1.0 REVISION RECORD

A. 01/01/85 - Preliminary Draft.

2.0 PREFACE

This document provides the external Interface Specification for the CYBER Vectorizing Code Generator (CVCG).

3.0 INTRODUCTION

The CYBER Vectorizing Code Generator (CVCG) supports the development of compilers for the following source languages: ADA, BASIC, C, COBOL, CYBIL, FORTRAN, and PASCAL; producing object code for execution on (any model of) the following target machines: CYBER 180, CYBER 205, and CYBER 250. A source program in one of these languages is first processed by the appropriate compiler's "Front End", which is language dependent and machine independent. The Front End performs scanning, parsing, and semantic analysis. The internal representation of the program used in the Front End is then transformed into the internal representation used in the Code Generator by a "Bridge", which is both language dependent and machine dependent. The Bridge receives support from a set of procedures provided by the Code Generator, which are collectively termed the CVCG "Interface". Finally the Code Generator transforms the program into object code for a specific target machine. CVCG, which is language independent and machine dependent, performs automatic vectorization, optimization, and memory and register allocation. Hereafter, the term "Host" will be used to refer to the Front End and Bridge as a single unit, while the term "Back End" will be used to refer to the Bridge and Code Generator as a single unit.

3.1 ARCHITECTURAL DIAGRAM

The typical architecture of a compiler which uses the CYBER Vectorizing Code Generator can be illustrated as follows.

```
+-------------------+
!        Host       !        <-+ The Host is specific to !
!                   !          ! one of the languages:   !
!                   !          !  ADA, BASIC, C, COBOL,  !
!  +-------------+  !          ! CYBIL, FORTRAN, PASCAL. !
!  !  Front End  !  !
!  +-------------+  !
!          !        !
!          !        !
!  +-------!--------+--+
!  !       !        !  !
!  !       V        !  !
!  !  +----------+  !  !
!  !  !  Bridge  !  !  !
!  !  +----------+  !  !
!  !       !        !  !
!  !       !        !  !       ! The Bridge and CVCG     !
+--+-------!--------+  !     <-+ communicate through the !
   !       !           !       ! Interface procedures.   !
   !       V           !
   !  +-------------+  !
   !  !  CVCG Code  !  !
   !  !  Generator  !  !
   !  +-------------+  !       ! The Back End generates  !
   !                   !       ! code for the following  !
   !                   !       ! machines:   CYBER 180,  !
   !     Back End      !     <-+  CYBER 205, CYBER 250.  !
   +-------------------+
```

4.0 INTERFACE PROCEDURES

The CVCG Code Generator provides a set of interface procedures which are callable from the Host (normally from the Bridge). They must be used by the Host to pass all information needed by CVCG for the generation of correct code with the desired level of optimization and vectorization. Procedures are also present which allow the Host to query CVCG about the object code it generates.

4.1 INITIATION AND TERMINATION PROCEDURES

A single invocation of the code generator consists of an ordered series of calls to the code generator interface procedures. In the most general case this will consist of the following steps:

1. A call to cvp$i_begin_module;
2. Multiple calls to various definition (cvp$i_define_...) and emission (cvp$i_emit_...) procedures;
3. A call to cvp$i_begin_generation;
4. Multiple calls to various query (cvp$i_query_...) and transmission (cvp$i_transmit_...) procedures;
5. A call to cvp$i_end_generation;
6. Multiple calls to various query (cvp$i_query_...) procedures;
7. A call to cvp$i_end_module.

Steps 2 through 6 are all optional, however if step 3 is performed then step 5 must also be performed.

Multiple invocations of the code generator are allowed. Each invocation is independent of all other invocations; that is, the code generator is completely (re)initialized each time step 1 is performed.

4.1.1 CVP$I_BEGIN_GENERATION

```
?? PUSH (LISTEXT := ON) ??

 *copyc cvt$i_generation_status

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_begin_generation }
```

4.0 INTERFACE PROCEDURES
4.1.1 CVP$I_BEGIN_GENERATION

```
?? POP ??

  PROCEDURE [XREF] cvp$i_begin_generation (

        generate_errors_binary: boolean;
    VAR generation_status: cvt$i_generation_status);
```

PURPOSE:

This procedure informs the code generator that the Host has completed passing to the code generator all information needed in order to generate the object code. At this point the code generator will generate the requested object code and place it on the binary file.

ORDERING:

All definition (cvp$i_define_...) and emission (cvp$i_emit_...) procedure calls must precede the call to cvp$i_begin_generation. All query (cvp$i_query_...) and transmission (cvp$i_transmit_...) procedure calls must follow the call to cvp$i_begin_generation. There must be a subsequent call to cvp$i_end_generation prior to the call to cvp$i_end_module.

4.1.2 CVP$I_BEGIN_MODULE

```
?? PUSH (LISTEXT := ON) ??

 *copyc cvt$i_code_generator_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_begin_module }
?? POP ??

  PROCEDURE [XREF] cvp$i_begin_module (

        code_generator_attributes: cvt$i_code_generator_attributes);
```

PURPOSE:

This procedure initiates the code generator.

ORDERING:

This procedure must be called prior to any other procedure in the

Interface. It may not be called again until after cvp$i_end_module has been called.

4.1.3 CVP$I_END_GENERATION

```
?? PUSH (LISTEXT := ON) ??

 *copyc cvt$i_code_generator_results

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_end_generation }
?? POP ??

  PROCEDURE [XREF] cvp$i_end_generation (

    VAR code_generator_results: cvt$i_code_generator_results);
```

PURPOSE:

This procedure informs the code generator that the Host has completed passing to the code generator all information that is to be placed on the binary file. At this point the code generator will finish generation of the binary file.

ORDERING:

All transmission (cvp$i_transmit_...) procedure calls must precede the call to cvp$i_end_generation. Only query (cvp$i_query_...) procedure calls, and the call to cvp$i_end_module, may follow the call to cvp$i_end_generation. There must be one call to cvp$i_end_generation for each call to cvp$i_begin_generation.

4.1.4 CVP$I_END_MODULE

```
?? PUSH (LISTEXT := ON) ??

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_end_module }
?? POP ??

  PROCEDURE [XREF] cvp$i_end_module;
```

4.0 INTERFACE PROCEDURES
4.1.4 CVP$I_END_MODULE

PURPOSE:

This procedure terminates the code generator.

ORDERING:

No other procedure in the interface may be called after cvp$i_end_module, unless and until cvp$i_begin_module is called to reinitialize the code generator. There must be one call to cvp$i_end_module for each call to cvp$i_begin_module. If cvp$i_begin_generation has been called, then cvp$i_end_module may not be called until after the corresponding cvp$i_end_generation is called. Otherwise, cvp$i_end_module may be called at any time after cvp$i_begin_module.

4.2 CONSTANT_DEFINITION_PROCEDURES

Each constant referenced in one of the code emission procedure calls must have been previously defined by one of the constant definition procedure calls.

4.2.1 CVP$I_DEFINE_ARRAY_CONSTANT

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_array_constant
*copyc cvt$i_code_generator_id

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_array_constant }
?? POP ??

  PROCEDURE [XREF] cvo$i_define_array_constant (

        array_constant: cvt$i_array_constant;
    VAR constant_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a one-dimensional array constant.

4.2.2 CVP$I_DEFINE_POINTER_CONSTANT

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_pointer_constant }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_pointer_constant (

    VAR constant_id: cvt$i_code_generator_id);
```

4.0 INTERFACE PROCEDURES
4.2.2 CVP$I_DEFINE_POINTER_CONSTANT

PURPOSE:

This procedure defines a pointer constant. The Code Generator will provide a bit pattern for the constant that corresponds to the standard NIL pointer for the target_system.

4.2.3 CVP$I_DEFINE_SCALAR_CONSTANT

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_scalar_constant

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_scalar_constant }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_scalar_constant (

        scalar_constant: cvt$i_scalar_constant;
    VAR constant_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a scalar constant.

4.3 TYPE DEFINITION PROCEDURES

Each type referenced in one of the type definition or object definition procedure calls must have been previously defined by one of the type definition procedure calls. A collection of primitive types are provided by the Code Generator for use in describing the newly defined types:

```
---- ---- ---- ----
?? PUSH (LISTEXT := ON) ??

?? POP ??

?? PUSH (LIST := ON) ??
{ cvt$i_code_generator_type }
?? POP ??

  TYPE
    cvt$i_code_generator_type = (

    cvc$i_typeless,
      { This is used when the type of an object or operation is unknown
      {or not applicable.

    cvc$i_type_integer_32,
      {  This primitive type is used for operations upon objects which are
      {represented at the hardware implementation level with 32-bit signed
      {integers.  Note that CVCG requires the objects themselves to be
      {defined in terms of another type, usually in terms of 64-bit signed
      {integers having value bounds constraints.

    cvc$i_type_real_32,
      {  Objects of this type must have a length of 32 bits.
      {  This primitive type is used for objects which are represented at
      {the hardware implementation level with 32-bit floating point values;
      {and for operations upon such objects.  E.g. FORTRAN half precision.

    cvc$i_type_integer_64,
      {  Objects of this type may have a fixed length between 1 and 64 bits.
      {  This primitive type is used for objects which are represented at
      {the hardware implementation level with 64-bit signed integers; and
      {for operations upon such objects.  Note that this primitive type is
      {also used in the definition of integer objects having value bounds
      {constraints.  E.g. CYBIL integer, ordinal, subrange.

    cvc$i_type_real_64,
      {  Objects of this type must have a length of 64 bits.
      {  This primitive type is used for objects which are represented at
```

```
    {the hardware implementation level with 64-bit floating point va  es;
    {and for operations upon such objects.  E.g. FORTRAN real.

  cvc$i_type_complex_64,
    {  Objects of this type must have a length of 64 bits.
    {  This primitive type is used for objects which are represented at
    {the hardware implementation level with a pair of 32-bit floating
    {point values; and for operations upon such objects.

  cvc$i_type_real_128,
    {  Objects of this type must have a length of 128 bits.
    {  This primitive type is used for objects which are represented at
    {the hardware implementation level with 128-bit floating point values;
    {and for operations upon such objects.  E.g. FORTRAN double precision.

  cvc$i_type_complex_128,
    {  Objects of this type must have a length of 128 bits.
    {  This primitive type is used for objects which are represented at
    {the hardware implementation level with a pair of 64-bit floating point
    {values; and for operations upon such objects.  E.g. FORTRAN complex.

  cvc$i_type_complex_256,
    {  Objects of this type must have a length of 256 bits.
    {  This primitive type is used for objects which are represented at
    {the hardware implementation level with a pair of 128-bit floating
    {point values; and for operations upon such objects.

  cvc$i_type_boolean_sign,
    {  Objects of this type may have a fixed length between 1 and 64 bits.
    {  This primitive type is used for truth-valued objects which are
    {represented at the hardware implementation level with a signed integer
    {or integer subrange; and for operations upon such objects.  All
    {non-negative values are treated as FALSE, and all negative values are
    {treated as TRUE.  E.g. CYBER 180 FORTRAN logical.

  cvc$i_type_boolean_0_1,
    {  Objects of this type may have a fixed length between 1 and 64 bits.
    {  This primitive type is used for truth-valued objects which are
    {represented at the hardware implementation level with a signed integer
    {or integer subrange; and for operations upon such objects.  The value
    {of zero is treated as FALSE, the value of one is treated as TRUE, and
    {all other values have an undefined truth value.  E.g. CYBIL boolean;
    {CYBER 200 FORTRAN logical.

  cvc$i_type_bit_string,
    {  Objects of this type may have any fixed bit length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level with a sequence of bits; and
    {for operations upon such objects.  E.g. CYBIL set.
```

```
cvc$i_type_disjoint,
  {  Objects of this type may have any fixed bit length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level with a sequence of bits; in
  {addition the object must have no optimization interference with
  {any other object of any type.  E.g. many kinds of compiler-generated
  {objects.

cvc$i_type_union,
  {  Objects of this type may have any fixed bit length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level with a sequence of bits; in
  {addition the object has an optimization interference with objects
  {of more than one type.  E.g. FORTRAN boolean; CYBIL cell, sequence.

cvc$i_type_bdp_0_pdu,
  {  Objects of this type may have any fixed character length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level like CYBER 180 BDP type 0.

cvc$i_type_bdp_1_pdulsd,
  {  Objects of this type may have any fixed character length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level like CYBER 180 BDP type 1.

cvc$i_type_bdp_2_pds,
  {  Objects of this type may have any fixed character length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level like CYBER 180 BDP type 2.

cvc$i_type_bdp_3_pdslsd,
  {  Objects of this type may have any fixed character length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level like CYBER 180 BDP type 3.

cvc$i_type_bdp_4_udu,
  {  Objects of this type may have any fixed character length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level like CYBER 180 BDP type 4.

cvc$i_type_bdp_5_udtsch,
  {  Objects of this type may have any fixed character length.
  {  This primitive type is used for objects which are represented
  {at the hardware implementation level like CYBER 180 BDP type 5.

cvc$i_type_bdp_6_udtss,
  {  Objects of this type may have any fixed character length.
  {  This primitive type is used for objects which are represented
```

```
    {at the hardware implementation level like CYBER 180 BDP type 6.

  cvc$i_type_bdp_7_udlsch,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 7.

  cvc$i_type_bdp_8_udlss,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 8.

  cvc$i_type_bdp_9_a,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 9.
    {E.g. FORTRAN character; CYBIL string.

  cvc$i_type_bdp_10_bu,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 10.

  cvc$i_type_bdp_11_bs,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 11.

  cvc$i_type_bdp_12_tpds,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 12.

  cvc$i_type_bdp_13_tpdslsd,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 13.

  cvc$i_type_bdp_14_tbu,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 14.

  cvc$i_type_bdp_15_tbs,
    {  Objects of this type may have any fixed character length.
    {  This primitive type is used for objects which are represented
    {at the hardware implementation level like CYBER 180 BDP type 15.

  cvc$i_type_pointer,
```

4.0 INTERFACE PROCEDURES
4.3 TYPE DEFINITION PROCEDURES

```
      {  This primitive type is used for objects which are represented
      {at the hardware implementation level with 48-bit pointers.
      {E.g. CYBIL fixed pointers.

    cvc$i_type_array,
      {  This primitive type is used for objects which are represented
      {at the hardware implementation level as an array of elements.

    cvc$i_type_record,
      {  This primitive type is used for objects which are represented
      {at the hardware implementation level as a record structure.

    cvc$i_type_procedure,
      {  This primitive type is used for procedures when they are treated
      {as objects of pointers.

    cvc$i_type_internal
      {  This primitive type is used only internally to the Code Generator.

     );

  CONST
    cvc$i_type_char_string = cvc$i_type_bdp_9_a,
    cvc$i_type_integer = cvc$i_type_integer_64;
```

---- ---- ---- ----

4.3.1 CVP$I_DEFINE_ARRAY_TYPE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_array_attributes
*copyc cvt$i_array_descriptor
*copyc cvt$i_code_generator_id

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_array_type }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_array_type (

        array_attributes: cvt$i_array_attributes;
        array_descriptor: ^cvt$i_array_descriptor;
    VAR type_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new array type.

ORDERING:

The array element_type must be previously defined by a call to one of the following type definition procedures: cvp$i_define_array_type, cvp$i_define_integer_subtype, cvp$i_define_pointer_type, cvp$i_define_proc_pointer_type, cvp$i_define_range_type, cvp$i_define_record_type, or cvp$i_define_scalar_type.

4.3.2 CVP$I_DEFINE_INTEGER_SUBTYPE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_integer_subtype }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_integer_subtype (
```

```
        parent_type: cvt$i_code_generator_id;
        lower_bound: integer;
        upper_bound: integer;
    VAR subtype_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new subtype of a previously defined parent_type. Associated with this subtype are bounds constraints. It is the responsibility of the Host to ensure that objects of this subtype do not have values outside the specified bounds. The Code Generator will not perform or introduce bounds checking based on the specified bounds constraints.

ORDERING:

The parent_type must be previously defined by a call to cvp$i_define_range_type or by a call to cvp$i_define_scalar_type. In the former case, the bounds of the subtype must not lie outside the bounds of the range type. In the latter case, the scalar_type must be cvc$i_type_integer.

4.3.3 CVP$I_DEFINE_POINTER_OBJECT

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_pointer_object }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_pointer_object (

        pointer_id: cvt$i_code_generator_id;
        object_type: cvt$i_code_generator_id);
```

PURPOSE:

This procedure describes a pointer type in terms of the object to which it can point.

ORDERING:

The described pointer type must be previously defined by a call to cvp$i_define_pointer_type. There must be one call to cvp$i_define_pointer_object for each call to cvp$i_define_pointer_type; thus no two calls to cvp$i_define_pointer_object may specify the same pointer_id. The object_type must be previously defined by a call to one of the following type definition procedures: cvp$i_define_array_type, cvp$i_define_integer_subtype, cvp$i_define_pointer_type, cvp$i_define_proc_pointer_type, cvp$i_define_range_type, cvp$i_define_record_type, or cvp$i_define_scalar_type.

4.3.4 CVP$I_DEFINE_POINTER_TYPE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_pointer_type }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_pointer_type (

    VAR type_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines a new pointer type.

ORDERING:

The object which can be pointed to by a pointer of this type must be described in a subsequent call to cvp$i_define_pointer_object. The pointer object must be described before any references to the new pointer type occur in an object definition procedure.

4.3.5 CVP$I_DEFINE_PROC_POINTER_TYPE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id

?? POP ??
```

```
?? PUSH (LIST := ON) ??
{ cvp$i_define_proc_pointer_type }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_proc_pointer_type (

    VAR type_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines a new pointer-to-procedure type.

4.3.6 CVP$I_DEFINE_RANGE_TYPE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_range_type }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_range_type (

        lower_bound: integer;
        upper_bound: integer;
    VAR type_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new integer type with which bounds constraints are associated. It is the responsibility of the Host to ensure that objects of this type do not have values outside the specified bounds. The Code Generator will not perform or introduce bounds checking based on the specified bounds constraints.

4.3.7 CVP$I_DEFINE_RECORD_TYPE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_record_attributes
```

```
*copyc cvt$i_record_descriptor

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_record_type }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_record_type (

        record_attributes: cvt$i_record_attributes;
        record_descriptor: ^cvt$i_record_descriptor;
    VAR type_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new record type.

ORDERING:

Each field_type of the record must be previously defined by a call to one of the following type definition procedures: cvp$i_define_array_type, cvp$i_define_integer_subtype, cvp$i_define_pointer_type, cvp$i_define_proc_pointer_type, cvp$i_define_range_type, cvp$i_define_record_type, or cvp$i_define_scalar_type.

4.3.8 CVP$I_DEFINE_SCALAR_TYPE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_scalar_type

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_scalar_type }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_scalar_type (

        scalar_type: cvt$i_scalar_type;
    VAR type_id: cvt$i_code_generator_id);
```

PURPOSE:

4.0 INTERFACE PROCEDURES
4.3.8 CVP$I_DEFINE_SCALAR_TYPE

This procedure defines a new scalar type.

4.4 OBJECT_DEFINITION_PROCEDURES

Each object referenced in one of the object definition, code emission, or query procedure calls must have been previously defined by one of the object definition procedure calls.

4.4.1 CVP$I_DEFINE_DATA_AREA

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_data_area_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_data_area }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_data_area (

        data_area_attributes: cvt$i_data_area_attributes;
    VAR data_area_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new data area. A data area is a region in virtual memory where an unordered collection of data items is placed (e.g. a CYBIL section). The relative location within the data area of each data item is undefined; the Code Generator may alter the item ordering from that given by the Host.

4.4.2 CVP$I_DEFINE_DATA_ITEM

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_data_item_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_data_item }
?? POP ??
```

```
  PROCEDURE [XREF] cvp$i_define_data_item (

        data_item_attributes: cvt$i_data_item_attributes;
    VAR data_item_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new data item. A data item is a positionally independent region in virtual memory (e.g. a CYBIL variable). If a data item has an internal structure, the Code Generator will treat that structure as inviolable.

ORDERING:

Every data item must be associated with the enclosing_routine within which it was declared in the source language program; except that a data item which is declared at the module level with no enclosing_routine has an enclosing_routine of cvc$i_nil_id specified. Except for the case of ADA-style separate compilations, the enclosing_routine must be previously defined by a call to cvp$i_define_routine. Every data item must reside in a data area. Note that data items associated with different enclosing routines may be placed in the same data area. The enclosing_data_area must be previously defined by a call to cvp$i_define_data_area. The type of the data item must be previously defined by a call to one of the following type definition procedures: cvp$i_define_array_type, cvp$i_define_integer_subtype, cvp$i_define_pointer_type, cvp$i_define_proc_pointer_type, cvp$i_define_range_type, cvp$i_define_record_type, or cvp$i_define_scalar_type.

4.4.3 CVP$I_DEFINE_PARAM_AREA

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_param_area_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_param_area }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_param_area (

        param_area_attributes: cvt$i_param_area_attributes;
```

```
    VAR param_area_id: cvt$i_code_generator_id;
    VAR preceding_word_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new parameter area. A parameter area is a region in virtual memory where an ordered collection of parameter items is placed (forming a parameter list). The relative location within the parameter area of each parameter item is predefined; the Code Generator will preserve the item ordering given by the Host. Each distinct routine call or declaration in the program must have its own distinct parameter area, even for different calls to the same routine; except that there is no associated parameter area for the CVCG intrinsic routines.

4.4.4 CVP$I_DEFINE_PARAM_ITEM

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_param_item_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_param_item }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_param_item (

        param_item_attributes: cvt$i_param_item_attributes;
    VAR param_item_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new parameter item. A parameter item is a region in virtual memory where a parameter list entry is placed. Each parameter item must have a physical layout (param_item_format) conforming to one of the layouts described in Section 5.2.5 of the SIS.

ORDERING:

Every actual parameter item must have as its enclosing_routine the routine within which the associated actual routine call occurs in the source language program. Every formal parameter item must have as its enclosing_routine the routine with which it is associated. The enclosing_routine must be previously defined by a call to cvp$i_define_routine. Every parameter item must reside in a parameter area. The enclosing_param_area must be previously defined by a call to cvp$i_define_param_area. The type of each field of the parameter item must be previously defined by a call to one of the following type definition procedures: cvp$i_define_array_type, cvp$i_define_integer_subtype, cvp$i_define_pointer_type, cvp$i_define_proc_pointer_type, cvp$i_define_range_type, cvp$i_define_record_type, or cvp$i_define_scalar_type.

4.4.5 CVP$I_DEFINE_ROUTINE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_name
*copyc cvt$i_nesting_routine

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_routine }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_routine (

        nesting_routine: cvt$i_nesting_routine;
        routine_name: cvt$i_name;
    VAR routine_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines a new routine (aka. procedure, function, or entry point).

ORDERING:

The routine must be described in a subsequent call to cvp$i_define_routine_attributes. Every routine must be associated with the nesting_routine within which it is statically nested; except that a routine at the outermost nesting level has an nesting_routine of cvc$i_nil_id specified. Except for the case of ADA-style separate compilations, the nesting_routine must be previously defined by a call to cvp$i_define_routine.

4.4.6 CVP$I_DEFINE_ROUTINE_ATTRIBUTES

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_routine_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_routine_attributes }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_routine_attributes (

        routine_id: cvt$i_code_generator_id;
        routine_attributes: cvt$i_routine_attributes);
```

PURPOSE:

This procedure describes a routine.

ORDERING:

The described routine must be previously defined by a call to cvp$i_define_routine. There must be one call to cvp$i_define_routine_attributes for each call to cvp$i_define_routine; thus no two calls to cvp$i_define_routine_attributes may specify the same routine_id. The type of every routine which returns a value (has the function property) must be previously defined by a call to one of the following type definition procedures: cvp$i_define_array_type, cvp$i_define_integer_subtype, cvp$i_define_pointer_type, cvp$i_define_proc_pointer_type, cvp$i_define_range_type, cvp$i_define_record_type, or cvp$i_define_scalar_type. The routine_type of routines which do not have the function property is specified as cvc$i_nil_id.

4.5 POSITION DEFINITION PROCEDURES

Each position (i.e. label or line) referenced in one of the position definition or code emission procedure calls must have been previously defined by one of the position definition procedure calls.

4.5.1 CVP$I_DEFINE_LABEL

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_name

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_label }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_label (

        label_name: cvt$i_name;
    VAR label_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines a new label.

ORDERING:

The label must be described in a subsequent call to cvp$i_define_label_attributes.

4.5.2 CVP$I_DEFINE_LABEL_ATTRIBUTES

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_label_attributes

?? POP ??

?? PUSH (LIST := ON) ??
```

```
{ cvp$i_define_label_attributes }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_label_attributes (

        label_id: cvt$i_code_generator_id;
        label_attributes: cvt$i_label_attributes);
```

PURPOSE:

This procedure describes a label.

ORDERING:

The described label must be previously defined by a call to cvp$i_define_label. There must be one call to cvp$i_define_label_attributes for each call to cvp$i_define_label; thus no two calls to cvp$i_define_label_attributes may specify the same label_id. Every label must be associated with the line_number on which it was defined in the source language program. The line_number must be previously defined by a call to cvp$i_define_line.

4.5.3 CVP$I_DEFINE_LINE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_line_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_define_line }
?? POP ??

  PROCEDURE [XREF] cvp$i_define_line (

        line_attributes: cvt$i_line_attributes;
    VAR line_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure defines and describes a new source line.

4.6 CODE_EMISSION_PROCEDURES

The Host may pass a code sequence to CVCG using the code emission procedure calls. Each instruction that may be placed in the code sequence is referred to in terms of that instruction's opcode. Thus an "s_add" instruction is one having an opcode of "cvc$i_op_s_add" (scalar numeric add). Every instruction must be associated with the line_number in the source language program which led to that instruction's emission. The line_number must be previously defined by a call to cvp$i_define_line.

4.6.1 CVP$I_EMIT_DEREF_INSTR

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_code_generator_opcode
*copyc cvt$i_instruction_attributes
*copyc cvt$i_instruction_operand

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_emit_deref_instr }
?? POP ??

  PROCEDURE [XREF] cvp$i_emit_deref_instr (

        instruction_attributes: cvt$i_instruction_attributes;
        operand#1: cvt$i_instruction_operand;
    VAR instruction_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure emits a "deref" (pointer dereference) instruction.

ORDERING:

The instruction operand must be a data item previously defined by a call to cvp$i_define_data_item, or must be the result of a previously emitted instruction. The instruction_id can be used as an operand in subsequent instructions.

4.6.2 CVP$I_EMIT_END_OF_DEBUG_PACKET

```
?? PUSH (LISTEXT := ON) ??

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_emit_end_of_basic_block }
?? POP ??

  PROCEDURE [XREF] cvp$i_emit_end_of_debug_packet;
```

PURPOSE:

This procedure is needed only when stylized debug code is to be generated. It indicates that the end of a debug packet (normally, the end of a source language statement) has been reached.

ORDERING:

This procedure must be called at the end of each debug packet after all other code emission procedure calls within that packet, and prior to any code emission procedure calls for later debug packets.

4.6.3 CVP$I_EMIT_FIELD_REFERENCE

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_instruction_attributes
*copyc cvt$i_record_index_list

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_emit_field_reference }
?? POP ??

  PROCEDURE [XREF] cvp$i_emit_field_reference (

        instruction_attributes: cvt$i_instruction_attributes;
        record_data_item: cvt$i_code_generator_id;
        record_index_count: integer;
        record_index_list: ^cvt$i_record_index_list;
    VAR instruction_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure emits the instruction necessary to reference a field within a record or nested record.

ORDERING:

The record_data_item must be a data item previously defined by a call to cvp$i_define_data_item, or must be the result of a previously emitted instruction. The instruction_id can be used as an operand in subsequent instructions.

4.6.4 CVP$I_EMIT_INSTR_WITH_RESULT

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_instruction_attributes
*copyc cvt$i_instruction_operand
*copyc cvt$i_instr_with_result

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_emit_instr_with_result }
?? POP ??

  PROCEDURE [XREF] cvp$i_emit_instr_with_result (

        instruction_attributes: cvt$i_instruction_attributes;
        opcode: cvt$i_instr_with_result;
        operand#1: cvt$i_instruction_operand;
        operand#2: cvt$i_instruction_operand;
        operand#3: cvt$i_instruction_operand;
        operand#4: cvt$i_instruction_operand;
    VAR instruction_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure is used for the emission of most instructions for which the Code Generator returns a result identifier to the Host.

ORDERING:

The instruction operands must be previously defined by a call to a definition procedure, or must be the result of a previously emitted instruction. Unused operands are specified as cvc$i_nil_id. The result instruction_id can be used as an operand in subsequent instructions.

4.6.5 CVP$I_EMIT_INSTR_WITHOUT_RESULT

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_instruction_attributes
*copyc cvt$i_instruction_operand
*copyc cvt$i_instr_without_result

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_emit_instr_without_result }
?? POP ??

  PROCEDURE [XREF] cvp$i_emit_instr_without_result (

        instruction_attributes: cvt$i_instruction_attributes;
        opcode: cvt$i_instr_without_result;
        operand#1: cvt$i_instruction_operand;
        operand#2: cvt$i_instruction_operand;
        operand#3: cvt$i_instruction_operand;
        operand#4: cvt$i_instruction_operand;
        target: cvt$i_code_generator_id);
```

PURPOSE:

This procedure is used for the emission of those instructions for which the Code Generator does not return a result identifier to the Host.

ORDERING:

The instruction operands and target must be previously defined by a call to a definition procedure, or must be the result of a previously emitted instruction. Unused operands are specified as cvc$i_nil_id.

4.6.6 CVP$I_EMIT_LABEL_LIST

```
?? PUSH (LISTEXT := ON) ??
```

```
*copyc cvt$i_code_generator_id
*copyc cvt$i_code_generator_id_list
*copyc cvt$i_instruction_attributes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_emit_label_list }
?? POP ??

  PROCEDURE [XREF] cvp$i_emit_label_list (

        instruction_attributes: cvt$i_instruction_attributes;
        label_count: integer;
        label_list: ^cvt$i_code_generator_id_list;
    VAR list_id: cvt$i_code_generator_id);
```

PURPOSE:

This procedure emits a "p_l_list" (label list) instruction.

ORDERING:

Each label in the list must be previously defined by a call to cvp$i_define_label. The list_id can be used as an operand in subsequent instructions.

4.6.7 CVP$I_EMIT_OPERAND_LIST

```
?? PUSH (LISTEXT := ON) ??

*copyc cvt$i_code_generator_id
*copyc cvt$i_instruction_attributes
*copyc cvt$i_instruction_operand_list

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_emit_operand_list }
?? POP ??

  PROCEDURE [XREF] cvp$i_emit_operand_list (

        instruction_attributes: cvt$i_instruction_attributes;
        operand_count: integer;
        operand_list: ^cvt$i_instruction_operand_list;
    VAR list_id: cvt$i_code_generator_id);
```

4.0 INTERFACE PROCEDURES
4.6.7 CVP$I_EMIT_OPERAND_LIST

PURPOSE:

This procedure emits a "p_list" (operand list) instruction.

ORDERING:

Each operand in the list must be previously defined by a call to a definition procedure, or must be the result of a previously emitted instruction. The list_id can be used as an operand in subsequent instructions.

4.7 QUERY_PROCEDURES

The Host may obtain certain information from CVCG for use in creation of a reference map and/or of a (debug) symbol table. This information may only be queried subsequent to a call to cvp$i_begin_generation.

4.7.1 CVP$I_QUERY_LOCATION

```
?? PUSH (LISTEXT := ON) ??

 *copyc cvt$i_code_generator_id
 *copyc cvt$i_location

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_query_location }
?? POP ??

  PROCEDURE [XREF] cvp$i_query_location(

        object_id: cvt$i_code_generator_id;
    VAR location: cvt$i_location);
```

PURPOSE:

This procedure returns the location (if any) associated with a constant, object, label, or record field.

ORDERING:

A constant must be previously defined by a call to a constant definition procedure. An object must be previously defined by a call to an object definition procedure. A label must be previously defined by a call to cvp$i_define_label. A record field must be previously identified as the result of a call to cvp$i_emit_field_reference.

4.7.2 CVP$I_QUERY_ROUTINE_LENGTH

```
?? PUSH (LISTEXT := ON) ??

 *copyc cvt$i_code_generator_id
```

4.0 INTERFACE PROCEDURES
4.7.2 CVP$I_QUERY_ROUTINE_LENGTH

```
 *copyc cvt$i_size_in_bytes

?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_query_routine_length }
?? POP ??

  PROCEDURE [XREF] cvp$i_query_routine_length(

        routine_id: cvt$i_code_generator_id;
    VAR length: cvt$i_size_in_bytes);
```

PURPOSE:

This procedure returns the byte length associated with a routine which has a 'local', 'main', or 'xdcl' routine_scope. If this is an alternate entry point of a multiple entry point routine, then the length is that associated with the primary entry point.

ORDERING:

The routine must be previously defined by a call to cvp$i_define_routine.

4.8 TRANSMISSION_PROCEDURES

CVCG will place a (debug) line table in the binary file automatically unless cvc$i_no_debug_line_table is included in the generation_restrictions field of the code_generator_attributes passed to cvp$i_begin_module. With the transmission procedures the Host may direct CVCG to place additional information directly into the binary file. The Host is responsible for the contents and structure of this information; it will not be altered by CVCG except as described below for the (debug) symbol table.

4.8.1 CVP$I_TRANSMIT_LOADER_TABLE

```
?? PUSH (LISTEXT := ON) ??
 *copyc cvt$i_loader_table
 *copyc cvt$i_size_in_bytes
?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_transmit_loader_table }
?? POP ??

  PROCEDURE [XREF] cvp$i_transmit_loader_table (

        loader_table: ^cvt$i_loader_table;
        loader_table_length: cvt$i_size_in_bytes);
```

PURPOSE:

This procedure transmits a loader table directly to the binary file. It should be used for all Host generated loader tables except the (debug) symbol table.

4.8.2 CVP$I_TRANSMIT_SYMBOL_TABLE

```
?? PUSH (LISTEXT := ON) ??
 *copyc cvt$i_loader_table
 *copyc cvt$i_size_in_bytes
?? POP ??

?? PUSH (LIST := ON) ??
{ cvp$i_transmit_symbol_table }
?? POP ??
```

4.0 INTERFACE PROCEDURES
4.8.2 CVP$I_TRANSMIT_SYMBOL_TABLE

```
  PROCEDURE [XREF] cvp$i_transmit_symbol_table (

        loader_table: ^cvt$i_loader_table;
        loader_table_length: cvt$i_size_in_bytes);
```

PURPOSE:

This procedure transmits a (debug) symbol table directly to the binary file. The line table will be inserted into the symbol table when the target system is a CYBER 200. Otherwise the symbol table is transmitted unchanged.

ORDERING:

If cvp$i_transmit_symbol_table is called, then cvc$i_no_debug_symbol_table must not be included in the generation_restrictions field of the code_generator_attributes passed to cvp$i_begin_module. If cvp$i_transmit_symbol_table is not called, then cvc$i_no_debug_symbol_table must be included in the generation_restrictions field of the code_generator_attributes passed to cvp$i_begin_module.

A1.0 STACK_FRAME_LAYOUT

Each stack frame on the system stack consists of two sections: a fixed part and a variable part. The fixed part of the stack frame is allocated memory by CVCG at compile time. The Code Generator has full control of the memory layout of this fixed part. The variable part of the stack frame is allocated memory dynamically by the compiled program at execution time. The Code Generator has no control of the memory layout of this variable part. Note that the Code Generator may allocate memory in the variable part independently of Host directives to allocate memory in the variable part. Thus consecutive Host allocations of memory in the variable part are not guaranteed to allocate consecutive locations of memory.

A1.1 CYBER 180 STACK FRAME DIAGRAM

A stack frame on the CYBER 180 system, as generated by CVCG, can be illustrated as follows.

```
        !     (preceding frames)     !
        :             :              :
        !============================!      <===========+
   +0   !  Reserved for NOS/VE use   !                  !
        !----------------------------!                  !
   +1   !   Parameter List Pointer   !                  !
        !----------------------------!                  !
   +2   !   Reserved for Host use    !                  !
        !----------------------------!                  !
   +3   !  Reserved for future use   !                  !
        !----------------------------!         Fixed    !
   +4   !           Display          !          Part    !
        :             :              :                  !
        !----------------------------!                  !
   +j   !    Stack Variables Area    !                  !
        :             :              :                  !
        !----------------------------!                  !
   +k   !    Register Spill Area     !                  !
        :             :              :                  !
        !----------------------------!      <-----------+
   +m   !        Dynamic Space       !                  !
        :             :              :       Variable   !
        !----------------------------!         Part     !
   +n   !     Register Save Area     !                  !
        :             :              :                  !
        !============================!      <===========+
        :             :              :
        !    (succeeding frames)     !
```

Associated with each procedure are the following registers.

A0 The Dynamic Space Pointer contains the address of the next available word on the stack.
A1 The Current Stack Frame Pointer contains the address of the first word on the stack for the given procedure.
A2 The Previous Save Area Pointer contains the address on the stack of the calling procedure's Register Save Area.
A3 The Binding Section Pointer contains the address of the binding section.
A4 The Parameter List Pointer, upon entry to a procedure, contains the address of the first word of the parameter list. The CVCG stores this address in the Parameter List Pointer of

the called procedure's stack frame. Subsequent to storing A4, CVCG may reuse A4 for other purposes during execution of the rest of the procedure.

A1.1.1 CYBER 180 STACK FRAME

+0 The word at word-offset 0 from the beginning of each stack frame is reserved by CVCG for use by the NOS/VE Operating System.

+1 The word at word-offset 1 from the beginning of each stack frame is used by CVCG to contain the Parameter List Pointer, left justified.

+2 The word at word-offset 2 from the beginning of each stack frame is reserved by CVCG for use by the Host language.

+3 The word at word-offset 3 from the beginning of each stack frame is reserved by CVCG for future use. No code should define or reference it.

+4 The words starting at word-offset 4 from the beginning of each stack frame are used by CVCG to contain the static display, consisting of pointers which enable a nested procedure to access variables declared in its enclosing procedures. The size of the display depends on the nesting level. There is one word in the display for each enclosing procedure; the word contains the Current Stack Frame address of the enclosing procedure, left justified.

+j The words immediately following the Display in each stack frame contain space for automatic variables and workspaces having a fixed length at compile time. The size and layout of this space is determined by CVCG.

+k The words immediately following the Stack Variables Area of each stack frame contain workspace used by CVCG to hold the contents of hardware registers which must be spilled at execution time.

+m The Dynamic Space in each stack frame contains space for variables and workspaces having an unknown length at compile time.

+n The Register Save Area is created and used by the hardware CALL and RETURN instructions, for saving and restoring registers across a procedure call.

A1.2 CYBER 200 STACK FRAME DIAGRAM

A stack frame on the CYBER 200 system, as generated by CVCG, can be illustrated as follows.

```
        !     (preceding frames)     !
        :             :              :
        !============================!      <===========+
  +0    !     Register Save Area     !                  !
        :             :              :                  !
        !----------------------------!                  !
+256    !   Parameter List Pointer   !                  !
        !----------------------------!                  !
+257    !   Reserved for Host use    !                  !
        !----------------------------!                  !
+258    !  Reserved for future use   !                  !
        !----------------------------!        Fixed     !
+259    !          Display           !         Part     !
        :             :              :                  !
        !----------------------------!                  !
  +j    !    Stack Variables Area    !                  !
        :             :              :                  !
        !----------------------------!                  !
  +k    !    Register Spill Area     !                  !
        :             :              :                  !
        !----------------------------!      <-----------+
  +m    !       Dynamic Space        !       Variable   !
        :             :              :         Part     !
        !============================!      <===========+
        :             :              :
        !     (succeeding frames)    !
```

Associated with each procedure are the following registers.

- #1B The Dynamic Space Pointer (DSP) contains the address of the next available even-word on the stack.
- #1C The Current Stack Frame (CSF) contains the address of the first word on the stack for the given procedure.
- #1D The Previous Save Area (PSA) contains the address on the stack of the calling procedure's Register Save Area.
- #17 The Parameter List Pointer, upon entry to a procedure, contains the address of the first word of the parameter list. The CVCG stores this address in the Parameter List Pointer of the called procedure's stack frame. Subsequent to storing Register #17, CVCG may reuse #17 for other purposes during execution of the rest of the procedure.

A1.2.1 CYBER 200 STACK FRAME

+0 The Register Save Area is created and used by instructions generated by CVCG, for saving and restoring registers across a procedure call.

+256 The word at word-offset 256 from the beginning of each stack frame is used by CVCG to contain the Parameter List Pointer, right justified.

+257 The word at word-offset 257 from the beginning of each stack frame is reserved by CVCG for use by the Host language.

+258 The word at word-offset 258 from the beginning of each stack frame is reserved by CVCG for future use. No code should define or reference it.

+259 The words starting at word-offset 259 from the beginning of each stack frame are used by CVCG to contain the static display, consisting of pointers which enable a nested procedure to access variables declared in its enclosing procedures. The size of the display depends on the nesting level. There is one word in the display for each enclosing procedure; the word contains the Current Stack Frame address of the enclosing procedure, right justified.

+j The words immediately following the Display in each stack frame contain space for automatic variables and workspaces having a fixed length at compile time. The size and layout of this space is determined by CVCG.

+k The words immediately following the Stack Variables Area of each stack frame contain workspace used by CVCG to hold the contents of hardware registers which must be spilled at execution time.

+m The Dynamic Space in each stack frame contains space for variables and workspaces having an unknown length at compile time.

B1.0 INTRINSIC ROUTINE USAGE

CVCG supports a large number of intrinsic routines, i.e. routines (functions, subroutines, procedures, etc.) known to the Code Generator. Inline code will be generated for most of these routines. The rest of these will be generated as calls to library routines. The Host may request that the parameter list for a library call be placed in memory rather than in registers via the generation_restrictions field of the code_generator_attributes parameter of the cvp$i_begin_module call.

In order for the appropriate library routines to be present at execution time, the Host must include the math library (mlf$library on the CYBER 180) in the library_list field of the code_generator_attributes parameter of the cvp$i_begin_module call. If the Host references either of the code generator intrinsics cvc$i_sfunc_index or cvc$i_vfunc_index, then the Host must also include the Fortran library (flf$library on the CYBER 180) in the library_list.

B2.0 INTRINSIC ROUTINE NAMING CONVENTIONS

Most intrinsic names (such as ABS or DOTPRODUCT) are really generic names representing a whole family of separate, specific, routines. CVCG provides a different intrinsic_id for each specific intrinsic routine it supports. All routine identifiers start with "cvc$i_%_" where "%" is one of: "mcall", "mfunc", "rfunc", "scall", "sfunc", "tcall", "tfunc", "vcall", or "vfunc". Many routine identifiers also have a suffix to help identify which specific routine is of interest.

B2.1 IDENTIFIERS CVC$I_MCALL ...

These are used for miscellaneous routines which are implemented as subroutine calls. If they are source language functions then the function result appears as the first subroutine argument, and the n'th function argument appears as subroutine argument n+1. These routines are referenced in an instruction sequence by use of the "cvc$i_op_icall" code generator opcode; except for the "ranf" function with an array result, which uses the "cvc$i_op_v_ranf" opcode, and for the "ranget" and "ranset" subroutines, which use the "cvc$i_op_ranf" opcode.

B2.2 IDENTIFIERS CVC$I_MFUNC ...

These are used for miscellaneous functions having a non-character scalar result. They are referenced in an instruction sequence by use of the "cvc$i_op_ifunc" code generator opcode; except for the "ranf" function, which uses the "cvc$i_op_s_ranf" opcode.

B2.3 IDENTIFIERS CVC$I_RFUNC ...

These are used for array reduction functions which return a scalar result. These functions are all well-behaved (ie. have no side effects). They are referenced in an instruction sequence by use of the "cvc$i_op_v_ifunc_r" code generator opcode.

B2.4 IDENTIFIERS CVC$I_SCALL_...

These are used for scalar functions which return a character scalar result. Note that these functions are implemented as subroutine calls. The function result appears as the first subroutine argument, and the n'th function argument appears as subroutine argument n+1. For each 'scall' function there is a corresponding 'vcall' function. These functions are all well-behaved (ie. have no side effects). They are referenced in an instruction sequence by use of the "cvc$i_op_icall" code generator opcode.

B2.5 IDENTIFIERS CVC$I_SFUNC_...

These are used for scalar functions which return a non-character scalar result. For each 'sfunc' function there is a corresponding 'vfunc' function. These functions are all well-behaved (ie. have no side effects). They are referenced in an instruction sequence by use of the "cvc$i_op_s_ifunc" code generator opcode.

B2.6 IDENTIFIERS CVC$I_TCALL_...

These are used for those transformational functions which return a character array result, and for those which are array reduction functions over a dimension which is not known at compile-time. A transformational function is a function which in general can not be evaluated independently for each array element. Note that these functions are implemented as subroutine calls. The function result appears as the first subroutine argument, and the n'th function argument appears as subroutine argument n+1. These functions are all well-behaved (ie. have no side effects). They are referenced in an instruction sequence by use of the "cvc$i_op_icall" code generator opcode.

B2.7 IDENTIFIERS CVC$I_TFUNC_...

These are used for those transformational functions which return a non-character array result (including array reduction functions with an array result, provided that the reduction dimension is a compile-time constant). A transformational function is a function which in general can not be evaluated independently for each array element. These functions are all well-behaved (ie. have no side effects). They are referenced in an instruction sequence by use of the "cvc$i_op_ifunc" code generator opcode.

B2.8 IDENTIFIERS CVC$I_VCALL_...

These are used for non-scalar elemental functions which return a character array result. A non-scalar elemental function is a function with array arguments and an array result which can be evaluated independently for each array element. Note that these functions are implemented as subroutine calls. The function result appears as the first subroutine argument, and the n'th function argument appears as subroutine argument n+1. For each 'vcall' function there is a corresponding 'scall' function. These functions are all well-behaved (ie. have no side effects). They are referenced in an instruction sequence by use of the "cvc$i_op_icall" code generator opcode.

B2.9 IDENTIFIERS CVC$I_VFUNC_...

These are used for non-scalar elemental functions which return a non-character array result. A non-scalar elemental function is a function with array arguments and an array result which can be evaluated independently for each array element. For each 'vfunc' function there is a corresponding 'sfunc' function. These functions are all well-behaved (ie. have no side effects). They are referenced in an instruction sequence by use of the "cvc$i_op_v_ifunc" code generator opcode.

B2.10 IDENTIFIER SUFFIXES

bsign: This indicates that some of the operands and/or result have a value corresoonding to the code generator type of cvt$i_type_boolean_sign.

b01: This indicates that some of the operands and/or result have a value corresoonding to the code generator type of cvt$i_type_boolean_0_1.

char: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_char_string.

c64: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_complex_64.

c128: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_complex_128.

c256: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_complex_256.

i32: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_integer_32.

i64: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_integer_64.

r32: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_real_32.

r64: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_real_64.

r128: This indicates that some of the operands and/or result have a value corresponding to the code generator type of cvt$i_type_real_128.

1bit: This indicates that the operands and/or result have a length of 1 bit.

8bit: This indicates that the operands and/or result have a length of 8 bits.

16bit: This indicates that the operands and/or result have a length of 16 bits.

32bit: This indicates that the operands and/or result have a length of 32 bits.

64bit: This indicates that the operands and/or result have a length of 64 bits.

128bit: This indicates that the operands and/or result have a length of 128 bits.

collated: This indicates that the operation uses a character collation table.

B3.0 INTRINSIC ROUTINE DEFINITIONS

Most routines correspond to FORTRAN expression operators or to FORTRAN intrinsic routines (the specific version if there is both a specific and a generic version with the same name) as defined in the CDC Standard FORTRAN Language Specification. A few routines correspond to ADA operators as defined in the Military Standard for the ADA Programming Language; or to BASIC routines as defined in the Virtual BASIC External Reference Specification; or to CYBIL intrinsic routines as defined in the CYBIL Language Specification; or to PASCAL predefined routines as defined in the PASCAL User Manual and Report by Jensen and Wirth. Other miscellaneous routines are provided as needed.

The order of arguments is that defined for the positional (rather than keyword) form of the routine. For binary operators, the leftmost operand corresponds to the first argument.

The number of arguments (excluding function results which are implemented as subroutine arguments) is that defined in the appropriate language specification, with the following exceptions:

- routines listed below with a specified argument count.
- routines where an exception is specifically noted below.
- routines having the 'collated' identifier suffix; these routines have one additional argument (positionally the last) which is a 256-byte collation table.

Broadcast scalar arguments may be substituted for array arguments in the following cases:

- any one, but not both, arguments of a non-scalar elemental function having two arguments (excluding collation table).
- any one or two, but not all three, arguments of a non-scalar elemental function having three arguments (excluding collation table).
- any one, two, or three, but not all four, arguments of a non-scalar elemental function having four arguments (excluding collation table).
- the MASK argument of the following array reduction functions: maxval, minval, product, sum. If this argument is a broadcast scalar, it must also be a compile time constant.
- the first argument of the transformational function: diagonal.
- the third argument of the transformational function: unpack.

B4.0 CVT$I_INTRINSIC_ID

```
?? PUSH (LISTEXT := ON) ??

?? POP ??

?? PUSH (LIST := ON) ??
{ cvt$i_intrinsic_id }
?? POP ??

?? FMT (FORMAT := OFF) ??

{  The following list includes all those intrinsic routines which are
{ expected to be supported by the code generator in its first few
{ releases, in support of the ADA, BASIC, C, COBOL, CYBIL, FORTRAN, and
{ PASCAL languages.  Additional routines will be included for support if
{ and when their need is identified.  Not all of the listed routines will
{ be supported by the first release of the code generator.  Each routine
{ name in the list is followed by one of the following characters,
{ indicating in which code generator release it is expected to first be
{ supported.
{
{   1 - To be supported in the first code generator release (for CDC FORTRAN
{         on the CYBER 180, with a restricted set of array intrinsics).
{   2 - To be supported in the second code generator release (for CDC FORTRAN
{         on both the CYBER 180 and CYBER 200).
{   3 - To be supported in an early code generator release (for CDC FORTRAN
{         on both machine lines, with a full set of array intrinsics).
{   A - To be supported for the initial ADA and CYBIL releases.
{   B - To be supported for the initial BASIC release.
{   C - To be supported for the initial CYBIL release (see also 'A').
{   F - To be supported for a future release of CDC FORTRAN with extensions
{         for new source data types including: c64, c256, and i32.
{   P - To be supported for the initial PASCAL release.

TYPE
  cvt$i_intrinsic_id = (

 cvc$i_non_intrinsic,             {1: -This is used for non-intrinsic routines-
 cvc$i_mcall_current_stack_frame,{A: See CYBIL:     #current_stack_frame
 cvc$i_mcall_previous_save_area, {A: See CYBIL:     #previous_save_area
 cvc$i_mcall_ranf,                {1: See FORTRAN:   ranf      -See Note 2-
 cvc$i_mcall_ranget,              {B: See FORTRAN:   ranget    -See Note 3-
 cvc$i_mcall_ranset,              {B: See FORTRAN:   ranset    -See Note 3-
 cvc$i_mcall_scan_b01,            {C: See CYBIL:     #scan     -See Note 5-
 cvc$i_mcall_scan_bsign,          {C: See CYBIL:     #scan     -See Note 5-
 cvc$i_mfunc_ranf,                {1: See FORTRAN:   ranf      -See Note 2-
```

84.0 CVT$I_INTRINSIC_ID

```
cvc$i_rfunc_all_b01,            {2: See FORTRAN:  all      (one arg,     DIM)
cvc$i_rfunc_all_bsign,          {1: See FORTRAN:  all      (one arg,  no DIM)
cvc$i_rfunc_any_b01,            {2: See FORTRAN:  any      (one arg,  no DIM)
cvc$i_rfunc_any_bsign,          {1: See FORTRAN:  any      (one arg,  no DIM)
cvc$i_rfunc_count_1bit,         {P: See PASCAL:   card
cvc$i_rfunc_count_b01,          {2: See FORTRAN:  count    (one arg,  no DIM)
cvc$i_rfunc_count_bsign,        {1: See FORTRAN:  count    (one arg,  no DIM)
cvc$i_rfunc_dotproduct_c128,    {1: See FORTRAN:  dotproduct
cvc$i_rfunc_dotproduct_c256,    {F: See FORTRAN:  dotproduct
cvc$i_rfunc_dotproduct_c64,     {F: See FORTRAN:  dotproduct
cvc$i_rfunc_dotproduct_i32,     {F: See FORTRAN:  dotproduct
cvc$i_rfunc_dotproduct_i64,     {1: See FORTRAN:  dotproduct
cvc$i_rfunc_dotproduct_r128,    {1: See FORTRAN:  dotproduct
cvc$i_rfunc_dotproduct_r32,     {2: See FORTRAN:  dotproduct
cvc$i_rfunc_dotproduct_r64,     {1: See FORTRAN:  dotproduct
cvc$i_rfunc_maxval_i32_b01,     {F: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_i32_bsign,   {F: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_i64_b01,     {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_i64_bsign,   {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_r128_b01,    {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_r128_bsign,  {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_r32_b01,     {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_r32_bsign,   {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_r64_b01,     {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_maxval_r64_bsign,   {3: See FORTRAN:  maxval   (two args, no DIM)
cvc$i_rfunc_minval_i32_b01,     {F: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_i32_bsign,   {F: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_i64_b01,     {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_i64_bsign,   {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_r128_b01,    {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_r128_bsign,  {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_r32_b01,     {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_r32_bsign,   {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_r64_b01,     {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_minval_r64_bsign,   {3: See FORTRAN:  minval   (two args, no DIM)
cvc$i_rfunc_product_c128_b01,   {3: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_c128_bsign, {3: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_c256_b01,   {F: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_c256_bsign, {F: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_c64_b01,    {F: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_c64_bsign,  {F: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_i32_b01,    {F: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_i32_bsign,  {F: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_i64_b01,    {3: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_i64_bsign,  {3: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_r128_b01,   {3: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_r128_bsign, {3: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_r32_b01,    {3: See FORTRAN:  product  (two args, no DIM)
cvc$i_rfunc_product_r32_bsign,  {3: See FORTRAN:  product  (two args, no DIM)
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_rfunc_product_r64_b01,     {3: See FORTRAN:  product (two args,    DIM)
cvc$i_rfunc_product_r64_bsign,   {3: See FORTRAN:  product (two args, no DIM)
cvc$i_rfunc_sum_c128_b01,        {2: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_c128_bsign,      {1: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_c256_b01,        {F: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_c256_bsign,      {F: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_c64_b01,         {F: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_c64_bsign,       {F: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_i32_b01,         {F: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_i32_bsign,       {F: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_i64_b01,         {2: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_i64_bsign,       {1: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_r128_b01,        {2: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_r128_bsign,      {1: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_r32_b01,         {2: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_r32_bsign,       {2: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_r64_b01,         {2: See FORTRAN:  sum     (two args, no DIM)
cvc$i_rfunc_sum_r64_bsign,       {1: See FORTRAN:  sum     (two args, no DIM)
cvc$i_scall_char,                {1: See FORTRAN:  char    (FIXED collation)
cvc$i_scall_char_collated,       {1: See FORTRAN:  char    (USER collation)
cvc$i_scall_merge_char_b01,      {2: See FORTRAN:  merge
cvc$i_scall_merge_char_bsign,    {1: See FORTRAN:  merge
cvc$i_sfunc_abs,                 {1: See FORTRAN:  abs
cvc$i_sfunc_acos,                {1: See FORTRAN:  acos
cvc$i_sfunc_aimag,               {1: See FORTRAN:  aimag
cvc$i_sfunc_aint,                {1: See FORTRAN:  aint
cvc$i_sfunc_alog,                {1: See FORTRAN:  alog
cvc$i_sfunc_alog10,              {1: See FORTRAN:  alog10
cvc$i_sfunc_amax0,               {1: See FORTRAN:  amax0   (two args)
cvc$i_sfunc_amax1,               {1: See FORTRAN:  amax1   (two args)
cvc$i_sfunc_amin0,               {1: See FORTRAN:  amin0   (two args)
cvc$i_sfunc_amin1,               {1: See FORTRAN:  amin1   (two args)
cvc$i_sfunc_amod,                {1: See FORTRAN:  amod
cvc$i_sfunc_and_1bit,            {1: See FORTRAN:  and     (two args, bit)
cvc$i_sfunc_and_64bit,           {1: See FORTRAN:  and     (two args, boolean)
cvc$i_sfunc_anint,               {1: See FORTRAN:  anint
cvc$i_sfunc_asin,                {1: See FORTRAN:  asin
cvc$i_sfunc_atan,                {1: See FORTRAN:  atan
cvc$i_sfunc_atan2,               {1: See FORTRAN:  atan2
cvc$i_sfunc_atanh,               {1: See FORTRAN:  atanh
cvc$i_sfunc_bool_of_char,        {1: See FORTRAN:  bool    (character arg only)
cvc$i_sfunc_btol_b01,            {2: See FORTRAN:  btol
cvc$i_sfunc_btol_bsign,          {1: See FORTRAN:  btol
cvc$i_sfunc_c128_to_c128_power,  {1: See FORTRAN:  '**' operator
cvc$i_sfunc_c128_to_c256_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c128_to_c64_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c128_to_i32_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c128_to_i64_power,   {1: See FORTRAN:  '**' operator
cvc$i_sfunc_c128_to_r128_power,  {1: See FORTRAN:  '**' operator
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_sfunc_c128_to_r32_power,  {2: See FORTRAN:  '**' operator
cvc$i_sfunc_c128_to_r64_power,  {1: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_c128_power, {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_c256_power, {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_c64_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_i32_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_i64_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_r128_power, {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_r32_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c256_to_r64_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_c128_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_c256_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_c64_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_i32_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_i64_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_r128_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_r32_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_c64_to_r64_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_cabs,               {1: See FORTRAN:  cabs
cvc$i_sfunc_ccos,               {1: See FORTRAN:  ccos
cvc$i_sfunc_cdabs,              {F: See FORTRAN:  cdabs   (abs for c256)
cvc$i_sfunc_cdcos,              {F: See FORTRAN:  cdcos   (cos for c256)
cvc$i_sfunc_cdexp,              {F: See FORTRAN:  cdexp   (exp for c256)
cvc$i_sfunc_cdlog,              {F: See FORTRAN:  cdlog   (log for c256)
cvc$i_sfunc_cdsin,              {F: See FORTRAN:  cdsin   (sin for c256)
cvc$i_sfunc_cdsqrt,             {F: See FORTRAN:  cdsqrt  (sqrt for c256)
cvc$i_sfunc_ceil,               {8: See BASIC:    ceil    (r64 arg and result)
cvc$i_sfunc_cexp,               {1: See FORTRAN:  cexp
cvc$i_sfunc_chabs,              {F: See FORTRAN:  chabs   (abs for c64)
cvc$i_sfunc_chcos,              {F: See FORTRAN:  chcos   (cos for c64)
cvc$i_sfunc_chexp,              {F: See FORTRAN:  chexp   (exp for c64)
cvc$i_sfunc_chlog,              {F: See FORTRAN:  chlog   (log for c64)
cvc$i_sfunc_chsin,              {F: See FORTRAN:  chsin   (sin for c64)
cvc$i_sfunc_chsqrt,             {F: See FORTRAN:  chsqrt  (sqrt for c64)
cvc$i_sfunc_clog,               {1: See FORTRAN:  clog
cvc$i_sfunc_cmplx,              {1: See FORTRAN:  cmplx   (two args, real)
cvc$i_sfunc_conjg,              {1: See FORTRAN:  conjg
cvc$i_sfunc_cos,                {1: See FORTRAN:  cos
cvc$i_sfunc_cosd,               {1: See FORTRAN:  cosd
cvc$i_sfunc_cosh,               {1: See FORTRAN:  cosh
cvc$i_sfunc_cotan,              {1: See FORTRAN:  cotan
cvc$i_sfunc_csin,               {1: See FORTRAN:  csin
cvc$i_sfunc_csqrt,              {1: See FORTRAN:  csqrt
cvc$i_sfunc_dabs,               {1: See FORTRAN:  dabs
cvc$i_sfunc_dacos,              {1: See FORTRAN:  dacos
cvc$i_sfunc_dasin,              {1: See FORTRAN:  dasin
cvc$i_sfunc_datan,              {1: See FORTRAN:  datan
cvc$i_sfunc_datan2,             {1: See FORTRAN:  datan2
cvc$i_sfunc_dconjg,             {F: See FORTRAN:  dconjg  (conjg for c256)
```

B4.0 CVT$I_INTRINSIC_ID

```
 cvc$i_sfunc_dcos,              {1: See FORTRAN:  dcos
 cvc$i_sfunc_dcosh,             {1: See FORTRAN:  dcosh
 cvc$i_sfunc_dcotan,            {F: See FORTRAN:  dcotan
 cvc$i_sfunc_ddim,              {1: See FORTRAN:  ddim
 cvc$i_sfunc_dexp,              {1: See FORTRAN:  dexp
 cvc$i_sfunc_dim,               {1: See FORTRAN:  dim
 cvc$i_sfunc_dimag,             {F: See FORTRAN:  dimag   (aimag for c256)
 cvc$i_sfunc_dint,              {1: See FORTRAN:  dint
 cvc$i_sfunc_dlog,              {1: See FORTRAN:  dlog
 cvc$i_sfunc_dlog10,            {1: See FORTRAN:  dlog10
 cvc$i_sfunc_dmax1,             {1: See FORTRAN:  dmax1   (two args)
 cvc$i_sfunc_dmin1,             {1: See FORTRAN:  dmin1   (two args)
 cvc$i_sfunc_dmod,              {1: See FORTRAN:  dmod
 cvc$i_sfunc_dnint,             {1: See FORTRAN:  dnint
 cvc$i_sfunc_dprod,             {1: See FORTRAN:  dprod
 cvc$i_sfunc_dsign,             {1: See FORTRAN:  dsign
 cvc$i_sfunc_dsin,              {1: See FORTRAN:  dsin
 cvc$i_sfunc_dsinh,             {1: See FORTRAN:  dsinh
 cvc$i_sfunc_dsqrt,             {1: See FORTRAN:  dsqrt
 cvc$i_sfunc_dtan,              {1: See FORTRAN:  dtan
 cvc$i_sfunc_dtanh,             {1: See FORTRAN:  dtanh
 cvc$i_sfunc_eqv_1bit,          {1: See FORTRAN:  eqv     (two args, bit)
 cvc$i_sfunc_eqv_64bit,         {1: See FORTRAN:  eqv     (two args, boolean)
 cvc$i_sfunc_erf,               {1: See FORTRAN:  erf
 cvc$i_sfunc_erfc,              {1: See FORTRAN:  erfc
 cvc$i_sfunc_exp,               {1: See FORTRAN:  exp
 cvc$i_sfunc_extb,              {1: See FORTRAN:  extb    (boolean first arg)
 cvc$i_sfunc_floor,             {B: See BASIC:    int     (r64 arg and result)
 cvc$i_sfunc_fract_part,        {B: See BASIC:    fp      (r64 arg and result)
 cvc$i_sfunc_habs,              {2: See FORTRAN:  habs
 cvc$i_sfunc_hacos,             {2: See FORTRAN:  hacos
 cvc$i_sfunc_hasin,             {2: See FORTRAN:  hasin
 cvc$i_sfunc_hatan,             {2: See FORTRAN:  hatan
 cvc$i_sfunc_hatan2,            {2: See FORTRAN:  hatan2
 cvc$i_sfunc_hconjg,            {F: See FORTRAN:  hconjg  (conjg for c64)
 cvc$i_sfunc_hcos,              {2: See FORTRAN:  hcos
 cvc$i_sfunc_hcosh,             {2: See FORTRAN:  hcosh
 cvc$i_sfunc_hcotan,            {2: See FORTRAN:  hcotan
 cvc$i_sfunc_hdim,              {2: See FORTRAN:  hdim
 cvc$i_sfunc_hexp,              {2: See FORTRAN:  hexp
 cvc$i_sfunc_himag,             {F: See FORTRAN:  himag   (aimag for c64)
 cvc$i_sfunc_hint,              {2: See FORTRAN:  hint
 cvc$i_sfunc_hlog,              {2: See FORTRAN:  hlog
 cvc$i_sfunc_hlog10,            {2: See FORTRAN:  hlog10
 cvc$i_sfunc_hmax1,             {2: See FORTRAN:  hmax1   (two args)
 cvc$i_sfunc_hmin1,             {2: See FORTRAN:  hmin1   (two args)
 cvc$i_sfunc_hmod,              {2: See FORTRAN:  hmod
 cvc$i_sfunc_hnint,             {2: See FORTRAN:  hnint
 cvc$i_sfunc_hsign,             {2: See FORTRAN:  hsign
```

```
 cvc$i_sfunc_hsin,              {2: See FORTRAN:  hsin
 cvc$i_sfunc_hsinh,             {2: See FORTRAN:  hsinh
 cvc$i_sfunc_hsqrt,             {2: See FORTRAN:  hsqrt
 cvc$i_sfunc_htan,              {2: See FORTRAN:  htan
 cvc$i_sfunc_htanh,             {2: See FORTRAN:  htanh
 cvc$i_sfunc_i32_to_c128_power, {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i32_to_c256_power, {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i32_to_c64_power,  {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i32_to_i32_power,  {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i32_to_i64_power,  {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i32_to_r128_power, {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i32_to_r32_power,  {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i32_to_r64_power,  {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_c128_power, {1: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_c256_power, {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_c64_power,  {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_i32_power,  {F: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_i64_power,  {1: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_r128_power, {1: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_r32_power,  {2: See FORTRAN:  '**' operator
 cvc$i_sfunc_i64_to_r64_power,  {1: See FORTRAN:  '**' operator
 cvc$i_sfunc_iabs,              {1: See FORTRAN:  iabs
 cvc$i_sfunc_ichar,             {1: See FORTRAN:  ichar   (FIXED collation)
 cvc$i_sfunc_ichar_collated,    {1: See FORTRAN:  ichar   (USER collation)
 cvc$i_sfunc_idim,              {1: See FORTRAN:  idim
 cvc$i_sfunc_idnint,            {1: See FORTRAN:  idnint
 cvc$i_sfunc_ihnint,            {1: See FORTRAN:  ihnint
 cvc$i_sfunc_index,             {1: See FORTRAN:  index
 cvc$i_sfunc_insb,              {1: See FORTRAN:  insb    (boolean first arg)
 cvc$i_sfunc_isign,             {1: See FORTRAN:  isign
 cvc$i_sfunc_jabs,              {F: See FORTRAN:  jabs    (abs for i32)
 cvc$i_sfunc_jdim,              {F: See FORTRAN:  jdim    (dim for i32)
 cvc$i_sfunc_jmax0,             {F: See FORTRAN:  jmax0   (max for i32)
 cvc$i_sfunc_jmin0,             {F: See FORTRAN:  jmin0   (min for i32)
 cvc$i_sfunc_jmod,              {F: See FORTRAN:  jmod    (mod for i32)
 cvc$i_sfunc_jsign,             {F: See FORTRAN:  jsign   (sign for i32)
 cvc$i_sfunc_len,               {1: See FORTRAN:  len
 cvc$i_sfunc_leq_b01,           {2: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_leq_b01_collated,  {2: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_leq_bsign,         {1: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_leq_bsign_collated, {1: See FORTRAN: -See Note 1-
 cvc$i_sfunc_lge_b01,           {2: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_lge_b01_collated,  {2: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_lge_bsign,         {1: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_lge_bsign_collated, {1: See FORTRAN: -See Note 1-
 cvc$i_sfunc_lgt_b01,           {2: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_lgt_b01_collated,  {2: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_lgt_bsign,         {1: See FORTRAN:  -See Note 1-
 cvc$i_sfunc_lgt_bsign_collated, {1: See FORTRAN: -See Note 1-
```

84.0 CVT$I_INTRINSIC_ID

```
cvc$i_sfunc_lle_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_sfunc_lle_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_sfunc_lle_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_sfunc_lle_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_sfunc_llt_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_sfunc_llt_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_sfunc_llt_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_sfunc_llt_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_sfunc_lne_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_sfunc_lne_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_sfunc_lne_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_sfunc_lne_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_sfunc_ltob_b01,           {2: See FORTRAN:  ltob
cvc$i_sfunc_ltob_bsign,         {1: See FORTRAN:  ltob
cvc$i_sfunc_mask,               {1: See FORTRAN:  mask
cvc$i_sfunc_max0,               {1: See FORTRAN:  max0    (two args)
cvc$i_sfunc_max1,               {1: See FORTRAN:  max1    (two args)
cvc$i_sfunc_merge_128bit_b01,   {2: See FORTRAN:  merge
cvc$i_sfunc_merge_128bit_bsign, {1: See FORTRAN:  merge
cvc$i_sfunc_merge_16bit_b01,    {F: See FORTRAN:  merge
cvc$i_sfunc_merge_16bit_bsign,  {F: See FORTRAN:  merge
cvc$i_sfunc_merge_1bit_b01,     {2: See FORTRAN:  merge
cvc$i_sfunc_merge_1bit_bsign,   {1: See FORTRAN:  merge
cvc$i_sfunc_merge_256bit_b01,   {F: See FORTRAN:  merge
cvc$i_sfunc_merge_256bit_bsign, {F: See FORTRAN:  merge
cvc$i_sfunc_merge_32bit_b01,    {2: See FORTRAN:  merge
cvc$i_sfunc_merge_32bit_bsign,  {2: See FORTRAN:  merge
cvc$i_sfunc_merge_64bit_b01,    {2: See FORTRAN:  merge
cvc$i_sfunc_merge_64bit_bsign,  {1: See FORTRAN:  merge
cvc$i_sfunc_merge_8bit_b01,     {F: See FORTRAN:  merge
cvc$i_sfunc_merge_8bit_bsign,   {F: See FORTRAN:  merge
cvc$i_sfunc_min0,               {1: See FORTRAN:  min0    (two args)
cvc$i_sfunc_min1,               {1: See FORTRAN:  min1    (two args)
cvc$i_sfunc_mod,                {1: See FORTRAN:  mod     (ADA 'rem' operator)
cvc$i_sfunc_mod_ada,            {A: See ADA:      'mod' operator
cvc$i_sfunc_neqv_1bit,          {1: See FORTRAN:  neqv    (two args, bit)
cvc$i_sfunc_neqv_64bit,         {1: See FORTRAN:  neqv    (two args, boolean)
cvc$i_sfunc_nint,               {1: See FORTRAN:  nint
cvc$i_sfunc_not_1bit,           {1: See FORTRAN:  compl   (bit)
cvc$i_sfunc_not_64bit,          {1: See FORTRAN:  compl   (boolean)
cvc$i_sfunc_odd_b01,            {P: See PASCAL:   odd
cvc$i_sfunc_odd_bsign,          {P: See PASCAL:   odd
cvc$i_sfunc_or_1bit,            {1: See FORTRAN:  or      (two args, bit)
cvc$i_sfunc_or_64bit,           {1: See FORTRAN:  or      (two args, boolean)
cvc$i_sfunc_r128_to_c128_power, {1: See FORTRAN:  '**' operator
cvc$i_sfunc_r128_to_c256_power, {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r128_to_c64_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r128_to_i32_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r128_to_i64_power,  {1: See FORTRAN:  '**' operator
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_sfunc_r128_to_r128_power, {1: See FORTRAN:  '**' operator
cvc$i_sfunc_r128_to_r32_power,  {2: See FORTRAN:  '**' operator
cvc$i_sfunc_r128_to_r64_power,  {1: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_c128_power,  {2: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_c256_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_c64_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_i32_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_i64_power,   {2: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_r128_power,  {2: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_r32_power,   {2: See FORTRAN:  '**' operator
cvc$i_sfunc_r32_to_r64_power,   {2: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_c128_power,  {1: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_c256_power,  {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_c64_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_i32_power,   {F: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_i64_power,   {1: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_r128_power,  {1: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_r32_power,   {2: See FORTRAN:  '**' operator
cvc$i_sfunc_r64_to_r64_power,   {1: See FORTRAN:  '**' operator
cvc$i_sfunc_rprod,              {2: See FORTRAN:  rprod
cvc$i_sfunc_sgn_i64,            {B: See BASIC:    sgn     (i64 arg, i64 result)
cvc$i_sfunc_sgn_r64,            {B: See BASIC:    sgn     (r64 arg, i64 result)
cvc$i_sfunc_shift,              {1: See FORTRAN:  shift   (boolean first arg)
cvc$i_sfunc_sign,               {1: See FORTRAN:  sign
cvc$i_sfunc_sin,                {1: See FORTRAN:  sin
cvc$i_sfunc_sind,               {1: See FORTRAN:  sind
cvc$i_sfunc_sinh,               {1: See FORTRAN:  sinh
cvc$i_sfunc_sqrt,               {1: See FORTRAN:  sqrt
cvc$i_sfunc_tan,                {1: See FORTRAN:  tan
cvc$i_sfunc_tand,               {1: See FORTRAN:  tand
cvc$i_sfunc_tanh,               {1: See FORTRAN:  tanh
cvc$i_tcall_all_b01,            {2: See FORTRAN:  all     (two args)
cvc$i_tcall_all_bsign,          {1: See FORTRAN:  all     (two args)
cvc$i_tcall_any_b01,            {2: See FORTRAN:  any     (two args)
cvc$i_tcall_any_bsign,          {1: See FORTRAN:  any     (two args)
cvc$i_tcall_count_b01,          {2: See FORTRAN:  count   (two args)
cvc$i_tcall_count_bsign,        {1: See FORTRAN:  count   (two args)
cvc$i_tcall_diagonal_char,      {3: See FORTRAN:  diagonal
cvc$i_tcall_maxval_i32_b01,     {F: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_i32_bsign,   {F: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_i64_b01,     {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_i64_bsign,   {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_r128_b01,    {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_r128_bsign,  {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_r32_b01,     {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_r32_bsign,   {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_r64_b01,     {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_maxval_r64_bsign,   {3: See FORTRAN:  maxval  (three args)
cvc$i_tcall_minval_i32_b01,     {F: See FORTRAN:  minval  (three args)
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_tcall_minval_i32_bsign,    {F: See FORTRAN:  minval  (three args
cvc$i_tcall_minval_i64_b01,      {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_minval_i64_bsign,    {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_minval_r128_b01,     {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_minval_r128_bsign,   {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_minval_r32_b01,      {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_minval_r32_bsign,    {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_minval_r64_b01,      {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_minval_r64_bsign,    {3: See FORTRAN:  minval  (three args)
cvc$i_tcall_pack_char,           {1: See FORTRAN:  pack    (two args)
cvc$i_tcall_pack_insert_char,    {1: See FORTRAN:  pack    (three args)
cvc$i_tcall_product_c128_b01,    {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_c128_bsign,  {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_c256_b01,    {F: See FORTRAN:  product (three args)
cvc$i_tcall_product_c256_bsign,  {F: See FORTRAN:  product (three args)
cvc$i_tcall_product_c64_b01,     {F: See FORTRAN:  product (three args)
cvc$i_tcall_product_c64_bsign,   {F: See FORTRAN:  product (three args)
cvc$i_tcall_product_i32_b01,     {F: See FORTRAN:  product (three args)
cvc$i_tcall_product_i32_bsign,   {F: See FORTRAN:  product (three args)
cvc$i_tcall_product_i64_b01,     {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_i64_bsign,   {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_r128_b01,    {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_r128_bsign,  {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_r32_b01,     {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_r32_bsign,   {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_r64_b01,     {3: See FORTRAN:  product (three args)
cvc$i_tcall_product_r64_bsign,   {3: See FORTRAN:  product (three args)
cvc$i_tcall_replicate_128bit,    {3: See FORTRAN:  replicate
cvc$i_tcall_replicate_16bit,     {F: See FORTRAN:  replicate
cvc$i_tcall_replicate_1bit,      {3: See FORTRAN:  replicate
cvc$i_tcall_replicate_256bit,    {F: See FORTRAN:  replicate
cvc$i_tcall_replicate_32bit,     {3: See FORTRAN:  replicate
cvc$i_tcall_replicate_64bit,     {3: See FORTRAN:  replicate
cvc$i_tcall_replicate_8bit,      {F: See FORTRAN:  replicate
cvc$i_tcall_replicate_char,      {3: See FORTRAN:  replicate
cvc$i_tcall_spread_128bit,       {3: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_spread_16bit,        {F: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_spread_1bit,         {3: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_spread_256bit,       {F: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_spread_32bit,        {3: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_spread_64bit,        {3: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_spread_8bit,         {F: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_spread_char,         {3: See FORTRAN:  spread  (array first arg)
cvc$i_tcall_sum_c128_b01,        {2: See FORTRAN:  sum     (three args)
cvc$i_tcall_sum_c128_bsign,      {1: See FORTRAN:  sum     (three args)
cvc$i_tcall_sum_c256_b01,        {F: See FORTRAN:  sum     (three args)
cvc$i_tcall_sum_c256_bsign,      {F: See FORTRAN:  sum     (three args)
cvc$i_tcall_sum_c64_b01,         {F: See FORTRAN:  sum     (three args)
cvc$i_tcall_sum_c64_bsign,       {F: See FORTRAN:  sum     (three args)
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_tcall_sum_i32_b01,         {F: See FORTRAN:  sum      (three args
cvc$i_tcall_sum_i32_bsign,       {F: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_i64_b01,         {2: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_i64_bsign,       {1: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_r128_b01,        {2: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_r128_bsign,      {1: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_r32_b01,         {2: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_r32_bsign,       {2: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_r64_b01,         {2: See FORTRAN:  sum      (three args)
cvc$i_tcall_sum_r64_bsign,       {1: See FORTRAN:  sum      (three args)
cvc$i_tcall_transpose_char,      {3: See FORTRAN:  transpose
cvc$i_tcall_unpack_char,         {1: See FORTRAN:  unpack
cvc$i_tfunc_all_b01,             {2: See FORTRAN:  all      (two args)
cvc$i_tfunc_all_bsign,           {1: See FORTRAN:  all      (two args)
cvc$i_tfunc_alt_b01,             {3: See FORTRAN:  alt
cvc$i_tfunc_alt_bsign,           {3: See FORTRAN:  alt
cvc$i_tfunc_any_b01,             {2: See FORTRAN:  any      (two args)
cvc$i_tfunc_any_bsign,           {1: See FORTRAN:  any      (two args)
cvc$i_tfunc_count_b01,           {2: See FORTRAN:  count    (two args)
cvc$i_tfunc_count_bsign,         {1: See FORTRAN:  count    (two args)
cvc$i_tfunc_diagonal_128bit,     {3: See FORTRAN:  diagonal -See Note 4-
cvc$i_tfunc_diagonal_16bit,      {F: See FORTRAN:  diagonal -See Note 4-
cvc$i_tfunc_diagonal_1bit,       {3: See FORTRAN:  diagonal -See Note 4-
cvc$i_tfunc_diagonal_256bit,     {F: See FORTRAN:  diagonal -See Note 4-
cvc$i_tfunc_diagonal_32bit,      {3: See FORTRAN:  diagonal -See Note 4-
cvc$i_tfunc_diagonal_64bit,      {3: See FORTRAN:  diagonal -See Note 4-
cvc$i_tfunc_diagonal_8bit,       {F: See FORTRAN:  diagonal -See Note 4-
cvc$i_tfunc_matmul_b01,          {2: See FORTRAN:  matmul
cvc$i_tfunc_matmul_bsign,        {1: See FORTRAN:  matmul
cvc$i_tfunc_matmul_c128,         {1: See FORTRAN:  matmul
cvc$i_tfunc_matmul_c256,         {F: See FORTRAN:  matmul
cvc$i_tfunc_matmul_c64,          {F: See FORTRAN:  matmul
cvc$i_tfunc_matmul_i32,          {F: See FORTRAN:  matmul
cvc$i_tfunc_matmul_i64,          {1: See FORTRAN:  matmul
cvc$i_tfunc_matmul_r128,         {1: See FORTRAN:  matmul
cvc$i_tfunc_matmul_r32,          {2: See FORTRAN:  matmul
cvc$i_tfunc_matmul_r64,          {1: See FORTRAN:  matmul
cvc$i_tfunc_maxval_i32_b01,      {F: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_i32_bsign,    {F: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_i64_b01,      {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_i64_bsign,    {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_r128_b01,     {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_r128_bsign,   {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_r32_b01,      {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_r32_bsign,    {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_r64_b01,      {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_maxval_r64_bsign,    {3: See FORTRAN:  maxval   (three args)
cvc$i_tfunc_minval_i32_b01,      {F: See FORTRAN:  minval   (three args)
cvc$i_tfunc_minval_i32_bsign,    {F: See FORTRAN:  minval   (three args)
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_tfunc_minval_i64_b01,      {3: See FORTRAN:  minval  (three args
cvc$i_tfunc_minval_i64_bsign,    {3: See FORTRAN:  minval  (three args)
cvc$i_tfunc_minval_r128_b01,     {3: See FORTRAN:  minval  (three args)
cvc$i_tfunc_minval_r128_bsign,   {3: See FORTRAN:  minval  (three args)
cvc$i_tfunc_minval_r32_b01,      {3: See FORTRAN:  minval  (three args)
cvc$i_tfunc_minval_r32_bsign,    {3: See FORTRAN:  minval  (three args)
cvc$i_tfunc_minval_r64_b01,      {3: See FORTRAN:  minval  (three args)
cvc$i_tfunc_minval_r64_bsign,    {3: See FORTRAN:  minval  (three args)
cvc$i_tfunc_packin_128bit_b01,   {2: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_128bit_bsign, {1: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_16bit_b01,    {F: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_16bit_bsign,  {F: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_1bit_b01,     {2: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_1bit_bsign,   {1: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_256bit_b01,   {F: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_256bit_bsign, {F: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_32bit_b01,    {2: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_32bit_bsign,  {2: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_64bit_b01,    {2: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_64bit_bsign,  {1: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_8bit_b01,     {F: See FORTRAN:  pack    (three args)
cvc$i_tfunc_packin_8bit_bsign,   {F: See FORTRAN:  pack    (three args)
cvc$i_tfunc_pack_128bit_b01,     {2: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_128bit_bsign,   {1: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_16bit_b01,      {F: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_16bit_bsign,    {F: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_1bit_b01,       {2: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_1bit_bsign,     {1: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_256bit_b01,     {F: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_256bit_bsign,   {F: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_32bit_b01,      {2: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_32bit_bsign,    {2: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_64bit_b01,      {2: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_64bit_bsign,    {1: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_8bit_b01,       {F: See FORTRAN:  pack    (two args)
cvc$i_tfunc_pack_8bit_bsign,     {F: See FORTRAN:  pack    (two args)
cvc$i_tfunc_product_c128_b01,    {3: See FORTRAN:  product (three args)
cvc$i_tfunc_product_c128_bsign,  {3: See FORTRAN:  product (three args)
cvc$i_tfunc_product_c256_b01,    {F: See FORTRAN:  product (three args)
cvc$i_tfunc_product_c256_bsign,  {F: See FORTRAN:  product (three args)
cvc$i_tfunc_product_c64_b01,     {F: See FORTRAN:  product (three args)
cvc$i_tfunc_product_c64_bsign,   {F: See FORTRAN:  product (three args)
cvc$i_tfunc_product_i32_b01,     {F: See FORTRAN:  product (three args)
cvc$i_tfunc_product_i32_bsign,   {F: See FORTRAN:  product (three args)
cvc$i_tfunc_product_i64_b01,     {3: See FORTRAN:  product (three args)
cvc$i_tfunc_product_i64_bsign,   {3: See FORTRAN:  product (three args)
cvc$i_tfunc_product_r128_b01,    {3: See FORTRAN:  product (three args)
cvc$i_tfunc_product_r128_bsign,  {3: See FORTRAN:  product (three args)
cvc$i_tfunc_product_r32_b01,     {3: See FORTRAN:  product (three args)
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_tfunc_product_r32_bsign,  {3: See FORTRAN:  product (three args
cvc$i_tfunc_product_r64_b01,    {3: See FORTRAN:  product (three args)
cvc$i_tfunc_product_r64_bsign,  {3: See FORTRAN:  product (three args)
cvc$i_tfunc_replicate_128bit,   {3: See FORTRAN:  replicate
cvc$i_tfunc_replicate_16bit,    {F: See FORTRAN:  replicate
cvc$i_tfunc_replicate_1bit,     {3: See FORTRAN:  replicate
cvc$i_tfunc_replicate_256bit,   {F: See FORTRAN:  replicate
cvc$i_tfunc_replicate_32bit,    {3: See FORTRAN:  replicate
cvc$i_tfunc_replicate_64bit,    {3: See FORTRAN:  replicate
cvc$i_tfunc_replicate_8bit,     {F: See FORTRAN:  replicate
cvc$i_tfunc_seq_i32,            {F: See FORTRAN:  seq     (three args)
cvc$i_tfunc_seq_i64,            {3: See FORTRAN:  seq     (three args)
cvc$i_tfunc_spread_128bit,      {3: See FORTRAN:  spread  (array first arg)
cvc$i_tfunc_spread_16bit,       {F: See FORTRAN:  spread  (array first arg)
cvc$i_tfunc_spread_1bit,        {3: See FORTRAN:  spread  (array first arg)
cvc$i_tfunc_spread_256bit,      {F: See FORTRAN:  spread  (array first arg)
cvc$i_tfunc_spread_32bit,       {3: See FORTRAN:  spread  (array first arg)
cvc$i_tfunc_spread_64bit,       {3: See FORTRAN:  spread  (array first arg)
cvc$i_tfunc_spread_8bit,        {F: See FORTRAN:  spread  (array first arg)
cvc$i_tfunc_sum_c128_b01,       {2: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_c128_bsign,     {1: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_c256_b01,       {F: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_c256_bsign,     {F: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_c64_b01,        {F: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_c64_bsign,      {F: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_i32_b01,        {F: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_i32_bsign,      {F: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_i64_b01,        {2: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_i64_bsign,      {1: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_r128_b01,       {2: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_r128_bsign,     {1: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_r32_b01,        {2: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_r32_bsign,      {2: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_r64_b01,        {2: See FORTRAN:  sum     (three args)
cvc$i_tfunc_sum_r64_bsign,      {1: See FORTRAN:  sum     (three args)
cvc$i_tfunc_transpose_128bit,   {3: See FORTRAN:  transpose
cvc$i_tfunc_transpose_16bit,    {F: See FORTRAN:  transpose
cvc$i_tfunc_transpose_1bit,     {3: See FORTRAN:  transpose
cvc$i_tfunc_transpose_256bit,   {F: See FORTRAN:  transpose
cvc$i_tfunc_transpose_32bit,    {3: See FORTRAN:  transpose
cvc$i_tfunc_transpose_64bit,    {3: See FORTRAN:  transpose
cvc$i_tfunc_transpose_8bit,     {F: See FORTRAN:  transpose
cvc$i_tfunc_unpack_128bit_b01,  {2: See FORTRAN:  unpack
cvc$i_tfunc_unpack_128bit_bsign,{1: See FORTRAN:  unpack
cvc$i_tfunc_unpack_16bit_b01,   {F: See FORTRAN:  unpack
cvc$i_tfunc_unpack_16bit_bsign, {F: See FORTRAN:  unpack
cvc$i_tfunc_unpack_1bit_b01,    {2: See FORTRAN:  unpack
cvc$i_tfunc_unpack_1bit_bsign,  {1: See FORTRAN:  unpack
cvc$i_tfunc_unpack_256bit_b01,  {F: See FORTRAN:  unpack
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_tfunc_unpack_256bit_bsign,{F: See FORTRAN:  unpack
cvc$i_tfunc_unpack_32bit_b01,   {2: See FORTRAN:  unpack
cvc$i_tfunc_unpack_32bit_bsign, {2: See FORTRAN:  unpack
cvc$i_tfunc_unpack_64bit_b01,   {2: See FORTRAN:  unpack
cvc$i_tfunc_unpack_64bit_bsign, {1: See FORTRAN:  unpack
cvc$i_tfunc_unpack_8bit_b01,    {F: See FORTRAN:  unpack
cvc$i_tfunc_unpack_8bit_bsign,  {F: See FORTRAN:  unpack
cvc$i_vcall_char,               {1: See FORTRAN:  char    (FIXED collation)
cvc$i_vcall_char_collated,      {1: See FORTRAN:  char    (USER collation)
cvc$i_vcall_merge_char_b01,     {2: See FORTRAN:  merge
cvc$i_vcall_merge_char_bsign,   {1: See FORTRAN:  merge
cvc$i_vfunc_abs,                {1: See FORTRAN:  abs
cvc$i_vfunc_acos,               {1: See FORTRAN:  acos
cvc$i_vfunc_aimag,              {1: See FORTRAN:  aimag
cvc$i_vfunc_aint,               {1: See FORTRAN:  aint
cvc$i_vfunc_alog,               {1: See FORTRAN:  alog
cvc$i_vfunc_alog10,             {1: See FORTRAN:  alog10
cvc$i_vfunc_amax0,              {1: See FORTRAN:  amax0   (two args)
cvc$i_vfunc_amax1,              {1: See FORTRAN:  amax1   (two args)
cvc$i_vfunc_amin0,              {1: See FORTRAN:  amin0   (two args)
cvc$i_vfunc_amin1,              {1: See FORTRAN:  amin1   (two args)
cvc$i_vfunc_amod,               {1: See FORTRAN:  amod
cvc$i_vfunc_and_1bit,           {1: See FORTRAN:  and     (two args, bit)
cvc$i_vfunc_and_64bit,          {1: See FORTRAN:  and     (two args, boolean)
cvc$i_vfunc_anint,              {1: See FORTRAN:  anint
cvc$i_vfunc_asin,               {1: See FORTRAN:  asin
cvc$i_vfunc_atan,               {1: See FORTRAN:  atan
cvc$i_vfunc_atan2,              {1: See FORTRAN:  atan2
cvc$i_vfunc_atanh,              {1: See FORTRAN:  atanh
cvc$i_vfunc_bool_of_char,       {1: See FORTRAN:  bool    (character arg only)
cvc$i_vfunc_btol_b01,           {2: See FORTRAN:  btol
cvc$i_vfunc_btol_bsign,         {1: See FORTRAN:  btol
cvc$i_vfunc_c128_to_c128_power, {1: See FORTRAN:  '**' operator
cvc$i_vfunc_c128_to_c256_power, {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c128_to_c64_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c128_to_i32_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c128_to_i64_power,  {1: See FORTRAN:  '**' operator
cvc$i_vfunc_c128_to_r128_power, {1: See FORTRAN:  '**' operator
cvc$i_vfunc_c128_to_r32_power,  {2: See FORTRAN:  '**' operator
cvc$i_vfunc_c128_to_r64_power,  {1: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_c128_power, {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_c256_power, {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_c64_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_i32_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_i64_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_r128_power, {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_r32_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c256_to_r64_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c64_to_c128_power,  {F: See FORTRAN:  '**' operator
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_vfunc_c64_to_c256_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c64_to_c64_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c64_to_i32_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c64_to_i64_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c64_to_r128_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c64_to_r32_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_c64_to_r64_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_cabs,               {1: See FORTRAN:  cabs
cvc$i_vfunc_ccos,               {1: See FORTRAN:  ccos
cvc$i_vfunc_cdabs,              {F: See FORTRAN:  cdabs   (abs for c256)
cvc$i_vfunc_cdcos,              {F: See FORTRAN:  cdcos   (cos for c256)
cvc$i_vfunc_cdexp,              {F: See FORTRAN:  cdexp   (exp for c256)
cvc$i_vfunc_cdlog,              {F: See FORTRAN:  cdlog   (log for c256)
cvc$i_vfunc_cdsin,              {F: See FORTRAN:  cdsin   (sin for c256)
cvc$i_vfunc_cdsqrt,             {F: See FORTRAN:  cdsqrt  (sqrt for c256)
cvc$i_vfunc_ceil,               {B: See BASIC:    ceil    (r64 arg and result)
cvc$i_vfunc_cexp,               {1: See FORTRAN:  cexp
cvc$i_vfunc_chabs,              {F: See FORTRAN:  chabs   (abs for c64)
cvc$i_vfunc_chcos,              {F: See FORTRAN:  chcos   (cos for c64)
cvc$i_vfunc_chexp,              {F: See FORTRAN:  chexp   (exp for c64)
cvc$i_vfunc_chlog,              {F: See FORTRAN:  chlog   (log for c64)
cvc$i_vfunc_chsin,              {F: See FORTRAN:  chsin   (sin for c64)
cvc$i_vfunc_chsqrt,             {F: See FORTRAN:  chsqrt  (sqrt for c64)
cvc$i_vfunc_clog,               {1: See FORTRAN:  clog
cvc$i_vfunc_cmplx,              {1: See FORTRAN:  cmplx   (two args, real)
cvc$i_vfunc_conjg,              {1: See FORTRAN:  conjg
cvc$i_vfunc_cos,                {1: See FORTRAN:  cos
cvc$i_vfunc_cosd,               {1: See FORTRAN:  cosd
cvc$i_vfunc_cosh,               {1: See FORTRAN:  cosh
cvc$i_vfunc_cotan,              {1: See FORTRAN:  cotan
cvc$i_vfunc_csin,               {1: See FORTRAN:  csin
cvc$i_vfunc_csqrt,              {1: See FORTRAN:  csqrt
cvc$i_vfunc_dabs,               {1: See FORTRAN:  dabs
cvc$i_vfunc_dacos,              {1: See FORTRAN:  dacos
cvc$i_vfunc_dasin,              {1: See FORTRAN:  dasin
cvc$i_vfunc_datan,              {1: See FORTRAN:  datan
cvc$i_vfunc_datan2,             {1: See FORTRAN:  datan2
cvc$i_vfunc_dconjg,             {F: See FORTRAN:  dconjg  (conjg for c256)
cvc$i_vfunc_dcos,               {1: See FORTRAN:  dcos
cvc$i_vfunc_dcosh,              {1: See FORTRAN:  dcosh
cvc$i_vfunc_dcotan,             {F: See FORTRAN:  dcotan
cvc$i_vfunc_ddim,               {1: See FORTRAN:  ddim
cvc$i_vfunc_dexp,               {1: See FORTRAN:  dexp
cvc$i_vfunc_dim,                {1: See FORTRAN:  dim
cvc$i_vfunc_dimag,              {F: See FORTRAN:  dimag   (aimag for c256)
cvc$i_vfunc_dint,               {1: See FORTRAN:  dint
cvc$i_vfunc_dlog,               {1: See FORTRAN:  dlog
cvc$i_vfunc_dlog10,             {1: See FORTRAN:  dlog10
cvc$i_vfunc_dmax1,              {1: See FORTRAN:  dmax1   (two args)
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_vfunc_dmin1,               {1: See FORTRAN:  dmin1   (two args)
cvc$i_vfunc_dmod,                {1: See FORTRAN:  dmod
cvc$i_vfunc_dnint,               {1: See FORTRAN:  dnint
cvc$i_vfunc_dprod,               {1: See FORTRAN:  dprod
cvc$i_vfunc_dsign,               {1: See FORTRAN:  dsign
cvc$i_vfunc_dsin,                {1: See FORTRAN:  dsin
cvc$i_vfunc_dsinh,               {1: See FORTRAN:  dsinh
cvc$i_vfunc_dsqrt,               {1: See FORTRAN:  dsqrt
cvc$i_vfunc_dtan,                {1: See FORTRAN:  dtan
cvc$i_vfunc_dtanh,               {1: See FORTRAN:  dtanh
cvc$i_vfunc_eqv_1bit,            {1: See FORTRAN:  eqv     (two args, bit)
cvc$i_vfunc_eqv_64bit,           {1: See FORTRAN:  eqv     (two args, boolean)
cvc$i_vfunc_erf,                 {1: See FORTRAN:  erf
cvc$i_vfunc_erfc,                {1: See FORTRAN:  erfc
cvc$i_vfunc_exp,                 {1: See FORTRAN:  exp
cvc$i_vfunc_extb,                {1: See FORTRAN:  extb    (boolean first arg)
cvc$i_vfunc_floor,               {B: See BASIC:    int     (r64 arg and result)
cvc$i_vfunc_fract_part,          {B: See BASIC:    fp      (r64 arg and result)
cvc$i_vfunc_habs,                {2: See FORTRAN:  habs
cvc$i_vfunc_hacos,               {2: See FORTRAN:  hacos
cvc$i_vfunc_hasin,               {2: See FORTRAN:  hasin
cvc$i_vfunc_hatan,               {2: See FORTRAN:  hatan
cvc$i_vfunc_hatan2,              {2: See FORTRAN:  hatan2
cvc$i_vfunc_hconjg,              {F: See FORTRAN:  hconjg  (conjg for c64)
cvc$i_vfunc_hcos,                {2: See FORTRAN:  hcos
cvc$i_vfunc_hcosh,               {2: See FORTRAN:  hcosh
cvc$i_vfunc_hcotan,              {2: See FORTRAN:  hcotan
cvc$i_vfunc_hdim,                {2: See FORTRAN:  hdim
cvc$i_vfunc_hexp,                {2: See FORTRAN:  hexp
cvc$i_vfunc_himag,               {F: See FORTRAN:  himag   (aimag for c64)
cvc$i_vfunc_hint,                {2: See FORTRAN:  hint
cvc$i_vfunc_hlog,                {2: See FORTRAN:  hlog
cvc$i_vfunc_hlog10,              {2: See FORTRAN:  hlog10
cvc$i_vfunc_hmax1,               {2: See FORTRAN:  hmax1   (two args)
cvc$i_vfunc_hmin1,               {2: See FORTRAN:  hmin1   (two args)
cvc$i_vfunc_hmod,                {2: See FORTRAN:  hmod
cvc$i_vfunc_hnint,               {2: See FORTRAN:  hnint
cvc$i_vfunc_hsign,               {2: See FORTRAN:  hsign
cvc$i_vfunc_hsin,                {2: See FORTRAN:  hsin
cvc$i_vfunc_hsinh,               {2: See FORTRAN:  hsinh
cvc$i_vfunc_hsqrt,               {2: See FORTRAN:  hsqrt
cvc$i_vfunc_htan,                {2: See FORTRAN:  htan
cvc$i_vfunc_htanh,               {2: See FORTRAN:  htanh
cvc$i_vfunc_i32_to_c128_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i32_to_c256_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i32_to_c64_power,    {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i32_to_i32_power,    {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i32_to_i64_power,    {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i32_to_r128_power,   {F: See FORTRAN:  '**' operator
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_vfunc_i32_to_r32_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i32_to_r64_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_c128_power,  {1: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_c256_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_c64_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_i32_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_i64_power,   {1: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_r128_power,  {1: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_r32_power,   {2: See FORTRAN:  '**' operator
cvc$i_vfunc_i64_to_r64_power,   {1: See FORTRAN:  '**' operator
cvc$i_vfunc_iabs,               {1: See FORTRAN:  iabs
cvc$i_vfunc_ichar,              {1: See FORTRAN:  ichar   (FIXED collation)
cvc$i_vfunc_ichar_collated,     {1: See FORTRAN:  ichar   (USER collation)
cvc$i_vfunc_idim,               {1: See FORTRAN:  idim
cvc$i_vfunc_idnint,             {1: See FORTRAN:  idnint
cvc$i_vfunc_ihnint,             {1: See FORTRAN:  ihnint
cvc$i_vfunc_index,              {1: See FORTRAN:  index
cvc$i_vfunc_insb,               {1: See FORTRAN:  insb    (boolean first arg)
cvc$i_vfunc_isign,              {1: See FORTRAN:  isign
cvc$i_vfunc_jabs,               {F: See FORTRAN:  jabs    (abs for i32)
cvc$i_vfunc_jdim,               {F: See FORTRAN:  jdim    (dim for i32)
cvc$i_vfunc_jmax0,              {F: See FORTRAN:  jmax0   (max for i32)
cvc$i_vfunc_jmin0,              {F: See FORTRAN:  jmin0   (min for i32)
cvc$i_vfunc_jmod,               {F: See FORTRAN:  jmod    (mod for i32)
cvc$i_vfunc_jsign,              {F: See FORTRAN:  jsign   (sign for i32)
cvc$i_vfunc_len,                {1: See FORTRAN:  len
cvc$i_vfunc_leq_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_leq_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_leq_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_leq_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lge_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lge_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lge_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lge_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lgt_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lgt_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lgt_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lgt_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lle_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lle_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lle_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lle_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_llt_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_llt_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_llt_bsign,          {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_llt_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lne_b01,            {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lne_b01_collated,   {2: See FORTRAN:  -See Note 1-
cvc$i_vfunc_lne_bsign,          {1: See FORTRAN:  -See Note 1-
```

B4.0 CVT$I_INTRINSIC_ID

```
cvc$i_vfunc_lne_bsign_collated, {1: See FORTRAN:  -See Note 1-
cvc$i_vfunc_ltob_b01,           {2: See FORTRAN:  ltob
cvc$i_vfunc_ltob_bsign,         {1: See FORTRAN:  ltob
cvc$i_vfunc_mask,               {1: See FORTRAN:  mask
cvc$i_vfunc_max0,               {1: See FORTRAN:  max0    (two args)
cvc$i_vfunc_max1,               {1: See FORTRAN:  max1    (two args)
cvc$i_vfunc_merge_128bit_b01,   {2: See FORTRAN:  merge
cvc$i_vfunc_merge_128bit_bsign, {1: See FORTRAN:  merge
cvc$i_vfunc_merge_16bit_b01,    {F: See FORTRAN:  merge
cvc$i_vfunc_merge_16bit_bsign,  {F: See FORTRAN:  merge
cvc$i_vfunc_merge_1bit_b01,     {2: See FORTRAN:  merge
cvc$i_vfunc_merge_1bit_bsign,   {1: See FORTRAN:  merge
cvc$i_vfunc_merge_256bit_b01,   {F: See FORTRAN:  merge
cvc$i_vfunc_merge_256bit_bsign, {F: See FORTRAN:  merge
cvc$i_vfunc_merge_32bit_b01,    {2: See FORTRAN:  merge
cvc$i_vfunc_merge_32bit_bsign,  {2: See FORTRAN:  merge
cvc$i_vfunc_merge_64bit_b01,    {2: See FORTRAN:  merge
cvc$i_vfunc_merge_64bit_bsign,  {1: See FORTRAN:  merge
cvc$i_vfunc_merge_8bit_b01,     {F: See FORTRAN:  merge
cvc$i_vfunc_merge_8bit_bsign,   {F: See FORTRAN:  merge
cvc$i_vfunc_min0,               {1: See FORTRAN:  min0    (two args)
cvc$i_vfunc_min1,               {1: See FORTRAN:  min1    (two args)
cvc$i_vfunc_mod,                {1: See FORTRAN:  mod     (ADA 'rem' operator)
cvc$i_vfunc_mod_ada,            {A: See ADA:      'mod' operator
cvc$i_vfunc_neqv_1bit,          {1: See FORTRAN:  neqv    (two args, bit)
cvc$i_vfunc_neqv_64bit,         {1: See FORTRAN:  neqv    (two args, boolean)
cvc$i_vfunc_nint,               {1: See FORTRAN:  nint
cvc$i_vfunc_not_1bit,           {1: See FORTRAN:  compl   (bit)
cvc$i_vfunc_not_64bit,          {1: See FORTRAN:  compl   (boolean)
cvc$i_vfunc_odd_bol,            {P: See PASCAL:   odd
cvc$i_vfunc_odd_bsign,          {P: See PASCAL:   odd
cvc$i_vfunc_or_1bit,            {1: See FORTRAN:  or      (two args, bit)
cvc$i_vfunc_or_64bit,           {1: See FORTRAN:  or      (two args, boolean)
cvc$i_vfunc_r128_to_c128_power, {1: See FORTRAN:  '**' operator
cvc$i_vfunc_r128_to_c256_power, {F: See FORTRAN:  '**' operator
cvc$i_vfunc_r128_to_c64_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_r128_to_i32_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_r128_to_i64_power,  {1: See FORTRAN:  '**' operator
cvc$i_vfunc_r128_to_r128_power, {1: See FORTRAN:  '**' operator
cvc$i_vfunc_r128_to_r32_power,  {2: See FORTRAN:  '**' operator
cvc$i_vfunc_r128_to_r64_power,  {1: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_c128_power,  {2: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_c256_power,  {F: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_c64_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_i32_power,   {F: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_i64_power,   {2: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_r128_power,  {2: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_r32_power,   {2: See FORTRAN:  '**' operator
cvc$i_vfunc_r32_to_r64_power,   {2: See FORTRAN:  '**' operator
```

```
 cvc$i_vfunc_r64_to_c128_power,  {1: See FORTRAN:  '**' operator
 cvc$i_vfunc_r64_to_c256_power,  {F: See FORTRAN:  '**' operator
 cvc$i_vfunc_r64_to_c64_power,   {F: See FORTRAN:  '**' operator
 cvc$i_vfunc_r64_to_i32_power,   {F: See FORTRAN:  '**' operator
 cvc$i_vfunc_r64_to_i64_power,   {1: See FORTRAN:  '**' operator
 cvc$i_vfunc_r64_to_r128_power,  {1: See FORTRAN:  '**' operator
 cvc$i_vfunc_r64_to_r32_power,   {2: See FORTRAN:  '**' operator
 cvc$i_vfunc_r64_to_r64_power,   {1: See FORTRAN:  '**' operator
 cvc$i_vfunc_rprod,              {2: See FORTRAN:  rprod
 cvc$i_vfunc_sgn_i64,            {B: See BASIC:    sgn     (i64 arg, i64 result)
 cvc$i_vfunc_sgn_r64,            {B: See BASIC:    sgn     (r64 arg, i64 result)
 cvc$i_vfunc_shift,              {1: See FORTRAN:  shift   (boolean first arg)
 cvc$i_vfunc_sign,               {1: See FORTRAN:  sign
 cvc$i_vfunc_sin,                {1: See FORTRAN:  sin
 cvc$i_vfunc_sind,               {1: See FORTRAN:  sind
 cvc$i_vfunc_sinh,               {1: See FORTRAN:  sinh
 cvc$i_vfunc_sqrt,               {1: See FORTRAN:  sqrt
 cvc$i_vfunc_tan,                {1: See FORTRAN:  tan
 cvc$i_vfunc_tand,               {1: See FORTRAN:  tand
 cvc$i_vfunc_tanh                {1: See FORTRAN:  tanh

    );

{ Notes:
{
{  1) All of the following code generator intrinsics appear in FORTRAN as
{ character relational operators.  Some also appear in FORTRAN as
{ specific intrinsics.
{
{   _leq_b01,              '.EQ.' operator                     (FIXED collation)
{   _leq_b01_collated,     '.EQ.' operator                     (USER collation)
{   _leq_bsign,            '.EQ.' operator                     (FIXED collation)
{   _leq_bsign_collated,   '.EQ.' operator                     (USER collation)
{   _lge_b01,              '.GE.' operator, LGE intrinsic      (FIXED collation)
{   _lge_b01_collated,     '.GE.' operator                     (USER collation)
{   _lge_bsign,            '.GE.' operator, LGE intrinsic      (FIXED collation)
{   _lge_bsign_collated,   '.GE.' operator                     (USER collation)
{   _lgt_b01,              '.GT.' operator, LGT intrinsic      (FIXED collation)
{   _lgt_b01_collated,     '.GT.' operator                     (USER collation)
{   _lgt_bsign,            '.GT.' operator, LGT intrinsic      (FIXED collation)
{   _lgt_bsign_collated,   '.GT.' operator                     (USER collation)
{   _lle_b01,              '.LE.' operator, LLE intrinsic      (FIXED collation)
{   _lle_b01_collated,     '.LE.' operator                     (USER collation)
{   _lle_bsign,            '.LE.' operator, LLE intrinsic      (FIXED collation)
{   _lle_bsign_collated,   '.LE.' operator                     (USER collation)
{   _llt_b01,              '.LT.' operator, LLT intrinsic      (FIXED collation)
{   _llt_b01_collated,     '.LT.' operator                     (USER collation)
{   _llt_bsign,            '.LT.' operator, LLT intrinsic      (FIXED collation)
{   _llt_bsign_collated,   '.LT.' operator                     (USER collation)
```

B4.0 CVT$I_INTRINSIC_ID

```
{    _ine_b01,              '.NE.' operator                        (FIXED co  ation)
{    _ine_b01_collated,     '.NE.' operator                        (USER collation)
{    _ine_bsign,            '.NE.' operator                        (FIXED collation)
{    _ine_bsign_collated,   '.NE.' operator                        (USER collation)
{
{  2) Function "cvc$i_mfunc_ranf" corresponds to the FORTRAN "ranf"
{ intrinsic having no source arguments or having a scalar source
{ argument.  In either case the code generator intrinsic has no
{ arguments.  Function "cvc$i_mcall_ranf" corresponds to the FORTRAN
{ "ranf" intrinsic having an array argument and result.  In this case the
{ code generator intrinsic has no arguments, but has an array result
{ (appearing as the first and only subroutine argument).  For both scalar
{ and array versions of "ranf", the generated CYBER 180 code references
{ the variables "mlv$random_multiplier" and "mlv$random_seed".
{
{  3) Functions "cvc$i_mcall_ranget" and "cvc$i_mcall_ranset" correspond
{ to the FORTRAN processor-supplied subroutines "RANGET" and "RANSET"
{ respectively.  The generated CYBER 180 code references the variables
{ "mlv$random_seed" and (RANSET only) "mlv$initial_seed".
{
{  4) Except for "cvc$i_tcall_diagonal_char", the Code Generator
{ "diagonal" intrinsics require an extra (third) argument which is the
{ (constant) default value for all array result elements which are not on
{ the array diagonal.  For "cvc$i_tcall_diagonal_char", a blank string
{ is always used for the default value, so there are only two arguments.
{
{  5) Functions "cvc$i_mcall_scan_b01" and "cvc$i_mcall_scan_bsign"
{ correspond to the CYBIL intrinsic "#scan" with the following restrictions.
{ The third operand must be a variable with the code generator type of
{ cvt$i_type_integer_64, and the fourth operand must be a variable with
{ the code generator type of either cvt$i_type_boolean_0_1 or
{ cvt$i_type_boolean_sign (as reflected in the function name).  Both the
{ third and fourth operands must be variables with a length of 64 bits.

?? FMT (FORMAT := ON) ??
```

C1.0 INSTRUCTION_OPCODE_USAGE

CVCG processes (optimizes, vectorizes, etc.) code for a sequence of instructions passed to it by the Host. The set of possible instructions can be thought of as the assembly language for an abstract computer. Note that the actual code generated for a particular machine will not be a one-for-one translation from the instructions passed by the Host. CVCG supports a large number of instruction opcodes. Inline code will be generated for all of these opcodes on the CYBER 180, and for most of these opcodes on the CYBER 200. The rest of these opcodes will be generated as calls to library routines. The parameter list for such a library call is always placed in registers.

In order for the appropriate library routines to be present at execution time, the Host must include the appropriate library in the library_list field of the code_generator_attributes parameter of the cvp$i_begin_module call.

C2.0 INSTRUCTION_OPCODE_NAMING_CONVENTIONS

-to be added later-

C3.0 INSTRUCTION_OPCODE_DEFINITIONS

-to be added later-

C4.0 CVT$I_CODE_GENERATOR_OPCODE

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
?? PUSH (LISTEXT := ON) ??

?? POP ??

?? PUSH (LIST := ON) ??
{ cvt$i_code_generator_opcode }
?? POP ??

?? FMT (FORMAT := OFF) ??

TYPE
  cvt$i_code_generator_opcode = (

{ Instructions available in Interface, with result (aka Hashed Result):

    cvc$i_op_b_and,       {bit string logical and

    cvc$i_op_b_andn,      {bit string logical and_not

    cvc$i_op_b_biteq,     {bit string compare equal

    cvc$i_op_b_bitne,     {bit string compare not_equal

    cvc$i_op_b_cat,       {BDP string concatenation

    cvc$i_op_b_ceq,       {BDP collated compare equal

    cvc$i_op_b_ceq_i,     { -to be deleted-

    cvc$i_op_b_cge,       {BDP collated compare greater_or_equal

    cvc$i_op_b_cge_i,     { -to be deleted-

    cvc$i_op_b_cgt,       {BDP collated compare greater_than

    cvc$i_op_b_cgt_i,     { -to be deleted-

    cvc$i_op_b_cle,       {BDP collated compare less_or_equal

    cvc$i_op_b_cle_i,     { -to be deleted-

    cvc$i_op_b_clt,       {BDP collated compare less_than

    cvc$i_op_b_clt_i,     { -to be deleted-
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
cvc$i_op_b_cne,       {BDP collated compare not_equal
cvc$i_op_b_cne_i,     {BDP index of collated compare not_equal
cvc$i_op_b_deq,       {BDP decimal compare equal
cvc$i_op_b_dge,       {BDP decimal compare greater_or_equal
cvc$i_op_b_dgt,       {BDP decimal compare greater_than
cvc$i_op_b_dle,       {BDP decimal compare less_or_equal
cvc$i_op_b_dlt,       {BDP decimal compare less_than
cvc$i_op_b_dne,       {BDP decimal compare not_equal
cvc$i_op_b_eqv,       {bit string logical equivalent
cvc$i_op_b_lcmp_bc,   {BDP lexical compare with broadcast constant
cvc$i_op_b_leq,       {BDP lexical compare equal
cvc$i_op_b_leq_i,     { -to be deleted-
cvc$i_op_b_lge,       {BDP lexical compare greater_or_equal
cvc$i_op_b_lge_i,     { -to be deleted-
cvc$i_op_b_lgt,       {BDP lexical compare greater_than
cvc$i_op_b_lgt_i,     { -to be deleted-
cvc$i_op_b_lle,       {BDP lexical compare less_or_equal
cvc$i_op_b_lle_i,     { -to be deleted-
cvc$i_op_b_llt,       {BDP lexical compare less_than
cvc$i_op_b_llt_i,     { -to be deleted-
cvc$i_op_b_lne,       {BDP lexical compare not_equal
cvc$i_op_b_lne_i,     {BDP index of lexical compare not_equal
cvc$i_op_b_nand,      {bit string logical nand
cvc$i_op_b_nor,       {bit string logical nor
cvc$i_op_b_not,       {bit string logical not
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
    cvc$i_op_b_or,        {bit string logical or
    cvc$i_op_b_orn,       {bit string logical or_not
    cvc$i_op_b_scan,      {BDP string scan for member
    cvc$i_op_b_scan_i,    {BDP string index of scan for member
    cvc$i_op_b_xor,       {bit string logical xor
    cvc$i_op_call,        {subroutine call
    cvc$i_op_call_p,      {subroutine call with specific parameter list pointer
    cvc$i_op_entry,       {procedure entry
    cvc$i_op_func,        {function call
    cvc$i_op_icall,       {intrinsic subroutine call
    cvc$i_op_paramptr,    {create specific parameter list pointer
    cvc$i_op_pop,         {pop from stack
    cvc$i_op_ptradd,      {add integer to pointer value
    cvc$i_op_p_arrelt,    {describe array element reference
    cvc$i_op_p_arrref,    {describe array reference
    cvc$i_op_p_arrsec,    {describe array section reference
    cvc$i_op_p_field,     {describe record field reference
    cvc$i_op_p_list,      {describe list of operands
    cvc$i_op_p_l_list,    {describe list of labels
    cvc$i_op_p_param,     {describe actual parameter
    cvc$i_op_p_recelt,    {describe pseudo-array element reference
    cvc$i_op_p_recsec,    {describe pseudo-array section reference
    cvc$i_op_p_ref,       { -unneeded?-
    cvc$i_op_p_substr,    {describe substring reference
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
cvc$i_op_reset,      {reset stack pointer to previous stack frame
cvc$i_op_s_add,      {scalar numeric add
cvc$i_op_s_and,      {scalar logical and
cvc$i_op_s_andn,     {scalar logical and_not
cvc$i_op_s_conv,     {scalar numeric type conversion
cvc$i_op_s_div,      {scalar numeric divide
cvc$i_op_s_eq,       {scalar numeric compare equal
cvc$i_op_s_eqv,      {scalar logical equivalent
cvc$i_op_s_ge,       {scalar numeric compare greater_or_equal
cvc$i_op_s_gt,       {scalar numeric compare greater_than
cvc$i_op_s_ifunc,    {scalar intrinsic function call
cvc$i_op_s_le,       {scalar numeric compare less_or_equal
cvc$i_op_s_lt,       {scalar numeric compare less_than
cvc$i_op_s_mul,      {scalar numeric multiply
cvc$i_op_s_nand,     {scalar logical nand
cvc$i_op_s_ne,       {scalar numeric compare not_equal
cvc$i_op_s_nor,      {scalar logical nor
cvc$i_op_s_not,      {scalar logical not
cvc$i_op_s_or,       {scalar logical or
cvc$i_op_s_orn,      {scalar logical or_not
cvc$i_op_s_ranf,     {scalar ranf intrinsic function call
cvc$i_op_s_shfc,     {scalar circular shift
cvc$i_op_s_shfe,     {scalar end_off shift
cvc$i_op_s_sub,      {scalar numeric subtract
cvc$i_op_s_xor,      {scalar logical xor
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
    cvc$i_op_v_add,      {vector numeric add
    cvc$i_op_v_and,      {vector logical and
    cvc$i_op_v_andn,     {vector logical and_not
    cvc$i_op_v_conv,     {vector numeric type conversion
    cvc$i_op_v_div,      {vector numeric divide
    cvc$i_op_v_eq,       {vector numeric compare equal
    cvc$i_op_v_eqv,      {vector logical equivalent
    cvc$i_op_v_ge,       {vector numeric compare greater_or_equal
    cvc$i_op_v_gt,       {vector numeric compare greater_than
    cvc$i_op_v_ifunc,    {vector intrinsic function call
    cvc$i_op_v_ifunc_r,  {vector intrinsic reduction function call
    cvc$i_op_v_le,       {vector numeric compare less_or_equal
    cvc$i_op_v_lt,       {vector numeric compare less_than
    cvc$i_op_v_mul,      {vector numeric multiply
    cvc$i_op_v_nand,     {vector logical nand
    cvc$i_op_v_ne,       {vector numeric compare not_equal
    cvc$i_op_v_nor,      {vector logical nor
    cvc$i_op_v_not,      {vector logical not
    cvc$i_op_v_or,       {vector logical or
    cvc$i_op_v_orn,      {vector logical or_not
    cvc$i_op_v_ranf,     {vector ranf intrinsic function call
    cvc$i_op_v_shfc,     {vector circular shift
    cvc$i_op_v_shfe,     {vector end_off shift
    cvc$i_op_v_sub,      {vector numeric subtract
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
    cvc$i_op_v_xor,       {vector logical xor

{ Instructions available in Interface, without result (aka Non-Hashed):

    cvc$i_op_br_eq,       {branch on numeric equal

    cvc$i_op_br_f,        {branch on logical false

    cvc$i_op_br_ge,       {branch on numeric greater_or_equal

    cvc$i_op_br_gt,       {branch on numeric greater_than

    cvc$i_op_br_i,        {branch indirect

    cvc$i_op_br_le,       {branch on numeric less_or_equal

    cvc$i_op_br_lt,       {branch on numeric less_than

    cvc$i_op_br_ne,       {branch on numeric not_equal

    cvc$i_op_br_t,        {branch on logical true

    cvc$i_op_br_u,        {branch unconditionally

    cvc$i_op_b_add,       {BDP decimal add

    cvc$i_op_b_bitmove,   {bit string move

    cvc$i_op_b_conv,      {BDP decimal type conversion

    cvc$i_op_b_div,       {BDP decimal divide

    cvc$i_op_b_edit,      {BDP string edit

    cvc$i_op_b_move,      {BDP string move

    cvc$i_op_b_move_bc,   {BDP string move of broadcast constant

    cvc$i_op_b_mul,       {BDP decimal multiply

    cvc$i_op_b_shfc,      {BDP decimal shift end_off
{ **to be renamed cvc$i_op_b_shfe**

    cvc$i_op_b_shfc_r,    {BDP decimal shift end_off rounded
{ **to be renamed cvc$i_op_b_shfe_r*

    cvc$i_op_b_sub,       {BDP decimal subtract

    cvc$i_op_b_tran,      {BDP string translation
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
    cvc$i_op_deref,        {pointer dereference
    cvc$i_op_hw_spec,      {hardware specific instruction
    cvc$i_op_jumptbl,      {create jump table
    cvc$i_op_labelref,     {create label reference
    cvc$i_op_procref,      {create procedure reference
    cvc$i_op_ptrmove,      {pointer move
    cvc$i_op_ptrref,       {create pointer reference
    cvc$i_op_push,         {push onto stack
    cvc$i_op_p_block,      {start of basic block
    cvc$i_op_p_blockend,   {end of basic block
    cvc$i_op_p_def,        { -unneeded?-
    cvc$i_op_p_init,       {start of initialization code
    cvc$i_op_p_initend,    {end of initialization code
    cvc$i_op_p_label,      {start of labelled code
    cvc$i_op_p_line,       {start of source line
    cvc$i_op_p_proc,       {start of procedure
    cvc$i_op_p_procend,    {end of procedure
    cvc$i_op_return,       {procedure return
    cvc$i_op_s_l_to_s,     {scalar move long to short, truncated on left
    cvc$i_op_s_move,       {scalar move
    cvc$i_op_s_paren,      {scalar parenthesization
    cvc$i_op_s_s_to_l,     {scalar move short signed to long, sign extended
    cvc$i_op_s_u_to_l,     {scalar move short unsigned to long, zero extended
    cvc$i_op_v_gthr,       {vector gather, fixed interval
```

```
    cvc$i_op_v_gthr_b,   {vector gather blocks, fixed interval

    cvc$i_op_v_gthr_i,   {vector gather according to index vector

    cvc$i_op_v_l_to_s,   {vector move long to short, truncated on left

    cvc$i_op_v_move,     {vector move

    cvc$i_op_v_paren,    {vector parenthesization

    cvc$i_op_v_sctr,     {vector scatter, fixed interval

    cvc$i_op_v_sctr_b,   {vector scatter blocks, fixed interval

    cvc$i_op_v_sctr_i,   {vector scatter according to index vector

    cvc$i_op_v_u_to_l,   {vector move short unsigned to long, zero filled

    cvc$i_op_v_v_to_l,   {vector move short signed to long, sign extended
{ **to be renamed cvc$i_op_v_s_to_l**

{ Instructions internal to Code Generator, with result (aka Hashed Result):

    cvc$i_op_b_dcmp_c,   {BDP decimal compare with constant

    cvc$i_op_extb,       {bit string extraction

    cvc$i_op_extb_c,     {bit string extraction according to constant

    cvc$i_op_insb,       {bit string insertion

    cvc$i_op_insb_c,     {bit string insertion according to constant

    cvc$i_op_load_h,     {load with hashed result

    cvc$i_op_mark,       {mark logical value

    cvc$i_op_mskb,       {bit string mask creation

    cvc$i_op_mskb_c,     {bit string mask creation according to constant

    cvc$i_op_ptradd_c,   {add integer constant to pointer value

    cvc$i_op_p_array,    {describe array reference

    cvc$i_op_p_bdescr,   {describe BDP descriptor

    cvc$i_op_p_callinfo, {describe additional call information
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
    cvc$i_op_p_cpypair,  {describe register pair
    cvc$i_op_p_memref,   {describe memory reference
    cvc$i_op_p_string,   {describe string reference
    cvc$i_op_s_abs,      {scalar numeric absolute value
    cvc$i_op_s_add_c,    {scalar numeric add of a constant
    cvc$i_op_s_add_z,    {scalar numeric add of a special constant zero
    cvc$i_op_s_mul_c,    {scalar numeric multiply of a constant
    cvc$i_op_s_shfc_c,   {scalar shift circular by a constant
    cvc$i_op_s_shfe_c,   {scalar shift end_off by a constant
    cvc$i_op_s_xfer,     {scalar register transfer instruction
    cvc$i_op_s_xfer_c,   {scalar register and constant transfer instruction
    cvc$i_op_s_xmit,     {scalar register transmit instruction
    cvc$i_op_v_xfer_r,   {vector reduction transfer instruction
    cvc$i_op_v_xmit_r,   {vector reduction transmit instruction
{ Instructions internal to Code Generator, without result (aka Non-Hashed):
    cvc$i_op_b_add_c,    {BDP decimal add of a constant
    cvc$i_op_b_conv_c,   {BDP decimal type conversion of a constant
    cvc$i_op_b_lcmp_c,   {BDP lexical compare with a constant
    cvc$i_op_b_move_c,   {BDP string move of a constant
    cvc$i_op_load,       {load register
    cvc$i_op_p_cpylwr,   {describe reference to second register of a pair
    cvc$i_op_p_cpyuor,   {describe reference to first register of a pair
    cvc$i_op_p_mod,      {start of compilation module
    cvc$i_op_p_modend,   {end of compilation module
    cvc$i_op_p_solid,    {start of solid optimization block
```

C4.0 CVT$I_CODE_GENERATOR_OPCODE

```
    cvc$i_op_p_solidend, {end of solid optimization block
    cvc$i_op_set_call,   {set up for procedure call
    cvc$i_op_store,      {store register
    cvc$i_op_s_move_c,   {scalar move of a constant
    cvc$i_op_v_abs,      {vector numeric absolute value
    cvc$i_op_v_add_z,    {vector numeric add of a special constant zero
    cvc$i_op_v_xfer,     {vector transfer
    cvc$i_op_v_xmit      {vector transmit
      );
?? FMT (FORMAT := ON) ??
```

Table of Contents