Chapter 11

Execution Control

Execution of a main program begins with the first executable statement of the main program. Execution of a procedure begins with the first executable statement appearing after the invoked entry point. Executable statements normally are executed in the order in which they appear in the program until a `STOP`, `RETURN`, or `END` statement is executed. The sequential execution of statements can be altered by control statements and constructs as follows:

- Branching statements specify explicitly a new starting place for the execution sequence; execution of a branching statement causes execution to continue at the specified location.
- Execution of an `IF` construct, `CASE` construct, or `DO` construct may involve internal branching around statements nested within the construct.
- The use of an alternate return specifier, or the use of an `END=` or `ERR=` specifier in an input/output statement may result in a branch to a statement other than the next in sequence.

Statements in the first two categories are described in the following sections. Alternate return specifiers are described in Section 12.9, and the use of `END` and `ERR` in input/output statements are discussed in Chapter 14.

11.1 Blocks of Executable Statements

A *block* (R11-1) is a sequence of executable statements and constructs (possibly containing `FORMAT` and `DATA` statements) that is to be treated as a whole within `IF`, `CASE`, and block loop constructs. Since blocks may themselves contain executable constructs (Section 3.3.1), blocks may be nested within other blocks.

Copyright © 1989–1994 Thinking Machines Corporation

A *block* is: R11-1

[execution-part-construct]...

An *execution-part-construct* is one of: R3-7

- *executable-stmt*
- *data-stmt*
- *format-stmt*
- *entry-stmt*

♦ An *execution-part-construct* in a *block* must not be an *entry-stmt*.

The executable constructs that contain blocks and may be used to control the execution sequence are the `IF` construct, the `CASE` construct, the loop construct, and the `WHERE` construct. There are also nonblock forms of the loop construct.

Note that a block may be empty; execution of an empty block has no effect.

An example of a construct containing a block is the following `IF` construct:

```
IF ( A > 0. ) THEN
  B = SQRT( A )    ! these indented statements
  C = LOG( A )     !   form a block
END IF
```

11.1.1 Rules Governing Blocks

If a block contains an executable construct, that construct must be contained entirely within the block. Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a block and transfers from the interior of a block to a statement outside the block may occur. For example, if a statement inside a block has a statement label, a `GOTO` statement using that label may appear in the same block. Subroutine and function references may also appear in a block.

Execution of a block begins with the execution of the first executable construct in the block. Unless there is a transfer of control out of the block, execution of the block is completed when the last executable construct in the sequence is executed. The action that takes place at the terminating boundary statement is either a transfer of control out of the construct, or a transfer of control to the beginning of the block, depending on the particular construct.

These rules governing blocks also hold for the sequences of executable statements and constructs that are contained within the nonblock forms of loop constructs described in Section 11.5. Both nonblock forms of loop construct can be written in an equivalent block form (see Sections 11.5.2.1 and 11.5.2.2), and the above rules apply.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

11.2 Logical IF Statement

The logical `IF` statement (R11-2) controls the execution of a single action statement.

A *logical-if-stmt* is: R11-2
`IF (` scalar-*logical-expr* `)` *action-stmt*

- The *action-stmt* in a *logical-if-stmt* must not itself be a *logical-if-stmt*.

The logical `IF` statement causes a scalar-*logical-expr* to be evaluated. If the value of the expression is true, the *action-stmt* is executed; if the value of the expression is false, the *action-stmt* is not executed, and execution continues as though a `CONTINUE` statement had been executed.

The execution of a function reference in the scalar logical expression is permitted to affect entities in the action statement.

Examples of `IF` statements:

```
IF ( A > 0. ) A = LOG( A )  ! replace positive A with its
                             !logarithm

IF ( ERR ) GO TO 999    ! branch to statement 999 if error
```

The last example combines the `IF` statement with the `GO TO` statement (Section 11.6.1) to form a conditional branch.

11.3 Block IF Construct

The block `IF` construct (R11-3) controls the execution of several executable statements.

A *block-if-construct* is: R11-3
[c-*name* **:** *]* `IF (` scalar-*logical-expr* `) THEN`
  block
[`ELSE IF (` scalar-*logical-expr* `) THEN` *[* c-*name* *]*
  block]...
[`ELSE` *[* c-*name* *]*
  block]
`END IF` *[* c-*name* *]*

Copyright © 1989–1994 Thinking Machines Corporation

- If a construct name c-*name* is present, the same name must be specified on both the IF–THEN statement and the corresponding END IF. If a construct name appears on the IF–THEN statement, it may appear on any ELSE IF or ELSE statement belonging to that IF construct.

The block IF construct causes the evaluation of one or more scalar logical expressions, followed by the execution of at most one of the blocks contained within the IF construct depending on the result of the evaluation. If there is an ELSE statement in the construct, exactly one of the blocks within the IF construct is executed. The scalar logical expressions are evaluated in the order of their appearance in the construct until one of them yields a true value, or until an ELSE statement or an END IF statement is encountered. If a true value or an ELSE statement is found, the block immediately following is executed and execution of the construct is complete. If none of the evaluated expressions yields true and there is no ELSE statement, execution of the construct is completed without the execution of any blocks within the construct.

An ELSE IF statement or an ELSE statement must not be the target of a branch statement. It is permissible to branch to an END IF statement from within the IF construct and also from outside the construct. Execution of an END IF statement has no effect.

Examples of IF constructs follow.

```
IF ( CVAR .EQ. 'RESET' ) THEN  ! test command value
  ! reset state variables to initial values
  I = 0
  J = 0
  K = 0
END IF

! Stop if the proposition is true; otherwise, proceed to next
IF ( PROP ) THEN
  WRITE( 3, '("QED")' )
  STOP
ELSE
  PROP = NEXTPROP
END IF
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

```
OUTER: IF ( A .GT. 0 ) THEN
  B = C/A
  IF ( B .GT. 0 ) THEN
    D = 1.
  END IF
ELSE IF ( C .GT. 0 ) THEN OUTER
  B = A/C
  D = -1.
ELSE OUTER
  B = ABS( MAX( A, C ) )
  D = 0
END IF OUTER
```

The last example illustrates the use of construct names and the nesting of one `IF` construct within another.

11.4 CASE Construct

The `CASE` construct (R11-4) selects for execution one of several blocks of statements based on the value of a scalar expression. One and only one of the blocks of a `CASE` construct is executed.

A *case-construct* is: R11-4

[c-*name* **:** *]* `SELECT CASE (` *case-expr* `)`
[`CASE` *case-selector [* c-*name]*
block]...
`END SELECT` *[* c-*name]*

- If a construct name c-*name* is present, the same name must be specified on both the `SELECT CASE` statement and the corresponding `END SELECT`. If a construct name appears on the `SELECT CASE`, it may also optionally appear on any `CASE` statement belonging to that `CASE` construct.

A *case-expr* is one of: R11-5

- scalar-integer-*expr*
- scalar-*char-expr*
- scalar-*logical-expr*

A *case-selector* is one of: R11-6

- `(` *case-value-range(s)* `)`
- `DEFAULT`

- Only one `DEFAULT` *case-selector* may appear in any given *case-construct*.

Copyright © 1989–1994 Thinking Machines Corporation

A *case-value-range* is one of: R11-7

- *case-value*
- *case-value* **:**
- **:** *case-value*
- *case-value* **:** *case-value*

♦ A *case-value-range* using a colon must not be used if *case-expr* is of type logical.

A *case-value* is one of: R11-8

- scalar-integer-constant-*expr*
- scalar-char-constant-*expr*
- scalar-logical-constant-*expr*

♦ For a given *case-construct*, each *case-value* must be of the same type as *case-expr*.

The CASE construct selects for execution exactly one of its constituent blocks. The execution of the SELECT CASE statement causes the case expression to be evaluated, and the resulting value must match exactly one of the selectors of one of the CASE statements of the construct. For a *case-value-range(s)* list, a match occurs if the case index matches any of the case value ranges in the list. For a case expression with a value of *c*, a match is determined as follows:

- a single case value *v* matches if *c* is equal to *v*
- a case value range of the form "*low* **:** *high*" matches if $low \le c \le high$
- a case value range of the form *"low* **:** " matches if $low \le c$
- a case value range of the form " **:** *high*" matches if $c \le high$
- if no other selector matches and a DEFAULT selector is present, the case expression matches the selector
- if no other selector matches and a DEFAULT selector is absent, there is no match. Execution continues with the statement following the END SELECT statement of the construct.

If there is a match, the block immediately following the CASE statement containing the matching selector is executed, and this completes the execution of the construct.

The case value ranges in different selectors must not overlap (that is, there must be no possible value of the case index that matches more than one selector). Case value ranges within a single case selector may overlap.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

A CASE statement must not be the target of a branch statement. It is permissible to branch to an END SELECT statement only from within the CASE construct of which it is a part.

Examples of the CASE construct:

```
SELECT CASE ( N )
CASE ( :-1 )      ! if N <= -1
  SIGN = -1
CASE ( 0 )        ! if N == 0
  SIGN = 0
CASE ( 1: )       ! if N >= 1
  SIGN = -1
END SELECT
```

The following three code fragments are equivalent:

```
IF ( I .EQ. 1 ) THEN
  CALL A
ELSE
  CALL B
END IF
```

```
SELECT CASE ( I .EQ. 1 )
CASE ( .TRUE. )
  CALL A
CASE ( .FALSE. )
  CALL B
END SELECT
```

```
SELECT CASE ( I )
CASE ( 1 )
  CALL A
CASE DEFAULT
  CALL B
END SELECT
```

The following code fragment illustrates several selections in one block:

```
SELECT CASE ( N )
CASE ( 1, 3:5, 8 )      ! Selects 1, 3, 4, 5, 8
  CALL SUB1( N )
CASE ( 2, 13:20 )       ! Selects 2, 13,14...20
  CALL SUB2( N )
CASE DEFAULT            ! Selects all others
  CALL SUB3( N )
END SELECT
```


Copyright © 1989–1994 Thinking Machines Corporation

11.5 Loop Constructs

The loop constructs (R11-9) specify the repeated execution of a sequence of executable statements. This repeated execution is called a *loop*. The `EXIT` and `CYCLE` statements may be used in some forms of loops to prematurely terminate the execution of the loop or of a cycle of the loop.

A *loop-construct* is one of: R11-9
- *block-loop-construct*
- *nonblock-loop-construct*

All of the loop constructs may be written in either a block form or a nonblock form. The block form is generally considered easier to read, but the nonblock form is provided for compatibility. The execution of loop constructs is described in detail in Section 11.5.5.

11.5.1 Block Loop Constructs

The *block loop construct* (R11-10) is a form of loop terminated by either a `CONTINUE` statement or an `END DO` statement.

A *block-loop-construct* is: R11-10

[c-*name* **:** *]* `DO` *[label] [loop-control-or-while]*
block
[label] loop-termination

A *loop-control-or-while* is one of: R11-11
- *[* **,** *]* *index-variable* = scalar-*num-expr* **,** scalar-*num-expr [* **,** scalar-*num-expr]*
- (scalar-integer-*expr*) `TIMES`
- *[* **,** *]* `WHILE` (scalar-*logical-expr*)

An *index-variable* is: R11-12
scalar-*variable*

A *loop-termination* is one of: R11-13
- `END DO` *[* c-*name]*
- *continue-stmt*

- In a `DO` statement, the *index-variable* must be a named scalar variable of type integer or real. If an `INTEGER*8` is used as the index variable, the total number of iterations cannot exceed (2**31)–1.
- Each scalar-*num-expr* in a *loop-control-or-while* must be of type integer or real.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- If the DO statement of a *block-loop-construct* is identified by a construct name c-*name*, the corresponding *loop-termination* must be an END DO statement referring to the same construct name; if the DO statement is not so identified, the corresponding END DO statement must not refer to a construct name.
- If the DO statement has no *label*, the corresponding *loop-termination* must be an END DO statement.
- If the DO statement has a *label*, the corresponding *loop-termination* must be identified with the same *label*.

The *range* of a block loop construct is the *block* contained within. Transfer of control from outside the range of a block to statements within its range is prohibited. It is permissible to branch to the END DO statement or CONTINUE statement terminating a block loop construct from within the range of that loop construct, but not from outside its range.

The number of iterations of a loop may be determined at the beginning of execution of the loop construct, may be left indefinite, or may be determined by a test executed prior to each iteration of the loop. Execution of an EXIT statement in the range of the loop construct terminates the loop immediately. A particular iteration of a loop may be abbreviated using a CYCLE statement.

The three forms of block loop constructs are described briefly in the following sections. Details of the execution of all forms of loop constructs are given in Section 11.5.5.

11.5.1.1 Block DO Construct

The block DO construct (R11-14) specifies the repeated execution of a sequence of executable constructs while incrementing an *index variable* through successive values for each iteration of the loop. The block DO construct is a form of the block loop construct (R11-10).

A (DO) *block-loop-construct* is: R11-14

```
        DO [ label ] loop-control
          block
[label] END DO
```

A *loop-control* is: R11-15

[,] index-variable = scalar-*num-expr* **,** scalar-*num-expr [* **,** scalar-*num-expr]*

- The syntactic restrictions for *block-loop-construct* (R11-10, p. 140) apply to R11-14.

A construct name may be specified for a block DO construct, as indicated by the syntax specified for *block-loop-construct*.

Copyright © 1989–1994 Thinking Machines Corporation

The statements in the *block* are executed zero or more times. The *index-variable* takes on a succession of values during execution of the loop that depends upon the values of the expressions specified in the *loop-control*.

For a `DO` statement of the form

```
DO, I = N1, N2, N3
```

the index variable `I` takes on values in the range N_1 through N_2, with successive values differing by the value of N_3 (the range is empty if N_1 is greater than N_2). If N_3 is not present, the loop executes as if it were present with the value 1. The value of the index variable must not be changed by any statements within the range of the loop construct. Upon completion, the value of the index variable is $N_1 + N_3 * n$, where n is the number of iterations executed by the loop.

The following block `DO` construct contains another:

```
! Compute a tensor product of two arrays A and B
DO I = 1,M
  DO J = 1,N
    C(I,J) = SUM( A(I,J,:) * B(:,I,J) )
  END DO
END DO
```

11.5.1.2 Block DO TIMES Construct

The block `DO TIMES` construct specifies the repeated execution of a sequence of executable constructs. The block `DO TIMES` construct is a form of the block loop construct (R11-10).

A (DO TIMES) *block-loop-construct* is: R11-16

```
        DO [ label ] ( scalar-integer-expr ) TIMES
          block
[label] END DO
```

- The syntactic restrictions for *block-loop-construct* (R11-10, p. 140) apply to R11-16.

A construct name c-*name* may be specified for a block `DO TIMES` construct, as indicated by the syntax specified for *block-loop-construct.*

The statements of the block contained within the `DO TIMES` construct are executed repeatedly, zero or more times. The scalar integer expression is evaluated at the beginning of the loop to determine the number of loop iterations to execute. This form is useful when no loop variable is required within the range of the loop.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The following two examples of the block DO TIMES construct are equivalent; each calls subroutine DOIT repeatedly the number of times specified.

```
      DO ( N/10+1 ) TIMES
        CALL DOIT
      END DO

      DO 100 ( N/10+1 ) TIMES
        CALL DOIT
  100 END DO
```

11.5.1.3 Block DO WHILE Construct

The block DO WHILE construct (R11-17) specifies the repeated execution of a sequence of executable constructs under control of a scalar logical expression that is evaluated prior to each iteration of the loop. The block DO WHILE construct is a form of the block loop construct (R11-10).

A (DO WHILE) *block-loop-construct* is: R11-17

```
        DO [ label ] [ , ] WHILE ( scalar-logical-expr )
          block
[label] END DO
```

- The syntactic restrictions for *block-loop-construct* (R11-10, p. 140) apply to R11-17.

Note that, unlike other forms of loop, a construct name is not permitted on a *do-while-construct*.

The statements of the block contained within the DO WHILE construct are executed repeatedly, zero or more times. The scalar logical expression is evaluated at the beginning of the loop and prior to successive iterations of the loop to determine whether to execute the next loop cycle. Execution of the DO WHILE statement terminates when the scalar logical expression evaluates to false.

The following two examples of the block DO WHILE construct are equivalent; they call subroutine OUTPUT_TAB repeatedly (zero or more times) until N has been reduced to a value less than 8.

```
      DO WHILE ( N >= 8 )
        CALL OUTPUT_TAB
        N = N - 8
      END DO
```


Copyright © 1989–1994 Thinking Machines Corporation

```
      DO 100, WHILE ( N >= 8 )
        CALL OUTPUT_TAB
        N = N - 8
100   END DO
```

The DO WHILE construct below computes the number of bits required to hold the value of N.

```
      NBITS = 1
      K = 1
      DO WHILE ( K .LT. N )
        K = K * 2
        NBITS = NBITS + 1
      END DO
```

11.5.2 Nonblock Loop Constructs

The nonblock loop constructs are of two forms.

A *nonblock-loop-construct* is one of: R11-18

- *action-terminated-loop-construct*
- *shared-termination-loop-construct*

A nonblock loop construct that is contained within another may have either or both forms.

11.5.2.1 Action-Terminated Loop Constructs

An *action-terminated loop construct* (R11-19) is a nonblock loop construct that is terminated by an action statement other than a CONTINUE statement.

An *action-terminated-loop-construct* is: R11-19

```
        DO label [ loop-control-or-while ]
          loop-body
label     loop-termination-action-stmt
```

A *loop-control-or-while* is one of: R11-20

- *[,] index-variable* = scalar-*num-expr* , scalar-*num-expr [,* scalar-*num-expr]*
- (scalar-integer-*expr*) TIMES
- *[,]* WHILE (scalar-*logical-expr*)

A *loop-body* is: R11-21

[execution-part-construct]...

Copyright © 1989–1994 Thinking Machines Corporation

A *loop-termination-action-stmt* is one of: R11-22

- *assign-stmt*
- *assignment-stmt*
- *call-stmt*
- *forall-stmt*
- *logical-if-stmt*
- *input-output-stmt*
- *pause-stmt*
- *where-stmt*

♦ A *loop-termination-action-stmt* must be identified with a *label*, and that *label* must be referred to in the corresponding `DO` statement of the loop construct.

The *loop-termination-action-stmt* of an action-terminated loop construct is called the *loop termination statement* of that construct.

The *range* of an action-terminated loop construct consists of the *loop-body* and the following loop termination statement. The range is subject to the same restrictions with regard to control transfer as those imposed on blocks. Transfer of control into the *loop-body* or to the loop termination statement from outside the range is prohibited; it is permissible to branch to a *loop-termination-action-stmt* only from within the corresponding *loop-body*.

An action-terminated loop construct, such as

```
      DO 100 [ loop-control-or-while ]
        statements
100     loop-termination-action-stmt
```

has the same effect as the form

```
      DO [ loop-control-or-while ]
        statements
100     loop-termination-action-stmt
      END DO
```

The following action-terminated `DO TIMES` construct calls subroutine `DOIT` repeatedly the number of times specified.

```
      DO 100 ( N/10+1 ) TIMES
100     CALL DOIT
```


Copyright © 1989–1994 Thinking Machines Corporation

The following action-terminated DO WHILE construct below computes the number of bits required to hold the value of N.

```
        NBITS = 1
        K = 1
        DO 100 WHILE ( K .LT. N )
          K = K * 2
100       NBITS = NBITS + 1
```

11.5.2.2 Shared-Termination Loop Constructs

It is possible for the DO statements of several loop constructs to refer to the same label, in which case they are said to *share* the statement identified by that label. Such loop constructs are called *shared-termination loop constructs* (R11-23).

A *shared-termination-loop-construct* is: R11-23

DO *label [loop-control-or-while]*
[execution-part-construct]...
DO *label [loop-control-or-while]*
inner-shared-loop-body
label *loop-termination-shared-stmt*

An *inner-shared-loop-body* is one of: R11-24

- *[execution-part-construct]...*
  DO *label [loop-control-or-while]*
  inner-shared-loop-body
- *[execution-part-construct]...*

A *loop-termination-shared-stmt* is one of: R11-25

- *loop-termination-action-stmt*
- *continue-stmt*

♦ A *loop-termination-shared-stmt* must be identified with a *label*, and all the nonblock DO statements of the *shared-termination-loop-construct* and the *inner-shared-loop-body* must refer to that *label*.

The *loop-termination-shared-stmt* of a shared-termination loop construct is called the loop termination statement of that construct.

The *range* of a shared-termination loop construct consists of all statements following the initial DO statement through and including the corresponding loop termination statement. This range is subject to the same restrictions with regard to control transfer as those imposed on blocks. Transfer of control into the range of the loop from outside its range is prohibited; it is permissible to branch to a *loop-termination-shared-stmt* only from within a corresponding *shared-termination-loop-construct* or *inner-shared-loop-body*.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Several nonblock loop constructs that share a terminating statement, such as

```
        DO 100 [ loop-control-or-while ]
          statements
          DO 100 [ loop-control-or-while ]
            statements
            DO 100 [ loop-control-or-while ]
              inner-shared-loop-body
100           loop-termination-shared-stmt
```

have the same effect as the block form

```
        DO [ loop-control-or-while ]
          statements
          DO [ loop-control-or-while ]
            statements
            DO [ loop-control-or-while ]
              inner-shared-loop-body
100           loop-termination-shared-stmt
            END DO
          END DO
        END DO
```

These equivalent forms are useful for understanding how the nonblock forms of loop constructs are executed.

The following DO constructs share the same loop termination action statement:

```
        ! Compute a tensor product of two arrays A and B
        DO 100 I = 1,M
          DO 100 J = 1,N
100         C(I,J) = SUM( A(I,J,:) * B(:,I,J) )
```

11.5.3 EXIT Statement

The EXIT statement (R11-26) provides one way of terminating execution of a loop.

An *exit-stmt* is: R11-26
EXIT *[* loop-c-*name]*

- If an *exit-stmt* refers to a loop construct name loop-c-*name*, the *exit-stmt* must be within the range of the *loop-construct* named with loop-c-*name*. If a construct name is not present, the *exit-stmt* must be in the range of at least one *loop-construct*.

Copyright © 1989–1994 Thinking Machines Corporation

An EXIT statement is said to *belong* to a particular loop construct. If the EXIT statement refers to a loop construct name, it belongs to that loop construct; otherwise, it belongs to the innermost loop construct in which it appears.

The EXIT statement causes immediate termination of the loop construct to which it belongs. Execution of an EXIT statement effectively branches to the statement immediately following the loop construct to which it belongs. See Section 11.5.5 for more information on the execution of the EXIT statement.

The following block DO construct executes until an end-of-file or an I/O error occurs. Each value of X read from a file is passed to subroutine SUBR. The READ statement is executed at least once, but the call to SUBR may not be executed.

```
DO ! an N-and-one-half DO loop
  READ ( IUN, *, IOSTAT = IOS ) X
  IF ( IOS .NE. 0 ) EXIT
  CALL SUBR( X )
END DO
```

11.5.4 CYCLE Statement

The CYCLE statement (R11-27) provides a means of cutting short the execution of a cycle of a loop.

A *cycle-stmt* is: R11-27
CYCLE *[* loop-c-*name]*

- If a *cycle-stmt* refers to a loop construct name loop-c-*name*, the *cycle-stmt* must be within the range of that *loop-construct*. If the *cycle-stmt* does not refer to a loop construct name, it must be in the range of at least one *loop-construct*.

A CYCLE statement is said to *belong* to a particular loop construct. If the CYCLE statement refers to a loop construct name, it belongs to that loop construct; otherwise, it belongs to the innermost loop construct in which it appears.

Execution of a CYCLE statement causes immediate termination of the *current iteration* of the loop construct to which it belongs. For action-terminated loop constructs, the loop termination statement itself is not executed. For block loop constructs, execution of a CYCLE statement effectively branches to the END DO or CONTINUE statement of the loop construct to which it belongs. For nonblock loop constructs, the CYCLE statement effectively branches to the END DO statement of the corresponding block form of the loop to which the CYCLE statement belongs, where the corresponding block form is obtained from

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

the nonblock form as illustrated in Sections 11.5.2.1 and 11.5.2.2. See Section 11.5.5 for detailed information on the execution of the CYCLE statement.

The following nonblock DO loop executes until an end-of-file or an I/O error occurs.

```
        DO 70
          READ ( IUN, '(1X, G14.7)', IOSTAT = IOS ) X
          IF ( IOS .NE. 0 ) EXIT
          IF ( X < 0. ) GOTO 70
          CALL SUBR( X )
          CYCLE
70      CALL SUBNEG( X )  ! Called only when X < 0
```

11.5.5 Execution of Loop Constructs

A loop construct is said to be *active* when it is being executed, and *inactive* otherwise. A loop construct becomes active only when its DO statement is executed, and becomes inactive when the construct it specifies is terminated. When a loop construct specifying an *index-variable* becomes inactive, the associated *index-variable* retains its last defined value.

There are three phases in the execution of a loop construct: initiation of the loop, execution of the loop range, and termination of the loop.

11.5.5.1 Loop Initiation

When the DO statement of a loop construct is executed, the loop construct becomes active.

If the loop contains a *loop-control-or-while* of the form

index-variable = scalar-*num-expr*$_1$ **,** scalar-*num-expr*$_2$ *[* **,** scalar-*num-expr*$_3$ *]*

then the following steps are performed in sequence:

(1) An *initial parameter* m_1, a *final parameter* m_2, and an *increment* m_3 are established by evaluating scalar-*num-expr*$_1$, scalar-*num-expr*$_2$, and scalar-*num-expr*$_3$, respectively. The resulting values are converted, if necessary, to the type of the *index-variable* according to the rules for numeric conversion in assignment statements (Table 9, p. 119). If scalar-*num-expr*$_3$ is not present, m_3 has a value of one. The value of m_3 must not be zero.

(2) The index variable becomes defined with the value of the initial parameter m_1.

Copyright © 1989–1994 Thinking Machines Corporation

(3) The *iteration count* is established and is the value of the expression

MAX(INT(($m_2 - m_1 + m_3$) / m_3), 0)

Note that the iteration count is zero whenever

$m_1 > m_2$ and $m_3 > 0$, or

$m_1 < m_2$ and $m_3 < 0$

If the loop contains a *loop-control-or-while* of the form

(scalar-integer-*expr*) TIMES

then the scalar-integer-*expr* is evaluated and the resulting value determines the iteration count. If the value is positive, it becomes the iteration count; otherwise, the iteration count is zero.

If there is a *loop-control-or-while* of the form

[,] WHILE (scalar-*logical-expr*)

the loop has no iteration count, but the loop is controlled by the repeated evaluation of the scalar logical expression, as described in the next section.

If the loop contains no *loop-control-or-while*, the loop has no iteration count.

11.5.5.2 Loop Execution Cycle

The *execution cycle* of a loop construct consists of the following steps performed in sequence:

1. The iteration count, if any, is tested. If the iteration count is zero, the loop terminates and the loop construct becomes inactive. If the loop construct begins with a DO WHILE statement, the parenthesized scalar logical expression is evaluated; if the result is false, the loop terminates and the loop construct becomes inactive.

   In the case of a shared-termination loop construct, if as a result of the test, all the loop constructs sharing the loop termination statement are inactive, the execution of all of these constructs is complete. However, if some of the loop constructs sharing the loop termination statement are active, execution continues with step (3) of the execution cycle of the active loop construct whose DO statement was most recently executed.

2. If the iteration count is not zero, or if there is no iteration count, the range of the loop is executed.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

This step may be abbreviated by executing a CYCLE statement from within the range of the loop. This causes immediate execution of step (3) of the current execution cycle of the loop construct to which the CYCLE statement belongs.

In a block loop construct, a transfer of control to the loop termination statement has the same effect as execution of a CYCLE statement belonging to that construct.

In a nonblock loop construct, transfer of control to the loop termination statement causes that statement itself to be executed. Unless a further transfer of control results, step (3) of the current execution cycle of the loop construct is then executed. When a CYCLE statement is executed from within the range of a nonblock loop construct, the loop termination statement is *not* executed, but step (3) of the current execution cycle is executed immediately.

3. The iteration count, if any, is decremented by one. The index variable, if any, is incremented by the value of the increment parameter *m*3.

4. This cycle is executed repeatedly from step (1) until the loop is terminated.

11.5.5.3 Loop Termination

A loop terminates and the associated loop construct becomes inactive when any of the following occurs:

- The iteration count is determined to be zero or the logical expression following the DO WHILE statement is determined to be false, when tested during step (1) of the above loop execution cycle.
- Execution of an EXIT statement belonging to the loop construct.
- Execution of an EXIT statement or a CYCLE statement that is within the range of the loop construct, but that belongs to an outer loop construct.
- Transfer of control to a statement that is out of the range of the loop construct.
- Execution of a RETURN statement within the range of the loop construct.
- Execution of a STOP statement anywhere in the program, or termination of the program for any other reason.

Copyright © 1989–1994 Thinking Machines Corporation

11.5.6 Examples of Loop Constructs

The DO WHILE construct in the following example is terminated when end-of-file or an I/O error is detected. Most of the loop body is skipped if a negative X is read.

```
READ ( IUN, '(1X, G14.7)', IOSTAT = IOS ) X
DO WHILE ( IOS .EQ. 0 )
  IF ( X .GE. 0 ) THEN
    CALL SUBR( X )
  END IF
  READ ( IUN, '(1X, G14.7)', IOSTAT = IOS ) X
END DO
```

The loop below behaves the same as that of the previous example, but only one copy of the READ statement is required, since an EXIT statement has been placed in the interior of the loop. The CYCLE statement eliminates the extra level of IF nesting.

```
DO ! an N-and-one-half DO loop
  READ ( IUN, '(1X, G14.7)', IOSTAT = IOS ) X
  IF ( IOS .NE. 0 ) EXIT
  IF ( X .LT. 0 ) CYCLE
    CALL SUBR( X )
END DO
```

In the following example, the construct name is required in the EXIT statement, but is optional in the CYCLE statements (since each CYCLE statement belongs to its innermost loop).

```
SUM = 0
READ ( IUN ) N
OUTER: DO ( N ) TIMES ! loop MAX( N, 0 ) times
  READ ( IUN ) IQUAL, M, ARRAY(1:M)
  IF ( IQUAL < IQUAL_MIN ) CYCLE OUTER ! skip inner loop
  INNER: DO 40 I = 1,M
    CALL CALCULATE( ARRAY(I), RESULT )
    IF ( RESULT < 0. ) CYCLE ! don't sum negative results
    SUM = SUM + RESULT
    IF ( SUM > SUM_MAX ) EXIT OUTER ! exit both loops
40  END DO INNER
END DO OUTER
```


Copyright © 1989–1994 Thinking Machines Corporation

In the following example, the inner loop is skipped by forcing M to zero. If SUM grows too large, both loops are terminated by branching to the statement labeled 81.

```
        SUM = 0
        READ ( IUN ) N
        DO 80, ( N ) TIMES
          READ ( IUN ) IQUAL, M, ARRAY(1:M)
          IF ( IQUAL < IQUAL_MIN ) M = 0       ! skip inner loop
          DO 80 I = 1,M
            CALL CALCULATE( ARRAY(I), RESULT )
            IF ( RESULT .LT. 0. ) CYCLE ! don't sum negatives
            SUM = SUM + RESULT
            IF ( SUM > SUM_MAX ) GOTO 81
       80   CONTINUE   ! shared by both loops
       81   CONTINUE
```

After execution of the following loop, I = 11, J = 10, K = 6, L = 5, and N = 50.

```
        DO 50, I = 1,10
          DO K = 1,5
            L = K
            N = N + 1  ! This statement executes 50 times
          END DO
  50      J = I
```

After execution the following loop, I = 11, J = 10, K = 5, N = 0, and L is undefined.

```
        N = 0
        DO I = 1,10
          J = 1
          DO 60, K = 5,1  ! This inner loop is never executed
            L = K
    60      N = N + 1
        END DO
```

After execution of the following loop, I = 11, J = 10, K = 6, L = 5, and N = 50.

```
        N = 0
        DO 100 I = 1,10
          J = I
          DO 100 K = 1,5
            L = K
  100         N = N + 1   ! This statement executes 50 times
```


Copyright © 1989–1994 Thinking Machines Corporation

The following example is similar to the previous, except that the inner loop is never executed. After execution, `I = 11`, `J = 10`, `K = 5`, `N = 0`, and `L` is undefined.

```
        N = 0
        DO 200 I = 1,10
          J = I
          DO 200 K = 5,1  ! This inner loop is never executed
            L = K
200           N = N + 1
```

The following are examples of *invalid* loop constructs.

```
        DO I = 1,10 ! INVALID
          ...
        END DO LOOP ! No matching construct name above

        LOOP: DO 100 I = 1,10         ! No matching construct
name below
          ...
100     CONTINUE

        LOOP1: DO
          ...
        END DO LOOP2      ! Construct names don't match

        DO I = 1,10 ! Label required or...
          ...
200     CONTINUE ! ... END DO required

        DO 101 I = 1,10
          ...
102     END DO   ! Labels don't match

        FIRST: DO I = 1,10
          SECOND: DO J = 1,5
            ...
          END DO FIRST    ! Improperly nested DOs
        END DO SECOND
```

11.6 Branching Statements

Branching statements alter the normal execution sequence. A *branch* causes a transfer of control from one statement in a program unit to a labeled statement, called a *branch target*

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

statement, in the same program unit. A branch target statement can be any one of the following:

- an action statement (R3-10, p. 19)
- the `END` statement (R3-11, p. 19) of a program, function, or subroutine
- an `IF–THEN` statement or an `END IF` statement (R11-2, R11-3, p. 135)
- a `SELECT` statement or an `END SELECT` statement (R11-4, p. 137)
- a `DO` statement or a loop termination statement (R11-10, p. 140, R11-19, p. 144, R11-23, p. 146)
- a `WHERE` statement (R10-2, p. 121)

It is permissible to branch to an `END IF` statement from within its `IF` construct, and also from outside the construct. It is permissible to branch to an `END SELECT` statement only from within its `CASE` construct. It is permissible to branch to a loop termination statement only from within its loop construct.

11.6.1 GO TO Statement

The `GO TO` statement (R11-28) causes a transfer of control to a (branch target) statement with the specified label.

A *goto-stmt* is: R11-28
`GO TO` *label*

A *label* is: R2-10
digit [digit [digit [digit [digit]]]]

- The *label* used in a *goto-stmt* must be the statement label of a branch target statement that appears in the same program unit as the *goto-stmt*.

The statement identified by the *label* is executed next.

11.6.2 Computed GO TO Statement

The computed `GO TO` statement (R11-29) causes control to be transferred to one of a list of labels associated with the statement. The statement evaluates a scalar integer expression to determine the label to which it should branch.

Copyright © 1989–1994 Thinking Machines Corporation

A *computed-goto-stmt* is: R11-29

GO TO (*label(s)*) *[,]* scalar-integer-*expr*

- Each *label* in the *label(s)* list must be the statement label of a branch target statement that appears in the same program unit as the *computed-goto-stmt*.

The same statement label may appear more than once in a *label(s)* list.

Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If the resulting value i is such that $1 \le i \le n$, where n is the number of labels in the *label(s)* list, a transfer of control occurs so that the next statement executed is the one identified by the ith label in the list of labels. If i is less than 1 or greater than n, execution continues as though a CONTINUE statement had been executed.

An example code fragment illustrating a computed GO TO statement:

```
      GO TO ( 100, 200, 300 ) ITRANS
      ...
100   CONTINUE
      ...
      GO TO 500
200   CONTINUE
      ...
      GO TO 500
300   CONTINUE
      ...
500   CONTINUE
```

In the example, the statements labeled 100, 200, or 300 will be executed if ITRANS has the value 1, 2, or 3, respectively; otherwise, execution continues immediately after the computed GO TO statement.

11.6.3 ASSIGN and Assigned GO TO Statement

An ASSIGN statement (R11-30) causes a statement label to be assigned to an integer variable. An assigned GO TO statement (R11-31) causes a transfer of control to the branch target statement identified by the statement label currently assigned to the integer variable.

NOTE: The ASSIGN statement is a deprecated feature.

An *assign-stmt* is: R11-30

ASSIGN *label* TO scalar-integer-*variable*

Copyright © 1989–1994 Thinking Machines Corporation

- The *label* must be the statement label of a branch target statement or *format-stmt* that appears in the same program unit as the *assign-stmt*.

The ASSIGN statement causes the statement *label* to be assigned to the scalar-integer-*variable*. While defined with a statement label value, the integer variable must not be referenced in any other context. An integer variable defined with a statement label value may be redefined with a statement label value or an integer value.

When an input/output statement containing the integer variable as a FMT= specifier is executed, the integer variable must be defined with the label of a FORMAT statement.

An *assigned-goto-stmt* is: R11-31

GO TO scalar-integer-*variable [[,]* (*label(s)*) *]*

- Each *label* in the *label(s)* list must be the statement label of a branch target statement that appears in the same program unit as the *assigned-goto-stmt*. The same *label* may appear more than once in the same *assigned-goto-stmt*.

If a parenthesized *label(s)* list is present in an assigned GO TO statement, the label assigned to the integer variable must be one of the statement labels in the list. The variable must be defined with a statement label value only by an ASSIGN statement in the same program unit as the assigned GO TO statement. At the time of execution of an assigned GO TO statement, the integer variable must be defined with the value of a statement label of a branch target statement that appears in the same program unit as the assigned GO TO. The assigned GO TO statement causes a transfer of control to the statement identified by that statement label.

An example of the use of an ASSIGN statement and assigned GO TO statement:

```
      ASSIGN 120 TO BACK   ! set up return point
      GO TO 500
120   CONTINUE          ! resume execution here

500   CONTINUE
      ...
      GO TO BACK
      ...
```


Copyright © 1989–1994 Thinking Machines Corporation

The ASSIGN statement also allows the label of a FORMAT statement to be assigned at run time to an integer variable that can be used later as a FMT= specifier in READ, WRITE, or PRINT statements. An example of this is:

```
      ASSIGN 10 TO IFMT
10    FORMAT( 1XI4 )
      WRITE ( 6, IFMT ) J+3
      ASSIGN 12 TO IFMT
12    FORMAT( 1X20I8 )
      WRITE ( 6, IFMT ) INTS(21:40)
```

11.6.4 Arithmetic IF Statement

The arithmetic IF statement (R11-32) causes control to be transferred to one of three labeled statements depending on the value of a scalar numeric expression.

An *arithmetic-if-stmt* is: R11-32

IF (scalar-*num-expr*) *label* , *label* , *label*

- Each *label* must be the label of a branch target statement that appears in the same program unit as the arithmetic IF statement.
- The *scalar-num-expr* must not be of type complex.

The same *label* may appear more than once in an arithmetic IF statement.

The arithmetic IF statement causes a scalar numeric expression to be evaluated, and causes control to be transferred to one of the statements identified by the first, second, or third labels depending on whether the value is less than zero, equal to zero, or greater than zero, respectively.

Examples of the arithmetic IF statement:

```
IF ( ADDR - OFFSET ) 200,200,300  ! to 300 if ADDR>0, else 200

IF ( OKAY ) IF ( VAL ) 100,200,300  ! don't branch if not OKAY
```

The last example shows the arithmetic IF statement as the action statement of a (logical) IF statement (Section 11.2).

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

11.7 CONTINUE Statement

The CONTINUE statement (R11-33) has no effect on the execution of a program, but provides an "anchor" to which a label can be attached.

A *continue-stmt* is: R11-33
CONTINUE

The CONTINUE statement has traditionally been used as the terminating statement of a loop construct or as a branch target statement.

11.8 STOP Statement

The STOP statement (R11-34) displays a number or message, if specified, and stops program execution.

A *stop-stmt* is: R11-34
STOP *[stop-code]*

A *stop-code* is one of: R11-35
- scalar-char-*constant*
- *digit [digit [digit [digit [digit]]]]*

♦ Leading zero digits in a numeric stop code are ignored.

Execution of a STOP statement causes a program to terminate its execution with a message to standard output (unit number 6) indicating that a STOP statement was reached. The stop code, if specified, is also displayed.

11.9 PAUSE Statement

The PAUSE statement (R11-36) provides a means to suspend temporarily the execution of a program.

A *pause-stmt* is: R11-36
PAUSE *[stop-code]*

Execution of a PAUSE statement causes a program to be suspended with an indication that a PAUSE statement was reached. The stop code, if specified, is displayed to standard output. If no stop code is provided, the program displays the string 'FORTRAN PAUSE'.

Copyright © 1989–1994 Thinking Machines Corporation

Program execution can be resumed by typing “continue” (or an abbreviation consisting of the first few characters), followed by a carriage return. Typing any other character string causes program execution to terminate. If execution is resumed, the execution sequence continues as though a `CONTINUE` statement had been executed.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation