VGEN USER'S MANUAL

RELEASE 3.7

Source III, Inc.
3450 Palmer Drive
#4-199
Cameron Park CA 95682
(530) 676-9329 Phone
(530) 676-0932 Fax

TABLE OF CONTENTS

1.0 INTRODUCTION 1

2.0 WHAT IS VGEN? 2

3.0 HOW VGEN WORKS 3

3.1 VGEN Syntax 5

4.0 DATA TYPES AND RULES 6

4.1 PIN Data Type 6

4.2 INTEGER Data Type..7

4.3 STRING Data Type 8

4.4 KEYWORD Data Type 10

4.5 Mixing of INT and STRING Data Types 10

4.6 BUS/VECTOR NOTATION..11

5.0 VGEN STATEMENTS 13

5.1 DECLARATIVE STATEMENTS 13

5.2 PATTERN GENERATION (PG) STATEMENTS 21

5.3 ASSIGNMENT STATEMENTS..27

5.4 EXPRESSIONS 29

6.0 DATA TABLES 31

7.0 VGEN OPERATORS 32

8.0 TIMING AND TIMING CONTROL 33

8.1 SYNCHRONOUS PIN TIMING..34

8.1.1 ASSERT..35

8.1.2 PULSE 36

8.1.3 PINTYPE 38

8.2 ASYNCHRONOUS PIN TIMING...41

8.3 MULTIPLE TIME LINES..41

9.0 STATEMENT LOOP CONTROL 46

10.0 SUBROUTINES AND SUBROUTINE CALLS 49

10.1 PLACEMENT AND CONTROL FLOW...49

10.2 SUBROUTINE PARAMETERS..49

11.0 CONDITIONAL STATEMENTS...51

12.0 COMMENTS...53

13.0 OUTPUT FORMAT CONTROL 55

13.1 USER DEFINED FORMATS 56

14.0 EXTERNAL FILES..58

14.1 INCLUDE FILES 58

14.2 MERGE FILES...58

14.3 EXPECTED OUTPUT FILES...59

14.4 READING EXTERNAL DATA FILES..59

15.0 OUTPUT AND BIDIRECTIONAL PIN DATA...63

16.0 INVOKING AND USING THE VGEN COMPILER..66

16.1 DEBUGGING VGEN PROGRAMS...66

APPENDIX A. - VGEN STATEMENT SYNTAX A-1

APPENDIX B. - TIMING WAVEFORMS...B-1

APPENDIX C. - INTERMEDIATE VECTOR FILE (IVF) FORMAT..................................C-1

INDEX

1.0 INTRODUCTION

The process of designing an ASIC or PCB circuit today, relies heavily on the use of CAD/CAE tools. Computers and software play a very important role in the development process providing both accuracy and productivity that would otherwise be unattainable. For each of the steps involved in designing these circuits, various tools are available, with varying degrees of maturity and effectiveness. One area which has been somewhat neglected historically is the task of creating, modifying and maintaining simulation stimulus files to be used by logic simulators, and later by testers.

As circuits became more and more complex, with gate counts exceeding 10,000 gates and over 100 I/O pins, the number of stimulus patterns (test vectors) necessary to exercise, debug and verify the circuit can become quite large. In most cases, the designer is faced with manually creating these stimulus patterns (usually a table of 1's and 0's) using a text editor. This is a monotonous and error-prone process. Not only are these patterns difficult to create, but modifying them can also be a painful task. Because of this, stimulus pattern files that get created typically fall substantially short of optimal and very seldom incorporate timing that reflects the circuit's real environment.

The Logic Simulators that are on the market today offer the designer a substantial amount of capability in terms of accurate logic and timing simulation, as well as timing checking. A good portion of a circuit's timing begins with the timing provided at the I/O pins. Thus, without a CAD/CAE tool to powerfully and flexibly deal with input stimulus pattern timing, the designer is unable to realize the full benefits of timing checking available in the simulator. VGEN is a high-level stimulus language which fills these important needs.

2.0 WHAT IS VGEN?

VGEN is a high-level stimulus pattern generation language. Its primary objectives are to:

- Significantly reduce the time and monotony of creating, debugging and modifying simulation stimulus files. VGEN provides a programmatic framework for employing hierarchical, modular and algorithmic methodologies in the pattern generation task. This task can typically be reduced from weeks to days or from days to hours.

- Provide a flexible mechanism for incorporating and modifying timing into simulation stimulus files. This, in turn, enables the designer to more fully exploit the capabilities of his logic simulator.

- Provide a mechanism for documenting the flow and logic of stimulus files. Reading through thousands of 1's and 0's, it is very difficult to comprehend the circuit functions being tested. It is quite easy to follow a VGEN program's logic and flow, and it offers flexible commenting capabilities.

- Provide interfaces to multiple simulators. Each logic simulator on the market today has its own special requirements for input stimulus files, none of them are the same. Creating stimulus files with VGEN allows the designer to generate files which are compatible with a large number of popular simulators by simply changing one command in his VGEN program.

In form, VGEN is much like other popular programming languages such as C, Pascal and Basic. However, VGEN has been optimized for the task of creating stimulus pattern files. It contains special constructs and functions that maximize the flexibility and efficiency with which it can accomplish this task. While VGEN provides many powerful features for creating simulation stimulus in a generic simulator-independent form, it also provides facilities for customizing formatting options of specific simulator interfaces. Refer to the VGEN Stimulus Interfaces Guide for information on each of the logic simulator formats currently supported.

3.0 HOW VGEN WORKS

The VGEN program itself is actually an interpreter which "compiles" your VGEN source program directly into a stimulus pattern file whose syntax is compatible with the target simulator. Your source program is a sequence of statements which can be of three basic types (refer to Section 5.0). The example below illustrates a simple VGEN program.

```
INPUTS  PIN4 PIN3 PIN2 PIN1 CLK;
CYCLE   200;
INT     I;
GROUP   P PIN4 PIN3 PIN2 PIN1;
P = HI;
CLK = LO;
VGEN;
FOR I = 0 UNTIL 15 BEGIN
   P = I;  VGEN;
   CLK = HI;  VGEN;
   CLK = LO;  VGEN;
   END
END
```

This simple example illustrates the basic elements of a VGEN program and their typical organization. Each line of the program contains one or more statements, where a statement is terminated by a semicolon (;). The first line defines the pins which are INPUTS to the circuit and for which the designer wishes to define stimulus states. This may include input, output and bidirectional pins. Note that the order in which the pins are listed in the INPUTS, OUTPUTS and BIDIRECTS statements dictates the order that they will be listed in the stimulus pattern file. Multiple INPUTS, OUTPUTS and BIDIRECTS statements may be used to exactly tailor the desired order in the stimulus pattern file.

The second line of the program defines the basic time-step for stimulus pattern generation. In this case, a time increment of 200 time units will be used between patterns. VGEN uses unitless time units which typically are interpreted as nS (nanoseconds). Time units must be integers, thus the smallest positive time unit is 1.

The third line declares an integer variable (I) that will be used by the program. Line four defines a GROUP named P which is composed of pins PIN4, PIN3, PIN2 and PIN1. This provides a means of referring to this group of pins with a single name (P). It also establishes an order for

these pins with PIN4 being the most significant bit (msb) and PIN1 being the least significant bit (lsb) of P. Thus P is defined as a 4-bit vector with PIN4 as its msb and PIN1 as its lsb. We have now declared everything we need for the generation of the stimulus patterns and can proceed with the actual generation itself.

In line 5, we set the group of pins P to a value of "HI". HI is a global pre-defined keyword which refers to a string of all 1's. Thus the four pins collectively referred to by P are all set to a logic 1. The next line uses a similar (but opposite) keyword "LO" which is a string of all 0's to assign the pin CLK a logic state of 0.

Up to this point, we have not actually generated any patterns in our output file. The VGEN statement on line seven is what causes this to happen. Since the time line for the stimulus pattern file defaults to start at time = 0, this VGEN statement will cause a stimulus pattern line or vector to be placed in the output file with a time tag of 0. Such a vector might look like this (for some simulator format):

```
0 - 1 1 1 1 0
```

where the first 0 is the time, the "-" separates time from the pin states, the next four 1's are the states of PIN4 through PIN1 and the 0 at the end is the state of CLK. The actual syntax or format of this pattern vector may be significantly different for different simulators, but the basic information content will not change. Following each VGEN statement, the time is incremented by the value specified in the CYCLE statement (200 in this case).

In line eight of the program, we initiate a program loop using a FOR statement. Here the variable I is used as the loop index with a starting value of 0, a terminating value of 15 and a default increment of 1. The loop consists of all statements between the BEGIN and END keywords. In this case, the loop contains three lines, each having two program statements. The first statement in the loop assigns the current value of the loop index (I) to the pin group named P. When assigning an integer value to a list or group of pins, the state assignment is done on a bit-by-bit basis, starting with the lsb of the group being assigned the state (1 or 0) of the lsb of the integer (in its binary representation). For example, in the sixth pass through the loop (when I = 5) the pins associated with the group P would be assigned the values:

```
PIN4 = 0
PIN3 = 1
PIN2 = 0
PIN1 = 1
```

which corresponds to the binary equivalent of 5 (0101).

Each pass through the loop then results in three output pattern vectors being generated (since there are three VGEN statements in the loop). In the first pattern, the state of the pin group P will have been changed, in the second the CLK pin is set to 1 and in the third, the CLK pin will be set to 0 again. After two passes through the loop, our output file might look as follows:

```
  0 - 1 1 1 1 0
200 - 0 0 0 0 0
400 - 0 0 0 0 1
600 - 0 0 0 0 0
800 - 0 0 0 1 0
1000 - 0 0 0 1 1
1200 - 0 0 0 1 0
```

Note that after each VGEN statement, the time tag associated with each stimulus vector is automatically advanced by an amount defined in the CYCLE statement.

3.1 VGEN Syntax

The syntax that must be used in writing VGEN programs is basically free-form. Blanks, new lines (\n or LF/CR) and tabs are all considered separators and are ignored by the VGEN compiler. Commas may also be optionally used in most cases as separators, however, they are required as separators for subroutine parameter lists. The detailed syntax for each legal VGEN statement is defined in Appendix A. The following sections detail the syntax rules for pin names, variable names, integers, strings, expressions, etc. In general, all VGEN statements should be terminated with a semicolon (;), and are not case sensitive.

4.0 DATA TYPES AND RULES

Within the VGEN language, the objects (or words) used fall into four general data type categories. These are

PIN

INTEGER

STRING

KEYWORD

4.1 PIN Data Type

All pins which are to have a stimulus pattern generated must be declared in an INPUTS, OUTPUTS or BIDIRECTS statement which automatically assigns them a data type of PIN. The names that can be used for pins can be any continuous string of ASCII characters (up to 24 characters in length) with the exception of the following special characters:

" " (space) - used as a separator

"," (comma) - used as a separator

";" (semi-colon) - used as a statement terminator

"=" (equal) - assignment statement separator

"~" (tilda) - signifies compliment

"<>"(angle brackets) - denotes vectors or busses

"[ ]" (square brackets) - denotes vectors or busses

"?", "\", "(", "", "$", ":" - Reserved

"+", "-", "&", "|", "^" - Arithmetic & logical operators

It is legal to use "*" and "/" in pin names, but when you do, a space must surround the symbol whenever it is used in its arithmetic sense.

Since GROUP's are essentially just collections of pins under a single name, GROUP names also have a data type of PIN. GROUP names, however, are limited to a maximum length of twelve characters, but otherwise follow the same rules as for pin names.

Some examples of legal pin and GROUP names are:

```
PIN13
CLOCKPULSE
14MHZ
LAST_GROUP
```

Once they have been declared with INPUTS, OUTPUTS, BIDIRECTS or GROUP statements, pin and group names can be used on the left side of Assignment Statements, or in expressions as any other variable.

4.2 INTEGER Data Type

Any immediate integer number or any integer variable name that has been declared with the INT statement has a data type of INTEGER. Immediate integer numbers (i.e. 11, 7456, -55) can take on a range of values from -[2**(n-1)-1] to 2**(n-1)-1, where n is the word length of the computer (typically 32 bits). Likewise, the values that can be assigned to integer variables is limited to this range. Immediate integer numbers are always specified as decimal radix numbers.

The rules that apply to integer variable names are essentially the same as those that apply to GROUP names, with a maximum length of twelve characters. Examples of INT declarative statements are:

```
INT I, J, K;
INT INDEX1 INDEX2 MAR_KER 0;
```

Note that in the first example, the INT variables are simply declared, but not initialized. In the second example, the variables are initialized to 0.

Three integer variable names are pre-defined in VGEN and should be used only as described below. These are:

```
$TIME , $CYCLE and $FILE_STAT
```

When these variable names are used in expressions, they return a value equal to the current time ($TIME), the current cycle setting ($CYCLE) and the status flag value for the last FREAD command ($FILE_STAT). They can also be assigned a value using the assignment statement, just like any other variable. Some examples of their use is as follows:

```
VECTORS = $TIME + 4000;
$TIME =$TIME + 4000;
$CYCLE = $CYCLE / 2;
WHILE K \<=\ $CYCLE BEGIN . . . END;
```

```
IF ($FILE_STAT \!=\ 0) THEN break;
```

The TIME and CYCLE declaratives can also be used for changing the current TIME and CYCLE values.

4.3 STRING Data Type

This data type is associated with immediate string values and variable names which have been declared with the STRING statement. An object with a STRING data type is represented within the computer as a string of characters which can have values of 1, 0, X, Z or any other legal ASCII character (except single quote). The length of STRING variables can range from 1 to 1024 characters, with the default being 64. Immediate STRING values are specified by a string of characters (usually 1's & 0's) surrounded by single quotes and terminated by an optional radix designator. Legal Radix designators are B (binary), O (octal) and H (hex). Some examples of immediate strings are:

```
'10101101';
'10101101'B;
'AD'H;
'255'O;
'0000010101101'B;
'ZZ'H;
```

The radix designator essentially dictates how many string characters are associated with each character between the quotes, since the strings are always translated to a binary format before being stored. When using strings in assignment statements or tests, high-order bits not defined or specifically designated default to 0. In the first example above, no radix is given and hence the system default radix is used (Hex unless changed by the DEFAULT RADIX statement). Any characters in an immediate string which are not legal for the radix being used map directly into the number of string bits dictated by the radix being used. Thus the last example above would result in the following string of 8 Z's:

```
ZZZZZZZZ
```

Note that the second, third, fourth and fifth strings are exactly equal.

STRING declared variable names follow the same rules as those for INT variable names with a maximum length of 12 characters. Some examples of declaring string variables are given below:

```
STRING CMD1[12], CMD2[12], CMD3[12];
STRING CLR '1010111'B;
STRING OPA, OPB, OPC, 3;
```

Again, the first example declares several string variables, each of length 12, but does not

initialize them. They will automatically be assigned a state of all (12) X's. The second two declarations do not declare a string length but do initialize the variables; CLR gets a 64-character (default length) value of '00 ... 01010111' and OPA, OPB, OPC each get a 64-character value of '00 ... 011'.

When using string variables in assignment statements, expressions or conditional tests, individual bits or ranges of bits within the string can be accessed. This is done using notation similar to that used for pin buses, string[index] for a single bit position, or string[index1:index2] for a range. The index for either case can also be an expression. Some examples are:

```
strs[12] = '1';            { bit 12 of string  strs set to a '1' }
str1[15:12] = 'f';         { bits 15-12 set to '1' }
str2[n:m] = str1[n-2:m-2];
if str1[3:0] \==\ 7 then . . .
```

When assigning (or testing) a shorter length string to a longer length string, higher-order character positions in the shorter string are assumed to be '0'.

Three string variable names are pre-defined in VGEN and should be used only as described below. The three names are:

```
HI
LO
$SIMULATOR
```

The HI string variable is a 1024 character string of all 1's. Likewise LO is a string of all 0's. These are useful for assigning 1 or 0 states to pins, busses or groups. The $SIMULATOR string variable contains a string that is equal to the simulator set using the SIMULATOR declarative. This is sometimes useful for setting switches in a VGEN source program where more than one target simulator is anticipated and slight customizing is required to support each simulator. A typical example of this would be when the high-impedance character used by two simulators is different as illustrated below:

```
string Z 'ZZZZ';
if $simulator \==\ 'advansim_1076'b then Z = 'HHHH';
```

Note that in the test above, the string 'advansim_1076' is used with the b (binary) suffix. This is necessary since it would otherwise use the default hex radix and be interpreted as the string:

```
'10101101vvvv1010nnnnssssiiiimmmm____0001000001110110'
```

4.4 KEYWORD Data Type

As with any programming language, VGEN has a collection of objects (words) that are reserved and hence are given the data type of KEYWORD. These words have special meaning to the VGEN compiler and hence should not be used as pin, group, variable or subroutine names. The keywords recognized by VGEN are:

ANALOG	FORK	SIMULATOR
ASSERT	FREAD	STATE_TRANS
BEGIN	GROUP	STEP
BIDIRECTS	HEADER	STRING
BREAK	IF	SUBROUTINE
BUSFORMAT	INCLUDE	SWITCH
CALL	INITIALIZE_PINS	SYS_CLOCK
CASE	INPUTS	SYSTEM_CALL
COMMENT(S)	INT	TABLE
CONTINUE	KEEP_LOWERCASE	THEN
CYCLE	KEEP_IVF	TIME
DEFAULT	MAKE_SF	TITLE
DEFINE_HEADER	MERGE_BIDIRECTS	TRACE
ECHO	MERGE_FILE	UNITS
ELSE	OUTPUTS	UNTIL
END	PINTIMING	VGEN
EXPECT_FILE	PINTYPE	WAVEFORM
FCLOSE	PULSE	WAVES
FOPEN	REPEAT	WHILE
FOR	SCALE	WHITESPACE

Although VGEN only recognizes these keywords at appropriate places in the program, it would be very risky to use them as names.

4.5 Mixing of INT and STRING Data Types

Within a VGEN program, all pin names, group names, INT and STRING variable names defined at the top program level are global once they have been declared. INT and STRING variables declared within a subroutine are local to the subroutine. Likewise, subroutine parameter names which appear in the parameter list are local to the subroutine. In general, there are few

restrictions in mixing values and variables of INT and STRING data type. Assigning a string value to an integer variable, assigning an integer value to a string variable, performing arithmetic and logical operations between integer and string variables and values will result in an automatic data type conversion prior to the operation to maintain compatibility. Thus, in the following example:

STRING P1[24];

P1 = 7;

The integer 7 is first converted to a 24-bit character string '00 ... 0111' and this is then stored in the string variable P1. The only rule that applies to mixing of data types is that only names of the same data type can appear together on the left side of an assignment statement. This applies to PIN, INTEGER and STRING data types. Since VGEN allows you to list multiple pins or variables on the left of an assignment statement, an error will be generated if the names are not all of the same data type.

4.6 BUS/VECTOR NOTATION

Special characters can be used in VGEN to denote signal busses or vectors. This provides a very flexible and compact way of dealing with system busses.

Pin names which end with a sub-scripted notation are vectored pins. These are essentially pins which have the same name, but differ only by a sub-script. Legal sub-script formats for vectored pins are any of the following:

[n] e.g. BUS[0], BUS[j+1], ...

<n> e.g. BUS<0>, BUS<j>, ...

where n is some positive integer, variable or expression. Another form of vectored pin name includes a list or range of sub-scripts. Legal formats for this form include the following:

[n1,n2,n3,n4] e.g. BUS[3,2,1,0] list

[n1..n4] e.g. BUS[k..0] range

[n1:n4] e.g. BUS[3:0] range

[n1,n2,n3..n4] e.g. BUS[12,10,8..1] list + range

where ni is a positive integer, variable or expression, and the [ ] could be replaced with <>. If the variable k is equal to 3, then the first three of the above are equivalent in that they refer to 4 pins called BUS[3] thru BUS[0]. The order of the sub-script or range numbers is important when defining them with an INPUTS, OUTPUTS or BIDIRECTS Declarative, and may also be important in GROUP or Assignment statements since they define an order for msb and lsb. In general, the msb is the left-most sub-script index and the lsb the right-most sub-script. For example, in the following statement:

GROUP GP1 PIN1, PIN2, PING<4,2,1>;

The lsb of GP1 is the pin PING<1>and the msb is the pin PIN1. Since groups are essentially just a list of pins assembled under a common name, groups may also be treated as vectors. In the above example,

```
GP1<4>  is pin PIN1
GP1<3>  is pin PIN2
GP1<2>  is pin PING<4>
GP1<1>  is pin PING<2>
GP1<0>  is pin PING<1>
```

If a vectored pin or group is referenced by its name only, without the sub-script, it refers to all of the pins in the vectored pin or group. For example, if a vectored pin has been defined in an INPUT statement as PINA<15..0>, then the statement

```
PINA = HI;
```

will set all 16 pins (PINA[15] through PINA[0]) to a logic 1. When using bus vector notation in expressions, the [ ] subscript delimiters must be used to avoid confusing the compiler due to the role that the < and > characters play as logical shift operators.

5.0 VGEN STATEMENTS

Statements in the VGEN language can be divided into three general categories:

DECLARATIVE STATEMENTS

PATTERN GENERATION (PG) STATEMENTS

ASSIGNMENT STATEMENTS

Although Assignment Statements could be included under the PG Statement category, they are addressed separately here due to their importance.

5.1 DECLARATIVE STATEMENTS

VGEN Declarative Statements are those which define things such as pin names, variable names, subroutines, groups, pin timing, etc. These statements are typically found at the beginning of the program, however, many of them can be used anywhere in the program. The only statement ordering rules that must be followed, with the exception of subroutines, is that 1) things must be declared before they can be used in other statements, and 2) all pins must be declared before any VGEN commands are executed. Subroutines may be placed anywhere in the program. The following is a summary of VGEN Declarative Statements (see Appendix A for more details).

5.1.0 ANALOG

Example: ANALOG busname [, VMAX = value] [, VMIN = value] ;

This command is used only with the SPICE analog simulator interface, and is used to define a bus input as a single analog pin where the bits of resolution of the analog pin becomes the width of the bus. The analog value of the pin can then vary between the voltages VMAX and VMIN with this resolution.

5.1.1 ASSERT

Example: ASSERT CLK @ 20:

This declarative is used to define when, relative to the beginning of a VGEN cycle, a pin or pins will assume its new states. If not defined with an ASSERT statement, or other timing statement, pins assume new states at cycle boundaries.

5.1.2 BIDIRECTS

Example: BIDIRECTS db[7:0], mode;

The BIDIRECTS declarative is used declare circuit pins which are bidirectional. It actually

causes two sets of pins to be defined; a set of input pin names (db[7:0] and mode in the above example) and a set of output pin names (db.o[7:0] and mode.o in the example). State values can then be assigned to the input and output versions of the bidirectional pins separately. For those logic simulators which provide an explicit field in the stimulus file for expected output states, the BIDIRECTS command provides a way of dealing separately with these. For logic simulators which do not provide an explicit output field in the vector file, this directive would typically not be used; the bidirectional pins would be listed as INPUTS. The exception to this is if the EXPECT_FILE command is being used to create a separate file with expected output states for later processing with VCAP. The MERGE_BIDIRECTS command can also be used to re-combine bidirectional pin data onto the single input version using a set of precedence rules declared with the command.

5.1.3 BUSFORMAT

Example: BUSFORMAT ADR = HEX, DB = OCT;

Pins that are defined as busses (vectors) in the INPUTS, OUTPUTS or BIDIRECTS declarative can be formatted in the stimulus pattern file as either binary (default), octal or hex. The BUSFORMAT declarative allows you to define the radix to be used.

5.1.4 CYCLE

Example: CYCLE 350 [, all] ;

The CYCLE statement defines the basic time step to be used between VGEN output statements. After each VGEN statement, the time tag for the output patterns is advanced by CYCLE units. Normally, VGEN will output a new vector only if at least one pin has changed state. The optional "all" parameter specifies that a vector should be output every CYCLE, regardless of whether there were any changes in pin states.

5.1.5 DEFAULT RADIX

Example: DEFAULT RADIX OCT;

The DEFAULT RADIX for immediate string values is initially set to HEX by VGEN. With this statement, the default can be changed to any of the three legal radices (HEX, OCT or BIN).

5.1.6 DEFINE_HEADER

Example: DEFINE_HEADER "inputs db[7:0], rw-, addr[4];

outputs stat[4:0], db[7:0];";

The DEFINE_HEADER directive can be used with some simulators to override the normal header placed at the top of the vector file by VGEN. It essentially behaves like a COMMENT statement but has the additional effect of inhibiting the normally generated header. It does not

have any direct effect on the vectors, their format or order.

5.1.7 EXPECT_FILE

Example: EXPECT_FILE "fname" -ascii ;

This command directs VGEN to place any state information defined in the source file for output pins, or the output version of bidirectional pins, in a separate file "fname". This expected output state data file is intended for later use by VCAP for checking simulation results. The data in this file in stored in IVF (Intermediate Vector File) format. The IVF file defaults to its compacted form unless the -ascii option is specified, in which case the expect file is output in IVF ascii format.

5.1.8 GROUP

Example: GROUP GPA PIN1 PIN2 DB[7..0] CLK;

The GROUP statement defines a group of pins to be referenced by a single name (GPA above). The group name is considered as a vectored pin so the order of the pins in the list dictates their relative order in the group vector. MSB is always the left-most pin (first in the list) and LSB, the right-most pin (last in the list). The NULL pin can also be specified in the pin list to skip a bit position in the list.

5.1.9 HEADER

Example: HEADER 100;

This declarative tells the compiler to place pin name labels in the stimulus file as comments (for those simulators that support comments) every n (100 in example) lines. The pin names are listed vertically above their state columns, making the reading of the data in large vector files easier.

5.1.10 INCLUDE

Example: INCLUDE subs_file;

This declarative results in the specified file contents being included (inserted) at this point in the program.

5.1.11 INITIALIZE_PINS

Example: INITIALIZE_PINS inputs '0' ;

This single command can be used to initialize the state of all input and/or output pins to a specified state.

5.1.12 INPUTS

Example: INPUTS BUS[7..0], CLK "attr1", ADR[15..0] "attr2" ;

This statement is used to define the circuit input pins. The order of listing determines the order in which the pins will be listed in the output pattern file. This is usually one of the first statements in a VGEN program. Optional pin attributes can be specified after each pin name, which are used by some of the interfaces.

5.1.13 INT

Example: INT A B C 1;

This declarative is used to define integer variables to be used within the program and optionally set them to an initial value. All variables must be declared before they can be used.

5.1.14 KEEP_IVF

Example: KEEP_IVF;

This declarative prevents the compiler from deleting the intermediate files it creates during compilation. This file(s) contains pin information and a flattened representation of the state and time data to be translated to the desired simulator format. The format of this data is referred to as the IVF (Intermediate Vector File) format and is common to all Simulation Data Management products (See Appendix C).

5.1.15 KEEP_LOWERCASE

Example: KEEP_LOWERCASE;

This declarative keeps the compiler from converting lowercase letters in immediate string data to upper case.

5.1.16 MAKE_SF

Example: MAKE_SF "ckt1.map" ;

When using both VGEN and VCAP for stimulus generation, as well as simulation results verification, this command can be used to provide additional information to VCAP concerning time stamps and line numbers in the VGEN source file.

5.1.17 MERGE_BIDIRECTS

Example: MERGE_BIDIRECTS 0 1H L X Z ;

When assigning state information to bidirectional pins where the timing associated with input pin assignments and expected output states is different, and where the simulator requires bidirectional pin data in a single pin, this command can be used to merge the data from the input version and the output version of the pin onto the input pin.

5.1.18 MERGE_FILE

Example: MERGE_FILE "group1.vec" ;

The MERGE_FILE command tells VGEN to read the state/time data contained in an external file, which must be in IVF format, and merge it in with the vectors being generated by the active source file compilation. This file may have been created by a previous VGEN source file compilation, by a VTRAN translation of other data, or by a text editor manually.

5.1.19 OUTPUTS

Example: OUTPUTS Y<7:0>, SUM, CARRY;

The outputs declarative is similar to the INPUTS declarative, except it is used to define pins which are outputs only. If the target logic simulator supports expected outputs in its stimulus file, then pins declared with this statement are treated as expected outputs and formatted as such in the file. If the EXPECT_FILE command has been specified, then all state/time information for these pins is placed in a separate file for later comparisons with the simulation results data by VCAP.

5.1.20 PINTYPE

Example: PINTYPE NRZ DB[7:0] @ 20 ;

Any pin can be assigned a pintype with this declarative, as well as being assigned timing parameters. The possible pintypes are NRZ, STB, RZ, RO, RC, STB, SBC, BIDR, RX, RZ2X and RO2X. For simulators which integrate the timing information into the vector files, the PIMTIMING ON; command must be used to enable this timing after it is defined.

5.1.21 PULSE

Example: PULSE WE* 30 100 0 ;

Any input pin can be defined as a PULSE pin with this declarative. This means that the pin will pulse from its initial state to an asserted state and then back to its de-asserted state all within one VGEN cycle. The parameters following the pin name (WE*) in the example above indicate the assertion time, the pulse width and the assertion polarity, respectively.

5.1.22 SCALE

Example: SCALE 0.8;

The SCALE statement automatically multiplies all timing values by the scale value. This provides a fast and global mechanism for compressing (or expanding) the timing being used.

5.1.23 SIMULATOR

Example: SIMULATOR ADVANSIM_1076;

The SIMULATOR declarative is used to tell the VGEN compiler what the target simulator is for the stimulus patterns. It will then automatically format the pattern file to be compatible with that simulator. If no SIMULATOR command is specified in the source file, then the resulting data in the vector file will use the IVF generic format with compaction (See Appendix C).

5.1.24 STATE_TRANS

Example: STATE_TRANS outputs '1'->'H', '0'->'L' ;

This command tells VGEN to translate state characters on input and/or output pins. It is useful for simulators which support expected output data in their vector files, where the state characters for outputs is different than 1's and 0's.

5.1.25 STRING

Example: STRING P1, P2[12] 'FA' ;

String variables are declared with the STRING statement. These are variable-length character strings (typically with each character being a 1 or 0). If no length is specified (the [n] suffix) the length defaults to 64 characters. In the example above, the string variables are set to an initial value of '11111010'.

5.1.26 SUBROUTINE

Example:

```
SUBROUTINE ADD(A,B,C) BEGIN
DB = A + B + C;
VGEN;
END
```

The SUBROUTINE declarative is used to define the start of a block of statements which collectively make-up the body of the subroutine. The keyword END terminates the block. The subroutine can have parameters associated with it which are passed inside the parentheses immediately following the subroutine name.

5.1.27 SYS_CLOCK

Example: SYS_CLOCK clk 100 50 0;

This is a special directive which is applicable only to a few specific simulators. It enables the designer to access built-in clock generators available in some simulators which can greatly reduce the number of vectors generated. The parameters specified are: pin name, clock hi time, clock low time and start state. Refer to the VGEN Stimulus Interfaces Guide for information on which simulator interfaces support this feature.

5.1.28 SYSTEM_CALL

Example: SYSTEM_CALL " .. text ..";

Where the text between the quotes is passed to the system upon completion of the VGEN compilation.

5.1.29 TABLE

Example:

```
TABLE TB1 BEGIN
1  4   7  11  15  21 '55'  'ZZ'
'7F'  'FF'  'BC'  0  0
END;
```

This declarative defines a table of data constants. The nth entry can then be accessed as TB1(n), where the first entry is the 0th entry. The data constants may be either immediate integer or string data.

5.1.30 TITLE

Example: TITLE "Circuit1" ;

Several simulators provide facilities for specifying descriptive information in the vector files. This command provides a method by which the user can access these facilities. See the VGEN Stimulus Interfaces Guide for more information.

5.1.31 UNITS

Example: UNITS .1;

Some simulators accept non-integer time in their stimulus vector files. This declarative tells the compiler what the basic units are. See the VGEN Stimulus Interfaces Guide for more information.

5.1.32 WAVEFORM

Example: WAVEFORM PIN1 250 175 1;

This declarative defines PIN1 as having a waveform input with period, pulse width and polarity as stated. This is useful for defining asynchronous, periodic waveforms that run continuously.

5.1.33 WAVES

Example: WAVES ;

Using the Waveform Tool software option, the stimulus vector results of a VGEN compilation can be viewed graphically with this command. After compilation, this command directs VGEN to invoke the Waveform Tool program and pass it the vector data results of the compilation.

5.1.34 WHITESPACE

Example: WHITESPACE '>', '=' ;

When reading state information from external data files, whitespace characters are automatically ignored. The whitespace characters are initially blanks and tabs, but can have additional characters added to the whitespace list with this command.

5.2 PATTERN GENERATION (PG) STATEMENTS

The second category of VGEN statements is the Pattern Generation (or PG) Statements. These are the active statements which directly or indirectly cause stimulus patterns or screen output to be generated. The following is a summary of VGEN PG Statements (See Appendix A for a more detailed description).

5.2.1 BEGIN

Example:

```
BEGIN
 ..
 END
```

This statement is used to mark the start of a series of statements to be treated as a block.

5.2.2 BREAK

Example: IF A\==\7 THEN BREAK;

The BREAK statement is used to exit program loops.

5.2.3 CALL

Example: CALL SUB1(A, 2, '100101'B);

The CALL statement transfers program flow to the subroutine named. Parameters may be passed within parentheses and program flow continues with the statement immediately following the CALL statement after the subroutine returns.

5.2.4 CASE

Example:

```
CASE 7: call zip(x);
        break;
CASE  8: call zap(y);  break;
```

The CASE statement is used in conjunction with the SWITCH statement to delineate cases to be tested.

5.2.5 COMMENT

Example: COMMENT "**TEST 4**";

The COMMENT statement causes the quoted character string to be placed directly into the vector file.

5.2.6 COMMENTS

Example: COMMENTS OFF;

Enables (ON) or Disables (OFF) the COMMENT command, allowing placement of comments for debugging.

5.2.7 CONTINUE

Example: IF C \>\ 12 THEN CONTINUE;

This statement is used to force program flow back to the top of the inner-most program loop.

5.2.8 ECHO

Example: ECHO "Starting generation of ROM test";

The ECHO statement causes the quoted string to be printed on the terminal.

5.2.9 END

The END statement is used to terminate the following:

- FOR loops
- WHILE loops
- REPEAT loops
- BEGIN/END Program Segments
- SUBROUTINES
- SWITCH blocks
- Programs

5.2.10 FCLOSE

Example: FCLOSE fptr1;

This command is used to close an external data file which was previously opened with the

FOPEN command.

5.2.11 FOPEN

Example: FOPEN (fptr1, "fname") ;

Used to open an external data file for reading.

5.2.12 FOR

Example:

```
FOR I = 0 UNTIL 15 BEGIN
ADDR<7..0> = I; VGEN;
CLK = HI; VGEN;
CLK = LO; VGEN;
END ;
```

The FOR statement initiates a FOR loop for which an index variable is defined, its initial value is set, a step increment is optionally defined and its terminal value is stated. All statements between the BEGIN and END keywords are included in the loop.

5.2.13 FORK

Example: FORK b;

The FORK statement is used to switch to an alternate time line for generation of vectors on asynchronous or "parallel synchronous" pins. Up to 5 time lines can be used; A, B, C, D and E. The initial fork defaults to fork A.

5.2.14 FREAD

Example: FREAD (fptr1, " %d > %6H, %4B ; ", ntime, bus, group1);

This command reads a vector line from the file pointed to by fptr1. The quoted string is a format descriptor, with each data variable (identified with the %) being assigned to a program variable or pin(s).

5.2.15 IF/THEN/ELSE

Example:

```
IF a & 1\==\0 THEN CALL MARK0;
ELSE CALL MARK1;
```

This provides for conditional execution of program statements or groups of statements.

5.2.16 PINTIMING

Example: PINTIMING ON DBUS, ADDR;

After defining a pintype and timing for pins using the PINTYPE statement, the defined behavior and timing can be enabled (ON) or disabled (OFF) with the PINTIMING statement.

5.2.17 PULSE

Example: PULSE ON CLK;

This form of the PULSE statement actually enables (ON) or disables (OFF) the pulsing of the specified pin (which has been previously declared as a pulse pin). Once enabled, each VGEN statement will cause the specified pulse to occur on the pin in the stimulus pattern file.

5.2.18 REPEAT

Example:

```
REPEAT 12 BEGIN
CLK1 = HI; CLK2 = LO; VGEN;
CLK1 = LO; CLK2 = HI; VGEN;
END
```

The REPEAT statement defines the beginning of a statement loop to be repeated a specified number of times. All statements between the BEGIN and END keywords are contained in the loop.

5.2.19 SWITCH

Example:

```
SWITCH (xmp) begin
  case 1:        call zip1();   break;
  case 2:        call zip2();   break;
  case 3:        call zip3();   echo "help";   break;
  default:       break;
  end;
```

The SWITCH/CASE structure provides an effective way of dealing with multiple tests in a VGEN program.

5.2.20 TIME

Example: TIME 23500;

This statement is used to alter the current time tag associated with stimulus patterns. The time tag assumes the stated time value.

5.2.21 TRACE

Example: TRACE ON;

This statement is intended as a debugging aid and enables (ON) or disables (OFF) program tracing to the screen. Using TRACE, one can control the portions of a program to be traced.

5.2.22 VGEN

Example: VGEN(4);

This is the actual output statement which causes a stimulus pattern or patterns to be generated in the output file. The optional parameter indicates how many times VGEN is to be executed. Since an output pattern vector is typically only generated if at least one pin has changed state (unless the "all" option was specified in the CYCLE statement), the only time that using a parameter with VGEN would make sense is if a PULSE pin, WAVEFORM or PINTYPE (RZ, RO, RC, . .) has been defined and enabled. Otherwise, a statement like that in the example above would only result in the time tag for the stimulus patterns being advanced by 4 * CYCLE. In the example below:

```
CYCLE 200;
PINTYPE RZ CLK  @ 0, 100 ;
CLK = 1;
PINTIMING ON CLK;
VGEN(16);
```

the result would be the generation of 16 positive CLK pulses, 100 units wide with a period of 200 units.

5.2.23 WAVEFORM

Example: WAVEFORM ON CLKB 33;

This form of the WAVEFORM statement enables (ON) or disables (OFF) the generation of a pre-defined waveform pattern on a pin. The optional number at the end of the statement is an offset from the current time for the waveform to commence.

5.2.24 WHILE

Example:

```
WHILE  y\<=\12  BEGIN
        . . .
    END;
```

The WHILE loop is a general purpose loop statement which continues to loop through the BEGIN/END block as long as the test is true.

5.3 ASSIGNMENT STATEMENTS

The Assignment Statement is the basic mechanism for pin and variable value modification. All Assignment Statements have a left side, followed by an "=" (equal sign), followed by a right side. The right side of an Assignment Statement contains an expression, followed by an optional relative time tag for the assignment. The left side of an Assignment Statement may contain any of the following:

1. pin name or list of pin names
2. group name or list of group names
3. a mixture of the above two
4. an integer variable name or list of names
5. a string variable name or list of names

If the left side of an Assignment Statement contains a list of pin or group names, the order is important. An assignment will be made to this list as a result of evaluating the expression on the right side, converting it to a string data type and assigning the logical state values on a bit-by-bit basis with the lsb of the string going to the right-most pin in the list, etc. In the example below:

```
PIN3 PIN2 PIN1 = 2 + 2;
```

The expression on the right is first evaluated to get an integer value of 4. This is then converted to a string '00 ... 0100' and the pins are then assigned the logic states:

```
PIN1 = 0
PIN2 = 0
PIN3 = 1
```

In the following two examples:

```
DB[7:0] = HI;
DB[7:0] = 1;
```

note that the two are not equivalent. The first results in DB[7] through DB[0] all getting set to a logic 1. In the second, only DB[0] is set to a logic 1, all the other bits get a logic 0 state.

If the left hand side of an Assignment Statement contains a variable (either integer or string) or a list of variables, the assignment of the evaluated expression value on the right is made to each variable name on the left side. For example, if PAT1, PAT2 and PAT3 are previously defined as string variables, then in the following assignment statement:

```
PAT1 PAT2 PAT3 = '8F0F'H;
```

each of the string variables will be assigned the 64-bit pattern '00 ... 01000111100001111'.

When the left hand side of an Assignment statement contains pins and/or groups, then expressions on the right hand side may optionally include a relative time tag which tells the compiler the time offset for making the state assignments. The time tag is relative to the current cycle boundary. The syntax for this type of assignment is:

```
pin or pinlist  =  expr  @  time ;
```

Where time is an immediate integer value, variable or expression. The time offset applies only to this specific state assignment. This feature provides a simple alternative to the ASSERT and PINTYPE methods of dealing with pin timing, but may not map well to some simulators and later to many testers. This type of assignment statement also offers a way of overriding any ASSERT or PINTYPE timing which may be in effect. For example, if a bus has had timing defined as : PINTYPE RX 'X' bus @ 55, 95 ; and was enabled with PINTIMING ON bus ; Then the assignment statements:

```
bus  = 'AF' ;
bus  =  '33'  @ 0;
```

will have significantly different effects. In the first case, the bus will have the value 'AF' assigned to it at time 55 into the cycle and then be assigned the state 'XX' at time 95 into the cycle. The second assignment, however, will cause the value '33' to be assigned to the bus at time 0 (beginning of the cycle) at it will remain in this state until 95, at which time it will be set to 'XX'. Both assignment statements taken together would set bus to '33' at time 0 (cycle boundary), 'AF' at time 55, and then 'XX' at time 95.

Some further examples of Assignment Statements are:

```
A<7..0> = (PA1 | PA2) & PA3;
string1[12:4] = string2[8:0] ;
DB[7:4] = A ^ B ^ C ^ D  @  23 ;
I, J, K = 7;
PIN1 PIN2 P[3..0] = ~PAR  (X + Y);
DB = V1 + ((I*4)<2)  @  db_delay ;
adr[j:k] = pattern1  @  (base_delay + adr_offset) ;
A  =       TBL(n+1)>7;
```

In the last example, a Data Table entry is referenced; the n+ first entry in table TBL.

5.4 EXPRESSIONS

VGEN provides full support for expressions including complex expressions using any of the arithmetic or logical operators and nested parentheses. Evaluation of expressions is always from left-to-right, except as modified by parentheses. Within expressions, data types can be mixed. This includes string, integer as well as pin data types. Conversions are done automatically as needed. Thus, in the example below:

```
DBUS = ((PINA & PINB) + STR1) ^ INT1;
```

if STR1 is a string variable, INT1 an integer variable and PINA, PINB are pins, then type conversions will be done to each prior to performing an operation so that both operands are of equivalent data types. As an example, suppose the operands have the following values:

PINA is a logic 1

PINB is a logic 1

STR1 is '0..01110'B

INT1 is 22

The evaluation will proceed by first ANDing PINA with PINB to produce a 1. Next this will be added to the string '0...01110'B to get '0...01111'B and finally the integer INT1 will be converted to its binary equivalent ('0..010110'B) which is then exclusive OR'ed with the '0...01111'B to give '0...011001'B. This value is then assigned to the bus DBUS starting with the least significant DBUS pin getting the least significant (right-most) binary value. If DBUS is an 8 bit vector then its pin values after the assignment would be:

DBUS[7] = 0

DBUS[6] = 0

DBUS[5] = 0

DBUS[4] = 1

DBUS[3] = 1

DBUS[2] = 0

DBUS[1] = 0

DBUS[0] = 1

There are only two restrictions in using pin names in expressions. The first restriction was mentioned earlier in Section 4.1 That is that if any pin name or groupname uses the characters * or / as part of its name, then whenever using the * or / arithmetic operators in an expression, these operators must be surrounded by a blank space. For example if an INPUTS list has a pin name in it like CLOCK* then the following expressions would produce error messages:

```
X*CLOCK*
CLOCK**4
A*X
```

The correct format for these would be:

```
X  *  CLOCK*
CLOCK*  *  4
A  *  X
```

If neither character is used in any pin, group or variable names then the surrounding spaces are not necessary.

The second restriction is that, while pin, bus and group names may be used in expressions, if bus vector notation is used, then the [ ] delimeters must be used instead of the < > delimeters. It does not matter which are used elsewhere in your program as they can be used interchangeably, except in expressions where the [ ] delimeters must be used. This requirement exists because of the dual role that the < and > characters play as both vector delimeters and logical shift operators. Thus if an INPUTS declarative defines two busses as ADR<7..0> and DB<7..0> then the following expressions would produce error messages:

```
TADR = ADR<4>+3
NDB = DB<7..0>>n
```

On the other hand,

```
TADR = ADR[4]+3
NDB = DB[7:0]>n
```

are legal expressions.

The reason for both of these restrictions is that the characters involved (*, /, > and <) play a dual role in names as well as arithmetic/shift operators.

Expressions can be used in many places within a VGEN program, in addition to the right side of an assignment statement. The formal syntax of each VGEN statement given in APPENDIX A indicates where expressions may be used.

6.0 DATA TABLES

In many applications, it is necessary to drive input stimulus for periods of time from a fixed set of data patterns. This might occur when loading registers within a circuit with initial values via a data bus. VGEN provides a facility for accomplishing this thru its TABLE declarative. A TABLE is essentially a collection of constant data (either integer or string) which can be accessed by way of the table name and an index. The following is an example of using the TABLE declarative:

```
TABLE   INIT_VALUES  BEGIN
   '034'   'F01'
   '214'   '33D'
   '447'   7
    13      22
   END;
```

The data constants within the data table can then be referenced by

```
INIT_VALUES(n)
```

where n = 0 to 7. The index (n) in the above reference can also be an expression, so a typical use might be as follows:

```
FOR  I = 0 UNTIL 7 BEGIN
   WRITE_MODE = 1;
   ADR = OFFSET + I;
   CALL WRITE_REGS (ADR, INIT_VALUES(I));
   END;
```

where WRITE_REGS is a subroutine that loads the sequential values of the data table into registers at address ADR.

7.0 VGEN OPERATORS

The legal arithmetic and logical operators currently supported by VGEN, which can be used in expressions are the following:

+ add the next object

- subtract the next object

* multiply by the next object

/ divide by the next object

& logically AND with the next object

| logically OR with the next object

^ logical exclusive OR with next object

< shift left by next object times (0 fill)

<< shift left by next object times (circular shift on full string)

> shift right by next object times (0 fill)

>> shift right by next object times (circular shift on full string)

All of these operators are binary operators and must appear between two objects. For example:

Q | M + D - A

(A+B) * (M-7)

Here, an object can be any of the following:

1. An immediate integer value
2. An immediate string value
3. An integer variable name
4. A string variable name with optional bit or bit range sub-script
5. A data table name and index
6. A pin or group name with optional bit or bit range sub-script
7. An expression in parenthesis

Any object may be prefixed with the ~ (tilda) character to signify that it is to be complemented. The complementing operation has precedence over the logical and arithmetic operators, but no other operator precedence exists. Expression evaluations proceed left-to-right except as modified by parenthesis.

8.0 TIMING AND TIMING CONTROL

The various methods for defining and controlling pin timing discussed in this section are illustrated in Appendix B.

All logic simulators use the notion of time, or at least event sequencing, as an integral part of their operation. Stimulus patterns, therefore, must typically have the notion of time associated with the various state transitions of its input pins. Each simulator, of course, has its own unique format for associating time to state transitions, but the basic information must be there.

In order to maintain this time association in the stimulus files generated, VGEN uses an internal "time" variable ($TIME) which can be modified with program statements and is maintained by the VGEN compiler. Each time that the program causes a pin to change state, the execution of the next "VGEN" statement causes the pin's new state to be formatted in the output file along with a "time tag" indicating when the change takes place. This "time tag" is derived directly from the "time" variable.

The "time" variable maintained by the VGEN compiler is always initialized to 0 at the start of the program. From this point on, each time that a "VGEN" statement is executed, it is followed by advancing the "time" by an amount dictated by the CYCLE and SCALE declaratives. Actually, "time" will be advanced by the following after each VGEN statement execution:

```
"time" = "time" + (CYCLE * SCALE)
```

If SCALE is not declared in the program, it is defaulted to a value of 1.0. CYCLE defaults to a value of 100 if not declared. As stated earlier, "time" is a unitless number. The following simple program segment example illustrates this concept:

```
INPUTS A B C;
CYCLE 200;
STRING P '001'B;
A B C = P;
VGEN;
A B C = P  <  1;
VGEN;
A B C = P  <  2;
VGEN;
END ;
```

The compilation of this program results in the following stimulus pattern being created (the format is a generic one - for a particular target simulator it would look different but the basic information would be the same):

```
  0 - 0 0 1
200 - 0 1 0
400 - 1 0 0
```

On the left is the "time tag" and on the right are the logic states for the pins A, B and C respectively. If we inserted a SCALE statement:

```
SCALE   .8;
```

after the CYCLE statement and recompiled, the stimulus pattern file would contain:

```
  0 - 0 0 1
160 - 0 1 0
320 - 1 0 0
```

Notice that for each "VGEN" statement encountered, the stimulus pattern file shows the pins assuming their new states at a time which is equal to the current "time" variable maintained by the compiler. The timing as to when pins assume their new logic states is referred to as the pin's assertion time and is a number that is specified relative to the current value of the "time" variable. Thus, in the example above, the pins A, B and C have an assertion time of 0 (the default value) associated with them.

8.1 SYNCHRONOUS PIN TIMING

In real-world synchronous circuits, it is most often the case that input pins change "near" cycle boundaries, but seldom if ever do they all change exactly at the same time right at cycle boundaries. VGEN has several facilities for dealing with synchronous pin timing which can be employed to generate a set of stimulus files that more closely match the real-world timing the circuit will see in its operating environment. The first of these facilities provides for associating assertion times as part of individual assignment statements. Here the assertion time offset is applicable only to the particular assignment statement to which it is attached, and it overrides any other timing that may have been defined for the pin. An example would be:

```
dbus = temp1  @  27 ;
```

where dbus will take on the value in temp1 27 time units after the cycle boundary. See Appendix B - Figure 1. The other methods of modifying the time at which pins assume their new states involve associating global timing parameters with pins which will apply to any state assignments made.ÿThe first and simplest of these is the ASSERT declarative. See Appendix B - Figure 2.

8.1.1 ASSERT

We can define a general formula for when a pin will assume its next state (if it is different than its last state) after encountering a VGEN statement as:

```
"time tag" = "time" + ("assert time" * SCALE)
```

Note that, again, the SCALE is applied to a pin's "assert time". If we now go back to the program segment example given above (leaving SCALE at 1.0) and insert an ASSERT declarative after the CYCLE declarative as follows:

```
ASSERT  A  @ 20 ;
```

Recompiling will result in the following stimulus pattern:

```
  0 - X 0 1
 20 - 0 0 1
200 - 0 1 0
400 - 0 0 0
420 - 1 0 0
```

We can now see that pin A does not assume its new state until 20 time units after the current value of the "time" variable for each "VGEN" statement. Thus, the 20 is relative to the current "time". This "assert time" can also be a negative number, which will result in the pin (or pins) changing state before the current value of "time" (except for the first pattern since the VGEN compiler will not generate a negative "time tag"). The absolute value of an assert time value should never exceed the CYCLE time. In a given VGEN program, up to 18 different ASSERT times can be declared with each being applied to one or more pins. Assert times can also be changed dynamically within the VGEN program. In fact, the value specified in the ASSERT statement can be an expression so that, as shown in the following example, we can cause the assertion time for a signal to sweep thru a range of values:

```
INPUTS A B C;
INT I;
FOR I = 50 UNTIL 10 BEGIN
   ASSERT A @ -I;
   CALL WRITE ();
   END
```

If WRITE is a subroutine which writes the state on pin A into a latch with a clock edge that occurs at the beginning of each VGEN cycle (assert time of 0), then the loop will successively apply the new state of A with a smaller and smaller set-up time. If the simulator supports set-up time checking, it will flag the designer as to what point a violation occurs. As can be seen by the

examples above, the ASSERT declarative provides a very powerful tool for generating stimulus patterns which more accurately reflect a circuit's interface environment. It also allows the designer to more comprehensively utilize the simulator's timing checking capabilities in understanding the circuit's potential performance.

8.1.2 PULSE

Another method of modifying the timing behavior of a pin (or pins) in a stimulus pattern file is through the use of the PULSE statement (see Appendix B - Figure 3). This is essentially a mechanism for putting the state and timing generation on a pin in "automatic" mode. The PULSE statement has two formats. In the first format, the pin to be pulsed and the parameters associated with pulse timing and state are declared. An example is:

```
PULSE  CLK   0   100   1;
```

In this declaration, the first integer after the pin name is the "start" time for asserting the pulse. This is, again, relative to the current value of "time". The second integer indicates the "width" of the pulse, and the third indicates the logical state of the pulse. In this example, each time that a VGEN statement is executed, the pin CLK will be asserted to a logic 1 at time "time" and then de-asserted to a logic state 0 at time "time" + 100. As with the ASSERT statement, the "start" time and "width" time specified in the PULSE statement can be integer variables or expressions. The "start" time value can also be negative, however, the "width" value must always be positive. A maximum of seven PULSE pins can be declared in a given program.

Once a pin has been declared as a PULSE pin, it can then be either enabled or disabled. A pin which is declared as a PULSE pin for the first time in a program always starts out disabled. A pin which is re-declared (possibly with different timing) as a PULSE pin assumes the pin's current enabled/disabled state. Pulse pins can be enabled and disabled within a VGEN program, once they have been declared, using the second PULSE command format as illustrated below:

```
PULSE ON CLK;
   or
PULSE OFF CLK;
```

After executing the first statement above, a pulse (as defined previously) will be generated on the CLK pin every time that a VGEN statement is executed.

While a PULSE pin is enabled, attempting to assign the pin a state with the normal Assignment Statement will have no effect. Once the PULSE pin is disabled with the OFF parameter, however, it behaves just like any other pin. The following example illustrates the use of the PULSE feature:

```
INPUTS A[3..0],  D, WE*;
CYCLE 100;
PULSE WE* 10 50 0;
INT I;
A D = LO; WE* = HI;
VGEN;
PULSE ON WE*;
FOR I = 1 UNTIL 15 BEGIN
   A = I;
   D = I;
   VGEN;
   END
PULSE OFF WE*;
```

Compiling this program will produce a stimulus pattern file that starts out as follows:

```
   0 - 0 0 0 0 0 1
100 - 0 0 0 1 1 1
110 - 0 0 0 1 1 0
160 - 0 0 0 1 1 1
200 - 0 0 1 0 0 1
210 - 0 0 1 0 0 0
260 - 0 0 1 0 0 1
300 - 0 0 1 1 1 1
. . .
etc.
```

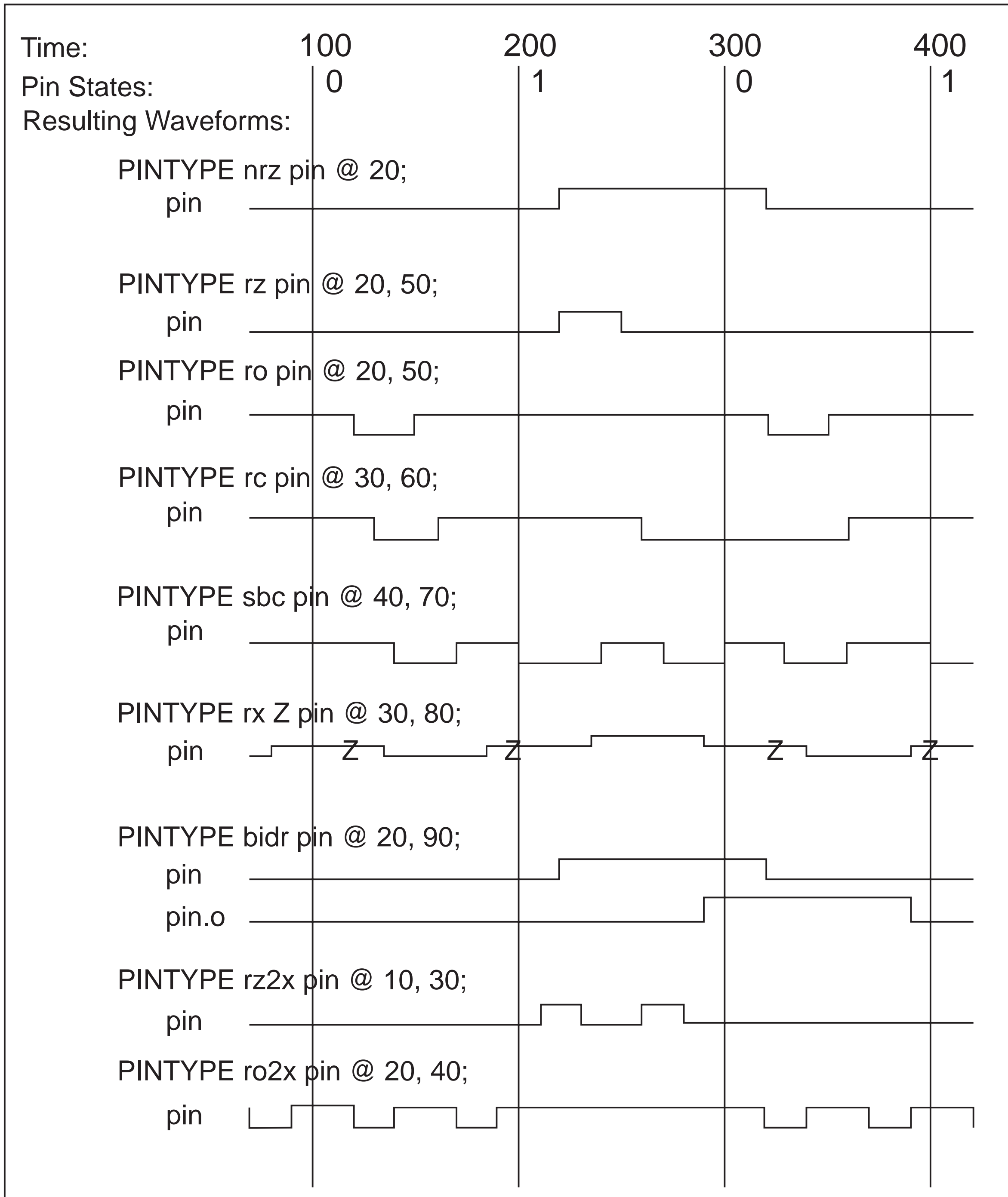
Note that each VGEN statement in the loop results in the generation of three unique "time tagged" patterns. The pins A[3..0] and D assume their new states at the start of each CYCLE step (since their assert times are defaulted to 0). The WE* pin, however, is defined as a PULSE pin and after the first VGEN statement (at time 0) it is enabled. During the course of executing the loop then, each VGEN statement causes WE* to go to a logic 0 state 10 time units after the current "time" and return to a logic state 1 after 50 more time units.

8.1.3 PINTYPE

A more generalized method for dealing with pin timing has been incorporated in VGEN beginning with Release 2.0. This method utilizes concepts commonly used in the test industry and, because of this, make the translation of simulation stimulus patterns into test vectors for physical testers much simpler. This method is actually a super-set of the ASSERT and PULSE methods, and its use is strongly recommended over the alternate methods.

This method of defining and controlling the timing behavior of input pins is based upon the concept of assigning pins a PINTYPE property. The PINTYPES supported are:

NRZ (STB)	Non-return-to-zero (Strobe time for outputs)
RZ	Return-to-zero
RZ2X	Return-to-zero, double pulse
RO	Return-to-one
RO2X	Return-to-one, double pulse
RC	Return-to-compliment
RX 'x'	Return-to-x (x is user-defined)
BIDR	Bidirectional timing
SBC	Surround-by-compliment



PINTYPE Timing Waveforms

When a pin, or group of pins, has been assigned a PINTYPE property, and has been enabled, its timing behavior is dictated by the specific PINTYPE and associated timing assigned (see Figure above and Appendix B - Figure 4). An example of assigning a PINTYPE property to a bus is as follows:

```
PINTYPE  RZ  DBUS @ -20   150;
```

In this example the bus DBUS is defined as RZ. The two numbers following the @ character specify the time, relative to the current "time", at which the bus will assume its new state value and the time it will "return to a value of 0" respectively. After defining the PINTYPE of the bus, one can enable or disable this property by use of the PINTIMING statement. After definition, the property is initially disabled, and behaves like any other pin. Enabling and disabling is done as follows:

```
PINTIMING ON DBUS;
   or
PINTIMING OFF DBUS;
```

In this example, after defining and enabling pintiming, each time a VGEN statement is executed the DBUS will assume its current assigned state 20 time units prior to the current "time" value and return to all 0's 150 time units after. Note that pins on the bus which have been assigned a value of 0 will remain low during this time. This is the case for any of the return-to pintypes; if the pin value assigned is the same as the "return-to" value, then it will not make a transition. The RZ2X and RO2X pintypes generate double pulses during the cycle. The first pulse begins at t1 (first time parameter) and goes to its return-to value at t2 (second time parameter). The second pulse begins and ends at t1 and t2 offset by cycle/2.

Pintype RX is a special, general-purpose return-to function, where the return-to state is user-definable. RX '0' and RX '1' are identical to RZ and RO respectively. The primary purpose of this pintype is to define return-to Z or X pin behavior. In addition, the return-to character '*', has the meaning "return-to the previous state". In this case, the pin takes on its assigned state at t1, and then at t2 it returns to the state it was in prior to t1.

Assigning an NRZ pintype is identical to using the ASSERT statement. The format is:

```
PINTYPE  NRZ  PIN1, PIN2, ABUS @ 50;
```

Note that only one timing parameter is required since the assigned pins make only one transition, at this specified time, and then remain in this state for the remainder of the cycle.

The pintype RC, causes the state of the pin to assume its current cycle state at the first parameter time and then be complemented at the second parameter time. An SBC pintype behaves similarly, except that at the beginning of each cycle the pin is first assigned the compliment of its current cycle state so that the state is always surrounded by its compliment. This is very useful for set/hold time checking.

For pins which have been defined as BIDIRECTS, the BIDR pintype can be used to specify timing behavior. The first time parameter specifies the time offset for driving the input version of the pin, while the second time parameter specifies when expected outputs should be applied to the output version.

The PINTYPE declarative provides an assortment of powerful tools for emulating pin timing in synchronous systems. Pin PINTYPES can also be changed dynamically in a program to modify pin behavior or timing for different tests in a set of stimulus patterns.

8.2 ASYNCHRONOUS PIN TIMING

In many applications, it is necessary to generate periodic patterns on a pin or pins which are asynchronous to the rest of the pins in the circuit. VGEN provides a mechanism for dealing with the generation of patterns on asynchronous pins thru the WAVEFORM declarative. This declarative defines an asynchronous periodic waveform stimulus to be applied to a pin (see Appendix B - Figure 5). The WAVEFORM declarative has the form:

```
WAVEFORM INTR  750 150 1 5250;
```

Here the pin INTR is defined as having a periodic stimulus applied to it with a period of 750 time units, a pulse width of 150 and a polarity of 1. The last parameter is an optional start time for the application of the stimulus in absolute time units. In this case the periodic waveform stimulus will be applied to INTR beginning at time 5250. If no start time is specified then the waveform is initially disabled. After defining a waveform stimulus, it can then be enabled or disabled with the following commands:

```
WAVEFORM ON INTR 327;
          or
WAVEFORM OFF INTR;
```

The last parameter in the ON statement is an optional time offset from current "time" at which to commence the application of the pattern. Up to 18 pins can individually be declared with asynchronous periodic waveform stimulus in a single program. The WAVEFORM facility is most useful in generating asynchronous events, such as an interrupt that occurs periodically, or asynchronous clocks. It can also be used for synchronous clocks if, for example a synchronous circuit has a bus cycle time of 500 so the basic CYCLE is set to this, but it has a fast clock that runs with a period of 100. The WAVEFORM declarative could be used to stimulate the fast clock while the rest of the patterns are synchronized to the 500 time unit CYCLE.

8.3 MULTIPLE TIME LINES

Another powerful mechanism for dealing with asynchronous timing or "parallel-synchronous" timing in VGEN is the FORK statement. This statement allows the user to define independent time lines in his stimulus files which are asynchronous to each other. Up to 5 independent time lines can be defined; A, B, C, D and E with A being the default main time line. In a VGEN program, switching between time lines is done using the FORK statement as follows:

```
FORK B;
```

When the VGEN compiler encounters a FORK statement, it switches context to the stated time line. Since all time lines start out at time = 0, all vectors generated on a new time line will be

origined back to time=0. It is also possible to switch back and forth between time lines as the VGEN program progresses. Each time a switch is done, time is set to the point at which it left off the last time the time line was activated. When the VGEN compiler is done compiling a multiple time line program, it then merges the time lines together to produce a single parallel vector set for the target simulator. The following program segment illustrates the use of multiple time lines:

```
INPUTS BUSA[7..0]  BUSB [7..0]  CKA CKB;
BUSFORMAT HEX;
CYCLE   200;
INT I;
BUSA CKA = LO;   VGEN;
FOR  I = 0  UNTIL 212  BEGIN
   BUSA = I; VGEN;
   CKA = ~CKA; VGEN;
   CKA = ~CKA; VGEN;
   END;
FORK B;
CYCLE    333;
BUSB CKB = LO; VGEN;
FOR I = 1 UNTIL 255 BEGIN
   BUSB = I; VGEN;
   CKB = HI; VGEN;
   CKB = LO; VGEN;
   END;
FORK A;       . . .
```

In this example, two "parallel-synchronous" busses and clocks are being generated. One with a cycle time step of 200 between vectors, the other with a step of 333. The first several lines of the vector file (generic format) would look as follows with the pins in order defined in INPUTS list and buses in hex;

0	-	00	00	0	0
200	-	01	00	0	0
333	-	01	01	0	0
400	-	01	01	1	0
600	-	01	01	0	0
666	-	01	01	0	1
800	-	02	01	0	1
999	-	02	01	0	0
1000	-	02	01	1	0
1200	-	02	01	0	0
1332	-	02	02	0	0
1400	-	03	02	0	0
. . .					
. . .					

When using multiple time lines, the pins being stimulated on a given time line are typically not assigned any states on any other time lines. This avoids having a conflict where two different time lines are attempting to drive a pin to different states at the same time. If this occurs, the merging portion of the VGEN compiler will detect it and generate an error message. It is, however, sometimes necessary to drive a single set of pins (say a bus) from more than one time line. This can be done if certain rules are followed. These rules are as follows:

1) Two time lines may never drive a pin to different logic states (1 or 0) at the same time.

2) After driving a pin, a time line can relinquish driving control of a pin by placing it in a high impedance (Z) state for input pins, or don't care (X) state for output pins. Once this is done, any other time line can take over driving control of the pin and begin driving it. All pins are initially under driving control of FORK A.

In order to effectively switch driving control of a pin or pins between two time lines, it is obviously necessary to have some mechanism to synchronize the event between the two asynchronous time lines. There are two ways to accomplish this. In both methods we FORK back and forth between time lines when we wish to transfer control. The only requirement is that the new fork's "current time" is less than that of the previous fork's. The following program segment illustrates the method:

```
  . . .
FORK  A ;
BUS = LO; VGEN;
BUS = 'ZZ' ; VGEN;

FORK  B ;
VGEN(*) ;
BUS = HI; VGEN;
  . . .
```

Here we have used the VGEN(*) form to bring the fork B time line up to that of the fork A time line. The VGEN statement can be used with an optional parameter such as VGEN(12) or VGEN(X), where the parameter dictates the number of VGEN statements to be executed. When the special * parameter is used, the compiler looks at all of the time lines, finds the one with its "current time" farthest advanced, and generates enough VGEN statements to bring the active fork up to that time.

A second method for synchronizing this event would be to save the "current time" as a variable. Then use the TIME statement in the new fork to bring its "current time" up to this value. This is illustrated below:

```
  . . .
FORK  A ;
BUS = LO; VGEN;
BUS = 'ZZ' ; VGEN;
TEMP = $TIME;

FORK B;
TIME  =  TEMP ;
BUS = HI; VGEN;
  . . .
```

The "current time" of the first fork is saved in the integer variable TEMP and then, after switching to fork B, it is used to bring the new fork's "current time" up to the time at which the previous fork released driving control of the bus. This is done using the TIME statement, which sets the "current time" of the active fork to the time specified. Note that this new time cannot be less than the existing "current time" of the fork as time can never be set back.

The TIME statement is also useful for concatenating separately developed stimulus patterns into one continuous pattern. To do this, one would simply observe the maximum vector time of the first set and then set TIME to this value at the start of the second vector set program prior to compiling it.

9.0 STATEMENT LOOP CONTROL

VGEN provides three mechanisms for defining program loops. Program loops are especially powerful ways of generating large stimulus pattern files when the required patterns can be algorithmically derived. The first, and most general mechanism, is the FOR loop. FOR loops use an index loop variable and have the general form:

```
FOR I = a UNTIL b STEP c BEGIN
   . . .
   . . .
   END
```

where I is the loop index and must have been declared with an INT statement previously. The values a, b and c can be either immediate integer values, variable names or expressions. The " STEP c " is optional and is only needed if the step value per loop iteration is to be different than + or - 1. All statements between the BEGIN and END keywords are part of the loop and will be executed each pass through until the index variable reaches its terminal (b) value or a BREAK statement is executed. Statements contained within a loop can be any PG Statements (including other loop statements), Assignment Statements and any Declarative Statements with the exception of the following:

```
INPUTS (OUTPUTS & BIDIRECTS also)
GROUP
INT
STRING
SUBROUTINE
TABLE
```

Including these declaratives within any section of program that might be executed more than once would result in an error, since the second time they were encountered by the compiler, it would think you were trying to define something with the same name as one already defined.

A second mechanism for defining program loops is the REPEAT statement. It is really just a short-hand form of the FOR loop which requires no dedicated loop index variable and has a fixed step of -1. The format of the REPEAT loop is shown below:

```
REPEAT a BEGIN
   . . .
   . . .
   END
```

where *a* can be an immediate integer value, a variable name or an expression. The statements between the BEGIN and END keywords are executed exactly *a* times. Other than this, the same rules apply to the REPEAT loop that apply to the FOR loop.

A third mechanism for defining program loops is the WHILE statement. This has the form:

```
WHILE test BEGIN
   . . .
   . . .
   END;
```

The statements between BEGIN and END are executed repeatedly as long as "test" is true. The syntax for "test" is exactly the same as for IF/THEN/ELSE statements and is described in Section 11.0 and Appendix A. Some examples would be:

```
WHILE VARA\<=\15 BEGIN
   . . .
   END;


WHILE  (a\<\7) & (adr\!=\0)  BEGIN
   . . .
   END ;
```

In each of these the loop will be repeated as long as the test is true. Note that compound test can be constructed using the logical operators & (AND) and | (OR).

Program loops can also be nested. Program flow within loops will be repeated between the BEGIN and END statements, as described above for each of the three forms, except when either a BREAK or CONTINUE statement is encountered. The BREAK statement will cause the inner-most loop to be terminated and flow passes to the first statement past the END of the loop. The CONTINUE statement causes program flow to jump immediately to the top of the loop where the loop index or "test" is made. An example is given below:

```
   . . .
FOR I=0 UNTIL MAX BEGIN
   . . .
   IF I\==\17 THEN BREAK;
   . . .
   END;
```

Normally the BREAK and CONTINUE statements would be used with a Conditional Statement (IF) in a program loop. The BREAK statement can also be used to force a return from a subroutine when used outside of a loop in a subroutine program segment.

10.0 SUBROUTINES AND SUBROUTINE CALLS

VGEN supports the use of subroutines within a program through the use of the SUBROUTINE and CALL statements. Subroutines provide a way of collecting statements together that need to be repeatedly executed in a fixed order and assigning a name to this collection of statements. The execution of these statements can then be initiated by simply "calling" the subroutine name. The SUBROUTINE declarative has the following format:

```
SUBROUTINE  SUBNAM(a,b,c)  BEGIN
   . . .
   . . .
   END
```

The statements contained within the BEGIN and END keywords comprise the body of the subroutine and are executed each time the subroutine is called. The above subroutine would be invoked with the statement:

```
CALL SUBNAM(d,e,f);
```

10.1 PLACEMENT AND CONTROL FLOW

Subroutine declarations can be placed essentially anywhere in a VGEN program, although it is generally good practice to group them together at either the beginning or the end of the program. Prior to compiling the program, the VGEN compiler scans through and catalogues the name and location of all subroutines. During program execution, the sections of program statements contained within a SUBROUTINE block can only be accessed (executed) by a CALL statement. Once a CALL statement to a subroutine is encountered, flow transfers to the first statement after the BEGIN keyword of the SUBROUTINE and continues until it reaches the END statement for the SUBROUTINE. At this time, the program flow transfers back to the statement immediately following the original CALL statement. As mentioned in the previous section, the BREAK statement can force an early return from a subroutine.

10.2 SUBROUTINE PARAMETERS

As stated earlier, all variables declared at the top level of a VGEN program with either the INT or STRING statements are global variables. INT and STRING declarations which appear inside subroutines become local variables within the context of the subroutine. Variable names used in the subroutine parameter list being passed to the subroutine by the CALL statement, are also local to the subroutine block and need not be declared anywhere in the program. Because of this, subroutine parameter names (as well as local variable names) can be the same in different subroutines. If a subroutine locally declared variable name or parameter name is the same as a globally defined variable name, the local name takes precedence. Thus, in the example below:

```
INT A B;
A = 12;
CALL SUB1(8);
   . . .
   . . .
SUBROUTINE SUB1(A) BEGIN
   B = A;
   END ;
```

the variable B will have the value 8 assigned to it after execution of the CALL statement.

Parameters passed to subroutines by the CALL statement can be either values or pointers. Parameters passed as values can be immediate integer or string values, integer variables or string variables or expressions. Pointers are passed by pre-fixing the desired name (pin, bus, group, table, string or integer variable) with an & (ampersand) character. The number of parameters in the CALL statement must, however, be the same as the number of parameters defined in the SUBROUTINE statement or an error message will be generated. The following examples illustrate some legal CALL statements:

```
CALL SUB1;
CALL SUB2(5, X, &Y);
CALL SUB3(7,  '1010111'B,   &DBUS);
CALL SUB4(A+B, (MAX-1)*2, C);
CALL SUB1();
```

Note that in the first example, if the SUBROUTINE statement which defines the subroutine has no parameters, no parentheses are required in the CALL statement. The last example is equivalent to the first. When the parameter values are passed to the SUBROUTINE, the local parameter names are automatically assigned a data type that is the same as the value being passed to them. Also, when pointers are passed as in the second two examples above, the subroutine parameter names corresponding to the pointers can be used on the left side of subroutine assignment statements.

11.0 CONDITIONAL STATEMENTS

Two types of statements which control the execution of statements (or blocks of statements) based upon the results of a "test", are supported by VGEN. The first of these is the IF/THEN/ELSE statement. Using this statement puts a condition on whether or not a program statement, or group of statements, are executed. This conditional statement has the form:

```
IF test THEN statement1;
ELSE statement2;
   . . .
   . . .
```

Where "test" is the evaluation of a relationship between two expressions and "statement" is any legal VGEN statement (including another IF/THEN/ELSE statement) or a group of statements surrounded by the BEGIN/END keywords. Compound "tests" can be constructed using the & (AND) and | (OR) logical operators between the individual tests which are each enclosed in parentheses. In words, the conditional statement works as follows:

> IF test (a relationship between two expressions) is true THEN execute statement1, otherwise (ELSE) execute statement2.

The legal tests that can be made between two expressions are as follows:

\==\	equal to
\<=\	less than or equal to
\>=\	greater than or equal to
\<\	less than
\>\	greater than
\!=\	not equal to

Some examples of tests and compound tests using these relationships would be:

```
A\<=\27
(PINA\==\PINB) | (temp\>=\21)
(BUSA+3)\!=\BUSB^'F7'
```

The test will always evaluate to either true (1) or false (0) and program flow will proceed accordingly.

The second type of conditional statement is the SWITCH/CASE statement. This provides a much more efficient way of dealing with situations where there are multiple condition branches. The syntax for this statement is as follows:

```
SWITCH (expr) BEGIN
   CASE a:        statements;
                  break;
   CASE b:        statements;
                  break;
   CASE c:        statements;
                  break;
   . . .
   DEFAULT:       statements;
                  break;
   END;
```

Execution begins with the evaluation of the (expr) expression. The compiler then scans down the list of CASE's to try and find a match to this value. If a match is found, the statements following it are executed until a "break' is reached, at which time the SWITCH block is exited. If no match is found then the optional DEFAULT case is taken and its "statements" are executed. If a logical "OR" of several CASE's is needed, then simply list these consecutively with no "break" between them as shown in the example below:

```
   . . .
CASE 'A':
CASE 'B':
CASE 12:       call zip();
               m = lo; vgen;
               break:
CASE v2:       call zop();
   . . .
```

Note that the CASE value can be either a constant or an expression that is dynamically evaluated.

12.0 COMMENTS

VGEN provides two facilities for commenting. The first is the ability to place comments in the VGEN program itself. This is accomplished by enclosing comments in curly brackets { }. These program comments can be placed anywhere in the program and are ignored by the compiler. A commented section of a VGEN program might look as follows:

```
   {NOW TEST THE ALU}
DB = PAT1;
CTRL = OP1;
VGEN;            {DO FIRST OPERATION}
DB = PAT2;
VGEN;            {NOW TRY NEW DATA}
. . .
. . .
```

Note that the curly bracket comment delimeters can be nested. This provides a quick way for removing sections of source program during development and debugging.

A second capability that VGEN provides for commenting is the ability to place text or comments directly in the simulation stimulus pattern file. This is accomplished with the COMMENT statement and has the following form:

```
COMMENT  "  THE ALU TESTS START HERE  " ;
```

When the VGEN compiler encounters a COMMENT statement, it places the text contained between the "quotes" directly in the stimulus pattern file. The \ (backslash) character can be used as an escape character for printing special characters such as the " (quote) character. Each COMMENT goes on a new line in the pattern file. The COMMENT statement also supports the printing of integer and string variable values as part of the text string between the "quotes". One way of accomplishing this is by using the special $ delimiter as shown below:

```
$TIME$  or  $INT_VAR$  or  $STRING_VAR$ or $DATE$
or  $expression$
```

TIME is the current VGEN "time" variable, INT_VAR is the name of any integer variable that has previously been defined, STRING_VAR is the name of any string variable that has previously been defined, DATE will cause the current Date/Time to be printed (UNIX platforms only) and expression is any legal VGEN expression. An example of their usage would be:

```
int cntr;
```

```
string pat1;
  . . .
  . . .
comment  "{ test = $cntr$; pattern = $pat1$ }";
comment  "{ next_test = $cntr+1$; pattern = $pat1$ }";
```

While VGEN is compiling, it will insert the above string into the vector file when it is evaluated, with each of the variables being replaced with its current value. For example, when the statement is encountered with cntr=65 and pat1='FA', the following string will be placed in the vector file:

```
{ test = 65; pattern = '11111010'B }
{ next_test = 66; pattern = '11111010'B }
```

A second method of printing pin or variable values within a comment string is to specify the desired format with a % designator within the character string, and then following the quoted string with the names of the variables to be printed. The legal format designators are:

%[-][n]d	decimal, - means left-justify, n is number of digits in field
%x or %h	Hex
%o	Octal
%b	Binary

An example would be:

```
COMMENT  "{ test = %5d; pattern = %x }", cntr, pat1;
```

Which would result in:

```
{ test =    66; pattern = FA }
```

Adding COMMENTs to vector files can be very helpful in debugging and documenting the vectors. However, some simulators do not support comments in their stimulus pattern files. To help get around this, a user may wish to place COMMENT statements in his VGEN program, compile it with the COMMENTs for his own documentation. He can then add the

```
COMMENTS OFF;
```

statement at the beginning of his VGEN program and recompile to get the same set of vectors, without comments, to be used by the simulator.

13.0 OUTPUT FORMAT CONTROL

In generating the actual stimulus vectors for a specific logic simulator as defined by the SIMULATOR command, VGEN offers several facilities for controlling the format of the vectors. The degree to which these are implemented varies depending on the target simulator and the features it supports. Refer to the VGEN Stimulus Interfaces Guide for more details on specific interfaces. In general, for stimulus vectors which are generated in a tabular format, it is often helpful to have the pins listed vertically above their respective columns of states. This can be done using the HEADER statement. If HEADER is used with a parameter, such as:

```
HEADER 60;
```

then this list of pin names is repeated every 60 lines.

For most simulator stimulus files, VGEN automatically generates some header information at the top of the file which typically defines pin names and bus radixes. The generation of this information can be inhibited with the DEFINE_HEADER command and replaced with user-defined text.

Another formatting control feature that VGEN offers is the ability to format bus vectors in either Hex or Octal radix formats. This is done with the BUSFORMAT statement as shown below:

```
BUSFORMAT HEX;
   or
BUSFORMAT ADR=HEX, CTRL=OCTAL;
```

In the first form, all busses are formatted to Hex. In the second, specific radices are specified for each bus.

In VGEN programs, all text is normally not case sensitive. Everything is converted to upper-case when the program source file is read in. There are some simulators, however, that have states which are distinguished by lower-case letters. In order to accommodate this, the KEEP_LOWERCASE statement can be used. It should be placed at the top of your program and results in all immediate string values in the program maintaining their lower (or upper) case form. Thus the two strings

```
'ZZZZ' and 'zzzz'
```

are identical if no KEEP_LOWERCASE statement is used, but are not identical if the statement is used.

Some simulators (such as Mentor) allow you to specify non-integer time in the vector stimulus files. Since all time in VGEN is integer numbers, a UNITS statement is provided to modify the time in the vector file. This has the form:

UNITS .1; or UNITS .01; or UNITS .001;

and results in the time being multiplied by the indicated decimal fraction.

Another VGEN command which provides format control for vector files is the STATE_TRANS command. This allows one to map the 1 and 0 states of input and/or output pins to other characters (such as H and L) when required by the simulator. When defining pin states within a VGEN source file, it is often desirable to compute them using a combination of arithmetic and logical operations. The STATE_TRANS command allows you to generate these states numerically and then have the 1 and 0 states mapped automatically by the compiler. An example of the syntax is :

STATE_TRANS outputs '1'->'H', '0'->'L' ;

Note that the STATE_TRANS command is only applicable for simulators which use characters other than 1 and 0 for states.

13.1 USER DEFINED FORMATS

In addition to the canned output formatters which produce output simulation data files for specific target logic simulators, VGEN also supports a User-Defined output vector format. This format must be of the tabular format type; i.e., a time stamp and complete set of pin state data comprises each line. This option is invoked with the SIMULATOR command as follows:

SIMULATOR UDF "format-string" [,bit_sep='c'] [,include_bi_outs] ;

Where the "format-string" specifies the actual format of each vector line, bit_sep='c' can be optionally used to change the character used for separating bits in the vector (this defaults to a blank, and is invoked by using double commas -,,- between pin names where you wish to have a separator), and the optional include_bi_outs forces the bidirectional output pin states to be listed as separate data columns.

Within the "format-string", there can be two types of information. These are variable data and hard-character data. Hard-character data is simply copied exactly as listed to the output vector. Variable data is of two sub-types; the first is the time variable (which always begins with the % character), and the second is the pin states variable ($states). The following are the variable data forms supported by UDF:

- %[n]at -- n digits of absolute integer time stamp. If no n is specified then field length will be exactly as wide as needed for current time stamp. If n is specified, time stamp will be right-justified in field; if -n is specified, time stamp will be left-justified in field.

- %[n]af -- n digits of floating-point absolute time stamp. Same rules as above apply to n. In addition, n can be used to specify the number of decimal points after the '.' to be used in the time stamp in a manner identical to the C Language. For example %-10.2af would specify a left-justified 10 character field with 2 digits after the decimal point. This time format would only normally be used if the UNITS command had been used in the VGEN program to

specify decimal fraction time units.

- %[n]dt -- Same as %[n]at except that a delta time stamp is used instead of an absolute time stamp.

- %[n]df -- Same as %[n]af except that a delta time stamp is used instead of an absolute time stamp.

- $states -- All pinstates. Includes bidirect outs only if the include_bi_outs option is specified. The formats of busses is as specified with the BUSFORMAT command. The order of pins is as defined by the INPUTS, OUTPUTS and BIDIRECTS commands. Also, the use of double commas between pins in these commands will result in a separator being placed between their states in the vector line.

If a header of some kind is needed at the top of the output vector file, the DEFINE_HEADER command can be used. Some examples follow for the pins defined below:

```
INPUTS a, b,, bus[7:0];
OUTPUTS ,,c, d,, outb[3:0];
```

EXAMPLE1 -

SIMULATOR udf " %8at - $states;";

SAMPLE OF VECTORS PRODUCED:

```
4355 - 01 11110000 10 0000;
4398 - 10 11110011 10 0001;
                    . . . etc.
```

EXAMPLE2 -

BUSFORMAT hex;
SIMULATOR udf "Time=%-8at States=$states;", bit_sep=',';
SAMPLE OF VECTORS PRODUCED -

```
Time=4355     States=01,F0,10,0;
Time=4389     States=10,F3,10,A;
     . . . etc.
```

14.0 EXTERNAL FILES

The VGEN compiler normally operates on two files during the compilation process. The first of these is the source file which contains the VGEN program statements. The second file is the stimulus vector file which the compiler produces during compilation. These two files are both considered primary files, and represent the minimum file set accessed during compilation. There are, however, a number of other "external" files which VGEN can be directed to either read or create. These files, and their application, are summarized below.

14.1 INCLUDE FILES

The VGEN compiler supports the inclusion of external files containing VGEN source program statements during compilation with the INCLUDE command. This command has the syntax:

```
INCLUDE  "filename" ;
```

The INCLUDE command allows users to partition their programs across several files. For example, one might collect a set of subroutines which are common to several stimulus programs into a single file and then simply INCLUDE this file in each of the source files. When the VGEN compiler loads a source file, all INCLUDE files are read directly into the source file at the location where the INCLUDE command is encountered. The compiler does not support nested INCLUDE files.

14.2 MERGE FILES

VGEN supports the merging of previously generated vector data with the vectors generated during compilation of a source file. This is accomplished with the MERGE_FILES command, which has the following syntax:

```
MERGE_FILE  "filename" ;
```

Only one merge file can be specified per VGEN source file compilation. The vector data in the external file must be in IVF format (See Appendix C). There are several possible ways in which this external file might have been created, depending upon the particular application. First, it could have been produced by a previous VGEN compilation where the SIMULATOR directive was defaulted to IVF. In this application, the external file might contain a separate set of pins with cycle time and pin timing different than that in the source file. This would be roughly equivalent to using two time lines (forks) in a single source file. In a different application, the external file may have been generated by VTRAN; again with the SIMULATOR command defaulted to IVF. Here VGEN and VTRAN play powerful supporting roles to each other by enabling the user to merge a newly generated set of vector data with a set of translated vector data.

14.3 EXPECTED OUTPUT FILE

While some logic simulators support the inclusion of expected output state data in their stimulus vector files, and give the user warnings when the expected output states differ from the simulation-generated output states, most simulators provide no such facility. In order to support the verification of expected output state data against simulation results data, the VGEN command EXPECT_FILE is provided for use with Source III's VCAP (Vector Comparison and Analysis Program). The syntax of this command is:

```
EXPECT_FILE  "filename"  [-ascii] ;
```

When used in a VGEN source program, this command causes the state data of all pins defined as OUTPUTS, as well as the output versions of BIDIRECTS pins, to be placed in the external file specified, rather than in the primary vector file. Data placed in this external file is in IVF format and can easily be read by VCAP for comparison against simulation results. The -ascii option forces the expect_file data to be in an ASCII tabular format, instead of the default compacted format.

14.4 READING EXTERNAL DATA FILES

External Data files can be opened and read from a VGEN source file. This capability allows VGEN to incorporate simulation data from outside sources into a new set of data. This data might be from a truth table, from another simulation run or from several such sources. There are four VGEN commands which support the reading of data from external files. These are:

```
FOPEN (fptr, "fname");

FREAD (fptr, "format", variable_list);

FCLOSE (fptr);

WHITESPACE . . ;
```

In order to read data from an external file, it is first necessary to open the file. This is accomplished with the FOPEN command. The first parameter associated with this command, "fptr", is an integer that points to the open file. This integer must be unique for each file opened and must not be declared as a general purpose integer variable with an INT statement elsewhere in the program. The second parameter is the name of the file to be opened. Two examples are:

```
fopen(data1, "sim_data.out") ;

fopen(ptr2, "/usr/joe/dmacircuit/qsim.list") ;
```

After opening an external file, data can be read using the FREAD command. The first parameter in the parentheses is the pointer to the file to be read from (fptr). This allows us to distinguish between several files which might be open simultaneously. The second parameter is a quoted

format string which specifies the format of the data to be read. After the quoted format string is an optional list of variables to which the formatted data items are to be assigned. Some examples are:

```
fread (fptr1, " %d  %32B\n", stime, tstring) ;

fread (dset4, "\5S%4H%6B%4B%B\n", dbus, adr, cntrl, r/w) ;
```

During a read, VGEN tries to match each item in the quoted format string with successive data in the file being read. During this process, information (characters) from the file can fall into one of three categories. These are:

1. Whitespaces: these are composed initially of blanks, tabs and newlines, and are ignored during the read operation. This list of whitespace characters can be added to with the WHITESPACE command. It has the syntax:

```
WHITESPACE '.', ':', ',' ;
```

where any number of individual characters can be added to the list, and are then ignored (treated like blanks) in the data file being read.

2. Variable data strings: these are identified in the quoted format string with a proceeding % character. When reading data strings, whitespaces between string data characters are ignored. The following are legal formats for reading data:

%d -- This reads an integer number from the data file

beginning at the current position in the vector.

%nB -- This reads n BINARY characters from the data

file. If n is omitted, one character is read.

Whitespace characters are ignored during read.

%nO -- This reads n OCTAL characters from the data

file. If n is omitted, one character is read.

Whitespace characters are ignored during read.

%nH -- This reads n HEX characters from the data file.

If n is omitted, one character is read.

Whitespace characters are ignored during read.

As each data string is read, it is interpreted in the format specified and assigned to the corresponding variable in the parameter list. For each % data string in the format, there must be a corresponding variable name in the parameter list.

3. "Hard" characters: characters in the format string which are expected to be found in the data string at an exact location. The matching of "hard" characters in the format string with characters in the vector line is case sensitive.

Within quoted format strings, there are several special operators. These are:

\nN -- Skip n lines. \n skips to the next vector line.
\nS -- Skip forward n character positions in the vector line.

Also, when a space is found in the quoted format string, it will skip ahead in the vector line until it finds a character which is not a whitespace. Thus a single space in the quoted format string will match zero or more whitespaces in the vector line.

Variables to which data is assigned as it is read from the external data file can be either INT, STRING, pins (as defined with INPUTS, OUTPUTS or BIDIRECTS commands) or GROUP.

During the execution of an FREAD command, VGEN attempts to match the quoted format string entries with data being read from the external file. If a "hard character" from the quoted string does not match the character being read from the external file, or if the end of the file is found during a read operation, then a global variable, $FILE_STAT, is set. This variable can thus be used to control the FREAD process as illustrated below:

```
FOPEN (fdata, "other.vecs");
$FILE_STAT = 0;
fread (fdata, "TIME= %d ; DATA= %48B;\n", newtime, groupall);
WHILE $FILE_STAT\==\0 begin
   call new_vecs() ;      { add new vectors }
   if newtime\>\$time then $time = newtime;
   vgen;
   fread (fdata, "TIME= %d ; DATA= %48B;\n", newtime, groupall);
   end;
```

The following are several examples of vector lines in an external file, followed by the FREAD command to read it.

EXAMPLE1

VECTOR LINES:

```
11 22 33 44
55 66 77 88
43 42 41 40
```

QUOTED FORMAT STRINGS:

```
fread(fptr1, "%2H %2H %2H %2H\n", busa, busb, busc, busd);

fread(fptr1, "%8H\n", group1);
```

In the first quoted string, the data is in HEX format and is being loaded directly into some 8-bit (or larger) busses. In the second quoted string, a group (group1) is being used to retain the entire vector line. Since blanks are whitespaces, they are automatically ignored during the read.

EXAMPLE2

VECTOR LINES:

```
1430 > 1,1,0,0,2F,DD3E;
1477 > 1,1,0,1,2E,DD00;
```

QUOTED FORMAT STRINGS:

```
fread(fptr1, " %d > %B,%B,%B,%B,%2H,%4H ; \n", newtime,p1,p2,p3,p4,b1,b2);

fread(fptr1, " %d  %4B%6H\n", newtime, str1, str2);
```

In the second quoted string, it would be necessary to have defined the > and , characters as whitespaces prior to any reads:

```
WHITESPACE  '>', ',' ;
```

15.0 OUTPUT AND BIDIRECTIONAL PIN DATA

For most logic simulators, the data in vector files read for simulation can contain only stimulus information. These simulators contain no run-time facilities for verifying the actual simulated state on an output pin against an expected state provided in the vector file applied to it. For these simulators, all pins are typically defined in the VGEN source file as INPUTS and pins which are actually bidirectional are assigned 'Z' or high-impedance states when they are supposed to be outputting data. In order to provide a facility to perform verification of simulation output data against expected state data for these simulators, VGEN provides a link (the EXPECT_FILE command) to VCAP, the Vector Comparison and Analysis Program.

To verify simulation results against expected states with these simulators, the OUTPUTS and BIDIRECTS commands should be used where appropriate in VGEN in addition to the INPUTS command. Within the VGEN program, the output pins and output versions of bidirectional pins (.O suffixed) have state assignments made where appropriate for verifying simulation output states at desired times. The EXPECT_FILE command is then used to place this output data into a separate file for later processing by VCAP against the simulation results. Remember that when an expected output state is assigned to a pin, this state will remain in effect until it is changed. An output pin should be specifically assigned an 'X' state (or whatever the don't care state is) during cycles where its state is either unknown or don't care.

When assigning expected output states to output pins, it is often desirable to define a specific timing window, or point in time within the cycle where the check against the simulation results data should be made. This is most easily accomplished in this case by using the PINTYPE RX 'X' command. It has the syntax:

```
PINTYPE  RX  'X'  pinlist  @  t1, t2 ;
```

Here, when we assign an expected output state to an output (or .O version of a bidirectional pin), we are defining a specific time window within the cycle (t1 to t2) during which the expected data should be checked. During the rest of the cycle we define the state to be 'X' (don't care).

There are also a number of logic simulators which do provide for verification of expected output data against simulation results data. Here, we can define pins as OUTPUTS or BIDIRECTS, as well as INPUTS, and include the expected output data right in the same vector file as the stimulus data (we don't use the EXPECT_FILE command). For simulators of this type, there are several situations which arise. These situations have to do primarily with the state characters used for input vs. output state data and whether bidirectional data is merged on a single pin column (tabular formats) or has separate columns for input and output data of the same pin.

In order to deal with simulators which use state characters other than 1 and 0 for output and/or input pin data, VGEN has the STATE_TRANS command. This command allows the user to generate stimulus and expected response data using the 1 and 0 state characters (which can be manipulated consistently with arithmetic and logical operators), and then have the appropriate

states mapped as a post-process during formatting for the target simulator. For example, the following statement will cause all output pins and the output versions of bidirectional pins (.O suffixed) to have their 1 and 0 states to be mapped to H and L respectively in the vector file:

```
STATE_TRANS outputs  '1'->'H' , '0'->'L' ;
```

If the simulator reserves separate pin columns in the vector file for input and output data on bidirectional pins, then VGEN simply maintains the separation of this data as it is defined in the VGEN source program. For some simulators, however, bidirectional data needs to be re-combined onto a single pin column with input and output data being distinguished by state characters. For example, a 1 and 0 might be used when driving the pin (input) while an H and L are used to indicate the pin is outputting data. For this case, a STATE_TRANS command such as the example above could be used to do the state mapping. However, we also need to be able to re-combine the pin and pin.O data for bidirectional pins back onto a single pin. To accomplish this, the MERGE_BIDIRECTS command can be used. The example below illustrates this:

```
BIDIRECTS bus[7:0];
PINTYPE  nrz  bus  @ 20;
PINTYPE  rx 'X' bus.O  @ 90, 95 ;
PINTIMING ON bus, bus.O;
  . . .
bus = '22';          { drive input with '22'  }
bus.O = 'XX';
vgen;
bus = 'ZZ';
bus.O = '55';        { expecting '55' output on bus  }
vgen;
  . . .
```

State assignments proceed separately to the bus and bus.O pins. Now, in order to re-combine the input and output data onto a single pin, with H and L used to signify output data, the following commands are used (they can be placed anywhere in program):

```
STATE_TRANS outputs '1'->'H', '0'->'L' ;
MERGE_BIDIRECTS  1 0 H L Z X ;
```

This will cause the bus and bus.O pin data to be combined onto the bus pins using the state precedence defined by the MERGE_BIDIRECT command. When doing this type of processing, VGEN performs the STATE_TRANS function prior to the MERGE_BIDIRECTS function.

Another group of simulation interfaces referred to as Testbench interfaces perform expected results data checking against simulation results data without need of either of these commands. The current Testbench interfaces available are for the VHDL and Verilog simulators. See the Stimulus Interfaces Guide for the latest information on using these interfaces, and for specific information on any of the simulator interfaces.

16.0 INVOKING AND USING THE VGEN COMPILER

Once a VGEN program has been created, typically using an editor of some sort, the VGEN compiler can be invoked to compile the program and generate the stimulus pattern file. This is accomplished by typing the following command:

```
VGEN  INFILE  [OUTFILE]
```

Here INFILE is the name of your VGEN program file and OUTFILE is the name of the file in which you wish to have the stimulus patterns placed. The OUTFILE name is optional and, if omitted, the stimulus patterns are placed in a file whose name is the same as the INFILE name except that an extension of .vec is added. Thus, invoking VGEN with:

```
VGEN TEST
```

will result in the stimulus patterns being placed in a file named TEST.VEC.

16.1 DEBUGGING VGEN PROGRAMS

As with any program, particularly as their complexity increases, VGEN programs must be "debugged" so that the stimulus vectors that it generates are exactly what you intended and there are no syntax errors in the program.

When the VGEN compiler encounters a syntax or other functional error in a program, it will generally issue an error message to the screen and terminate. This message gives a description of the error and the line number at which it was encountered. Line numbers are of the form

```
file : line
```

where "file" is the source file name and "line" is the line number within the file. If INCLUDE files are used, the "file" name will reflect this.

There is also a facility within VGEN to trace the execution of VGEN programs. This is very helpful for debugging programs and can be invoked as follows:

```
VGEN   INFILE   TRACE!
```

Here again, INFILE is the name of the file containing the VGEN program. The trace data is presented on the screen and the stimulus patterns are still placed in the file INFILE.VEC.

The TRACE facility can also be controlled from within a VGEN program using the TRACE statement. It has the form:

```
TRACE ON;
  or
TRACE OFF;
```

Thus, one can use this to trace specific segments of a program by placing the TRACE ON command before it and the TRACE OFF command after it.

A final mechanism is also available to both aid in program debugging and to provide compilation progress feedback to the user. This is the ECHO statement which has the form:

```
ECHO "text";
```

The ECHO statement, when encountered by the compiler, causes the quoted text to be sent to the CRT screen. This text string may contain integer variables, string variables or other special variables (this is also true for the COMMENT statement string) which are surrounded by $ characters. An example of this usage would be:

```
int cntr, loopvar;
   . . .
   . . .
echo "Test $CNTR$; Loop $LOOPVAR$; time=$TIME$";
```

While VGEN is compiling, it will send the above string to the screen (which can be re-directed to a file) with each of the variables replaced with its current value. For example, when the statement is encountered in a loop with cntr=99, loopvar=14 and system time at 234000, the following string will be issued:

```
Test 99; Loop 14; time=234000
```

A second method of printing pin or variable values within an ECHO string is to specify the desired format with a % designator within the character string, and then following the quoted string with the names of the variables to be printed. The legal format designators are:

%[-][n]d	decimal, - means left-justify, n is number of digits in field
%x or %h	Hex
%o	Octal
%b	Binary

An example would be:

```
ECHO  "{ test = %5d; pattern = %x }", cntr, pat1;
```

Which would result in the following screen line:

```
{ test =    66; pattern = FA }
```

The ability to generate strings, to the screen or to a vector file, which contain variable values is quite useful for debugging and documenting VGEN programs. Using the ECHO statement and re-directing the screen output to another file, for example, allows you to create a time map of the test sections of your vector file.

APPENDIX A

VGEN STATEMENT SYNTAX

APPENDIX A

VGEN STATEMENT SYNTAX

DECLARATIVES

ANALOG

USAGE: ANALOG *busname* [,VMAX=value] [,VMIN=value];

WHERE: *busname* is the name of a previously defined input bus having a width greater than 1 bit.

EXAMPLES: ANALOG abus ;

ANALOG dbus vmax = 12.5, vmin = .5 ;

DESCRIPTION: The ANALOG command is used only with the SPICE analog simulator interface. Here it is used to define a bus input (say dbus[11:0]) as a single analog pin where the bits of resolution of the analog pin is equal to the width of the bus. The analog value of the pin can then range between VMAX and VMIN, with all 1's equaling VMAX (dbus = 'FFF' means dbus = 12.5 volts in the above example), and all 0's equaling VMIN. In the example above, dbus = '7ff' would set the analog pin to a value of 6.5 volts.

ASSERT

USAGE: ASSERT (list of *pinnames* or *groupnames*) @ [+,-] *xx*;

WHERE: *pinname* is a pin name declared in INPUTS declarative; groupname is a group declared in GROUP declarative and *xx* is a integer number, variable or expression (0 default)

EXAMPLES: ASSERT CNTRL[3..0] @ 20;

ASSERT DB[15:0], BIP, TOP @ -10;

ASSERT MS, ADR[12..0] @ -t;

DESCRIPTION: ASSERT defines the time, relative to the start of a time step, that a group of signals is to be asserted. Normally all signals, which are going to change state in a new pattern, do so at the beginning of the cycle.

BIDIRECTS

USAGE:	BIDIRECTS *pinlist* ;
WHERE:	*pinlist* is a list of legal pin names
EXAMPLES:	BIDIRECTS pina pinb pinc ;
	BIDIRECTS bus[19:7], flag ;
DESCRIPTION:	The BIDIRECTS declarative is used to define pins which are bidirectional. This should be used only with simulators which support the definition of separate pin state information for bidirectional pins in the stimulus file, or when the EXPECT_FILE command is being used to keep expected output state data. The compiler will create two pins for each bidirect pin, one for the input version having the name listed and a second one for the output version having a .o suffix. This allows users to assign expected output states to the latter. The compiler then will drop the .o suffix in the stimulus file.

BUSFORMAT

USAGE:	BUSFORMAT *radix* ; or
	BUSFORMAT busa = *radix*, . . . , busn = *radix* ;
WHERE:	*radix* is HEX, OCT or BIN (BIN is the default); busa thru busn are bus names which have previously been defined with INPUTS, OUTPUTS or BIDIRECTS declaratives.
EXAMPLES:	BUSFORMAT HEX;
	BUSFORMAT addr=hex, ctrl=oct;
DESCRIPTION:	Allows the user to specify the radix to be used in the stimulus file for each bus in his circuit.

CYCLE

USAGE:	CYCLE [=] *xx* [, *all*] ;
WHERE:	*xx* is a positive integer, variable or expression (100 default). The optional *all* parameter forces the compiler to output a vector every CYCLE period whether or not any of the pins have changed state.

EXAMPLES: CYCLE 100;

CYCLE = 175 , all ;

CYCLE next+12;

DESCRIPTION: CYCLE is used to define the basic time step for successive stimulus patterns. All signals assume their new pattern states at the start of each cycle unless their timing has been modified by an ASSERT, PINTYPE, WAVEFORM or PULSE declarative. Since most logic simulators include a time stamp with each new vector, it is not necessary (or sometimes not allowed) to include two or more successive vectors in the stimulus file which have no change in pin states. Thus, VGEN will suppress the output of a vector if there has been no change in any of the pin states. This suppression can, however, be overridden with the *all* parameter.

DEFAULT RADIX

USAGE: DEFAULT RADIX *radix*;

WHERE: *radix* is HEX, OCT or BIN (HEX default)

EXAMPLES: DEFAULT RADIX HEX;

DEFAULT RADIX BIN;

DESCRIPTION: DEFAULT RADIX defines the *radix* for pattern strings used in assignment statements or expressions. This applies only to strings that contain no *radix* specifier themselves.

DEFINE_HEADER

USAGE: DEFINE_HEADER " *text string* " ;

WHERE: *text string* is an ASCII character string which may span multiple lines.

EXAMPLES: DEFINE_HEADER " inputs = bow(7->0), p1, p2, p3; " ;

DEFINE_HEADER " display all;

include pinnames; " ;

DESCRIPTION: VGEN normally writes a header at the top of the vector file which includes a pin list formatted for the target simulator. It determines

what this should consist of through the INPUTS, OUTPUTS, BIDIRECTS and BUSFORMAT commands. The DEFINE_HEADER command allows you to inhibit the automatic generation of this header and instead replace it with a custom text string.

EXPECT_FILE

USAGE: EXPECT_FILE "*filename* " [-ascii] ;

WHERE: *filename* is a legal file name

EXAMPLES: EXPECT_FILE "exp_outs.vec" ;

DESCRIPTION: This command is provided primarily for use with the VCAP program. When used, the command results in all expected output state data specified for output pins and the output versions of bidirectional pins, to be placed in the specified file. This data is in IVF format and can be read later by VCAP for comparisons against simulation results data. The -ascii option forces the expect_file data to be in an ASCII format instead of the default compacted format.

GROUP

USAGE: GROUP *groupname* pina, pinb, ... pinx;

WHERE: *groupname* is a label; pina ... are pin names declared in an INPUTS, OUTPUTS or BIDIRECT declarative or other *groupnames* or the bit position-holder $NULL. Order is: MSB = pina, LSB = pinx.

EXAMPLES: GROUP CTRL IN[3:0] RLS QIN;

GROUP A DI[3..0] $NULL PLB REX1 REX0;

DESCRIPTION: GROUP is used to define a set of signals which are to be considered as a single array (bus) for the purpose of referencing. The special name $NULL can be used as a bit position holder for aligning specific pins to bit positions in the group array. The array has the name *groupname* so that in the examples above;

A[5] is the signal DI[2];

CTRL[2] is the signal IN[0];

The name A refers to the entire group of 7 signals, with the high-order 4 bits being DI[3:0] and the low-order 3 bits being PLB, REX1 and REX0. A[3] is a null (X) bit.

HEADER

USAGE: HEADER *n* ;

WHERE: *n* is a positive integer.

EXAMPLES: HEADER 55;

HEADER 110;

DESCRIPTION: The HEADER declarative can be used to cause a vertical list of pins to appear as comments in the vector file every *n* lines. The pin names are aligned vertically over the columns containing their states. This will only work for those target simulators which have tabular formats and which allow comments in their vector files.

INCLUDE

USAGE: INCLUDE *filename* ;

WHERE: *filename* is the file to be inserted at this point.

EXAMPLES: INCLUDE subs.lib;

INCLUDE section1.dat;

DESCRIPTION: INCLUDE provides a mechanism for inserting the contents of other files into the VGEN source program. This allows users to create modular programs and optimally partition them for utility and documentation.

INITIALIZE_PINS

USAGE: INITIALIZE_PINS [inputs / outputs] 'C' ;

WHERE: 'C' can be any state character.

EXAMPLES: INITIALIZE_PINS inputs '0';

INITIALIZE_PINS outputs 'X';

DESCRIPTION: The states of input and/or output pins can be initialized at time 0

(actually at any time in the program) with this command. If used for initialization at time 0, the command should follow any PINTYPE or ASSERT commands used to set timing prior to the first VGEN.

INPUTS

USAGE: INPUTS pin1, pin2, ... pinn;

WHERE: pin1 ... are input pins to the circuit

EXAMPLES: INPUTS A, B, DB[12:0], Q2;

INPUTS R,4,5, S, BLUE;

DESCRIPTION: INPUTS declares all of the pins which are inputs to the circuit, this can include bidirectional pins. The order in which they are listed is the order in which they will appear in the stimulus pattern file.

INT

USAGE: INT (name or namelist) [*value*];

WHERE: (name or namelist) are labels; *value* is an optional integer.

EXAMPLES: INT A B C 0;
INT X Y Z ;

DESCRIPTION: The INT declarative is used to define integer variables and optionally assign an initial value to them.

KEEP_IVF

USAGE: KEEP_IVF;

DESCRIPTION: The KEEP_IVF declarative is used to prevent the compiler from erasing some of the intermediate files it creates during the compilation process. These files are named __VGENx.VEC and contain IVF-formatted data as well as data from multiple time lines if they are being used. These files may be helpful in the debugging process.

KEEP_LOWERCASE

USAGE:	KEEP_LOWERCASE;
DESCRIPTION:	The KEEP_LOWERCASE declarative is used to prevent quoted string values from being converted to upper case. This allows users to generate lower case character states in their pattern files - useful for some simulators.

MAKE_SF

USAGE:	MAKE_SF " *filename* " ;
WHERE:	*filename* is any legal file name.
EXAMPLE:	MAKE_SF "ckt1.side" ;
DESCRIPTION:	When using VCAP for performing simulation data results verification, this command produces information in a side file that enables VCAP to relate mismatches between expected data and simulation data back to specific line numbers in the VGEN source file used to generate the simulation stimulus that produced the mismatch.

MERGE_BIDIRECTS

USAGE:	MERGE_BIDIRECTS *precedence-list* ;
WHERE:	*precedence-list* is a list of state characters in order of their precedence when combining the input and output (.O) data of bidirectional pins onto the input version of the pin.
EXAMPLES:	MERGE_BIDIRECTS 1 0 H L Z X ; merge_bidirects 1 0 "h" "l" "x" "z" ;
DESCRIPTION:	This command can be used to re-combine bidirectional pin data, where the generation of the data used pin and pin.O state assignments separately. Normally this merging would also involve state translations of some kind for output data states. The precedence list determines how the pin and pin.O data is to be combined onto the pin.

MERGE_FILE

USAGE: MERGE_FILE " *filename* " ;

WHERE: *filename* is a legal file name.

EXAMPLE: MERGE_FILE "set1.vec" ;

DISCUSSION: This command defines an external file, which must be in IVF format, that contains vector data to be merged with the vector data being generated by the current VGEN compilation.

OUTPUTS

USAGE: OUTPUTS pin1, pin2, . . . , pinn ;

WHERE: pin1 . . . pinn are output pins to the circuit.

EXAMPLES: OUTPUTS a, b, c, d;
OUTPUTS bus<12 . . 8>, flag, stop ;

DESCRIPTION: OUTPUTS declares all of the pins which are outputs from the circuit. This declarative should only be used with simulators which allow you to specify expected output states in the stimulus file, or if the EXPECT_FILE command is being used to direct expected output state data to an external file for later access by VCAP.

PINTYPE

USAGE: PINTYPE *pintype pinlist* @ *start end* ,

. . .

pintype pinlist @ *start end* ;

WHERE: *pintype* is one of the following:

NRZ (DT) - non-return-to-zero
STB - strobe time for outputs
RZ (PP) - return-to-zero
RZ2X - return-to-zero, double pules
RO (NP) - return-to-one
RO2X - return-to-one, double pulse
RC - return-to-compliment
SBC - surround-by-compliment
RX 'x' - return-to-x, where x is user-specified

BIDR - bidirectional timing

pinlist is a pin name, group name or list of these.

start is the time (relative to the start of each cycle) when the pin will assume its current state. This may be an integer value, a variable or an expression. For RO2X and RZ2X, the second pulse start time will be (cycle/2)+*start* . For BIDR pintypes, this is the time offset for applying the new input state.

end is the time (relative to the start of each cycle) when the pin will go to its return-to value. This may be an integer, variable or expression. This parameter is omitted for NRZ pintypes since it does not return-to an inactive value as in the case of the other pintypes. For RO2X and RZ2X, the second pulse end time will be (cycle/2)+*end* . For BIDR pintypes, this is the strobe (application) time for expected output states.

EXAMPLES:

```
PINTYPE NRZ BUSA @ -40;
PINTYPE RC PINA,PINB,BUSB @ -20 HOLDTIME;
PINTYPE RZ ADDR<7 . . 4>  @ -10  100,
           RC   ADDR<3 . . 0> @ -30  80 ;
PINTYPE  RC  CLK  @ 0  ($CYCLE/2) ;
PINTYPE   RX   'U'  dbus  @  45,  110 ;
```

DESCRIPTION:

PINTYPE defines the behavior and timing to be used for input and/or output pins. When using VGEN, the timing of pins can be modified to take on their newly assigned states at the *start* time (for NRZ and BIDR) or during the timing "window" between *start* and *end*. Once pins have been assigned a pintype and timing with this declarative, the properties can be enabled and disabled with the PINTIMING statement. After PINTYPE definition, PINTIMING is initially disabled. If expressions are used to define any of the timing parameters, they must be enclosed in parenthesis as in the example above. For SBC pintypes, the pin will be assigned the compliment of its new state at time 0 (start of the cycle), the new state at *start* time and return-to compliment state at *end* time. For RX pintypes, the character immediately following the RX designator is the state character used as the return-to state. This pintype is useful for specifying the behavior of bidirectional pins, where the return-to state is Z, or for specifying expected output states within strobe windows where the return-to state is X (don't care) for the rest of the cycle. The return-to state of '*' has a special meaning. It causes the pin to return to its previous (before *start*) state after the *end* time. BIDR pintype should only be used for pins defined as BIDIRECT

PULSE

USAGE: PULSE pinname *start width polarity*;

WHERE: pinname is a single pin name specified previously in an INPUTS declarative.

start is the time (relative to the start of each cycle) when the pin pulse is asserted. This may be an integer, variable or expression.

width is the time width of the asserted pulse. This may be an integer, variable or expression.

polarity is either 1 or 0 (1 default) and defines the assertion state for the pulse.

EXAMPLES:

```
PULSE  CLK  -20  50  1;
PULSE  WE*  30  30  0;
PULSE   A  0  ($CYCLE/2)  1 ;
PULSE  CE*  -a  (b+12)  0;
```

DESCRIPTION: PULSE, when used in this form, defines the timing and polarity for applying a pulsed signal to a pin. A second PULSE statement format is used to enable and disable the actual pulsing on the pin, once it has been defined. The second PULSE format is described in the Pattern Generation Statements section. Up to 7 pulse pins may be defined in a program. If expressions are used to define any of the timing parameters, they must be enclosed in parenthesis as in the example above.

SCALE

USAGE: SCALE [=] *xx*;

WHERE: *xx* is a real number (1.0 default); = is optional.

EXAMPLES:

```
SCALE 1.5;
SCALE 0.4;
```

DESCRIPTION: SCALE globally modifies the timing of all signals. The value specified is a multiplier which changes values of CYCLE, ASSERT , PINTYPE, WAVEFORM and PULSE declaratives.

SIMULATOR

USAGE: SIMULATOR *simulatorname* ;

WHERE: *simulatorname* is one of the logic simulators supported with an interface. See the VGEN Stimulus Interfaces Guide for details on supported interfaces.

EXAMPLES: SIMULATOR ADVANSIM_1076 ;
SIMULATOR MENTOR_FORCE ;

DESCRIPTION: This declarative defines the target logic simulator to which the stimulus patterns are to be applied. If no SIMULATOR declarative is specified, then the stimulus file will use the VGEN generic IVF format.

STATE_TRANS

USAGE: STATE_TRANS [inputs/outputs] 'x'->'y', . . . 'w'->'z' ;

WHERE: x and w are states (usually 1 and 0) being assigned to pins in the program file, and y and z are the states to which these should be mapped in the vector file.

EXAMPLES: STATE_TRANS outputs '1'->'H', '0'->'L' ;

STATE_TRANS inputs '1'->'P', '0'->'N' ;

DESCRIPTION: This command defines a set of state mappings between input and/or output pin states used in the source program and a set of desired state characters for pins in the vector file. This is useful for several simulators which require the state characters to be different than 1 or 0 for input pins, output pins or both.

STRING

USAGE: STRING (name or name list) [*pattern*];

WHERE: (name or name list) is a string variable name or list of names with optional length subscripts.

[*pattern*] is an optional legal vgen string or an integer. If no pattern is specified, then the stringname(s) is assigned an initial value of 'X'.

EXAMPLES: STRING MASK 5 ;
STRING BLUE '111001'B ;
STRING P1[12] P2[8] P3[4] ;

DESCRIPTION: STRING is used to define a string variable(s) and optionally assign an initial value to it. Strings defined with this command can also have a length specified (see last example above) which indicates the number of characters contained in this string variable. The maximum length of a string variable is 1024 characters. If no length is specified then it defaults to 64-bits long. The value of a string variable is always a string. Thus, if the value [*pattern*] in a STRING statement is an integer, it is first converted to a string before assignment.

SUBROUTINE

USAGE:

```
SUBROUTINE subname (n1, n2, ... nn)  BEGIN

     . . .

     . . .

END;
```

WHERE: *subname* is the name of the subroutine being defined. The statements between BEGIN and END comprise the body of the subroutine. Subroutines may call other subroutines.

n1, n2, ... nn are optional parameter values or pointers passed to the subroutine. They may be either strings or integer values, variables, expressions or pointers. A parameter can only appear on the left side of an assignment statement if it is a pointer.

EXAMPLE:

```
SUBROUTINE  CLKCYCLE(S, PINS)  BEGIN
        PINS = S; VGEN;
        CLK = HI; VGEN;
        CLK = LO; VGEN;
        END ;
```

DESCRIPTION: The SUBROUTINE declarative can be placed anywhere in the VGEN program. Calls to the statements that lie between the BEGIN and END keywords is accomplished (for the above example) with the

```
CALL  clkcycle( 4,  &dbus) ;
```

statement. The parameters passed to the subroutine are contained between ( ) and may be integers, strings, expressions or pointers separated by commas. Pointers are pre-fixed with the & (ampersand) character in the calling parameter list.

SYS_CLOCK

USAGE: SYS_CLOCK *name hi_time lo_time start_state* ;

WHERE: *name* is a signal name (not defined as a pin)

hi_time and *lo_time* are the values which define the amount of time the signal is hi and low, respectively.

start_state is the state in which the signal starts out at the beginning of its period.

EXAMPLES: SYS_CLOCK clk 55 45 1;

SYS_CLOCK ck12 125 225 0;

DESCRIPTION: This is a special declarative for defining the clock pin behavior in simulators which provide special facilities for clock definition. Check with the VGEN Stimulus Interfaces Guide to see which simulator interfaces support this feature. In the second example above, a clock pin named ck12 is defined which has a period of 350; starting out low for 225 and then going hi for 125.

SYSTEM_CALL

WHERE: Text means up to 128 characters.

EXAMPLES: SYSTEM_CALL ".. text .. ";

DESCRIPTION: Where the text between the quotes is passed to the system upon completion of the VGEN compilation.

TABLE

USAGE:

TABLE *tablename* BEGIN
    val1 val2 val3 ..

    .. valn

END;

WHERE: *tablename* is the name of the data table being defined. The values between BEGIN and END comprise the constant data contained in the table.

val1 val2 .. valn are the constant data values. These values must be either immediate integer or string data.

EXAMPLE:

```
TABLE   INITDATA   BEGIN
        55   22   45   'F7'
        88   12   33   'E3'
        'EEF4'  'DDCC'  '11010011'B
        END;
```

DESCRIPTION: The TABLE declarative can be used to define tables of constant data to be used in the VGEN program. Data contained in a table is referenced by the tablename followed by an index in parentheses. The first entry in a table is always index 0, so in the example above INITDATA(0) has the value 55. The index can also be a variable or expression such as INITDATA(X-1).

TITLE

USAGE: TITLE " text string " ;

EXAMPLE: TITLE " Circuit1 " ;

DESCRIPTION: Several logic simulators provide facilities within their vector files to place a title or other descriptive name. This command provides a way of defining a text string to be used for this purpose. See the VGEN Stimulus Interfaces Guide for details on which interfaces support this.

UNITS

USAGE: UNITS [=] *xx*;

WHERE: *xx* is either .1, .01 or .001 ; = is optional.

EXAMPLES: UNITS .1;

UNITS = .01;

DESCRIPTION: UNITS declarative affects the format of the time tag in a stimulus pattern file. Normally time is an integer number in most simulator vector formats. Some, such as Mentor and TIMEMILL, allow the use of decimal fractions on time tags. For these the time

tag will be multiplied by the UNITS value in the vector pattern file.

WAVEFORM

USAGE: WAVEFORM pinname *period width polar* [*start*];

WHERE: pinname is a single pin name specified previously in an INPUTS statement.

period is the time period or repetition time of the waveform. This may be an integer, variable or expression.

width is the time width of the asserted pulse. The pin will be asserted at the start of the period and remain asserted for this length of time. At the end of this time the pin will return to a state opposite its assertion state. This value may be an integer, variable or expression.

polar is either 1 or 0 and defines the assertion state for the waveform.

[*start*] is an optional absolute start time for enabling the waveform. If no start time is specified then the waveform is initially disabled and can be enabled or disabled with the WAVEFORM ON/OFF statement as described below.

EXAMPLES:

```
WAVEFORM  CLK  250  100  1;

WAVEFORM  CLK  20  5   1  2500;

WAVEFORM  INT  (MAXLP*4)  100  1;
```

DESCRIPTION: WAVEFORM, when used in this form, defines the timing of an asynchronous periodic waveform to be applied to a pin. If expressions are used to define any of the timing parameters, they must be enclosed in parenthesis as in the example above.

WAVES

USAGE: WAVES [*pinlist*] ;

WHERE: *pinlist* is an optional list of pin names to appear in the wave display. If no *pinlist* is present then all pins are included in the display.

EXAMPLES: WAVES ;

WAVES dbus, adrbus, mode, rw, clk ;

DESCRIPTION: The WAVES statement is used to tell VGEN that it should create a vector file for Waveform Tool during compilation, and then invoke it when the compilation is complete.

WHITESPACE

USAGE: WHITESPACE *char-list* ;

WHERE: *char-list* is a list of state characters enclosed individually in single or double quotes.

EXAMPLE: WHITESPACE '>' , ':' , '\n' ;

DESCRIPTION: When reading state information from external data files using the FREAD command, whitespaces are ignored (skipped). Characters included as whitespaces are initially blanks and tabs, but additional characters can be added to this set with this command. Characters listed in the WHITESPACE command are treated exactly like blanks during an FREAD. The special character '\n' adds newlines to the whitespace list, which essentially causes the read to scan multiple lines.

PATTERN GENERATION STATEMENTS

BEGIN

USAGE: BEGIN

. . .
END;

EXAMPLE: if a \==\ 4 then BEGIN

. . .
END ;

DESCRIPTION: The BEGIN statement initiates a statement block which typically contains multiple VGEN statements. It is normally used in conjunction with conditional statements, loop statements and subroutines.

BREAK

USAGE: BREAK;

EXAMPLE: IF A\==\7 THEN BREAK;

DESCRIPTION: The BREAK statement alters program flow inside of a program loop by forcing an immediate exit from the inner-most loop. If encountered in a subroutine, but outside of any program loops within the subroutine, it will cause an immediate return from the subroutine. BREAK is also used in the SWITCH/CASE statement to exit the SWITCH block.

CALL

USAGE: CALL *subname* (n1, n2, ... nn);

WHERE: *subname* is the name of a subroutine that has been declared with a SUBROUTINE declarative somewhere else in the program.

n1, n2 ... nn are parameters and may be integers, strings, expressions or pointers.

EXAMPLES: CALL CKCYC;

CALL BUS_CYCLE(ADR1, &J, (K-J), 5);

DESCRIPTION: The CALL statement transfers program execution flow to the named subroutine. After executing the statements in the subroutine, program flow returns to the statement immediately following this one. Subroutine parameters may be passed by value - i.e. variables are evaluated and their current values at the time of the CALL are passed to the subroutine, or they may be passed as pointers to a pin, bus, group or table. Pointers are passed by pre-fixing the name with the & (ampersand) character.

CASE

USAGE: CASE *value* : *statements* ;

WHERE: *value* is a constant, a variable or an expression. *statements* can be either null, a single statement or a sequence of statements.

EXAMPLES:

```
CASE  55 :     call  zip();

CASE   (x+4):          call  zap(4);

                        pin4 = hi;

                        break;
```

DESCRIPTION: The CASE statement may only be used within a SWITCH statement block. It is used to delineate the test values with which the SWITCH parameter will be compared, and upon a successful comparison the statements following the CASE statement are executed.

COMMENT

USAGE: COMMENT " *string* " ;

WHERE: *string* is an ASCII text string which may optionally contain variables delineated by $ symbols.

EXAMPLES:

```
COMMENT  " { The ALU tests begin here } " ;

COMMENT " /* Loop = $loopvar$  */ " ;

COMMENT  " /* Loop = %d  */ ", loopvar ;
```

DESCRIPTION: The COMMENT statement is used to place comments in the stimulus file being created by the VGEN program. Each comment will appear at the time in the vector file corresponding to when the

COMMENT statement is encountered in the program. Variable values can be placed in the text string by delimiting the variable name with $ characters as shown in the second example above. Variable values can also be printed using a % format designator and then following the string with the variable name, as shown in the last example - this produces a comment identical to the second example.

COMMENTS ON/OFF

USAGE: COMMENTS ON ; or COMMENTS OFF;

DESCRIPTION: This statement either enables (ON) or inhibits (OFF) the printing of comments in the stimulus file. The default for COMMENT printing is ON. If a COMMENTS OFF statement is encountered in the VGEN program, all subsequent COMMENT statements will be ignored.

CONTINUE

USAGE: CONTINUE;

EXAMPLES: IF A\!=\7 THEN CONTINUE;

DESCRIPTION: The CONTINUE statement alters program flow inside of a program loop by forcing an immediate jump to the top of the loop. Here the loop test or index is evaluated as usual and the loop either continues or is exited.

ECHO

USAGE: ECHO " *text* ";

WHERE: *text* is any ASCII character string including variable names delineated by $ characters.

EXAMPLE: ECHO "Beginning The Memory Tests";

ECHO " MEMORY pass = $pcnt$, at time $TIME$ " ;

ECHO " MEMORY pass = %d, at time %d ", pcnt, $time;

DESCRIPTION: The ECHO statement provides a mechanism for printing text on the CRT screen. It is useful for program debugging as well as providing compilation progress status to the user. Variable values can be placed in the text string by delineating the variable name

with $ characters as shown in the second example above. Variable values can also be printed using a % format designator and then following the string with the variable name, as shown in the last example - this prints a screen line identical to the second example.

END

USAGE: END;

DESCRIPTION: The END statement is used to indicate the end of a VGEN program. The same keyword is also used to indicate the end of a FOR loop, REPEAT loop, WHILE loop, a SWITCH block, data TABLE and a SUBROUTINE definition. If the END statement is encountered without a matching BEGIN then it is assumed to be the end of the program. Thus, inserting an END statement in a program before its end allows the programmer to control where the compilation stops.

FCLOSE

USAGE: FCLOSE *file-pointer* ;

WHERE: *file-pointer* is a pointer to a file that was opened previously with the FOPEN command.

EXAMPLE: FCLOSE fptr1 ;

DESCRIPTION: When reading data from an external file, the file is first opened with the FOPEN command, which designates a file pointer to be used when referencing this file for reads and also to close the file.

FOPEN

USAGE: FOPEN (*file-pointer*, " *fname* ") ;

WHERE: *file-pointer* is an integer file pointer which is used for all later references to this opened file by FREAD and FCLOSE commands. *fname* is the name of the file to be opened for reading.

EXAMPLE: FOPEN (fptr1, " prev.vec ") ;

DESCRIPTION: This command opens an external data file for reading by the VGEN program using FREAD commands. Multiple external data files can be open simultaneously.

FOR

USAGE: FOR *var* =valu1 UNTIL valu2 [STEP valu3] BEGIN

. . .

. . .

END

WHERE: *var* is an integer variable name that has been declared with an INT statement.

valu1, valu2, valu3 are either integer values, integer variables or expressions.

EXAMPLES:

```
FOR i = 0 UNTIL (X+33) BEGIN CALL CKCYC END;

FOR i = j UNTIL 0 STEP -2 BEGIN
    ...
    END
```

DESCRIPTION: The FOR statement provides for looping on a series of statements. Any variables (e.g. var) must have been previously declared. The body of the FOR loop is contained between the BEGIN and END keywords and may include other FOR loops, subroutine calls or any other valid VGEN statement. STEP valu3 is optional. If not specified, the step value is either 1 or -1 depending on valu1 and valu2.

FORK

USAGE: FORK *x*;

WHERE: *x* is the FORK name and can be any of the following; a, b, c, d, e.

EXAMPLE: FORK B;

DESCRIPTION: The FORK statement switches to a time line that is parallel to the current time line. All statements following a FORK statement accumulate vectors on this parallel time line and each time line starts at time = 0 initially. Parallel time lines can have different CYCLE times and a program can switch back and forth between time lines using the FORK statement. FORK A is the default main

time line. Different FORKs will typically manipulate different input pins - all of which must have been declared with an INPUTS statement. Different FORKs can also manipulate common pins; however, two FORKs cannot drive a pin to different logic states at the same time. A FORK which wishes to relinquish control of a pin it has been driving so that another FORK can begin driving it, must assign a 'Z' or 'X' state on the pin at the time it wishes to release it. At the end of the VGEN program, all time lines (FORKs) are merged together into a single vector file.

FREAD

USAGE: FREAD (*file-pointer*, " *format-string* ", *arg-list*);

WHERE: *file-pointer* is the name of the file pointer used when the file was opened with the FOPEN command.

" *format-string* " is a quoted string of both hard-characters and variable format descriptors. The legal variable format descriptors are:

%d	integer number
%nB	n Binary characters
%nO	n Octal characters
%nH	n Hex characters
\nN	skip n lines (no n means go to next line)
\nS	Skip forward n character positions

arg-list is a list of pin, group, integer or string variable names.

EXAMPLES: FREAD (fptr1, " %d > %32B ; \n", ntime, allpins) ;

FREAD (fptr2, " %4H, %2B : %d \n", bus1, mode);

DESCRIPTION: This command performs a read operation on the file specified by the file-pointer. During the read, it tries to match items in the format-string with data in the file. Variable data, identified with % descriptors, is assigned to arguments in the arg-list. A successful read sets $FILE_STAT to 0, a failed read or EOF sets it to 1.

IF/THEN/ELSE

USAGE:

IF *test* THEN *statement* ;
[ELSE *statement* ;]

WHERE:

test has the form:

a 'rel' b [op (test)]

a and b are expressions and 'rel' can be:

\==\	equal to
\<=\	less than or equal to
\>=\	greater than or equal to
\!=\	not equal to
\<\	less than
\>\	greater than

compound tests can be formed using the & (AND) and | (OR) operators (op) between tests; use parentheses to separate tests.
statement is any legal VGEN statement or a statement block starting with BEGIN and terminating with END.

EXAMPLES:

```
IF (A\==\B) & (SET\<\4) THEN CALL WRITE(A+1);
ELSE CALL WRITE(B-1);
IF $TIME \<=\ (MAXT+12)/2 THEN BEGIN

        . . .
        END;
ELSE BEGIN

        . . .
        END;
```

DESCRIPTION:

The IF statement provides the capability to do conditional execution of program segments. There are six tests that can be performed to determine whether or not the statement or statement block following the THEN keyword is executed. Each test evaluates whether or not a relationship holds between the expression on the left and the one on the right. If the relationship is evaluated to be true then the statement or statement group following THEN is executed. If it is not true, then an ELSE is looked for. If it is found the statement or statement block following the ELSE keyword is executed. If no ELSE is found, then program flow continues with the next statement. Compound tests can also be specified.

PINTIMING

USAGE: PINTIMING ON *pinlist* ; or

PINTIMING OFF *pinlist* ;

WHERE: *pinlist* is a pin name, group name or list of these.

ON or OFF specifies the enabling or disabling of the timing properties.

EXAMPLES: PINTIMING ON DBUS, ABUS, CLK;

PINTIMING OFF MODE[3,2,1], CONTROL;

DESCRIPTION: The PINTIMING statement is used to enable and disable the behavior and timing properties on pins once they have been defined using the PINTYPE declarative. When a pin's PINTIMING is disabled (OFF) it behaves as though no special timing had been defined. Newly assigned states are assumed by the pin at the beginning of each cycle.

PULSE

USAGE: PULSE ON *pinname* ; or

PULSE OFF *pinname* ;

WHERE: ON or OFF specifies the enabling or disabling of pulsing on the pin.

pinname is a pin name which has been previously defined as a PULSE pin with the declarative PULSE statement format.

EXAMPLES: PULSE ON CLK;

PULSE OFF WE*;

DESCRIPTION: PULSE, in this format, is used to dynamically enable and disable the application of pulses to pins. When enabled, a pulse will be asserted on the named pin during every VGEN cycle.

REPEAT

USAGE:
```
REPEAT value  BEGIN
. . .
END
```

WHERE: *value* is either an integer, a variable or an expression which is evaluated.

EXAMPLES:
```
REPEAT  32  BEGIN  CALL   CKCYC   END;

REPEAT (I+7) BEGIN
. . .
END
```

DESCRIPTION: REPEAT is essentially a short-hand form of implementing a loop where there are no variables involved and the step is always -1. The body of the loop is the statements between the BEGIN and END keywords and the loop is repeated exactly value times.

SWITCH

USAGE:
```
SWITCH  (exp )  BEGIN

    case  ex1:     [statements] ; [break] ;
    . . .
    case exn:      [statements] ; [break] ;
    default:       [statements] ; [break] ;
    END;
```

WHERE: *exp* is an expression; ex1 ... exn are constants, variables or expressions; statements is null, a single statement or any sequence of statements.

EXAMPLE:
```
SWITCH  (expr)  BEGIN

    case  0:
    case  1:      call zip(7);
                         break;
    case  2:      call zap(8);  vgen;
                         pina = ~pinb;  break;
    default:             call zop(2);  break;
    END;
```

DESCRIPTION: The SWITCH/CASE statement block allows the designer to very effectively deal with multiple test situations, i.e. where the action

to be taken is dependent on the value of some variable or function. Execution begins with the evaluation of the SWITCH (expr) expression, which may be either a string or an integer type. The compiler then scans down the list of CASE's to try and find a match, evaluating each CASE value. If a match is found, then the statements following the CASE are executed sequentially until a "break" is reached, at which time the SWITCH block is exited. If no match is found then the optional "DEFAULT" case is taken and its statements are executed. If the logical 'OR' of several CASE's is needed, then simply list these consecutively with no "break" between them as shown in the example above.

TIME

USAGE: TIME [=] *value*;

WHERE: *value* is a positive integer, a variable or an expression; and = is optional.

EXAMPLES: TIME = 5200;

TIME 47000;

TIME ($TIME + MAXDELTA);

DESCRIPTION: The TIME statement allows the advancement of the current time to any value. The new value must be greater than the current value of "time" at the time the statement is executed.

TRACE

USAGE: TRACE ON; or
TRACE OFF;

WHERE: ON or OFF specifies the enabling or disabling of tracing to the CRT screen.

EXAMPLE: TRACE ON;

DESCRIPTION: The TRACE declarative is used to dynamically enable and disable the tracing of program flow to the screen. When enabled, the execution of each VGEN statement causes a one-line trace statement to be printed on the CRT screen which describes the current line number and the type of statement being executed. This facility is intended to aid in the debugging of VGEN programs.

VGEN

USAGE: VGEN(*n*);

WHERE: *n* is an optional positive integer, variable or expression.

EXAMPLES: VGEN;

VGEN(5+(MAXT/$CYCLE));

VGEN(*);

DESCRIPTION: The VGEN statement causes the next set of output patterns that correspond to the current time to be generated. The time is then incremented by the CYCLE value. One VGEN statement may cause the output of several pattern vectors if any of the pins have timing properties defined (and enabled) with the ASSERT, PULSE, WAVEFORM or PINTYPE statements. VGEN will only output a new pattern vector at the next time step if at least one pin has changed state, unless the 'all' option is specified in the CYCLE statement. The optional parameter n can be used to cause repeated execution of the VGEN statement n times. If a * is used as the n parameter then VGEN statements will be executed until the active time line's current "time" is equal to the furthest advanced time line's current "time". This is a mechanism for synchronizing multiple time lines in a program and should only be used when the FORK statement is being used.

WAVEFORM

USAGE: WAVEFORM ON pinname [*offset*]; or
WAVEFORM OFF pinname;

WHERE: ON or OFF specifies the enabling or disabling of the waveform on the pin.

pinname is a pin name which has been defined with the declarative WAVEFORM statement format previously.

[*offset*] is an optional time offset, relative to current "time" at which to begin application of the waveform signal. If no offset is specified the first transition of the waveform is applied at the start of the cycle.

EXAMPLES: WAVEFORM ON CLK 23;

WAVEFORM OFF WE*;

DESCRIPTION: WAVEFORM, in this format, is used to dynamically enable and disable the application of waveform stimulus to a pin. When enabled, a continuous periodic waveform, as defined in the declarative WAVEFORM statement, will be applied to the pin.

WHILE

USAGE: WHILE *test* BEGIN

. . .

END;

WHERE: test has the form:

a 'rel' b [op (*test*)]

a, b are expressions and 'rel' can be ;

```
\==\    equal to
\<=\    less then or equal to
\>=\    greater than or equal to
\!=\    not equal to
\<\     less than
\>\     greater than
```

compound tests can be specified using the & (AND) and | (OR) logical operators; separate tests with parentheses.

EXAMPLES:

```
WHILE  (A\!=\B) | (C\==\D) BEGIN
        CALL   ALB(A)
        END;
WHILE $TIME \>\MAXTIME BEGIN
        DBUS = TOP;
        CALL CLOCK(C+4);
        IF $TIME \>=\ TYPTIME THEN CALL HO;
        END;
```

DESCRIPTION: The WHILE statement is another form of program loop. As long as the test relationship evaluates to "true" then the program statements between the BEGIN and END keywords will continue to be looped through. After each pass thru the program statements, program flow passes back to the top of the loop where the relationship is checked again. If it is true, then another pass thru the loop is performed. If it is false then program flow jumps to the first statement past the loop END keyword. Compound tests can be formed using the & and | logical operators; parentheses should be used to control the order of evaluation.

APPENDIX B

VGEN TIMING WAVEFORMS

APPENDIX B

VGEN TIMING WAVEFORMS

FIGURE B-1

GENERAL SYNCHRONOUS PIN TIMING

VGEN PROGRAM SEGMENT

```
CYCLE 200;
        . . .
FOR I=0 UNITL 256 BEGIN
        AD = I;
        DB = (I * 24) < 4;
        CTRL = 0  @  40 ; VGEN;
        CLK = LO; VGEN;
        CTRL = 1;
        CLK = HI; VGEN;
        END;
  ...
  ...
```

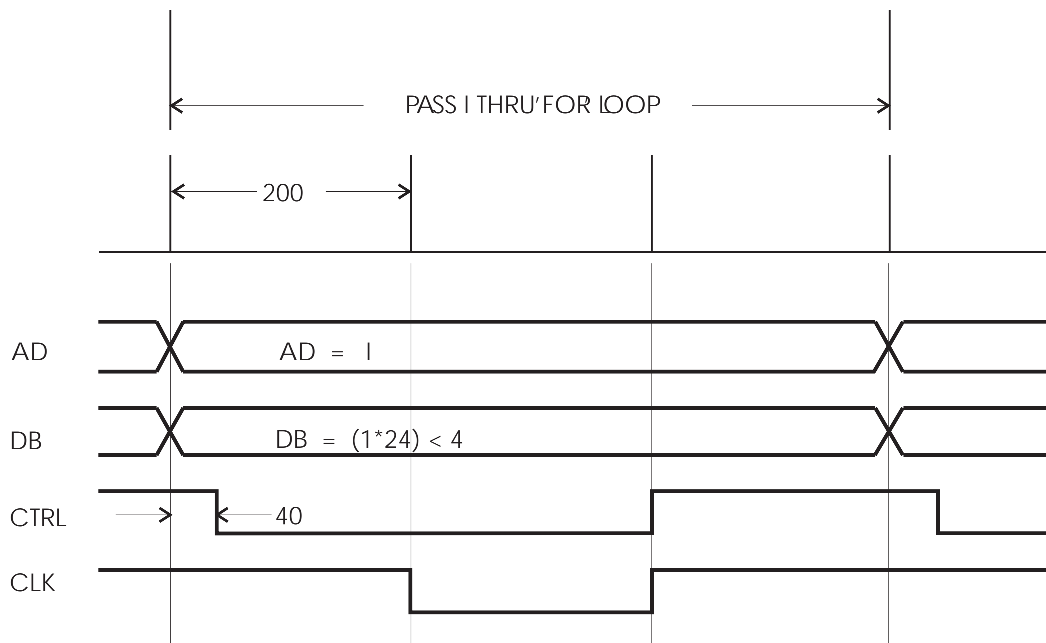


FIGURE B-2

PIN ASSERTION   TIMING

VGEN PROGRAM SEGMENT

```
CYCLE 200;
ASSERT AD @ -20;
ASSERT DB @ -50;
ASSERT CTRL @ 50;
      . . .
FOR I=0 UNITL 256 BEGIN
      AD = I;
      DB = (I * 24) < 4;
      CTRL = 0;
      CLK = 0; VGEN;
      CTRL = 1;
      CLK = 1; VGEN;
      END;
   ...
   ...
```

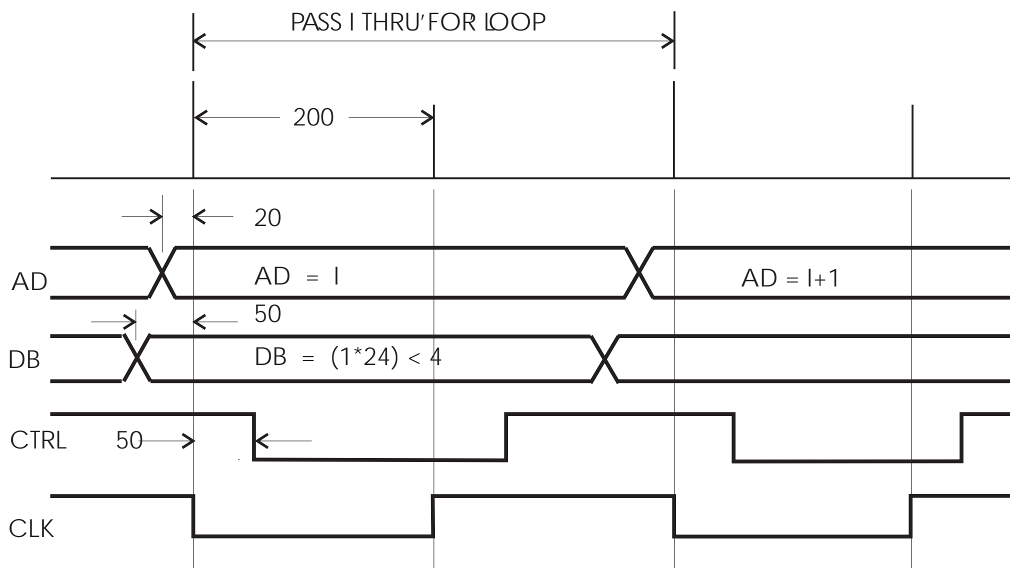


FIGURE B-3

PIN TIMING USING PULSE

VGEN PROGRAM SEGMENT

```
CYCLE 200;
PULSE WE*  20  80  0;
      . . .

PULSE  ON  WE*;
DBUS = '22';
ADDR = '3C2';
VGEN;          { OUTPUT VECTOR CYCLE N }
PULSE OFF;
   ...
```

FIGURE B-4

PIN TIMING USING PINTYPE

VGEN PROGRAM SEGMENT

```
CYCLE 200;
PINTYPE  NRZ  DBUS  @  60 ,
     NRZ  ADDR  @  -20 ,
     RO  WE*  @  20  100 ;
PINTIMING  ON  DBUS,  ADDR,  WE* ;
     . . .
     . . .
DBUS = '22';
ADDR = '3C2';
WE* = 0 ;
VGEN;        { OUTPUT VECTOR CYCLE N }
WE* = 1 ;
  . . .
```

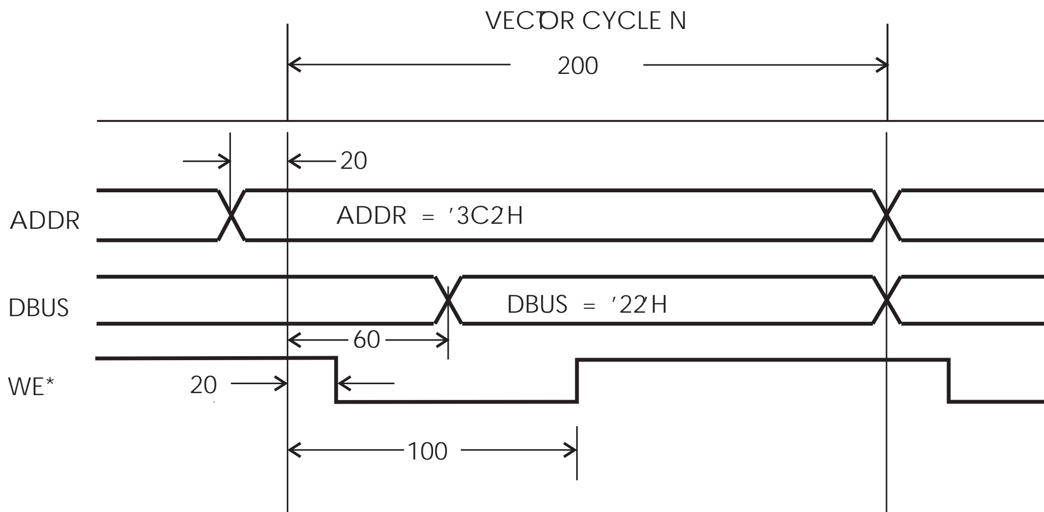


FIGURE B-5

PIN TIMING USING WAVEFORM

VGEN PROGRAM SEGMENT

```
CYCLE 200;
PULSE  WE*  20  80  0 ;
WAVEFORM  CLK  92  48  0 ;
      . . .
      . . .
WAVEFORM  ON  CLK  28 ;
PULSE  ON  WE* ;
DBUS = '22';
ADDR = '3C2';
VGEN;          { OUTPUT VECTOR CYCLE N }
   . . .
```

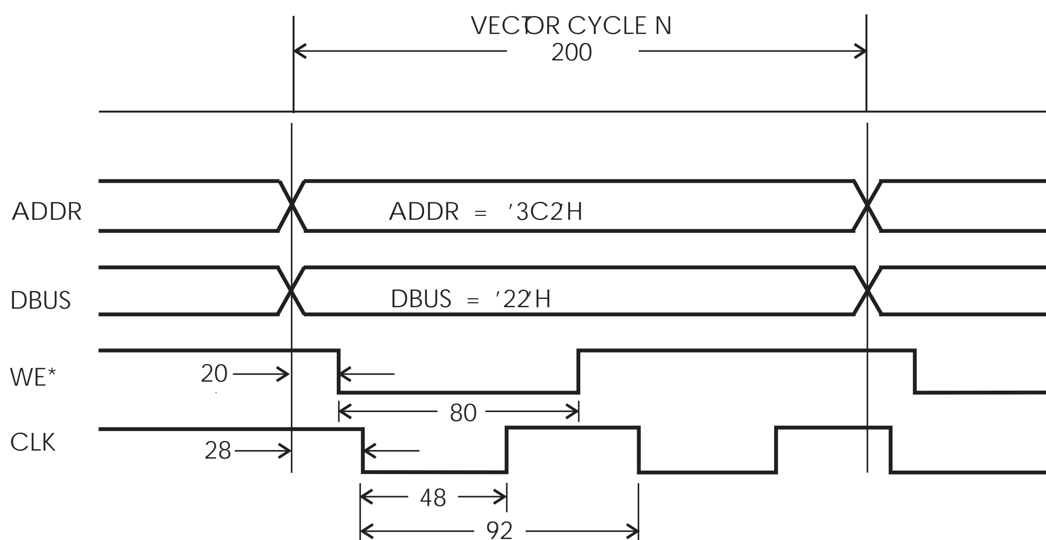


FIGURE B-6

APPENDIX C

INTERMEDIATE VECTOR FILE (IVF) FORMAT

APPENDIX C

INTERMEDIATE VECTOR FILE (IVF) FORMAT

All Source III Simulation Data Management products utilize the file format defined in this specification for the storage of intermediate data files during vector generation, translation and analysis. The format is referred to as the Intermediate Vector File (IVF) format. VGEN generates a file in this format as an intermediate step, prior to formatting vectors for the target simulator. The file can be saved for viewing or later use with the KEEP_IVF command. VTRAN also uses this file format for intermediate data storage after loading and processing the Original Vector File. Again, the file can be saved for later use or viewing with the KEEP_IVF command. For both VGEN and VTRAN, if no SIMULATOR command is specified, the resulting output vectors will be in the IVF format. IVF files are similarly used (read) and generated by VCAP and VTEST in the course of processing their data.
The IVF format is composed of two sections and can be directed to be either a pure ASCII file, or a compacted file where most of the vector data is compacted into a binary format. The first section of the file is the header and is always ASCII text. This section contains a UNITS declaration followed by a pin list and has the following format:

```
UNITS u
PINS xpin1 xpin2 ... xpinn;
```

where u is either 1, .1, .01 or .001 and x is the integer 0, 1, 2, or3 and defines the pin direction according to the following rules:

0 - input pin

1 - output pin

2 - input version of a bidirectional pin

3 - output version of a bidirectional pin (.O)

The UNITS value specifies the time units to be applied to the absolute or delta time stamps of the vector lines which follow. The list of pins is separated by spaces, may span multiple lines and terminates with a semicolon. Pins which are busses are split out into individual pins with a single bit vector subscript. For example, the bus ADR[3:0] would appear in the pin list as "ADR[3] ADR[2] ADR[1] ADR[0]" . Bidirectional pins are listed twice; once for the input version of the pin (using its specified name) and again for its output version (using its specified name with a .O suffix). A typical IVF header might look as follows:

```
UNITS .1
PINS 0clk 0ad[3] 0ad[2] 0ad[1] 0ad[0] 0pin1 0pin2
     1op1 1op2 1op3 2bus[1] 2bus[0] 3bus.O[1] 3bus.o[0] ;
```

In the first line the UNITS is specified as .1 meaning that the integer time stamps are tenths. The next lines define input pins clk, ad[3:0], pin1 and pin2; output pins op1, op2, op3; bidirectional inputs bus[1:0] and the corresponding output versions bus.O[1:0]. The order of the pins in the PINS header defines the order of pins in the following vector data of the IVF. The first pin is pin number 1, corresponding to the first column of the vector data, and so forth.

The second section of the IVF contains the actual vector data. This data is comprised of vector lines, where each line represents a new time stamp and set of state data for the pins. As mentioned above, the order of the state data for the pins in each vector is defined by the order of the pins in the PINS header. The individual vector lines in this section can be in one of three formats. The first format is shown below:

> "dtime" - "states";

where *dtime* is a positive integer number representing the delta time stamp for the state data - relative to the last vector time, and *states* is a list of characters designating state information for the pins defined in the header. An example vector might be:

> 160 - 1100101ZZZ;

A second ASCII vector format which is supported has the following form:

@ "time" - "states";

where *time* in this case is an absolute integer time stamp. The first vector after the PINS header information in an IVF file is always in one of these two formats. Successive vector data in the IVF may be either in one of these formats or in an alternate compacted binary format. This compacted format has the following form:

^dtime (pin/state list)0

where *dtime* is a 4-byte (long integer) binary number representing the delta time stamp for the vector data - stored with the most significant byte first, (pin/state list) is a list of pin number / state character data sets, and 0 is a binary 0 byte (or 2 bytes) which terminates the pin/state list. Each entry in the pin/state list is composed of a pin number and a state character. The pin number may be either 1 or 2 bytes, depending on whether or not the number of pins in the PINS list is less than 255. When the number of pins is less than 255, a single byte is used for the pin number. When the number of pins is greater than or equal to 255, two bytes are used with the first byte being the most significant byte. In this case, the terminating 0 is also 2 bytes. Pin numbers start with number 1 (corresponding to the first pin in the PINS header) and hence the 0 byte (or 2 bytes) terminates the list. Each vector line is then terminated with a "newline" (\n in C). In the compacted format, each vector contains only those pins which have changed state from the last vector and hence can result in a significant reduction in file size.
The IVF file is the terminated with two left-angle brackets on separate lines:

<
<

Within a single IVF, the vector line types can be intermixed, but the first vector must always be one of the two ASCII forms.

INDEX

ASSERT .... 13, 35 ,A-2, B-3

Assignment Statement 27

BEGIN 21, 46, A-18

BIDIRECTS 6, 13, 40, 64, A-3

BREAK 21,46, 49, A-18

BUSFORMAT 14, 55, A-3

BUS/VECTOR Notation 11

CALL 21, 49, A-18

CASE 21, 52, A-19

Characters (Special) 6

COMMENT(S) 22, 53, A-19

CondItional Statements 51

CONTINUE 22, 47, A-20

CYCLE 4, 14, 33, 35, A-3

Data Tables 31

Data Types 6

Declarative Statements 13, A-2

DEFAULT 8, 14, 52, A-4

DEFINE_HEADER 14, 55, A-4

ECHO 22, 67, A-20

ELSE 23, 51, A-24

END 22, 46, A-21

EXPECT_FILE 15, 59, 63, A-5

Expressions 29

External Files 58, 59

FCLOSE 22, 59, A-21

FOPEN 23, 59, A-21

FOR 23, 46, A-22

FORK 23, 41, A-22, B-7

FREAD 23, 59, A-23

GROUP 3, 6, 15, A-5

HEADER 15, 55, A-6

HI 3, 9

IF 23, 51, A-24

INCLUDE 15, 58, A-6

INITIALIZE_PINS 15, A-6

INPUTS 3, 16, A-7

INT 7, 10, 16, A-7

INTEGER Data Type 7

IVF Format .... 58, 59 Appendix C

KEEP_IVF 16, A-7

KEEP_LOWERCASE 16, 55, A-8

Keywords 10

LO 4, 9

MAKE_SF 16, A-8

MERGE_BIDIRECTS 17, 64, A-8

MERGE_FILE 17, 58, A-9

Multiple Time Lines 41, B-7

Operators (Arithmetic/Logical) 32

OUTPUTS 17, 63, A-9

Pattern Generation Statements 21, A-18

PIN Data Type 6

PINTIMING 24, 40, A-25, B-5

PINTYPE 17, 38, 64, A-9, B-5

INDEX (continued)

PULSE 17, 24, 36, A-11, A-25, B-4
REPEAT 24, 46, A-26
SCALE 18, 33, A-11
SIMULATOR 18, 56, A-12
STATE_TRANS 18, 56, 63, A-12
STEP 46, A-22
STRING 8, 10, 18, A-12
SUBROUTINE 18, 49, A-13
SWITCH 24, 52, A-26
Syntax (VGEN) 5
SYS_CLOCK 19, A-14
SYSTEM_CALL 19, A-14
TABLE 19, 31, A-14
THEN 23, 51, A-24
TIME 25, 44, A-27
Timing Control (of pins) .... 33, Appendix B
TITLE 19, A-15
TRACE 25, 67, A-28
UNITS 19, 55, A-15
UNTIL 23, 46, A-22
USER DEFINED FORMATS (UDF) 56
VCAP 59, 63, A-5
VGEN 3, 4, 25, 44, 66, A-28
VTRAN 58
WAVEFORM 20, 25, 41, A-16, A-29, B-6
WAVES 20, A-16
WHILE 26, 47, A-29
WHITESPACE 20, 60, A-17
$CYCLE 7
$DATE$ 53
$FILE_STAT 7, 61
$SIMULATOR 9
$TIME 7
$TIME$ 53, 67