TITAN

PROGRAMMER'S GUIDE

Ardent Computer Corporation

Part Number: 340-0015-03

Change History

340-0015-02 Original
340-0015-03 February, 1989 – Software Release 2.0

CONTENTS

4 Using Libraries and the Link Editor

5 Methods For Debugging Code

6 Using The Debugger

7 dbg Language Syntax

8 Tuning and Porting Code

A Compiler Assembly Interface

EFFICIENT PROGRAMMING TECHNIQUES

CHAPTER TWO

The TITAN compilers automatically optimize your programs to run on TITAN's unique vector and parallel hardware. TITAN's compilers do these things well by themselves:

- They look at all loops in a nest for vectorization and parallelization opportunities.
- They select the best loops for vectorization based on data access to memory.
- They select the best loops for parallelization based on outermost position.
- They run sophisticated dependence tests to determine accurately the data usage order in arrays.
- They replace induction variables with constant values where possible to aid in vectorization.
- They accurately analyze equivalenced variables to permit vectorization.

You need do nothing more than write standard code to have your programs run in a quick and efficient manner on TITAN. If you have been optimizing your code to run on a scalar machine, you may need to undo some of your optimizations; they may actually hinder TITAN's compilers and slow computation.

The first section of this chapter explains programming techniques that take the best advantage of the automatic optimizations of TITAN's Fortran and C compilers. The second section explains the programming techniques of interfacing Fortran and C code. All examples in this chapter are illustrated using the Fortran language. Differences between Fortran and C are noted.

2.1 Structuring Loops

TITAN's Fortran compiler optimizes code by optimizing loop constructs. The more computation you can code in structured loops, the more efficient the compiler can make your programs run.

The compiler operates more efficiently on iterative loops than on **DO WHILE** constructs. The compiler only considers iterative loops as candidates for vectorization. It does contain a pass to convert **WHILE** loops to iterative loops. Many **WHILE** loops are written in such a way to make it difficult for the compiler to convert. If you rewrite such loops explicitly as iterative loops, you achieve better performance.

To illustrate, consider storing numbers in an array until the array is full.

```
      DO 70 WHILE (I .LE. MY_SIZE)
         ARRAY(I) = A(I)*X + B(I)
70  CONTINUE
```

The compiler, where it can, converts the previous example to the example below.

```
      DO 70 I = 1, MY_SIZE
         ARRAY(I) = A(I)*X + B(I)
70  CONTINUE
```

Although the compiler is capable of converting **DO WHILE** statements beware of limitations. The compiler cannot convert the following example.

```
DO WHILE (I .NE. 0)
        B(I) = B(I) + 1
        I = NEXT(I)
END DO
```

When in doubt about whether a particular **DO WHILE** loop can be converted, compile your program with the *-vreport* compiler option.

2.1.1 IF-THEN-ELSE Statements

The compiler is much less effective on loops that contain random **GO TO**'s than it is on codes containing structured **IF-THEN-ELSE** statements. Random jumps defeat the purpose of the optimizing hardware.

Convert conditional branches to structured **IF-THEN-ELSE** statements wherever possible. Consider the following calculation.

```
      IF (N .GT. 50) GO TO 95
         M = 10
         A(N) = B(M*N)
      GO TO 98
95    M = 5
      A(N) = B(M*N)
98    CONTINUE
```

The compiler parallelizes but does not vectorize **GO TO's**. The use of **GO TO's** is not recommended. The following code achieves the same result and runs on the vector unit.

```
IF (N .LE. 50) THEN
     M = 10
     A(N) = B(M*N)
ELSE
     M = 5
     A(N) = B(M*N)
ENDIF
```

Jumps that exit loops inhibit parallelization entirely. The same principle of avoiding random jumps applies to most error checking procedures. Consider the following example.

```
      DO 80 I = 1,100
         IF (A(I) .LE. EPSILON) GO TO 81
         A(I) = A(I) + B(I)*C(I)
80    CONTINUE
      RETURN
81    CONTINUE
      CALL ERROR()
```

This loop does not vectorize because of the **GO TO** statement. Recoding should be done so that the error checking is not in the same loop as the computation. Splitting off the error checking into a separate loop creates at least one vectorizable loop.

```
DO I = 1, 100
     IF (A(I) .LE. EPSILON) CALL ERROR()
END DO
DO I = 1, 100
     A(I) = A(I) + B(I)*C(I)
END DO
RETURN
```

2.1.2 Procedure and Function Calls

Procedure calls and function calls within loops prohibit optimization when the calls constitute random jumps in the program. However, the compiler attempts optimization if it can copy the procedure or function into the statement line, a technique called *in-lining*. When writing your own functions and procedures, use iterative loops where feasible so that the procedures themselves are vectorizable when inlined. Then use compiler directives to request in-lining and procedure vectorization.

2.1.3 Loop Unrolling

Loop unrolling is a process used to simulate vector processing on some scalar machines. It allows the scalar machine to start an operation before another has finished. Loop unrolling duplicates sections of code within a loop so that each section operates with a different increment of the loop variable. You may have unrolled loops to enhance performance on scalar machines. TITAN's compiler usually vectorizes loops whether they are unrolled or not, but re-rolled loops execute a little faster on TITAN hardware. The compiler automatically unrolls a loop if unrolling is necessary for optimization.

In the following examples, the first **DO** routine executes faster than the **DO** routine that increments in steps of 5.

```
DO I = 1, N
        X(I) = X(I) + Y(I)
END DO

DO I = 1, N, 5
        X(I) = X(I) + Y(I)
        X(I + 1) = X(I + 1) + Y(I + 1)
        .
        .
        .
        X(I + 4) = X(I + 4) + Y(I + 4)
END DO
```

2.1.4 Fortran Intrinsics

Use intrinsic Fortran functions such as **MAX** and **MIN** where possible instead of writing your own. The intrinsic functions have been coded to use the most efficient machine instructions available and to take the best advantage of TITAN's special reduction hardware.

Instead of coding

```
IF (A(I) .LT. T)  THEN
        T = A(I)
```

use a Fortran intrinsic function, such as

```
T = MIN(T, A(I))
```

2.2 Programming for Memory Access

You may be accustomed to optimizing your code by saving time used in calculation. For example, some programs save multiplication time by testing for and branching around elements that multiply by zero.

```
IF (A(I) .NE. 0) THEN B(I) = B(I) + A(I)*C(I)
```

TITAN speeds up calculations by using its vectorization and parallelization abilities, not by skipping operations; the compiler cannot vectorize efficiently operations which require jumps in memory. It is more efficient to let the vectorizer operate on all elements of an array than to check a condition before each calculation.

2.2.1 Array Density

In supercomputing, the density of an array refers to the number of elements on which a given operation is performed and not to the number of elements not equal to zero. Operating on *dense* arrays means operating on most of the elements of an array, and operating on *sparse* arrays means operating on only a few elements. The same array can be both dense and sparse for different operations.

Optimizations differ for dense and sparse arrays. Optimizing operations on sparse arrays uses the scatter/gather capability of the hardware. Optimizing operations on dense arrays means vectorizing the array.

In the following code example the compiler assumes use of most of the array elements. The loop performs a divide under the condition that a given element is non-zero.

```
IF (A(I) .NE. 0)
        B(I) = B(I) / A(I)
```

There are two modes of operating on a vector and the mode chosen should depend on the percentage of the vector being worked on. The previous example illustrates a way to code when most of the array elements are used. The following example specifically tells the compiler you are only working on a few elements in the array and are packing and unpacking the array yourself.

```
      J = 1
C     COPY THOSE ELEMENTS TO TOTEMPORARY ARRAY.
      DO I = 1, N
       IF (B(I) .NE. 0)  THEN
C T(J) IS THE TEMPORARY ARRAY.
         T(J) = B(I)
C S(J) REMEMBERS THE ORIGINAL POSITIONS OF THE T ELEMENTS.
         S(J) = I
         J = J +1
       ENDIF
      END DO
C OPERATE ON THE NON-ZERO ELEMENTS.
      DO I = 1, J-1
       T(I) = T(I) + FUNCTION(T(I))
      END DO
C PUT THE RESULTS IN THE ORIGINAL ARRAY.
      DO I = 1, J-1
       B(S(I)) = T(I)
      END DO
```

2.2.2 Data Storage

Fortran stores the elements of a multi-dimensional array by columns, a procedure called *column-major format*. That is, the leftmost subscript varies fastest as elements are accessed in storage order. Then the 3 x 3 matrix

$$\begin{matrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{matrix}$$

appears in memory with each column's elements stored together.

$$\begin{aligned} \text{memory}(1) &= A_{1,1} \\ \text{memory}(2) &= A_{2,1} \\ \text{memory}(3) &= A_{3,1} \\ \text{memory}(4) &= A_{1,2} \\ \text{memory}(5) &= A_{2,2} \\ \text{memory}(6) &= A_{3,2} \\ \text{memory}(7) &= A_{1,3} \\ \text{memory}(8) &= A_{2,3} \\ \text{memory}(9) &= A_{3,3} \end{aligned}$$

Suppose your program includes an array of **Pressure, Volume,** and **Temperature** for each of 100 grid points. You could arrange the data in one array as **PVT(3,100)** with each column representing the values for a single grid point. Or you could arrange the data in the array **PVT(100,3)**. Then calculations of **Pressure,** for example, involve elements in a single column, and the values of these elements are in adjacent locations in memory. The hardware operates more efficiently on long contiguous vectors.

2.2.3 Temporary Variables

If you use a temporary variable in a loop, try not to reference the temporary variable outside of the loop. When you reference a temporary variable outside of its loop, the compiler may identify that variable as permanent and hence may assume that different parts of the loop cannot be assigned to different processors. Using temporary variables outside of their loops prevents the possible parallelization of a loop; instead, use unique names for each loop's temporary variables.

2.2.4 COMMON and EQUIVALENCE

Using **COMMON** and **EQUIVALENCE** statements can also hinder optimization by creating misalignment and aliasing errors of data. The following declarations of the variables **A, B,** and **C** illustrate the problem.

```
DIMENSION A(2,2)
INTEGER*4 B
DOUBLE C
COMMON A, B
EQUIVALENCE B, C
```

The example statements above force the compiler to share the memory space between **B** and **C**. But **B** is an integer and **C** is a double precision floating point number. The compiler handles this deliberate misalignment with a software trap for every reference to **C**; it cannot be handled directly in the hardware.

Similarly, the shared memory between **A** and **B** means that every reference to the integer **B** can be a reference to an element of the array **A**. This aliasing conflict means that the compiler may not vectorize operations involving **A** or **B**.

Use **COMMON** and **EQUIVALENCE** statements with great care.

2.3 Compiler Directives

Restructuring compilers determine at compile-time the optimal run-time use of parallel and vector hardware. The compiler cannot have full knowledge at compile-time of important run-time information, so in some cases it may be desireable to supply compiler directives which override the compiler's natural caution. Use the compiler option *-vreport* to help you decide if you need to supply a compiler directive. Directives influence all phases of compilation and are essential for optimal use of vector/parallel hardware.

Through compiler directives, you help the compiler to generate more efficient code by

- Forcing vectorization or parallelization of a particular loop.
- Indicating the best loops to vectorize or parallelize.
- Requesting that a loop be executed exactly as written.
- Requesting that a procedure be in-lined within a loop.
- Declaring a procedure that can take vectors as arguments.

The following four points summarize the information supplied by the directives to the compiler.

- Symbolic array references. The compiler cannot always accurately compute dependencies. You can provide an **IVDEP** or **IPDEP** directive to help the compiler out.
- Best loop selection. The compiler cannot always know loop lengths at compile-time. You can provide **PBEST** or **VBEST** directives to help the compiler in its selection of the best loop.
- Stability or time considerations. A vectorizer may spend a lot of time analyzing a loop which you do not want vectorized for stability reasons. It may also spend a lot of time analyzing a loop which can not be vectorized for dependency reasons. You can provide an **ASIS** directive to make the compiler bypass the loop.

- Procedure calls. Procedure calls hide information from the compiler and prevent it from vectorizing or parallelizing a loop. You can provide vector calculation routines and the **VPROC** directive or ask that procedures be in-lined (**INLINE**) to help loops containing procedure calls.

The following sections discuss the formats of the TITAN Fortran and C compiler directives. All examples and syntax are expressed in Fortran notation, however it is important to note the syntax for the directives for C programs. The syntax for the directives for use in C programs is to replace the **C$DOIT** before the directive with **#pragma**. As an example, the **ASIS** directive becomes

#pragma ASIS

for C programs.

2.3.1 ASIS

The **ASIS** directive tells the compiler not to change or to optimize the following loop for vector hardware. The vectorizer leaves an **ASIS** loop alone. Use this directive for loops with very sensitive numeric properties. The format of the **ASIS** directive is

SYNTAX

C$DOIT ASIS

To illustrate using the **ASIS** directive, consider summing a vector that contains very large and very small values alternated to avoid overflow. The sum reduction hardware in the TITAN very nearly preserves the associativity of original operations, but not quite. Hence, summing this vector on the reduction hardware may change the results. To prevent the TITAN vectorizer from using the reduction hardware, place an **ASIS** directive around the loop.

```
C$DOIT ASIS
      DO 160 I = 1, N
        T = T + A(I)
  160 CONTINUE
```

2.3.2 INLINE

The **INLINE** directive tells the compiler to in-line the named functions in the following loops, if possible. Its format is

SYNTAX

C$DOIT INLINE *function1, function2, ...*

Use the **INLINE** directive to remove function calls from important loops, thereby permitting the loops to be vectorized. This directive must appear at the place where you want to inline the function, not in the function to be inlined..

2.3.3 IVDEP

Upon encountering the **IVDEP** directive, the TITAN compiler ignores any array dependencies. Any defined scalars are either in a straightforward recurrence or need to be expanded into temporaries and the compiler decides which. The format of the **IVDEP** directive is

SYNTAX

C$DOIT IVDEP

Note that the **IVDEP** directive does not guarantee vector execution. The TITAN compiler applies the directive only to array dependences and assumes that the array correctly calculates scalar dependences. Thus, scalars may inhibit vector execution even when an **IVDEP** directive is present.

In many cases, a loop can be vectorized even though it appears to the compiler that the loop cannot be vectorized. Consider this common loop in relaxation codes.

```
      DO 130 I = 1, N
        DO 131 J = 1, N
          A(I,J) = (A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))/4.0
131   CONTINUE
130 CONTINUE
```

Even though this loop computes slightly different results if executed in the vector unit, the differences are negligible in these loops' contexts. Hence, you want to force vector execution of the J loop for speed. Use the **IVDEP** directive to tell the compiler to ignore dependences that appear to inhibit vectorization.

```
      DO 130 I = 1, N
C$DOIT IVDEP
        DO 131 J = 1, N
          A(I,J) = (A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))/4.0
  131   CONTINUE
  130 CONTINUE
```

2.3.4 IPDEP

The **IPDEP** directive is the analog to the **IVDEP** directive and tells the compiler to ignore the dependences that appear to inhibit parallelization in the following loop. Any defined scalars are assumed to be temporaries local to each processor. The format of the **IPDEP** directive is

SYNTAX

C$DOIT IPDEP

In the **IVDEP** example, the outer loop is the best candidate for parallelization. Insert an **IPDEP** directive in addition to the **IVDEP** directive as shown below.

```
C$DOIT IPDEP;
      DO 130 I = 1, N
C$DOIT IVDEP
        DO 131 J = 1, N
          A(I,J) = (A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))/4.0
  131   CONTINUE
  130 CONTINUE
```

2.3.5 VBEST

The **VBEST** directive indicates the best loop to vectorize **if the loop is vectorizable.** It does **not** mean that the loop can be vectorized. The format of **VBEST** is

SYNTAX

C$DOIT VBEST

Because the compiler does not always know loop lengths at compile-time, it may vectorize a less-than-optimal loop. When presented with the following code and no knowledge of *M* and *N*,

```
      DO 140 I = 1, M
        DO 141 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
141   CONTINUE
140 CONTINUE
```

the compiler assumes the the **J** loop is the best loop to vectorize because it accesses locations which are adjacent to each other in memory. If, however, **N** is equal to 3 and **M** is equal to 1000, this choice would produce poor execution because the vector loop is too short to make profitable use of the vector unit. In this case, insert a **VBEST** directive, telling the compiler to vectorize the **I** loop.

```
C$DOIT VBEST
      DO 140 I = 1, M
        DO 141 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
  141   CONTINUE
  140 CONTINUE
```

Now the compiler vectorizes the outer loop (if possible) rather than the inner loop, resulting in more effective operation.

2.3.6 PBEST

PBEST is similar to VBEST; it indicates the best loop to parallelize rather than to vectorize. Its format is

SYNTAX

```
C$DOIT PBEST
```

Normally, the TITAN compiler parallelizes the outermost loop long enough to justify parallel execution. Consider the VBEST example.

```
      DO 150 I = 1, M
        DO 151 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
151   CONTINUE
150 CONTINUE
```

With no knowledge of M or N, the compiler parallelizes the outer (I) loop. This is a poor choice if M is only 2 or 3 and N is 1000 and you are running on a four processor system. In this case, use a PBEST directive to signal parallelization of the inner loop.

```
      DO 150 I = 1, M
C$DOIT PBEST
        DO 151 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
151   CONTINUE
150 CONTINUE
```

With the PBEST directive, the compiler parallelizes the 151 loop and moves it to the outermost position. This now becomes

```
DO PARALLEL
        J = 1, N
   DO I = 1, M
        A(J,I) = B(J,I) + C(J,I)
```

To summarize, the VBEST/PBEST directives tell the compiler which of several parts of a nested loop is best to parallelize or vectorize.

The compiler optimizes loops in two phases. First, it chooses the best loop to vectorize. The best loop to vectorize is the loop with the smallest stride and the longest length. If the loop bound is read in at runtime, the compiler must assume a loop length long enough to make vectorization profitable.

Next, the compiler chooses the best loop to parallelize. The best loop to parallelize is the loop with the longest length. Again, if the loop bound is read in at runtime, the compiler must assume a loop length long enough to make parallelization worthwhile.

Use **VBEST/PBEST** directives to keep the compiler from assuming that the longest lengths are those of loops whose bounds are read at runtime.

2.3.7 VPROC

The **VPROC** directive tells the compiler that the named function has an assembly-language version that can take vector arguments. Upon encountering the **VPROC** directive, the compiler assumes that the named function has no side effects and attempts to vectorize any loop containing a call to that function. If vectorizable, the compiler replaces the function with the vector function name *vfname*.

SYNTAX

C$DOIT VPROC *fname, vfname*

If no *vfname* is provided, the compiler calls *fname* with vector arguments. Note that using **VPROC** this way is dangerous because you assume that the compiler vectorizes all loops containing this function.

2.3.8 VREPORT

The **VREPORT** directive invokes the vector reporting facility on the next loop and is used to tell the user what vectorization has been done to the program. **VREPORT** provides a detailed listing of exactly what the compiler did to each loop nest.

SYNTAX

C$DOIT VREPORT

The **VREPORT** directive can only be turned on for one loop at a time and only at the outermost loop level.

2.3.9 NO_PARALLEL

The **NO_PARALLEL** directive tells the compiler not to run the next loop in parallel.

SYNTAX

C$DOIT NO_PARALLEL

The following is an example of when you might want to use this directive.

```
        SUBROUTINE  FUZZY(A,M,N)
        DOUBLE PRECISION A(N)
C$DOIT IVDEP
C$DOIT NO_PARALLEL
        DO I = 1, N
          A(I) = A(I) + A(M)
        ENDDO
        END
```

2.3.10 OPT_LEVEL

The **OPT_LEVEL** compiler directive allows you to set the optimization level for a specific procedure within a file. The syntax is

SYNTAX

C$DOIT OPT_LEVEL *n*

where is the number 0, 1, 2, or 3. This is the optimization level.

The syntax for using this directive in C the same as Fortran except to replace C$DOIT with #pragma.

This directive should be placed immediately before the first line of the procedure declaration, and overrides any command line options. At the end of the procedure, the optimization reverts to what it was previously.

2.3.11 SCALAR

The **SCALAR** directive tells the compiler to run the next loop in scalar.

SYNTAX

C$DOIT SCALAR

This directive has the effect of not allowing vectorization or parallelization on the loop.

2.3.12 Cray Directives

Directives for the Cray compiler have existed for roughly a decade. Because their functionality is very similar to TITAN's directives, the TITAN compilers accept Cray directives on a switched on or off mode, as in the Cray compiler. The following three directives have been added to the TITAN compilers for compatibility with Cray directives.

2.3.12.1 IVDEP

The IVDEP directive is a Cray directive specifying that the compiler should ignore vector dependences. This directive does not guarantee vector execution. Its format is

SYNTAX

CDIR$ IVDEP

This directive is provided solely for compatibility with Cray.

2.3.12.2 NORECURRENCE

The NORECURRENCE directive is a Cray directive inhibiting generation of vector recurrence code for loops that are below a given size. Similar to the NOVECTOR node, its format is

SYNTAX

CDIR$ NORECURRENCE = N

This directive is provided only for compatibility with Cray. If you are coding for the TITAN, VBEST and ASIS are better directives to use.

2.3.12.3 NOVECTOR

The NOVECTOR directive is a Cray directive inhibiting vectorization of loops below a given size. Its format is

SYNTAX

CDIR$ NOVECTOR = N

This directive is provided for compatibility with Cray directives only. If you are coding for the TITAN, VBEST and ASIS are better alternative directives.

2.3.12.4 VECTOR

The **VECTOR** directive is a Cray directive toggling **NORECURRENCE** and **NOVECTOR** between default and specified values. Its format is

SYNTAX

```
CDIR$ VECTOR
```

This directive is provided solely for compatibility with Cray.

2.4 Summary

The following briefly summarizes the programming techniques that take the best advantage of the automatic optimizations of TITAN's Fortran compiler.

- Use iterative loops wherever possible.
  - Use **DO** loops, **DO WHILE** statements, and **IF-THEN-ELSE** statements. Convert branches into structured **IF-THEN-ELSE** statements instead of using **GO TO** statements.
  - Make nested loops into larger single loops. Put as much calculation inside loops as possible to minimize synchronization.
  - Do not unroll loops to enhance performance.
  - Use language intrinsics, like **MAX** and **MIN**, instead of writing code to test-and-branch.
- Program for memory access and not to save memory.
  - Do not program to avoid computing null elements in sparse arrays.
  - Do not reference temporary variables outside of the loops in which they are used.
  - Specify array dimensions as longest dimension first to shortest dimension last in Fortran and the opposite in C.
  - Use **COMMON** and **EQUIVALENCE** statements only when absolutely necessary.
- Use compiler directives to handle loops with hidden bounds.

2.5 Interfacing Fortran and C

This section specifies the items you must consider in attempting to compile and link Fortran and C modules into a single executable program. The considerations include case-sensitivity, parameter passing conventions, and retention of parameter values between calls.

2.6 Case-Sensitivity

C allows mixed upper and lower case function names and parameter names. The case of the letters is significant so that the name **MyFunction** is not the same as **myfunction**. However, in Fortran, all function names and parameter names are folded into upper case. That is, all lower case letters become upper case as far as Fortran is concerned. Although you are permitted to write in mixed upper and lower case for your programs, the compiler translates lower case to upper case when your program is compiled.

To create a C language function that is callable from Fortran, you must write the function name in all upper case letters. The loader distinguishes between upper and lower case. Thus, unless your C function is written in upper case letters, the name can not match the name that Fortran passes to the loader.

2.7 Parameter Passing

The parameter passing conventions employed by Fortran and C have the following differences.

- In Fortran, when you specify a parameter name in a subroutine call, the parameter is passed by **reference**. That is, the address of the parameter is passed to the subroutine. Likewise, when a Fortran subroutine is called, it expects to receive its parameters by reference, rather than by value. Fortran returns its results by reference rather than by value. It is possible to force a Fortran call to **pass** a parameter by value by using the **%VAL** built-in function, but it is **not** possible to force a Fortran subprogram to receive arguments passed by value.

- In C, when you specify a parameter name in a subroutine call, the parameter is passed by **value**. By appropriate declarations, it is possible in C to specify that a parameter is to be passed by reference.

2.8 Fortran Calling Sequences

If you want to create a Fortran program that calls a C routine, you have two ways you can write the calling sequence, depending on how you have written your C function.

2.8.1 Passing To C By Reference

You can write a C function to accept a parameter by reference, as in this example.

EXAMPLE

```
void  MYFUNC (int* X);

                          /* X is passed by
                          reference to MYFUNC*/
{
                          /* Call a C procedure which
                          expects a value parameter*/
Value_Proc(*X);
                          /* The * de-references
                          X to make a value*/

                          /* Call a C procedure which
                          expects a reference parameter*/
Ref_Proc(X);
                          /* X is already an
                          address - pass it on*/
return *X + 1;
                          /* Return an integer
                          value - X+1*/
}
```

The corresponding Fortran call can be made as

```
D = MYFUNC(M)
```

A function or subroutine passed as an argument is also passed by reference in Fortran. Assume that you wish to write a C routine that integrates a real-valued argument function between the bounds of zero and one and returns the function value as a real result. A piece of Fortran to use the integration routine might look like the following example.

```
REAL INTEGRATE

REAL PolyFunc

EXTERNAL PolyFunc          ! Tell Fortran this is a function
Value = INTEGRATE(PolyFunc)
```

The definition of the C routine might be something like this

```
typedef  float (*integral)(float*);

float  INTEGRATE (integral I)
{
                          /* Code to do the integration */
        ...
        Y = (*I)(&X);     /* Evaluate at one point */
                ...
 return Sum;
}
```

2.8.1.1 Passing String Arguments To C

Fortran strings are more complicated than other Fortran data types; the compiler maintains a length for each string as well as the string data storage. Normally this length is invisible, but when a Fortran string is used as an argument, the length is normally passed to the called routine so that it can know how long the string is. When calls are made from one Fortran routine to another, passage of the length is of no concern to the Fortran programmer because the compiler takes care of it. However, when a C program is called, the programmer must manage this detail explicitly.

There are two methods of passing a Fortran string to C. In the first method, the passing of the length is defeated by the use of the the built-in function **%REF**. A typical call might look like those in the examples that follow.

EXAMPLES

```
CHARACTER*7  RESULT

CALL C_PRINT_IT(%REF(RESULT))

CALL C_PRINT_IT(%REF('Once again' // RESULT))
```

The effect of **%REF** when applied to a Fortran string argument is to cause the address of the first character of the string to be passed to the called routine. The C procedure definition would have the form that follows.

```
void  C_PRINT_IT(char* Input)
{
                          /* Use the characters of Input
                          like any C string variable */
return;
}
```

Notice that Fortran strings are not null-terminated. When using this method to pass strings, it is the programmer's responsibility to develop a convention for the termination of the string. Some possible conventions are to assume a string length, to pass the string length explicitly in a separate argument, or to have the Fortran program explicitly store a null character somewhere in the string.

The second method requires the C procedure to understand the mechanism by which Fortran subprograms communicate the string lengths among themselves. The Fortran statement

```
CALL F_PRINT_IT(RESULT)
```

passes a pointer to a two word data structure to the routine **F_PRINT_IT**; the structure contains the address of **RESULT** and its length. Of course, if **F_PRINT_IT** is written in Fortran, the compiler takes care of the details. If it is written in C, the definition would look something like the example that follows.

```
typedef  struct { char* Addr;
                  int  Length;
                } Str_Desc;

void  F_PRINT_IT(Str_Desc* SD)
{
                                  /* Don't change the  */
                                 /* parameter passed in */

        char* My_Addr    = SD->Addr;
        int     My_Length = SD->Length;

        for (i=1; i<- My_Length; i++, My_Addr++)

                                 /* Do whatever needed  */

        return;
}
```

2.8.1.2 Passing A Hollerith Constant To C

Hollerith constants are treated differently than strings. Hollerith constants are typeless. In the following example, when you specify the Hollerith constant, it is laid out in a Fortran constant

storage area. What your C program receives is the address of the first character. The string length is not available unless you explicitly encode it as a parameter that you pass to your routine.

EXAMPLE

```
RESULT = Herman(18HA Hollerith string,3,6)
```

In the above example, your C program gets the address of the first character of the string, the address of the constant 3, and the address of the constant 6.

2.8.2 Passing To C By Value

Fortran can pass non-string arguments to C routines by value. The **%VAL** built-in function is used; look at the following example.

EXAMPLE

```
REAL X, Y
COMPLEX C

CALL C_Proc(%VAL(X), Y, %VAL(C))
```

The corresponding C routine would be written as follows.

```
typedef  struct { float  Real;
                  float  Imag;
                } Complex;

void  C_Proc (float A, float* B, Complex Z)
{
                        /* Notice that A and Z
                         receive values, but that
                         B receives a pointer to a float  */
return;
}
```

2.8.3 Passing to Fortran by Reference

If you have created a C program that calls a Fortran subroutine, simply pass the arguments by address, usually using the C & operator.

2.8.4 Passing To Fortran By Value

You cannot pass parameters to Fortran by value. It is only possible for **%VAL** to operate on actual parameters.

2.9 Retaining Variable Values Between Calls

To allow C variables to retain their value between calls, (just as in TITAN Fortran), declare the variables to be of a static type.

EXAMPLE

```
int C_FUNC(int* x)
 {
      static int local_val;

      local_val = x*17 + 2;

      return x*x;
 }
```

Variable **local _val** retains its value between calls because it is a static variable.

2.10 Setting Up Fortran I/O

When the Fortran compiler is used to load an executable program, it automatically includes some code that initializes and terminates the Fortran I/O library. However, if you build an executable program with the C compiler or the **ld** command, you do not normally include this setup code. If a Fortran routine attempts to execute Fortran I/O statements and the I/O library has not been set up, unpredictable effects may occur which may be hard to diagnose and debug.

If you plan to mix Fortran and C in the same program, you should have a Fortran main program and use the Fortran compiler to load your program. If this is not possible, insure that any Fortran routines you include in your executable program do not perform any I/O (be on the alert for error message handlers). Finally, if you must use Fortran I/O and you must load with the C compiler or **ld**, make sure to include the same libraries and calls as the Fortran compiler automatically provides.

2.11 Unformatted I/O In Fortran

There are two programming techniques available that can improve program performance when using unformatted I/O. The first consideration is the alignment of data and the second is using arrays versus **DO** loops.

2.11.1 Data Alignment

The file system block size is 4096 bytes, and if data can be arranged to stay on block boundaries there is a small speed improvement. When coding any I/O that requires the statement specifier **RECL** to be set in the **OPEN** statement, use multiples of 4096 bytes or 1024 REAL's, and so on. When using sequential unformatted I/O without specifying a record length, try to align the data using the formula 4096 * n - 8, where n is the number of 4K byte blocks you need.

If aligning data on block boundaries is not possible, try to at least avoid mixing odd size data elements with elements that are multiples of words. Moves are faster if they are word to word or double word to double word.

2.11.2 Arrays Versus DO Loops

In general it is faster to move data in large chunks as opposed to moving data elements one item at a time. It is better to use long arrays or vectors than to use implied **DO** loops. The Titan attempts to remove **DO** loops which can use block data moves, but this is not always possible. Any help that the programmer can provide in removing **DO** loops is useful.

USING THE TITAN COMPILERS

CHAPTER THREE

This chapter describes how to compile TITAN Fortran and TITAN C source programs into object modules that you can execute. This chapter

- Describes the functions of the compilers.
- Describes how to call the compilers.
- Describes compiler options.
- Describes the formats of the listings produced by the compilers.
- Shows how to locate and correct run time errors.

This chapter is divided into three sections. The first section contains a general discussion of the functions of the compilers, the options that are common to both languages, preprocessor control, and linking. The last two sections contain the details of calling the compilers, options to the compilers, and locating errors for Fortran and C, respectively.

3.1 The TITAN Compilation System

The TITAN compilers construct object files from source language files. The TITAN compilers execute under UNIX. The compilers generate binary object files; these files may be combined by the loader with one another and with system libraries to create executable programs. Ordinarily, specifications of these files are of little interest, but a simplified description of the contents is found in *Chapter 5, Methods for Debugging Code.*

3.1.1 Functions of the TITAN Compilers

The TITAN compilers perform the following functions

- They check that your program is correct and tell you where you have made errors.
- They translate your source program into machine language instructions which the TITAN can execute.
- They group these executable instructions into object modules which may be linked (by the loader) into a complete executable program.

In addition to the executable instructions, the compilers also generate lists of all subroutines and common blocks defined and used in each module. The loader combines object modules into complete programs, adding additional routines from user and system libraries as needed. During the loading process, a symbol defined in one routine is linked to all the other routines which use the symbol. The compilers and the loader also communicate to pass debugging information to the debugger.

3.1.2 Specifying the Output Files

You use compiler options to specify what kind of output is to be produced. For example, the *-S* option produces a machine code source list. Refer to the option lists for each language later in the chapter for more information about controlling the output.

3.2 Compilation Control

There are three ways to control the compilation of your program: with command line options, compilation control statements, and compiler directives. This section discusses only those options and statements that can be used from either Fortran or C.

3.2.1 Command Line Options

The following three tables list all possible options that can be used in both Fortran and C. Detailed descriptions and explanations of these options are discussed following the table. Defaults are indicated in italics. The negative form of the option only applies to Fortran. Options are case-sensitive; upper and lower case options have different meanings.

If the Fortran command line compiler option **-cpp** is not specified when coding in Fortran, most of the command line preprocessor options are ignored. The options **-P** and **–E** always invoke the preprocessor, however the others do not do this unless the **-cpp** option is specified.

Table 3-1. Command Line Preprocessor Options for Fortran and C

Option	Negative Form	Description
-D*name*		Define *name* to have a value of 1
-D*name=val*		Define *name* to have a value of *val*
-E		Output expanded source
-I		Do not search for included files in default directory
-I*dir*		Search for included files in *dir*
-i		Suppress production of #ident information
-P		Preprocess only and copy to *file.i*
-u*name*		Undefine *name*

Table 3-2. Command Line Compiler Options for Fortran and C

Option	Negative Form	Description
-c		Compile only; do not link
-catalog=*name*.in		Create a database of inlined functions
-full_report		Invoke vector reporting facility
-fullsubcheck		Add code to check each linear array subscript
-g	*-nodebug*	Add debug data to object file
-inline		Inline functions
-Npaths=*name*.in		Use inlined functions in *name*.in
-O0	-nooptimize	Turn off optimization
-O1		Do subexpression elimination
-O2		Do -O1 and vectorization
-O3		Do -O2 and parallelization
-O		Same as -O1
-o *filename*		Put output into filename
-ploop		Profile each loop separately
-S		Generate assembly language source file
-subcheck		Add code to check linear array subscripts
-V		Print version information
-vreport		Invoke vector reporting facility
-vsummary		Invoke vector reporting facility
-w		Suppress warning messages during compilation
-43		Use BSD capability

Table 3-3. Command Line Loader Options for Fortran and C

Option	Negative Form	Description
-B*hhhhhhhh*		*a.out* has bss address at *hhhhhhhh*
-D*hhhhhhhh*		*a.out* has data address at *hhhhhhhh*
-e*sym*		Set default entry point address
-L		Do not search for libraries in /lib or /usr/lib
-L*dir*		Search for libraries in *dir*
-l*tag*		Search library called *tag.a*
-m		Generate a simple load map
-n		Generate NMAGIC file type
-opct		Produce count of FPU ops
-p	*-noprofile*	Generate profiling code
-r		Produce a relocatable output file
-s		Strip line numbers and symbol table information
-T*hhhhhhhh*		*a.out* has text address at *hhhhhhhh*
-t		Turn off certain warnings
-y*name*		Trace *name*

3.2.1.1 Preprocessor Options

-D*name*

Define *name* to have the value of 1 to the preprocessor.

-D*name=val*

Define *name* to have the value of *val* to the preprocessor.

-E

Output expanded source to standard output. This option stops the compilation process after all macros and conditional compilation expressions have been expanded. A "real" compilation does not take place. Output of the preprocessor appears on standard output.

-I

Suppress the default searching for preprocessor included files in */usr/include*. This does not affect the INCLUDE statement.

-I*dir*

Search for preprocessor include files in *dir*. This does not affect the **INCLUDE** statement.

-i

Suppress the automatic production of #ident information.

-P

Preprocess only and copy *x.c* or *x.f* to *x.i*.

-u*name*

Undefine *name*.

3.2.1.2 Compiler Options

-c

Compile the source files, but do not invoke the loader. Generate only object code output.

-catalog=*filename.in*

Create a database of functions that are frequently inlined. Be aware that the use of **-catalog** creates two files; the database and a *filename.id*. This can use up a large amount of disk space, as the use of this option adds to an existing catalog.

To create a catalog in Fortran, specify the following on the command line:

```
fc  -catalog=file.in  filename
```

To create a catalog in C, you only need to specify *filename.in* on the command line, without using the **-catalog** option. The compiler assumes the creation of the catalog from this.

-full_report

This invokes the vector reporting facility and shows you how things are vectorized, what code the compiler generated, and explains why things have been done. This is the most detailed report that is generated by the vector reporting facility. Refer to the next main section in this chapter, titled *Vector Reporting Facility*, for examples and complete descriptions of the output produced by invoking any of the three options **-vsummary**, **-vreport**, or **-full_report**. This output is in Fortran-like notation.

-fullsubcheck

Generate code to check that every subscript in every array reference is within the bounds of the appropriate array dimensions. The check is more stringent than that of option *-subcheck*.

CAUTION
Use of this option increases object code size and decreases execution speed.

-g

Generate information for the TITAN debugger.

-inline

Instruct the compiler to inline functions. This works in conjunction with **-Npaths** (in Fortran) and compiler directives.

It is best to focus on small functions inside loops. Be careful about inlining intrinsics and then vectorizing. As a final caution, always look at **-vreport** when inlining to make sure that things are actually running faster.

The compiler does not inline some things and these are: character valued functions or functions taking character arguments, functions it thinks are varargs, and inlining into a condition of a while loop.

-Npaths=*filename.in*

Instruct the compiler to make use of the database of inlined functions (*filename.in*). Specifying **-Npaths=** avoids getting system functions.

-O0

Turn off all optimizations.

-O1

Perform common subexpression elimination and instruction scheduling.

-O2

Perform **-O1** and vectorization.

-O3

Perform **-O2** and parallelization.

-O

This is the same as **-O1**.

-o *filename*

Place the output into *filename*.

-ploop

Profile each loop in a single routine separately. Refer to *Chapter 8, Tuning and Porting Code* for detailed information on using this option. You may also refer to the *Commands Reference Manual.*

-S

Do not generate object files; instead, generate assembly files.

-subcheck

CAUTION
Use of this option increases object code size and decreases execution speed.

Produce code to check at runtime to ensure that each array element accessed is actually part of the appropriate array. Consider the fragment

```
REAL SAM(10,20,30)
...
...
... = SAM(I,J,K) ...
```

If **I** has any value between 1 and 10, **J** between 1 and 20, and **K** between 1 and 30, then the access to an element of array **SAM** is legal. However, because of the way Fortran arrays are stored in memory, the reference still lies in the storage allocated for **SAM** when **I**=25, **J**=2 and **K**=4; in fact, the selected element is **SAM(5,4,4)**. In general, there are an infinity of ways to write subscripts that are not strictly legal, but which name an element lying in the array. If used, this option warns only about an access that falls completely outside of the storage for an array. In other words, this option does **not** check array subscripts individually, but simply checks that the result of the calculation of the array element's location is actually within the bounds of the array.

-V

Print version information.

-vreport

This invokes the vector reporting facility and is used to tell the user what vectorization has been done to the program. A detailed listing is provided of exactly what the compiler did to each loop nest. This option does not include suggestions on how to achieve better performance. Refer to the section titled *Vector Reporting Facility* for detailed information. The output from this option is in Fortran-like notation.

-vsummary

This invokes the vector reporting facility and is used to tell the user what vectorization has been done to the program. This option, **-vsummary**, prints out what statements are and are not vectorized in each loop nest. The output from this option is in Fortran-like notation.

-w

This option suppresses warning messages during compilation.

-43

If this option is being invoked by a C program, this option supports BSD capabilities. This consists of a BSD header file and a number of BSD libraries. Use this option immediately after the **cc** command. If this option is being invoked by a Fortran program, the effect of this option is to add the Berkeley library *libc.a* to the end of the load line. This is not as powerful as using the **-43** option with the **cc** command.

3.2.1.3 Loader Options

-B*hhhhhhhh*

Generate an *a.out* file with the bss address at *hhhhhhhh*.

-D*hhhhhhhh*

Generate an *a.out* file with the data address at *hhhhhhhh*.

-e*sym*

Set the default entry point address for the output file to be that of *sym*.

-L

Do not search for **-l** libraries in */lib* or */usr/lib*.

-L*dir*

Search for **-l** libraries in *dir*.

-l*tag*

Search a library called *libtag.a*. The default is to search first in */lib* and then in */usr/lib*.

-m

Generate a simple load map on standard output.

-n

Generate NMAGIC file type.

-opct

Insert code into the program that produces a count of all the FPU ops executed by the program, dynamically.

-p

Generate code to profile the source file during execution. Refer to *Chapter 8, Tuning and Porting Code* for detailed information on using this option.

-r

Produce a relocatable output file.

-s

Strip line numbers and symbol table information from the output object file.

-T*hhhhhhhh*

Generate an *a.out* file with the text address at *hhhhhhhh*.

-t

Turn off warnings about multiply defined symbols that are not the same size.

-y*name*

Trace *name* and print out all uses and definitions.

3.2.2 Compilation Control Statements

In addition to the compilation control options that you specify on the command line, you can communicate with the TITAN preprocessor by including certain statements in your source code.

3.2.2.1 The #IF Statement

This statement allows the conditional inclusion of source statements in your program. These source statements are included only if the integer expression following **#if** is non-zero.

3.2.2.2 The #IFDEF Statement

This directive allows conditional inclusion of source statements in your program if the preprocessor identifier is defined. No values are tested with this statement; only a test for a definition of an argument.

3.2.2.3 The #IFNDEF Statement

This directive allows conditional inclusion of source statements in your program only if the preprocessor identifier is not defined. This is the reverse of the **#ifdef** statement.

3.2.2.4 The #ELSE Statement

This statement can be used with the **#if**, **#ifdef**, and **#ifndef** statements to create nesting levels.

3.2.2.5 The #ELIF Statement

This statement can be used with the **#if**, **#ifdef**, and **#ifndef** statements to create nesting levels.

3.2.2.6 The #ENDIF Statement

This statement terminates the **#if**, **#ifdef**, and **#ifndef** statements.

3.2.2.7 The #UNDEF Statement

This statement terminates the **#define** statement and causes the argument to become undefined.

3.2.2.8 The #INCLUDE Statement

This statement causes the compiler to read external source code during compilation. The external code is included at the position of the statement in your program module, just as though it had been there originally as part of the source code.

3.2.2.9 The #LINE Statement

This statement provides you with the ability to label lines of source text.

3.2.2.10 The #DEFINE Statement

This statement is most commonly used to assign values to symbols, although any kind of text can be substituted for an identifier. Once the assignment has been made, that value is now available to all statements that follow in the module.

The **INCLUDE** statement is described in detail in *Chapter 2, Fortran Statements* in the *Fortran Reference Manual.* For more information on the **#DEFINE** statement and specific references on **#INCLUDE** for the C programmer, a C reference manual should be consulted. Refer to the section in this chapter *The TITAN C Compiler* for a complete recommendation of references in this area.

3.2.3 Compiler Directives

Compiler directives are used to override the compiler's optimizations and influence all phases of compilation. A complete discussion of these directives is located in *Chapter 2, Efficient Programming Techniques.*

3.2.4 Linking Fortran and C Programs

If you do not specify the *-c* option on your command line, then a successful compilation of your program results in a file named *a.out* in the same directory in which your source program was located.

If you wish to compile several program modules individually or you wish to compile other programming language modules to be linked with your program, then use the *-c* option when you compile. Instead of a file named *a.out,* you get a file whose name is the same as your source file name with the *.f* or *.c* removed and a *.o* appended in its place. This is the object file that you later provide to the loader. See *Chapter 4, Using Libraries and the Link Editor* for more details.

3.3 Vector Reporting Facility

The way in which you can tell what the compiler has done to your code is by invoking the vector reporting facility. This can be done by the use of three different compiler options or by using a compiler directive, **VREPORT**. Be aware that all output from the reports is in Fortran-like notation and may cause problems for some C programs. Be sure to read the section that discusses the dangers of using **-vreport** and C.

3.3.1 -vsummary

This option produces the most general level of reporting possible. It only discusses whether or not the code has been vectorized. It does not discuss how things are vectorized, or why. This can be good to use when you are not sure if code is going to vectorize, and if it does not vectorize, running more detailed reports is not going to help you.

EXAMPLE

The following is an example of a program fragment and the output produced by compiling with **-vsummary**.

```
DOUBLE PRECISION A(100,100), B(100,100), C(100,100)
INTEGER I, J, K
DO I = 1, 100
   DO J = 1, 100
      C(I,J) = 0.0
      DO K = 1, 100
         C(I,J) = C(I,J) + A(I,K) * B(K,J)
      ENDDO
   ENDDO
ENDDO
END
```

and the **-vsummary** report is

```
Vectorization summary for file x.f
**************************************
Line     4:  All statements vectorized.
```

3.3.2 -vreport

This option produces the middle level of reporting. It tells you if the code vectorized and how things vectorized, but there is no discussion of why things vectorized. Using the same example fragment in the previous section on **-vsummary**, the following

example shows what the output would be if you compiled with the **-vreport** option.

EXAMPLE

```
        Vectorized Results from File x.f
                Origin - - Line 4

Line    Stmt    Time            Program

5       6       3               b2 = 100
7       10      3               b3 = 100
5       *       4               DO PARALLEL J=1, 100
4       *       4                 DO iv=1, 100, 32
4       *       8                    rv = MIN(100, 31 + iv)
*       *       9                    vl = rv - iv + 1
*       *       9                    vl = rv - iv + 1
4       *       6                    DO VECTOR I=iv, rv
6       8       33                    C(I, J) = 0.0D0
                                     END DO
7       *       4                    DO K=1, 100
4       *       6                       DO VECTOR I=iv, rv
8       12      119                       C(I, J) = C(I, J) + A(I, K) * B(K, J)
                                        END DO
                                     END DO
                                   END DO
                                 END DO
```

The left column of the printout is the line number of the source file. The middle column represents the compiler statement numbers and the right column, time, is a static guess as to how long the statement takes to execute (in ticks). Although the time statement may not be exact, you can get an indication of the relative times of each statement.

The asterisks (*) are statements that the compiler has generated. The variables that are in upper case are from the user, and the variables in lower case are compiler generated. *ts* stands for temporary scalar and *tv* stands for temporary vector.

3.3.3 -full_report

This option produces the most detailed output. It tells you if the code vectorized, how things vectorized, and why it did it. The reasons that are listed at the bottom of the report relate to vectorizer strategy, which is discussed immediately following this. It is important to understand this in order to efficiently tune your code.

EXAMPLE

The following example uses the same example fragment from the other compiler options **-vsummary** and **-vreport**. This is the output produced when **-full_report** in invoked.

```
        Vectorized Results from File x.f
                Origin - - Line 4

Line    Stmt    Time            Program

5       6       3               b2 = 100
7       10      3               b3 = 100
5       *       4               DO PARALLEL J=1, 100
4       *       4                 DO iv=1, 100, 32
4       *       8                    rv = MIN(100, 31 + iv)
*       *       9                    vl = rv - iv + 1
*       *       9                    vl = rv - iv + 1
4       *       6                    DO VECTOR I=iv, rv
6       8       33                    C(I, J) = 0.0D0
                                     END DO
7       *       4                    DO K=1, 100
4       *       6                       DO VECTOR I=iv, rv
8       12      119                       C(I, J) = C(I, J) + A(I, K) * B(K, J)
                                        END DO
                                     END DO
                                   END DO
                                 END DO

        Vector Parallel Statements

Stmt  Parallel Choices  Chosen  Reason  Vector Choices  Chosen  Reason

  8             1, 2       2                  1, 2       1    (2)
 12             1, 2       2               1, 2, 3       1    (2)


(2)   Stride one access influenced selection of vector loop.
```

The left column of the printout is the line number of the source file. The middle column represents the compiler statement numbers and the right column, time, is a static guess as to how long the statement takes to execute (in ticks). Although the time statement may not be exact, you can get an indication of the relative times of each statement.

The asterisks (*) are statements that the compiler has generated. The variables that are in upper case are from the user, and the variables in lower case are compiler generated. *ts* stands for temporary scalar and *tv* stands for temporary vector.

3.3.4 Vectorizer Strategy

In order to not only make sense out of the reports produced by the vector reporting facility, but to use them as a tool for improving your code, it is necessary to understand the strategy of the vectorizer. When the **-full_report** option is used, the reasons that are listed at the bottom of the report come from the following strategy.

In general, the strategy is as follows:

(1) Find all loops that can be done in parallel and vector.

(2) Figure out the best loop that can be done in vector:

a) remove all loops which are involved in nonlinear subscripts.

Then, starting from the innermost loop

b) choose the loop with fewest scatter-gathers.

c) choose the loop with the most stride-ones.

d) choose a loop with a known length rather than unknown length.

(3) Figure out the best loop that can be done in parallel:

a) remove the vector loop from consideration.

b) starting from the outermost loop, find the first loop that is eligible.

c) if there are no candidates, choose the vector loop.

3.3.5 Dangers in C with Vectorizing

(1) C parameter passing almost invariably requires the use of two compiler options: **-safe=parms** and **-safe=ptrs**. **-safe=parms** says that when any two parameters are passed by reference, storing into one of them does not change the other one. **-safe=ptrs** works in the same manner. When the value of one pointer changes you are not changing the value of another.

(2) **-vreport** and the compiler directive **VREPORT** are not designed for C. These are provided mostly as a convenience, as can be evidenced by the Fortran-like notation of the output.

(3) For loops are converted to while loops. This must occur because they must become DO loops to vectorize. Be careful with examples such as:

```
for (i = 0; i < *n; i = i + *s)
```

(4) Dynamic memory usage often leads to scatter-gather vector code. An example is:

```
**p     or    *p[ ]
```

(5) Use of structures or exotic pointers may cause bad things to happen in the compiler.

3.4 The TITAN Compilation System and Fortran

This section of the chapter discusses things that specifically apply to using the TITAN compilation system when coding in Fortran. The topics discussed are invoking the compiler, compiler options (not previously discussed), compilation control statements, the format of the compiler listing, and identifying errors.

The discussions in this section assume that you have read the first section of this chapter which talks about concepts that apply to using the compilation system for any language.

3.5 Calling the TITAN Fortran Compiler

The TITAN Fortran compiler command syntax allows you to intermix multiple options and file specifications

SYNTAX

fc [*options*] [*filespec*] [*options*] [*filespec...*]

where

options
is a set of options, formatted as described below.

NOTE
Options and file names may be intermingled.

filespec
is a UNIX file path by which a source file may be accessed. The TITAN Fortran compiler can handle a mix of different programming languages and object files on the same command line. For example, file names can end with the characters *.f*, *.c*, *.o*, *.a*, or *.s*. If the compiler does not know how to create or to access a file you have specified, an error notice is generated.

EXAMPLE

NOTE
Filenames or parts of pathnames (between /'s) cannot exceed 14 characters.

An example *filespec* is:

```
/usr/test/fortran/myfort.f
```

In the example, the pathname part of the *filespec* is **/usr/test/fortran** and the filename is **myfort.f**.

3.6 Form of the Options

TITAN compiler options, when used, take one of the forms shown in Table 3-4.

Table 3-4. Form of Compiler Options

Form	Option
1	*-option*
2	*-option symbol_or_file_path*
3	*-option number*
4	*-option symbol*
5	*-option=number*
6	*-option=symbol_or_file_path_list*

A *symbol_or_file_path_list* can be empty; if the list is not empty, the list should be a comma-separated list of symbols and file pathnames. Generally, options have the first, fifth, or sixth forms; the second, third, and fourth forms are loader options which are passed on to the loader.

A *symbol_or_file_path_list* must not include blanks. The symbols generally allow the use of letters, numbers, and special characters that are not otherwise meaningful to the shell or to the option parser.

Options are applied to the compiler in the order written on the command line, from left to right; similarly, suboptions (the items to the right of an equal sign in form 6) are applied to an option in the order written, from left to right. Options for the loader are passed to the loader in the order of the command line from left to right. Options and filenames may be intermingled so that libraries may be searched in the proper order with reference to particular object files; *-ltag* loader options must sometimes precede filenames. However, other options are collected and applied all at once to each of the processes invoked; placement of an option such as *-O1* after a filename still causes the *-O1* option to be applied to all files compiled.

3.7 Fortran Compiler Options

Options direct the compiler in its operations. Table 3-5 lists the possible options followed by a detailed explanation. This table does not duplicate the options already mentioned in the first section of this chapter that apply to both Fortran and C programs.

Several options have extensive lists of possible suboptions, including the suboptions *all* and *none* and, generally, a negative option prefixed with the characters *no*. Many options appear in both positive and negative forms (for example, *-list* and *-nolist*). These options are switches; the last appearance in the command line controls the position of the switch. Each option has a default position indicated in italics.

Table 3-5. Fortran Compiler Options

Option	Negative Form	Descriptions
-blanks72		Pad source file lines with blanks
-check	*-nocheck*	Add run-time error checking code
-continuations=n		Specify acceptable number of continuation lines
-cpp		Invoke the C preprocessor
-cross_reference	*-nocross_reference*	Generate a cross-reference listing in listing file
-debug	*-nodebug*	Add debug data to object file
-no_directive		Do not apply directives during compilation.
-double_precision	*-nodouble_precision*	Use double precision for all undeclared variables
-d_lines	*-nod_lines*	Compile debug lines that start with D in column 1
-fast		Load a faster math library
-implicit	*-noimplicit*	Untype all variables
-include=pathname		Specify a directory to search for included files
-include_listing	*-noinclude_listing*	Add included files to the listing file
-i4	-noi4	Interpret all INTEGER and LOGICAL as though declared *4
-list	*-nolist*	Generate a listing file
-messages	*-nomessages*	Allow printing of warning messages
-object	-noobject	Generate an object file
-onetrip	*-noonetrip*	Make sure all DO loops execute at least once
-save	-nosave	All variables declared are saved
-standard	*-nostandard*	Check for standard Fortran 77 usage
-verbose		Use verbose message output

-blanks72

Pad all lines in the source file on the right to column 72. Lines longer are untouched. Does not work with **-cpp**, **-P**, or **-E**. Affects only *.f* files. Slows down compilation time.

-check

Generate code to check a variety of runtime errors. This option may appear in form 1 or 6. If form 6 is used, then it may take suboptions from the table below. The default is *-check=overflow*.

Table 3-6. Suboptions for Option Form 6

overflow	**nooverflow**
bounds	**nobounds**
underflow	**nounderflow**
all	**none**

-nocheck

This option has the same structure as *-check*. It turns off all checking, except as modified by its suboptions.

When you specify *overflow*, the compiler generates code to check for overflow operations involving BYTE, INTEGER*2 and INTEGER*4 calculations. Specifying *bounds* is the same as specifying the parameter *-subcheck*, as described later.

-continuations=*n*

Set the number of continuation lines the compiler accepts for any statement. The number *n* may range from 0 to 99; the default is 19.

-cpp

This invokes the C preprocessor. If this is not used, all source lines with a pound sign (#) in column 1 are going to be silently ignored. The options **-P** and **-E** always invoke the preprocessor even if it is not used.

NOTE
Line numbers on messages may be confused when **-cpp** is used. You can match up line numbers with the source file by using the **-list** option. Look at the first column of numbers in the *xx.L* file that is produced.

-cross_reference

Generate a cross-reference, if a listing is generated. The default is *-nocross_reference*.

-nocross_reference

Turn off any cross-reference listing.

-debug
-g

Generate information for the TITAN debugger. The default is *-nodebug*. Options **-debug** and **-g** are synonymous.

-nodebug

Do not generate debugging information.

-d_lines

Compile source file lines with a 'D' or a 'd' in column 1. The default is *-nod_lines*. The 'D' or 'd' is treated as if a blank were substituted for it.

-nod_lines

Source file lines with a 'D' or a 'd' in column 1 are to be treated as comments.

-double_precision

Set all undeclared floating point variables and all variables and constants declared as floating point to be as if they were declared REAL*8 (or double precision). *-nodouble_precision* is the default.

Be aware that use of this option affects functions and their arguments, which may cause them not to work. As an example, *cputim*(3F) does not work with this option because a REAL is converted to a REAL*8. Be sure to understand the implications of using this option with functions.

-nodouble_precision

Treat floating point variables and constants by the Fortran 77 rules; specifically, compile them as REAL*4 unless otherwise specified in the source.

-fast

This causes a faster math library to be loaded, and affects the actual code that is generated as well as the libraries linked into the executable code. This may also cause math functions in your program to be potentially less accurate; as much as two decimal places in a double precision number. It is also not quite as observant of some math rules in the last bit or two. In general, this option allows optimization which may lose small amounts of precision.

-implicit

Make all variables in a program untyped. Using this option, all variables must be declared or an error occurs. This option has the same effect as an **IMPLICIT NONE** statement at the beginning of each routine in a source file. *-noimplicit* is the default.

-noimplicit

Follow the ordinary Fortran 77 typing rules.

-include=*pathname*

Add the directory named by *pathname* just before the standard list of directories. That is, each file name in an INCLUDE statement is searched for in order through a list of directories. Initially, the list consists of the directory from which the source file came and then a standard system-defined list of directories. Each instance of a *-include* option adds another directory to be searched; the new directory is placed just before the standard list and thus the directories are searched in the same left-to-right order as the *-include* options appeared on the command line. If *pathname* is empty, the standard list of directories is treated as if it were empty for this compilation only.

As an example, assume that source file *choochoo.f* needs to use header files from directories */leaders/miller* and */singers/eberle* in this order; then the command line would be

```
fc -I/leaders/miller -I/singers/eberle  choochoo.f
```

If the standard header file should not be searched, then the command line would be

```
fc -I ... same as above ...
```

The **-I** option applies to all the source files compiled, regardless of the placement of the **-I** options and the source file names on the command line.

-include_listing

List included source as well as the original source file when generating a listing. The default is *-noinclude_listing*.

-noinclude_listing

Do not copy included source files to the listing.

-i4

Interpret INTEGER and LOGICAL declarations as if they had been written INTEGER*4 and LOGICAL*4.

-noi4

Interpret INTEGER and LOGICAL declarations as if they had been written INTEGER*2 and LOGICAL*2. The default is *-i4*.

-list

Generate a listing from the compilation. The listing file is named after the source file. The default is *-nolist*.

-nolist

Do not generate a listing file.

-messages

Allow warning messages to be printed. The default is *–nomessages*.

-nomessages

Do not allow warning messages to be printed.

-no_directive

Do not apply compiler directives. The default is to apply directives.

-object

Generate object files from this compilation. This is not quite the same as the *-c* option, which precludes loading; *-noobject* precludes even the generation of object files.

-noobject

Do not generate object files from this compilation; check correctness of the source code only. The default is *-object*.

-onetrip

Generate code to guarantee that all DO loops execute at least once. This is a compatibility feature for programs originally written for use with Fortran 66. The default is *-noonetrip*.

-noonetrip

DO loops may execute zero times.

-save

Save all declared variables. The default is *-save*.

-nosave

This has no effect because TITAN Fortran automatically saves all variables.

-standard

Check for standard Fortran 77 usage and flag TITAN extensions with warnings. This option may appear in forms 1 or 6 with the suboptions shown in Table 3-7.

Table 3-7. Suboptions for -standard option

syntax	**nosyntax**
source_form	**nosource_form**
all	**none**

The default is *-nostandard*. If this option appears in form 1, it is the same as *-standard=all*.

-nostandard

Turn off checking for extensions to Fortran 77. This option has the same structure as *-standard*.

-verbose

Generate more messages tracking the progress of the compilation. As a default, these additional messages are suppressed.

3.8 Compilation Control Statements

In addition to the compilation control statements that are discussed in the first section of this chapter, TITAN Fortran offers one other, the **OPTIONS** statement. An **OPTIONS** statement allows you to specify certain compiler options in lieu of specifying them on the command line. The **OPTIONS** statement is described in detail in *Chapter 2, Fortran Statements*, in the *Fortran Reference Manual*.

3.9 Format of the Fortran Listing

The listing of the compiler output contains

- A section that contains your source code with line numbers prefixed to each source line.
- A storage map.
- A summary of the compilation, including the settings of the options and a summary of the numbers of errors.

3.9.1 Source Code Section

This section contains the program source code as it appears in your source program. The example below shows a sample source code listing from a subroutine.

EXAMPLE

```
(date)    (time)
Source Listing  File: /tmp/ftest.f

   19   *      Function subprogram unit follows.
   20          INTEGER*4 FUNCTION nfunc(k)
   21          nfunc = 0
   22          DO 10 i = 1,k !Loop to compute sum.
   23          nfunc = nfunc+i
   24       10 CONTINUE
   25          RETURN  !Return value in function name.
   26          END
```

3.9.2 Storage Map

After each program unit, the compiler listing includes a storage map that lists information about the program unit. The storage map includes the following:

- Entry Points: all entry points into the program are listed. If an entry point returns a specific data type, that data type is declared along with the entry point.
- Storage Blocks: the storage usage of the program is listed.
- Variables: the name and data type of each variable declared for use in this program unit is listed, as well as its offset addresses within the program unit.
- Arrays: the name, size, location, and type of each array is listed.
- Intrinsic Functions: all intrinsic functions that are called from within this program unit are listed.
- Statement Functions: all statement functions are listed by name and by their declared data type.
- Externals: all externals and their declared data types are listed.

EXAMPLE

```
(date)  (time)
Symbol Storage Map  File: /tmp/ftest.f

ENTRY POINTS
  Offset  Type  Name

       0  I*4   NFUNC

STORAGE BLOCKS
    Size  Block

       4  local
      28  constants

VARIABLES
  Offset    Size  Type  Block           Name

       0       4  I*4   local           I
     ---       4  I*4   dummy           K

ARRAYS
  Offset    Size  Type  Block           Name

PARAMETERS (Offset is within Storage Block 'constants')
  Offset  Type  Name                            Value

INTRINSIC FUNCTIONS
  Name

STATEMENT FUNCTIONS
  Type  Name

EXTERNALS
  Type  Unit_Kind     Name
```

3.9.3 Cross Reference

If, as part of the command line, you specify the command option *-cross_reference*, then a cross-reference listing is generated with the listing file (if and only if *-list* was also specified). This cross-reference listing shows all symbols and labels, listing each symbol and the line numbers on which it appears. It also lists labels, showing the line number at which each label is defined and the line number at which a reference to that label occurs.

EXAMPLE

```
(date)  (time)
Symbol Cross Reference  File: /tmp/ftest.f

Symbol                              Line Number(s)

EXONE                                    1
N                                       10     12     13
NFUNC                                    7     12
SUM                                      7     12     13

(date)  (time)
Label Cross Reference  File: /tmp/ftest.f

Label  Defined  References(s)

33         14         13
```

3.9.4 Compilation Summary

The summary section lists the command qualifiers in effect for this compilation as well as statistics about the source files processed. Here is a sample compilation summary.

EXAMPLE

```
(date)  (time)
Source Listing  File: /tmp/ftest.f

COMMAND QUALIFIERS

  -optimize=0 -assemble -list
  -include=/la/cross/include /tmp/ftest.f

-BACKEND=ffe3,ffe1
-CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
-STANDARD=(NOSYNTAX,NOSOURCE_FORM)
-INCLUDE=/la/cross/include
-CONTINUATIONS=19  -FE_FISHERBURKE  -FE_VERBOSE
-I4  -LIST  -OPTIMIZE=0  -PADDING=3  -S  -WARNINGS


FRONT END STATISTICS

  Input File:           /tmp/ftest.f
  Base Filename:        ftest.f
  Source Processed:     26 lines
```

3.10 Errors and Compiler Diagnostics

The TITAN Fortran compiler identifies syntax errors and language violations. It outputs various error messages to your terminal, describing the source of the error, usually with enough information about the error to allow you to correct it.

If you have requested a listing file to be produced (*-list* option), any errors in your source file are printed immediately following the offending statement.

If you need help in interpreting the error message, you'll find the error messages and their meanings listed in *Appendix A* in the *Fortran Reference Manual.*

3.11 The TITAN Compilation System and C

This section of the chapter discusses things that specifically apply to using the TITAN compilation system when coding in C. The topics discussed are elements of the compiler and its invocation, compilation control statements, extensions to the language such as a new storage class keyword, and identifying errors.

The discussions in this section assume that you have read the first section of this chapter which talks about concepts that apply to using the compilation system for any language.

A sound knowledge of the C programming language is also assumed. *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr., Second Edition, Prentice-Hall, 1987 is an excellent modern reference on the details of C, and is mentioned in this section. More information on the proposed ANSI standard for C may be found in the draft standard dated October 10, 1986. Some C language features have been derived from ideas developed in the C++ language; more information about C++ may be found in *The C++ Programming Language* by Bjarne Stroustrup.

3.12 Elements of the TITAN C Compiler

This section describes the elements of the TITAN C compiler, paying particular attention to the front end. The C language front end is based on the front end distributed with System V Release 3 of AT&T UNIX. This includes the following programs and utilities:

cc	the C language compiler
cpp	the C language preprocessor
ld	the link editor
cb	a C program beautifier
regcmp	a program to compile C program regular expressions

See the *Commands Reference Manual* for full details of these utilities.

TITAN C supports *ANSI* function call semantics. For a function declared with a function prototype, the TITAN C compiler enforces strict type matching rules between the actual arguments in a call of the function and the formal arguments in the definition of the function.

3.13 Calling the TITAN C Compiler

You invoke the TITAN C compiler with the **cc** command.

SYNTAX

cc [*options*] [*filespec*] [*options*] [*filespec...*]

NOTE
Options and file names may be intermingled.

where

options
is a set of options, formatted as described below.

filespec
is a UNIX file path by which your source file may be accessed.

The compiler understands filenames ending with the suffix **.c** to be C source files.

For example, *cc program.c* is the most basic form of the command and performs the following actions on *program.c*

- Invokes the C preprocessor, **cpp**.
- Invokes the C compiler itself.
- Invokes the assembler, **as**.
- Invokes the link editor, **ld**.

When the whole compilation is finished, the resulting executable file, *a.out*, is left in the current directory. You can rename *a.out* to any legal filename.

3.14 C Compiler Options

Options direct the compiler in its operations. Table 3-8 lists the options followed by an explanation. This table does not duplicate the options already mentioned in the first section of this chapter that apply to both Fortran and C programs.

Table 3-8. C Compiler Options

Option	Description
-NW	Suppress compiler warnings.
-n	Suppress the standard C startup routine.
-safe=parms	Parameters are not affected.
-safe=ptrs	Pointers are not affected.
-safe=loops	Upper bounds do not vary.
-v	Use verbose message output.
-vector_c	Equivalent to **-safe=loops -safe=parms**.

-NW

Suppress compiler warnings.

-n

Suppress the standard C startup routine.

-safe=parms

When any two parameters are passed by reference, storing into one of them does not change the value of the other one.

-safe=ptrs

When any two pointers are manipulated, changing the value of one pointer does not change the value of the other.

-safe=loops

This guarantees that all **for** loops within the program have upper bounds that do not vary within the loop. This is necessary for loops that have array references or (*) values as upper bounds.

-v

Generate more messages tracking the progress of the compilation. As a default, these messages are suppressed.

-vector_c

This is equivalent to specifying **-safe=parms -safe=loops**.

3.15 Extensions to the Compiler

This section discusses either new or unusual features that have been incorporated into the compiler. These features include new compilation control statements, a new storage class, argument (function) prototypes, & arguments in declarations and calls, and two new compiler directives.

3.15.1 Compilation Control Statements

In addition to the compilation control statements that were identified in the first section of this chapter and control lines that are specified in the C reference manual by Harbison and Steele, TITAN C is supporting two new control lines. The format of these statements is

SYNTAX

#ident "*comment*"
#pragma *identifier*

The *comment* line puts the comment string into the **.comm** section of the *.o* file. It is usually used for version control. The loader **ld** and other tools interpret and list these comments. By default, all comment sections in the **.o** files are concatenated in the *a.out* file.

The **#pragma** directive is used as a general mechanism for providing optimization information to the compiler. It may also be used to provide runtime information which can be used to generate more efficient code. Refer to *Chapter 2, Efficient Programming Techniques* for a discussion of how to use this control line.

3.15.2 Storage Class threadlocal

TITAN C includes an additional storage class keyword, **threadlocal**, that permits you to allocate storage that is local to each thread cooperating on the task to be performed. The **threadlocal** keyword allows you to declare an uninitialized variable that is passed to each thread. Refer to *thread* in the *Commands Reference Manual* for an example of a multithreaded task. You declare such a variable as shown in the following example:

```
threadlocal int i;
```

The loader collects all the thread local storage together and puts it in a contiguous block of virtual memory. The operating system then marks these pages so that upon a *thread* call, each thread gets a separate copy of these locations. The initial contents of these locations are inherited from the caller.

3.15.3 Compiler Directives

TITAN C supports two additional compiler directives: **NOTREACHED** and **NO_FLOAT**. The **NOTREACHED** directives turns off the compiler warning about code that is not reached. The format of the directive is

SYNTAX

#pragma NOTREACHED

This directive should be placed **at** the location of the code that is questionable.

The **NO_FLOAT** directive tells the compiler that the program does not use the FPU even though *vararg* has been specified. The assumption by the compiler is that if *varargs* is specified, floating point arguments are used. It is more efficient for the program not to have to touch the floating point unit if it is not needed. The format of the directive is

SYNTAX

#pragma NO_FLOAT

EXAMPLE

The following is an example of when you might want to specify NO_FLOAT.

```
#include varargs
#pragma NO_FLOAT

int printf(char *format, va_del, va_list)
{
        va_list ap;
        va_start(ap);
        count = duprint(format, ap, stdout);
        va_end(ap);
        return(error);
}
```

3.15.4 Argument (Function) Prototypes

TITAN C includes the ability to declare the types of arguments for functions. Strong type checking is provided with this. For a complete discussion of this topic refer to the reference manual by Harbison and Steele and the ANSI draft standard.

3.15.5 & Arguments

TITAN C supports the use of & arguments in declarations and in function calls. When an & argument is used in a function declaration, that argument is then passed by reference. The semantics for using this are the same as in the C++ language.

EXAMPLE

The following is an example of coding an & argument in a function declaration in which x is passed by reference.

```
int f(int &x)
```

If an & argument is used in a function call, you are allowed to take the address of a constant.

EXAMPLE

```
main()
{
        f(&);
}
```

The function argument is the address of a local variable which has had that constant stored in it.

3.16 Errors and Compiler Diagnostics

The C compiler generates error messages for statements that do not compile. The messages are generally understandable, but in common with most language compilers they sometimes point several statements beyond where the actual error occurred. For example, if you inadvertently put an extra semicolon (;) at the end of an **if** statement, a subsequent **else** is flagged as a syntax error. In the case where a block of several statements follows the semicolon (;), the line number of the syntax error caused by the **else** starts you looking for the error well past where it is. Unbalanced curly braces, { }, are another common producer of syntax errors.

3.16.1 Compiler Advisory Warnings

The Fortran and C compilers may, for a variety of reasons, change the optimization level set by the user or may inhibit optimization of various kinds of variables. Whenever this occurs, the compilers issue warning messages to you. You may then change your code if you wish to enable optimization.

Use of setjmp inhibits all optimizations.

Using the setjmp function in C causes all optimization of the procedure from which setjmp is called to be inhibited. If this is not done, the values of local variables may not be correct at the sight of the longjmp.

Use of assigned format statements inhibits all optimizations.

If a label variable is used to hold a format label and is then used in a read or write statement, the compiler shuts down all optimization. Another standard allows format statements to use variables in formats. When a specific label is used in a read or write, the compiler can tell which variables are used by that IO statement, but when an assigned format is used, the compiler cannot. The only way in which it can be safe and ensure that needed values are present is to shut off all optimization.

Size of procedure x inhibits all optimizations.

Procedure *x* in this case is so large that that optimization cannot get enough memory or disk space or such to do optimization. As a result, it shuts down all optimizations and attempts to continue. It is not uncommon for the compilation to abort after this, because a procedure too large to optimize is usually too large to compile.

Short and byte variables not optimized.

At present, some quirks in the C front end mean that if the compiler were to optimize short and byte variables, incorrect results would be produced in some cases. As a result, the optimizer does not perform any optimizations on short or byte variables in either Fortran or C. Note that this does not inhibit optimization of other types of variables.

Use of many calls inhibits external variable optimization.

Optimizers can take a lot of memory to perform optimizations. When a procedure contains lots of calls to other procedures, the amount of memory may be enormous, and with little resulting optimization. In particular, many classes of variables appear to be both used and defined at every call sight and external or COMMON variables are one such beast. The compiler attempts to anticipate when optimization will take too much memory and shut down optimizations for various classes of variables. One of the first to go is typically external (COMMON in Fortran) variables, which is what this message signifies. Most of the time there is little optimization that is missed here, because external variables typically cannot be optimized anyway if there are lots of calls it looks like its changing all the time anyway.

Use of many calls inhibits reference parameter optimization.

This is the same as above for variables that are passed by address to another procedure.

Use of many calls inhibits static variable optimization.

This is the same as above for statics in C, local variables in Fortran. Starting to get serious here.

Non-integer loop control suppresses some optimization.

The compiler has extensive optimizations for loops, but it is based on having loops that are controlled by integers. Loops that are controlled by floats, and so on, cause many of these optimizations to be inhibited.

METHODS FOR DEBUGGING CODE

CHAPTER FIVE

The TITAN compilers supply debugging information whether or not the debugger option *-g* is specified during compilation of your program. It is necessary to understand what debugging information is available and at what phase of compilation this information is created, in order to select the appropriate debugging tool for your program. This chapter discusses TITAN versus standard UNIX compilation systems, why and when you should use a debugger and related debugging support tools that are available.

5.1 TITAN Versus Standard UNIX Compilation Systems

The TITAN compilation system differs from the AT&T System V and BSD 4.3 UNIX compilation systems, and these differences can affect your choice of debugging tools and the specification or lack of a debug option in your program. Each compilation system begins with compiling a source program and producing a **.o** file. Next, the link editor is invoked and it produces an *a.out* file. A symbol table has also been created by the compilation. Although both compilation systems produce the same files, the contents of these files are quite different.

Table 5-1 illustrates what information is present in the **.o** and *a.out* files on both compilation systems, with and without specifying the *-g* option. Notice that the TITAN compilation system, **without** the *-g* option specified, includes all of the information that is only available to the standard UNIX compilation systems when the *-g* option is specified.

Using the *-g* option with the TITAN compilation system has different effects than using it with the standard UNIX system. The *-g* option (on the TITAN) removes all optimization. It suppresses this by not allowing register allocation of variables. In addition, there is overhead with the *a.out* file.

The default TITAN compilation system, that is without the *-g* option, should normally provide you with adequate debugging

Table 5-1. Comparison of Availability of Debugging Information

Compilation	Information present on standard UNIX systems	Information present on the TITAN system
Without *-g* option after creation of **.o** file	1) COFF symbol table	1) COFF symbol table 2) Local "other" information present, such as argument type and relative location
after creation of **a.out** file	1) Addresses for static references 2) All information from above section	1) Addresses for static references 2) Basic block line number addressing 3) All information from above section 4) All addresses are bound (where applicable)
With *-g* option after creation of **.o** file	1) COFF symbol table 2) Line numbers	1) COFF symbol table 2) Line numbers 3) All optimization **removed** 4) Local "other" information present, such as argument type and relative location
after creation of **a.out** file	1) Addresses for static references 2) Basic block line number addressing 3) All addresses bound (where applicable 4) Local "other" information (see above) 5) All information from above section	1) Variables kept in memory - no register allocation of variables 2) Optimization removed 3) Line numbers

information. The appropriate time to include the *-g* option is when you have already isolated the problem in the program and need to single step through the execution of the program.

5.2 When to use a Debugger

A debugger is a program revealer, used to figure out what happened to the code. When any behavior in your program is different than what you expect, that is the time you should use a debugger. There are generally four circumstances in which you would want to use a debugger.

1) The program does not work and you need to determine what the error in logic is.

2) The program works but the answers are not what you expected. You must isolate the differences between your expectations and the results.

3) The program works and the answers are correct, but the program runs extremely slowly.

4) The program works as designed but you want a post mortem on what happened during execution.

Do not confuse using a debugger with the need to include the *-g* option in the compilation of your program. The TITAN compilation system usually provides more than adequate information to debug your program without the *-g* option. In fact, using this option may have undesirable effects unless used in the proper circumstances. Refer to the previous section for more information on the *-g* option.

5.3 Debugging Tools

In addition to the TITAN debugger, **dbg**, there are a number of other tools available to aid you in debugging and analyzing programs. This section provides you with a brief description and discussion of some of these debugging tools, as well as an introduction to the TITAN debugger.

5.3.1 dbg

dbg is the TITAN debugger and can best be described as a program revealer. It is a tool that allows you to find out what has happened to your code. The TITAN debugger has some abilities and features which are different from the standard UNIX symbolic debugger. The most significant of these abilities can be summarized as follows:

- The "master" instructions in the debugger reflect the position or operation of the code in the IPU.
- The debugger allows you to ask what the FPU is doing.
- The debugger can tell you when a process goes parallel and what the threads are doing.
- The debugger can be used to keep track of the synchronism between the IPU and the FPU.
- The debugger can be used to grab any process, and then look at it, stop it, kill it, or take control of it.

dbg can be used for the same tasks as any other debugger would be, such as controlling the execution of processes, setting break points, examining the contents of memory and registers and gaining source and object file information. For more information on the specifics of using **dbg**, refer to *Chapter 6, Using the Debugger.*

5.3.2 nm

The **nm** command is a tool that can be used to display the symbol table from your *a.out* file. Some of the types of information that are displayed for each symbol in the table include storage class, type, size in bytes, source line number at which the symbol is defined, and the object file section containing the symbol.

Output for this command may be controlled by a large number of options, which can help you organize and sort external and static symbols. For more information on this command, refer to the *Commands Reference Manual.*

5.3.3 od

The octal dump command, **od**, dumps and displays a file in one or more formats. These formats allow interpretation of certain data types in forms that include hexadecimal, octal, and signed and unsigned decimal.

Dumping a file may be a useful tool in debugging certain portions of your code. Refer to the *Commands Reference Manual* for more information on using **od**.

5.3.4 prof

The **prof** command produces a report on the amount of execution time spent in various portions of your program and the number of times each function is called. This can be a useful tool if your program is producing the desired results but is running very slowly.

To use this command, your Fortran or C program may be linked with the *-p* option, and then when the program is run a file called *mon.out* is produced. *mon.out* and *a.out* then become input to the **prof** command.

A program may be profiled by using the **mkprof** command. This requires no compilation of your program. This produces the same kind of results as running the **prof** command.

For detailed information on using the **prof** command, refer to the *Commands Reference Manual* and *Chapter 8, Tuning and Porting Code* in this manual.

5.3.5 size

The **size** command is another tool which can be used to gain more information about your program and its subsequent execution. This command produces information on the number of bytes occupied by the three sections (text, data, and bss) of a common object file when the program is brought into main memory to be run. Refer to the *Commands Reference Manual* for more information.

USING THE DEBUGGER

CHAPTER SIX

This chapter is divided into two sections. The first section provides an overview of **dbg** and its key concepts. The second section discusses the commands, in detail, through examples. This chapter is designed to allow you to begin using the debugger as soon as possible. Refer to *Chapter 7, dbg Language Syntax* for the full usage of all commands.

6.1 Overview of Key Concepts

dbg is one of several tools that can be used to help you debug your program or reveal what the program is doing. This chapter makes the assumption that you have read *Chapter 5, Methods For Debugging Code* and have determined that **dbg** is the correct tool for analyzing your program.

6.1.1 Command Language

The TITAN debugger provides extensive features for analyzing and debugging your program. Any Fortran or C expression can be used while in the **dbg** environment and there are over 90 keywords to help you. *WHERE* and *WHEN* clauses provide great flexibility when using simple expression, keyword, or compound commands. Keyword commands are divided into nine categories which allow you, among other things, to control execution of a program, set break and watch points, and log a debugging session. **dbg** also allows you to examine aspects of your program and its execution that are indigenous to the TITAN. These include keeping track of the synchronism between the IPU and the FPU, finding out what the FPU is doing, and seeing when a process goes parallel and what the threads are doing.

Summary of valid commands:

- any Fortran or C expression
- **dbg** keyword
- **IF, FOR, WHILE,** and **DO ... ENDDO** statements
- declaration of temporary variables

6.1.1.1 WHERE and WHEN Clauses

Every command has both a *WHERE* and *WHEN* clause associated with it. The *WHERE* clause forces the command to be executed at a specified address. The *WHEN* clause specifies how often the command is to be executed.

A *WHERE* clause can be specified with the words **AT** or **IN,** followed by an address. The *WHEN* clause can be specified with the words **ONCE** or **ALWAYS.**

Each command in **dbg** can then be thought of in this general format:

```
[ONCE | ALWAYS]    COMMAND    [IN | AT]    [ADDRESS]
```

6.1.2 Scope

The **dbg** command language allows you to ask questions about your program and its execution. In order to ask these questions you need to know where you are in your program, and how to ask a question about something that is in another portion of your program. This is accomplished by the use of **scope,** which defines a location within your program.

All symbols and temporary variables within your program have a scope associated with them, and this scope is their relative location within your program. For example, a variable might be local to a function, external, or local to a file. In order to find out information about this variable, you must either be in the section of your program that defines or knows about that variable, or use a scope specifier in front of the variable name. You may also explicitly set the scope of the variable.

List scope, or the scope of the list file, is used to identify the current list file, and may also be explicitly set. When this scope is explicitly set, the **list** and **file** commands can then determine what file and line numbers to display.

6.1.3 Addresses

An address is a representation of a memory location, and **dbg** allows four ways in which to express this. These are:

- virtual address: an integer expression or a name
- line number: an integer expression that cannot be a virtual address or #n, where n is an integer constant
- scope specifier
- breakpoint number: n, where n is an integer constant specifying the block

6.1.4 Lexical Conventions

All keywords and Fortran names are recognized on a case insensitive basis. Keywords are also recognized on an unambiguous prefix basis, which means that **dbg** attempts to understand what keyword you are looking for with as few letters as necessary. A conflict between a keyword and a source name is resolved in favor of the keyword. If you do not want to change this default, but still force recognition of a source name rather than a keyword for one command, precede the name with a pound sign #.

Blank input lines are ignored, and a line may be continued by using a backslash as its last character. There is no limit for line continuations. If there are many multi-line commands, it may be useful to execute the commands as a script, using the INPUT command.

6.1.5 Abbreviations

Following are the list of allowable **dbg** abbreviations, and they must be the **only** contents of a command line.

!! repeat the previous command

!n repeat command 'n' by reparsing it (n an integer constant)

&n repeat command 'n' by re-executing the internal parse tree that has already been created for it

. NEXT

, STEP

.. CONTINUE

? HELP

6.2 Command Examples

This section of the chapter shows you how to actually use **dbg** to debug your program. The command examples provided are designed to give you a basic understanding of using **dbg**. This does not include all possible elements of the command language. Be sure to refer to *Chapter 7, dbg Language Syntax* and the on-line help facility for more complete information.

When you debug a program there are some tasks that you commonly want to perform, such as specifying an address. There are also some things you may want to do that are necessary because of the hardware your program is executing on. This section covers some of the more common debugging activities as well as those special tasks that you must be aware of when running on a TITAN.

6.2.1 Compilation

dbg can be used to debug any executable and core file when compiled with *-g*, the debugging flag. Compilation with the debugging flag results in two actions: line number information for basic block entries are put into the **.o** file, and a **.db** file is created with local symbol table information. The line number information is created through the loader and passed to the executable file. The symbol table information is not created this way and therefore the **.db** file must be available for **dbg** to read.

Compiling with the *-g* flag implies an *-O0* flag, which causes the compiler to minimize optimization and ensures that the current values of all variables are in memory when the program counter crosses a source level instruction line boundary.

6.2.2 Invoking dbg

The syntax for invoking the TITAN debugger is

SYNTAX

dbg [*a.outfile*] [*corefile*]

where

a.outfile
defaults to **a.out** produced by compilation or link editing. This may be changed to reflect your own file name.

corefile
defaults to **core**, and may be changed to another corefile name.

Upon entering **dbg** the debugger reads header information from *a.outfile* and read registers, data size and location, and stack size and location from *corefile*, if *corefile* exists. **dbg** then enters interactive mode and prompt for input with

dbg < *n* >

where

n is the **dbg** command number, starting at zero (0).

6.2.3 Execution Control

It is necessary to control the execution of the process being debugged. The following examples explain the execution control commands.

EXAMPLES

Begin execution of the *a.out* being debugged:

`run`	*These commands kill the current running program, and place the program into its initial state and pass it execution control.*
`rerun`	

Step through the code:

`mnext 5`	*Step through the next 5 machine instructions. This command does not enter any functions or subroutines.*
`mstep`	*Go to the next basic block, at machine level.*
`next 10`	*Step through the next 10 source level instructions and advance the program counter at source line level. This command does not enter any functions or subroutines.*
`step 2`	*Go to the second source level basic block and advance the program counter at source line level. This command may step into procedures.*
`continue`	*Execute the code up to the next break point.*
`,`	*Go the the next source level basic block.*

Jump to another place in the code:

`goto #26`	*Jump to line number 26.*
`goto my_func`	*Jump to the beginning of the function my_func.*

Execute a function or subroutine:

`stop in my_func` `stop at new_address`	*Breakpoints can be set around a function and you may then use any valid commands to step through the function.*

Grab and process and examine it:

`grab 0196`	*Grab process 0196 and suspend it. This now becomes the process being debugged.*

Let go of the process you grabbed:

`kill`	*This kills the current running process.*
`release 0196`	*This releases process 0196 and continues normal execution.*

6.2.4 Breakpoints

Breakpoints are used to temporarily suspend execution of the process being debugged, usually so that you can examine the execution up to a certain place in your code. Breakpoints can be set, watched and unset. The following examples show you how you might do these things.

EXAMPLES

The following are general examples of setting a breakpoint, noting that **break** and **stop** are equivalent:

```
stop in main
break at main
stop at my_func
stop at someaddress
stop at line#
break in main
```

The following command gives the status of breakpoints:

```
status
```

The following commands unset a break or watch point:

`cancel 0 1`	*This removes command #1 in block #0.*
`suspend (2,1)`	*This turns the inactive bit on for command #1 in block #2. The sticky bit remains as originally set.*

`activate (2,1)`	*This turns both the inactive and sticky bits off for command #1 in block #2. The command can now be executed but it is not sticky.*
`resume 3 2`	*This turns the sticky bit on and the inactive bit off for command #2 in block #3. The be executed.*

6.2.5 Examining the Contents of a Process

Examining a process can involve such things as looking at part of memory, dumping a register, or looking at the stack. These examples provide some help for these and other tasks.

EXAMPLES

Display 100 words of memory at location **x**:

`dis x 100`	*This is displayed as machine instructions.*

Display the contents of a variable, my_var:

```
my_var
```

Dump register x:

`dump $x`	*This displays the contents in hex and ASCII.*

Display a trace of the stack:

```
where
traceback
```

Display the contents of an array, **arr_name**:

```
arr_name
dump arr_name
dis arr_name
```

6.2.6 Where am I?

With complex programs it is easy to temporarily lose track of where you are in your code and what the scope of the current process is. The following examples use commands that help you with some of these questions.

EXAMPLES

What line number am I currently executing or have just executed?

`line` — *This echos your current line number.*

What function or subroutine am I in?

`scope` — *This echos your current scope.*

`func` — *This echos the current function.*

What files am I in?

`file` — *This echos the current file being debugged.*

6.2.7 Lexical Settings

You may want to display some portion of memory in hexadecimal or decimal, or you may want to see the type declaration of a function. These types of tasks can be handled as follows:

EXAMPLES

Display 100 words of memory at location x in hexadecimal:

```
dump x 100
```

What is the type declaration of function x?

```
whatis x
```

6.2.8 Recording a Debugging Session

You may want to keep a record of the execution steps and controls you have invoked while in **dbg**. The following examples explain the syntax of doing this.

EXAMPLES

To start logging a debugging session, you might say:

`log "my_file.log"`	*All commands and results are placed in the file my_file.log.*

To suspend logging a debugging session:

`unlog`	*This suspends the logging which can be restarted at any time with another log command.*

6.2.9 Displaying the Program

Program text may be displayed as source or assembly instructions. The **list** and **window** commands display text as source. The **dis** and **mwindow** commands display text as machine instructions.

EXAMPLES

To display lines in the list file:

`list`	*This displays a window a source lines starting from the current list location.*
`list #10 #40`	*This lists the lines in the specified line range.*
`mwindow`	*This displays a window of machine instructions.*

6.2.10 Using Control Structures

dbg allows the use of **IF, FOR, WHILE** and **DO** statements which can be used to create compound commands.

EXAMPLES

Test a value and control subsequent actions:

```
{ if ( x > 10 ) { window ; break } } in my_func
```

Declare a temporary variable and control subsequent actions:

```
int limit = 100

WHILE ( x = limit ) { x = x - 1 ; window ; break }
```

6.2.11 Getting Help

An extensive on-line help facility is part of **dbg**. This can be invoked by typing the word **HELP** after the interactive **dbg** prompt. The help facility discusses most of the concepts that are presented in the next three sections of this chapter. In addition, a list of all keywords available in **dbg** can be obtained by typing the word **KEYWORD** after the **dbg** prompt.

6.2.12 Tracking the IPU and FPU Synchronism

There are times when you want to see when a certain variable has been assigned a specific value. The line numbers may not necessarily be of any help, therefore you must be able to look at the state of the machine to determine where the IPU and FPU are executing. This is when it is necessary to keep track of the synchronism between the IPU and the FPU.

The interaction you have with **dbg** always reflects the position or operation in the IPU. You can obviously then set breakpoints (in the IPU) that stop a process when you want. But, you may not set a breakpoint in the FPU. What you can do is ask what the FPU is doing, and this is done by looking at the contents of the FPU control registers.

EXAMPLES

`$i4`	*This displays the contents of the fifth integer scalar register.*
`$iv0[.1.5]`	*This displays the contents of the first bank, second block, sixth cell vector register.*

6.2.13 Parallel Processing

There may be times when it is necessary to know when a process goes parallel and what the threads are doing. This can be examined, as shown in the following examples.

EXAMPLES

How do you when a process goes parallel?

`status`	*A line of this command indicates if a process is parallel.*

How may parallel process are there?

```
threads
```

What thread am I currently executing in?

```
thread
```

Switch the context to another thread, such as thread 2:

```
thread #2
```

dbg LANGUAGE SYNTAX

CHAPTER SEVEN

This chapter provides a detailed discussion of the elements of the **dbg** environment, such as components of the language, lexical conventions and syntax. The last part of the chapter is a table and description of all keywords that can be used in **dbg**.

7.1 Elements of the dbg Environment

The elements of the debugging environment that are discussed in this section include the basic components of the language, such as commands and expressions, lexical conventions and the general syntax of all components.

A means of interacting with the TITAN debugger is necessary once you have entered the **dbg** environment. The items that follow provide the means of communicating with **dbg** and these are commands, keywords, *WHEN* and *WHERE* clauses, declarations, symbols from the program being debugged, addresses, Fortran and C expressions, and symbols which represent abbreviations.

7.1.1 Commands

There are three types of commands that are accepted by **dbg**

- simple expression commands.
- keyword commands.
- compound commands.

Each type of command can have from one to three components and these are the command itself (action), a *WHEN* clause, and a *WHERE* clause.

The simplest **dbg** command is a C or Fortran expression, and **dbg** decides from the context used which language is intended. Any variable name is also an expression because the expression **is** the value of the variable.

A keyword command is a **dbg** keyword followed by zero or more arguments. These arguments may either appear delimited by blanks or as a parenthesized, comma separated list. Keyword commands are subdivided into nine groups based on their function, such as setting a break point.

Compound commands are constructed from simpler **dbg** commands or by combining keywords and simpler commands.

7.1.1.1 Scope

In order to understand how commands are to be entered and then interpreted by **dbg**, it is necessary to understand the concept of *scope*. All symbols from your program have a scope associated with them. The scope of the symbols are their relative location within the program. For example, a symbol might be local to a function, local to a line range within a function, or external. **dbg** can use this concept of scope with regard to your program symbols to move around in your program.

The concept of scope also applies to the program counter in **dbg**. When executing your program the scope of the program counter is a line number within a source file.

When issuing commands to **dbg**, the scope of what you are specifying must be taken into consideration and expressed correctly. You may explicitly change the scope by issuing a **SCOPE** command. The syntax for specifying any scope is

SYNTAX

SCOPE ' [*func*] : [*line #*] : [*file*] '

where

func is the name of a function.

line #
is an integer constant optionally preceded by a #.

file is the name of a source or object file.

If any two of *func, line #,* or *file* are missing from a scope specifier, **dbg** makes the most general assumptions about the missing fields. You must use a scope specifier to qualify an ambiguous name, such as **'main'argc**. If no arguments are present, **dbg** echos back the current scope. Scope specifiers must be delimited by back quotes.

7.1.1.2 WHERE and WHEN Clauses

A *WHERE* and a *WHEN* clause are associated with every command that you enter into **dbg**. A *WHERE* clause associated with a command allows you to force the command to be executed at the point specified by the *WHERE* clause. There are three forms that the *WHERE* clause can take and these are

SYNTAX

AT *address*

IN *address*

<empty>

where

address
is a representation of a memory location that **dbg** understands. There are four ways in which *addresses* may be specified:

- virtual address: an integer expression or a name
- line number: an integer expression that cannot be a virtual address or #n, where n is an integer constant
- scope specifier
- breakpoint number: n, where n is an integer constant specifying the block

WHERE clauses can not be nested.

If a **dbg** command has no *WHERE* clause, it is executed immediately and never again. This can be referred to as a *non-sticky* command. A *sticky* command is executed every time control passes from the executing code or program back to **dbg**.

You may override the "stickiness" of a command by preceding it with a *WHEN* clause. The valid forms of a *WHEN* clause are

SYNTAX

ALWAYS

ONCE

<empty>

If both the *WHERE* and *WHEN* clauses are omitted from a command, that is they are both empty, the default for the command is **ONCE**.

WHEN clauses can not be nested.

EXAMPLES

An example of a valid nesting level for a compound command with a *WHEN* and a sticky designation is:

```
if (y)  {  always  stop  in  main  }
```

An example of an invalid nesting level is:

```
if (y)  {  always  stop  }  in  main
```

Examples of a valid sticky designations are:

```
always  *pointer
always  stop at 0x401ed4
```

Examples of non-sticky designations are:

```
once stop in my_func
once stop at 182
```

7.1.1.3 Simple Expression Commands

A simple expression command is any Fortran or C expression. Any variables named in the expression are assumed to be within the current scope unless they are qualified with a scope specifier.

When a simple expression command is typed in, the value of that expression is printed on the screen by **dbg**. An assignment expression that you enter causes the assignment to actually take place and **dbg** responds accordingly. A function call in an expression you enter causes **dbg** to call the function. Any expression with a null type does not have its value printed by **dbg**. When a simple expression consists only of an integer constant, **dbg** interprets that as an alias for the previous command whose command number is equal to the value of the integer.

EXAMPLES

An assignment expression might look like this:

```
you enter        size = 82
```

dbg responds size: 82

size has now been assigned the value **82**.

A function expression might look like this:

```
you enter        10 + func(y)
```

The function **func** is called with an argument of **y**. If the result of the function is equal to **20**, then

dbg responds 30

which is equal to the result of the function call added to **10**.

7.1.1.4 Compound Commands

A compound command is a command that is created by combining simple commands or by using one of the designated keywords with simpler commands. Multiple compound statements are separated with a semicolon. There are only six ways that a compound command can be created and their forms are as follows:

SYNTAX

WHILE (*expression*) { *simpler commands* }

FOR (*expression; expression; expression*) { *simpler commands* }

IF (*expression*) { *simpler commands* } [**ELSE** { *simpler commands* }]

DO left-hand-side = *expression, expression* [, *expression*]
simpler command
simpler command
.
.
.
ENDDO

simpler command; simpler command; ...; simpler command

{ *simpler command; simpler command; ...; simpler command* }

where

expression
is any simple expression command.

simpler command
is either a simple expression command, keyword command, or another compound command.

The assignment and comparison operators that are supported by **dbg** are:

```
assignment:     =,  ++
comparison:     >,  >=,  ==,  <=,  <
```

7.1.1.5 Keyword Commands

A keyword command is a keyword followed by zero or more arguments. The format for a keyword is

SYNTAX

keyword [*arg1*] [*arg2* ...]

where

arg is any C argument.

Keywords are case insensitive, and arguments may be separated by white space or in the form of a parenthesized, comma separated list.

EXAMPLES

The following keyword commands all result in a command to continue execution to the next breakpoint, with the assumption that **i** has a value of 1.

```
cont(2-i)
CONT 2-i
continue i
CONT 1
```

7.1.1.6 Declarations

As discussed previously, you can declare a variable in **dbg** that you may use for the storage of temporary values. The scope of these variables and their values is external. The format for a declaration of this type is

SYNTAX

typename undeclared_name [*=initial value*] [, ...]

where

typename
: is the data type of the variable. Allowable data types are **char, unsigned char, short, ushort, int, integer, uint, string, logical, float, real, double, complex,** and **double complex.**

undeclared_name
is the name you give to the variable.

initial value
is the value you assign to the variable.

7.2 Keyword Commands

This section of the chapter begins the detailed definition and description of each keyword command in **dbg**. The first part contains a table of all keywords followed by a detailed syntax and description of each keyword.

Table 6-1. Statement Keywords

ACTIVATE	GRAB	RELEASE
ALWAYS	HELP	RERUN
AT	HEX	RESUME
BINARY	HISTORY	RUN
BREAK	IF	SAVE
C	IN	SCOPE
CALL	INPUT	SEND
CANCEL	INT	SET
CATCH	INTEGER	SHORT
CHAR	INTERCEPT	SIZE
COMPLEX	INTERPRET	SIZEOF
CONTINUE	KEYWORD	SOURCE
DCOMPLEX	KILL	STATUS
DEBUG	LINE	STEP
DECIMAL	LIST	STOP
DECLARE	LOG	STRING
DIS	LOGICAL	SUSPEND
DO	MNEXT	THREAD
DOUBLE	MSTEP	THREADS
DUMP	MWINDOW	TRACEBACK
ELSE	NEXT	UCHAR
ENDDO	OBJECT	UINT
EXIT	OCTAL	UNCATCH
FAVOR_KEYWORDS	ONCE	UNINTERCEPT
FAVOR_NAMES	POP	UNINTERPRET
FILE	PRINT	UNLOG
FINISH	PUSH	UNSET
FLOAT	QUIT	USHORT
FOR	RADIX	WHATIS
FORTRAN	REAL	WHERE
FUNC	REFRESH	WHILE
GOTO	REGNAMES	WINDOW

7.2.1 Keyword Descriptions

A description for each keyword follows, as well as any syntactical information.

ACTIVATE [(*blk# , cmd #)*]
ACTIVATE [*blk# cmd #*]

Control the sticky bit in anything that the **STATUS** command displays. (This is the same as the sticky bit being either inactive or active). The **ACTIVATE** command turns the sticky bit off and inactive off.

When no arguments are present, all block numbers are affected. When one argument is present, it must be the block number (*blk #*). When two arguments are present, only a specific block number and command number (*cmd #*) are affected.

ALWAYS

Make a command sticky, that is re-interpret and re-execute the command each time control passes from the code being debugged to **dbg**.

AT *address*

Specify the location of the command that must be executed. This is a form of the *WHERE* clause.

BINARY

Change the output format for integer type values from decimal to binary. To have the output format displayed in another radix and leave binary, type the new radix name, such as **DECIMAL**.

BREAK

Implement a simple breakpoint. This is synonymous with **STOP**.

C

Tell **dbg** that you are in a C program, not Fortran. This is not necessary if you are currently in a C program; the debugger knows which language you are working in.

CALL *name* [([*arg1*[,*arg2*...]])]

Call subroutine *name* with arguments *arg1*, *arg2*.... This is the same as the Fortran **CALL** statement. Refer to the *Fortran Reference Manual* for details of the semantics.

CANCEL [(*blk#* , *cmd #*)]
CANCEL [*blk#* *cmd #*]

The **CANCEL** command removes the command(s) and it can not be retrieved.

When no arguments are present, all commands in all block numbers are affected. When one argument is present, all commands within that block number (*blk #*) are affected. When two arguments are present, only a specific block number and command number (*cmd #*) are affected.

CATCH [*signo ,signo...*]

Catch the designated signal, *signo*, and stop before the process running sees the signal. The signal is specified as a number. The default it to catch almost all signals. Use **UNCATCH** to let a signal be recognized by the process. If no arguments are present, the **CATCH** command list the set of signals that may be caught by **dbg**.

CHAR

Declare variable(s) to be of type char. This may also be used for typecasting in C.

COMPLEX

Declare variable(s) to be of type complex.

CONTINUE [*n*]

The code being debugged is continued past the next *n* - 1 breakpoints. If no arguments are present, the code runs without stopping until it reaches the next break point or *where* command point. The default for *n* is 1.

DCOMPLEX

Declare variable(s) to be of type double complex.

DEBUG [*"a.out" "core"*]

Make *a.out* or *core* the file being debugged. If no arguments are present, the name of the program currently being debugged is displayed.

DECIMAL

Change the output format for integer type values from another radix to decimal. To have the output format displayed in another radix and leave decimal, type the new radix name, such as **BINARY**.

DECLARE [*datatype varname*]

Declare a variable to exist. This is not needed because you may simply type in the data type followed by a variable name.

DIS [*address winlength*]

Disassemble; print the contents of memory following *address* for a window length of *winlength*. If no arguments are present, **DIS** uses the last address of a **DIS** command that was printed out. If there was not a previous **DIS**, the value

of the program counter is used. The default window length is 16 words.

When only one argument is present, **DIS** tries to assume that the argument is an address. However, if there is no way that the argument is an address, **DIS** uses the argument as a window length and takes the appropriate default *address*.

When the *address* expressed is a line number, it must be preceded by a pound sign (#). When the *address* expressed is a block number, it must be preceded by an at sign (@). Addresses must be word aligned.

DO *args* [*args*] **ENDDO**

Perform a Fortran **DO** loop. This multi-line command is the same as a Fortran **DO** loop, with no statement numbers. You type the first **DO** line in and then **dbg** prompts you for more input until you type in an **ENDDO** statement.

DOUBLE

Declare variable(s) to be of type double.

DUMP *address* [*n*]
DUMP [*n*]

When *address* is present, dump the first *n* words at address *address*. When no address is present, dump the first *n* words starting at the last address displayed by **DIS, WINDOW** or **DUMP**. This command displays words in both numeric and ASCII.

ELSE

Perform the **else** part of a C **if** statement.

ENDDO

Terminating statement of the **DO** command.

EXIT

Exit from **dbg**. This is synonymous with **QUIT**.

FAVOR_KEYWORDS

Resolve a conflict between a keyword and a source name in favor of the keywords. Preceding a name with a pound sign (#) resolves the conflict between the name and a keyword.

FAVOR_NAMES

Resolve a conflict between a keyword and a source name in favor of the source name. Preceding a name with a pound

sign (#) resolves the conflict between the name and a keyword.

FILE ["*file*"]

Set the current scope to the specified file. You must use a scope specifier to qualify an ambiguous name, such as **'main'argc**. If no arguments are present, **dbg** echos back the current scope. The **FILE** command sets both the variable and list scope to the same value.

FINISH

End the **dbg** session. This is synonymous with **QUIT** and **EXIT**.

FLOAT

Declare variable(s) to be of type float.

FOR

Perform a C **for** statement. The commands associated with the **FOR** statement must be in braces and no semicolon can appear after the last statement.

FORTRAN

Tell **dbg** that you are in a Fortran program, not C. This is not necessary if you are currently in a Fortran program; the debugger knows which language you are working in.

FUNC [["]*func* ["]]

Set the current scope to the specified function. You must use a scope specifier to qualify an ambiguous name, such as **"main"argc**. If no arguments are present, **dbg** echos back the current scope.

GOTO *address*

Perform a Fortran **GOTO** statement. You must **GOTO** an address and never a label.

GRAB *pid*

Grab the currently running process with process id *pid* and make it the process currently being debugged. **dbg** automatically releases all processes that were grabbed when you exit.

HELP [*keyword*]

Print a useful description of how to use the given keyword. With no arguments, a menu of options is presented.

HEX

Change the output format for integer type values from another radix to hexadecimal. To have the output format displayed in another radix and leave hex, type the new radix name, such as **BINARY**. Be aware that all input is now hexadecimal, as well as output.

HISTORY [*n*]

Print a list of the past *n* issued commands. The default for *n* is equal to 15.

IF

Perform a C **if** statement.

IN *address*

Specify the location of the command that must be executed. This is a form of the *WHERE* clause.

INPUT *"filename"*

Use *filename* as the next stream of input commands. On receipt of EOF, resume input from the terminal. This must be an ASCII file.

INT

Declare variable(s) to be of type int.

INTEGER

Declare variable(s) to be of type integer.

INTERCEPT [*signo*]

Do not let the running process see the signal, *signo*. The default is to intercept all signals.

INTERPRET

Interpret all machine level instructions one at a time, showing address and registers for loads and stores. This command allows you to obtain additional information while advancing the PC at assembly level.

KEYWORD

Display a list of all **dbg** keywords.

KILL

Kill the currently running process.

LINE [["]*lineno* ["]]
LINE [*#lineno*]

Set the current scope to the specified line number. You must use a scope specifier to qualify an ambiguous name, such as **"main"argc**. If no arguments are present, **dbg** echos back the current scope.

LIST [*address count*]

Display a portion of the source file. If no arguments are present, the **LIST** command displays a window of lines in the list file starting from the current list location. The starting location, *address* can be an address, a procedure name or a scope specifier. *count* specifies the number of lines to display. If *count* is the first argument, the list starts from the current list location. The **LIST** command can also be used to set the current list scope.

LOG *"filename"*

Start a log of the current session in file *filename*.

LOGICAL

Declare variable(s) to be of type logical.

MNEXT [*n*]

Step through the next *n* machine instructions. **MNEXT** does not enter functions or subroutines.

MSTEP [*n*]

Go to the next *n* basic blocks at machine level.

MWINDOW [*address winlength*]

Display a window of machine instructions centered around a program location. Refer to **WINDOW** keyword for further syntactic details.

NEXT [*n*]

Step through the next *n* source level instructions. **NEXT** does not enter functions or subroutines.

OBJECT " *obj1* [: *obj2*] "

Find object file(s) and pass them to **dbg**. The syntax for *obj* is the same as a UNIX pathname.

OCTAL

Change the output format for integer type values from

another radix to octal. To have the output format displayed in another radix and leave octal, type the new radix name, such as **BINARY**. Be aware that all input is now octal, as well as output.

ONCE

Make a command non-sticky, that is immediately execute the command and then never again.

POP [*n*]

Pop the top *n* instructions from the stack if there is an active stack or coredump. This resets the scope. The default for *n* is 1. This is used in conjunction with **PUSH** and **TRACEBACK**.

PRINT *var*

Prints the variable(s) in *var*.

PUSH [*n*]

Push the next *n* routines if there is an active stack or coredump. This resets the scope. The default for *n* is 1. This is used in conjunction with **POP** and **TRACEBACK**.

QUIT

End the **dbg** session. This is synonymous with **EXIT** and **FINISH**.

RADIX

Display the current radix, that is, hexadecimal, decimal, and so on. To enter a constant in a radix other than the default radix, precede the constant with:

```
0b  - for binary
0o  - for octal
0d  - for decimal
0x  - for hexadecimal
```

REAL

Declare variable(s) to be of type real.

REFRESH

Flush the cache that is used by **dbg**.

REGNAMES [0, 1 or 2]

Use the defined registers. **0** looks at the operating system register names. **1** looks at the hardware register names. **2**

looks at the assembler register names. The default is **2**. This command is followed by a **$** command which prints the contents of the named registers. When the **$** command is invoked with no arguments it prints the contents of the MIPS general purpose registers.

RELEASE [*pid*]

Release the process, *pid*. **dbg** must be able to find the *a.out* file to perform this. The debugger automatically releases all grabbed processes upon exit. The default is to release the current process.

RERUN

Run the code being debugged with the most recent previously issued command line parameters.

RESUME [(*blk# , cmd #*)]
RESUME [*blk# cmd #*]

Control the sticky bit in anything that the **STATUS** command displays. (This is the same as the sticky bit being either inactive or active). The **RESUME** command turns the sticky bit on and inactive off.

When no arguments are present, all block numbers are affected. When one argument is present, it must be the block number (*blk #*). When two arguments are present, only a specific block number and command number (*cmd #*) are affected.

RUN [*argv*] [<*new stdin*] [>*new stdout*]

Run the code to be debugged with the corresponding parameters.

SAVE " *filename*"

Dump a core image of the current process into *filename*.

SCOPE ' [*func*] | [*line #*] : [*file*] '

Set the current scope. If any two of *func, line #,* or *file* are missing from a scope specifier, **dbg** makes the most general assumptions about the missing fields. You must use a scope specifier to qualify an ambiguous name, such as **'main'argc**. If no arguments are present, **dbg** echos back the current scope.

SEND *signo*

Send a signal to the running process. This sets the pending signal mask.

SET *command* [= *value*]

Set the default window length in the **WINDOW** or **DIS** command. *command* is either the **WINDOW** or **DIS** keyword. The original default is 16 words, which can be changed with *value*.

SHORT

Declare variable(s) to be of type short.

SIZE

Print the text, data and stack size parameters of the code being debugged. This is the size of *a.out*.

SIZEOF *arg*

Perform the C **sizeof** function.

SOURCE " *file1* [: *file2*] "

Find source file(s) and pass them to **dbg**. The syntax for *file* is the same as a UNIX pathname.

STATUS

Print the current thread and scope of the running process. This also lists all pending actions by block number. Block #0 always refers to sticky commands (those not associated with a *WHERE* clause). All other blocks refer to non-sticky commands.

The outer numbers in the **STATUS** display are block numbers. Each block number corresponds to an instruction address at which an action has been specified. The numbers under each block are arbitrary command numbers and only refer to the temporary time they were entered. These command numbers change with the addition, cancellation or deletion of commands.

STEP [*n*]

Go to the next basic block at source level.

STOP

Implement a simple break point. This is synonymous with **BREAK**.

STRING

Declare variable(s) to be of type string. This is a **dbg** data type and it is a pointer to a character. This prints the character string rather than just the value.

SUSPEND [(*blk# , cmd #)*]
SUSPEND [*blk# cmd #*]

Control the sticky bit in anything that the **STATUS** command displays. (This is the same as the sticky bit being either inactive or active). The **SUSPEND** command turns the inactive on and suspends sticky commands.

When no arguments are present, all block numbers are affected. When one argument is present, it must be the block number (*blk #*). When two arguments are present, only a specific block number and command number (*cmd #*) are affected.

THREAD [*number*]

Display the parallel thread that **dbg** is looking at now. When a thread number, *number*, is present, **dbg** switches context to that thread.

THREADS

This tells you how many parallel process there are.

TRACEBACK

Print a stack trace. This command displays a scope string and a PC.

UCHAR

Declare variable(s) to be an unsigned character.

UINT

Declare variable(s) to be an unsigned integer.

UNCATCH [*signo*]

Do not catch signal *signo*. Let the signal be seen by the running process. If no arguments are present, the **UNCATCH** command list the set of signals that may be uncaught by **dbg**.

UNINTERCEPT

Allow the running process to see the interrupt.

UNINTERPRET

Turn off instruction interpreting.

UNLOG

Suspend logging the current session.

UNSET *command*

Reset the default window length in the **WINDOW** or **DIS** *command* to be 16 words.

USHORT

Declare variable(s) to be an unsigned short.

WHATIS *var*

Print type information for *var*.

WHERE

Print a stack trace. Synonymous with **TRACEBACK**. These commands display a scope string and a PC.

WHILE

Perform a C **while** loop. The loop must be in brackets and there can be no semicolon after the last statement.

WINDOW [*address winlength*]

Print the source lines for *winlength*/2 words above and *winlength*/2 words below *address*. If no arguments are present, **WINDOW** uses the last address of a **WINDOW** command that was printed out. If there was not a previous **WINDOW** command, the value of the program counter is used. The default window length is 16 words.

When only one argument is present, **WINDOW** tries to assume that the argument is an address. However, if there is no way that the argument is an address, **WINDOW** uses the argument as a window length and takes the appropriate default *address*.

When the *address* expressed is a line number, it must be preceded by a pound sign (#). When the *address* expressed is a block number, it must be preceded by an at sign (@). Addresses must be word aligned.

$ [*regno*]

Print the contents of the MIPS general purpose registers when no argument is present, else print the contents of the named register. The following format for *regno* addresses the FPU registers:

scalar integer:	$i0-31
scalar real:	$f0-31
scalar double:	$d0-31
vector:	$iv0-3[. [0-7] . [0-31]]

accumulators: $ia0-3
$fa0-3

 # source name

Precede the *source name* with # when you want to force the recognition of a source name rather than a **dbg** keyword for just one command, without changing the default behavior.

TUNING AND PORTING CODE

CHAPTER EIGHT

This chapter shows you how to use the profiler to determine where your program is spending the most time. It also describes some of the issues involved in porting your code from another system to the TITAN.

8.1 Tuning Code

With the information produced from the profiler, you can determine if you want to either change your source code or use compiler directives to modify slow code for faster execution. When the profiler is invoked by using either the **prof** or **mkprof** command, each routine in your program is interpreted. You may also instruct the profiler to interpret each loop within a routine by specifying the **-ploop** command line compiler option.

8.1.1 prof and mkprof

Five columns of information are produced from profiling your program (with the **-v** option) with either the **prof** command or the **mkprof** command. Three columns of information are produced if **-v** is not specified. Refer to the *Commands Reference Manual* for the syntax and detailed descriptions of both commands.

The information is presented in the order of highest usage. The first column, **count**, represents the number of times the routine is entered. The second column, **microseconds**, is exactly that, the total number of microseconds that were spent in the routine. The third column, **%**, is the percentage of total time that was spent in the routine. The fourth column, **time/call**, is the number of microseconds it took to execute each call to the routine. The fifth column is the name of the procedure.

The first step in tuning your code should be to take the top routines that are listed and determine whether they are I/O or compute bound. From here, you can adjust the code manually. You may see user routines that are listed as having very high usage. At this point it might be helpful to compile with either **-vreport** or **-full_report** to determine if the efficiency of these routines can be increased. Refer to *Chapter 2, Efficient Programming Techniques* for specific information.

EXAMPLE

The following is an example of a short Fortran program, with its output. This program was compiled with the **-p** option. Then the *a.out* file was run, in order to produce the *mon.out* file needed by the **prof** command.

```
PROGRAM CHAR
CHARACTER*5 char_array(5)
DATA (char_array(1) ,i=1,5)/5*'xxxxx'/
WRITE (6,'(A)') (char_array(1) (1:i), i=1,5)
END
```

The following commands were issued to compile the program and to produce the output, and to create the *mon.out* file needed to run **prof**.

```
fc  -p  prog.f

a.out
```

The output from the program is:

```
x
xx
xxx
xxxx
xxxxx
```

Following is the output produced from running the **prof** command with the **-v** option on the *a.out* file from the previous program. The command issued was

```
prof  -v  a.out
```

```
count   microsec.     %   time/call   NAME (1 proc.)
   10         1114  12.1         111    _IO_Setup_Format_Item
    1          827   9.0         827    _IO_Initialize_FORTRAN_IO
    5          756   8.2         151    malloc
    1          467   5.1         467    _IO_Format_Write_Driver
    6          450   4.9          75    _IO_Acquire_Next_Data_Item
   20          392   4.3          19    _flsbuf
   15          334   3.6          22    _IO_Write_External_Putc
    5          289   3.1          57    calloc
    6          277   3.0          46    Increment_List_Loc
    5          272   3.0          54    _IO_Write_A_Format
    1          243   2.6         243    _IO_Write_Fmt_Seq
    5          204   2.2          40    Write_Data_Format
    3          199   2.2          66    _IO_Allocate_Unit
    1          191   2.1         191    _IO_Translate_Format_String
    5          182   2.0          36    _IO_Write_External_Slash
    6          169   1.8          28    Write_Non_Data_Formats
    4          158   1.7          39    isatty
    4          155   1.7          38    ioctl
    5          148   1.6          29    write
    1          147   1.6         147    _IO_Check_Seq_Fmt_Ext
    3          144   1.6          48    _IO_Can_This_File_Seek
    1          142   1.5         142    _IO_Finish_FORTRAN_IO
    7          133   1.4          19    _IO_Find_Unit
    1          129   1.4         129    Setup_Implicit_Do
    1          123   1.3         123    _IO_Initialize_Format
    1          113   1.2         113    Parse_Format_Specification
    3          109   1.2          36    fstat
    6          108   1.2          18    _IO_Is_Data_Transmitter
    5          103   1.1          20    _xflsbuf
    2          103   1.1          51    sbrk
    3           93   1.0          31    _IO_Close
    5           91   1.0          18    _IO_Translate_CC_Putc
    1           88   1.0          88    Translate_Outer_Parens
    5           82   0.9          16    _IO_Fill_With_Char
    1           77   0.8          77    Evaluate_Parameters
    3           76   0.8          25    Allocate_Format_Item
    5           72   0.8          14    Strip_Leading_White_Space
    4           68   0.7          17    _IO_Unit_Walk
    1           48   0.5          48    Increase_Format_Buffer
    1           42   0.5          42    _start
    1           41   0.4          41    _IO_Initialize_Unit_Walk
    1           35   0.4          35    Parse_AL_Format
    3           34   0.4          11    _IO_Change_Unit_Number
    1           32   0.3          32    _findbuf
    2           28   0.3          14    fflush
    5           28   0.3           5    __fort_program$0
    1           26   0.3          26    _wrtchk
    1           24   0.3          24    _IO_Initialize_Data_List_Reader
    1           16   0.2          16    _fort_program
    1            0   0.0           0    exit
```

8.1.2 -ploop

The **-ploop** compile time option allows loops within a single routine to be profiled separately, and may be invoked from Fortran or C. This can help you to then determine which loops dominate the execution time of a routine.

When this option is invoked at compile time, the compiler inserts labels into the generated code before and after all loop nests. The labels have the form

```
_$lp_InLoop_ . . .
```

and

```
_$lp_AfterLoop_ . . .
```

where the . . . indicates a label that identifies the source routine and the line number of the loop within that routine.

Use of this option causes the body of the loop to appear as a function, **_$lp_InLoop_ . . .**, and the code following the loop to appear as **_$lp_AfterLoop_ . . .**.

EXAMPLE

The following is an example of a Fortran program which uses the **-ploop** option in compilation.

```
program main
double precision a(100,100), b(100,100)
data n/100/, m/100/
data b/10000 * 1.0d0/
if (m .eq. n) then
   do i = 1, n
      do j = 1, n
          a(i,j) = a(i,j) * b(i,j)
      enddo
   enddo
else
   do i = 1, n
       do j = 1, m
          a(i,j) = a(i,j) / b(i,j)
       enddo
   enddo
endif
if (m .ne. n) then
   do i = 1, n
      do j = 1, n
          a(i,j) = a(i,j) * b(i,j)
```

```
      enddo
   enddo
else
   do i = 1, n
      do j = 1, m
         a(i,j) = a(i,j) / b(i,j)
      enddo
   enddo
endif
end
```

The following commands were issued to compile the program, create the profile file and run the profiler.

```
fc -O2 -ploop prog.f
mkprof a.out x.out
x.out
prof -v x.out
```

Following is the output produced from running the **prof** command with the **-v** option.

```
count    microsec.     %    time/call    NAME (1 proc.)
   1        20493   80.4      20493     _$lp_InLoop_main_program_25
   1         2111    8.3       2111     _$lp_InLoop_main_program_6
  31          591    2.3         19     sigset
   1          450    1.8        450     _IO_Initialize_FORTRAN_IO
   3          396    1.6        132     malloc
   3          180    0.7         60     _IO_Allocate_Unit
   3          170    0.7         56     _IO_Close
   2          150    0.6         75     sbrk
   3          145    0.6         48     calloc
   6          112    0.4         18     _IO_Find_Unit
   1          104    0.4        104     _IO_Finish_FORTRAN_IO
   3          103    0.4         34     fstat
   3           96    0.4         32     _IO_Can_This_File_Seek
   3           91    0.4         30     isatty
   3           79    0.3         26     ioctl
   4           59    0.2         14     _IO_Unit_Walk
   3           43    0.2         14     _IO_Change_Unit_Number
   1           36    0.1         36     _start
   2           35    0.1         17     fflush
   1           34    0.1         34     _IO_Initialize_Unit_Walk
   1           14    0.1         14     _fort_program
   1            9    0.0          9     _$lp_AfterLoop_main_program_6
   1            0    0.0          0     exit
```

The profile results show that the loop starting at line 25 of the program takes the most time. It also show that the loop at line 6 takes some time, and the sequential code between the loop starting at line 6 and the one at line 25 takes almost no time.

Loops are defined as **DO** and **DO WHILE** statements in Fortran. In C they are defined as **while () do, do while()**, and **for()** statements.

8.2 Porting Code

This section describes differences between the TITAN Fortran and C languages and other implementations of the languages. If you wish to port your existing programs to run under TITAN Fortran and TITAN C, this section shows where incompatibilities may arise and source code might have to be changed.

8.2.1 Fortran Considerations

TITAN Fortran is VAX/VMS source language compatible and compatible with Cray compiler directives. Following are some items that may require you to change your code if there are problems with compilation or execution:

- Mixing of integers and floating point data types, specifically double precision floating points. Double precision word types must start on even number 32 bit boundaries.

  This may be a problem in labelled COMMON blocks if integers are mixed with reals, and so on. You may get a floating point exception error if the data is not correctly aligned.

  This may also be a problem in dynamically allocated arrays. If REALs and integers are mixed, you must make sure that the starting points are correct. The starting point must be on an even word boundary. Additionally, it is better to declare all working arrays as double precision, because they process faster. It is always better to separate integers and REALs into separate arrays.

- The data type INTEGER*8 is not supported.

- Hollerith strings may be handled differently than you are used to. Be sure to check the *Fortran Reference Manual* for proper usage.

- Carriage control characters are VAX/VMS compatible. Check the *Fortran Reference Manual* for the exact specification.

- Refer to the *Experienced Programmer's Quick Start* for all VAX/VMS and TITAN extensions to the language.

8.2.2 Machine considerations

- Refer to the section *Data Layout in Memory* in *Appendix A* for any concerns you have about machine precision.
- On the TITAN there are 32 bits per numeric storage unit, 4K bytes per block, 1K bytes per sector, and 4 bytes per word.
- Machine dependent (hardware dependent) code may have to be altered.

8.2.3 Operating System Considerations

- Date functions may have to be converted to the **DATE** or **IDATE** Fortran functions.
- To compute CPU seconds, use the **CPUTIM** function in Fortran.
- Use the **TIME** function for wall clock time in Fortran.
- Use the **RAN** function for generating random numbers in Fortran.

COMPILER ASSEMBLY INTERFACE

APPENDIX A

This appendix contains information that may be useful for system programmers. The topics discussed in this appendix are register sets, floating point computations, the TITAN stack frame, calling subprograms, and data layout in memory.

A.1 Register Sets

The following section describes the CPU, scalar, and vector register sets. Each processor in the TITAN system contains its own set of CPU registers, scalar registers, and vector registers. The scalar registers are simply selected members of the vector register set. The use of these registers is described in this chapter.

A.1.1 CPU Registers

Table A-1 gives the names and functions of the CPU registers.

Table A-1. CPU Registers

Register	Function
$0	Always 0
$at	Assembler temporary (scratch)
$v0	Function return value (int, pointers)
$v1	Scratch register (caller saves)
$a0–$a3	First four function parameters
$t0–$t7	Scratch registers (caller saves)
$s0–$s7	Register variables (callee saves)
$t8–$t9	Reserved for code generator
$h0–$h1	Reserved for kernel use
$gp	Global pointer
$sp	Stack pointer
$fp	Reserved register (used for profiling)
$ra	Return address

A.1.2 Scalar Registers

Table A-2 gives the names and functions of the scalar floating point registers.

Table A-2. Scalar Floating Point Registers

Register	Function
F0	Real/double/complex function return
F1	Complex function return
F2–F3	Scratch registers
F4–F7	First four floating point input arguments
F8–F11	Scratch registers (caller saves)
F12–F15	Register variables (callee saves)
F16–F19	Scratch registers (caller saves)
F20–F23	Register variables (callee saves)
F24–F27	Scratch registers (caller saves)
F28–F31	Register variables (callee saves)

The *F* in the register names in Table A-2 means the register is holding a single precision floating point quantity. It may be replaced with one of the following builtin names: **D** or **d**, if the register is holding a double-precision floating point quantity; or **I** or **i**, if the register is holding an integer quantity.

In addition, there are four accumulator registers indicated by the builtin name **A** or **a**.

A.1.3 Vector Registers

Vector registers are arranged in four symmetric and interchangeable banks. There are 1024 vector registers available to each user process for working storage. Each bank may be thought of as containing eight vector blocks, 0 through 7, each containing 32 registers, 0 through 31.

Block 0 of each bank (which includes the scalar registers) is reserved for system use. Table A-3 gives the general scheme for usage of the first 32 vector registers in each bank.

Most vector operations take place on vector blocks, but this is not necessary. Vector operations may begin at any register and continue upwards within the same bank. A vector block is

Table A-3. Vector Registers

Registers	Use
0–7	Scalar registers
8–15	Scratch and constants (used by the compiler)
16–31	Graphics

designated V*b.n* and a particular register within that block is designated V*b.n.m*. A particular register within a bank may be referred to by

The other vector registers are assumed to be scratch registers, although library functions may use other conventions.

A.2 Floating Point Computations

The TITAN architecture permits floating point computations to operate in parallel with the integer processor. Special instructions are used to synchronize the floating point and integer processors at key points in the computation, notably conditional branches and function returns. When a value is computed into a scalar or vector register and then used by a later instruction, the second instruction stalls until the needed value is available. However, because loads from and stores to memory may be proceeding in parallel and take complicated forms, avoiding conflicts in memory is the software's responsibility.

The following summary lists the actions taken by the software to avoid memory conflicts:

- All functions and subroutines must wait for floating point memory stores to finish before returning.
- Functions that return floating point values must complete all floating point memory stores, but may return while the computation of the return value is in progress.
- All loads from the stack must be completed before returning from the function.
- Programs must wait for values to be stored before loading them into integer or vector registers.

The compiler automatically generates code to ensure that these conditions are satisfied.

Table A-4. FPU Instructions, by Mnemonic

Mnemonic	Parameters	Location	Comments
d1	AC	0xffff9f00	Accumulator Op 16
dabs	Dx, Dz	0xffff9c10	Scalar 36, Unconditional
dabs,c	Dx, Dz	0xffff9c90	Scalar 36, Nullify if StStack<0> clear
dadd	Aw, Dy, Dz	0xffff9840	Scalar 64, Unconditional
dadd	Aw, Dz, Aw	0xffff9440	Scalar 48, Unconditional
dadd	Dx, Dy, Dz	0xffff9800	Scalar 16, Unconditional
dadd	Dx, Dz, Aw	0xffff9c40	Scalar 80, Unconditional
dadd,c	Aw, Dy, Dz	0xffff98c0	Scalar 64, Nullify if StStack<0> clear
dadd,c	Aw, Dz, Aw	0xffff94c0	Scalar 48, Nullify if StStack<0> clear
dadd,c	Dx, Dy, Dz	0xffff9880	Scalar 16, Nullify if StStack<0> clear
dadd,c	Dx, Dz, Aw	0xffff9cc0	Scalar 80, Nullify if StStack<0> clear
damax	Dx, Dy, Dz	0xffff9c70	Scalar 92, Unconditional
damax,c	Dx, Dy, Dz	0xffff9cf0	Scalar 92, Nullify if StStack<0> clear
dceq	Dy, Dz	0xffff9170	Scalar CMC, Set Mask, Cond=3, Op=4
dcge	Dy, Dz	0xffff9150	Scalar CMC, Set Mask, Cond=2, Op=4
dcgt	Dy, Dz	0xffff9130	Scalar CMC, Set Mask, Cond=1, Op=4
dcle	Dy, Dz	0xffff91b0	Scalar CMC, Set Mask, Cond=5, Op=4
dclt	Dy, Dz	0xffff91d0	Scalar CMC, Set Mask, Cond=6, Op=4
dcne	Dy, Dz	0xffff9190	Scalar CMC, Set Mask, Cond=4, Op=4
dcun	Dy, Dz	0xffff9110	Scalar CMC, Set Mask, Cond=0, Op=4
ddiv	Aw, Dy, Dz	0xffff9858	Scalar 70, Unconditional
ddiv	Dx, Dy, Dz	0xffff9818	Scalar 22, Unconditional
dhalf	AC	0xffff9f20	Accumulator Op 24
dlog10	AC	0xffff9f38	Accumulator Op 30
dloge	AC	0xffff9f30	Accumulator Op 28
dm1	AC	0xffff9f08	Accumulator Op 18
dma	Dx, Dy, Dz, Aw	0xffff9400	Scalar 0, Unconditional
dma,c	Dx, Dy, Dz, Aw	0xffff9480	Scalar 0, Nullify if StStack<0> clear
dma	Dx, Dy, Aw, Dz	0xffff9410	Scalar 4, Unconditional
dma,c	Dx, Dy, Aw, Dz	0xffff9490	Scalar 4, Nullify if StStack<0> clear
dmax	Dx, Dy, Dz	0xffff9428	Scalar 10, Unconditional
dmax,c	Dx, Dy, Dz	0xffff94a8	Scalar 10, Nullify if StStack<0> clear
dmin	Dx, Dy, Dz	0xffff9420	Scalar 8, Unconditional
dmin,c	Dx, Dy, Dz	0xffff94a0	Scalar 8, Nullify if StStack<0> clear
dmove	Dx, Dz	0xffff9020	Scalar CMC, NoSetMsk, Cond=1, Op=0

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
dms	Dx, Dy, Aw, Dz	0xffff9418	Scalar 6, Unconditional
dms	Dx, Dy, Dz, Aw	0xffff9c00	Scalar 32, Unconditional
dms,c	Dx, Dy, Aw, Dz	0xffff9498	Scalar 6, Nullify if StStack<0> clear
dms,c	Dx, Dy, Dz, Aw	0xffff9c80	Scalar 32, Nullify if StStack<0> clear
dmul	Aw, Dy, Dz	0xffff9850	Scalar 68, Unconditional
dmul	Aw, Dz, Aw	0xffff9450	Scalar 52, Unconditional
dmul	Dx, Dy, Dz	0xffff9810	Scalar 20, Unconditional
dmul	Dx, Dz, Aw	0xffff9c50	Scalar 84, Unconditional
dmul,c	Aw, Dy, Dz	0xffff98d0	Scalar 68, Nullify if StStack<0> clear
dmul,c	Aw, Dz, Aw	0xffff94d0	Scalar 52, Nullify if StStack<0> clear
dmul,c	Dx, Dy, Dz	0xffff9890	Scalar 20, Nullify if StStack<0> clear
dmul,c	Dx, Dz, Aw	0xffff9cd0	Scalar 84, Nullify if StStack<0> clear
dneg	Dx, Dz	0xffff9c18	Scalar 38, Unconditional
dneg,c	Dx, Dz	0xffff9c98	Scalar 38, Nullify if StStack<0> clear
dninf	AC	0xffff9f18	Accumulator Op 22
dpinf	AC	0xffff9f10	Accumulator Op 20
dradabs	AC, Vy	0xffff9990	Vec.Reduc. 20, Non-Masked
dradabs,mf	AC, Vy	0xffff99d0	Vec.Reduc. 68, Use /Mask
dradabs,mt	AC, Vy	0xffff9950	Vec.Reduc. 68, Use Mask
dradd	AC, Vy	0xffff9980	Vec.Reduc. 16, Non-Masked
dradd,mf	AC, Vy	0xffff99c0	Vec.Reduc. 64, Use /Mask
dradd,mt	AC, Vy	0xffff9940	Vec.Reduc. 64, Use Mask
drma	AC, Vy, AC, Vz	0xffff9590	Vec.Reduc. 4, Non-Masked
drma	AC, Vy, Vz, AC	0xffff9580	Vec.Reduc. 0, Non-Masked
drma	AC, Vy, [Vz], AC	0xffff9998	Vec.Reduc. 22, Non-Masked
drma,mf	AC, Vy, AC, Vz	0xffff95d0	Vec.Reduc. 52, Use /Mask
drma,mf	AC, Vy, Vz, AC	0xffff95c0	Vec.Reduc. 48, Use /Mask
drma,mf	AC, Vy, [Vz], AC	0xffff99d8	Vec.Reduc. 70, Use /Mask
drma,mt	AC, Vy, AC, Vz	0xffff9550	Vec.Reduc. 52, Use Mask
drma,mt	AC, Vy, Vz, AC	0xffff9540	Vec.Reduc. 48, Use Mask
drma,mt	AC, Vy, [Vz], AC	0xffff9958	Vec.Reduc. 70, Use Mask
drmax	AC, Vy	0xffff95a8	Vec.Reduc. 10, Non-Masked
drmax,mf	AC, Vy	0xffff95e8	Vec.Reduc. 58, Use /Mask
drmax,mt	AC, Vy	0xffff9568	Vec.Reduc. 58, Use Mask
drmin	AC, Vy	0xffff95a0	Vec.Reduc. 8, Non-Masked
drmin,mf	AC, Vy	0xffff95e0	Vec.Reduc. 56, Use /Mask
drmin,mt	AC, Vy	0xffff9560	Vec.Reduc. 56, Use Mask
drms	AC, Vy, AC, Vz	0xffff9598	Vec.Reduc. 6, Non-Masked
drms	AC, Vy, Vz, AC	0xffff9d80	Vec.Reduc. 32, Non-Masked
drms,mf	AC, Vy, AC, Vz	0xffff95d8	Vec.Reduc. 54, Use /Mask
drms,mf	AC, Vy, Vz, AC	0xffff9dc0	Vec.Reduc. 80, Use /Mask
drms,mt	AC, Vy, AC, Vz	0xffff9558	Vec.Reduc. 54, Use Mask
drms,mt	AC, Vy, Vz, AC	0xffff9d40	Vec.Reduc. 80, Use Mask
drmxabs	AC, Vy	0xffff9988	Vec.Reduc. 18, Non-Masked
drmxabs,mf	AC, Vy	0xffff99c8	Vec.Reduc. 66, Use /Mask
drmxabs,mt	AC, Vy	0xffff9948	Vec.Reduc. 66, Use Mask
drpi	AC	0xffff9f28	Accumulator Op 26

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
drsm	AC, AC, Vz, Vy	0xffff9588	Vec.Reduc. 2, Non-Masked
drsm	AC, Vz, Vy, AC	0xffff9d88	Vec.Reduc. 34, Non-Masked
drsm,mf	AC, AC, Vz, Vy	0xffff95c8	Vec.Reduc. 50, Use /Mask
drsm,mf	AC, Vz, Vy, AC	0xffff9dc8	Vec.Reduc. 82, Use /Mask
drsm,mt	AC, AC, Vz, Vy	0xffff9548	Vec.Reduc. 50, Use Mask
drsm,mt	AC, Vz, Vy, AC	0xffff9d48	Vec.Reduc. 82, Use Mask
dsm	Dx, Aw, Dy, Dz	0xffff9408	Scalar 2, Unconditional
dsm	Dx, Dz, Dy, Aw	0xffff9c08	Scalar 34, Unconditional
dsm,c	Dx, Aw, Dy, Dz	0xffff9488	Scalar 2, Nullify if StStack<0> clear
dsm,c	Dx, Dz, Dy, Aw	0xffff9c88	Scalar 34, Nullify if StStack<0> clear
dsub	Aw, Aw, Dz	0xffff9448	Scalar 50, Unconditional
dsub	Aw, Dy, Aw	0xffff9458	Scalar 54, Unconditional
dsub	Aw, Dy, Dz	0xffff9848	Scalar 66, Unconditional
dsub	Dx, Aw, Dz	0xffff9c48	Scalar 82, Unconditional
dsub	Dx, Dy, Aw	0xffff9c58	Scalar 86, Unconditional
dsub	Dx, Dy, Dz	0xffff9808	Scalar 18, Unconditional
dsub,c	Aw, Aw, Dz	0xffff94c8	Scalar 50, Nullify if StStack<0> clear
dsub,c	Aw, Dy, Aw	0xffff94d8	Scalar 54, Nullify if StStack<0> clear
dsub,c	Aw, Dy, Dz	0xffff98c8	Scalar 66, Nullify if StStack<0> clear
dsub,c	Dx, Aw, Dz	0xffff9cc8	Scalar 82, Nullify if StStack<0> clear
dsub,c	Dx, Dy, Aw	0xffff9cd8	Scalar 86, Nullify if StStack<0> clear
dsub,c	Dx, Dy, Dz	0xffff9888	Scalar 18, Nullify if StStack<0> clear
dtdnm	Dz	0xffff9128	Scalar CMC, Set Mask, Cond=1, Op=2
dteq	Dy	0xffff9160	Scalar CMC, Set Mask, Cond=3, Op=0
dtge	Dy	0xffff9140	Scalar CMC, Set Mask, Cond=2, Op=0
dtgt	Dy	0xffff9120	Scalar CMC, Set Mask, Cond=1, Op=0
dtinf	Dz	0xffff9108	Scalar CMC, Set Mask, Cond=0, Op=2
dtle	Dy	0xffff91a0	Scalar CMC, Set Mask, Cond=5, Op=0
dtlt	Dy	0xffff91c0	Scalar CMC, Set Mask, Cond=6, Op=0
dtnan	Dz	0xffff9188	Scalar CMC, Set Mask, Cond=4, Op=2
dtne	Dy	0xffff9180	Scalar CMC, Set Mask, Cond=4, Op=0
dtnrm	Dz	0xffff9148	Scalar CMC, Set Mask, Cond=2, Op=2
dtof	Fx, Dz	0xffff9c2c	Scalar 43, Unconditional
dtof,c	Fx, Dz	0xffff9cac	Scalar 43, Nullify if StStack<0> clear
dtoi	Ix, Dz	0xffff9c30	Scalar 44, Unconditional
dtoi,c	Ix, Dz	0xffff9cb0	Scalar 44, Nullify if StStack<0> clear
dtsgn	Dz	0xffff91e8	Scalar CMC, Set Mask, Cond=7, Op=2
dtun	Dy	0xffff9100	Scalar CMC, Set Mask, Cond=0, Op=0
dtze	Dz	0xffff9168	Scalar CMC, Set Mask, Cond=3, Op=2
duvceq	Vy, Vz	0xffff9270	Vector CMC, Move, Cond=3, Op=4
duvcge	Vy, Vz	0xffff9250	Vector CMC, Move, Cond=2, Op=4
duvcgt	Vy, Vz	0xffff9230	Vector CMC, Move, Cond=1, Op=4
duvcle	Vy, Vz	0xffff92b0	Vector CMC, Move, Cond=5, Op=4

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
duvclt	Vy, Vz	0xffff92d0	Vector CMC, Move, Cond=6, Op=4
duvcne	Vy, Vz	0xffff9290	Vector CMC, Move, Cond=4, Op=4
duvcun	Vy, Vz	0xffff9210	Vector CMC, Move, Cond=0, Op=4
duvdnm	Vz	0xffff9228	Vector CMC, Move, Cond=1, Op=2
duvinf	Vz	0xffff9208	Vector CMC, Move, Cond=0, Op=2
duvnan	Vz	0xffff9288	Vector CMC, Move, Cond=4, Op=2
duvnrm	Vz	0xffff9248	Vector CMC, Move, Cond=2, Op=2
duvsgn	Vz	0xffff92e8	Vector CMC, Move, Cond=7, Op=2
duvze	Vz	0xffff9268	Vector CMC, Move, Cond=3, Op=2
dvabs	Vx, Vz	0xffff9e90	Vec.Dyadic 36, Non-Masked
dvabs,mf	Vx, Vz	0xffff9ed0	Vec.Dyadic 84, Use /Mask
dvabs,mt	Vx, Vz	0xffff9e50	Vec.Dyadic 84, Use Mask
dvadd	Vx, Vy, Vz	0xffff9b80	Vec.Triadic 16, Non-Masked
dvadd	Vx, Vz, Aw	0xffff9680	Vec.Dyadic 0, Non-Masked
dvadd,mf	Vx, Vy, Vz	0xffff9bc0	Vec.Triadic 64, Use /Mask
dvadd,mf	Vx, Vz, Aw	0xffff96c0	Vec.Dyadic 48, Use /Mask
dvadd,mf	Vx, [Vz], Vy	0xffff9ac0	Vec.Dyadic 64, Use /Mask
dvadd,mt	Vx, Vy, Vz	0xffff9b40	Vec.Triadic 64, Use Mask
dvadd,mt	Vx, Vz, Aw	0xffff9640	Vec.Dyadic 48, Use Mask
dvadd,mt	Vx, [Vz], Vy	0xffff9a40	Vec.Dyadic 64, Use Mask
dvam	Vx, Vy, Vz, Aw	0xffff9780	Vec.Triadic 0, Non-Masked
dvceq	Vy, [Vz]	0xffff9360	Vector CMC, Compare, Cond=3, Op=0
dvceq	Vy, Vz	0xffff9370	Vector CMC, Compare, Cond=3, Op=4
dvcge	Vy, [Vz]	0xffff9340	Vector CMC, Compare, Cond=2, Op=0
dvcge	Vy, Vz	0xffff9350	Vector CMC, Compare, Cond=2, Op=4
dvcgt	Vy, [Vz]	0xffff9320	Vector CMC, Compare, Cond=1, Op=0
dvcgt	Vy, Vz	0xffff9330	Vector CMC, Compare, Cond=1, Op=4
dvcle	Vy, [Vz]	0xffff93a0	Vector CMC, Compare, Cond=5, Op=0
dvcle	Vy, Vz	0xffff93b0	Vector CMC, Compare, Cond=5, Op=4
dvclt	Vy, [Vz]	0xffff93c0	Vector CMC, Compare, Cond=6, Op=0
dvclt	Vy, Vz	0xffff93d0	Vector CMC, Compare, Cond=6, Op=4
dvcne	Vy, [Vz]	0xffff9380	Vector CMC, Compare, Cond=4, Op=0
dvcne	Vy, Vz	0xffff9390	Vector CMC, Compare, Cond=4, Op=4
dvcun	Vy, [Vz]	0xffff9300	Vector CMC, Compare, Cond=0, Op=0
dvcun	Vy, Vz	0xffff9310	Vector CMC, Compare, Cond=0, Op=4
dvdnm	Vz	0xffff9328	Vector CMC, Compare, Cond=1, Op=2
dvinf	Vz	0xffff9308	Vector CMC, Compare, Cond=0, Op=2
dvma	Vx, Vy, Aw, Vz	0xffff9790	Vec.Triadic 4, Non-Masked
dvma,mf	Vx, Vy, Aw, Vz	0xffff97d0	Vec.Triadic 52, Use /Mask
dvma,mf	Vx, Vy, Vz, Aw	0xffff97c0	Vec.Triadic 48, Use /Mask
dvma,mt	Vx, Vy, Aw, Vz	0xffff9750	Vec.Triadic 52, Use Mask
dvma,mt	Vx, Vy, Vz, Aw	0xffff9740	Vec.Triadic 48, Use Mask
dvmax	Vx, Vy, Vz	0xffff97a8	Vec.Triadic 10, Non-Masked

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
dvmax	Vx, [Vz], Vy	0xffff96a8	Vec.Dyadic 10, Non-Masked
dvmax,mf	Vx, Vy, Vz	0xffff97e8	Vec.Triadic 58, Use /Mask
dvmax,mf	Vx, [Vz], Vy	0xffff96e8	Vec.Dyadic 58, Use /Mask
dvmax,mt	Vx, Vy, Vz	0xffff9768	Vec.Triadic 58, Use Mask
dvmax,mt	Vx, [Vz], Vy	0xffff9668	Vec.Dyadic 58, Use Mask
dvmin	Vx, Vy, Vz	0xffff97a0	Vec.Triadic 8, Non-Masked
dvmin	Vx, [Vz], Vy	0xffff96a0	Vec.Dyadic 8, Non-Masked
dvmin,mf	Vx, Vy, Vz	0xffff97e0	Vec.Triadic 56, Use /Mask
dvmin,mf	Vx, [Vz], Vy	0xffff96e0	Vec.Dyadic 56, Use /Mask
dvmin,mt	Vx, Vy, Vz	0xffff9760	Vec.Triadic 56, Use Mask
dvmin,mt	Vx, [Vz], Vy	0xffff9660	Vec.Dyadic 56, Use Mask
dvmove	Vx, Vz	0xffff9220	Vector CMC, Move, Cond=1, Op=0
dvms	Vx, Vy, Aw, Vz	0xffff9798	Vec.Triadic 6, Non-Masked
dvms	Vx, Vy, Vz, Aw	0xffff9f80	Vec.Triadic 32, Non-Masked
dvms,mf	Vx, Vy, Aw, Vz	0xffff97d8	Vec.Triadic 54, Use /Mask
dvms,mf	Vx, Vy, Vz, Aw	0xffff9fc0	Vec.Triadic 80, Use /Mask
dvms,mt	Vx, Vy, Aw, Vz	0xffff9758	Vec.Triadic 54, Use Mask
dvms,mt	Vx, Vy, Vz, Aw	0xffff9f40	Vec.Triadic 80, Use Mask
dvmul	Vx, Vy, Vz	0xffff9b90	Vec.Triadic 20, Non-Masked
dvmul	Vx, Vz, Aw	0xffff9690	Vec.Dyadic 4, Non-Masked
dvmul	Vx, [Vz], Vy	0xffff9a90	Vec.Dyadic 20, Non-Masked
dvmul,mf	Vx, Vy, Vz	0xffff9bd0	Vec.Triadic 68, Use /Mask
dvmul,mf	Vx, Vz, Aw	0xffff96d0	Vec.Dyadic 52, Use /Mask
dvmul,mf	Vx, [Vz], Vy	0xffff9ad0	Vec.Dyadic 68, Use /Mask
dvmul,mt	Vx, Vy, Vz	0xffff9b50	Vec.Triadic 68, Use Mask
dvmul,mt	Vx, Vz, Aw	0xffff9650	Vec.Dyadic 52, Use Mask
dvmul,mt	Vx, [Vz], Vy	0xffff9a50	Vec.Dyadic 68, Use Mask
dvnan	Vz	0xffff9388	Vector CMC, Compare, Cond=4, Op=2
dvneg	Vx, Vz	0xffff9e98	Vec.Dyadic 38, Non-Masked
dvneg,mf	Vx, Vz	0xffff9ed8	Vec.Dyadic 86, Use /Mask
dvneg,mt	Vx, Vz	0xffff9e58	Vec.Dyadic 86, Use Mask
dvnrm	Vz	0xffff9348	Vector CMC, Compare, Cond=2, Op=2
dvsgn	Vz	0xffff93e8	Vector CMC, Compare, Cond=7, Op=2
dvsm	Vx, Aw, Vy, Vz	0xffff9788	Vec.Triadic 2, Non-Masked
dvsm	Vx, Vz, Vy, Aw	0xffff9f88	Vec.Triadic 34, Non-Masked
dvsm,mf	Vx, Aw, Vy, Vz	0xffff97c8	Vec.Triadic 50, Use /Mask
dvsm,mf	Vx, Vz, Vy, Aw	0xffff9fc8	Vec.Triadic 82, Use /Mask
dvsm,mt	Vx, Aw, Vy, Vz	0xffff9748	Vec.Triadic 50, Use Mask
dvsm,mt	Vx, Vz, Vy, Aw	0xffff9f48	Vec.Triadic 82, Use Mask
dvsub	Vx, Aw, Vz	0xffff9688	Vec.Dyadic 2, Non-Masked
dvsub	Vx, Vy, Vz	0xffff9b88	Vec.Triadic 18, Non-Masked
dvsub	Vx, Vz, Aw	0xffff9698	Vec.Dyadic 6, Non-Masked
dvsub	Vx, [Vz], Vy	0xffff9a98	Vec.Dyadic 22, Non-Masked
dvsub,mf	Vx, Aw, Vz	0xffff96c8	Vec.Dyadic 50, Use /Mask
dvsub,mf	Vx, Vy, Vz	0xffff9bc8	Vec.Triadic 66, Use /Mask
dvsub,mf	Vx, Vy, [Vz]	0xffff9ac8	Vec.Dyadic 66, Use /Mask

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
dvsub,mf	Vx, Vz, Aw	0xffff96d8	Vec.Dyadic 54, Use /Mask
dvsub,mf	Vx, [Vz], Vy	0xffff9ad8	Vec.Dyadic 70, Use /Mask
dvsub,mt	Vx, Aw, Vz	0xffff9648	Vec.Dyadic 50, Use Mask
dvsub,mt	Vx, Vy, Vz	0xffff9b48	Vec.Triadic 66, Use Mask
dvsub,mt	Vx, Vy, [Vz]	0xffff9a48	Vec.Dyadic 66, Use Mask
dvsub,mt	Vx, Vz, Aw	0xffff9658	Vec.Dyadic 54, Use Mask
dvsub,mt	Vx, [Vz], Vy	0xffff9a58	Vec.Dyadic 70, Use Mask
dvze	Vz	0xffff9368	Vector CMC, Compare, Cond=3, Op=2
f1	AC	0xffff9f04	Accumulator Op 17
fabs	Fx, Fz	0xffff9c14	Scalar 37, Unconditional
fabs,c	Fx, Fz	0xffff9c94	Scalar 37, Nullify if StStack<0> clear
fadd	Aw, Fy, Fz	0xffff9844	Scalar 65, Unconditional
fadd	Aw, Fz, Aw	0xffff9444	Scalar 49, Unconditional
fadd	Fx, Fy, Fz	0xffff9804	Scalar 17, Unconditional
fadd	Fx, Fz, Aw	0xffff9c44	Scalar 81, Unconditional
fadd,c	Aw, Fy, Fz	0xffff98c4	Scalar 65, Nullify if StStack<0> clear
fadd,c	Aw, Fz, Aw	0xffff94c4	Scalar 49, Nullify if StStack<0> clear
fadd,c	Fx, Fy, Fz	0xffff9884	Scalar 17, Nullify if StStack<0> clear
fadd,c	Fx, Fz, Aw	0xffff9cc4	Scalar 81, Nullify if StStack<0> clear
fam	Fx, Fy, Fz, Aw	0xffff9404	Scalar 1, Unconditional
famax	Fx, Fy, Fz	0xffff9c74	Scalar 93, Unconditional
famax,c	Fx, Fy, Fz	0xffff9cf4	Scalar 93, Nullify if StStack<0> clear
fam,c	Fx, Fy, Fz, Aw	0xffff9484	Scalar 1, Nullify if StStack<0> clear
fceq	Fy, Fz	0xffff9174	Scalar CMC, Set Mask, Cond=3, Op=5
fcge	Fy, Fz	0xffff9154	Scalar CMC, Set Mask, Cond=2, Op=5
fcgt	Fy, Fz	0xffff9134	Scalar CMC, Set Mask, Cond=1, Op=5
fcle	Fy, Fz	0xffff91b4	Scalar CMC, Set Mask, Cond=5, Op=5
fclt	Fy, Fz	0xffff91d4	Scalar CMC, Set Mask, Cond=6, Op=5
fcne	Fy, Fz	0xffff9194	Scalar CMC, Set Mask, Cond=4, Op=5
fcun	Fy, Fz	0xffff9114	Scalar CMC, Set Mask, Cond=0, Op=5
fdiv	Aw, Fy, Fz	0xffff985c	Scalar 71, Unconditional
fdiv	Fx, Fy, Fz	0xffff981c	Scalar 23, Unconditional
fhalf	AC	0xffff9f24	Accumulator Op 25
flog10	AC	0xffff9f3c	Accumulator Op 31
floge	AC	0xffff9f34	Accumulator Op 29
fm1	AC	0xffff9f0c	Accumulator Op 19
fma	Fx, Fy, Aw, Fz	0xffff9414	Scalar 5, Unconditional
fma,c	Fx, Fy, Aw, Fz	0xffff9494	Scalar 5, Nullify if StStack<0> clear
fmax	Fx, Fy, Fz	0xffff942c	Scalar 11, Unconditional
fmax,c	Fx, Fy, Fz	0xffff94ac	Scalar 11, Nullify if StStack<0> clear
fmin	Fx, Fy, Fz	0xffff9424	Scalar 9, Unconditional
fmin,c	Fx, Fy, Fz	0xffff94a4	Scalar 9, Nullify if StStack<0> clear
fmove	Fx, Fz	0xffff9024	Scalar CMC, NoSetMsk, Cond=1, Op=1

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fms	Fx, Fy, Aw, Fz	0xffff941c	Scalar 7, Unconditional
fms	Fx, Fy, Fz, Aw	0xffff9c04	Scalar 33, Unconditional
fms,c	Fx, Fy, Aw, Fz	0xffff949c	Scalar 7, Nullify if StStack<0> clear
fms,c	Fx, Fy, Fz, Aw	0xffff9c84	Scalar 33, Nullify if StStack<0> clear
fmul	Aw, Fy, Fz	0xffff9854	Scalar 69, Unconditional
fmul	Aw, Fz, Aw	0xffff9454	Scalar 53, Unconditional
fmul	Fx, Fy, Fz	0xffff9814	Scalar 21, Unconditional
fmul	Fx, Fz, Aw	0xffff9c54	Scalar 85, Unconditional
fmul,c	Aw, Fy, Fz	0xffff98d4	Scalar 69, Nullify if StStack<0> clear
fmul,c	Aw, Fz, Aw	0xffff94d4	Scalar 53, Nullify if StStack<0> clear
fmul,c	Fx, Fy, Fz	0xffff9894	Scalar 21, Nullify if StStack<0> clear
fmul,c	Fx, Fz, Aw	0xffff9cd4	Scalar 85, Nullify if StStack<0> clear
fneg	Fx, Fz	0xffff9c1c	Scalar 39, Unconditional
fneg,c	Fx, Fz	0xffff9c9c	Scalar 39, Nullify if StStack<0> clear
fninf	AC	0xffff9f1c	Accumulator Op 23
fpinf	AC	0xffff9f14	Accumulator Op 21
fradabs	AC, Vy	0xffff9994	Vec.Reduc. 21, Non-Masked
fradabs,mf	AC, Vy	0xffff99d4	Vec.Reduc. 69, Use /Mask
fradabs,mt	AC, Vy	0xffff9954	Vec.Reduc. 69, Use Mask
fradd	AC, Vy	0xffff9984	Vec.Reduc. 17, Non-Masked
fradd,mf	AC, Vy	0xffff99c4	Vec.Reduc. 65, Use /Mask
fradd,mt	AC, Vy	0xffff9944	Vec.Reduc. 65, Use Mask
frma	AC, Vy, AC, Vz	0xffff9594	Vec.Reduc. 5, Non-Masked
frma	AC, Vy, Vz, AC	0xffff9584	Vec.Reduc. 1, Non-Masked
frma	AC, Vy, [Vz], AC	0xffff999c	Vec.Reduc. 23, Non-Masked
frma,mf	AC, Vy, AC, Vz	0xffff95d4	Vec.Reduc. 53, Use /Mask
frma,mf	AC, Vy, Vz, AC	0xffff95c4	Vec.Reduc. 49, Use /Mask
frma,mf	AC, Vy, [Vz], AC	0xffff99dc	Vec.Reduc. 71, Use /Mask
frma,mt	AC, Vy, AC, Vz	0xffff9554	Vec.Reduc. 53, Use Mask
frma,mt	AC, Vy, Vz, AC	0xffff9544	Vec.Reduc. 49, Use Mask
frma,mt	AC, Vy, [Vz], AC	0xffff995c	Vec.Reduc. 71, Use Mask
frmax	AC, Vy	0xffff95ac	Vec.Reduc. 11, Non-Masked
frmax,mf	AC, Vy	0xffff95ec	Vec.Reduc. 59, Use /Mask
frmax,mt	AC, Vy	0xffff956c	Vec.Reduc. 59, Use Mask
frmin	AC, Vy	0xffff95a4	Vec.Reduc. 9, Non-Masked
frmin,mf	AC, Vy	0xffff95e4	Vec.Reduc. 57, Use /Mask
frmin,mt	AC, Vy	0xffff9564	Vec.Reduc. 57, Use Mask
frms	AC, Vy, AC, Vz	0xffff959c	Vec.Reduc. 7, Non-Masked
frms	AC, Vy, Vz, AC	0xffff9d84	Vec.Reduc. 33, Non-Masked
frms,mf	AC, Vy, AC, Vz	0xffff95dc	Vec.Reduc. 55, Use /Mask
frms,mf	AC, Vy, Vz, AC	0xffff9dc4	Vec.Reduc. 81, Use /Mask
frms,mt	AC, Vy, AC, Vz	0xffff955c	Vec.Reduc. 55, Use Mask
frms,mt	AC, Vy, Vz, AC	0xffff9d44	Vec.Reduc. 81, Use Mask
frmxabs	AC, Vy	0xffff998c	Vec.Reduc. 19, Non-Masked
frmxabs,mf	AC, Vy	0xffff99cc	Vec.Reduc. 67, Use /Mask
frmxabs,mt	AC, Vy	0xffff994c	Vec.Reduc. 67, Use Mask
frpi	AC	0xffff9f2c	Accumulator Op 27

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
frsm	AC, AC, Vz, Vy	0xffff958c	Vec.Reduc. 3, Non-Masked
frsm	AC, Vz, Vy, AC	0xffff9d8c	Vec.Reduc. 35, Non-Masked
frsm,mf	AC, AC, Vz, Vy	0xffff95cc	Vec.Reduc. 51, Use /Mask
frsm,mf	AC, Vz, Vy, AC	0xffff9dcc	Vec.Reduc. 83, Use /Mask
frsm,mt	AC, AC, Vz, Vy	0xffff954c	Vec.Reduc. 51, Use Mask
frsm,mt	AC, Vz, Vy, AC	0xffff9d4c	Vec.Reduc. 83, Use Mask
fsm	Fx, Aw, Fy, Fz	0xffff940c	Scalar 3, Unconditional
fsm	Fx, Fz, Fy, Aw	0xffff9c0c	Scalar 35, Unconditional
fsm,c	Fx, Aw, Fy, Fz	0xffff948c	Scalar 3, Nullify if StStack<0> clear
fsm,c	Fx, Fz, Fy, Aw	0xffff9c8c	Scalar 35, Nullify if StStack<0> clear
fsub	Aw, Aw, Fz	0xffff944c	Scalar 51, Unconditional
fsub	Aw, Fy, Aw	0xffff945c	Scalar 55, Unconditional
fsub	Aw, Fy, Fz	0xffff984c	Scalar 67, Unconditional
fsub	Fx, Aw, Fz	0xffff9c4c	Scalar 83, Unconditional
fsub	Fx, Fy, Aw	0xffff9c5c	Scalar 87, Unconditional
fsub	Fx, Fy, Fz	0xffff980c	Scalar 19, Unconditional
fsub,c	Aw, Aw, Fz	0xffff94cc	Scalar 51, Nullify if StStack<0> clear
fsub,c	Aw, Fy, Aw	0xffff94dc	Scalar 55, Nullify if StStack<0> clear
fsub,c	Aw, Fy, Fz	0xffff98cc	Scalar 67, Nullify if StStack<0> clear
fsub,c	Fx, Aw, Fz	0xffff9ccc	Scalar 83, Nullify if StStack<0> clear
fsub,c	Fx, Fy, Aw	0xffff9cdc	Scalar 87, Nullify if StStack<0> clear
fsub,c	Fx, Fy, Fz	0xffff988c	Scalar 19, Nullify if StStack<0> clear
ftdnm	Fz	0xffff9138	Scalar CMC, Set Mask, Cond=1, Op=6
fteq	Fy	0xffff9164	Scalar CMC, Set Mask, Cond=3, Op=1
ftge	Fy	0xffff9144	Scalar CMC, Set Mask, Cond=2, Op=1
ftgt	Fy	0xffff9124	Scalar CMC, Set Mask, Cond=1, Op=1
ftinf	Fz	0xffff9118	Scalar CMC, Set Mask, Cond=0, Op=6
ftle	Fy	0xffff91a4	Scalar CMC, Set Mask, Cond=5, Op=1
ftlt	Fy	0xffff91c4	Scalar CMC, Set Mask, Cond=6, Op=1
ftnan	Fz	0xffff9198	Scalar CMC, Set Mask, Cond=4, Op=6
ftne	Fy	0xffff9184	Scalar CMC, Set Mask, Cond=4, Op=1
ftnrm	Fz	0xffff9158	Scalar CMC, Set Mask, Cond=2, Op=6
ftod	Dx, Fz	0xffff9c28	Scalar 42, Unconditional
ftod,c	Dx, Fz	0xffff9ca8	Scalar 42, Nullify if StStack<0> clear
ftoi	Ix, Fz	0xffff9c34	Scalar 45, Unconditional
ftoi,c	Ix, Fz	0xffff9cb4	Scalar 45, Nullify if StStack<0> clear
ftsgn	Fz	0xffff91f8	Scalar CMC, Set Mask, Cond=7, Op=6
ftun	Fy	0xffff9104	Scalar CMC, Set Mask, Cond=0, Op=1
ftze	Fz	0xffff9178	Scalar CMC, Set Mask, Cond=3, Op=6
fuvceq	Vy, Vz	0xffff9274	Vector CMC, Move, Cond=3, Op=5
fuvcge	Vy, Vz	0xffff9254	Vector CMC, Move, Cond=2, Op=5
fuvcgt	Vy, Vz	0xffff9234	Vector CMC, Move, Cond=1, Op=5
fuvcle	Vy, Vz	0xffff92b4	Vector CMC, Move, Cond=5, Op=5

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fuvclt	Vy, Vz	0xffff92d4	Vector CMC, Move, Cond=6, Op=5
fuvcne	Vy, Vz	0xffff9294	Vector CMC, Move, Cond=4, Op=5
fuvcun	Vy, Vz	0xffff9214	Vector CMC, Move, Cond=0, Op=5
fuvdnm	Vz	0xffff9238	Vector CMC, Move, Cond=1, Op=6
fuvinf	Vz	0xffff9218	Vector CMC, Move, Cond=0, Op=6
fuvnan	Vz	0xffff9298	Vector CMC, Move, Cond=4, Op=6
fuvnrm	Vz	0xffff9258	Vector CMC, Move, Cond=2, Op=6
fuvsgn	Vz	0xffff92f8	Vector CMC, Move, Cond=7, Op=6
fuvze	Vz	0xffff9278	Vector CMC, Move, Cond=3, Op=6
fvabs	Vx, Vz	0xffff9e94	Vec.Dyadic 37, Non-Masked
fvabs,mf	Vx, Vz	0xffff9ed4	Vec.Dyadic 85, Use /Mask
fvabs,mt	Vx, Vz	0xffff9e54	Vec.Dyadic 85, Use Mask
fvadd	Vx, Vy, Vz	0xffff9b84	Vec.Triadic 17, Non-Masked
fvadd	Vx, Vz, Aw	0xffff9684	Vec.Dyadic 1, Non-Masked
fvadd,mf	Vx, Vy, Vz	0xffff9bc4	Vec.Triadic 65, Use /Mask
fvadd,mf	Vx, Vz, Aw	0xffff96c4	Vec.Dyadic 49, Use /Mask
fvadd,mf	Vx, [Vz], Vy	0xffff9ac4	Vec.Dyadic 65, Use /Mask
fvadd,mt	Vx, Vy, Vz	0xffff9b44	Vec.Triadic 65, Use Mask
fvadd,mt	Vx, Vz, Aw	0xffff9644	Vec.Dyadic 49, Use Mask
fvadd,mt	Vx, [Vz], Vy	0xffff9a44	Vec.Dyadic 65, Use Mask
fvam	Vx, Vy, Vz, Aw	0xffff9784	Vec.Triadic 1, Non-Masked
fvceq	Vy, [Vz]	0xffff9364	Vector CMC, Compare, Cond=3, Op=1
fvceq	Vy, Vz	0xffff9374	Vector CMC, Compare, Cond=3, Op=5
fvcge	Vy, [Vz]	0xffff9344	Vector CMC, Compare, Cond=2, Op=1
fvcge	Vy, Vz	0xffff9354	Vector CMC, Compare, Cond=2, Op=5
fvcgt	Vy, [Vz]	0xffff9324	Vector CMC, Compare, Cond=1, Op=1
fvcgt	Vy, Vz	0xffff9334	Vector CMC, Compare, Cond=1, Op=5
fvcle	Vy, [Vz]	0xffff93a4	Vector CMC, Compare, Cond=5, Op=1
fvcle	Vy, Vz	0xffff93b4	Vector CMC, Compare, Cond=5, Op=5
fvclt	Vy, [Vz]	0xffff93c4	Vector CMC, Compare, Cond=6, Op=1
fvclt	Vy, Vz	0xffff93d4	Vector CMC, Compare, Cond=6, Op=5
fvcne	Vy, [Vz]	0xffff9384	Vector CMC, Compare, Cond=4, Op=1
fvcne	Vy, Vz	0xffff9394	Vector CMC, Compare, Cond=4, Op=5
fvcun	Vy, [Vz]	0xffff9304	Vector CMC, Compare, Cond=0, Op=1
fvcun	Vy, Vz	0xffff9314	Vector CMC, Compare, Cond=0, Op=5
fvdnm	Vz	0xffff9338	Vector CMC, Compare, Cond=1, Op=6
fvinf	Vz	0xffff9318	Vector CMC, Compare, Cond=0, Op=6
fvma	Vx, Vy, Aw, Vz	0xffff9794	Vec.Triadic 5, Non-Masked
fvma,mf	Vx, Vy, Aw, Vz	0xffff97d4	Vec.Triadic 53, Use /Mask
fvma,mf	Vx, Vy, Vz, Aw	0xffff97c4	Vec.Triadic 49, Use /Mask
fvma,mt	Vx, Vy, Aw, Vz	0xffff9754	Vec.Triadic 53, Use Mask
fvma,mt	Vx, Vy, Vz, Aw	0xffff9744	Vec.Triadic 49, Use Mask
fvmax	Vx, Vy, Vz	0xffff97ac	Vec.Triadic 11, Non-Masked

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fvmax	Vx, [Vz], Vy	0xffff96ac	Vec.Dyadic 11, Non-Masked
fvmax,mf	Vx, Vy, Vz	0xffff97ec	Vec.Triadic 59, Use /Mask
fvmax,mf	Vx, [Vz], Vy	0xffff96ec	Vec.Dyadic 59, Use /Mask
fvmax,mt	Vx, Vy, Vz	0xffff976c	Vec.Triadic 59, Use Mask
fvmax,mt	Vx, [Vz], Vy	0xffff966c	Vec.Dyadic 59, Use Mask
fvmin	Vx, Vy, Vz	0xffff97a4	Vec.Triadic 9, Non-Masked
fvmin	Vx, [Vz], Vy	0xffff96a4	Vec.Dyadic 9, Non-Masked
fvmin,mf	Vx, Vy, Vz	0xffff97e4	Vec.Triadic 57, Use /Mask
fvmin,mf	Vx, [Vz], Vy	0xffff96e4	Vec.Dyadic 57, Use /Mask
fvmin,mt	Vx, Vy, Vz	0xffff9764	Vec.Triadic 57, Use Mask
fvmin,mt	Vx, [Vz], Vy	0xffff9664	Vec.Dyadic 57, Use Mask
fvmove	Vx, Vz	0xffff9224	Vector CMC, Move, Cond=1, Op=1
fvms	Vx, Vy, Aw, Vz	0xffff979c	Vec.Triadic 7, Non-Masked
fvms	Vx, Vy, Vz, Aw	0xffff9f84	Vec.Triadic 33, Non-Masked
fvms,mf	Vx, Vy, Aw, Vz	0xffff97dc	Vec.Triadic 55, Use /Mask
fvms,mf	Vx, Vy, Vz, Aw	0xffff9fc4	Vec.Triadic 81, Use /Mask
fvms,mt	Vx, Vy, Aw, Vz	0xffff975c	Vec.Triadic 55, Use Mask
fvms,mt	Vx, Vy, Vz, Aw	0xffff9f44	Vec.Triadic 81, Use Mask
fvmul	Vx, Vy, Vz	0xffff9b94	Vec.Triadic 21, Non-Masked
fvmul	Vx, Vz, Aw	0xffff9694	Vec.Dyadic 5, Non-Masked
fvmul	Vx, [Vz], Vy	0xffff9a94	Vec.Dyadic 21, Non-Masked
fvmul,mf	Vx, Vy, Vz	0xffff9bd4	Vec.Triadic 69, Use /Mask
fvmul,mf	Vx, Vz, Aw	0xffff96d4	Vec.Dyadic 53, Use /Mask
fvmul,mf	Vx, [Vz], Vy	0xffff9ad4	Vec.Dyadic 69, Use /Mask
fvmul,mt	Vx, Vy, Vz	0xffff9b54	Vec.Triadic 69, Use Mask
fvmul,mt	Vx, Vz, Aw	0xffff9654	Vec.Dyadic 53, Use Mask
fvmul,mt	Vx, [Vz], Vy	0xffff9a54	Vec.Dyadic 69, Use Mask
fvnan	Vz	0xffff9398	Vector CMC, Compare, Cond=4, Op=6
fvneg	Vx, Vz	0xffff9e9c	Vec.Dyadic 39, Non-Masked
fvneg,mf	Vx, Vz	0xffff9edc	Vec.Dyadic 87, Use /Mask
fvneg,mt	Vx, Vz	0xffff9e5c	Vec.Dyadic 87, Use Mask
fvnrm	Vz	0xffff9358	Vector CMC, Compare, Cond=2, Op=6
fvsgn	Vz	0xffff93f8	Vector CMC, Compare, Cond=7, Op=6
fvsm	Vx, Aw, Vy, Vz	0xffff978c	Vec.Triadic 3, Non-Masked
fvsm	Vx, Vz, Vy, Aw	0xffff9f8c	Vec.Triadic 35, Non-Masked
fvsm,mf	Vx, Aw, Vy, Vz	0xffff97cc	Vec.Triadic 51, Use /Mask
fvsm,mf	Vx, Vz, Vy, Aw	0xffff9fcc	Vec.Triadic 83, Use /Mask
fvsm,mt	Vx, Aw, Vy, Vz	0xffff974c	Vec.Triadic 51, Use Mask
fvsm,mt	Vx, Vz, Vy, Aw	0xffff9f4c	Vec.Triadic 83, Use Mask
fvsub	Vx, Aw, Vz	0xffff968c	Vec.Dyadic 3, Non-Masked
fvsub	Vx, Vy, Vz	0xffff9b8c	Vec.Triadic 19, Non-Masked
fvsub	Vx, Vz, Aw	0xffff969c	Vec.Dyadic 7, Non-Masked
fvsub	Vx, [Vz], Vy	0xffff9a9c	Vec.Dyadic 23, Non-Masked
fvsub,mf	Vx, Aw, Vz	0xffff96cc	Vec.Dyadic 51, Use /Mask
fvsub,mf	Vx, Vy, Vz	0xffff9bcc	Vec.Triadic 67, Use /Mask
fvsub,mf	Vx, Vy, [Vz]	0xffff9acc	Vec.Dyadic 67, Use /Mask

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fvsub,mf	Vx, Vz, Aw	0xffff96dc	Vec.Dyadic 55, Use /Mask
fvsub,mf	Vx, [Vz], Vy	0xffff9adc	Vec.Dyadic 71, Use /Mask
fvsub,mt	Vx, Aw, Vz	0xffff964c	Vec.Dyadic 51, Use Mask
fvsub,mt	Vx, Vy, Vz	0xffff9b4c	Vec.Triadic 67, Use Mask
fvsub,mt	Vx, Vy, [Vz]	0xffff9a4c	Vec.Dyadic 67, Use Mask
fvsub,mt	Vx, Vz, Aw	0xffff965c	Vec.Dyadic 55, Use Mask
fvsub,mt	Vx, [Vz], Vy	0xffff9a5c	Vec.Dyadic 71, Use Mask
fvze	Vz	0xffff9378	Vector CMC, Compare, Cond=3, Op=6
i0	AC	0xffff9b20	Accumulator Op 8
i0	Aw	0xffff9b24	Accumulator Op 9
iadd	Ix, Iy, Iz	0xffff9830	Scalar 28, Unconditional
iadd,c	Ix, Iy, Iz	0xffff98b0	Scalar 28, Nullify if StStack<0> clear
iand	Ix, Iy, Iz	0xffff943c	Scalar 15, Unconditional
iand,c	Ix, Iy, Iz	0xffff94bc	Scalar 15, Nullify if StStack<0> clear
iceq	Iy, Iz	0xffff917c	Scalar CMC, Set Mask, Cond=3, Op=7
icge	Iy, Iz	0xffff915c	Scalar CMC, Set Mask, Cond=2, Op=7
icgt	Iy, Iz	0xffff913c	Scalar CMC, Set Mask, Cond=1, Op=7
icle	Iy, Iz	0xffff91bc	Scalar CMC, Set Mask, Cond=5, Op=7
iclt	Iy, Iz	0xffff91dc	Scalar CMC, Set Mask, Cond=6, Op=7
icne	Iy, Iz	0xffff919c	Scalar CMC, Set Mask, Cond=4, Op=7
icov	Iy, Iz	0xffff911c	Scalar CMC, Set Mask, Cond=0, Op=7
icsgn	Iy, Iz	0xffff91fc	Scalar CMC, Set Mask, Cond=7, Op=7
imax	Ix, Iy, Iz	0xffff983c	Scalar 31, Unconditional
imax,c	Ix, Iy, Iz	0xffff98bc	Scalar 31, Nullify if StStack<0> clear
imin	Ix, Iy, Iz	0xffff9834	Scalar 29, Unconditional
imin,c	Ix, Iy, Iz	0xffff98b4	Scalar 29, Nullify if StStack<0> clear
imove	Ix, Iz	0xffff902c	Scalar CMC, NoSetMsk, Cond=1, Op=3
inot	Ix, Iz	0xffff9c38	Scalar 46, Unconditional
inot,c	Ix, Iz	0xffff9cb8	Scalar 46, Nullify if StStack<0> clear
ior	Ix, Iy, Iz	0xffff9434	Scalar 13, Unconditional
ior,c	Ix, Iy, Iz	0xffff94b4	Scalar 13, Nullify if StStack<0> clear
iradd	AC, Vy	0xffff99b0	Vec.Reduc. 28, Non-Masked
iradd,mf	AC, Vy	0xffff99f0	Vec.Reduc. 76, Use /Mask
iradd,mt	AC, Vy	0xffff9970	Vec.Reduc. 76, Use Mask
irand	AC, Vy	0xffff95bc	Vec.Reduc. 15, Non-Masked
irand,mf	AC, Vy	0xffff95fc	Vec.Reduc. 63, Use /Mask
irand,mt	AC, Vy	0xffff957c	Vec.Reduc. 63, Use Mask
irmax	AC, Vy	0xffff99bc	Vec.Reduc. 31, Non-Masked
irmax,mf	AC, Vy	0xffff99fc	Vec.Reduc. 79, Use /Mask
irmax,mt	AC, Vy	0xffff997c	Vec.Reduc. 79, Use Mask
irmin	AC, Vy	0xffff99b4	Vec.Reduc. 29, Non-Masked
irmin,mf	AC, Vy	0xffff99f4	Vec.Reduc. 77, Use /Mask
irmin,mt	AC, Vy	0xffff9974	Vec.Reduc. 77, Use Mask

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
iror	AC, Vy	0xffff95b4	Vec.Reduc. 13, Non-Masked
iror,mf	AC, Vy	0xffff95f4	Vec.Reduc. 61, Use /Mask
iror,mt	AC, Vy	0xffff9574	Vec.Reduc. 61, Use Mask
irxor	AC, Vy	0xffff95b8	Vec.Reduc. 14, Non-Masked
irxor,mf	AC, Vy	0xffff95f8	Vec.Reduc. 62, Use /Mask
irxor,mt	AC, Vy	0xffff9578	Vec.Reduc. 62, Use Mask
isub	Ix, Iy, Iz	0xffff9838	Scalar 30, Unconditional
isub,c	Ix, Iy, Iz	0xffff98b8	Scalar 30, Nullify if StStack<0> clear
iteq	Iy	0xffff916c	Scalar CMC, Set Mask, Cond=3, Op=3
itge	Iy	0xffff914c	Scalar CMC, Set Mask, Cond=2, Op=3
itgt	Iy	0xffff912c	Scalar CMC, Set Mask, Cond=1, Op=3
itle	Iy	0xffff91ac	Scalar CMC, Set Mask, Cond=5, Op=3
itlt	Iy	0xffff91cc	Scalar CMC, Set Mask, Cond=6, Op=3
itne	Iy	0xffff918c	Scalar CMC, Set Mask, Cond=4, Op=3
itod	Dx, Iz	0xffff9c20	Scalar 40, Unconditional
itod,c	Dx, Iz	0xffff9ca0	Scalar 40, Nullify if StStack<0> clear
itof	Fx, Iz	0xffff9c24	Scalar 41, Unconditional
itof,c	Fx, Iz	0xffff9ca4	Scalar 41, Nullify if StStack<0> clear
itov	Iy	0xffff910c	Scalar CMC, Set Mask, Cond=0, Op=3
itsgn	Iy	0xffff91ec	Scalar CMC, Set Mask, Cond=7, Op=3
iuvceq	Vy, Vz	0xffff927c	Vector CMC, Move, Cond=3, Op=7
iuvcge	Vy, Vz	0xffff925c	Vector CMC, Move, Cond=2, Op=7
iuvcgt	Vy, Vz	0xffff923c	Vector CMC, Move, Cond=1, Op=7
iuvcle	Vy, Vz	0xffff92bc	Vector CMC, Move, Cond=5, Op=7
iuvclt	Vy, Vz	0xffff92dc	Vector CMC, Move, Cond=6, Op=7
iuvcne	Vy, Vz	0xffff929c	Vector CMC, Move, Cond=4, Op=7
iuvcov	Vy, Vz	0xffff921c	Vector CMC, Move, Cond=0, Op=7
iuvcsgn	Vy, Vz	0xffff92fc	Vector CMC, Move, Cond=7, Op=7
ivadd	Vx, Vy, Vz	0xffff9bb0	Vec.Triadic 28, Non-Masked
ivadd	Vx, [Vz], Vy	0xffff9ab0	Vec.Dyadic 28, Non-Masked
ivadd,mf	Vx, Vy, Vz	0xffff9bf0	Vec.Triadic 76, Use /Mask
ivadd,mf	Vx, [Vz], Vy	0xffff9af0	Vec.Dyadic 76, Use /Mask
ivadd,mt	Vx, Vy, Vz	0xffff9b70	Vec.Triadic 76, Use Mask
ivadd,mt	Vx, [Vz], Vy	0xffff9a70	Vec.Dyadic 76, Use Mask
ivand	Vx, Vy, Vz	0xffff97bc	Vec.Triadic 15, Non-Masked
ivand	Vx, [Vz], Vy	0xffff96bc	Vec.Dyadic 15, Non-Masked
ivand,mf	Vx, Vy, Vz	0xffff97fc	Vec.Triadic 63, Use /Mask
ivand,mf	Vx, [Vz], Vy	0xffff96fc	Vec.Dyadic 63, Use /Mask
ivand,mt	Vx, Vy, Vz	0xffff977c	Vec.Triadic 63, Use Mask
ivand,mt	Vx, [Vz], Vy	0xffff967c	Vec.Dyadic 63, Use Mask
ivceq	Vy, [Vz]	0xffff936c	Vector CMC, Compare, Cond=3, Op=3
ivceq	Vy, Vz	0xffff937c	Vector CMC, Compare, Cond=3, Op=7
ivcge	Vy, [Vz]	0xffff934c	Vector CMC, Compare, Cond=2, Op=3
ivcge	Vy, Vz	0xffff935c	Vector CMC, Compare, Cond=2, Op=7
ivcgt	Vy, [Vz]	0xffff932c	Vector CMC, Compare, Cond=1, Op=3
ivcgt	Vy, Vz	0xffff933c	Vector CMC, Compare, Cond=1, Op=7
ivcle	Vy, [Vz]	0xffff93ac	Vector CMC, Compare, Cond=5, Op=3

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
ivcle	Vy, Vz	0xffff93bc	Vector CMC, Compare, Cond=5, Op=7
ivclt	Vy, [Vz]	0xffff93cc	Vector CMC, Compare, Cond=6, Op=3
ivclt	Vy, Vz	0xffff93dc	Vector CMC, Compare, Cond=6, Op=7
ivcne	Vy, [Vz]	0xffff938c	Vector CMC, Compare, Cond=4, Op=3
ivcne	Vy, Vz	0xffff939c	Vector CMC, Compare, Cond=4, Op=7
ivcov	Vy, [Vz]	0xffff930c	Vector CMC, Compare, Cond=0, Op=3
ivcov	Vy, Vz	0xffff931c	Vector CMC, Compare, Cond=0, Op=7
ivcsgn	Vy, [Vz]	0xffff93ec	Vector CMC, Compare, Cond=7, Op=3
ivcsgn	Vy, Vz	0xffff93fc	Vector CMC, Compare, Cond=7, Op=7
ivmax	Vx, Vy, Vz	0xffff9bbc	Vec.Triadic 31, Non-Masked
ivmax	Vx, [Vz], Vy	0xffff9abc	Vec.Dyadic 31, Non-Masked
ivmax,mf	Vx, Vy, Vz	0xffff9bfc	Vec.Triadic 79, Use /Mask
ivmax,mf	Vx, [Vz], Vy	0xffff9afc	Vec.Dyadic 79, Use /Mask
ivmax,mt	Vx, Vy, Vz	0xffff9b7c	Vec.Triadic 79, Use Mask
ivmax,mt	Vx, [Vz], Vy	0xffff9a7c	Vec.Dyadic 79, Use Mask
ivmin	Vx, Vy, Vz	0xffff9bb4	Vec.Triadic 29, Non-Masked
ivmin	Vx, [Vz], Vy	0xffff9ab4	Vec.Dyadic 29, Non-Masked
ivmin,mf	Vx, Vy, Vz	0xffff9bf4	Vec.Triadic 77, Use /Mask
ivmin,mf	Vx, [Vz], Vy	0xffff9af4	Vec.Dyadic 77, Use /Mask
ivmin,mt	Vx, Vy, Vz	0xffff9b74	Vec.Triadic 77, Use Mask
ivmin,mt	Vx, [Vz], Vy	0xffff9a74	Vec.Dyadic 77, Use Mask
ivmove	Vx, Vz	0xffff922c	Vector CMC, Move, Cond=1, Op=3
ivnot	Vx, Vz	0xffff9eb8	Vec.Dyadic 46, Non-Masked
ivnot,mf	Vx, Vz	0xffff9ef8	Vec.Dyadic 94, Use /Mask
ivnot,mt	Vx, Vz	0xffff9e78	Vec.Dyadic 94, Use Mask
ivor	Vx, Vy, Vz	0xffff97b4	Vec.Triadic 13, Non-Masked
ivor	Vx, [Vz], Vy	0xffff96b4	Vec.Dyadic 13, Non-Masked
ivor,mf	Vx, Vy, Vz	0xffff97f4	Vec.Triadic 61, Use /Mask
ivor,mf	Vx, [Vz], Vy	0xffff96f4	Vec.Dyadic 61, Use /Mask
ivor,mt	Vx, Vy, Vz	0xffff9774	Vec.Triadic 61, Use Mask
ivor,mt	Vx, [Vz], Vy	0xffff9674	Vec.Dyadic 61, Use Mask
ivsub	Vx, Vy, Vz	0xffff9bb8	Vec.Triadic 30, Non-Masked
ivsub	Vx, Vy, [Vz]	0xffff9ab8	Vec.Dyadic 30, Non-Masked
ivsub	Vx, [Vz], Vy	0xffff96b0	Vec.Dyadic 12, Non-Masked
ivsub,mf	Vx, Vy, Vz	0xffff9bf8	Vec.Triadic 78, Use /Mask
ivsub,mf	Vx, Vy, [Vz]	0xffff9af8	Vec.Dyadic 78, Use /Mask
ivsub,mf	Vx, [Vz], Vy	0xffff96f0	Vec.Dyadic 60, Use /Mask
ivsub,mt	Vx, Vy, Vz	0xffff9b78	Vec.Triadic 78, Use Mask
ivsub,mt	Vx, Vy, [Vz]	0xffff9a78	Vec.Dyadic 78, Use Mask
ivsub,mt	Vx, [Vz], Vy	0xffff9670	Vec.Dyadic 60, Use Mask
ivxor	Vx, Vy, Vz	0xffff97b8	Vec.Triadic 14, Non-Masked
ivxor	Vx, [Vz], Vy	0xffff96b8	Vec.Dyadic 14, Non-Masked
ivxor,mf	Vx, Vy, Vz	0xffff97f8	Vec.Triadic 62, Use /Mask
ivxor,mf	Vx, [Vz], Vy	0xffff96f8	Vec.Dyadic 62, Use /Mask

Table A-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
ivxor,mt	Vx, Vy, Vz	0xffff9778	Vec.Triadic 62, Use Mask
ivxor,mt	Vx, [Vz], Vy	0xffff9678	Vec.Dyadic 62, Use Mask
ixor	Ix, Iy, Iz	0xffff9438	Scalar 14, Unconditional
ixor,c	Ix, Iy, Iz	0xffff94b8	Scalar 14, Nullify if StStack<0> clear
pass	Aw, Vz	0xffff9b38	Accumulator Op 14
pass	Dx, Dz	0xffff9060	Scalar CMC, NoSetMsk, Cond=3, Op=0
pass	Fx, Iz	0xffff9064	Scalar CMC, NoSetMsk, Cond=3, Op=1
pass	Ix, Iz	0xffff906c	Scalar CMC, NoSetMsk, Cond=3, Op=3
pass	Vx, Aw	0xffff9b30	Accumulator Op 12
pass4	AC, Vz	0xffff9b3c	Accumulator Op 15
pass4	Vx, AC	0xffff9b34	Accumulator Op 13
passcf	Dx, Dz	0xffff90e0	Scalar CMC, NoSetMsk, Cond=7, Op=0
passcf	Fx, Fz	0xffff90e4	Scalar CMC, NoSetMsk, Cond=7, Op=1
passcf	Ix, Fz	0xffff90ec	Scalar CMC, NoSetMsk, Cond=7, Op=3
passct	Dx, Dz	0xffff90c0	Scalar CMC, NoSetMsk, Cond=6, Op=0
passct	Fx, Fz	0xffff90c4	Scalar CMC, NoSetMsk, Cond=6, Op=1
passct	Ix, Fz	0xffff90cc	Scalar CMC, NoSetMsk, Cond=6, Op=3
vdtof	Vx, Vz	0xffff9eac	Vec.Dyadic 43, Non-Masked
vdtof,mf	Vx, Vz	0xffff9eec	Vec.Dyadic 91, Use /Mask
vdtof,mt	Vx, Vz	0xffff9e6c	Vec.Dyadic 91, Use Mask
vdtoi	Vx, Vz	0xffff9eb0	Vec.Dyadic 44, Non-Masked
vdtoi,mf	Vx, Vz	0xffff9ef0	Vec.Dyadic 92, Use /Mask
vdtoi,mt	Vx, Vz	0xffff9e70	Vec.Dyadic 92, Use Mask
vftod	Vx, Vz	0xffff9ea8	Vec.Dyadic 42, Non-Masked
vftod,mf	Vx, Vz	0xffff9ee8	Vec.Dyadic 90, Use /Mask
vftod,mt	Vx, Vz	0xffff9e68	Vec.Dyadic 90, Use Mask
vftoi	Vx, Vz	0xffff9eb4	Vec.Dyadic 45, Non-Masked
vftoi,mf	Vx, Vz	0xffff9ef4	Vec.Dyadic 93, Use /Mask
vftoi,mt	Vx, Vz	0xffff9e74	Vec.Dyadic 93, Use Mask
vitod	Vx, Vz	0xffff9ea0	Vec.Dyadic 40, Non-Masked
vitod,mf	Vx, Vz	0xffff9ee0	Vec.Dyadic 88, Use /Mask
vitod,mt	Vx, Vz	0xffff9e60	Vec.Dyadic 88, Use Mask
vitof	Vx, Vz	0xffff9ea4	Vec.Dyadic 41, Non-Masked
vitof,mf	Vx, Vz	0xffff9ee4	Vec.Dyadic 89, Use /Mask
vitof,mt	Vx, Vz	0xffff9e64	Vec.Dyadic 89, Use Mask
vpass	Vx, Vz	0xffff9260	Vector CMC, Move, Cond=3, Op=0
vpass,mf	Vx, Vz	0xffff92c0	Vector CMC, Move, Cond=6, Op=0
vpass,mfc	Vx, Vz	0xffff9280	Vector CMC, Move, Cond=4, Op=0
vpass,mt	Vx, Vz	0xffff92e0	Vector CMC, Move, Cond=7, Op=0
vpass,mtc	Vx, Vz	0xffff92a0	Vector CMC, Move, Cond=5, Op=0
vsexp	AC, [Vz]	0xffff9b2c	Accumulator Op 11
vsexp	Vx, [Vz]	0xffff9240	Vector CMC, Move, Cond=2, Op=01

A.3 TITAN Stack Frame

The stack frame for a function call is illustrated in Figure A-1.

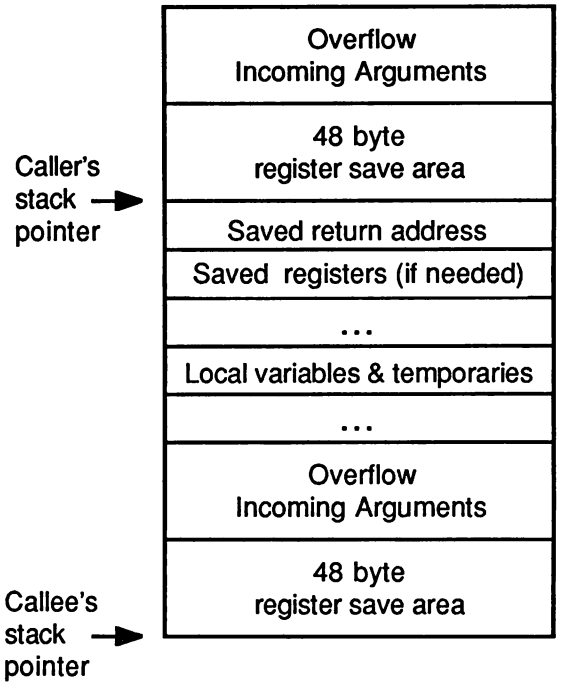


Figure A-1. TITAN Stack Frame

Double precision arguments are aligned on 8-byte boundaries and holes are left if needed for this alignment. Short and byte arguments are widened to 32-bit values and stored on 4-byte boundaries.

By convention, the first several arguments are stored in registers. However, space is always kept on the stack for these arguments, in case they need to be saved.

The stack frame moves exactly once for each call and return. Very simple functions may not need to use the stack and the stack frame may not move at all. The stack pointer is always 8-byte aligned. All stack sizes must be rounded up to multiples of 8 bytes.

The call/return process consists of 4 phases:

- The calling program:
  - Saves those of registers $v1–$t7, and the *caller save* floating point registers that are needed after the call.
  - Puts the arguments in registers.
  - Puts arguments that won't fit in registers on the stack.
  - Does a branch and link to the beginning of the subroutine.

- The called program:
  - Decrements the stack pointer to allocate the new stack frame.
  - If the called program in turn calls other programs, the return address is saved on the stack.
  - Saves those of registers $s0–$s7 and the *callee save* floating point registers that are used.
  - Begins execution.

- To return to the caller, the called program:
  - Puts the return value into register $v0 if it is an integer or a pointer, and into register F0 if it is a float or a double.
  - Restores the return address to the link register.
  - Restores any of the registers $s0–$s7 and the *callee save* floating point registers that were saved.
  - Increments the stack pointer to the previous value.
  - Returns to the return address.

- After the return, the calling program:
  - Restores any of the registers $v1–$t7 and the *caller save* floating point registers that were saved before the call was made.

If a function makes no calls or does not change registers $s0–$s7 and the *callee save* floating point registers, there is no need for the function to touch the stack (except, perhaps, to store its arguments there). In this case, the code is simplified because there is no need to get a new stack frame nor to do any stores or reloads of the stack. The link address remains in register $ra throughout the called function's execution.

A.5 Data Layout In Memory

NOTE
In the floating point formats, when dealing with numbers at or close to the limits of the range, it is possible to exceed the range during ASCII to binary conversion and vice versa. This is caused by rounding errors.

Ardent Fortran has eleven data types:

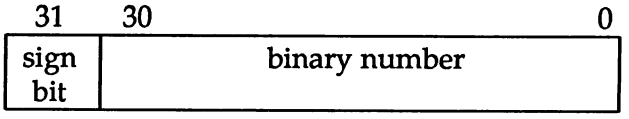
- integer, also called INTEGER*4
- short integer, also called INTEGER*2
- BYTE
- real, also called REAL*4
- double precision, also called REAL*8
- complex, also called COMPLEX*8
- double complex, also called COMPLEX*16
- logical, also called LOGICAL*4
- short logical, also called LOGICAL*2
- byte logical, also called LOGICAL*1
- character

When stored in memory, each has the format described in this appendix.

A.5.0.1 Integer Format

An integer datum is always an exact representation of a short integer of value positive, negative, or 0. The integer format, INTEGER*4, occupies one 32-bit word and has a range of -2^{31} to $+2^{31}-1$ or –2 147 483 648 to +2 147 483 647.

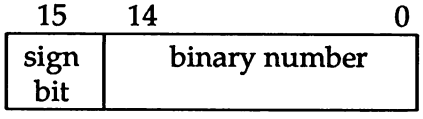
Table A-5. Integer Data Format (INTEGER*4)



A.5.0.2 Short Integer Format

A short integer format, **INTEGER*2**, occupies half of one 32-bit word and has a range of -2^{15} to $+2^{15}-1$ or -32768 to 32767.

Table A-6. Short Integer Format (INTEGER*2)



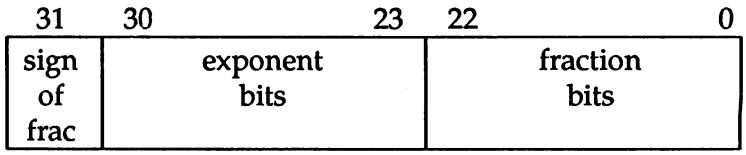
A.5.0.3 Real Format

A real datum is a processor approximation to a real number having a positive, negative, or zero value. In Ardent Fortran, real format corresponds to IEEE Single Format. This format is capable of representing numbers in the range of – inf to zero to + inf, as well as **NaN**, which stands for "Not a Number". The real format, **REAL*4**, occupies one 32-bit word in memory and has an approximate range of 1.175495E-38 to 3.402823E+38.

The real format has 1-bit sign, an 8-bit exponent, and a 23-bit fraction. Significance is approximately seven decimal digits.

- The sign bit is 0 for plus, 1 for minus.
- The exponent field contains 127 plus the actual exponent (power of 2) of the number. Exponent fields containing all 0's and all 1's are "reserved." Special interpretations given to the numeric representation are as follows:
  - If the exponent is 0 and the fraction 0, the number is interpreted as a signed 0.
  - If the exponent is 0 and the fraction not 0, the number is assumed to be "denormalized." Floating point numbers are usually stored in a "normalized" form with a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point.
  - If the exponent is all 1s and the fraction is 0, the number is regarded as a signed infinity. If the exponent is all 1s and the fraction is not 0, then the interpretation is "not-a-number" (NaN).

Table A-7. Real Format



A.5.0.4 Double Precision Format

A double precision datum is a processor approximation to a real number having a positive, negative or zero value. The Ardent Fortran double precision format corresponds to the IEEE Double Format. The double precision format, **REAL*8** or **DOUBLE PRECISION**, occupies two consecutive 32-bit words in memory, and has an approximate range of

$-1.79769313486231*10^{308}$ to $-2.22507385850721*10^{-308}$
0
$+2.22507385850721*10^{-308}$ to $+1.79769313486231*10^{308}$

The double precision format has an 11-bit exponent and a 52-bit fraction. Significance is approximately 16 decimal digits. The sign bit is 0 for plus, 1 for minus. The exponent field contains 1023 plus the actual exponent (power of 2) of the number. Exponent fields containing all 0's and all 1's are reserved.

- If the exponent is 0 and the fraction 0, the number is interpreted as a signed 0.
- If the exponent is 0 and the fraction not 0, the number is assumed to be "denormalized." Floating point numbers are usually stored in a "normalized" form with a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point.
    - If the exponent is all 1s and the fraction is 0, the number is regarded as a signed infinity.
    - If the exponent is all 1s and the fraction is not 0, then the interpretation is "not-a-number" (NAN).

In Table A-8 below, the position at which the implied binary decimal point is placed is just to the left of bit position 51.

Table A-8. Double Precision Format

63	62 ... 52	51 ... 0
sign bit	exponent bits	binary fraction bits

A.5.0.5 Complex Format

A complex datum is a processor approximation to the value of a complex number. The complex format, **COMPLEX*8**, occupies two consecutive 32-bit words in memory. Both the real and imaginary parts have an approximate range of $1.2*10^{-39}$ to $3.4*10^{38}$.

Both the real and the imaginary parts have 23-bit fractions and 8-bit exponents; both have the same significance as a real number. The sign of the exponent is determined in the same manner as that of a real number.

Table A-9. Complex Format

Real Part

31	30 ... 23	22 ... 0	
sign of frac	exponent bits	fraction bits	Word 1

Imaginary Part

31	30 ... 23	22 ... 0	
sign of frac	exponent bits	fraction bits	Word 2

A.5.0.6 Double Complex Format

A complex datum is a processor approximation to the value of a complex number. The double complex format, **COMPLEX*16** or **DOUBLE COMPLEX**, occupies four consecutive 32-bit words in memory. Both the real and imaginary parts have an approximate range of $2.2*10^{-308}$ to $1.8*10^{308}$. Both the real and the imaginary parts have 52-bit fractions and 11-bit exponents; both have the same significance as a double precision number. The sign of the exponent is determined in the same manner as that of a double precision number.

Table A-10. Double Complex

63	62 — 52	51 — 0	
sign bit	exponent bits	binary fraction bits	Words 1,2
Imaginary Part			
63	62 — 52	51 — 0	
sign bit	exponent bits	binary fraction bits	Words 3,4

A.5.0.7 Logical Format

A logical datum is a representation of *true* or *false*, with 0 representing *false*, and any nonzero value representing *true*. The logical format, **LOGICAL*4**, occupies one 32-bit word in memory.

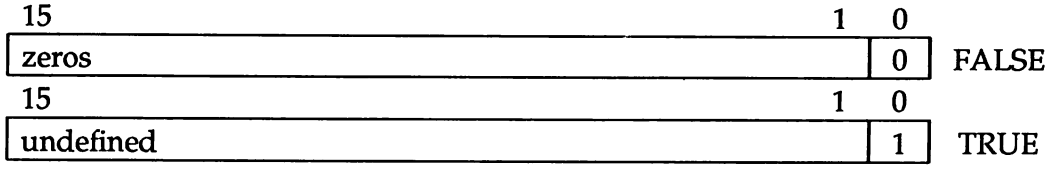
Table A-11. Logical Data Format

31 — 1	0	
zeros	0	FALSE
31 — 1	0	
undefined	1	TRUE

A.5.0.8 Short Logical Format

A logical datum is a representation of *true* or *false*, with 0 representing *false*, and any nonzero value representing *true*. The short logical format, **LOGICAL*2**, occupies half of one 32-bit word in memory.

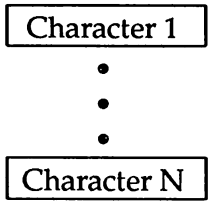
Table A-12. Short Logical Data Format



A.5.0.9 Character Format

A character datum is a character string taken from the ASCII character set. ASCII characters occupy one byte (eight bits) of a 32-bit word, and are packed four to a word in memory. Character strings are stored in memory as a sequence of ASCII codes, 1 per byte.

Table A-13. Character Data Format



INDEX

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y