PUDAL DIAGNOSTIC LANGUAGE
SOFTWARE SUBSYSTEM DESCRIPTION
NO. 3 ELECTRONIC SWITCHING SYSTEM

CONTENTS PAGE

CONTENTS PAGE

Figure

1. GENERAL

PURPOSE

1.01 This section describes the Peripheral Unit Diagnostic Assembly Language (PUDAL) used for No. 3 ESS. PUDAL is intended to provide a common high level language which can be used in the generation of diagnostic tests for the No. 3 ESS peripheral units. This section should assist the reader in understanding peripheral unit diagnostic test program listings as well as the control monitor and execution subroutines.

1.02 When this section is reissued, the reason(s) for reissue will be included in this paragraph.

REFERENCES

1.03 The following Bell System Practices are relative to and may aid in the understanding and utilization of this section:

- Section 233-153-115 Trunk, Test, and Service Circuit Diagnostic Tests
- Section 233-153-120 Peripheral Unit Diagnostic Tests
- Section 254-340-102 Basic and Extended 3A CC Instruction Set
- Section 254-340-104 Program Listing Organization and Usage

PUDAL CONTROL AND EXECUTION PROGRAMS

1.04 Two programs implement the major portion of peripheral unit diagnostic control and execution:

(a) Diagnostic Control Program (DCON)—PR-3H266: is the monitor and control portion of the peripheral unit diagnostic tests written in PUDAL.

(b) Diagnostic PUDAL Statement Execution Subroutines (DCNSUB)—PR-3H265: contain the execution subroutines which interpret the PUDAL data tables and convert these tables to 3A executable statements.

2. PERIPHERAL UNIT DIAGNOSTIC TEST ORGANIZATION

TABLE-DRIVEN PROGRAM STRUCTURE

Note: In order to better understand the PUDAL diagnostic language, a short description of the diagnostic organization is presented.

2.01 Most No. 3 ESS peripheral unit diagnostic tests are in a ***table-driven*** structure. A table-driven program consists of three elements:

(a) Data tables

(b) Control program

(c) Interpretive routines.

The relationship of the three elements as applied to peripheral unit diagnostic test programs is shown in Fig. 1. The data table element is composed of the various peripheral unit tests. When a diagnostic test is being run, the control program (DCON) determines which block of data from the data table is to be accessed. A 5-bit code in each data table entry defines a subroutine in DCNSUB which will interpret and operate on data of the table entry in this block resulting in the desired machine operations. When a task is completed, the control program selects the next data table entry and the sequence continues until the test is completed.

2.02 Major advantages realized by using the table-driven program approach are:

(a) A higher level programming language is made available for generating test code.

(b) Once the diagnostic tests are generated they can be used for field testing or factory frame testing with changes required primarily only to the control and interpretive routines.

PAGING DIAGNOSTIC PROGRAMS INTO MEMORY

2.03 Peripheral unit diagnostic programs are not resident in the main store of the control unit as used in the No. 3 ESS. The diagnostic tests are stored on magnetic tape in order to conserve memory. When diagnostics are to be used, they are brought into main memory from the tape cartridge. A segment of main memory, known as the paging buffer, is reserved to accommodate programs that must be brought in from tape and that are not required on a high-usage basis.

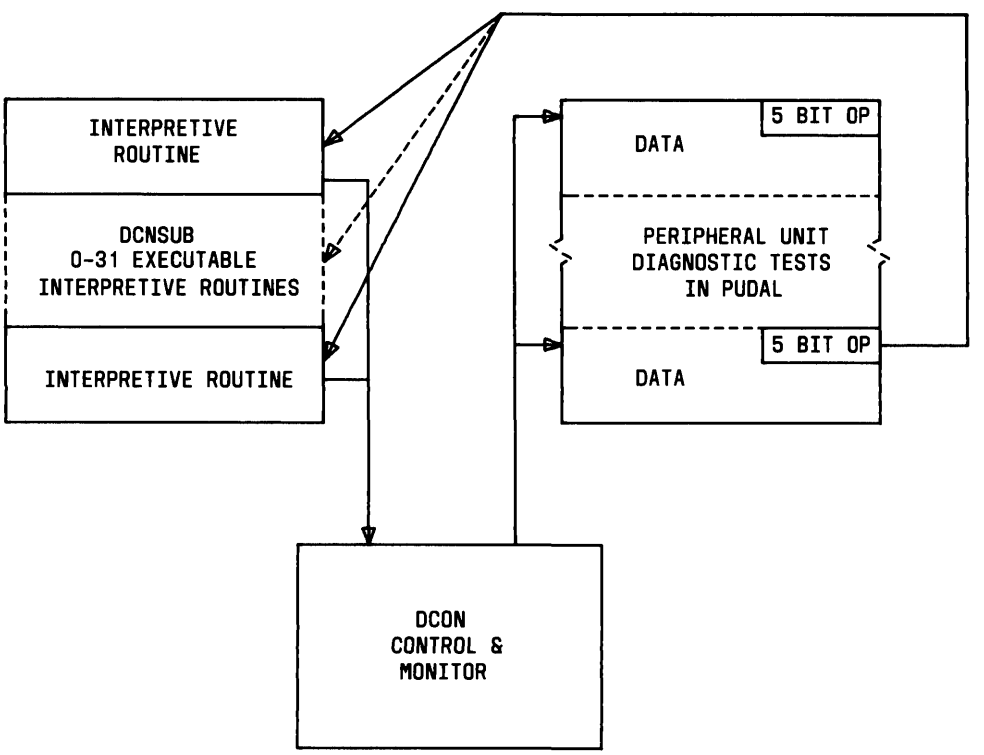


Fig. 1—Table-Driven Program Element Relationship

2.04 When peripheral unit diagnostic tests are requested, program DCON (diagnostic control program), program DCNSUB (diagnostic PUDAL statement execution subroutines), and phase 0 of the diagnostic test are loaded into the paging buffer for the duration of the peripheral unit diagnostic testing. The diagnostic tests, phases 1 through n (data tables), are brought in, one phase at a time, and stored (until executed) in that portion of the paging buffer reserved for diagnostic tests. The size of the diagnostic test phase is limited by the area reserved within the paging buffer.

DCON OPERATION

2.05 DCON is the control and monitor program for the peripheral unit diagnostic tests. A small number of DCON routines are resident in memory to accommodate requests for diagnostic tests and the linkages required for performing daisy chain diagnostic tests.

2.06 When a diagnostic test(s) is requested, the request is analyzed by the resident diagnostic request subroutine. If the request is to be honored, other segments of DCON are brought into the paging buffer. Phase O of DCON contains routines which may be required by all diagnostic tests. Phase 1 is used primarily to:

(a) Determine which diagnostic is to be performed

(b) Initialize the parameter area for the diagnostic program performed by DCON3

(c) Call phase 2A if the diagnostic uses only the on-line control unit

(d) Call phase 2B if the diagnostic uses instructions that may use the on-line or the off-line control unit (and the off-line control unit is available).

2.07 Phase 2A of DCON is entered if the requested diagnostic is to be performed using only the on-line control unit (diagnostic tests for trunks, test circuits, service circuits, etc). Then Phase 3A is entered to determine the entry point for the requested diagnostic. DCON 2A then calls the required phases and begins PUDAL execution. A routine is also available to determine if the diagnostic is in the step or repeat mode. Phase 2A also controls printing of TTY and the lighting of the pass/fail lamps under the step/repeat mode.

2.08 Phase 3A of DCON is used to determine the location of Phase O of each diagnostic test which is run exclusively by the on-line control unit.

2.09 Phase 4A of DCON is entered from 2A when a diagnostic terminates. If the fail flag is set, the failure data is printed. If the abort flag is set, the abort code is printed. If neither flag is set, the success message is printed.

2.10 DCON phases 2B, 3B, and 4B perform similar functions to 2A, 3A, and 4A except they operate in conjunction with diagnostic tests being executed on peripheral controllers.

DCNSUB OPERATION

2.11 DCNSUB contains the interpreting routines for the PUDAL diagnostic language. The routines are grouped for interpreting controller diagnostics and trunk and service circuit diagnostics.

2.12 DCON (the diagnostic controller) calls DCNSUB as a subroutine through one of two entry routines called DCONXA (for trunks and service circuits) or DCONXB (for controllers). These routines read the current operation (op) code, load the instruction into the general registers, and resolve any indirection (indirect specification of data, etc) required. Control is then transferred to the individual routine via a branch long (BL) for execution of the command. Upon completion, control is returned to DCON with

the instruction pointer setting at the next instruction to be executed. A return code is passed to DCON stating that this was either a control statement, a failure, or an abort.

3. PUDAL STATEMENTS WHICH ASSEMBLE DATA TABLE ENTRIES

3.01 This part is provided to document the PUDAL statements available to assemble data table entries. A switching assembly program (SWAP) macro package is used to assemble these PUDAL statements into data table entries which can be interpreted and executed to perform the desired functions. The statements will be defined with respect to operation and syntax as well as showing the assembled data table format.

3.02 The following notational conventions apply to PUDAL statement definition and usage:

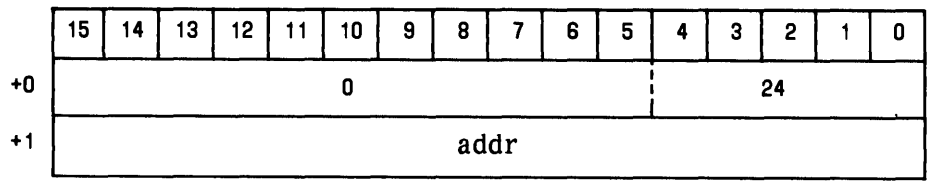
- Brackets [ ] are to indicate an optional field.
- Words in upper case letters are key words and must be entered as shown.
- Parentheses ( ) and the equal sign must be entered as shown.
- Words in lower case letters indicate data to be supplied by the user.
- Except as noted, where operand arguments are shown on the following statement definitions, the absence of the prefix "=" indicates that either a literal value (prefixed by "=") or indirect specification (without the prefix "=") can be made. If the statement operand does not support indirection, a prefix "=" will be shown as part of the operand argument. In this case, the argument must be coded with the prefix "=".

3.03 BRANCH

Function: Branch to an absolute address.

Syntax: This command is intended only for use in patches. It cannot be coded at source level.

Assembled Format:



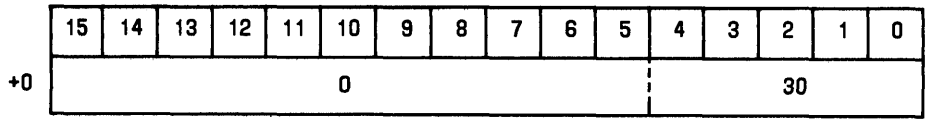
Note: addr is the (16 bit) absolute address of the branch to location.

3.04 BREAK

Function: Cause a real time break, ie, a return to base level.

Syntax: Test Break

Assembled Format:



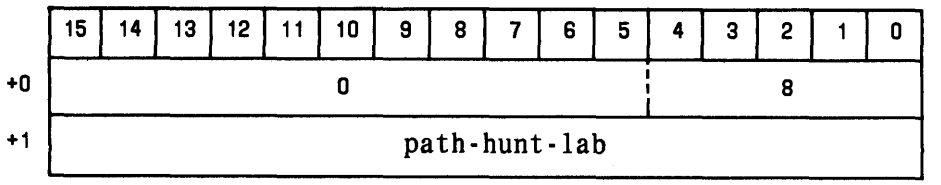
3.05 CONNECT

Function: The CONNECT statement is used to establish connections between circuits via the network fabric. The circuits to be connected have been specified by the PATH_HUNT statement.

Syntax: CONNECT path_hunt_lab

Operands: Path_hunt_lab is the label of the referenced PATH_HUNT statement.

Assembled Format:



3.06 CONVERSE

The CONVERSE statement is invalid as applied to No. 3 ESS diagnostics. It is intended for use in lab and frame testing to issue a message to the operator console. The op code is specified as decimal 25.

3.07 DECLARE

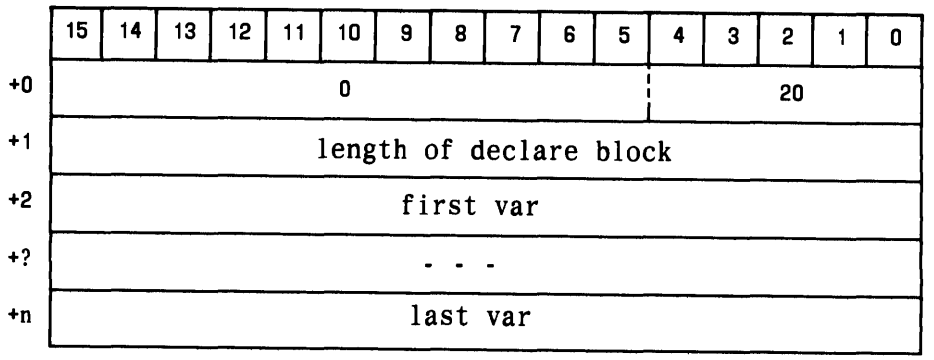
Function: Reverse storage area for variables. Must be in Phase 0.

Syntax: DECLARE varspec1, varspec2, ...

Operands: varspec - one of the following forms:

```
var
var (array_size)
var (INIT [init_var])
var (array_size, INIT [iv1, iv2,...])
```

Assembled Format:



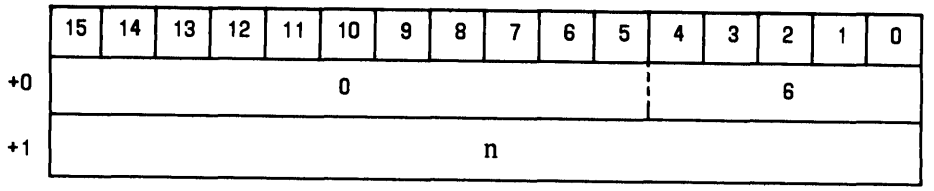
3.08 DELAY

Function: Cause the interpreter program to delay for at least the specified time.

Syntax: DELAY [MS (=n)] DEFAULT: n = 10 if unspecified

Operands: n - the number of 1 millisecond delay intervals.

Assembled Format:



Note: Delay is for at least the specified time interval. Maximum time interval is not restricted.

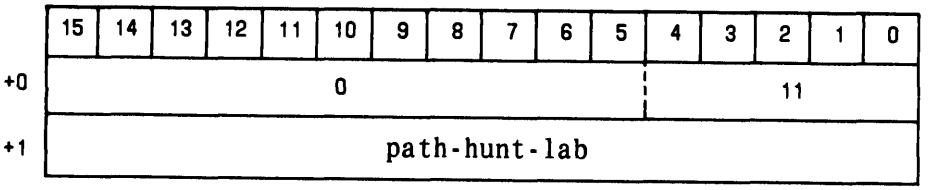
3.09 DISCONNECT

Function: The DISCONNECT statement is used to hardware idle connections established by use of the CONNECT statement. The circuits to be disconnected have been previously specified by the PATH_HUNT statement.

Syntax: DISCONNECT path_hunt_lab

Operands: Path_hunt_lab is the label of the referenced PATH_HUNT statement.

Assembled Format:



DOLOOP—DOEND PAIR

3.10 The DOLOOP—DOEND pair provides a FORTRAN-like loop capability. Execution of the DOLOOP statement initializes the DOLOOP variable to the value specified by the FROM operand argument. When the corresponding DOEND statement is executed, an intermediate result (denoted as IRES hereafter) is created. Its value is the present value of the DOLOOP variable, altered according to the DOLOOP FUNC and BY operand arguments. At this point, a test for the end of the loop is made. If the loop is ***not*** finished, execution is resumed at the statement following the corresponding DOLOOP statement after setting the DOLOOP variable equal to IRES (the intermediate expression). If the loop is finished, execution is resumed at the statement following the DOEND statement without updating the DOLOOP variable.

3.11 The DOLOOP statement, unlike FORTRAN, does not allow indirect specification of the FROM, BY, or TO operand arguments. That is, literal arguments must be specified for these operands. Because of this, some assembly time checks are performed on the DOLOOP statement operand arguments for consistency of the FUNC, BY, TO, and FROM operand arguments. This results in the assembly of a DOLOOP—DOEND pair which will always cause the statements between to be executed at least once.

3.12 The three functions which can be used to modify the DOLOOP variable are ADD, SHIFT, and ROTATE. When the ADD function is specified, IRES is created by adding the value of the BY operand argument to the present value of the DOLOOP variable each time the DOEND statement is executed. If the BY operand argument is positive, the loop stops when IRES is greater than the TO operand argument. If the BY operand argument is negative, the loop stops when IRES is less than the TO operand argument. When the function is SHIFT, IRES is created by shifting the present value of the DOLOOP variable right by the low 4 bits of the BY operand argument if it is positive, or by shifting the present value of the DOLOOP variable left by the low 4 bits of the BY operand argument if it is negative. When the function is ROTATE, IRES is formed by taking the present value of the DOLOOP variable and rotating it right by the low 4 bits of the BY operand argument. This implies left rotate for negative BY operand arguments.

3.13 In case of either the SHIFT or ROTATE FUNC argument, the number of times to execute the "loop" is specified by the TO operand argument and is limited to a maximum of 16 times and a minimum of 1 time.

3.14 DOLOOP variables can be "items" which can affect the way the DOLOOP functions are performed. In particular, when the FUNC argument is ROTATE, the rotate takes place within the width of the "item" which is the DOLOOP variable. When the loop is to continue, the intermediate result is insertion masked into the word containing the "item". Loops can be nested to any depth. Proper nesting is checked at assembly time.

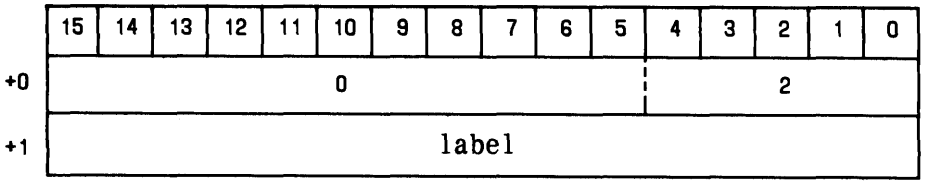
3.15 DOEND

Function: The interpreter program accesses the DOLOOP table entry indicated by the label operand of this statement. The function indicated by the DOLOOP FUNC operand argument is performed on an intermediate result (IRES) using the DOLOOP BY operand argument. A test for the end of the loop is made (see the DOLOOP—DOEND description). If the loop end test indicates the loop is not done yet, the DOLOOP variable is set equal to IRES and control is transferred to the statement immediately following the DOLOOP entry. Otherwise, control is given to the statement following the DOEND statement.

Syntax: DOEND label

Operands: label - must be the symbol used as the label on the DOLOOP statement which this statement is ending.

Assembled Format:



3.16 DOLOOP

Function: When executed, this statement initializes the DOLOOP variable to the value specified by the FROM operand argument.

Syntax: ***Label DOLOOP var***[,FROM(=fv)][TO(=tv)][,BY(=bv)] [,FUNC(fcn)]

Defaults: fv = 1

tv = 1

bv = 1

fcn = ADD

Operands: label—the symbol to be used as the DOEND operand.

var—the DOLOOP variable symbolic reference symbol.

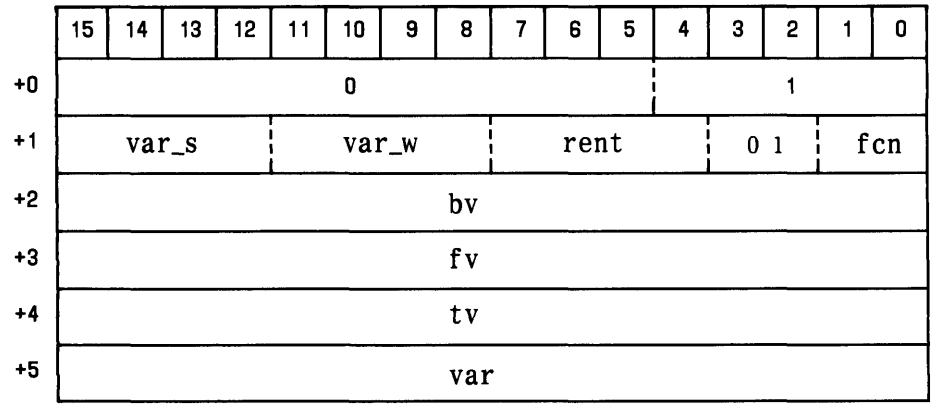
fv—the value to which the DOLOOP variable is to be initialized.

tv—used to determine the end of the loop. For SHIFT and ROTATE functions, it is the absolute number of times to execute the loop.

bv—the quantity to be added or number of places to shift or rotate.

fcn—can be the characters A (for ADD), S (for SHIFT RIGHT), or R (for ROTATE RIGHT).

Assembled Format:



Notes: var_s - shift factor of DOLOOP variable item

var_w—width factor, one of DOLOOP variable item

rcnt - use to count loop executions if fcn is shift (S) or rotate (R)

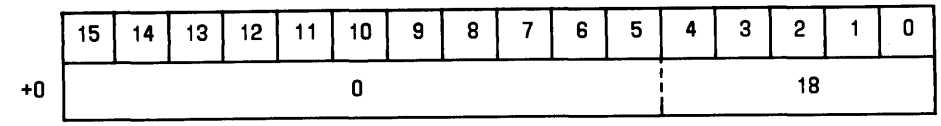
var - offset of DOLOOP variable.

3.17 EPILOGUE

Function: Denotes the beginning of an epilogue.

Syntax: EPILOGUE

Assembled Format:

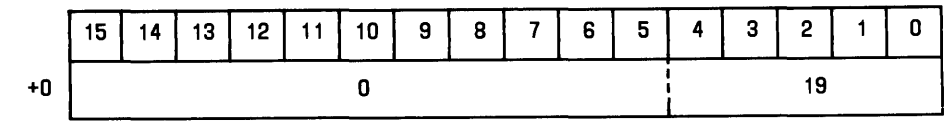


3.18 EPILOGUE_END

Function: Denotes the end of an epilogue.

Syntax: EPILOGUE_END

Assembled Format:



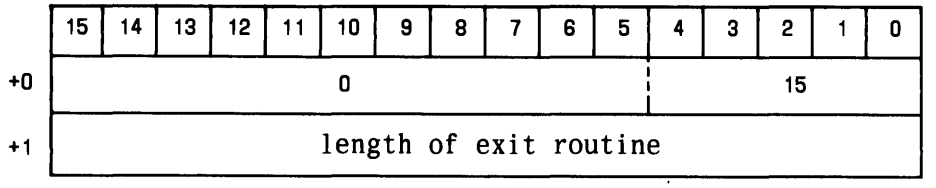
Note: No operands are associated with the EPILOGUE or EPILOGUE_END statements. They are used to delimit code which must always be executed (always at end of phase).

3.19 EXIT

Function: Denotes the beginning of a user-coded routine. The following table entries are to be executed directly by the machine rather than interpreted by the controller.

Syntax: EXIT

Assembled Format:



3.20 EXITEND

Function: Denotes the end of a user-coded routine. The interpreter program reverts back to the interpretive mode.

Syntax: EXITEND

Assembled Formats: No table entry is assembled. This statement is included since it affects the assembly of the EXIT statement.

3.21 GETDATA

Function: Obtain data stored in locations external to this diagnostic (ie, office data)

Syntax: GETDATA var, FROM (fvar) [,LENGTH (length)]

Defaults: length = 1

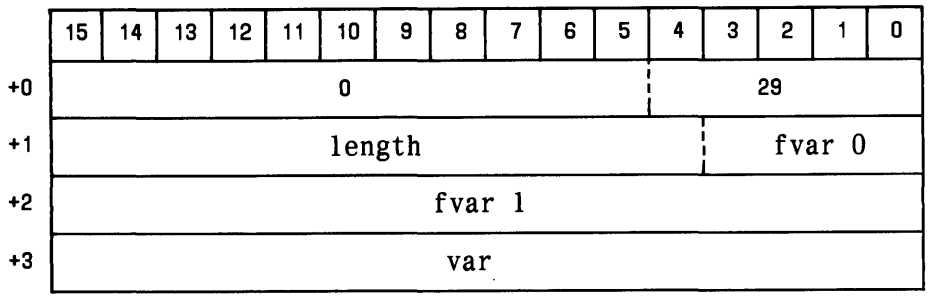
Operands:

var - the symbolic reference symbol (for the starting address) of the destination variable

fvar - the symbolic reference symbol (for the starting address) of the external variable to be moved

length - the number of words to be moved

Assembled Format:



3.22 IF

Function: Cause the interpreter program to conditionally execute I/O instructions based on the truth value of the comparison expression.

Note:

(1) The range of the affected table entries is bounded by this statement and the IFEND statement with the label specified by this statement.

(2) Only I/O to the periphery is inhibited if the comparison expression is true (ie, MOVEDATA and DOLOOP statements are unconditionally executed).

(3) IF—IFEND statements can be nested.

(4) DOLOOP and IF nests are treated identically and must follow the same rules.

Syntax: IF item 1 op item 2 THEN label

Operands:

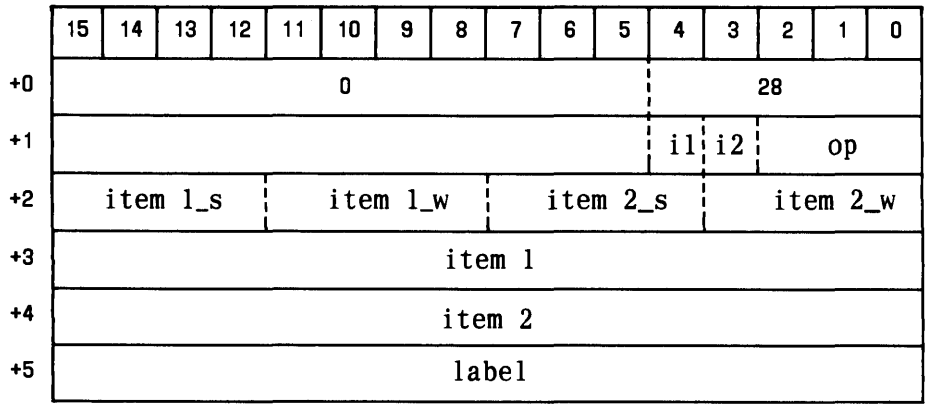
item 1 and item 2 - the two items, variables, or literals to be compared

op - one of the symbols =, >, <, >=, <, or =

label-the symbol to be used as the IFEND operand

i1,i2 - indirect flags for item 1 and item 2

Assembled Format:



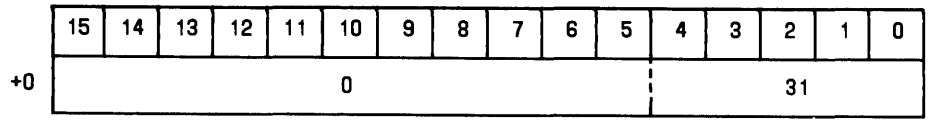
3.23 IFEND

Function: Designates the end of an IF range.

Syntax: label IFEND

Operands: label - must be the same as the label operand of the associated IF statement.

Assembled Format:



3.24 MOVEDATA

Function: Manipulate the contents of variables or items. The manipulation is performed in the following manner. The FROM variable or item offset or the FROM literal value is obtained as an intermediate value. The INDEX variable, item, or literal value is added to this intermediate value to form a new intermediate value. If the FROM operand argument is a literal, this result is used as is in succeeding operations. Otherwise, this result is used as an offset to extract a variable or item (right adjusted) which becomes the new intermediate result to be used in succeeding operations. The fcnvar (which can be a literal, item, or variable) is obtained as a secondary intermediate result. The func (which can be the keywords ADD, SHIFT, ROTATE, AND, OR, XOR, SETBIT, or ZEROBIT) is determined and this FUNCTION is performed upon the first intermediate result using the second intermediate result as the second operand. The only unusual operation is ROTATE where if the FROM operand argument is an item, the ROTATE takes place within the width of this item. The new result is masked by the MASK operand argument (variable, item, or literal) to form the final result. This final result is then insertion masked into the tovar item (if it is an item, otherwise just stored). For the ROTATE and SHIFT operations, positive operand arguments cause right shift or rotate. Negative operand arguments cause left shift or rotate.

Syntax: MOVEDATA tovar,FROM(fvar)[,INDEX (i)] [MASK(mk)] [,func (fcnvar)]

Defaults: fvar = 0

i = 0

mk = X (FFFF)

func = ADD

fcnvar = 0

Operands:

tovar - the destination symbol

fvar - the source symbol

i - the offset to add to the source symbol

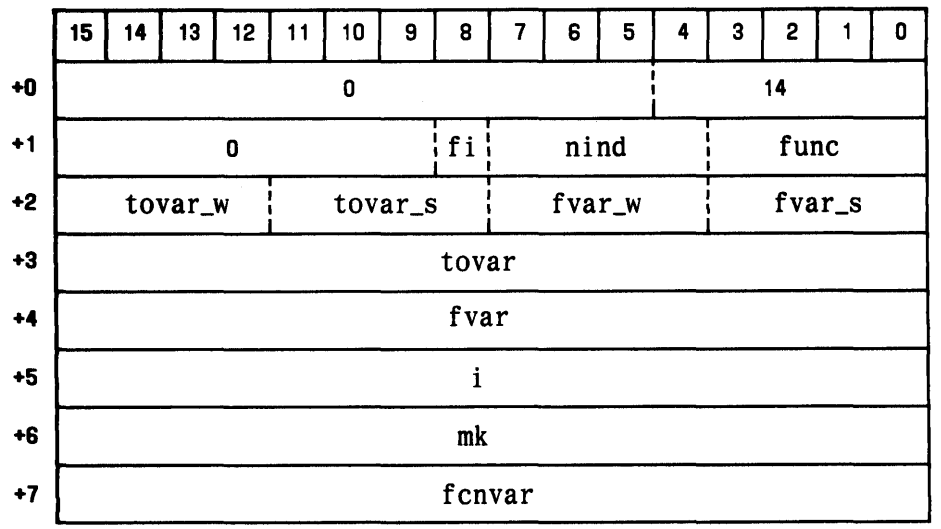
mk - a mask to be ANDed with the final intermediate result

func - arithmetic or logical function to be performed. May be any one of: ADD, AND, XOR, OR, ROTATE, SHIFT, SETBIT, or ZEROBIT.

fcnvar - the argument to be combined with the from argument

fi - fvar indirect flag according to the specified function

Assembled Format:



Notes: nind - number of indirects

tovar_w - to variable width factor

tovar_s - to variable shift factor

fvar_w - from variable width factor

fvar_s - from variable shift factor.

3.25 OPERATE

Function: Set the specified circuit to the indicated state.

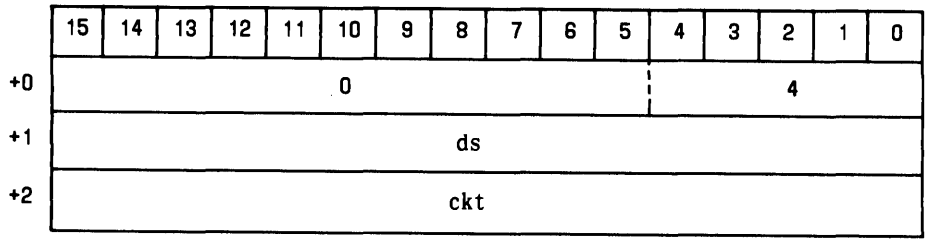
Syntax: OPERATE ckt,STATE(ds)

Operands:

ckt - the circuit whose state is to be set. Must be the object of the previous SEIZE_BLOCK and SEIZE commands.

ds - a literal state (prefixed by =) or the symbolic state, defined by the SEIZE_BLOCK, into which the circuit is to be placed.

Assembled Format:



3.26 PATH_HUNT

Function: Select an idle network path between the specified circuits and make the selected elements busy.

Syntax: label PATH_HUNT ckt 1 to ckt 2 [VIA ckt 3]

Operands:

label - the symbol to be referenced by subsequent CONNECT and DISCONNECT statements.

ckt 1 and ckt 2 - the two circuits to be connected

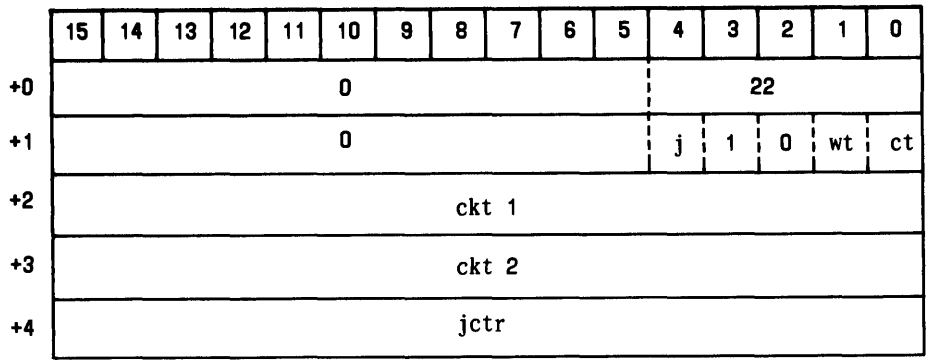
ckt 3 - CTV/WTV/jctr where jctr is a symbolic reference to a SEIZE_JUNCTOR_BLOCK

ct - circuit test vertical (CTV) specified

wt - wire test vertical (WTV) specified

j - junctor (JCTR) specified

Assembled Format:



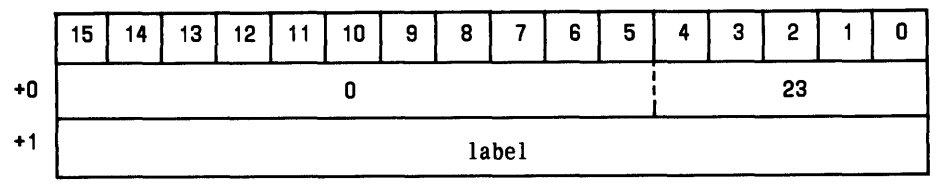
3.27 PATH_RELEASE

Function: Release (make idle) the network elements made busy by a PATH_HUNT

Syntax: PATH_RELEASE label

Operands: label - the label used in the associated PATH_HUNT statement

Assembled Format:



3.28 PD_ORD

Function: Send a series of peripheral pulse distributor (PPD) orders to the PPD with the intent of effecting the operation of a peripheral decoder (PD).

Syntax: PD_ORD [UNIT (unit)] [,POINT (point)] [,NBITS (n)] [,BITS (bits)]

Defaults: point = 0

n = 7

bits = 0

Operands:

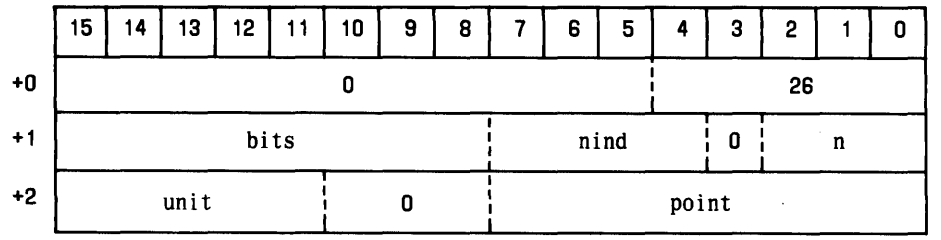
unit - the PPD logical unit

point - the PPD point to be addressed (implies a PD)

n - the number of bits from the BITS operand to send

bits - the right-justified data to be sent to the PD

Assembled Format:



3.29 PHEND

Function: Define the end of a diagnostic phase

Syntax: PHEND [NEXT_PHASE (=next)]

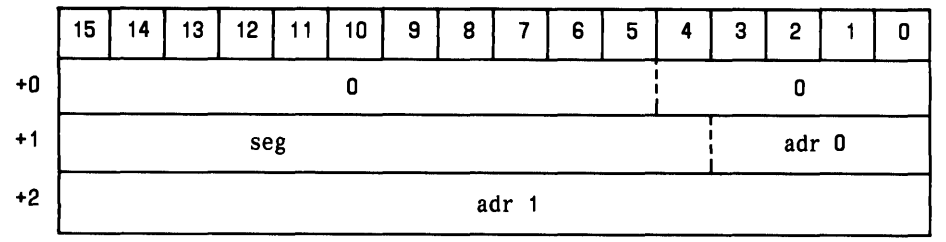
Operands:

next - the next phase number

seg - segment number of next phase

adr - the address of the next phase

Assembled Format:



3.30 PHSTART

Function: Initiate the assembly of a diagnostic phase.

Syntax: PHSTART [PHASE (= phase)] [UNIT(=unit)]

Defaults: phase = 0

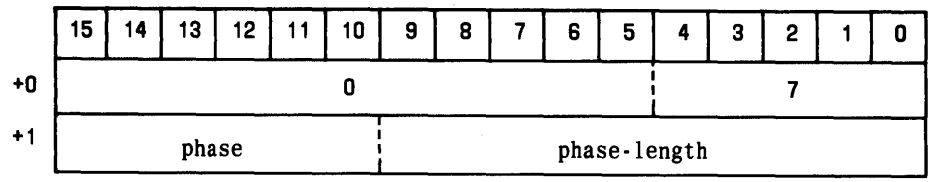
unit = 0

Operands:

phase - the phase number of the phase

unit - specifies the default logical unit for all operations in this phase (if they are not specified)

Assembled Format:

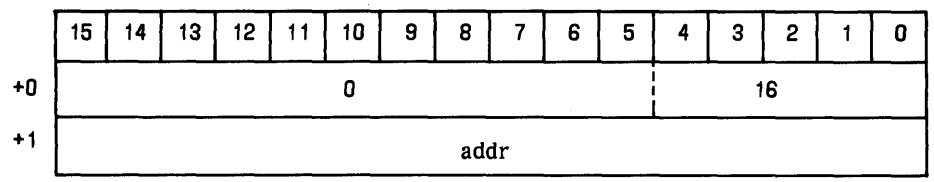


3.31 PROLOGUE

Function: Define the start of a prologue

Syntax: PROLOGUE

Assembled Format:



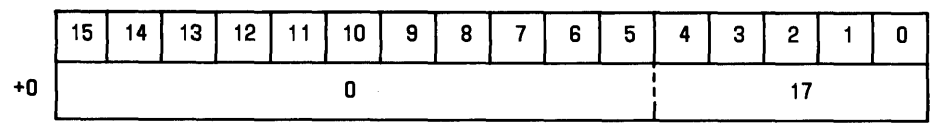
Note: addr is the address of the next epilogue in the phase, prologue, or end of the phase, whichever comes first.

3.32 PROLOGUE_END

Function: Define the end of a prologue

Syntax: PROLOGUE_END

Assembled Format:



Note: No operands are associated with the PROLOGUE or PROLOGUE_END statements. They are used to delimit code which must always be executed.

3.33 PUOP

Function: This is the basic I/O operation. The data specified by the DATA operand argument is transmitted over the channel-subchannel specified by the UNIT operand argument with the start code specified by the SNDSTC operand argument. After an appropriate delay (14 microseconds min), the channel is queried for completion. If the channel did not complete (ie, no reply), the result data is set to all zeros. Otherwise the

result data is set to the reply from the peripheral unit. The result data is exclusive ORed with the data specified by the EXPR operand argument and ANDed with the MASK operand argument. If the result is nonzero, a failure is indicated. The reply start code from the peripheral unit is compared with the EXPSTC operand argument specification and an error is indicated if the actual received start code does not match the states allowed by the EXPSTC argument operand.

Syntax: PUOP [UNIT (unit)] [[,DATA (data)] [,Mask (mk)] [,SNDSTC (sc)] [,EXPSTC (es)] [,EXPR (er)] [,NETQ (=cond)]

Defaults:

unit = the unit specified by PHSTART or, if none, 0

data = 0

mk = X (FFFF)

sc = NORM

es = NORM

er = 0

cond = 0

Operands:

unit - the logical unit to which to direct the data

data - the 16 bits of data to be sent

mk - the 16-bit mask specifying which bits in the response to compare with the expected value.

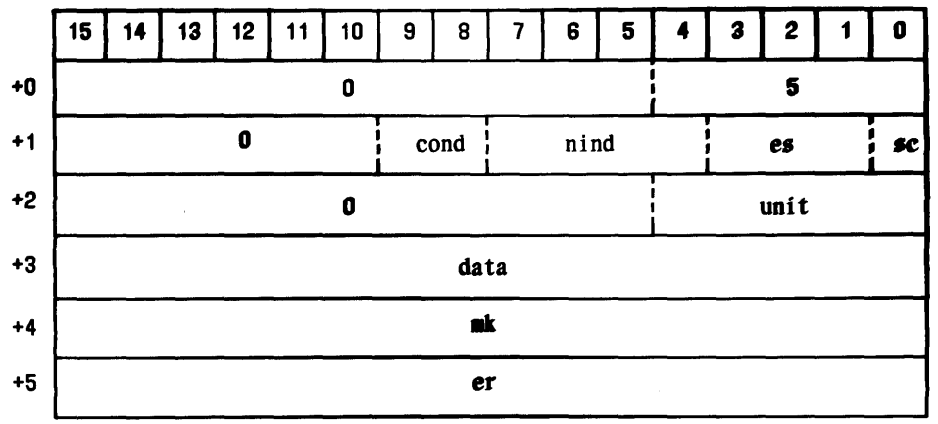
sc - the start code to send. May be either NORMal or MainTenanCe.

es - the expected start code. May be one of NORMal, MainTenanCe, Don't Care, EITHER, or NEITHER

er - the 16 bits of expected response data

cond - (DCON) action to be taken on the call processing network queue. May be one of 0 (no change), 1 (unblock), 2 (block), 3 (block, send message, then unblock).

Assembled Format:



3.34 RELEASE

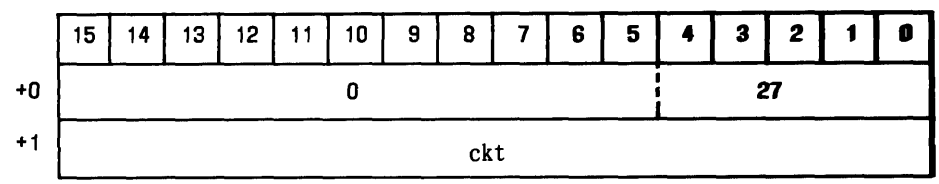
Function: Release (make idle) a previously SEIZEed circuit.

Syntax: RELEASE ckt

Operands:

ckt - the circuit to be released. Must be the object of previous SEIZE_BLOCK and SEIZE commands.

Assembled Format:



3.35 SCAN

Function: Scan the circuit specified and compare the states of its scan points to the expected scan state specified by this statement. A mismatch causes a failure to be indicated.

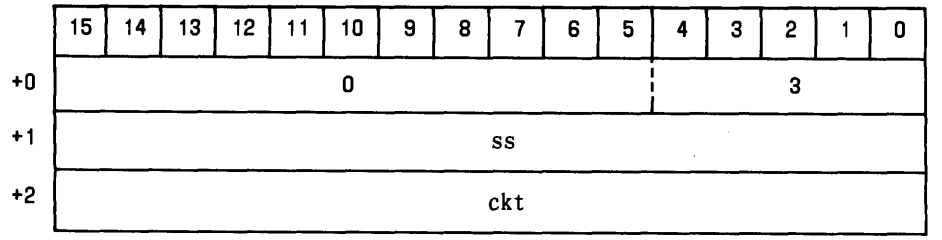
Syntax: SCAN ckt, STATE (ss)

Operands:

ckt - the circuit whose scan points are to be interrogated. Must be the object of previous SEIZE_BLOCK and SEIZE commands.

s - the right-justified expected state

Assembled Format:



3.36 SEIZE

Function: Cause the interpreter program to make the indicated circuit busy.

Syntax: SEIZE ckt [GROUP (gp)] [,MEMBER (mem)]

Defaults: gp = 0

mem = 0

Operands:

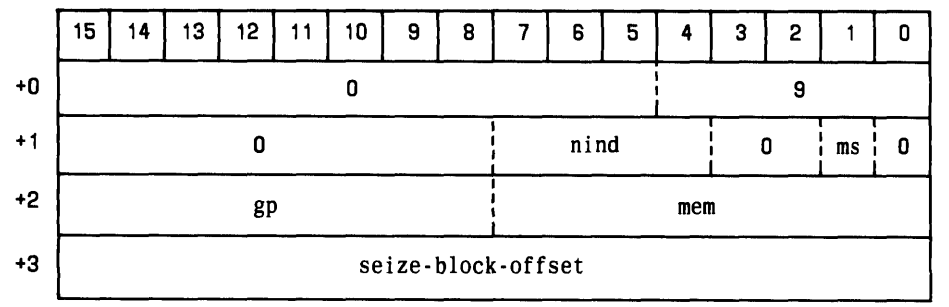
ckt - the circuit to be seized. Must be the object of a previous SEIZE_BLOCK command.

gp - group number of the circuit to be seized

mem - member number of the circuit to be seized

ms - member specified

Assembled Format:



3.37 SEIZE_JUNCTOR

Function: Select and make busy a specified junctor.

Syntax: SEIZE_JUNCTOR jctr (jctrno)

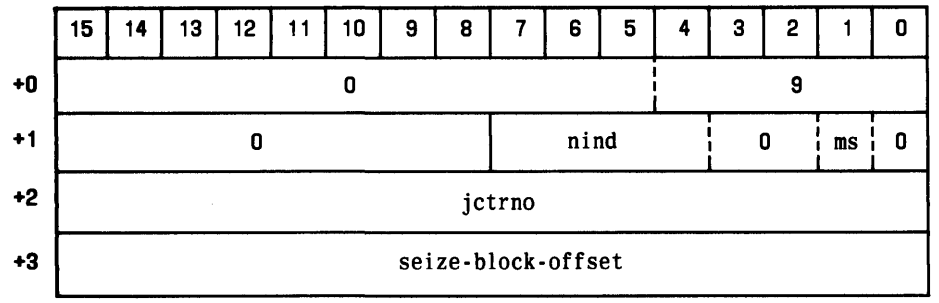
Operands:

jctr - the symbol by which this junctor will be referenced

jctrno - literal, variable, or item specifying the absolute junctor number

ms - member (junctor) specified

Assembled Format:



Note: This statement is similar to the SEIZE statement but provides for absolute junctor selection.

3.38 SEIZE_BLOCK

Function: Provide a store area for the parameters associated with a seized circuit needed by the interpreter. The SEIZE_BLOCK is a specialized DECLARE.

Syntax: SEIZE_BLOCK ckt [,NDPTS(=np)] [,NSPTS (=ns)] [,SPCONST (spc)] [,SS (ss1, ss2,...)] [,DS(ds1,ds2,...)] [,ODA] [SPN (=spn)] [,TEN (=ten)] [,DTA (=dta)] [,SSPN (=sspn)]

Defaults: np = 3

ns = 1

spc = SUP

ODA operands = all null

Operands:

ckt - the symbolic name assigned to the circuit being described

np - number of distribute points associated with the circuit

ns - number of scan points associated with the circuit

spc - scan point construct

ss - list of symbolic scan ferrod states

ds - list of symbolic distribute states

spn - supervisory scan point number

ten - terminal equipment number assigned to the circuit

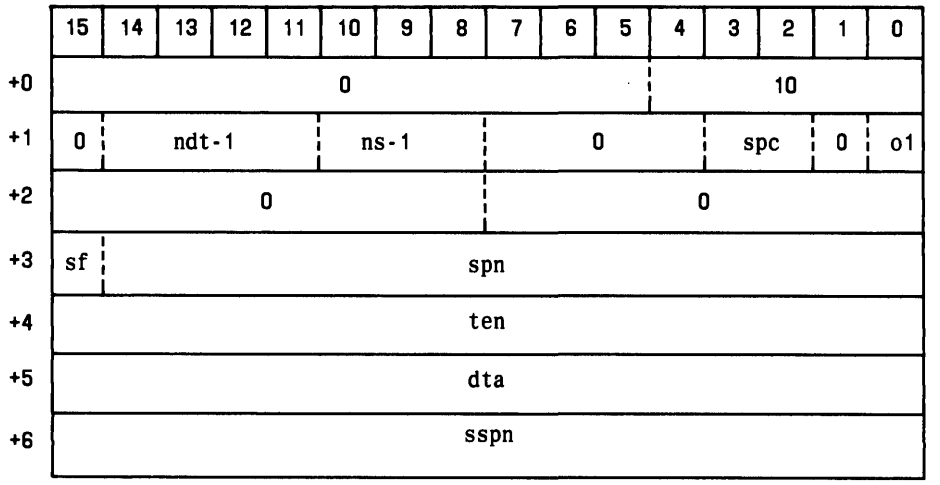
dta - distributor triplet address assigned to the circuit

sspn - supplementary scan point number

ndt - number of distributor triplets associated with this circuit

01 - office data (ODA) directly specified

Assembled Format:



3.39 SEIZE_JUNCTOR_BLOCK

Function: Same as SEIZE_BLOCK

Syntax: Seize-Junctor-Block jctr [,SS(ss1,...)] [,DS (ds 1,...)]

Operands:

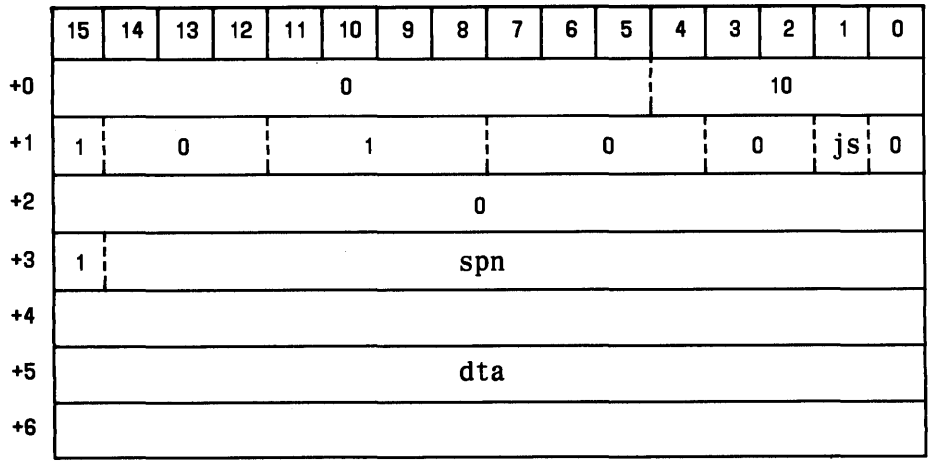
jctr - the symbol by which this junctor will be referenced

ss 1, ... - symbolic scan states

ds 1, ... - symbolic distribute states

js - junctor specified

Assembled Format:



3.40 TEST

Function: Denote the beginning of a new test (for record keeping purposes).

Syntax: TEST [+] n

Operands: n - if + is omitted, the test number for the next test, if + is coded, the amount to be added to the current test number

pl - plus (+) coded

Assembled Format: