Pattern Language Reference

6

Overview

The FLASH 750 pattern language is the programming language for writing microcode patterns. Patterns may include microcode produced by the FLASH 750 Address Data Generator (ADG).

A FLASH 750 pattern for testing a memory device consists of a named block of *microinstructions*, which are like C++ functions. Patterns are independent of the memory device geometry.

The pattern timing and other tester functions are defined symbolically by user-defined structures external to the pattern, although both the patterns and C++ source may be present in the same source file. The pattern compiler translates the pattern microcode into C++ compatible data structures. Once the patterns are compiled, they are linked with test programs; refer to *Pattern Compiler and Pattern Reverse Compiler*, Chapter 7.

This chapter is divided into the following sections:

- Memory device testing, page6-2
- FLASH 750 pattern language for testing memory with or without embedded logic devices, page 6-12

See also:

Address Data Generator (ADG), Appendix B.

Routine Reference, Appendix G.

Testing Memory Devices

Overview

A memory device holds data by storing charges in individual cells that are addressed by their X, Y, and Z locations in the array. These locations are tested to determine if they can read, write, and hold data under various conditions. However, it is not practical to test every cell under every condition because most memory arrays contain a large number of cells. Consequently, the typical test patterns are designed to detect faults in the minimum amount of test time.

See also:

Unscrambled Memory Arrays on page 6-6

Scrambled Memory Arrays on page 6-7

Testing a Memory Array on page 6-9

Topological Inversion on page 6-11

Types of Memory Tests

Overview

A test pattern contains functional tests that isolate problems in the design and fabrication of memory arrays. These tests consist of geometric patterns of 0s and 1s that are addressed to the array cells. These patterns are created by the Address Data Generator (ADG). To generate the pattern, the ADG puts out a sequence of x and y cell addresses, along with the data (0 or 1) to be written to the cell or read back from the cell. From this data, the tester determines if the cells can properly read, write, and hold data.

Checkerboard and Diagonal Patterns

Two common patterns for functional testing of memory devices are the checkerboard pattern and diagonal patterns.

As shown below, the *checkerboard pattern* writes alternate 0's and 1's so each cell is surrounded by its complement in the four adjacent cells. The checkerboard pattern is then read back

0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0

As shown below, this *diagonal pattern* writes all cells to 0, and then writes a 1 to each cell with equal x and y addresses. It then reads back all cells.

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1

Other Memory Test Patterns

Patterns other than the checkerboard and diagonal can be generated by the ADG to test a memory array. The following memory array functions can be verified by the various test patterns:

- Each device cell can store a logical 1 and a logical 0:

  Write all 1s - read all 1s

  Write all 0s - read all 0s

- Every device cell is present and addressable:

  Write all 0s, then read/modify/write to 1s

  Write all 1s, then read/modify/write to 0s

- Address decoders can select the correct cell address. This is tested by applying patterns in various address sequences:

  Galloping address sequence

  Row galloping address sequence

  Column galloping address sequence

  Diagonal galloping address sequence

- For multi-wide devices, verify each memory array is separate; no shorts between the inputs and outputs by storing a different data pattern in adjacent arrays
- Chip can be deselected
- Search for weak cells or interaction between cells by using surround disturb tests that write a 1 in a cell and write 0’s in the eight adjacent cells (example: butterfly pattern).
- Verify the timing and level specifications by running each test at different combinations of timing and levels.

Memory Arrays

Overview

Each cell in a memory array has an (X,Y) address. The location of the starting address determines the type of array:

- unscrambled—An unscrambled memory array has its addresses start at zero in one corner of the array. The addresses increase sequentially to the maximum cell address. In an unscrambled array, the physical cell addresses are sequential; refer to *Unscrambled Memory Arrays* on page 6-6.
- scrambled—In a *scrambled memory* array, cells with sequential addresses are scrambled or out-of-sequence locations in the array, rather than adjacent locations; refer to *Scrambled Memory Arrays* on page 6-7.

Unscrambled Memory Arrays

In an unscrambled memory, cells are addressed from address zero to the maximum address by sequencing the address decoders on the device from one corner of the device to the diagonally opposite corner, writing to or reading from the addressed cell. As shown below, the cell addresses in an unscrambled memory array increase in sequential order. The cell addresses are identical to the physical locations

.

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Because the cell addresses in an unscrambled array are sequential and identical to the physical locations of the cells, the arithmetic features of the ADG can generate the cell addresses for test patterns by algorithm, rather than by specifying each separate address. For example, you can specify N+2 rather than 2, 4, 6, 8.

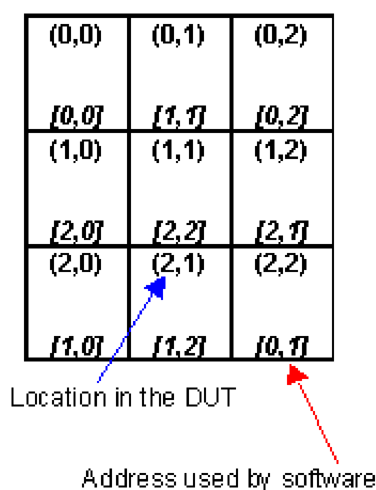
An unscrambled memory array is the ideal situation because the cell addresses for writing memory test patterns are identical to the physical locations of the cells. However, because of device design considerations, cell addresses that are logically consecutive are not usually physically adjacent to each other in the memory array. These cells are *scrambled.*

Scrambled Memory Arrays

In scrambled memory arrays, sequential cell addresses are in various physical locations on the device rather than in sequential physical order. Scrambling improves device performance by placing each memory cell in its most advantageous physical position, even though a test engineer considers the array as a group of logically sequential cells.

In the *scrambled* memory array shown below, the addresses of the sequential cells (in italics and square brackets) are in physical locations (in parentheses) that are not adjacent

.



To map the scrambled software addresses to the physical locations on the device, a *descramble table* is used**;** refer to the following sample:

Software Address	Physical Location
[0,0]	(0,0)
[0,1]	(2,2)
[0,2]	(0,2)
[1,0]	(2,0)
[1,1]	(1,0)
[1,2]	(2,1)
[2,0]	(1,0)
[2,1]	(1,2)
[2,2]	(1,1)

Addressing Scrambled Arrays

The first step in creating a test pattern for a scrambled array is to determine the logical addresses for each physical addresses of the device. How the physical cells are assigned affects the accuracy of the test pattern in determining how physically adjacent cells interact. To determine the address assignments, the logical cell addresses in the test program must be mapped to the physical locations of those cells in the array by using the *address scrambler* and the *descramble tables* provided by the device manufacturer.

Descramble Table

A descramble table, provided by the manufacturer, lists how the cell addresses used by software programs are assigned to the physical locations of those cells. Refer to the following sample:

Software Address	Physical Location
[0,0]	(0,0)
[0,1]	(2,2)
[0,2]	(0,2)
[1,0]	(2,0)
[1,1]	(1,0)
[1,2]	(2,1)
[2,0]	(1,0)
[2,1]	(1,2)
[2,2]	(1,1)

This address scrambling information is entered into a test program database. it is then programmed into the ADG address scrambler so it can generate the correct pattern data for a device.

Scrambling the Address

After the scrambling information is entered, the address scrambler maps the data output from the ADG to the correct location in the device memory. The desired memory test pattern is now correctly applied to the address-scrambled memory. Once the address scrambling information has been entered, you do not have to know the address scrambling arrangement of the device. You can create patterns as if the memory array was sequential from zero to the maximum cell address.

Testing a Memory Array

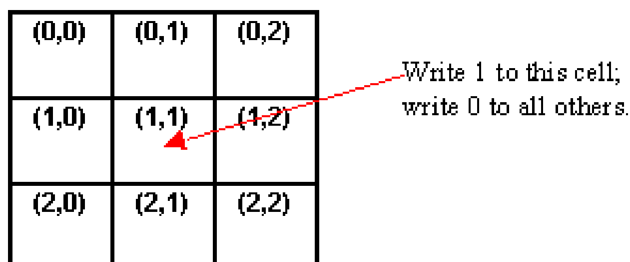
The following examples use a surround test pattern. In a surround pattern, a *target* cell receives a charge and then is surrounded by cells holding the complement of its charge. For example, the target cell receives a 1, and each of the cells surrounding it receives a 0. The device is then tested to see if it holds this pattern.

These kinds of patterns require address cells that are physically adjacent; consequently, the test system must translate the cell addresses generated by the ADG to the corresponding physical locations in the array.

Example: Unscrambled Memory Array

The test program writes a 1 to the center cell of a sample array shown below, whose address is (1,1), and then writes a 0 to all the surrounding cells.

Address	Data
(0,0)	0
(0,1)	0
(0,2)	0
(1,0)	0
(1,1)	1
(1,2)	0
(2,0)	0
(2,1)	0
(2,2)	0

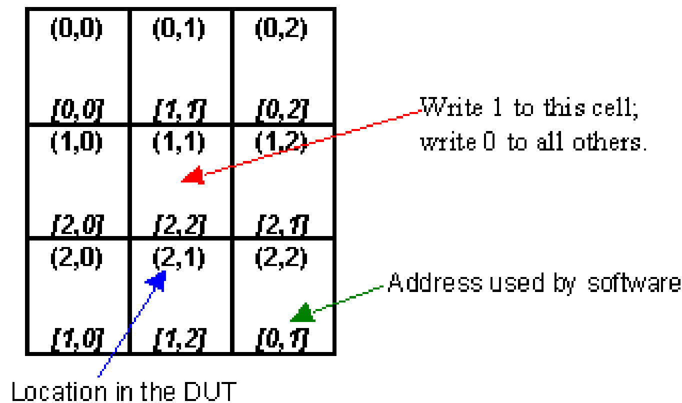


The ADG output for the surround test pattern does not have to be descrambled because the logical addresses correspond to the physical locations. Thus, the generated pattern can write the output to the logical address, without descrambling.

Example: Scrambled Memory Array

In the sample array shown in the figure below, the test program writes a 1 to the center cell, whose address is (1,1), and then writes a 0 to all the surrounding cells:

Address	Data
(0,0)	0
(0,1)	0
(0,2)	0
(1,0)	0
(1,1)	0
(1,2)	0
(2,0)	0
(2,1)	0
(2,2)	1



The address scrambler must use the descramble table to map the software address in the program (italic in figure) to the physical location. The address scrambler output writes the following data to the corresponding physical location.

Topological Inversion

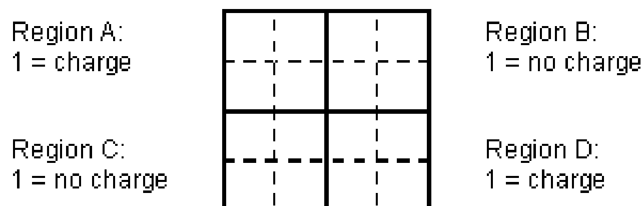
Overview

Like address scrambling, topological inversion in a memory device must be considered for accurately addressing specific locations in memory arrays. Topological inversion is a method of storing the inverse of the normal charge in specified regions of the DUT.

Memory arrays are either static or dynamic. A static array consists of latches, while a dynamic array stores a charge on a capacitor. Both types of memory arrays consume considerable power, even though they are designed to be energy efficient.

Memory devices can consume less power by designating certain regions as having no charged cells (0) when a cell is actually charged (1). This opposite condition is known as topological data inversion.

For example, consider a device in which a 1 is indicated by a charged capacitor; see the following figure. The device has four regions; two regions, B and C, are inverted. In the inverted regions, 1 means the capacitor is not charged.



When this device is tested for writing all 1s (a common memory test pattern), only half the amount of power is needed because half the cells indicate a 1 by no charge.

Compensating for Topological Inversion

Once the device regions using inversion have been specified in the test program, the device is tested by inverting individual bits of data based on the specified address information. In the regions identified as inverted, data written to the device is complemented: when 0 is written algorithmically to the device, the physical result is true 1 inside the device. In this way, the tester compensates for the device inversion to correctly perform surround and other tests.

Pattern File

Components

An ASCII pattern file consists of the following items:

Pinmap Statement
: Specifies the workbook and sheet that define the pins and pin groups. Unless only channels are used, this pin information is required from the control statements in the pattern file or from a compile-time option.

Preprocessor Statements
: Optional C++ preprocessor constructs.

Compiler Control Statements
: Available at compile time.

Imported tset or Label Statement
: Imports **tset** (timing set) names and any external labels.

Pin_setup Statement
: Specifies the pin mode and display forms. Required for **io_midband**, **io_valid**, high voltage, or frequency counter pins; recommended for **mux** mode pins.

Vector Statement
: Contains a pin list and vector data. Vector data includes:

a label, page 6-34
an opcode, page6-51, or control bits, page6-88 or both
a timing set (tset) field, page 6-39
the actual vector data, page6-31

Every pattern file must have only one vector statement.

Comments
: Optional C++-style comments.

Format

- All statements are in free format.
- Elements are delimited by white space, including tabs and line breaks.
- Line breaks in the syntax are not significant. Statements can occupy multiple lines.

Case

Case is significant and must be used as listed except for the following items that are case-insensitive:

- Labels
- **tset** (timing set) names
- Tester data characters and hexadecimal numbers. Examples: both **H** and **h** mean *expect high,* and **.dADB** and **.dadb** are both equivalent specifications in hex for drive data.

Syntax Notation

bold	Enter keyword or symbol as listed in lowercase.
ital	A placeholder. Example: *filename* or *integer*. Replace it with the appropriate name or value.
|	Logical *OR*; choose one of the items separated by the vertical line.
[**aaa**]	Item in square brackets is optional. Brackets are not part of the item.
...	Preceding item is repeated.

+ The syntax notation in this document emphasizes readability. Some users prefer a more formal representation of the syntax, for example, when writing a parser; refer to *Memory Syntax in Backus-Naur Form* on page 6-105.

Comments

Comment Formats

The pattern compiler accepts C++-style comments:

- form includes everything between the delimiters:

/* *comment-text* */

- end-of-line form is terminated by a carriage return:

// *comment-text*

Saving Comments in the Compiled File

By default, the pattern compiler strips comments to minimize the size of a file. To override the default, use the *Save Comments* option when compiling. When this option is used, the C++ preprocessor and the pattern compiler retains the comments for the pattern tools and pattern reverse-compiler.

Compiling Comments

- Pattern compiler associates any comments it finds with the vector it is compiling.
- Comments in the middle of a vector are moved to the end of the vector by the pattern tools and pattern reverse-compiler.
- If a vector has multiple comments, the comments are grouped as a single comment in the pattern output file; consequently, the ASCII output of the pattern reverse-compiler may be different from the original ASCII input.
- Pattern compiler retains information about the type of the comment, /* */ or //. This information is preserved during reverse-compilation.
- All comments before the first vector, including any before the vector statement, are combined into one comment during compilation.
- All comments after the vector statement are combined into one comment during compilation, which is not the same as the after-last-vector comment.

Reserved Words

User-defined symbols in pattern files, such as pin names, pin group names, labels, or **tset** names, cannot use any of the following reserved words:

allow_hidden	**compressed**	**global**
import	**import_all_undefineds**	**label**
min_period	**opcode_mode**	**output_filename**
pin_setup	**pinmap_workbook**	**pinmap_workbook**
start_label	**subr**	**subr**
subr	**to**	**tset**
tset		

In addition, a user-defined symbol cannot use any word that is the name of an opcode, page6-51, or control bit, page6-88.

Pattern Modes

Pattern files are compiled in either the normal or extended pattern mode. The choice of mode depends on how labels and opcodes are used in vector statement and the rate of pattern execution.

Normal Mode

In the normal mode, patterns can execute at a faster rate, up to the highest frequency supported by the hardware, with the minimum period equal to the shortest period time supported by the hardware. To achieve this rate, the vectors are paired and have the following restrictions:

- If a label is used, it must be on the first vector of a pair.
- If an opcode or control bits other than mask is used, they must be on the second vector of the pair, and they apply to both vectors in the pair. The mask control bit on one or both vectors of a pair applies only to the vector on which it is specified.
- Currently patterns in the normal mode do not support the MTM (memory) opcodes.

Extended Mode

In the extended mode, the maximum execution rate is about half the normal mode, while the minimum period is about twice as long. Extended mode supports an opcode and a label on every vector.

tset data and channel data are per-vector in both normal and extended modes.

For the actual execution rates and valid periods for normal and extended mode, refer to the *FLASH 750 System Specifications*.

Specifying the Mode

By default, the pattern compiler produces extended mode patterns. To select the normal mode, click *No* in the *Extended Mode* checkbox at compile time or include the following compiler control statement in the pattern file:

opcode_mode = normal;

Pattern Memory: LVM and SVM

Definitions

FLASH 750 has two types of pattern memory:

- SVM (subroutine vector memory) is a random access memory that supports all pattern opcodes, including ones that change execution flow, such as **return**, **end_loopA**. SVM can also support labels that are the target of a **jump** or **goto**.
- LVM (large vector memory) has a longer access time than SVM and executes only sequentially. Only a subset of opcodes can be used in LVM.

LVM is far larger than SVM:

- SVM has room for 1K vectors (extended mode) or vector pairs (normal mode).
- LVM has room for 4M vectors (extended mode) or vector pairs (normal mode).

Because a typical test program may have 500K vectors, most vectors must go to the LVM. The pattern compiler optimizes the LVM and SVM by putting as many vectors as possible into the LVM and by putting only vectors that may be executed non-sequentially into the SVM.

Pattern Memory Size

The FLASH 750 system has enough SVM memory for 1K vectors (extended mode) or 1K vector pairs (normal mode). This memory can be reloaded from the LVM, but not while a pattern burst is executing; thus, FLASH 750 limits the total number of vectors a pattern used in a burst can have in the SVM. The limit is 1K vectors in extended mode or 1K vector pairs in normal mode.

To determine how many SVM and LVM vectors are used by a given pattern, use the *Pattern Control Display* to load the pattern (online or offline) and then select the *Mem Usage* tab.

Managing SVM Usage

If your patterns use too many SVM vectors that they cannot be loaded into the SVM, you can try to reduce the number of SVM vectors required by using one of the following methods (in order of importance):

- Minimize labels; page6-18.
- Minimize SVM-only opcodes; page6-19.
- Compile to the minimum period at which the pattern will be executed; page6-19.
- Use **repeat** instead of **mrepeat** if runtime modification is not needed; page6-59.

Minimizing Labels

Labels

Fewer labels are the best way to minimize SVM usage. All global and local labels except start labels (**start_label** syntax) must be in SVM, because the pattern compiler assumes that any label will be branched to or called as a subroutine.

You should not use labels unless they are needed. Thus, do not use labels as visual markers in a pattern file; use comments instead.

Start Labels

Even start labels should be used only when necessary. Although start labels can be in the LVM, their presence may occasionally affect pattern split and cause more SVM vectors to be produced than would otherwise be the case.

Minimizing SVM-Only Opcodes

Opcodes in SVM

Most opcodes and control bits are SVM-only and require that the vectors containing them be placed in the SVM. The opcodes and control bits allowed in the LVM are **repeat**, **mrepeat**, **mask**, **stv**, **ccall**, **halt**, **end_module**, **call** (without an **if**) and **jump** (without an **if**). Using any other opcode or control bit forces the vector containing it to be placed in the SVM rather than the LVM.

MTM Opcodes in SVM

All non-default MTM microcode causes the vector containing it to be placed in the SVM. The default MTM microcode do not require that the vectors containing it be placed in the SVM.

Compiling to Minimum Period

Effect of Execution Rate

When creating execution transfers from the SVM to the LVM, the pattern compiler must compensate for the delay between the time the LVM address has been set and the time it becomes valid and execution can be transferred from the SVM to the LVM. When calculating the number of SVM vectors needed before the LVM address is valid, the pattern compiler assumes the patterns execute at the fastest rate supported by the hardware. This assumption limits the SVM optimization.

Specifying a Slower Execution Rate

If your patterns will run at a rate slower than the maximum supported, you can reduce the SVM usage by compiling your patterns for the minimum period at which they will be executed. You specify the minimum period at compile-time. In some cases, the slower execution rate means that fewer vectors will need to be placed in the SVM.

Pattern Statements

Overview

The following pattern statements are supported:

- Pinmap Statement, page6-21
- Preprocessor Statements, page6-22
- Compiler Control Statements, page 6-23
- Imported tset or Label Statement, page6-25
- Pin_setup Statement, page 6-26
- Vector Statement, page6-30

Pinmap Statement

Overview

The pattern compiler accesses the pinmap information to identify a pin or pin groups and to determine the number of pins in a group. The pinmap information is in the pinmap sheet of an **IG-XL** workbook.

You specify the workbook and the pinmap sheet name either by using the compile-time options of the **Pattern Compiler** or by putting a **pinmap_workbook** and a **pinmap_sheet** statement into the pattern file.

Syntax

pinmap_workbook = *filename*;

pinmap_sheet = *sheetname*;

where:

filename is the name of the **IG-XL** workbook with the extension .*xls*;

sheetname is the name of the *Pin Map* sheet (name on the tab) in the workbook.

Location

These optional statements must appear in the pattern file before the vectors.

Requirements

- If the workbook has only one *Pin Map* sheet, the **pinmap_sheet** specification is not required.
- If the workbook has more than one *Pin Map* sheet, the **pinmap_sheet** specification is required; thus, if no **pinmap_sheet** is specified, an error is generated.
- If the pattern file uses channel numbers instead of pin names, a *Pin Map* sheet is not required.

Compiling Pinmaps

- If the compile-time option and either the **pinmap_workbook** or the **pinmap_sheet** are used, the compile-time option overrides the statement without issuing a warning.
- Pattern compiler generates an error if a pin or group name in the **vector** statement pinlist is not in the *Pin Map* sheet.
- Pattern compiler does not verify that pins declared as inputs in the pinmap use only drive data, or that pins declared as outputs in the pinmap use only receive data.

Preprocessor Statements

Overview

The pattern language supports all C++ preprocessor constructs, such as **#include**, **#ifdef,** and **#define**.

The pattern compiler does not run the C++ preprocessor unless you invoke it by using a compile-time option (*Run Preprocessor* option or *–cpp* switch); refer to *Preprocessor Definitions* on page 7-9.

The pattern compiler supports passing macro definitions to the preprocessor.

Preprocessor Comments

If you do not use any preprocessor constructs other than comments, not running the C++ preprocessor speeds up pattern compilation.

Compiler Control Statements

Overview

The pattern language supports optional compiler control statements. If used, control statements must appear in the pattern file before the **vector** statement.

Control Statements and Compile-Time Options

These options can also be specified at compile time by using the pattern compiler; refer to *Compiler Options* on page 7-6. Using a control statement instead of a compile-time selection preserves the option during reverse-compilation from the pattern binary file, which may be less error-prone than compile-time selections. Not all compile-time selections have corresponding control statements.

If a compile-time selection conflicts with a control statement, the compile-time selection has precedence, and the control statement is ignored.

Compiler Control Statements

output_filename = filename;

Defines the output filename. Default is the input filename, with the extension replaced by *.pat*.

version = text;

A user-defined string is placed in the pattern binary, intended for customer-version tracking. This string is never compressed in the binary file.

min_period = double;

Defines the minimum length of a period, in seconds, when the pattern is a burst. The range of valid values depends on the pattern mode: normal or extended; refer to the *FLASH 750 System Specifications* for these minimum and maximum values. This information is optional, but specifying a value instructs the compiler to optimize the LVM and SVM during compilation; refer to *Managing SVM Usage* on page 6-18.

opcode_mode = normal | extended;

Allows an opcode on every vector (*extended*) or only on the second vector of each pair (*normal*). Default is *extended*; refer to *Pattern Modes* on page 6-16.

svm_only_file = no | yes;

Compiles all vectors for SVM (*yes*) or splits them between the LVM and the SVM (*no*). Default is *no*. Compiling SVM-only is required when relative addressing is used, or subroutines are called from other subroutines. The SVM-only option requires the entire file to have less than 1K vectors.

import_all_undefineds = no | yes;

Specifies how the pattern compiler reacts to references with undefined **tset** names or unresolved labels at the end of compilation: import them without issuing a warning or error message (*yes*) or issue error message (*no*). Default is *no*; refer to *Imported tset or Label Statement* on page 6-25.

You can use this control statement with any preprocessor that generates the names on the fly; however, Teradyne does not recommend using this control statement because the SVM usage cannot be minimized and incorrectly entered names are not caught until the pattern is loaded.

This control statement does not support references whose subroutine or label type cannot be determined by context. For example, if this control statement is used only with a **set_glo** opcode, and not also with a **call** or **jump** opcode, the pattern compiler issues an error message.

Imported tset or Label Statement

tset Names

All **tset** (time set) names in the pattern file must be imported first by using an **import tset** statement so the pattern compiler can detect misspellings; refer to *tset (Timing Set) Data* on page 6-39.

Label Names

External labels referenced in a pattern file must be imported first by using an **import label**, **import subr** or **import svm_subr** statement; refer to *Vector Labels* on page 6-34.

The **import svm_subr** statement should be used for subroutines in files compiled with the SVM-only switch. This statement must be before the **vector** statement in the pattern file. If a subroutine imported by a **import svm_subr** statement is not in the file compiled with the SVM-only switch, the linker issues an error message.

Case Sensitivity

Labels and **tset** names are not case-sensitive.

Pin_setup Statement

Required and Recommended Usage

This statement contains information about the pin mode and display format:

- It is required in any patterns using **io_midband**, **io_valid**, high voltage, or frequency counter pins.
- It is recommended for **mux** mode pins; refer to *Multiplex Mode* on page 6-48.

You can program the **mux** mode pins like ordinary pins, with the data for each muxed pin in a separate column. To have a single data column with two entries per vector for the muxed pins, you use the **pin_setup** statement to indicate which pins are connected; refer to *Vector Data in SCIO and Mux Modes* on page 6-46.

Location

If required, the **pin_setup** statement must appear in the pattern file before the **vector** statement and after any compiler control statements.

Syntax

```
pin_setup = {
        pin-item   io_midband | io_valid    drv_first | rcv_first   ;
        pin-item-1  mux  pin-item-2  ;
        pin-item  high_voltage  ;
        pin-item  freq_count  ;
        ...
}
```

pin-item

A *pin-item* is a pin name or pin group name defined on the *Pin Map* sheet.

Range

- Digital channel number from 0 to 2047.
- Range of digital channel numbers in parentheses:

(*channel-#* to *channel-#*)

- Any combination of pins and pin groups or any combination of channel numbers and ranges enclosed in parentheses and separated by commas:

(*item*, *item*, …, *item*)

Restrictions

- Channels and pins cannot be intermixed in the same pattern file.
- Pin groups in the **pin_setup** statement may be identical to pin groups in the **vector** statement pinlist, or supersets of them.
- Not allowed: using the same pin in multiple pin groups in the **pin_setup** statement.

io_midband and io_valid

With **io_midband** and **io_valid**, you specify either **drv_first** or **rcv_first**, indicating whether the drive or receive data is specified first in the pattern.

mux

With **mux,** specify *pin-item-2* as the pins to multiplex to the pins in *pin-item-1*. These pins are dummy pins defined on the *Pin Map* sheet. They are mapped to one of the multiplexed channels on the *Channel Map* sheet. The *Pin Map* sheet also defines a pin group with the two dummy pins; the group corresponds to the DUT pin. The **pin_setup** statement uses the dummy pins, not the pin group.

If more than one pin appears in the *pin-items*, both *pin-item-1* and *pin-item-2* must contain the same number of pins: the *n*th item in *pin-item-1* is paired with the *n*th item in *pin-item-2*.

If the paired pins do not satisfy the **mux** mode requirement that only adjacent even-odd channel pairs can be multiplexed, an error is generated. One pin must be mapped to an even channel and the other to the next highest odd channel. This pairing is checked at pattern load time, because the pin-to-channel mapping is not known at compile time.

The pins in *pin-item-1* appear in the vector pinlist, while the data is specified on the first line of each vector. The pins in *pin-item-2* do not appear in the pin list; their data is specified on the second line; refer to *Multiplex Mode* on page 6-48.

The order in the **pin_setup** statement affects only the order in which the data is specified in the two-line vectors of the pattern file. It does not affect the order of values applied to the DUT pin. The data for the pin mapped to the odd channel is applied before the data for the pin is mapped to the even channel. To map the order in the pattern file, make sure that *pin-item-1* contains pins mapped to odd channels and *pin-item-2* contains pins mapped to even channels.

Normal and Extended Modes

The items **io_midband**, **io_valid**, **mux**, and **freq_count** are used only in the extended mode. The pattern compiler issues an error message when a **pin_setup** statement in a normal mode pattern uses any of these items.

Only **high_voltage** is legal in normal mode.

Example:

```
pin_setup = {
ABUS      io_midband      drv_first;
(Q0, Q2)  mux             (Q1, Q3);
HVP       high_voltage;
FCP       freq_count;
}
```

Informal Syntax

+ The informal syntax notation emphasizes readability. For a formal representation of the syntax, refer to *Memory Syntax in Backus-Naur Form* on page 6-105.

pin_setup = {

<pin_setup_item>...

}

where:

<pin_setup_item>::= *<pingroup>* *<mode>* [*<parameter>*] **;**

<pingroup>::=	(*<field_list>*)	|
	<pin_chan_spec>	
<pin_chan_spec>::=		
	<pin_name>	|
	<pingroup_name>	|
	<digital_channel>	|
	(*<digital_channel>* to *<digital_channel>*)	
<field_list>::=	*<pin_chan_spec>*[,*<pin_chan_spec>*] ...	
<pingroup_name>	pin group name defined in the pinmap.	
<pin_name>	pin name defined in the Pin Map file.	
<digital_channel>	number for a digital channel, from 0 to 2047.	
<mode>	**io_midband**, **io_valid**, **high_voltage**, **freq_count** or **mux**.	
<parameter>	used only with **io_midband**, **io_valid** and **mux** modes. For **io_midband** and **io_valid**, it is either **drv_first** or **rcv_first**. For **mux**, it is a second <*pingroup*>.	

Pattern Language 6

Vector Statement

Requirements

The **vector** statement forms the body of a pattern file. It begins with the keyword **vector** followed by a pin list and the vector data for the pins. A pattern file must have only one **vector** statement.

Syntax

vector (*pin-list*)

{*vector-data*}

The *pinlist* assigns pins to the vector data that follows the pin list. A tset (timing set) value may be part of the pin list; it may also be part of the vector data.

Recommendations

Up to 4M vectors can be specified in a single vector statement, but a file this large takes a long time to compile and is difficult to debug; thus, Teradyne recommends files smaller than 4M.

Vector Data

The *vector-data* specifies the pattern microcode and channel data associated with each pin or pin group in the vector pin list.

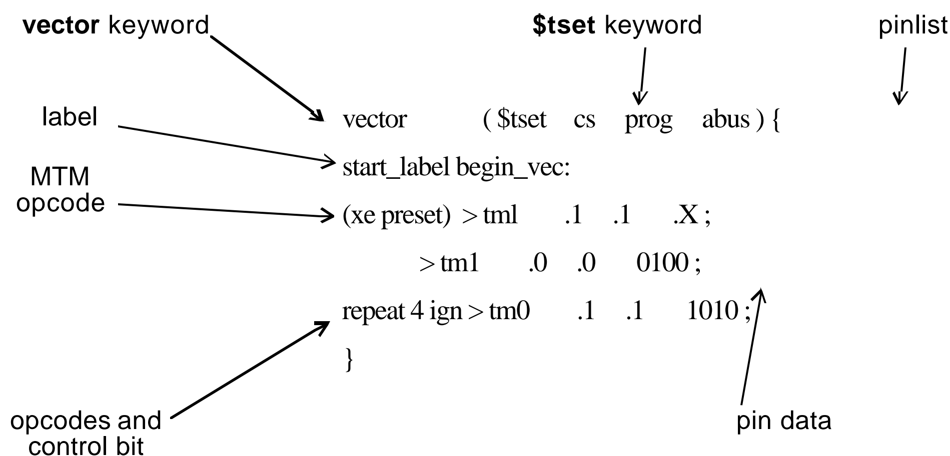
All *vector-data* for a pin list is enclosed within curly braces. Each individual vector is terminated with a semicolon. A vector may span multiple lines; end of line is not significant.

A single vector in the *vector-data* portion of the **vector** statement has the following fields:

- label (optional)
- MTM (memory) opcodes and arguments enclosed in parenthesis (optional)
- opcode and/or control bits (optional)
- *greater than* (>) delimiter (required)
- timing set (tset) field (optional)
- actual vector data fields (required)
- terminating semicolon (**;**) (required)

The *vector-data* portion of the **vector** statement can contain one or more vectors with this format as shown in the *Sample Vector Statement* on page 6-31.

Sample Vector Statement



Pin List in Vector Statement

The pin list assigns pins to the vector data that follow the pin list. Note that channel programming is supported; refer to *Channel Programming* on page 6-33.

Syntax

([$tset,] *pin-item* [:*radix*] [, *pin-item*[:*radix*]]…)

$tset

$tset is a keyword that designates the column for the **tset** number or name. The **$tset** keyword may be anywhere in the pin list, not only as the first item. If omitted from the pin list, a **tset** is specified for each vector in the statement; refer to *tset (Timing Set) Data* on page 6-39.

pin-item

Definition of *pin-item*:

- pin name or pin group name defined in the *Pin Map* file
- digital channel number from 0 to 2047
- range of digital channel numbers in parentheses:

(*channel-#* to *channel-#*)

- any combination of pins and pin groups, or any combination of channel numbers and ranges, enclosed in parentheses and separated by commas:

(*item*, *item*, …, *item*)

You can group pins within the pin list by enclosing them in parentheses. By grouping a number of pins this way, the pin data can be specified in a single column for all of the pins in the group. Individual items within such a group must be separated by commas.

Channels and pins cannot be intermixed in the same pattern file; refer to *Channel Programming* on page 6-33.

Radix

Each pin or pin group has a radix associated with it. The radix may be assigned in the pin list; if it is not specified, the default is symbolic. Vector data for a pin or pin group must use the associated radix, although the a numeric pin radix can be overwritten by the symbolic radix.

Radix is one of the following:

X or **H**	= hexadecimal
Q or **O**	= octal
D	= decimal
B	= binary
S	= symbolic (default)

See also:

Symbolic and Numeric Pin Vector Data on page 6-41.

Commas and White Space

Commas are optional between items in the pin list. If commas are used, a comma must not appear between the last item and the closing right parenthesis.

White space is allowed anywhere in a vector pinlist, including around the colon separating a pin group from a radix.

Sample pin list with white space:

```
vector ( $tset, (p4, p68), p13, abus:X, (p22, p23, p24):O )

        { vector-data }
```

Channel Programming

FLASH 750 uses channel programming: vector data is programmed by using pin and pin group names. By using names, the pattern data can be scrambled at load time. The name-to-channel mapping is resolved based on the test program's channel map. Pattern data is not scrambled if channels are used in the vector pinlist. A sample application of channel programming is to create multi-site loop patterns that explicitly program all channels.

You cannot mix channels and pins in the same pattern file.

Vector Labels

Format

- Label name must consist only of letters, digits and the underscore character (_) .
- Label name must start with a letter.
- Labels are not case-sensitive.
- Label must be followed immediately by a colon, with no intervening white space.
- Multiple labels on a given vector are allowed.
- In the ASCII source file, vectors may be on the same line as the vector data or on a different line. End-of-line is the same as white space.

Placement

Labels may be on the first vector of each vector pair in normal mode, or on any vector for patterns in extended mode; refer to *Pattern Modes* on page 6-16.

Kinds of Labels

start_label *label-name* : Start or modify a pattern. Start labels cannot be jumped to or called as subroutines; they are placed in the LVM by the pattern compiler, if possible. Start labels are exported so the test program can access them.

A label on the first vector of a pattern is not required. Pattern start and pattern modify default to the first vector of a pattern file if no label is specified.

subr *subroutine-name:* Called as a subroutine from within this pattern file; refer to *Pattern Subroutines* on page 6-37.

global *label-name:* Referenced by other patterns. A global label may be referenced by the test program or other patterns.

label-name: Referenced only within the same pattern file and not a subroutine. Local label has no keyword prefix, just the label name followed by a colon. A local label cannot be referenced by the test program or other patterns.

+ Any external labels (labels in other pattern files) must be imported before they can be used as an opcode operand.

global *subr subroutine-name:*
Subroutine is accessible to other patterns or the test program.

Labels and SVM

Using a label other than a start label forces a vector into the SVM, because it may be the target of a **jump** opcode. Thus, you should use labels only when necessary. Do not use them to improve the readability of a pattern; refer to *Minimizing Labels* on page 6-18.

Relative Addressing Instead of a Label

Any opcode using a label as an operand and in a file compiled for SVM-only can use a relative address instead of a label.

Restrictions:

- File must be compiled for SVM-only. This option is available as a compiler control statement and as a compile-time option.
- Address must resolve to a vector within the compiled file.
- For normal mode patterns, the address must resolve to a vector that is first in a vector pair, since the second vector in a vector pair cannot be executed without also executing the first vector.

Relative addressing format:

.	(Period) Refers to the current vector.
.+*N*	Moves execution down *N* vectors from the current vector. *N* is decimal.
.-*N*	Moves execution up *N* vectors from the current vector. *N* is decimal.

When using .+*N* or .-*N*, no white space is allowed after the period or before the *N*.

Pattern Subroutines

Overview

A subroutine in a pattern file is a set of vectors introduced by a label declared as a **subr** or **global subr**.

For information about calling and returning from a subroutine, refer to *Subroutine Opcodes* on page 6-61.

Rules for Subroutines

- Subroutine labels must be declared explicitly as subroutines by using the **subr** or **globalsubr** label prefix.
- If subroutines and non-subroutine vectors are in the same pattern file, the subroutines must be at the end of the file, after all non-subroutine vectors.

Once the pattern compiler encounters a **subr** label, the rest of the file is considered a subroutine; thus, subroutines must be at the end of a pattern file or in a separate pattern file.

- If a called subroutine is in another pattern file, the called subroutine label must be imported first. A subroutine in the same file as the calling vector does not have to be imported.
- A subroutine can call another subroutine only if the called subroutine is in a file compiled only with the SVM by using the SVM Only compile-time option or the following compiler statement:

  svm_only_file = yes;

If subroutines in one file call subroutines in another file, the pattern compiler verifies that subroutines call only subroutines declared **import svm_subr**. The pattern linker verifies that references declared **import svm_subr** are SVM-only files.

For the maximum depth of the subroutine stack, refer to *System Limits and Values* on page 6-112.

Pattern Data

Overview

The FLASH 750 pattern data may consist of the following elements:

- *tset (Timing Set) Data* on page 6-39
- *Symbolic and Numeric Pin Vector Data* on page 6-41
- *Numeric Vector Programming* on page 6-44
- *High-Voltage Data* on page 6-49
- *Frequency Counter Data* on page 6-50

tset (Timing Set) Data

Overview

The tester hardware supports 256 timing sets, called **tsets.** The program workbook defines up to 255 tsets on a single *Time Sets* sheet, while the remaining **tset** (**tset** 0) is reserved for run-time repeat of the **tset** number from the previously executed vector.

A given pattern may use no more than 255 **tset** names; otherwise, the pattern compiler issues an error message.

The **tsets** are programmed either by name or number:

- **tset** names, which are defined in the program workbook, must be imported by an **importtset** statement before they can be used in the vector statement. These names are resolved when the pattern is loaded. Names are not case-sensitive.
- **tset** numbers specify specific time set hardware locations.
- A single pattern cannot contain both **tset** numbers and **tset** names. Teradyne recommends that you use **tset** names.

Specifying tsets

You can specify a **tset** on each vector in a pattern, including each vector of a vector pair, by using one of the following methods:

- Define a **$tset** column in the **vector** statement pin list. Each column must have an entry for every vector: either a **tset** number or name, or a dash for runtime repeat; refer to *Runtime Repeat* on page 6-40.
- Specify a **tset** on selected vectors, to the left of the *greater than* sign (>), by using the keyword **$tset name** or **$tset number**. Teradyne recommends this method when a pattern file has only one timing set.

Pattern Language 6

Runtime Repeat

The runtime repeat **tset** cannot be used on the first vector executed in a pattern burst, because no **tset** exists to repeat. The pattern compiler cannot enforce this restriction, because it does not know where a pattern will start.

To specify runtime repeat of a **tset**, use a dash (-) instead of a **tset** name or number. The **tset** from the previously executed vector will be used.

If no **tset** is specified for a vector, the pattern compiler will use the **tset** runtime repeat for that vector.

Examples

The following example defines a **$tset** column. A dash in a column specifies runtime repeat.

```
import tset tm1, tm2;
vector      $tset   abus:X   bbus:X   clk){
>           tm2     .d0      .X       1;
>           tm1     .dFFF    .rFFF    0;
>           –       .dFFF    .rAAA    1;
halt >      –       .d0      .rFFF    0;
}
```

The following example specifies a **tset** for a specific vector:

```
import tset tm0;
vector      (       abus:X   bbus:X   clk) {
$tset tm0   >       .d0      .X       1;
            >       .dFFF    .rFFF    0;
            >       .dFFF    .rFFF    1;
halt        >       .dFFF    .rFFF    0;
}
```

Modifying at Runtime

You can read and modify the **tsets** in a loaded pattern file at runtime from an executing test program; refer to *Modifying Patterns at Runtime* on page 6-94.

See also:

Vector Statement on page 6-30

Symbolic and Numeric Pin Vector Data

Overview

The pin data field defines the vector data applied to each column in the pin list. It is the major component of the **vector** statement.

Types of Pin Data

Vector data is either symbolic or numeric:

- Symbolic vector programming uses a set of predefined characters; refer to *Symbolic Vector Programming* on page 6-42.
- Numeric vector programming uses binary, octal, hexadecimal, or decimal digits, plus prefixes; refer to *Numeric Vector Programming* on page 6-44.

The **vector** statement pin list specifies the radix for each column in the vector pin list, the default radix is symbolic.

See also:

Vector Statement on page 6-30

Symbolic Vector Programming

Symbolic Character Set

The pattern compiler accepts the following characters for symbolic datacodes:

0	Drive low
1	Drive high
2	Drive very high (refer to *High-Voltage Data* on page 6-49)
L	Expect low
H	Expect high
M	Expect midband
V	Expect valid (high or low)
X	Expect mask
D	Drive data from Address Data Generator (ADG)
E	Expect data from ADG
-	Runtime repeat (hyphen)

Symbolic data characters are not case-sensitive.

Note that **0** and **X** have special meanings for frequency counter data.

The datacodes **D** and **E** indicate that the ADG is the data source; refer to MTM programming.

Symbolic Radix

If no radix is specified in the **vector** statement pin list, the default radix is symbolic.

If a numeric radix is specified, you can use symbolic radix for a given vector and pin group by prefixing the symbolic data with **.s**. A symbolic radix for a pin or pin group cannot be overridden by numeric, however.

Symbolic Data for Pin Groups

You can specify a symbolic value for each pin in a pin group: one symbolic datacode for each pin.

To set an entire pin group to the same symbolic value, use a . (dot) before a symbolic datacode, which replicates the symbolic data through the current pin group. For example, for a group with five pins, these forms are equivalent:

XXXXX .X

Io_midband *and* io_valid *Pins*

Only certain characters are valid in the SCIO (Single Cycle I/O) modes:

Drive For both **io_midband** and **io_valid** pins, legal drive data characters are 0 and 1.

Receive For **io_midband** pins, legal receive data characters are H, L, M, and X.

For **io_valid** pins, legal receive data characters are **H**, **L**, **V**, and **X**.

See also:

Symbolic and Numeric Pin Vector Data on page 6-41

Numeric Vector Programming on page 6-44

High-Voltage Data on page 6-49

Frequency Counter Data on page 6-50

Numeric Vector Programming

Numeric Values

In numeric vector programming, values are expressed as either high (1) or low (0) per pin. A prefix, **.d** or **.r**, identifies drive data or receive data per pin group. The prefix is required on each numeric data item.

Radix

Data radix are specified in a **vector** statement pinlist by using the appropriate values and one of the following prefixes:

X or **H**	hexadecimal
Q or **O**	octal
D	decimal
B	binary

A number of bits equal to the maximum number of channels in the system may be expressed in all radices except decimal, which is limited to 32 bits.

If no radix is specified, the default is symbolic data, and that numeric data is not used for that pin or pin group. However, if an item in a column has the **.d** or **.r** prefix, the item is assumed to be in hex, and is interpreted as a hex number.

LSB and MSB

When using numeric data to specify pin groups, the LSB is assigned to the rightmost (last) pin in the pin group. If the number of pins exceeds the number of data bits, low (0) is assumed for the MSBs and is programmed into the first pins in the group.

In the example below, assume pin group *ABUS* is defined and ordered as pins P0, P1, P2, P3, P4 and P5 and programmed with a value of **.r**A. Pins P0 and P1 have a value of *0* (L), which is the default for the MSBs when not enough bits are specified. Pins P2 and P4 have a value of *1* (H), and pins P3 and P5 will get a value of *0* (L):

.rA = 1010		=	1	0	1	0
			⇓⇓	⇓⇓	⇓⇓	⇓⇓
	L	L	H	L	H	L
	P0	P1	P2	P3		
	P4	P5				

See also:

Symbolic and Numeric Pin Vector Data on page 6-41

Numeric Vector Programming on page 6-44

Vector Data in SCIO and Mux Modes

SCIO (Single Cycle I/O) Mode

In the **SCIO** mode, drive and receive data are programmed for the same pin or pin group. The **pin_setup** statement determines whether drive or receive data is specified first. In writing the vector data, the second SCIO data item is specified after all other data items. The second SCIO data item can be on the same line as the rest of the vector data, but Teradyne recommends that you use two lines.

Assume that P1 is an SCIO pin, with drive data specified first. You could show the data on one line, where the final data item represents the receive data for P1:

```
.       P0      P1      P2
.       1       0       H             L
```

You could also show the data on two lines, with the receive data lined up under P1:

```
.       P0      P1      P2
.       1       0       H
                L
```

Required pin_setup Statement

A pin_setup statement is required for SCIO data, whether one or two lines are used for the vector data. It states that either the drive data or receive data is specified first.

A **pin_setup** statement is required for **mux** mode only if you use two lines for the data: the statement identifies the second pin muxed to the pin listed in the pin list. If you use only one line for the data and both pins are in the pin list, the **pin_setup** statement is not required.

See also:

Symbolic and Numeric Pin Vector Data on page 6-41

Example

Pin groups **iombus** and **iovbus** are sample SCIO pin groups. In the **pin_setup** statement, the drive data is specified first for both **iombus** and **iovbus**:

```
pin_setup = {
                iombus          io_midband      drv_first;
                iovbus          io_valid        drv_first;
}
...
vector

 ($tset      abus:X     iombus:X     bbus:X     iovbus:X     cbus:X ) {
   >         1          .dFFF        .dABC      .X           .dCBA
.X
                                     .rFFF                   .rABC
      ;
}
```

See also:

Symbolic data codes for *Io_midband and io_valid Pins* on page 6-43

Multiplex Mode

In the **mux** (multiplex) mode, two adjacent channels are connected: one channel provides the multiplexed signal for the other channel. The pin map defines two dummy pins as the muxed channels. In the pattern file, the **vector** statement provides the data to be driven onto the muxed dummy pins.

In writing the vector data, the **mux** mode pins are programmed like ordinary pins: data for each muxed pin is in a separate column. The pins are displayed separately by the pattern compiler and pattern tools display the pins separately, as if the **mux** mode were not used.

For readability, Teradyne recommends a single data column for muxed pins, with two lines per vector. Use the **pin_setup** statement to state which pins to connect together. In addition, the first pin in the statement is the pin in the vector pin list; the second pin in the statement does not appear in the pin list. For the order of the pins, refer to *Pin_setup Statement* on page 6-26.

Assume that pins P1 and P2 are muxed. The pins could be programmed like ordinary pins, on a single vector line:

```
.        P0       P1       P2             P3
.        H        1        0              L
```

A **pin_setup** statement can represent the muxed pins in a single column:

```
pin_setup {  P1  mux  P2  ;  }
.        P0       P1       P3
.        H        1        L
                  0
```

Placing Vector Data on a Single or Double Line

SCIO and **mux** mode data can be programmed on a single vector line, with the data for each pin lined up under the proper pin or pin group name from the pin list. For readability, Teradyne recommends that you program the **SCIO** and **mux** mode data on two adjacent vector lines. Because carriage returns are not significant, you can break a single vector line into two lines to list both the **SCIO** and **mux** mode data.

High-Voltage Data

Overview

Most of the symbolic vector programming codes for the high-voltage pins do not have their usual meaning.

Note that the IG-XL test program workbook timing sheet must also set up the high voltage pins.

High Voltage Pins

For the high-voltage pins, valid data characters are *0*, *1*, and *2*. High voltage pins are drive-only.

The *2* datacode specifies the high-voltage level available when programming a high-voltage pin. The other datacodes have their usual meaning.

Only board-relative channels 0, 4, 32 and 36 can be used as high-voltage; they may also be used as regular digital channels; consequently, you must know the pin-to-channel mapping when selecting a pin used in the high-voltage mode; refer to *Channel Programming* on page 6-33.

See also:

Symbolic and Numeric Pin Vector Data on page 6-41

Frequency Counter Data

Overview

Symbolic vector programming codes are used for the frequency counter pins.

+ The **IG-XL** test program workbook timing sheet must also set up the frequency counter pins.

See also:

Symbolic and Numeric Pin Vector Data on page 6-41

Frequency Counter Pins

For frequency counter pins, valid data characters are *0* and *X*. These datacodes do not have their normal meaning when used with these pins. Instead, the datacodes control the opening and closing of the frequency counter window.

A pair of datacodes over 2 cycles is necessary to open or close the window, because the frequency counter datacodes operate on the R0 or R1 edge of the cycle prior to the cycle on which they are programmed.

- Before opening a window, the datacode for that pin on all cycles should be *X*.
- To open the window, program a *0* on the cycle on which the R0 edge should open the window, and another *0* on the cycle after that cycle. The *0* on the second cycle opens the window on the R0 edge of the first cycle.
- Continue programming a *0* while the window is open.
- To close the frequency counter window, program an *X* on the cycle after the one on which the R1 edge should close the window.

Example

```
vector      ($tset,   FCP)     {
            > 1       X;
            > 1       0;       // R0 for this cycle opens window
            > 1       0;       // "0" opens window in previous cycle
repeat N    > 1       0;       // Window stays open
            > 1       0;       // R1 for this cycle closes window
            > 1       X;       // "X" closes window in previous cycle
}
```

Pattern Execution and Control

Opcodes

Overview

An opcode controls the execution flow within the pattern file, such as loop on this vector, or jump to a specific label.

+ Because of their complexity, the MTM (memory) opcodes generated by the ADG are described in a separate subsection; refer to *MTM (Memory) Opcodes* on page 6-65.

Requirements and Restrictions

- Location in **vector** statement

  Opcodes and control bits must be after any label and before the *greater than* (>) character that separates the opcode field from the channel data.

- Number per vector

  Only one opcode may be placed on a single vector. A single vector may have any number of control bits, which may also contain an opcode. Use white space or commas to separate the control bits and opcode.

- Pattern mode

  In the normal mode, an opcode and control bits can be only on the second vector in a vector pair, except for a mask control bit, which can be on any vector. In the extended mode, an opcode and control bits can be on any vector; refer to *Pattern Modes* on page 6-16.

  Memory patterns run only in the extended mode. In this mode, one vector produces one tester cycle. Thus, you must compile the pattern file for the extended more. Because the memory pattern runs in the extended mode, the MTM (memory) microcode can appear on every vector.

- SVM and LVM

  The opcodes and control bits in a pattern file determine whether a vector is placed in the LVM (large vector memory) or is forced into the smaller SVM (subroutine vector memory); refer to *Pattern Memory: LVM and SVM* on page 6-17.

  Most opcodes and control bits can be used only in the SVM. Using any opcode or control bit other than the listed in *Opcodes in SVM* on page 6-19 forces the vector or vector pair containing it into SVM rather than LVM.

  The opcode command resume is used only in patterns compiled SVM-only.

Examples

Vectors using opcodes and control bits:

```
if (flag) jump p1 ign stv clr_cond icc ifc mask clr_fail > 1  L M V H;

if (fail) call subr1,clr_cond > 2 L H .d0 .d44444 0 .r1F ;

enable (fail or cpuB) > 7  0 H H H;
```

See also:

Vector Statement on page 6-30

Control Bits on page 6-88

Categories

The opcodes and control bits can be grouped into the following categories:

- Loop opcodes, page6-53
- Repeat opcodes, page6-58
- Branch opcodes, page 6-60
- Subroutine opcodes, page6-61
- Miscellaneous opcodes, page6-64
- MTM opcodes, page6-65
- Control bits, page6-88

Loop Opcodes

Opcode	Operand	Definition
set_loopA	1...65536	Push number onto loop stack. **set_loopA** executes vectors in loop once.
set_loopB	1...65536	Set loop counter B to *number*. No loop stack for loop counter B.
set_loopC	1...65536	Set loop counter C to *number*. No loop stack for loop counter C.
loopA	1...65536	Similar to **set_loopA**, but pushes *number* onto the stack the first time only, not when branching back via **end_loopA** to vector with the loop instruction.
loopB	1...65536	Similar to **set_loopB,** but sets loop counter B the first time only, not when branching back via **end_loopB** to vector with the loop instruction.
loopC	1...65536	Similar to **set_loopC,** but sets loop counter C the first time only, not when branching back via **end_loopC** to vector with the loop instruction.
end_loopA	*label*	Decrement loop counter A. If not zero go to *label*, else pop loop stack. Used with **loopA** or **set_loopA**
end_loopB	*label*	Decrement loop counter B. If not zero go to *label*, else continue. Used with **loopB** or **set_loopB.**
end_loopC	*label*	Decrement loop counter C. If not zero go to *label*, else continue. Used with **loopC** or **set_loopC.**
exit_loop	*label*	Pop loop stack and go to *label*. Used for premature exit from an iterative loop.Can be conditional**.**
pop_loop	---	Pop the loop stack; used to clear the stack after premature exit from the loop.

Notes

- Loop opcodes can simulate a match loop, refer to *Creating a Match Loop* on page 6-57.
- For maximum depth of the loop stack, refer to *System Limits and Values* on page 6-112.

See also**:**

Opcodes on page 6-51

Control Bits on page 6-88

Loop structures A, B, and C

The pattern language has three loop structures, **loopA**, **loopB**, and **loopC**:

- **loopA** uses a loop stack.
- **loopB** and **loopC** use a loop counter, not a loop stack.

The purpose of the **loopA** stack is to permit nesting. **LoopA** opcodes can be nested to a depth of 4 loops. **LoopB** and **loopC** do not permit nesting looping. Examples of valid loop structure:

set_loopA 7	
L1:	L1: loopA 7
[vectors]	[vectors]
end_loopA L1	end_loopA L1

set_loopA and **loopA** both push the loop number onto the loop stack. The **end_loopA** opcode checks the number at the top of the stack:

- If it is greater than zero, it branches to the label.
- If the number is zero, it pops the number off the stack, and continues with the next vector.

set_loop and loop

set_loopA always pushes the number onto the loop stack. In contrast, the **loopA** opcode pushes the number onto the loop stack only when it is executed sequentially, not when its label is the target of an **end_loopA**. On its first execution, **loopA** pushes the number of the loop stack; on subsequent executions of the loop, it is equivalent to **nop**. Because **loopA** has this built-in flexibility, Teradyne recommends using **loopA** instead of **set_loopA**.

The **loopB** and **loopC** opcodes set the loop counter only when either is executed sequentially, usually on the first execution. If they are the target of an **end_loopB** or **end_loopC**, they are equivalent to a **nop**. Teradyne recommends using these opcodes instead of **set_loopB** and **set_loopC**, which always set the loop counter.

Incorrect Loops

Both **set_loopA** and **loopA**, if put on the wrong vector, can cause stack overflow. The following examples put these opcodes after the vector whose label is the target of the **end_loop**:

```
L1: [vector]                 L1: [vector]
set_loopA  7                 loopA  7
Note:
   [vectors]         [vectors]
These examples are invalid:
end_loopA  L1                end_loopA  L1
```

The opcodes are executed again on each execution of the loop. Because **loopA** is executed sequentially, it is not equivalent to **nop**. Each execution pushes the operand onto the loop stack, eventually resulting in a runtime stack overflow.

The **loopB** and **loopC** structures do not use a stack; thus, they cannot cause a stack overflow. If the examples above had used **set_loopB** instead of **set_loopA**, the result would have been an endless loop: each execution of **set_loopB** would have reset the loop counter to 7.

Nesting Loops

As shown in the following example, the **loopA** opcodes can be nested. It also shows how **loopC** can be nested with **loopB**, because there is no **loopB** or **loopC** stack; however, these opcodes cannot be nested within themselves.

```
L1: loopA  20                L1: loopB  20
   [vectors]         [vectors]
   L2: loopA  12     L2: loopC  12
      [vectors]         [vectors]
   end_loopA  L2     endloopC  L2
end_loopA  L1                endloopB  L1
```

The inner loop is executed 12 times for each execution of the outer loop. Note that the **loopA** loops could be nested two more times, for a loop depth of 4.

Exiting the Loop with pop_loop and exit_loop

These opcodes prematurely exit a loop. Usually, an **end_loop** decrements the loop counter; if the counter is then 0, the loop is exited. Examples:

```
L1: loopA 6                        L1: loopA 7
    [vectors]            [vectors]
    if(fail) jump L2               if(fail) exit_loop L2
    [vectors]            [vectors]
end_loopA L1                       end_loopA L1
    [vectors]        L2: [vectors]
    halt
L2: pop_loop
    [vectors]
```

If you jump out of a **loopA** loop, which uses the loop stack, the loop counter is still on the loop stack; thus, you must use **pop_loop** to pop the counter off the loop stack. You must not execute both **end_loop** and **pop_loop**, which is the reason for the **halt** opcode before the L2 vector. Executing both can cause a stack underflow.

If you use **exit_loop**, instead of **jump**, **exit_loop** pops the counter off the loop stack.

Teradyne strongly recommends that you use **exit_loop** instead of **jump** and **pop_loop**.

Modifying Loop Microcode

The operand of a **loop** or **set_loop** opcode can be modified at runtime. In this way, a program can dynamically specify the number of times to execute a loop; refer to *Modifying Patterns at Runtime* on page 6-94.

Creating a Match Loop

The IG-XL pattern language does not contain a match opcode; however, you can simulate a match opcode by using **loop, if**, **jump** and **end_loop**. An advantage of this approach over the standard match opcode is that you can understand why the loop was exited, based on the label at which execution continues.

The following example is a 6-vector match sequence, with **repeat** forcing a failure on going into the loop:

```
        repeat 30, ign                              // force fail
        set_loopA 100, ign                          // vector 1
        ign                                         // vector 2
m_loop:
        ign                                         // vector 3
        ign                                         // vector 4
        ign                                         // vector 5
        if (pass) jump cont, ign                    // vector 6
        if (fail) jump l1, ign, clr_fail, clr_cond  // vector 1
l1:
        end_loopA m_loop, ign                       // vector 2
timeout:
        halt                                        // if you get here, loop timed out
cont:                                               // if you get here, you matched
        pop_loop                                    // since you jumped out of loop,
                                                    // pop the loop stack
```

The **clr_fail** control bit clears the per-channel fail count and the per-channel accumulated fail register (AFR). Because failures are expected during the match loop, you do not want these failures to be included in the fail count or the AFR. Consequently, you include **clr_fail** to clear the per-channel fail count and AFR. You want to count only the failures after exiting from the match loop.

Pattern Language 6

Repeat Opcodes

Opcode	Operand	Definition
repeat	2...65536	Execute vector specified number of times. Repeats both vectors in a vector pair in patterns compiled for normal mode. The operand of **repeat**, unlike **mrepeat**, cannot be modified at runtime; refer to *mrepeat and repeat* on page 6-59. Can be used in LVM.
mrepeat	2...65536	Execute vector specified number of times. Repeats both vectors in a vector pair in patterns compiled for normal mode. This opcode, unlike **repeat**, allows the operand to be modified at runtime by a function call. Refer to *mrepeat and repeat* on page 6-59. Can be used in LVM.
pipe_minus	0…*n*	Similar to **repeat,** but repeats the vector or vector pair for the depth of the fail pipeline minus the specified number. The upper limit on the operand *n*: *pipeline-depth* - 1 On the pipeline depth, refer to *System Limits and Values* on page 6-112. The operand cannot be modified at runtime.

See also:

Opcodes on page 6-51

Control Bits on page 6-88

mrepeat and repeat

The opcodes **repeat** *n* and **mrepeat** *n* both execute the vector *n* number of times.

The difference is that **mrepeat** can be modified at runtime. A driver function can modify the *n* operand when the number of required repetitions depends on circumstances known only at runtime; refer to *Modifying Patterns at Runtime* on page 6-94.

Use the **repeat** opcode when the value of the opcode does not have to be modified after the pattern has been loaded. If the repeat value must be modified after the pattern has been loaded, use the **mrepeat** opcode.

In some cases, using **repeat** instead of **mrepeat** reduces use of the SVM, thereby optimizing LVM/SVM performance. The pattern compiler counts the number of cycles to be executed to determine how many vectors must be put in the SVM after setting the LVM address but before transferring control to the LVM. The pattern compiler assumes **repeat** executes the specified number of cycles; thus, it uses this full number of cycles when compensating for the delay. On the other hand, it must assume that **mrepeat** may only execute a single cycle. Since **mrepeat** can be modified at runtime, the pattern compiler must count the **mrepeat** vector as a single cycle. It also may need to put additional vectors into the SVM to compensate for the delay.

Teradyne recommends that you should use **repeat**, which optimizes the LVM/SVM. You should use **mrepeat** only when runtime modification is necessary.

See also:

Managing SVM Usage on page 6-18

pipe_minus

The **pipe_minus** *n* opcode is also available. It repeats the vector or vector pair for the depth of the pipeline, minus the specified number; however, it cannot be modified. It supports code portability between systems with different pipeline depths. To compensate for the pipeline depth by repeating a vector, refer to opcode *pipe_minus* on page 6-59.

See also:

Repeat Opcodes on page 6-58

Loop Code and Control Bit *Categories* on page 6-52

Branch Opcodes

Opcode	Operand	Definition
jump	*label*	Go to label. Can be conditional. Can be used in LVM when used without an **if**.
jmp_glo	---	Jump to the address in the global register. Can be conditional.
keep_alive		Start keep-alive pattern.
enable	*flags*	Enable the specified condition flags for later use in an **if** (*flag*) statement; refer to *Conditional Statements* on page 6-91.
halt		Halts the pattern burst. Can be used in LVM.
end_module		Acts as a **halt** when running a single pattern and as a **nop** when running a group, which supports flow to the next module when executing a group. Can be used in LVM. Refer to *Using end_module instead of halt* on page 6-60.

See also:

Opcodes on page 6-51

Control Bits on page 6-88

Using end_module instead of halt

If a pattern file is to run either alone or a part of a pattern group, you should use **end_module** at the end of the pattern instead of **halt**. The effect of **end_module** depends on whether the pattern in which it occurs is running alone or as part of a group:

- When running in a single pattern, **end_module** is set to **halt**.
- When running in a group, **end_module** is set to **nop**.

In this way, **end_module** allows execution of one pattern in a group to flow to the next pattern, while still halting execution is the pattern is run by itself.

Last Pattern in Group

The last pattern in a group must end with halt, not **end_module**. Even in the last pattern, **end_module** is treated as a **nop**, and execution will flow beyond the end of the group. To avoid this problem, Teradyne recommends that you include a terminator pattern at the end of each group, consisting of a single vector with **halt**.

Subroutine Opcodes

Opcode	Operand	Definition
push	*label*	Push address of label onto subroutine stack.
pop	---	Pop vector address off top of subroutine stack. Do not change current vector address.
call	*label*	Execute subroutine at label. Push current vector address; go to label. Can be conditional. Can be used in LVM when used without an **if**.
ccall	*label*	Acts as either a **nop** (default) or **call**, depending on the value of a register programmed from the test program; refer to *ccall and call* on page 6-62. Can be used in LVM.
return	---	Return from subroutine. Jump to top of stack; pop stack. Can be conditional.
resume		Swap PC with top of stack. Passes argument vectors to subroutines; refer to *resume* on page 6-62. Note that arguments to subroutines must always be called for proper execution. Can be conditional. Note that arguments to subroutines must always be called for proper execution. This opcode is allowed only in pattern files compiled SVM-only.
call_glo		Call the address in the global register like a subroutine. Use **return** to return. Can be conditional. Note that the **call_glo** opcode is illegal within a subroutine.

See also:

Opcodes on page 6-51

Control Bits on page 6-88

ccall and call

call and **ccall** both execute the subroutine at the label operand.

ccall (conditional call) — Execution of **ccall** is conditioned from the test program by using the **Ccall** property of the Patgen driver object. By setting the driver property of **ccall** to **nop** or to **call**, you can control the subroutine execution. In addition, the Pattern Control display lets you toggle between *Ccall is Nop* and *Ccall is Call*. All **ccalls** in a pattern are set to the same value; they cannot be set individually.

call — Always calls the subroutine.

resume

The **resume** opcode swaps the program counter with the top of the stack.

Subroutine arguments must always be called for proper execution. **resume** assumes the vector arguments for a subroutine follow the subroutine call. The **resume** opcode switches execution back to the calling code to get the arguments, then another **resume** switches back to the subroutine code. In the following example, the code on the left side is the calling code, and the code on the right is the subroutine; numbers in parentheses indicate the order of execution; ignore blank lines and assume the lines in the calling code and the subroutine appear on sequential lines:

```
start_label foo: (1)
call subr1 (2)
                                        subr
subr1: (3)
[vectors1] (4)
                                        resume (5)
   [arg1] (6)
   [arg2] (7)
resume (8)
[vectors2] (9)
                                        return (10)
[vectors3] (11)
```

When the subroutine is called, the address of [arg1] is pushed onto the stack, the program counter (PC) is set to subr1, and execution begins in the subroutine. At the execution of resume (5) in the subroutine, the top of the stack and the PC are swapped, so [vectors2] is on the top of the subroutine stack, and execution resumes at [arg1] in the calling code. The two arguments are read, and the next resume (8) causes another swap; execution resumes in the subroutine at [vectors2]:, [vectors3] is now on top of the stack. At the return (10) in the subroutine, [vectors3] is popped off the stack and execution continues in the calling code from there.

+ Use **resume** only in pattern files compiled SVM-only.

return

This opcode assumes execution in the subroutine is completed. In contrast, **resume** causes execution alternate between the subroutine and the calling code. In a subroutine, execution is assumed to resume after be only temporarily suspended.

return and **resume** differ in how they use the stack. **return** pops the top address off the stack and does not push anything onto it. **resume** pops the top address off while pushing the next address onto it, so execution can later be resumed at that point.

Miscellaneous Opcodes

Opcode	Operand	Definition
set_glo	*label*	Put the address of the label in the global register.
clr_code	---	Clear the readback code. When readcode is set, it remains set until this command is used or a new pattern is started; refer to *set_code and clr_code* on page 6-64.
set_code	0...2047	Set readback code for test program to read; refer to *set_code and clr_code* on page 6-64.
set_cpu	*cpu flags*	Sets the specified condition flags, which must be one or more of **cpuA**, **cpuB**, **cpuC**, and **cpuD**. Unmentioned flags are not affected by this opcode.
clr_flag	*flags*	Clears the specified condition flags.

See also:

Opcodes on page 6-51

Control Bits on page 6-88

set_code and clr_code

These opcodes and functions allows the pattern burst to set a status value that the test program can read.

set_code sets a numeric value that the test program can read. The **PatGen** object has a **ReadCode** property that can read the value; it has a **ClearReadCode** method that can clear the code. Refer to the description of the **PatGen** object.

clr_code can also clear the code from the pattern file.

Modifying the Code

The operand of the **set_code** opcode can be modified at runtime; refer to *Modifying Patterns at Runtime* on page 6-94.

MTM (Memory) Opcodes

Overview

Every MTM opcode is programmed on every vector. If you do not explicitly include an MTM opcode on a vector, a default is used. For example, if a vector does not specify the address counter for the X and Y address, the default for the vector is **xa** and **ya**, even if the previous vector specified different, non-default counters.

The order of the opcodes appearing on the vector is not significant. All opcodes are processed before the vector is executed. Their results are available simultaneously within the same vector except for the opcodes for incrementing or decrementing the address counters: see the description of the order of incrementing or decrementing, the discussion of the *Address Counter Opcodes* on page 6-70.

They are grouped by the following categories:

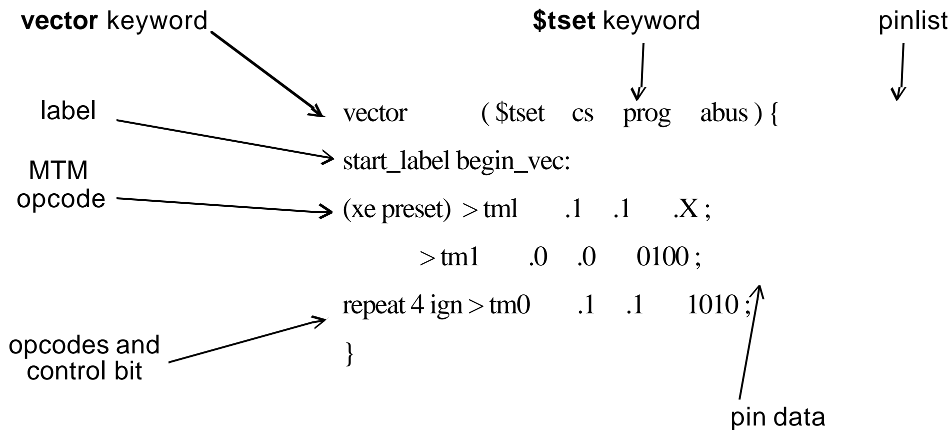
- *Address Counter Opcodes* on page 6-70
- *ALU Operation Opcodes* on page 6-76
- *Data Generator Opcodes* on page 6-81
- (Format) *Frame Select Opcodes* on page 6-84
- *Serial Device Opcodes* on page 6-85
- *DBM/ECR Opcodes* on page 6-85
- *DBM/ECR Opcodes* on page 6-85
- *Device Power Supply (DPS) Opcodes* on page 6-87

See also:

Summary of Memory Opcodes on page 6-66

Syntax

As shown in the figure below, the MTM microcode is enclosed within parentheses.



Summary of Memory Opcodes

The opcode syntax listed in the following tables emphasizes readability. For a formal representation of the opcode syntax in the Backus-Naur form, refer to *Memory Syntax in Backus-Naur Form* on page 6-105.

Address Counters

Opcode	Values	Default	Description
xa, **xb**, **xc**, **xd**, **ya**, **yb**, **yc**, **yd**, **za**, **zb**	**hold**, **inc**, **inc_link**, **dec**, **dec_link**, **preset**, **load_alu**	**hold**	Control an X, Y, or Z address counter.
xdevadr **ydevadr** **zdevadr**	**xa**, **xb**, **xc**, **xd**, **ya**, **yb**, **yc**, **yd** **za**, **zb**	**xa** **ya** **za**	X, Y, Z counter addresses the DUT.
load_xenable **load_yenable** **load_zenable**	None	None	Change the address value of address enable registers.
xdgadr **ydgadr**	**xa, xb, xc, xd** **ya, yb, yc, yd**	None	On-the-fly X/Y address generators output to DTOPO RAM mux and 2-bit Data Generator

ALU Operations

Opcode	Values	Default	Description
xalu= **yalu=** **zalu=**	*expression*	**0**	Arithmetic or logical operation
alu_constant=	*number*	**0**	Constant for ALU address operation

Data Generator

Opcode	Values	Default	Description
dset	0 to 7	None	Select a data set for 2-Bit Data Generator
dgroup	0, 1, 2, 3, -	None	Select a data group for 2-Bit Data Generator
dg_sel	**dga**, **dgb**	**dga**	Select output from one of two data registers of the 32-Bit Data Generator.
dga, **dgb**	**hold**, **inc**, **dec**, **shr**, **shl**, **rol**, **ror**, **ldlo**, **ldhi**	**hold**	Controls the register of the 32-Bit Data Generator
alu_constant=	*expression*	**0**	ALU constant for data operation

Frame Select

Opcode	Values	Default	Description
frame	*number*	None	Select formats for serial multiplexed device.

DBM/ECR

Opcode	Values	Default	Description
dbm_acc_zero	None	None	DBM accumulate zeros mode
wr_dbm	None	None	Data generator as DGM data source
dbm_enable	None	None	Enables masking of DBM output.
learn_dut	None	None	DBM learn DUT mode
capture_errs	None	None	Enables capturing of errors to ECR.
clr_ecrcount	None	None	Clears the ECR counters on the fly
ecr_boe0, **ecr_boe1**, **ecr_boe2**, **ecr_boe3**	**blc**, **wlc**, **rec**, **cec**, **tec**	None	Selects an ECR branch-on-error condition
latch_errs	None	None	Latch ECR failures
clr_maxcond	None	None	Clear ECR failure latches

DPS

Opcode	Values	Default	Description
dps0, **dps1** **dps2**, **dps3**	**hold**, **inc**, **dec**, **reset**	**hold**	Controls the VRATCHET for a individual device power supply
iddstart	None	None	Enables IDDQ/IPEAK current measurements for individual device power supply
iddstop	None	None	Disables IDDQ/IPEAK current measurements for individual device power supply
iddsync	None	None	Measure PEAK current for individual device power supply

Pin Level

Opcode	Values	Default	Description
vihhen	0, 1, 2, 3	None	Enables VIHH pin level control

Serial Device

Opcode	Values	Default	Description
sr_reg	**load**, **hold**, **shift**	**hold**	Supports serial device I/O

Default Field Values

xa hold	**xb hold**	**xc hold**	**xd hold**
ya hold	**yb hold**	**yc hold**	**yd hold**
za hold	**zb hold**		
xalu = 0	**yalu = 0**	**zalu = 0**	
xdevadr xa	**ydevadr ya**	**zdevadr za**	
ds_xsel xa	**ds_ysel ya**		
dgroup 0 dset -			
dga hold	**dgb hold**		
dg_sel dga			
frame 0			
dps0 hold	**dps1 hold**	**dps2 hold**	**dps3 hold**
sr_reg hold			
ecr_boe0	**alu_const 0**		
alu_constant 0			

How Pattern Data is Used: Memory Datacodes (D, E, X)

Data for an memory pattern comes from the Address Data Generator (ADG). In the pattern file, the following symbolic datacodes indicate how the data is used:

D	Drive data
E	Expect data
X	Expect mask

D means that the data to be driven on this pin is taken from the ADG.

Storing Opcodes in SVM

Using an MTM opcode causes the vector to be put into the SVM. The LVM does not support MTM opcodes.

Storing Opcodes in User Registers

Many MTM opcodes are stored in registers that can be set by the user. For example, **xaload_preset** loads a value from a preset register into the address counter **xa**; another register determines whether a 0 or 1 is shifted into a counter on an ALU shift operation.

The registers are set outside the pattern file and retain their value throughout the pattern execution. The registers can be set on the **DataTool** sheets for MTM, on the MTM test template, or through the **Visual Basic MTM** object. They can also be viewed and changed during execution using the **Debug Display**, known as the **IG-XL Display Manager**.

ALU Constants

Use the following syntax to input a 16-bit constant to the ALU:

alu_constant *number*

where *number* is decimal or hexadecimal (entered in the form 0xffff).

The ALU constant can be used in all ALU expressions, both address and data. It can also be used in the **dg_sel** expression for the 32-Bit Data Generator. The keyword **const** in the expression indicates the ALU constant.

The value of the constant is available in the same vector in which it is assigned. It is set in every vector; however, if a vector does not have an explicit assignment, the following default is used:

alu_constant 0

Address Counter Opcodes

Overview

There are 4 X address counters, 4 Y address counters, and 2 Z address counters:

- X: **xa**, **xb**, **xc**, **xd**
- Y: **ya**, **yb**, **yc**, **yd**
- Z: **za** and **zb**

The X, Y, and Z address registers are also known as **REGA**, **REGB**, and **REGC**, respectively.

The following counter operations are supported by these 16-bit counters:

- Specifying the counters used as the X, Y, and Z addresses of the DUT; refer to page6-74.
- Setting a counter address; refer to page6-72.
- Linking counters; refer to page6-73.

+ The Z address registers cannot be linked with its counterpart in X or Y. Also, the Z address generator does not output the 2-bit data generator address; refer to xxx.

- Specifying which bits will change when a counter is incremented or decremented; refer to page6-74.

Specifying the Counters for the DUT Addresses

Any of the four address counters can be selected on-the-fly as the address to the device: **xdevadr**, **ydevadr**, and **zdevadr**. These addresses to the devices (16 bits each) are sent to the Logical Address Crossbar which maps the logical X, Y, and Z addresses to crossed logical X and Y address inputs (16 bits each). The syntax to select one of the four counters as the address to the device:

xdevadr *x-counter*

ydevadr *y-counter*

zdevadr *z-counter*

where

x-counter is **xa**, **xb**, **xc**, or **xd**

y-counter is **ya**, **yb**, **yc**, or **yd**

z-counter is **za** or **zb**

Defaults:

- **xdevadr xa**
- **ydevadr ya**
- **zdevadr za**

If the address counter is not specified, then the defaults are used for that cycle.

Setting an Address Counter

Syntax

counter *state [= value]*

where:
value is an initial condition;

state is one of the following

hold	Retains the current value (default).
inc	Increments the counter.
dec	Decrements the counter.
inc_link	Increments this counter (X or Y only) half only if the other half of the counter generates a carry or borrow (overflow or underflow); refer to *Linking the Address Counters* on page 6-73.
dec_link	Decrements this counter (X or Y only) half only if the other half of the counter generates a carry or borrow (overflow or underflow); refer to *Linking the Address Counters* on page 6-73.
preset	Loads this counter from its preset register.
load_alu	Loads the current value of ALU register. ALU value is available in the cycle in which it is programmed.

Incrementing and Decrement the Address Counters

Address counters are the source of the X or Y address or an input to an ALU operation. If a counter is incremented or decremented and is used in the same vector as the address or ALU input, the resulting value is different for each operation:

- Same vector both increments or decrements the counter and uses it as the *address*: the address is the incremented or decremented value. For example, if you are incrementing the address on each vector, the counter is initially loaded with the first address minus one; thus, if the first address is 0, then -1 is loaded into the counter. The increment and decrement operation is at the end of the period. The counters are loaded with the first address used.
- Same vector increments or decrements the counter and uses it as the *ALU input*: the input is the pre-incremented or pre-decremented value. For example, assume counter A is 0 and counter B is 1. If, on one vector, you increment A and set B equal to A plus B, then B still equals 1, because A is added to B before A is incremented.

Linking the Address Counters

If the counters are linked, the linked counter is incremented or decremented only when the operation generates a carry or borrow in the other counter. For example:

```
xa inc   ya inc_link
```

In this example, the **xa** counter is incremented in each cycle, while the **ya** counter is incremented only when the **xa** counter changes from all 1's to all 0's. For example, using 2 bits in each counter, refer to the following values for each cycle:

xa inc	ya inc_link
00	00
01	00
10	00
11	00
00	01
01	01
10	01
11	01
00	10

Address counters are linked to cycle through all X addresses with the same Y address, and then increment to the next Y address.

Enabling (Masking) the Data Bits for Counter and ALU Operations

An enable register specifies (masks) which of the 16 address bits are changed for the counter or ALU operations.

- Setting the bit to *1* in the enable register enables the corresponding bit in the X, Y, or Z address counter;
- Setting the bit to *0* disables that bit. Only enabled bits increment or decrement a counter or shift an ALU.

The enable mask is a 16-bit register that masks an arbitrary combination of address bits for counter and ALU operations. Both the counter logic and the ALU support the masking of non-contiguous address bits and maintain valid counter operations and ALU operations across all 16 address bits.

The enable register can be accessed from the CPU or loaded on-the-fly from the last counter (**xd**, **yd** and **zb**).

Use the following syntax to load the enable registers on-the-fly:

- **load_xenable**
- **load_yenable**
- **load_zenable**

If these opcodes do not appear in a vector, the default is to not change the enable registers.

The enable register can be used to increment address counters by a power of two, which disables the lower address bits. This feature is used in testing burst parts, which increment by 4, 8, or 256.

For example, if the enable register has the value *5* (*101*), and if the **xa** counter is initially loaded with *0*, the **xa inc** opcode has the following affect on successive cycles:

Enable register: *101*

xa *value*: *000*
001
100
101
000

The enable register is normally set from registers before the pattern begins execution. They can be changed from within the pattern by loading them from the **REGD** register:

- Set **REGB** (**xd**, **yd**, or **zb**) to the value that enables the bits you want to change.
- Use one of the following opcodes to load on the fly the **REGD** value into the enable register:

  load_xenable

  load_yenable

  load_zenable

If these opcodes do not appear in a vector, the default is to not change the enable registers.

The enable register can be used to increment address counters by a power of two, which disables the lower address bits. This feature is used in testing burst parts, which increment by 4, 8, or 256.

Address Output for 2-Bit Data Generator and DTOPO RAM Multiplexer

A separate output for the X and Y address generators (16 bits each) can be selected as the input for the DTOPO RAM Multiplexer and the 2-bit Data Generator. Any of the four counters can be selected on-the-fly as this address. For more information about the 2-Bit Data Generator, refer to *On-the-Fly Change to Address Input to the 2-Bit Data Generator* on page 6-83.

ALU Operation Opcodes

Overview

The Arithmetic Logic Unit (ALU) is instructed by the address counters to add, subtract, or perform logical operations on the counter values.

Syntax

The X, Y, and Z counters have separate ALUs that are controlled by using the following syntax with the appropriate expression:

xalu = *x-expression*

yalu = *y-expression*

zalu = *z-expression*

where:

x-expression is the X counter expression;

y-expression is the Y counter expression;

z-expression is the Z counter expression.

For more information, refer to *ALU Expressions* on page 6-79.

If an ALU operation is not programmed on a vector, the following defaults are used:

xalu = 0

yalu = 0

zalu = 0

ALU operations are available in the cycle in which they are programmed. Use the **load_alu** keyword to load the result of the ALU operation into an address counter.

Loading the ALU Result into Address Counter

ALU operations are available in the cycle in which they are programmed. To load the result of the ALU operation into an address counter, the following keyword:

load_alu

Linking ALU Operations

The **xalu, yalu,** and **zalu** operations can use the **link** keyword with the **add**, opcode **subtract**, or **shift** operators. The linked ALU is incremented or decremented only when the other ALU generates a carry or a borrow.

Example:

```
xalu = xb + const xb load_alu
yalu = yb + 0 + link yb load_alu
```

If the **xalu** addition results in a carry, the linked **yalu** increments by one. The ALU values are loaded into **xb** and **yb** and stored for the next cycle; the values of **xalu** and **yalu** are not saved between cycles.

ALU Operands

Address ALUs have a 16-bit left and right inputs, also known as operands.

ALU Operations with Two Operands

For ALU operations with two operands (**add**, **subtract**, **and**, **or**, **xor**), the left and right operands must have different valid values.

Any valid operand other than the **REGA**, **REGB**, **REGC**, and **REGD** counters is available only on the right side of an expression.

ALU Operations with Single Operand

For operations with a single operand (assignment, inversion, shift), all operands are valid: the four address counters (**REGA**, **REGB**, **REGC**, **REGD**) plus **0**, **1**, *const*, and **xdevadr** or **ydevadr**.

Left Operand Sources

- **xalu: xa, xb, xc, xd, 0, 1,** *const*, **ydevadr**
- **yalu: ya, yb, yc, yd, 0, 1,** *const*, **xdevadr**
- **zalu: za, zb, 0, 1,** *const*

where:

0 or **1** — Value 0 or 1.

const — Value of the ALU constant set by the **alu_constant** opcode in this vector.

xdevadr, **ydevadr**, **zdevadr** — Address output from the specified address generator. **xalu** accepts **ydevadr** as input; **yalu** accepts **xdevadr**.

Right Operand Sources

- One of the four address counters: **REGA**, **REGB**, **REGC**, **REGD**.
- Any of the following address counters:

  xa, **xb**, **xc**, **xd**

  ya, **yb**, **yc**, **yd**

  za and **zb**

ALU Expressions

ALU operations are specified with the following C bitwise operators:

+	-	<<1	>>1
|	&	^	~

These operators are used to create expressions that map the following set of ALU codes:

nop	**add**	**add-link**	**sub**
sub-link	**shl**	**shl-link**	**shr**
shr-link	**OR**	**NOR**	**AND**
NAND	**XOR**	**XNOR**	**INV**

Valid ALU expressions for the address generators:

- *Examples of Arithmetic Expressions (add and subtract)* on page 6-79
- *Examples of Logical Expressions (AND, OR, XOR)* on page 6-79
- *Examples of Shift Expressions (shift right or shift left)* on page 6-80
- *Examples of Assignment, Inversion, and No-Operation Expressions* on page 6-80

+ For a formal representation of the opcode syntax in the Backus-Naur form, refer to *Memory Syntax in Backus-Naur Form* on page 6-105.

Examples of Arithmetic Expressions (add and subtract)

xalu = xc - 1 - link

xalu = xa + xb

Examples of Logical Expressions (AND, OR, XOR)

xalu = xc | const

xalu = ~(xa & xb)

Examples of Shift Expressions (shift right or shift left)

```
xalu = xc >> 1
xalu =ydevadr << 1 link
```

The shift operator shifts by one bit. Only the bits that are enabled with **load_xenable**, **load_yenable, load_zenable** are shifted; refer to *Enabling (Masking) the Data Bits for Counter and ALU Operations* on page 6-74.

The rightmost bit in a left shift and the leftmost bit in a right shift are filled with the value from a user register that is set before the pattern start. This value is usually set to *0*. If the link is enabled, its value sets the value that is filled in.

Examples of Assignment, Inversion, and No-Operation Expressions

```
xalu = xc
xalu = ~( const )
```

Data Generator Opcodes

Overview

The FLASH 750 has two data generators:

- 32-bit data generator is logically identical to the address generator. Data is algorithmically generated within the pattern by using the counters and opcodes that correspond to those used for the address counters. Also, refer to the *Address Counter Opcodes* on page 6-70.
- 2-bit data generator is defined outside the pattern file; refer to *Data Generator Sheet on page 3-161*, *DataTool*, Chapter 3.

32-Bit Data Generator Counters

The 32-bit data generator can provide output from two, independent, 32-bit registers (**dga** and **dgb**) by using the following syntax:

dg_sel [*counter]* [*state]*

where
counter is **dga** or **dgb**;

state is one of the counter states listed in *Data Counter States* on page 6-82.

+ The current data generator does not carry from the lower 16 bits to the upper 16 bits.

Data Counter States

The data generator counters can be set to one of the following *states*, which are the similiar as those used by the address counters:

hold Retains the current value (default).

inc Increments the counter.

dec Decrements the counter.

shl Shifts the counter left.

shr Shifts the counter right

rol Rotates the counter left.

ror Rotates the counter right.

ldlo Loads the **alu_constant** into the lower 16 bits of the counter.

ldhi Loads the **alu_constant** into the upper 16 bits of the counter

load_alu Loads the current value of the ALU register; also refer to *Linking ALU Operations* on page 6-77. The value of the ALU operation is available in the cycle in which they are programmed

The default is **hold**.

2-Bit Data Generator

Overview

The data values for the 2-Bit Data Generator are selected from data sets and data groups that are defined not by the pattern file, but by the IG-XL *Data Generator* sheet.

Data Groups and Data Sets

The 2-Bit Data Generator supports up to 4 data groups, each containing 8 data sets. Use the following opcodes to specify which data group and data set to use for this vector:

dset 0...7

dgroup 0...3 | -

The **dgroup** value **-** (hyphen) specifies a runtime repeat of the previous data group, which means not to change the data group.

Defaults: **dset 0 dgroup -**

The first vector of each pattern should set the data group. While the pattern runs, subroutines can be executed with different data by changing the group.

On-the-Fly Change to Address Input to the 2-Bit Data Generator

The address source for the 2-Bit Data Generator can be selected from a separate output on the X and Y address generators (16 bits each). Any of the four counters can be selected on-the-fly as this address by using the following syntax in a pattern file:

xdgadr xa | xb | xc | xd

ydgadr ya | yb | yc | yd

Defaults: **xdgadr xa ydgadr ya**

+ Only the X and Y address generators produce this output, the Z address generator does not.

Pattern Language 6

Frame Select Opcodes

The frame select opcodes are used for selecting a set of bits for testing devices with low-pin counts and many multiplexed inputs. These devices are controlled by a Frame Select RAM on the Memory Test Module (MTM). This RAM provides up to 8 frames or formats for the multiplexed devices.

+ A frame is a set of 8 bits delivered in a single time slice for mux-input devices, which receive X- and Y-address, data, and command data on a common set of pins in a time slice. Frames are used to serialize inputs for muxed input devices.

To select a frame, use the following opcode:

frame *n*

where *n* is the number, *0 <= n <= 7,* of the FrameID column in the IG-XL Frame Select sheet.

Pin Level Opcodes (VIHH)

To enable the VIHH control of the pin levels, specify one of the four enable bits with the appropriate opcode code:

vihhen0 | vihhen1 | vihhen2 | vihhen3

Serial Device Opcodes

Overview

A FLASH 750 has a serialization register to support serial I/O devices. It accepts 8-bit value from the Frame Select Multiplexer and outputs the bits serially. The output from the register is a shifted microinstruction that is executed in a loop, causing the data to be outputted serially.

Syntax

sr_reg [**hold**] [**load**] [**shift**]

where

hold	Retains the current value (default).
load	Loads the current value of the **sr_reg** register. The value of the operation is available in the cycle in which it is programmed.
shift	Shifts the counter left.

DBM/ECR Opcodes

Opcodes are provided for the following DBM/ECR functions:

- To select the DBM on the fly as the alternate data source instead of the Address Data Generator, use the following opcode:

  wr_dbm

- To select the DBM accumulate zeros mode, use the following opcode:

  dbm_acc_zero

- To enable masking of the DBM output, use the following opcode:

  dbm_enable

- To select the DBM learn-DUT mode, use the following opcode:

  learn_dut

- To enable capturing of errors to the ECR, use the following opcode:

  capture_errs

- To latch failure into the registers, add the following opcode to the appropriate memory opcode for the specified row or column:

  latch_errs

- To clear the failure latch failure from the specified row or column, use the following opcode:

  clr_maxcond

- To clear on the fly the ECR counters, use the following opcode:

  clr_ecrcount

- To select on the fly a branch-on-error source (bit line counter, word line counter, row error counter, column error counter, total error counter) for up to branch-on-error conditions. These are also specified in the ErrorSourceStateMap sheet in **DataTool.**

  Use the appropriate opcodes for the signal source for the desired branch-on-error condition:

  ecr_boe*X* | blc | wlc | rec | cec | tec

  where:

  X = 0 to 4, the branch-on-error condition;
  blc—bit line counter;
  wlc—word line counter;
  rec—row error counter;
  cec—column error counter;
  tec—total error counter

Device Power Supply (DPS) Opcodes

Overview

FLASH 750 provides an opcode for enabling the DPS ratcheting and IDDQ/IPEAK measurements on each DPS.

VRATCHET Control

Each station has 4 DPSs with current measure features that are controlled by a special control signal. Levels are set by using a DAC that is controlled by a gate array to provide pattern rate features, such as increment, decrement, or reset to an initial start value. These increment and decrement features are also known as the VRATCHET features. These features change the output level between a beginning and ending value by a pattern-control signal that increments the level at a fixed rate set by the user.

Use the following syntax to initiate ratching on a specific DPS:

vratchet0 | vratchet1 | vratchet2 | vratchet3

Use the following syntax to reset the specified DPS to its starting value:

vreset0 | vreset1 | vreset2 | vreset3

IDDQ/IPEAK Measurements

To enable or disable the IDDQ or IPEAK measurements for an individual DPS, use the following syntax to open or close the measurement window:

iddstart

iddstop

To enable instantaneous (IPEAK) current measurement for an individual DPS, use the following opcode:

iddsync

Control Bits

Overview

A control bit modifies the execution of the specific vector on which it appears. For example, mask failure on this vector or inhibit fail count on this vector.

The control bits do not take arguments or depend on conditions. Control bits accept more than one of these commands per vector, subject to the restrictions on normal mode. Note that **mask** can appear on any vector, even in normal mode.

Opcodes can be combined with the execution control opcodes, except that **ign** and **halt** are mutually exclusive.

Several of the control bits specify a non-default action on a failure; these are grouped as the failure-related control bits; refer to *Failure-Related Control Bits* on page 6-89. The other control bits are listed under *Other Control Bits* on page 6-90.

See also:

Opcodes on page 6-51

Control Bits on page 6-88

Failure-Related Control Bits

By default, when a vector fails, pattern execution halts N cycles later, where N is the pipeline depth; in addition, various fail counters and registers are incremented. Certain control bits let you specify a non-default action in the case of failure. The following control bits can be used in the LVM:

ign Do not halt-on-fail now for failure *N* cycles earlier, *N* is pipeline depth. Default: halt on fail. Cannot be combined with **halt**.

ifc Inhibit fail count from incrementing for a failing vector *N* cycles earlier, *N* is the pipeline depth.

mask Mask failures on this vector.

Unlike other opcodes, **mask** may appear on any vector, even in the normal mode. It applies only to a single vector in a vector pair unless it is specified on both vectors in the pair. Can be used in the LVM**.**

clr_fail Clear the accumulated fail register.

The effect of each control bit on a failing vector is listed below. The first column lists the default action when a vector has none of these control bits. Default action is yes. For control bits, any non-default actions (no) are in **bold**.

	Default	**mask**	**ifc**	**ign**	**clr_fail**
Patgen fail count bumped for this vector?	yes	**no**	**no**	yes	yes
Per-channel fail count bumped for this vector?	yes	**no**	yes	yes	**no**, cleared
Per-channel AFR (accumulated fail register) set on this vector?	yes	**no**	yes	yes	**no**, cleared
Per-channel HRAM fail bit set on this vector?	yes	**no**	yes	yes	yes
HRAM capture failing vector for this vector?	yes	**no**	yes	yes	yes
Halt-on-fail N cycles later?	yes	**no**	yes	yes	yes
Patgen fail flag set N cycles later?	yes	**no**	yes	yes	yes
Halt-on-fail now for a failure N cycles earlier?	yes	yes	yes	**no**	yes

Pattern Language 6

Other Control Bits

The following control bits force a vector into the SVM:

Control Bit	*Definition*
clr_cond	Clears any enabled condition flag if condition evaluates to true. Used to clear any flag except **pass**. May be used only with an **if** statement and can clear only conditions tested in the same vector.
icc	Inhibits cycle count from incrementing on this vector.
stv	Selects vector for HRAM collection for certain HRAM modes. (Store This Vector.). Can be used in LVM**.**

See also:

Opcodes on page 6-51

Control Bits on page 6-88

Conditional Statements

The pattern language supports the conditional execution of opcodes. A set of condition flags can be enabled, and then tested with the **if** opcode.

Condition Flags

The pattern language includes several flags for conditional pattern execution. During a pattern burst, the pattern generator monitors these flags. If one of the flags occurs, the corresponding condition latch is set. It remains set until cleared by the pattern microcode or until another pattern start is executed.

Conditional flags:

fail Set by the pattern generator when a vector fails during a pattern burst.

pass Not a separate flag, but the inverse of **fail**. It is a notational convenience. This flag cannot be cleared.

ext External flag set by the pattern generator.

In addition, the following CPU flags can be set or read from a test program or by opcodes in a pattern file:

cpuA, cpuB, cpuC, cpuD

To set or read the CPU flags from a test program, refer to the **Visual Basic** object **PatGen**.

To set CPU flags from a pattern file, use the **set_cpu** opcode. The **enable** opcode can test for any CPU flag; the **if** opcode can test for **cpuA** or **!cpuA**.

You must explicitly clear the CPU flags; refer to *Clearing Conditions* on page 6-93. Unlike system flags, the CPU flags are not cleared by the tester software when a pattern start is executed.

enable Opcode

Before using the **if** statement to test for one or more of the flags, you first use the **enable** statement to set up the flags. Then use the **if(flag)** statement; the **flag** keyword specifies the condition set up by the most recently executed **enable** statement.

The following example shows how these flags can be used in an **if** statement without a previous **enable** statement:

if(fail)	if(cpuA)	if(ext)
if(pass)	if(!cpuA)	if(!ext)

The **enable** statement is required only for one of the other flags or for a compound condition.

Syntax

enable (*flag-name*)

enable (*flag-name* and *flag-name* ...)

enable (*flag-name* or *flag-name* ...)

enable (none)

- Any of the flags can be negated by using a ! (exclamation point) character before it.
- If more than one flag, either **and** or **or** must be specified between each flag. You cannot mix **and** and **or** in the same **enable** statement.
- Use **none** to clear all enabled conditions without setting new ones.
- If a second **enable** occurs before the conditions previously enabled are cleared, the conditions set up by the first **enable** are replaced by those specified in the second **enable.**

if opcode

The **if (***flag***)** statement tests the conditions enabled by the most recently executed **enable** statement. If the conditions are true, the command is executed.

The flags listed in *enable Opcode* on page 6-91 can be used in an **if** statement without a previous **enable** statement.

If an **if**(*flag*) is executed before its **enable** has been executed, **if(***flag***)** behaves like an **if(pass)**.

The following execution control opcodes can be conditionally executed with the **if** opcode:

- **exit_loop**
- **jump**
- **call**
- **ccall**
- **return**
- **resume**
- **call_glo**
- **jmp_glo**

Clearing Conditions

To clear currently enabled conditions, use one of the following items:

- **clr_cond** control bit may be asserted on any vector that has an **if** opcode. **clr_cond** clears the flags that the **if** has tested, if the condition is true.
- **clr_flag (***flag-list***)** opcode clears the specified flags unconditionally.
- **enable(none)** statement clears all enabled conditions without setting new ones.

The **pass** flag cannot be cleared. The **pass** flag is the inverse of **fail**. It is a notational convenience.

See also:

Opcodes on page 6-51

Control Bits on page 6-88

Modifying Patterns at Runtime

Overview

Specific parts of a loaded pattern file from the executing test program code can be modified. The **IG-XL Visual Basic Hardware Interface** provides a set of objects with properties and methods for manipulating the tester hardware from a test program.

One available object is the **Pattern** object, which accesses a specified pattern. Among its properties and methods:

- set of properties for reading a vector value.
- set of methods for modifying a vector value.

You specify the pattern name and then provide a label and offset to identify the vector to be read or modified.

For more information, refer to **Visual Basic Pattern** object.

Specific Pattern Modifications

The **Pattern** object can be modified for a specific vector:

- Modify a **tset** name or number.
- Modify a numeric operand in the pattern vector microcode.

  The operand for **loop, set_loop, mrepeat,** or **set_code** can be modified. The numeric operand for **repeat** cannot be modified.

- Modify pin data.

  Some methods modify only the regular pin data. Others modify the regular pin data plus the special modes: SCIO, mux, high voltage, and frequency counter.

  Some methods modify the pin group data only for a specific site. Others modify the data for all sites. Other methods modify the data on channels, not pins.

Note that these methods modify only the loaded copy of the program. For permanent changes, you can use **PatternTool** to edit the compiled file, or you can edit the ASCII source and recompile.

Pattern Loading

Pattern loading is transparent to the user. Users specify which patterns for each burst and the patterns are loaded by the IG-XL software. To load a pattern, the IG-XL software must identify how the patterns are used in a burst:

- Patterns executed as a single uninterrupted burst are loaded contiguously in the tester memory. These patterns are referred to as members of a pattern group.
- Other patterns in a burst are called as subroutines or jumped to from patterns in the pattern groups. These patterns do not have to be loaded contiguously in memory.

Pattern data is loaded initially into the LVM and then downloaded to the SVM at pattern start time, if needed. Patterns remain in the LVM between test programs to minimize pattern load time, so that when users are switching between multiple programs, all of the LVM is not filled.

Pattern Loading Functions

The top-level pattern load functions:

- **TL_RESULT tl_PatternLoad(XBstr& PatObjList)**
- **TL_RESULT tl_PatternLoadMem(XBstr& PatObjList, TL_PATTERN_MEM_TYPE MemType)**

  The function **tl_PatternLoadMem()**, which loads vector memory blocks, also loads the memory blocks. The four vector memory blocks supported by **tl_PatternLoadMem()**:

- SVM data loaded in SVM
- SVM data loaded in LVM for download
- LVM data loaded in LVM
- Scan data loaded in LVM.

Pattern Unloading Functions

The top-level unload functions:

- **TL_RESULT tl_PatternUnload(XBstr& PatObjList)**
- **TL_RESULT tl_PatternUnloadMem(XBstr& PatObjList, TL_PATTERN_MEM_TYPE MemType)**

Pattern Linking

The pattern linker resolves imported label references, which can resolved only after the patterns are loaded. Unresolved symbols are cross-referenced in other patterns' global symbol tables, and unresolved addresses are patched by modifying patgen microcode in memory.

Pattern Linking Functions

Top-level pattern linking functions:

- **PatGrp::Link**
- **PatGrp::PatchLink**

Sample Pattern File

```
/*
;=====================================================================
;=                    8243_FUNC                    =
;=====================================================================
*/
import tset rw_inst, rw_data ;
vector ($tset        cs     prog    port2   port4   port5   port6   port7 ) {
start_label begin_vec:
 > rw_inst           .1     .1      .X      .X      .X      .X      .X ;// write to p4
   // ; (1) write 'a to ports 4 through 7 to start out ....
 > rw_inst           .0     .0      0100    .X      .X      .X      .X ; // write to p4
 > rw_data           .1     .1      1010    HLHL    .X      .X      .X ; // data 'a/ strobe p4 exp 'a
 > rw_inst           .0     .0      0101    HLHL    .X      .X      .X ; // wr p5 / strobe p4 exp 'a
 > rw_data           .1     .1      1010    HLHL    HLHL    .X      .X ; // data 'a/ strobe p4-5 exp 'a
 > rw_inst           .0     .0      0110    HLHL    HLHL    .X      .X ; // wr p6 / strobe p4-5 exp 'a
 > rw_data           .1     .1      1010    HLHL    HLHL    HLHL    .X ; // data 'a/ strobe p4-6 exp 'a
    // ; (9) do orl of '5 to all ports 4 through 7
 > rw_inst           .0     .0      1000    HLHL    HLHL    HLHL    .X ; // orl p4/ strobe p4-7 exp 'a
 > rw_data           .1     .1      0101    .H      HLHL    HLHL    .X ; // data '5/ strobe p4 exp 'f
 > rw_inst           .0     .0      1001    .H      HLHL    HLHL    .X ; // orl p5/ strobe p4 exp 'f
 > rw_data           .1     .1      0101    .H      .H      HLHL    .X ; // data '5/ strobe p4-5 exp 'f
 > rw_inst           .0     .0      1010    .H      .H      HLHL    .X ; // orl p6/ strobe p4-5 exp 'f
 > rw_data           .1     .1      0101    .H      .H      .H      .X ; // data '5/ strobe p4-6 exp 'f
halt
 > rw_inst           .1     .1      .X      .X      .X      .X      .X ;
}
```

Sample Memory Patterns

Overview

The following sample memory patterns are provided:

- *1M x 8 March Pattern* on page 6-99
- *Flash Program Pattern* on page 6-101
- *Flash Erase Program* on page 6-103

+ These examples are **not executable** as is. These simplified examples are listed to show how address counters can be manipulated to test different kinds of memory. For this reason, they do not include all the code required to be executable.

1M x 8 March Pattern

```
Import tset             inactive_cycle,
             write_cycle,
             read_cycle;
instruments = {
mtm;
}

// specify device size
#define XMAX                          1023
#define YMAX                1023
#define FULL_ADDR_SIZE                1048576        // TOO BIG FOR MREPEAT
#define RPT_FULL_ADDR_SIZE            1048575        // TOO BIG FOR MREPEAT

// cycle definition macros
#define CYCLE_DEFINITION \
             $tset      DQS:S     ADDRS:S        CS:S      WE:S      OE:S
// :S MEANS DATA FOR THIS PINGROUP IS SYMBOLIC
#define READ_CYCLE \
             read_cycle           .E        .D        .D        .D        .D
#define WRITE_CYCLE \
             write_cycle          .D        .D        .D        .D        .D
#define      INACTIVE_CYCLE \
             incactive_cycle      .X        .1        .1        .1        .1
// .D=DRIVE DATA FROM APG,
// .E=EXPECT DATA FROM APG
// .X=EXPECT MASK
// .1=DRIVE 1

// vector statement
vector (CYCLE_DEFINITION) {
start_label march:
// LOAD INITIAL VALUES FROM PRESET
             (
             xa preset ya preset
             dgroup 0
             )
             > INACTIVE_CYCLE;
```

```
// WRITE BACKGROUND DATA FROM MIN TO MAX
        (
        xa inc ya inc_link
        xdevadr xa ydevadr ya
        xdgadr xa ydgadr ya
        dset 0
        )
        mrepeat RPT_FULL_ADDR_SIZE
        > WRITE_CYCLE;

// LOOP FOR ALL LOCATION, READING DATA FROM
// EACH ADDRESS LOCATION, AND WRITING COMPLIMENT
// DATA TO THAT LOCATION.
        loop_start:
        (
        xa hold ya hold
        xdevadr xa ydevadr ya
        xdgadr xa ydgadr ya
        dset 0
        )
        loopA FULL_ADDR_SIZE
        > READ_CYCLE;

        (
        xa inc ya inc_link
        xdevadr xa ydevadr ya
        xdgadr xa ydevadr ya
        dset 1
        )
        end_loopA loop_start
        > WRITE_CYCLE;

// READ COMPLIMENT DATA FROM MIN TO MAX
        (
        xa inc ya inc_link
        xdevadr xa ydevadr ya
        xdgadr xa ydgadr ya
        dset 1
        )
        mrepeat RPT_FULL_ADDR_SIZE
        > READ_CYCLE;

// All finished
halt > INACTIVE_CYCLE;
}
```

Flash Program Pattern

```
import tset program_cycle, read_data_cycle, read_rb_cycle, tristate_cycle, hz50_cycle;
// specify device size (16M: 11X 10Y 8 I/O)

#define XMAX 2048
#define YMAX 1024
#define NUM_OF_PIPES 37
#define MAX_POLL 500

// cycle definition macros

#define CYCLE_DEFINITION \
($tset CS   WE   OE   ADDR   RBY DQS)
#define PROGRAM_COMMAND \
program_cycle        0     0     1     .D     X     01000000     // Program command = 0x40
#define PROGRAM_DATA \
program_cycle        0     0     1     .D     X     .D
#define POLL_READY_BUSY \
read_rb_cycle        0     1     0     .D     H     .X
#define INACTIVE_CYCLE \
tristate_cycle       1     1     1     .D     X     .D
#define HZ50_CYCLE \
hz50_cycle           1     1     1     .D     X     .D

vector CYCLE_DEFINITION {
// setup counters
start_patt: (  xa preset ya preset
               dgroup 0
               ) set_loopB YMAX
               > INACTIVE_CYCLE;

// issue program setup command
pgm_setup: (  xa hold ya hold
               ) loopA XMAX
               > PROGRAM_COMMAND;

// give address and data to program
pgm_data: (  xa hold ya hold
               dset 0
               )
               > PROGRAM_DATA;
```

```
// poll ready/busy pin
poll_rb: (  xa hold ya hold
                    ) loopA MAX_POLL, ign
                    > POLL_READY_BUSY;

// pipeline the error
                    ( xa hold ya hold
                    ) repeat NUM_OF_PIPES, ign
                    > HZ50_CYCLE;

// check to see if program is done
                    ( xa hold ya hold
                    ) if (pass) jump next_address, ign
                    > INACTIVE_CYCLE;

// timeout if rb fails more than MAX_POLL
                    ( xa hold ya hold
                    ) end_loopA poll_rb, ign, clr_fail
                    > INACTIVE_CYCLE;

// increment address for next byte to program
next_address: ( xa inc ya inc_link
                    ) end_loopA pgm_setup
                    > INACTIVE_CYCLE;

                    ( xa hold ya hold
                    ) end_loopB pgm_setup
                    > INACTIVE_CYCLE;
stop_patt: halt
                    >INACTIVE_CYCLE;
}
```

Flash Erase Program

```
import tset erase_cycle, read_data_cycle, read_rb_cycle, inactive_cycle, cycle_50mhz;

// specify device size (16Mb: 10X 6Y 5Z 8I/O)
#define XMAX 1024
#define YMAX 64
#define ZMAX 32                         // Number of blocks = 32
#define NUM_OF_PIPES 37
#define MAX_POLL 500                    // For timeout

// cycle definition macros
#define CYCLE_DEFINITION \
($tset CS WE OE ADDR RBY DQS)
#define ERASE_SETUP    \
erase_cycle        0    0    1    .D    X    00100000    // Erase setup = 0x20
#define ERASE_COMMAND \
erase_cycle        0    0    1    .D    X    11010000    // Erase command = 0xD0
#define POLL_READY_BUSY \
read_rb_cycle      0    1    0    .D    H    .X
#define INACTIVE_CYCLE \
inactive_cycle     1    1    1    .D    X    .D
#define CYCLE_50MHZ \
cycle_50mhz        1    1    1    .D    X    .D
vector CYCLE_DEFINITION {
// setup counters
start_patt: (xa preset ya preset za preset
            )
            > INACTIVE_CYCLE;

// issue erase setup command
erase_setup: ( xa hold ya hold za hold
            ) loopA ZMAX
            > ERASE_SETUP;

// issue erase confirm command
erase_confirm: ( xa hold ya hold za hold
            )
            > ERASE_COMMAND;

// poll ready/busy pin
poll_rb: ( xa hold ya hold za hold
            ) loopA MAX_POLL, ign
            > POLL_READY_BUSY;
```

```
// pipeline the error
                ( xa hold ya hold za hold
                ) repeat NUM_OF_PIPES, ign
                > CYCLE_50MHZ;

// check to see if erase is done
                ( xa hold ya hold za hold
                ) if (pass) jump next_block, ign
                > INACTIVE_CYCLE;

// timeout if rb fails more than MAX_POLL
                ( xa hold ya hold za hold
                ) end_loopA poll_rb, ign, clr_fail
                > INACTIVE_CYCLE;

// increment address for next block to erase
next_block: ( xa hold ya hold za inc
                ) end_loopA erase_setup
                > INACTIVE_CYCLE;

stop_patt: halt
                > INACTIVE_CYCLE;

}
```

Memory Syntax in Backus-Naur Form

```
               |        (FIELDS )           | 01

FIELDS ::=     |        XCOUNTER            |
               |        YCOUNTER            |
               |        ZCOUNTER            |
               |        XALU                |
               |        YALU                |
               |        ZALU                |
               |        XDEVADR             |
               |        YDEVADR             |
               |        ZDEVADR             |
               |        DATA_2BIT           |
               |        DATA_32BIT          |
               |        CONST               |
               |        ENABLE              |
               |        FRAME               |
               |        SHIFTREG            |
               |        wr_dbm              |
               |        dbm_enable          |
               |        dbm_acc_zero        |
               |        capture_errs        |
               |        VIHH                |
               |        VRATCHET            |
               |        IDDQ                |
               |        learn_dut           |
               |        clr_ecrcount        |
               |        clr_maxond          |
               |        BRONERR             | 1+
               |        latch_errs          |
```

```
XCOUNTER ::=   |   |  xa     |   |  hold       |     |
               |   |  xb     |   |  inc        |     |
               |   |  xc     |   |  inc_link   |     |
               |   |  xd     | 1 |  dec        |     |
                                 |  dec_link   |     |
                                 |  preset     |     |
                                 |  load_alu   | 1   |
               |                                     | 1+
YCOUNTER ::=   |   |  ya     |   |  hold       |     |
               |   |  yb     |   |  inc        |     |
               |   |  yc     |   |  inc_link   |     |
               |   |  yd     | 1 |  dec        |     |
               |                 |  dec_link   |     |
               |                 |  preset     |     |
               |                 |  load_alu   |1    |
               |                                     | 1+
ZCOUNTER ::=   |   |  za     |   |  hold       |     |
               |   |  zb     | 1 |  inc        |     |
               |                 |  dec        |     |
               |                 |  preset     |     |
               |                 |  load_alu   |1    |
               |                                     | 1+
XDEVADR ::=    |   xa  |
               |   xb  |
               |   xc  |
               |   xd  | 1
YDEVADR ::=    |   ya  |
               |   yb  |
               |   yc  |
               |   yd  | 1
ZDEVADR ::=    |   ya  |
               |   yb  | 1
```

```
XALU ::=   |    |  xa        |   +   |  xa        |   |  + link   |  01   |
           |    |  xb        |       |  xb        |                       |
           |    |  xc        |       |  xc        |                       |
           |    |  xd        | 1     |  xd        |                       |
           |                         |  0         |                       |
           |                         |  1         |                       |
           |                         |  const     |                       |
           |                         |  ydevadr   | 1                     |
           |                                                              |
           |    |  xa        |   -   |  xa        |   |  - link   |  01   |
           |    |  xb        |       |  xb        |                       |
           |    |  xc        |       |  xc        |                       |
           |    |  xd        | 1     |  xd        |                       |
           |                         |  0         |                       |
           |                         |  1         |                       |
           |                         |  const     |                       |
           |                         |  ydevadr   | 1                     |
           |                                                              |
           |    |  xa                |  | <<1     |   |  link     |  01   |
           |    |  xb                |  | >>1     | 1                     |
           |    |  xc                |                                    |
           |    |  xd                |                                    |
           |    |  0                 |                                    |
           |    |  1                 |                                    |
           |    |  const             |                                    |
           |    |  ydevadr           | 1                                  |
           |                                                              |
           |    |  xa        |  | &  |            |  xa        |          |
           |    |  xb        |  | |  |            |  xb        |          |
           |    |  xc        |  | ^  | 1          |  xc        |          |
           |    |  xd        | 1                  |  xd        |          |
           |                                      |  0         |          |
           |                                      |  1         |          |
           |                                      |  const     |          |
           |                                      |  ydevadr   | 1        |
```

```
XALU ::=    |                                                            |
            |                                                            |
            | ~(     |        xa  |     |  &          |  |  xa       |  )  |
            |        |        xb  |     |  |          |  |  xb       |     |
            |        |        xc  |     |  ^          | 1|  xc       |     |
            |        |        xd  | 1                    |  0
            |                                            |  |        |     |
            |                                            |  const    |     |
            |                                            |  ydevadr  | 1   |
            |                                                            |
            |    | ~ |        01        |  xa         |                    |
            |                           |  xb         |                    |
            |                           |  xc         |                    |
            |                           |  xd         |                    |
            |                           |  0          |                    |
            |                           |  1          |                    |
            |                           |  const      |                    |
            |                           |  ydevadr    | 1                  | 1
YALU ::=    |    |  ya        |    +    |  ya         |  |  + link  |   01  |
            |    |  yb        |         |  yb         |                    |
            |    |  yc        |         |  yc         |                    |
            |    |  yd        | 1       |  yd         |                    |
            |                           |  0          |                    |
            |                           |  1          |                    |
            |                           |  const      |                    |
            |                           |  xdevadr    | 1                  |
            |                                                            |
            |    |  ya        |    -    |  ya         |  |  - link  | 01    |
            |    |  yb        |         |  yb         |                    |
            |    |  yc        |         |  yc         |                    |
            |    |  yd        | 1       |  yd         |                    |
            |                           |  0          |                    |
            |                           |  1          |                    |
            |                           |  const      |                    |
            |                           |  xdevadr    | 1                  |
```

```
YALU ::=      |                                                              |
              |                                                              |
              |     |  ya           |     | <<1         |   |  link     | 01   |
              |     |  yb           |     | >>1         | 1                    |
              |     |  yc           |                                          |
              |     |  yd           |                                          |
              |     |  0            |                                          |
              |     |  1            |                                          |
              |     |  const        |                                          |
              |     |  xdevadr      | 1                                        |
              |                                                              |
              |     |  ya           |     |  &          |   |  ya       |      |
              |     |  yb           |     |  |          |   |  yb       |      |
              |     |  yc           |     |  ^          | 1 |  yc       |      |
              |     |  yd           | 1                     |  yd       |      |
              |                                           |  0        |      |
              |                                           |  1        |      |
              |                                           |  const    |      |
              |                                           |  xdevadr  | 1    |
              |                                                              |
              | ~(  |  ya           |     |  &          |   |  ya       |   )  |
              |     |  yb           |     |  |          |   |  yb       |      |
              |     |  yc           |     |  ^          | 1 |  yc       |      |
              |     |  yd           | 1                     |  0        |      |
              |                                           |  1        |      |
              |                                           |  const    |      |
              |                                           |  xdevadr  | 1    |
              |                                                              |
              |                           |  ya         |                      |
              |                           |  yb         |                      |
              |     |  ~         |   01   |  yc         |                      |
              |                           |  yd         |                      |
              |                           |  0          |                      |
              |                           |  1          |                      |
              |                           |  const      |                      |
              |                           |  xdevadr    | 1                    | 1
```

Pattern Language

6

```
ZALU ::=      |     |   za          |   +   |   za          |                             |
              |     |   zb          | 1     |   zb          |                             |
              |                             |   0           |                             |
              |                             |   1           |                             |
              |                             |   const       | 1                           |
              |                                                                           |
              |     |   za          |   -   |   za          |                             |
              |     |   zb          | 1     |   zb          |                             |
              |                             |   0           |                             |
              |                             |   1           |                             |
              |                             |   const       | 1                           |
              |                                                                           |
              |     |   za                  |   | <<1       |                             |
              |     |   zb                  |   | >>1       | 1                           |
              |     |   0                   |                                             |
              |     |   1                   |                                             |
              |     |   const               | 1                                           |
              |                                                                           |
              |     |   za          |   | & |               |   za              |         |
              |     |   zb          | 1 | | |               |   zb              |         |
              |                         | ^ |               |   0               |         |
              |                                             |   1               |         |
              |                                             |   const           | 1       |
              |                                                                           |
              | ~(      |           za |    |   &           |   |   za          |    )    |
              |         |           zb | 1  |   |           |   |   zb          |         |
              |                             |   ^           | 1 |   0           |         |
              |                                                 |   1           |         |
              |                                                 |   const       | 1       |
              |                                                                           |
              |     | ~ | 01                |   za          |                             |
              |                             |   zb          |                             |
              |                             |   0           |                             |
              |                             |   1           |                             |
              |                             |   const       | 1                           | 1

DATA_2BIT ::= |     DSET            |
              |     DSRC            | 1+

DSET ::=      |     dset S          |     dgroup G      |   01  | 1
```

S is a number between 0 and 7. G is a number between 0 and 3, or the hyphen character (-), for runtime repeat of the data group from the previous cycle.

```
DSRC ::=          |  xdgadr    |  xa          |          |
                  |            |  xb          |          |
                  |            |  xc          |          |
                  |            |  xd          | 1        |
                  |                                      |
                  |  ydgadr    |  ya          |          |
                  |            |  yb          |          |
                  |            |  yc          |          |
                  |            |  yd          | 1        | 1+

DATA_32BIT::=     |  DCOUNTER                            |
                  |  DSEL                                |
                  |  DGALU                               | 1+

DCOUNTER ::=      |  |      dga  |    |      hold    |   |
                  |  |      dgb  | 1  |      inc     |   |
                  |  |           |    |      dec     |   |
                  |                   |      shl     |   |
                  |                   |      shr     |   |
                  |                   |      ror     |   |
                  |                   |      ldlo    |   |
                  |                   |      ldhi    |   |
                  |  |           |    |      preset  |   |
                  |                   |      load_alu| 1 | 1+
```

```
DGSEL  ::=          |   dga          |
                    |   dgb          | 1
               |                                                   |
               |                                                   |
               |    |   dga        |   |    &          |   |  dga     |   |
               |    |   dgb        |   |    |          |   |  dgb     |   |
               |    |   dgc        |   |    ^          |1  |  dgc     |   |
               |    |   dgd        | 1                     |  dgd     |   |
               |                                           |  0       |   |
               |                                           |  1       |   |
               |                                           |  const   | 1 |
               |                                                   |
               |                                                   |
               |    |   dga        |   |    &  |       |  dga          |   |
               |    |   dgb        |   |    |  |       |  dgb          |   |
               |    |   dgc        |   |    ^  | 1     |  dgc          |   |
               | ~( |   dgd        | 1                 |  dgd          | ) |
               |                                       |  0            |   |
               |                                       |  1            |   |
               |                                       |  const        | 1 |
               |                                                   |
               |                                                   |
               |                               |     dga    |          |
               |                               |     dgb    |          |
               |    |   ~            | 01      |     dgc    |          |
               |                               |     dgd    |          |
               |                               |     0      |          |
               |                               |     1      |          |
               |                               |     const| 1          | 1
```

```
CONST ::=      alu_const N
```

N is a 16-bit hexadecimal (0 to 0xffff) or decimal (0 to 65535) number.

```
ENABLE  ::=    |    load_xenable  |
               |    load_yenable  |
               |    load_zenable  | 1+

FRAME  ::=     frame   N
```

N is a frame number from 0 to 7.

```
VIHH  ::=      |    vihhen0       |
               |    vihhen1       |
               |    vihhen2       |
               |    vihhen3       | 1+

BRONERR  ::=   |    ecr_boe0      |   |    tec          |
               |    ecr_boe1      |   }    blc          |
               |    ecr_boe2      |   |    wlc          |
```