value is SYSTEM.TICK.

# APPENDIX D

## WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

A39005G
This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

B97102E
This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

C97116A
This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

BC3009B
This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

CD2A62D
This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]
These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD2A81G, CD2A83G, CD2A84M & N, & CD50110
These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

CD2B15C & CD7205C
These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

CD2D11B
This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

CD5007B
This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

ED7004B, ED7005C & D, ED7006C & D [5 tests]
These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

CD7105A
This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK -- particular instances of change may be less (line 29).

CD7203B, & CD7204B
These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD7205D
This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

CE2107I
This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

CE3111C
This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

CE3301A
This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

CE3411B
This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

E28005C
This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

# APPENDIX E

## COMPILER OPTIONS AS SUPPLIED BY

## InterACT Corporation

| | |
|---|---|
| Compiler: | InterACT Ada Mips Cross-Compiler System<br>Version 1.0 |
| ACVC Version: | 1.10 |

# Chapter 4
# The Ada Compiler

The Ada Compiler translates Ada source code into Mips R2000/R3000 object code.

Diagnostic messages are produced if any errors in the source code are detected. Warning messages are also produced when appropriate.

Compile, cross-reference, and generated assembly code listings are available upon user request.

The compiler uses a program library during the compilation. An internal representation of the compilation, which includes any dependencies on units already in the program library, is stored in the program library as a result of a successful compilation.

On a successful compilation, the compiler generates assembly code, invokes the Mips Assembler to translate this assembly code into object code, and then stores the object code in the program library. (Optionally, the generated assembly code may also be stored in the library.) The invocation of the Assembler is completely transparent to the user.

## 4.1. The Invocation Command

The Ada Compiler is invoked by submitting the following VAX/VMS command:

$ *adamips{qualifier}* *source-file-spec*

### 4.1.1. Parameters and Qualifiers

Default values exist for all qualifiers as indicated below. All qualifier names may be abbreviated (characters omitted from the right) as long as no ambiguity arises.

*source-file-spec*

This parameter specifies the file containing the source text to be compiled. Any valid VAX/VMS filename may be used. If the file type is omitted from the specification, file type ada is assumed by default. If this parameter is omitted, the user will be prompted for it. The format of the source text is described in Section 4.2.

**/list**
**/nolist** (default)

The user may request a source listing by means of the qualifier /list. The source listing is written to the list file. Section 4.3.2 contains a description of the source listing.

If /nolist is active, no source listing is produced, regardless of any LIST pragmas in the program or any diagnostic messages produced.

In addition, the /list qualifier provides generated assembly listings for each compilation unit in the source file. Section 4.3.6 contains a description of the generated assembly listing.

**/xref**
**/noxref** (default)

A cross-reference listing can be requested by the user by means of this qualifier. If /xref is active and no severe or fatal errors are found during the compilation, the cross-reference listing is written to the list file. The cross-reference listing is described in Section 4.3.4.

**/library=*file-spec***
**/library=*adamips*_library** (default)

This qualifier specifies the current sublibrary and thereby also specifies the current program library which consists of the current sublibrary through the root sublibrary (see Chapter 2). If the qualifier is omitted, the sublibrary designated by the logical name *adamips*_library is used as the current sublibrary.

Section 4.4 describes how the Ada compiler uses the current sublibrary.

**/configuration_file=*file-spec***
**/configuration_file=*adamips*_config** (default)

This qualifier specifies the configuration file to be used by the compiler in the current compilation.

If the qualifier is omitted, the configuration file designated by the logical name *adamips*_config is used by default. Section 4.1.4 contains a description of the configuration file.

**/keep_assembly**
**/nokeep_assembly** (default)

When this qualifier is given, the compiler will store the generated assembly source code in the program library, for each compilation unit being compiled. By default this is not done. Note that while the assembly code is stored in the library in a compressed form, it nevertheless takes up a large amount of library space relative to the other information stored in the library for a program unit.

This qualifier does not affect the production of generated assembly listings.

/nocheck

When this qualifier is given, all units in this compilation will be compiled as though a pragma SUPPRESS, for each kind of check, is present at the outermost declarative part of each unit. (See Section F.2 for a description of the effect of pragma SUPPRESS.) By default this is not done.

/debug
/nodebug (default)

When this qualifier is given, the compiler will generate symbolic debug information for each compilation unit in the source file and store the information in the program library. By default this is not done.

This symbolic debug information is used by the InterACT Symbolic Debugging and Simulation System.

It is important to note that the identical object code is produced by the compiler, whether or not the /debug qualifier is active.

/nooptimize

A small portion of the optimizing capability of the compiler places capacity limits on the source program (e.g., number of variables in a compilation unit) that are more restrictive than those documented in Section F.13. If a compile produces an error message indicating that one of these limits has been reached (e.g., "*** 1562S-0: Optimizer capacity exceeded. Too many names in a basic block"), use of this /nooptimize qualifier will bypass this particular optimizing capability and allow the compilation to finish normally.

*IMPORTANT NOTE:* Do not use this qualifier for any other reason. Do not attempt to use it in its positive form (/optimize), either with or without any of its keyword parameters. The /optimize qualifier as defined in the delivered command definition file is preset to produce the most effective optimization possible; any other use of it may produce either non-optimal or incorrect generated code.

/progress
/noprogress (default)

When this qualifier is given, the compiler will write a message to sys$output as each pass of the compiler starts to run. This information is not provided by default.

**Examples of qualifier usage**

**$ adamips navigation_constants**

**$ adamips/list/xref event_scheduler**

**$ adamips/prog/lib=test_versions.alb sys$user:[source]altitudes**