1

Micro-D

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

A Functional-Test Specification Language

by

Dewi L. Williams, M.A.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of
Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University

August, 1988

© Copyright

1988, Dewi L. Williams

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-46269-8

The undersigned hereby recommends to the
Faculty of Graduate Studies and Research
acceptance of the thesis,
"A Functional-Test Specification Language"
submitted by Dewi L. Williams, M.A.,
in partial fulfilment of the requirements
for the degree of Master of Engineering.

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

September, 1988

ABSTRACT

Each digital circuit that is manufactured must be tested to ensure that it conforms to its specification. In spite of advances in Automatic Test Pattern Generation, for most circuits, a set of stimuli and expected responses must be created by a human being. The concept, syntax and vocabulary of existing languages for specifying functional tests are not adequate for the complexity of present-day digital circuits.

This thesis considers the features that should be part of a test-specification language, develops a language, and verifies its suitability by creating a functional test for a serial communications circuit.

The translation of test programs to the language of automatic test equipment is also discussed, and two methods are proposed: one based on known technology, the other, on a more speculative rule-based system.

ACKNOWLEDGEMENTS

I would like to thank Dr. J.P. Knight, my thesis supervisor, for his encouragement and assistance, and for the useful discussions that we had. In addition, I wish to thank Robert Hum of Bell-Northern Research and John Peacock of Cadence Computer Corp. for their considered criticism of drafts of this thesis.

I would also like to acknowledge the financial encouragement that Bell-Northern Research makes to those employees who become part-time students.

Above all, I would like to thank my wife, Eufron Williams, for her support.

TABLE OF CONTENTS

Chapter 1. Testing Digital Circuits	1
1.1 Introduction	1
1.2 Functional tests and Parametric tests	2
1.3 Measurement vs. Go/No-go testing	3
1.4 Test patterns	4
1.5 Higher level languages	8
1.6 Functional Test Programming Language	8
Chapter 2. Previous Test Languages	10
2.1 Fairchild's FACTOR	10
2.2 GENRAD's TPP (Test Pattern Processor) language	20
2.3 Other pattern languages	22
2.3.1 Vector-based languages	22
2.4 State-change languages	28
2.5 Transaction description languages	31
2.6 Deficiencies	33
Chapter 3. Philosophy of FTSL	34
3.1 Characteristics of test engineers	34
3.2 The transaction	35
3.3 Other language considerations	37
3.3.1 Translation	37
3.3.2 Scope	37
3.4 Desirable language characteristics	37
Chapter 4. The Development of a Functional-Test Specification Language	42
4.1 Transactions, procedures and functions	42
4.2 Pins	43
4.2.1 Signal levels	44
4.2.2 Forcing levels and expecting levels	45
4.3 Registers	49
4.3.1 Register levels	52
4.4 Timing: specification of events	53
4.4.1 Setting values of timing symbols	55
4.4.2 Using timing symbols	58
4.4.3 Flags	60
4.4.4 The concept of "present time"	61
4.4.5 Clocks and continuous processes	65
4.5 Specifying test data	68
4.5.1 Data definition	70
4.5.2 Data access	71
4.6 Passing data to Transactions	72

4.7	Test program	75
4.8	The language	75
4.9	Translating FTSL to ATE language	76
4.9.1	Using known techniques	77
4.9.2	A more speculative method	79
Chapter 5. Verification, Conclusions, and Future Work		86
References		97
Appendix A. FTSL		100
A.1	Transactions vs. Procedures	101
A.1.1	Transaction calls	102
A.2	Declarative sections	104
A.2.1	PIN	105
A.2.2	REGISTER	108
A.2.3	TIME	110
A.2.4	TESTDATA	113
A.2.5	VAR: Additional data types	116
A.2.5.1	The "flag" data type	116
A.3	Timing-relationship-statements	118
A.3.1	Concurrent statement	119
A.3.2	Waituntil statement	121
A.3.3	At statement	123
A.3.4	Forever statement	126
A.4	Pin-value-statements	127
A.4.1	Force statement	128
A.4.2	Expect statement	130
A.5	Functions and Procedures: Data control	132
A.5.1	GetData function	133
A.5.2	ResetData procedure	136
A.5.3	LastData function	137
A.5.4	EOD function	139
Appendix B. Example: Testing a UART		141
Appendix C. Circuit Specifications for the UART		154
Appendix D. Creating an Event-List from an FTSL Program		160
D.1	Creating the static data structure	161
D.2	The statement/expression chain	162
D.2.1	Concurrent	164
D.2.2	At	165
D.2.3	Waituntil	166
D.2.4	Forever	167
D.2.5	Force and Expect	168
D.3	Pin records	170

D.3.1	Pin record and field descriptions	171
D.4	Register records	174
D.4.1	Register record and field descriptions	176
D.5	Records for "flag" data type	179
D.5.1	Flag record descriptions	179
D.6	The dynamic data structure: execution	181
D.7	Process	184
D.8	Demon	188

LIST OF ILLUSTRATIONS

Figure 1.	Test pattern for 2-input AND gate	5
Figure 2.	Automatic Test Equipment waveshapes	12
Figure 3.	Timing diagram and full-rate pattern	26
Figure 4.	The effect of longer test period	27
Figure 5.	State-change notation	29
Figure 6.	Q-bus fragment	61
Figure 7.	Translation from FTSL to ATE language	77
Figure 8.	Structure of an FTSL test program	80
Figure 9.	Doubling the test period: before	84
Figure 10.	Doubling the test period: after	84
Figure 11.	Moving edges	85
Figure 12.	Statement/expression record	162
Figure 13.	Concurrent record	164
Figure 14.	"At" record	165
Figure 15.	Waituntil record	166
Figure 16.	Statement/expression record	167
Figure 17.	Force or Expect records	168
Figure 18.	Pin and PinRegister records	171
Figure 19.	Register and Register-to-pin records	176
Figure 20.	Flag records	179
Figure 21.	Process and stack records	185
Figure 22.	Demon records	188

GLOSSARY

ATE	Automatic Test Equipment: programmable test systems for performing specified tests on electronic devices.
ATPG	Automatic Test Pattern Generation: computer methods for generating digital test patterns from circuit description rather than test specifications.
DUT	Device Under Test: the device being tested by ATE (q.v.)
EOB	End Of Block: a byte marking the end of a block of data, in a serial communication scheme.
NRZ	Non-Return-to-Zero: a binary waveform that when forced from the 0-level to the 1-level, does not return to the 0-level after a specified interval, but remains at the 1-level until forced to 0.
PEP	Pulse Exceeds Period: a pulse that is started in one test period, but which does not end (does not return to the original level) until the next test period.
RO	Return to One: a binary waveform whose quiescent state is the 1-level: when forced to the 0 level, it returns to the 1-level after a specified interval.
RZ	Return to Zero: a binary waveform whose quiescent state is the 0-level: when forced to the 1 level, it returns to the 0-level after a specified interval.
SBC	Surround By Complement: a binary waveform consistent of a pulse whose level is specified: before and after the pulse (during the remainder of the test period), the level of the waveform is the complement of that during the pulse.
SOB	Start Of Block: a byte marking the start of a block of data, in a serial communication scheme.

Test Pattern A table of vectors (q.v.): each column in the table corresponds to one pin of the device (the circuit) to be tested, and each row forms one vector.

Test Period In Automatic Test Equipment, the period of the clock which causes vectors to be applied to the device under test, one vector per clock period.

UART Universal Asynchronous Receiver-Transmitter: a circuit which receives characters from an asynchronous communication line and makes them available to a parallel bus, and accepts characters from a parallel bus and transmits them on the asynchronous communication line.

Vector A set of data values, one per pin, to be applied to the device under test.

VHDL VLSI (Very Large Scale Integration) Hardware Description Language.

VOH A test which verifies that a particular output pin of a device has the capability of maintaining a specified voltage while driving a specified load.

CHAPTER 1. TESTING DIGITAL CIRCUITS

1.1 INTRODUCTION

Each digital circuit that is manufactured must be tested to ensure that it conforms to its specification. Whether the circuit is implemented as an integrated circuit, an assembly on a printed circuit board, or by some other method, it must still be tested; and the test engineer or the circuit designer must specify what stimuli are to be applied, what responses are to be expected from the circuit, and what the test environment should be.

There are techniques for deciding just how much testing is needed to ensure a given level of quality, and for some specific types of circuits (such as circuits that are purely combinatorial), there are Automatic Test Pattern Generation (ATPG) techniques that will generate the necessary stimuli and responses.

However, for most circuits, and for the greater part of each circuit, a human being must create the pattern of stimuli and expected responses, and must describe it in a form that can be understood by a computer or directly by Automatic Test Equipment (ATE). Most manufacturers of ATE try

to make it easier for the test engineer to write test patterns by providing a language in which the tests may be written. There have also been attempts to create a neutral Test Pattern Language, independent of any one manufacturer, in which tests could be written and then translated as needed to the language of specific ATE.

The increasing complexity of digital circuits has made it desirable to seek a better language in which tests could be described: one that allows the test engineer to think in a "natural" way about the circuit, but which could still be translated for Automatic Test Equipment.

This thesis defines such a language.

1.2 FUNCTIONAL TESTS AND PARAMETRIC TESTS

Every digital circuit needs testing in both functional and parametric categories.

"Functional testing" (used here in the sense in which it is used in the manufacturing testing community) is concerned with logic levels: it expects certain sequences of 1's and 0's at the output pins after other specific sequences of 1's and 0's have been applied to the input pins. Discrimination of the results is strictly binary: (is the value a logic 0, or is it a logic 1?), and generally there

is no consideration of rise or fall time (the time taken to change state): both input and output waveforms are deemed to switch instantaneously between the 0 and 1 levels. These tests are performed at speed, with clock rates approximating those that the circuit would encounter in actual service.

"Parametric testing" is more concerned with "analog" values: actual current, actual voltage, and actual time delay. An example is V_{oh} , the "output-voltage-high" test, which verifies that when an output pin is in the high state (logic level 1), and is driving a specified load, then the voltage is not less than some specified value. Another example is T_{cw} , the minimum clock width test, which verifies that the circuit will still perform properly even when the width of the clock pulse is reduced to the specified value.

Other parametric tests verify minimum or maximum currents or other timing values.

1.3 MEASUREMENT VS. GO/NO-GO TESTING

Functional testing is go/no-go: either the circuit runs correctly, or it does not (for example, in an adder circuit, either the value on the output bus equals the sum of the values on the two input buses, or it does not: an adder

that gives the sum of 10 and 20, as "approximately 30" is deemed to be a failure).

Parametric testing has two modes: the go/no-go mode in which the test equipment only determines that the parameter in question is within the given limit, and the measurement mode, in which the ATE must determine and record the actual value of the parameter.

1.4 TEST PATTERNS

Sequences of 1's and 0's in their correct timing relationships are obviously needed for functional testing. However, such sequences are also needed for parametric testing and parametric measurement.

- Timing tests and measurements can be made only while the circuit is running, and while the signals concerned in the test are being exercised.
- For voltage and current tests and measurements, the circuit must be put into a state appropriate to the measurement. For example, to make the V_{oh} (output-voltage-high) test, the circuit must be driven

to a state in which the logic level at the appropriate output signal is indeed high.

To exercise the circuit, and to drive it into a specific state, sequences of logic levels are again needed.

The conventional way to show such a pattern of 1's and 0's is as a table, with one column of the table for each external pin of the circuit. Each column may be given a label, which is the name of the signal connected to the corresponding pin. Such a table is known as a "test pattern".

INPIN2			
INPIN1			OUTPIN
0	0	0	
0	1	0	
1	0	0	
1	1	1	

Figure 1. Test pattern for 2-input AND gate

The test pattern for the 2-input AND gate is recognizable as the truth table for the circuit; for any combinatorial circuit, the truth table can be used as the pattern. For a sequential circuit the testing, and hence the pattern, is more complex.

The rows of the pattern are applied in succession. The pattern itself does not specify what voltages correspond to the 0 and 1 levels, nor does it specify at what times the

values are to be applied, other than the implication that the events of the second row follow those of the first row, and so on.

In fact, a specific time -- the test period -- is allowed for each row, and the test period and the voltages are specified in another part of the overall test program.

The test pattern is used as the notation for describing stimuli and expected responses because it corresponds closely to the architecture of the stimulus/response generator in the typical ATE (Automatic Test Equipment). In its simplest form, the pattern generator consists of a memory containing the 1's and 0's, which are clocked out, row by row. Each manufacturer of ATE has variations on the pattern generator theme which allow control over the sequencing of pattern rows, the specification of pins as input or output, and some control over test period and timing values. The extensions may take the form of dummy control rows added to the pattern, or of "micro-code" fields added to each row (see [Fair78], [Gehr85], and [Tek76]).

The test pattern notation is still a very low-level specification method, corresponding to the binary programming needed in the early days of digital computing before even symbolic assemblers were available.

As well as this general unsuitability for human use, there are two specific disadvantages of pattern notation which become more serious as the circuits to be tested become more complex:

1. All pins must be considered simultaneously: the state of every pin must be specified in every row of the pattern. This implies that the human test programmer must consider every part of the circuit simultaneously. This is simple for a 2-input AND circuit, or even for an IC consisting of six identical AND circuits. However, when different parts of the circuit are to perform different functions or are to have different stimulus/response sequences, it becomes difficult.

For example, consider a circuit that has separate transmit and receive paths which are to be tested simultaneously; or a circuit with multiple external buses that must be tested with different values on each bus.

2. The notation demands that the same period be imposed on all pins. This condition is indeed imposed by the hardware of most ATE, but a clever test programmer can often work around it.

For example, consider a data-rate converter circuit that has one clock rate on its input, and another on its output. The test programmer must decide whether to synchronize the ATE test period to the input clock or the output clock; but then may be able to synthesize the other clock by varying the timing specification of its edges, outside the pattern specification.

1.5 HIGHER LEVEL LANGUAGES

Several more or less general-purpose languages have been defined for test programs; that is, for specifying the power levels, the voltage, current, and other parametric levels, and starting and stopping test patterns.

None has the ability to specify a full functional test, or a test pattern to be used for a parametric test.

1.6 FUNCTIONAL TEST PROGRAMMING LANGUAGE

FTSL, the Functional Test Programming Language described in "Appendix A. FTSL" on page 100 allows the test programmer to specify exactly what tests are to be performed on what pins, and also to specify the tolerance on all timing values; in effect, the programmer creates a specifica-

tion for a pattern, rather than creating the pattern itself. The test specification can be compiled into the language of any ATE whose hardware will support the tests to be performed.

This gives the additional benefit that, from one specification, test patterns can be created for several types of ATE: thus, testing of the given circuit need not be tied to one piece of test equipment.

CHAPTER 2. PREVIOUS TEST LANGUAGES

Before examining the more recent work on languages for digital test stimulus-and-response specification, it is worth looking at the languages that were originally created by ATE manufacturers for controlling their proprietary test equipment. These languages are biased towards specifying the fine detail of controlling the ATE hardware rather than specifying, in understandable form, what tests are to be performed. Examining such languages then shows the difficulties that test engineers have had to cope with.

2.1 FAIRCHILD'S FACTOR

One of the earliest series of digital ATE was the SENTRY series, produced by Fairchild. The most numerous version was the SENTRY VII, but the language, FACTOR [Fair78], was developed from earlier versions. FACTOR covered both the control program (the care and feeding of the device under test, starting test patterns, computing results, etc) and the pattern itself, with variations in the pattern dialect for different hardware pattern generators.

In typical ATE, the details of waveshape (the number of signal level transitions, and their timing with each test

period) are controlled by "waveform generators". In the SENTRY VII, there are six waveform generators for stimulus and two for response checking. Each pin of the device under test (DUT) can be connected to a stimulus generator and to a response generator. All waveform generators are synchronized to the same test period, but each waveform can have different edges and different edge timing.

The test programmer has a repertoire of waveshapes to choose from: Figure 2 on page 12 shows the more important shapes. Four test periods are shown in the figure, with the vector data for the four periods being 0, 1, 1, and 0. Signal level changes from 0 to 1 and from 1 to 0 are not instantaneous; but for the purposes of digital testing, these transition times can be ignored; it is sufficient to know that the transition has occurred before a specific point in time. Waveform edges can thus be conventionally represented as vertical lines.

The first waveform line shows the signal being directly controlled by the vector data value; the signal takes on the new level (0 or 1) at the start of the period. The second waveform line shows the effect of a timing generator delaying the point at which the signal takes on a new level. The delay time is fixed, so that each edge, whether rising or falling, is always at the same point relative to the start

of the test period. This type of waveform is known as NRZ, or non-return-to-zero, since once the signal has been forced to the 1 level, it stays at that level until the 1-to-0 transition, in contrast to the next waveform (RZ, return-to-zero).

An RZ waveform is clearly a pulse, and the timing generator must provide two times, the delay of the rising edge with respect to the start of the test period, and either the pulse width or the delay of the falling edge. The effect is that a pulse is emitted where there is a 1 in the vector pattern, with no pulse for a 0. The following lines show variations on the RZ waveform: a return-to-one (RO) which gives a negative-going pulse where there is a 0 in the pattern, and an inverted RO, which gives a positive-going pulse for each 0 in the pattern. The remaining waveform of the RZ family shows that a large delay may be programmed, such that the falling edge occurs in the subsequent period.

The RZ waveform can be used for verifying a circuit's response to a positive-going pulse, and the RO waveform, for a negative-going pulse. However, the SBC (Surround-by-complement) allows both tests to be made in one pass. In this waveshape, during the pulse the signal takes on the level corresponding to value of the vector data; but outside the pulse, the signal takes on the complement of that value.

Some other waveshapes are also available, for the SENTRY and for other ATE, but they are generally made up of combinations of the waveforms shown in this figure. In earlier SENTRY models, the waveforms and edges timing could not be altered once the test pattern had been started. In later pattern generators, the waveshape can be altered "on the fly", though the edge timing must remain fixed. As the number of waveform generators is limited, the person writing the test pattern must assign several (possibly many) pins to each generator.

The test pattern itself, written in Fairchild's FACTOR language, contains only ones and zeros; one column of the pattern for each pin of the Device under Test (DUT), one row of the pattern for each period of the pattern generator clock (each "test period"). Each row constitutes one vector. A normal test vector must be preceded by the command "SET F"; all vectors are terminated by a semicolon (thus allowing vectors to overflow a physical line on the page or display).

The pattern-processor memory has only 4096 words, and consequently the test pattern writer must perform careful packing to make use of this limited resource.

There are four constructs which allow some compression of patterns:

- Two or more identical, consecutive vectors may be written as a single vector preceded by "SET FC" plus the repetition count.
- Recurring groups of vectors may be placed in subroutines, and called where needed. The subroutine start is marked by LSUBR, the end by -LEND following the last vector of the subroutine, and a call is made by -LCALL at the end of the vector after which the call is to occur.
- A subroutine can be created with a repetition count, so that one call to the subroutine will result in more than one execution of the vectors that it contains. However, the repetition count cannot be changed: the same group of vectors, with a different repetition count, must form a different subroutine.
- There is also an -LGOTO, which performs a "go to" a pattern address.

Several rules apply to the use of these constructs. The first vector in a subroutine may not be a SET FC, LCALL or LEND. The final vector of a subroutine must have LEND, but

not LCALL. An ordinary (SET F) vector may have LCALL, but only one LCALL. Thus two subroutines cannot be called in sequence without at least one intervening vector.

As well as vectors that contain genuine pattern data, there are also vectors that control auxiliary functions: specifying whether pins shall be in input mode or output mode, and controlling the shape of the waveform on each pin.

The control program connects pins to stimulus generators and response generators before starting the pattern. However, whether the ones and zeroes in the pattern mean "drive a pin to 0 or 1 level" or "compare this 0 or 1 to the actual level on the pin" is controlled by two other types of register: the Drive (a 1 in the D register means that the corresponding pin is to be driven by a stimulus) and the Mask (a 1 in the M register means that the corresponding pin is to be logically connected to a comparator). A 0 in both registers means that the corresponding pin has no stimulus and also is not being observed: it is a "don't care" condition. A 1 in both registers is not usually attempted.

There are in fact two D registers (DA and DB) and two M registers (MA and MB), though at any one time, only one D register and one M register can be active. Of each pair, either the A or B register can be selected by ENABLE MA/MB and ENABLE DA/DB statements, and the contents of each regis-

ter can be set during execution of the pattern with LSET MA/DA/MB/DB pseudo-vectors.

The selection of A or B occurs at the time the test program is compiled, not at the time it is run; the effect is to tag each vector permanently with "MA" or "MB", and with "DA" or "DB". This forms a trap for the unwary programmer. Suppose that there are vectors forming a subroutine, part of a block of vectors in which DB is active. The programmer may elect to call this subroutine from elsewhere, possibly from an environment where DA is the active register and the contents of DB are unimportant. At the instant when control passes to the subroutine, suddenly DB becomes the active register, but the contents of DB will still be those from the calling part of the pattern.

The auxiliary functions determine the waveshape to be used on each pin. There is a register for each function, one bit per pin, and the contents of each register are set by the corresponding LSET vector.

A "1" in the RZ register will transform the waveshape from NRZ to RZ, for the corresponding pin.

A "1" in the I register will invert (complement) the vector data in the corresponding column, for all test periods until the I register bit is set to 0.

There are also XOR (transforms RZ waveform to surround-by-complement), RO (return-to-one) and IX (invert and re-execute previous vector) registers.

Similarly, there are auxiliary functions for controlling the type of comparator strobe.

Provided the tester is running with a test period exceeding about 200ns, the auxiliary instructions appear to take no time, as they are executed between test vectors. But too many (and the rules for "too many" are not well expressed) LSETs between vectors will ~~cause~~ "cycle steal": the test pattern slips by one test period while the LSETs occur, which may cause havoc with verifying the timing of responses from the DUT.

The pattern generator has a pipelined architecture; most registers are loaded two test periods before they appear at the DUT pins. However, the LSET RZ and LSET XOR commands bypass the pipeline, and thus apparently occur two vector rows before they appear in the pattern. It is hard

for a test programmer to keep track of these idiosyncrasies in straight-line code; it is doubly difficult where an LSET of one of these two registers is needed within the last two vectors of a subroutine. Because of the pipeline bypass, the LSET must be placed in the calling program, one or two vectors after the LCALL, to take effect within the subroutine. Similar problems occur at the start of the subroutine.

Finally, there is an additional subroutine rule for LSETS:

- LCALL and LEND may be attached only to LSET IX, not to any other LSET.

This rule means that while LSETS may precede true data vectors in a subroutine, there may be no "dangling LSETS" at the end of the subroutine; if, for example, the subroutine has an LSET MB at the start, the contents of MB cannot be reset at the end of the subroutine, but must be reset in the mainline program after the LCALL.

From this long description, one can see that generating a test pattern in the FACTOR pattern-generator language is a massive task, and when this was the only language available, patterns tended to be short, rarely exceeding 1000 vectors.

2.2 GENRAD'S TPP (TEST PATTERN PROCESSOR) LANGUAGE

The GENRAD GR18 [Genr84] is one of a later generation of ATE for digital systems, and has more facilities:

- The main pattern generator memory still has only 4096 words, but the subroutine and loop facilities have been improved;
- Additional specialized pattern generators are available for testing specific types of circuit.
- The overall test language is a dialect of PASCAL.
- There are 16 waveform generators.

Genrad introduced the concept of "timeset": a timeset is one particular set-up of the parameters of all 16 waveform generators. Each vector has a timeset specified with it; so the timing of edges can vary from one vector to the next. In contrast to the SENTRY, the GENRAD GR18 can vary the timing during pattern execution, but cannot vary the waveshapes; once a pin has had a waveshape assigned to

it, it cannot be changed without stopping the pattern execution and re-loading the timeset tables.

Rather than only the two values 0 or 1, each position in the vector pattern can take one of 16 values, each of which is designated by a single character. The most important of these are 0 and 1 (drive the pin to the 0 or 1 levels), L and H (verify that the pin is at Low(0) or High(1) states), and X (don't care). This technique does away with the LSET MA/MB/DA/DB and ENABLE MA etc., and the other LSETS are done away with simply by forbidding waveshape change since this feature was rarely used.

Each vector may have, as well as a timeset, one or more items of "microcode". The microcode controls the hierarchy of execution; there are jsr (jump to subroutine), ret (return from subroutine), rpt n (repeat a single vector n times), and loop n (loop to the vector flagged with end_loop and perform n times). One can also initialize, increment, and test counters.

The language is "cleaner" than FACTOR pattern language, but still difficult to learn without destroying DUTs in the process.

Other ATE pattern languages are similar in style to these [Hood78, Accu83, LSIL84].

2.3 OTHER PATTERN LANGUAGES

It is occasionally pointed out that there is a specific language for testing [IEEE78]. However, this language (ATLAS) was designed for controlling individual analog test instruments, in particular for avionics, and not for digital circuits with perhaps hundreds of external signals. It is verbose, and its vocabulary is oriented to the analog world; it is totally unsuited for defining functional tests for digital circuits.

Since it is so difficult to write test patterns in the languages provided by ATE manufacturers, there have been many attempts to provide a better language. These fall roughly into three categories: those based on vectors, those based on specifying changes in signal states, and those based on capturing relationships.

2.3.1 Vector-based languages

The idea of a common vector language, one not too far removed from the ATE pattern language, but which could be more easily written and easily translated into patterns for more than one tester, is so attractive that there have been several attempts at it. This intuitively seems a better ap-

proach than writing patterns in the language of one tester and translating for use on other ATE [Have86].

Indeed, a common vector notation specifically for test pattern translation has recently been added to EDIF (Electronic Design Interchange Format) [Marx87]. The EDIF files are in character form. It is possible to write test patterns in that format, and it is probably easier to do this than, for example, to learn the peculiarities of SENTRY pattern generators. The language has the flavor of LISP, with an abundance of parentheses that the test programmer must count carefully.

The vector notation is inherently such a low-level language, that the main-stream of test pattern aids has been to provide higher level languages. A different path has been taken by some, to provide specialized editors for creating and modifying patterns [Gelm82, Chom86]. In spite of the enthusiasm of the authors in extolling the virtues of their approach ("one merely executes the following editor command: PAT 046=01 IN 50 THRU 100"), the data being worked on is still low level, and there is no way of knowing programmers' intent from the data itself, or knowing what is being tested for.

In the mainstream of vector languages, the GTPT language [Chan87] has tried to make pattern notation more ge-

neric and less oriented to specific ATE; but the result still bears a strong resemblance to GENRAD's notation.

The HOLIXZ language [Will85] introduced subroutines with parameters; data values passed to a subroutine of several vectors were substituted into fields in one or more of the vectors. During subsequent translation to the languages of several testers, a separate instance was created for each subroutine for each specific set of parameter values. This language and translation system did allow some capture of design intent; the structure of the source pattern was translated into comments in the ATE pattern, along with the author's own comments.

In an effort to create a standard language for digital functional testing, the IEEE has two committees (IEEE Standards Coordinating Committee SCC20 ATPG Subcommittee and the Test Subcommittee of the IEEE Computer Society Design Automation Standards Subcommittee) working on a Test Vector Language (TVL) [Lowe88]. This language is intended to be used in conjunction with VHDL [Lips86], and has the same general ADA-like "style", which gives it a complex structure. The syntax is not obvious to a person currently skilled in the art of digital hardware design, and this has been seen as a defect by at least one critic of the proposed standard [Rohr87]: "An intuitively obvious language will become ac-

cepted much more quickly than a powerful but obscure language".

All vector languages though suffer from a defect that becomes more important as the number of pins increases:

- There is only one stream of vectors: the writer must account for every pin in every vector. Even if the circuit consists of two entirely separate parts, the writer must combine the two tests into one stream of vectors, using a text editor program.

There is another defect that only becomes apparent when functional tests are written for silicon integrated circuits.

- Every chip is given a final functional test before it leaves the manufacturing plant. However, a functional test is also performed at the wafer stage, before the wafer is scribed and broken up into individual dice, so that defective circuits can be eliminated before they are mounted in the relatively expensive ceramic or plastic packages.

The final test is performed at a clock rate approximating that which the chip will encounter in use. The

wafer test though must use a prober: a fixture that makes contact with each individual circuit on the wafer by pressing probes against its contact pads. Due to the long lead lengths needed for the prober, the wafer test cannot normally be run at the full clock rate, but at some lower speed. The effect of this rate reduction, or lengthening of the test period, can be seen in Figure 3 (above) and Figure 4 on page 27.

At the full test clock rate, the response falls in the test period following the stimulus that caused it;

and this is shown in the first two vectors of the pattern.

When the test period is lengthened, the response occurs at the same delay as before, but now this response falls in the same period as the stimulus; and so the "1" from the second vector of the "response" column must be moved to the first vector. Notice though that if the stimulus had occurred in period 2 of the first diagram (with response in the following period), then when the period was lengthened, the response would still

have occurred in the following period ("1" in the vector following).

To change the test period then, using the vector notation, may necessitate a massive rebuilding of the pattern, using a text editor; but this cannot be done blindly or mechanically; each stimulus/response situation must be considered separately.

The skilled and experienced test engineer lives with these difficulties; but as circuit complexity increases, it becomes more and more difficult for the circuit designer to create even a simple test program.

2.4 STATE-CHANGE LANGUAGES

The vector principle is to divide the relevant time into fixed periods, to define a waveshape for each pin, and then to specify the salient state of each pin for each time period. This approach means that the test programmer must explicitly track the state of each pin at all times, even though a particular pin may never change state during a given test.

The state-change approach is to assume that a pin will remain in a given state unless that state is expressly

changed. The basic notation is more verbose than vector notation, but the redundancy is in a form that makes the test sequence more readily understood by the human reader. Furthermore, constructs such as looping, subroutining, and arithmetic expressions can be added to the notation in a manner that allows their parameters to form part of the pin state control: some examples are shown in Figure 5, though these do not represent any particular language. Most of the statements are self-explanatory; the term "apply" means "apply all changes specified since the last 'apply', and create a vector".

```

Clear all pins;
Set MRESET;          apply;

Clear MRESET;
Do for j = 1 to 36 by 5;
    Set ADDR to j;
    Set ALATCH to 1;    apply;
    Complement ADDR;
    Clear ALATCH;
    Set DATA to 5050'hex; apply;
Enddo;

```

Figure 5. State-change notation

The state-change notation originated as a means of specifying stimuli for digital simulators [ILOG79], but readability plus the prospect of being able to simulate a test

program before running it on ATE lead to its adoption as a test stimulus/response notation. A further advantage was that circuit designer and test engineer could now speak the same language, and indeed one person could begin to fill both roles.

ICTEST [Wats82] was written with this last feature in mind; it was to be used by researchers and students. It was unsuitable for production use as any timing specification was limited to integer multiples and sub-multiples of the test period. Furthermore, it was embedded in the C language, which is frequently not in the repertoire of circuit designers and test engineers, and the syntax does not fit their mind-set; for example, both pins and C variables are given values by the assignment ("=") operator. It did though introduce the idea of concurrent streams of events which could be merged to form vectors.

ANGEL [Snou81] uses the vocabulary of hardware people: clean declaration of pins and registers, "set" and "clear" as in Figure 5 on page 29, and infinitely-variable timing. The published information on ANGEL indicates that it handles only one stream of state-changes, but later versions [Peac87] did handle concurrent streams. The timing information in ANGEL is specified in the form of timing-generator

definition; timing generators must be assigned to pins, and subsequent state-setting statements generate vectors.

A rather similar language, VGEN, available commercially, is being put forward as a common means of creating stimulus/response files for digital simulators [VGEN87]. Parallel streams of events can be initiated with the FORK statement, though there is no facility to JOIN streams until the end of the VGEN run. Curiously, it is not being sold as a means of creating pattern files for ATE, even though its output is a set of vectors. Setting time values is limited to integer test periods.

The disadvantage of state-change languages that have so far been made public is that the methods of timing specification are very limited, and seem to have been derived more from ATE functionality than from circuit specification notation.

2.5 TRANSACTION DESCRIPTION LANGUAGES

Several authors have described methods of capturing the timing and edge dependencies that form part of interface specifications. Some of these are for specification rule checking or even for hardware synthesis[Bori87]. One [Arno85] forms part of an on-going research program for au-

tomatic test generation. A further paper [DenB87] is concerned with capturing edge dependencies: what event causes what other event, so that simulator output can be annotated for easier conversion to ATE functional test patterns.

These papers are all concerned with the capture of transactions that occur at an interface, not with using the transaction descriptions in the creation of test patterns, though the paper by Arnold [Arno85] suggests that this is the ultimate aim of the research.

The need for a standard way to describe the functionality of digital circuits has lead to the development of VHDL, the VHSIC Hardware Description Language [Lips86]. This language uses ADA constructs as a way to encapsulate the functionality of different parts of a circuit. However, there is no support in VHDL for creating test patterns; no facility for describing stimulus and expected-response data. Although ADA and ADA-based languages may in time become widely used, very few hardware designers and test engineers are familiar with ADA, and this is an impediment to the introduction of an ADA-based test programming language.

Other deficiencies have already been pointed out [Nash86]: in particular, for test programming, there is no construct such as `wait` or `wait for` in VHDL.

2.6 DEFICIENCIES

No existing language combines all the concepts that are needed in a means of programming functional tests patterns for complex circuits:

- encapsulation of transactions descriptions, with their timing specifications
- hierarchical programming
- concurrent transaction streams
- full algorithmic capability of a programming language
- clear specification of test data

CHAPTER 3. PHILOSOPHY OF FTSL

A language for test specification should allow the user (test programmer) to describe what tests are to be performed, in the user's own idiom.

To develop a good test specification language then, one must consider how test engineers think.

3.1 CHARACTERISTICS OF TEST ENGINEERS

Test engineers tend to be "hardware people" or "circuit people", as opposed to "software people". When asked about the details of signals on one or more pins, a hardware person will typically grab a pencil and sketch a timing diagram, and will then describe how it works in terms such as these: "If signal A is high, then when signal B rises, then the value from the bus is latched into register C. At the same time, signal D goes low, and after X nanoseconds the value that was latched in gets shifted out by clock Y. When the last bit is shifted out, D goes high again".

Once that has been understood, the test engineer may go on to describe the protocol that uses the interface that was just detailed: "Once transmission starts, bytes are transmitted continuously until the end of block. A block starts

with a SOB byte, then 200 data bytes, then an EOB". That is, consciously or unconsciously, the engineer has a structure, a design hierarchy in mind for the circuit, and the description follows that hierarchy.

In talking about timing specification, the engineer will probably give values when talking about the details of the interface, but will not speak about timing at the higher level, except to comment on response to variation in timing.

Hardware designers and test engineers use signal names rather than pin numbers, but this feature is common to most test languages.

A further characteristic of test engineers (but this does not necessarily apply to other hardware persons) is that they are familiar with Pascal, as most of the modern ATE manufacturers use Pascal, or a derivative of it, as the control language for tester control programs.

3.2 THE TRANSACTION

From the way in which designers and test engineers consider the operation of a circuit, it is convenient to develop the idea of "transaction".

A transaction is defined here as one complete interaction across an interface between two parts of a circuit,

or between a circuit or part of a circuit, and the outside world.

The concept of transaction can be applied at many levels:

- a high level transaction might be a transfer of data between machines: this transaction would include initial handshaking, block transfers, acknowledgement and retransmission as needed, and termination handshaking.
- each block transfer could also be considered as a transaction:
- each block would consist of byte-transfer transactions;
- each byte could be considered as a sequence of transactions, one for each bit;
- finally, each bit-transaction could be defined in terms of signals rising, falling, interacting.

3.3 OTHER LANGUAGE CONSIDERATIONS

3.3.1 Translation

The one advantage of the ATE manufacturers' pattern specification languages is that they can easily be compiled into instructions for the pattern generator hardware. If possible, a new test specification language should also have this characteristic; but ease of translation is a secondary consideration: it is sufficient that it be possible to translate a pattern specification into actual patterns in the more common manufacturers' protocols.

3.3.2 Scope

The language should cover only pattern specification; considerations of voltage levels, maximum and minimum supply current, and loads applied to specific pins (essentially the "care and feeding" of the circuit under test) should be left to the overall parametric test program.

3.4 DESIRABLE LANGUAGE CHARACTERISTICS

Any new language involves an investment in learning, by

those who are to use it. If possible then, the test specification language should be, or should be based on, a language already known to the user.

Knowing the characteristics of those who will write in the language, and the purpose of the language, one can formulate a list of other desirable features:

1. The language must let the user create a hierarchical description of transactions.
 - At the lowest level, the user describes individual signals and signal levels, their effects on each other, and the detailed timing of rising and falling edges.
 - At the highest level, the user may consider an overall transaction consisting only of sequences of other transactions, with no consideration of individual signals (pins).
 - The description of intermediate-level transactions may include both signals and lower-level transactions.

The specification language must allow for a smooth transition between these, rather than having separate vocabularies or a separate syntax for each level.

2. People talk about signals rising and falling and registers attaining specific values at specified times or after specific delays. The language should allow descriptions to be written in this vocabulary.
3. On a functional test, the exact position of an edge is not important, as long as it is within specification limits. The language should allow the timing limits to be specified, rather than an exact time: a compiler or translator program may use this information to optimize the translation to the language of ATE.
4. However, the language, or the translation system, should allow for the general specification of one or more edges to be over-ridden for a specific test.
5. Transaction definitions in the specification language should map easily into timing diagrams, and vice versa.

- the timing diagram is part of the "natural language" of a hardware person; such a person should be able to transcribe a timing diagram directly into a transaction definition.
 - translating the transaction definition into a timing diagram, and examining the timing diagram, is probably more effective a method of finding mistakes, than examining the transaction definition itself.
6. The language must allow users to think about each part of the circuit (or each interface) separately, to write transactions and test sequences for each, and to assemble the pieces into a whole test program. To do this, the language (and translation system) must support concurrency.
- The language should in any case support non-sequential descriptions of edges, so that a user can follow the train of thought for a signal, and then revert, as in the thought "At the same time, this happens:..."

One further characteristic, though not strictly part of the language itself, would be desirable. Each test program has, traditionally, been written individually; any standardizing or copying of code from one test program to another has been strictly informal: it has been particularly rare for code fragments written for testing a lower level circuit to be re-used in the test patterns for the system that uses that circuit.

However, once the transaction description for, say, a Unibus interface has been written, then it can be brought into any new test program as required; and so the language should have an "include" facility.

This on its own though is not enough. Consider a circuit that had two Unibus interfaces. If the "include" facility brings in two Unibus transaction descriptions, then both would have the same signal names; which is undesirable. A macro-processor facility then is needed, so that the signal names (and other symbol names used in the transaction description) can have other names substituted before the "include" occurs.

With these desirable characteristics in mind, the details of a language can be developed.

CHAPTER 4. THE DEVELOPMENT OF A FUNCTIONAL-TEST SPECIFICATION LANGUAGE

PASCAL is the programming language known to most test engineers, since most modern manufacturers of Automatic Test Equipment (ATE) use PASCAL, or a derivative of it, as the control language for test equipment. FTSL (Function-Test Specification Language) then is based on PASCAL. Some features of FTSL are in the form of changes to PASCAL, but most are additions to that language.

4.1 TRANSACTIONS, PROCEDURES AND FUNCTIONS

PASCAL allows the user to segment a program into sub-programs of two types: procedures and functions. In each of these, there may be computation or input-output operations or both.

In a test program, there are also operations on signals: setting signals to specific levels, or verifying that signals are at specified logic levels. During the translation of a test program to ATE language, a procedure or function that only involves computation, not operations on signals, can be handled more simply than the general test sub-program. It is helpful then to add a special category

of sub-program: one that does have effect on signals, or which invokes other sub-programs that affect signals. Sub-programs in this category are in fact those that describe transactions, and so FTSL has three types of sub-program: procedures, functions, and transactions. Transactions can invoke procedures, functions and other transactions; but procedures and functions cannot invoke transactions.

4.2 PINS

A data type of PIN is needed, to represent the signals that cross the interface between the circuit to be tested and the outside world. This could be defined using the standard PASCAL type-definition notation. However, in the typical circuit specification, such as the data sheet for a commercial integrated circuit, the pins of the circuit are described in a tabular fashion.

A test specification language should have a facility for pins to be specified in this way, since the pin-definitions can be transcribed directly from the data sheet.

In FTSL, the table has three columns: the name of each signal, the type of signal, and the number of the pin on which it appears. Punctuation and the pin-section keyword "pin" have been added, to give the flavor of PASCAL to the

table, and to enable a compiler to recover gracefully from syntax errors.

In the typical data sheet, every pin of the circuit or device is included in the table: FTSL also should allow every pin to be defined, even though some, such as power and ground and "no-connect" pins, do not take part in the actual tests.

In practice, it is not uncommon for a signal to appear on more than one pin; that is, some pins on the interface may be connected to each other. The third column of the pins table should therefore allow for more than one pin-number to be associated with a signal.

Pin definitions are global to every test in the program, and so the "pin" section may only appear in the main test program, and not in any sub-program.

4.2.1 Signal levels

The purpose of a functional-test programming language is to provide a convenient way to specify stimuli and expected responses from the digital circuit.

The stimuli consist of specific signal levels applied at specific times. The stimulus levels are of course 0 and 1, and the absence of signal level, known variously as "tri-

state", "Z-state" or "high-impedance". There are occasions though, especially in initialization sequences, when the test programmer wishes a signal to be driven to a level rather than being tri-state, though it does not matter whether 0 or 1. The usual practice is to assign an arbitrary 0 or 1 to the signal. However, specifying the level as "don't care" (X) rather than a specific level, would allow a post-processor program to assign either 0 or 1, which ever would result in the most compact resulting test pattern.

Expected-response levels are obviously 0 and 1, plus the Z (tri-state) "level"; but any description of expected-response must though include not only specific levels and when they are to be expected, but also the times when the level of the signal is immaterial: that is, the language must allow the "don't care" (X) state.

4.2.2 Forcing levels and expecting levels

In a test program, pins can take on values, but pins are not the same types of objects as programming variables such as integers and character strings. Setting a pin to a specific level results in action in the real world; control is applied to power supplies and pattern generators. It

seems appropriate, in a test programming language, to emphasize the difference by using something other than a simple assignment statement to force pins to specific levels; and in FTSL, the keyword **force** is used. (In common parlance, a signal can also be "set" to a level; but the word "set" has other meanings in programming languages). Thus an example of setting one signal, or pin, to the 0 level would be:

```
force RST, 0;
```

Similarly, to expect a level on a pin is to set comparator circuits in the real world, and so, rather than the "if A = 1" style of a computer programming language, the specific **expect** statement is used in FTSL.

In the design of a programming language, a balance must be achieved between the verbose, as in COBOL ("SUBTRACT TOTAL-PERSONAL-EXEMPTIONS FROM NET-ANNUAL-INCOME GIVING TAXABLE-INCOME") and the concise but cryptic, as in APL or C ("for (;*i++!=\0;);"). In FTSL, additional keywords could be added to the **force** and **expect** statements to make them read more smoothly, as in

```
force RST to 0;
```

and

```
expect ACK to be 1;
```

or the "semantic sugaring" could be omitted altogether or a comma could be used. Typing superfluous words could be a great nuisance, and yet in reading statements such as these aloud, there seems to be a natural break between the signal name, and the value. FTSL has a comma in this position rather than either a keyword or no punctuation at all, since experience with a previous language [Will85] has shown that test programmers tend to insert a comma at a "natural" break, whether that the language specification calls for it or not.

Quite frequently, more than one signal is to be set to a new value at the same time. While separate force statements could be used for this, it is more "natural" to have one statement to change all signals that are to take on the same signal level, simultaneously. A suitable syntax can be derived from the simple statement by adding parentheses:

```
force (RST, MRST, CS), 0;
force (DIN, ADDR), 1;
force (BUS), Z;
```

while retaining the original form for single signals:

```
force BUS, Z;
```

The pin lists in force and expect statements can still become very long when dealing with circuits that have many

pins, and some additional form of shorthand notation is needed. In many operating systems and shell languages, "wild-cards" may be used. This notation allows a specific character (normally, the "*") to stand for "a string of zero or more arbitrary characters". In this notation, one might write a force statement for RD0 through RD15 as "force RD*, 0". But this technique suffers from two disadvantages:

1. It casts too wide a net: "RD*" would pick also pick up "RDR_OK" if there were such a signal, without any way of warning the user.
2. The "*" character is frequently used to designate an inverted output, as for instance a flip-flop having Q and Q* outputs. It would be possible to have a facility for redefining the wildcard character, but this would be a further less-than-obvious hazard for the programmer.

The abbreviation method proposed here is to have a register definition facility, and to allow simple register arithmetic.

4.3 REGISTERS

In any but the simplest of circuits, the interface will have sets of signals (pins) that have some sort of affinity, such as the signals of a bus, or a set of control signals. It is convenient to group such signals into a conceptual register, so that they may be manipulated by single commands. For example, all signals of a register could be simultaneously forced to the logic 0 level; or the logic levels of the signals in a register could represent a binary number.

It is also useful for a signal to be allowed to be in more than one "conceptual register". Consider a set of signals that form a data bus: at one instant the whole set could represent data, a signed binary number; while a short time later, the signals could represent an un-signed address, while later still, an instruction consisting of an operation-code plus a base-register number plus an offset value.

The user should be able to represent values in the most natural form, and so FTSL should allow a signal to be assigned to more than one "conceptual register". For example, signals DB[0] through DB[15] might be assigned to register

DATABUS, while DB[0] through DB[3] might also be in register OPCODE.

There is obviously potential for conflict here, and while it should be the user's responsibility to use the registers in a non-conflicting manner, an FTSL compiler should check for conflict, such as a signal being simultaneously driven to high and low logical levels. Such a conflict could only arise if two transactions, each making force reference to the same signal, were simultaneously active; and, at the risk of overloading the user with warnings, it might be desirable to warn the user of this circumstance, even if both force statements were setting the signal to the same logic level.

In keeping with PASCAL style, the register declaration section will be introduced by the keyword "register", and terminated by encountering any other keyword. The flavor of PASCAL can be retained in the body of the register declarations: for example,

```
register
  RD[0..7]: RD0, RD1, RD2, RD3, RD4, RD5, RD6, RD7;
```

Register definitions, like pin definitions, will generally be global to the whole test program; but normal PASCAL

scoping rules will allow the local redefinition of a register within a transaction definition.

The technique of defining a register for every set of pins to be used in force and expect can be overdone, and can then obscure the meaning. As an example, consider a "walking zero": all pins of a register are to be set to 1, and then each in turn is to be set to 0. It would be possible to define a register for each of: all pins, all pins less the first, all pins less the second, and so on. But in this situation, it would be far clearer to define ~~just one~~ register, and to use an expression that described each of these cases for the respective force. Using the RD register just defined as an example, the sequence could be written as:

```

force RD, 1;
      . . . (other statements)
force RD, 0;
      . . . (other statements)
force RD-RD0, 1;
      . . . (other statements)
force RD, 0;
      . . . (other statements)
force RD-RD1, 1;
      . . . (other statements)

```

and so on.

These expressions are really logical ("and" with the complement of the signal to be omitted) rather than arithmetic: but the "-" operator (pronounced "less": "RD less RD1") is in keeping with common parlance.

4.3.1 Register levels

The individual signals of a register could be "forced" to specific levels, or specific levels could be "expected" from them. But it is much easier for the test engineer to think of the register as a whole, treating the signals as an ordered collection of bits, and to force and expect integer values representing binary numbers.

There is an advantage in allowing expressions as well as constant integers as the desired level in force and expect statements: for example, consider the testing of an adder circuit. The test engineer will wish to force two values on to the input interfaces (or successively onto the one input interface) and then compute the sum of the inputs, and expect that result from the output interface.

No matter how derived, the values that appear in "force" or "expect" must eventually become binary integers, one bit per pin, though those values could have been computed from a floating point or character representation.

They could also be symbolic, predefined values. In testing a computer CPU or similar processor circuit, the test engineer would find it useful to call operation codes by their symbolic names, rather than by their numeric equivalents; and so a test programming language should have a

symbol-definition facility. Fortunately, the Pascal language has the `const` declaration, which allows a name to be given to a constant value, or to the result of a constant expression.

4.4 TIMING: SPECIFICATION OF EVENTS

A test specification defines stimulus levels, and the times at which these levels are to be supplied: plus the expected response levels, and the times by which these response levels should have been attained. The timing diagrams which form part of circuit specification generally show the timing of events (an event being a signal edge: the attainment of a new level by a signal) as symbolic time values. The values of the time symbols are later specified in a table, generally as some or all of minimum, typical, and maximum values. Common examples of timing symbols are T_{setup} and T_{cwmin} .

- T_{setup} is the interval for which an input signal must have held a stable level, prior to the rising or falling edge of the strobe that latches the signal level. This symbol would probably have one limit value specified (the minimum) as well as a typical value for the guid-

ance of potential users. In effect, the designer of the circuit guarantees that as long as the Tsetup condition is met, the circuit will work properly. This method of specification is used whether the circuit in question is a printed circuit board, an integrated circuit, or even a standard cell to be used by designers of semi-custom integrated circuits.

- Tc_{wmin}: minimum clock width. For a clock signal, two parameters must be given: minimum high width and minimum low width, or the maximum clock frequency (or minimum period) and one of the width parameters.

The values of these timing parameters would be used when creating a test program.

- For a functional test, the "typical" values would be used, or, if no typical values were given, then values would be derived from the "maximum" or "minimum" limits, relaxing them to meet other constraints such as the maximum frequency at which a specific test fixture could operate.

- For timing tests, the actual minimum or maximum limits would be used.

4.4.1 Setting values of timing symbols

A functional-test language should let the user enter the timing description in the conventional form, as far as possible. A fragment of a timing value table might be:

parameter	min	typ	max	units
Tsetup	100	150		ns
Tcwmín	30			ns
Tcperiod	60			ns

A suitable syntax for the user to enter these values would be:

timing

```
Tsetup 100ns / 150ns / ;
Tcwmín 30ns / / ;
Tcperiod 60ns / / ;
```

This is unambiguous even without entering any trailing "slash" characters; but if only a maximum value were to be given, the "slashes" would be needed to act as "placeholders" for the empty columns.

In describing the operation of a circuit, or in specifying how it is to behave, there are two principal points at which timing symbols get their values defined:

1. If the circuit contains standard interfaces, such as the DEC Unibus, or is to operate to standard protocols, such as the RS232C serial communications protocol, then the time values will be set in the document, or part of the document, that defines that interface or protocol;
2. For other parts of the circuit, all timing symbols will generally be collected and have the values defined in one part of the document.

However, timing values set in a protocol document may be over-ridden or augmented in a circuit specification: a very common example is to specify the use of the RS232C interface, but then to set a maximum baud rate.

A test specification language must cater to these situations.

- It must allow standard interface specifications to be packaged as standard routines that can be included in the overall functional test program.

- It must allow the specification timing values to be encapsulated with the routine, but allow some or all of these values to be over-ridden or augmented in a particular test program.

This re-definition approach is quite the reverse of the "scoping" rules of most computer programming languages. In those languages, as for instance in PASCAL itself, a symbol defined in an "outer layer," may well be redefined in an inner subroutine. In circuit specification, a timing value (and this only seems to apply to timing values) defined in an inner specification may well be redefined in the overall (outer) specification.

The latter method, which is that incorporated into the FTSL language, is a better match to the hardware design process. Standard interfaces can be programmed as library routines, and the normal user need never consider the content of each routine; yet specific values can be given to the timing parameters if needed.

The principle of allowing an outer procedure to "reach into" an inner one is not unknown in programming technology; in fact a constrained version forms part of the ADA package concept. However, the principle has generally been applied to procedure entry points; here, it is applied to timing.

variables. The method proposed is to have an optional second parameter list on a call to a library routine; the second list would specify the actual values of any timing parameters whose values needed modification. For example,

```
transmit(27);
```

would invoke the "transmit" protocol with a value 27 to be transmitted with standard timing values; while

```
transmit(27)(BitClock:=300ns)
```

would invoke the same protocol, but would over-ride the internal definition of "BitClock". A refinement is to allow the use of the words `min`, `max` and `typ` in expressions in the second parameter list. Each word would stand for the corresponding value in the timing table set within the library procedure. For example,

```
transmit(27)(BitClock:=min+200ns);
```

4.4.2 Using timing symbols

Timing symbols that have been defined can be used with "force" and "expect" phrases to describe the behavior of the signals, and other transactions, involved in a transaction.

A test specification may contain phrases such as "when the REQUEST line becomes 1, the circuit will acknowledge by raising the ACK signal within 100ns". A test programming

language should of course let the user describe tests in this manner. However, the phrase just quoted is also describing a dependency: one event controlling another. Taking it a little further, the description of the levels on a signal may be used to synchronize two parts of a test, as well as defining the levels to be forced on or expected of the signal. The designer of a test program may, for example, think "when SYNC rises, start the transmit-test", where "transmit-test" might be a sequence of tests already defined; that is, a major transaction (which itself could have been created from other transactions).

In considering a language construct for describing this type of situation, the word "when" appears somewhat unsuitable, since it carries a slight connotation of "whenever". A better method is to transform the thought into "wait until SYNC rises...." and hence FTSL should use a made-up word waituntil.

It is perhaps somewhat surprising that the "expect" values may be used in a "when the level rises..." expression. There is no known commercial ATE that would allow part of a test to be synchronized to the change in level of a signal output by the device (or circuit) under test. However, in the specification of the test, it is not the real signal level coming from the circuit that is being used in a

synchronization expression, but the value to be expected: and this value has been specified, as to both signal levels and times at which the levels are to be tested for (i.e. "expected"), by the author of the test program.

4.4.3 Flags

There can be situations where parts of a test program must be synchronized, but there is no signal that can act as the control. For example, where a piece of hardware contains two entirely separate circuits, there is no common signal; yet the test engineer may wish to have the test for one circuit start at a specific point in the second circuit's cycle. To handle this, a "pseudo-signal" is needed. The flag fills this role. It can be forced to 0 or 1, and can, like a pin, be used in a `waituntil` statement.

For consistency with the usage of pins, and so that all inter-transaction synchronizing agents can be seen at a glance, flags should be declared at the head of the test program, and be global to all transactions; but there seems no reason to make this an absolute requirement, merely a recommendation to users.

4.4.4 The concept of "present time"

In a conventional computer programming language, there is no concept of the time at which a statement occurs; there is a concept of statements being executed sequentially.

In a functional-test language, statements are still executed sequentially, but a statement may be an instruction for an event to occur at a specific time, and these times may not be sequential.

Figure 6 shows a fragment of the DATI cycle in the DEC Q-bus timing diagram. Some edges are defined with respect

to the rise of SYNC, but others, like the rise of RPLY, are defined with respect to an edge which is itself defined with respect to SYNC.

As the eye moves across a timing diagram, the reader (or writer) of the diagram establishes each edge in turn as a base; and edges controlled by that one are thought of as "at time T, (some other edge occurs)"; the thought that is implied, but not stated, is "at time T with respect to the edge I am now considering, (...)". In the mind, the "present time" is the time at which the base edge occurs; and as the eye advances to the next edge under consideration, the "present time" advances to the time of that edge.

At some point of course the eye backtracks to a previous edge, and from there picks up and follows another chain of edges.

A tree is built up in the mind, as "at time T1, X occurs; at time T2 (with respect to X), Y and Z occur" and so on. The constructs of a test language must allow expression of this tree in a form close to the thought process.

In FTSL, the *at* specifies when an event is to occur with respect to the present time, and advances the present time to the time of that event.

The backtracking situation means that two or more events are specified with respect to one edge; the time de-

lay for each of those events begins at the same instant; the at statements for those events are therefore concurrent. This thought can be expressed directly in a functional-test language; in FTSL, any at statements between concurrent and the corresponding endconcurrent are deemed to start at the same instant.

The introduction of the concurrent facility helps to solve another problem. There are situations where more than one event is dependant on one edge, and each has its own time limits, with respect to that edge. The concurrent statement allows a clean representation of this in the case of simple actions:

```
concurrent
    at time-1 thing-1 happens;
    at time-2 thing-2 happens;
    at time-3 thing-3 happens;
endconcurrent
```

The construction is not so clean when each action is longer than one line:

```

concurrent
  at time-1 BEGIN
    thing-1 happens;
    thing-1a happens;
    thing-1b happens;
  END;

  at time-2 BEGIN
    thing-2 happens;
    thing-2a happens;
    thing-2b happens;
  END;
  at time-3 thing-3 happens;
endconcurrent

```

As each Begin-End pair gets longer, it becomes harder to remember that the third at clause is referenced to the same instant as the first. A different construct is needed: the method proposed here is to label a particular time, using the Pascal label facility. An example will show the use of a label as a current-time marker:

```

waituntil RDR_OK rises;

OK_edge:

  at time-1 after OK_edge
    BEGIN
      thing-1 happens;
      thing-1a happens;
      thing-1b happens;
    END;

  at time-2 after OK_edge
    BEGIN
      thing-2 happens;
      thing-2a happens;
      thing-2b happens;
    END;
  at time-3 after OK_edge thing-3 happens;

```

Once the concept of a label has been introduced, there is the danger of unstructured references to that label, analogous to unstructured GO TOs. Rules are needed therefore for the use of labels:

1. A label may only be referenced after it has been defined;
2. A label may only be referenced from within the program block in which it is contained;
3. If a program block contains one or more labels, then all statements in the block must use the "after" clause.

The word **after** is not ideal, as it is possible for the delay-time to be negative. There seems to be no short word that has the needed meaning, and so **wrt** is proposed, meaning "with respect to". At first glance, a made-up word strikes a discord, but there are precedents: there are languages that include "fi" and "esac" in their lexicon.

4.4.5 Clocks and continuous processes

It would be very annoying for the test programmer to

have to specify every edge, or even every cycle, of a clock signal.

A simple way to avoid this is to introduce a **forever** statement into the language. But this introduces the problem of stopping a **forever** statement! Fortunately, a simple rule will suffice, since there is no situation in which a **forever** statement exists on its own. There will always be at least one concurrent statement or stream of statements.

Even the extreme case of a circuit being "kept alive" by a clock alone, will need some concurrent controlling statement, otherwise the ATE will never stop. Thus a basic stopping rule is:

At the end of each pass through a **forever** statement, examine all statements that are concurrent with it. If any concurrent statement forms a transaction stream that is still running, then re-start the subject of the **forever** at its beginning; otherwise, end the **forever**.

For example:

```
concurrent;
  forever xmit_clock;
  lengthy_xmitter_test;
endconcurrent;
```

Here, each time the "xmit_clock" comes to the end of its cycle, if "lengthy_xmitter_test" has not ended,

then `xmit_clock` is restarted at the begining to give a new clock cycle.

However: this rule does not work for two or more concurrent clocks! A modification to the rule is needed for this case:

At the end of a pass through a `forever` statement, if any non-`forever` concurrent stream is still running, then restart the subject of the `forever` at its beginning; otherwise, end the `forever`.

For example:

```
concurrent;
  forever xmit_clock;
  forever rcvr_clock;
  lengthy_xmitter_test;
  lengthy_receiver_test;
endconcurrent;
```

In this situation, once the two tests have completed, the first clock to complete a cycle will find that there is no non-`forever` stream running, and will therefore not produce any more clock cycles. Finally, the other clock transaction will end, find there are no parallel transactions, and the whole `concurrent` statement will end.

4.5 SPECIFYING TEST DATA

The specification of a test sequence normally requires the specification of data values to be applied during the test. Some values can be explicitly stated in command statements, as in "force (RESET, 1)", but generally, a test specification sets up the conditions for a test, and then applies the test using explicit data values.

In a programming language such as PASCAL, a sequence of data values to be used in a procedure could be generated by a for statement, or a sequence of calls to the procedure could be made, each with its explicit data value. Either of these methods could be used in a test specification language, but there are considerations which make another method desirable.

- For a specification, it is more convenient to have test data values shown "up front", grouped together in the test document, rather than distributed throughout the description.
- A stream of apparently independant data values is frequently needed, such as values obtained by consideration of the intended use of the circuit. Since these values

are independant, the test programmer cannot create an algorithm to generate them, and each value must be explicitly specified.

- The stream of data values may derived from a file of data, and it would be desirable to have the data access method reflect the "style" of file access.
- A circuit designer, wishing to apply a stream of data values to a prototype circuit (a "breadboard") will program a waveform generator or data generator and connect it to the appropriate pin of the circuit. A test data access method that followed this way of thinking would make it easier for circuit designers to write test specifications.

There are actually two parts to this subject:

1. a method for specifying the data values
2. a method for presenting them, one at a time in sequence, to force or expect statements.

4.5.1 Data definition

The method proposed is to have a `testdata` section in the preamble to the test specification, along with the `var` and `timing` sections. In this section, data generators may be defined, with the stream of values that each is to produce. A data generator, as used here, could produce any type of data, and would not be limited to integer values. Non-integers would be converted, in the body of the test program, to integers for use in `force` or `expect` statements.

An incrementing or decrementing construct would be desirable here, in addition to the obvious list-of-data-values construct. The simplest is to use an extension to the PASCAL "for" method:

```
msg_no := i to j
or msg_no := i to j by k
```

and to allow this to be imbedded in a list:

```
msg_no := 1, 5, 3, 9 to 17 by 2, 1, 5, 3;
```

The values generated by this statement would be:

```
1, 5, 3, 9, 11, 13, 15, 17, 1, 5, and 3.
```

4.5.2 Data access

There are three functions that must be performed on the data stream emanating from a data generator:

1. Get the next item of data in the stream
2. Query the data generator, to know when the last item has been obtained from the stream
3. Reset the data generator, so that the next item obtained is again the first one in the stream.

In keeping with the PASCAL theme, there need to be functions to perform each of these;

1. GetData to return the next item
2. LastData and EOD to indicate whether the end of the data list has been reached;
3. ResetData to reset the data list so that on the next GetData, the first item in the list will be returned.

There are two functions for querying the end-of-list, because there are two conventions, or ways of thinking, about the subject.

LastData returns "true" if the item just obtained was the last one in the list;

EOD returns "true" if the item just obtained was invalid, because the item previously obtained was the last item in the list.

EOD is provided because of analogy with the PASCAL "EOF" function for querying whether the end of a file has been reached; LastData fits in better with conventional thinking.

4.6 PASSING DATA TO TRANSACTIONS

In a transaction definition that has data passed to it as a parameter, there is the possibility, in a system that allows concurrency, of the data value varying during the time that the transaction is active. In particular, the value of a parameter could change in the interval between the transaction being called, and the data being used. This

could well happen if a transaction description included a `waituntil` before the data were used.

From another point of view, the value of the parameter at the time the data was used (for example, in a `force` statement) might not be the same as at the time control entered the transaction (that is, at the time the transaction was called). Generally, the value needed is that at the time the parameter is used; as in setting a register to a specific value after a specified delay.

Global variables could be used rather than transaction parameters, but this would nullify one advantage of a Pascal-like language: the names to variables given when creating a transaction need not be the same as those given when the transaction is used.

Standard Pascal has two mechanisms for passing parameters:

- The pass-by-value mechanism takes a copy of the value at the instant of the call.
- The pass-by-reference (VAR) mechanism takes a copy of the address of the data, at the time of the call. Should the value vary, the transaction would still refer to the latest value.

The VAR mechanism will usually be what is required, though both should be available to the user. There is a problem with using VAR; since a value may be passed back to the caller, the entity in the calling sequence MUST be a variable; it cannot be a constant or an expression. This rule would be acceptable for many applications, but not for all; it could cause artificial sequences such as

```
data := 3;
shift_out(data);
data := 5;
shift_out(data);
```

rather than

```
shift_out(3);
shift_out(5);
```

There is a third mechanism in the IBM dialect of Pascal [IBM81], known as the "pass by read-only reference", in which the address of the value is passed, but the called routine is forbidden to change the value. This method will allow a transaction to obtain the actual value at the time it is needed, as well as permitting constants and expressions in the call; it will be adopted in FTSL. Parameters to be passed by this mechanism are introduced by the keyword **const**.

4.7 TEST PROGRAM

Since FTSL is so similar to PASCAL, there is the danger that a user may use an inappropriate compiler: a PASCAL compiler for an FTSL test program, or the test program compiler for a PASCAL program.

A way is needed for the FTSL compiler to check quickly that the program that it is compiling is in fact in FTSL language and not PASCAL.

In FTSL therefore, every test program should begin with "Test Program . . ."; whereas in PASCAL, every program begins with "Program . . .". This allows the FTSL compiler not only to verify that an FTSL program has been submitted, but also to give a very specific error message to the user if not.

4.8 THE LANGUAGE

Appendix A forms the specification of FTSL (Functional Test Specification Language), a language that has been developed to meet the constraints and requirements set out in this and the preceding chapter. The language is based upon PASCAL, and supports concurrent processes, but it is not Concurrent Pascal [Brin77]. The latter allows for the se-

quencing of processes relative to each other, but not for the specification of the relative timing of processes. Further, Concurrent Pascal does not have the facilities for the specification and control of stimulus and response signal levels.

The aim of the present thesis is to define the specification of the language, rather than writing the software to use and translate the language to ATE. However, as part of the process of refining the definition, a functional test of a UART (Universal Asynchronous Receiver Transmitter) has been written in FTSL: this forms Appendix B.

FTSL uses the vocabulary of the hardware (circuit) designer: it encourages the encapsulation of interface timing diagram fragments as protocols, or transaction templates: and it allows the person writing the test pattern to work at the most comfortable level of abstraction.

4.9 TRANSLATING FTSL TO ATE LANGUAGE

A test program written in FTSL can be considered as a set of instructions for an ideal tester. Since practical ATE is not ideal, the FTSL instructions must be translated into the language of a real tester. Two methods for the translation are proposed here, to demonstrate that test pro-

grams written in FTSL can indeed be translated into the language of suitable ATE.

- One method is concrete, making use of known techniques.
- The other method is more speculative, and is given here as a suggestion for future research.

4.9.1 Using known techniques

The route from a functional-test specified in FTSL to the same test in the language of particular ATE could be direct, or it could be via an intermediate list of events.

Figure 7. Translation from FTSL to ATE language

The translation can be performed in the following steps:

1. "Execute" the FTSL program, creating a list of events plus an interpretation of the test program structure.
2. Apply rules to the list of events, to "regularize" it into test periods.
3. Apply rules to reduce the number of timing generators required.
4. Use the structure given by the procedures and transactions to create subroutines and repeats from the list of events.
5. Write out the list of events into a test pattern, using the syntax of ATE.

Commercial software is available for obtaining test patterns in ATE languages from an event-list [TSSI87]. Dealing with the thousands of events is a massive data-processing task, and typically such programs are expensive in CPU usage, and may also need "hints" from a human as to the best allocation of ATE resources. Nevertheless, a program of this type would perform steps 2 through 5 above, so

the problem is now reduced to creating an event-list from the FTSL test specification.

"Appendix D. Creating an Event-List from an FTSL Program" on page 160 covers this in some detail, giving a possible data structure for each construct of the FTSL language. It can be summarized here as:

- Treat the FTSL program as a set of co-operating parallel processes, with an "operating system" to control them;
- Load the whole test program into memory, representing the test program as a static data structure.
- Execute the test program from the static data structure, adding a dynamic data structure to it to represent the course of the execution.

The end result will be a file of events, which can be used as input to the ATE-program generator.

4.9.2 A more speculative method

The method just described -- event-list -- is akin to unrolling the FTSL test program into its component edges,

and then re-creating the structure from the pieces, for use in ATE.

A human being would not follow this process. A person would adopt a more analytic approach, to reduce the massive data-processing required.

The structure of an FTSL program can be regarded as a three-dimensional network: the calling-sequence components form a tree, and dependencies form cross-connections.

The analyst would use this type of diagram, whether as a mental image or as a more formal Gantt-like chart, to determine

- the length or overall elapsed time of each type of transaction;
- the start time of each instance of a transaction.

There are three constraints that would prevent immediate computation of transaction width:

1. dependencies on external (to the transaction) flags and signals; in particular, a transaction being dependant on more than one external event.
2. dependencies on data values
3. unknown widths of subordinate transactions

Dealing with each in turn:

1. A transaction that has several dependencies can be transformed into a set of new transactions, each of

which is in "standard form"; a dependency (which may be null) on some external flag or signal, followed by the edges and events that depend on it.

2. Data values cannot alter detailed timing; a data value can only control looping, or whether a specific subordinate transaction is to occur or not. Any transaction that has a data-dependant width can be split into new types of transaction: one for each of the possible outcomes.
3. The width can be computed directly for the lowest level transactions, that depend on no others. These values can be used in computing the widths of transactions that call them.

Thus the human approach to creating the ATE form of the test program is:

- static analysis of each type of transaction
- top-down creation of structure network diagram

- bottom-up synthesis of higher and higher level meta-transactions until the human arrives at the completely re-synthesized test program.

The process thus far is reasonably straightforward, and can be reproduced as a computer program. Because this approach analyses one transaction and applies the results to all transactions of the same type, it should require less computation. However, even though all the timing has, at this point, been determined, transforming the network into structured test vectors (akin to adding the bar lines to the musical notation of a free-flowing madrigal to transform it into a formal song) is more dependant on applying heuristics, and a rule-based system would be more appropriate.

Several aims need to be met: attaining any one has an effect on the others.

- reducing the total number of vectors to be executed by the ATE by using the longest possible test period. To take a simple example, if two transactions triggered each other to produce the timing shown in Figure 9 on page 84, then it should be possible to double the period and halve the number of vectors (as in Figure 10 on page

84): but it could mean that a separate timing generator would be needed for each signal, rather than one for both.

Figure 9. Doubling the test period: before

Figure 10. Doubling the test period: after

- reducing the memory requirements by preserving, as far as possible, the subroutine structure of the source test program; since ATE generally does not support passing parameters to subroutines, this would entail creating a

unique instance of each subroutine for every set of data values, and then reducing the number of instances by looking for "don't care" data and signal level values.

- reducing the number of timing generators needed by moving edges within the test period, while keeping the edges within the timing values specified in the source test program. In Figure 11, if the maximum allowed value of t_2 , as set in the timing section, were greater than t_1 , and the minimum allowed value were less than t_1 , then the actual value used for t_2 could be the same as for t_1 , and signal B could be assigned to the same timing generator as signal A.

Figure 11. Moving edges

The creation of such a system is beyond the scope of this thesis; but it appears to be a promising approach.

2

MicroD

CHAPTER 5. VERIFICATION, CONCLUSIONS, AND FUTURE WORK

A language, FTSL, has been defined to meet the criteria laid out in Chapter 4; the language definition forms Appendix A. The language allows circuit designers and test engineers to write functional test patterns using vocabulary and syntax that is consistent with the "hardware" way of thinking, and that can be translated into vector patterns in ATE languages.

To demonstrate that one can indeed write functional tests in the language, a test program for a UART (Universal Asynchronous Receiver-Transmitter) is shown in Appendix B. This device incorporates two almost separate circuits: receiver and transmitter; and each has two parts that operate, to a certain extent, asynchronously. In the transmitter, the buffer register can be loaded at any point in the cycle up to the time at which the data is required; and in the receiver, once a byte has been received and stored in the output register, it can be removed at any time before the buffer is needed for the next byte.

This circuit forms a good test of FTSL; its specification sheets are reproduced in "Appendix C. Circuit Specifications for the UART" on page 154.

The discussion that follows is not intended as an exposition of the functional test, but of the way in which features of FTSL contribute to making an understandable test program.

For the example, the test to be performed should verify that the UART performs correctly for no parity, even parity, and odd parity; for 5, 6, 7 and 8 data bits; for 1 and 2 stop bits; and for bytes presented individually, and as a continuous stream.

Following the style of Pascal programs, a test program written in FTSL will be described bottom-up, with the lower-level transactions being described before the high-level ones. Most of the transaction descriptions bear a close resemblance to the corresponding timing diagrams given in the manufacturer's specification sheet for the UART. Each line of each transaction has been numbered for reference here; these numbers do not form part of the language.

The first part of the example tests the transmitter; a receiver test will be added later. The first transaction, `xmit_clock` on page 143, defines the transmitter-clock. The one signal involved, TCP, is a pin of the device under test and so its declaration is in the preamble to the test program, not in the `xmit_clock` transaction. The period of the clock, and the "low" time, are set in lines 22 and 23. These

two, taken together, define the high time, or duty cycle, of the clock.

The concurrent/endconcurrent statement pair (in lines 29 and 32) illustrates an alternative use of this construct: not so much to define parallel streams of transactions, but so that the timing of both edges of the clock can be specified with respect to the start of the transaction.

The period of the clock is also the width of the transaction: the current time at the start of the concurrent statement in line 29 is also the time of the start of the transaction, and the final event of the transaction (forcing TCP low at line 31) is after a delay of TCPperiod. This transaction corresponds to Figure 9 in the UART specification sheet.

The control_strobe transaction on page 144 does not correspond directly with any one figure in the specification sheet, but it can be synthesized from Figure 8 in the UART specification and the timing given for CS.

In the UART specification, CS may be a level or a strobe: a strobe is used here, as it is the more stringent requirement. To verify that the circuit is insensitive to changes in control signal levels when CS is low, the four control signals (NP, EPS, TSB, and NB) are forced to their

complements (in lines 39-42) after their current values have been strobed in.

The next transaction, `load_byte`, does two things; it causes the DUT (device-under-test) to load its input register with the value of the byte to be transmitted, and it also saves that value so that it can be used later for verification that the correct value is received serially from the UART. The interlocking between loading a byte and transmitting it, is the flag `TBMTflag`. The signal `TBMT` forms part of the the specification timing diagram: the flag `TBMTflag` is not part of the specification, and does not appear in the timing diagram of page 145.

`Load_byte` waits (in line 27) until the flag is 1, then loads the byte and forces `TBMTflag` to 0.

`Output_byte` (on page 148) waits until the flag is 0, then starts to output the byte, and sets `TBMTflag` to 1.

There is no possibility of deadlock, as `TBMTflag` can only take on two values; if it is 0, a byte is loaded; if it is 1, a byte is output.

The `load_byte` transaction also illustrates the use of a negative-time delay, at lines 29 and 34. There is no

`waituntil` statement that can be triggered from either of these cases, and so it is safe to use the negative time, as described in "At statement" on page 123. In effect, this construct sets a reference time for the transaction at the rising edge of the DS pulse, which is a "natural" position for it.

The `output_byte` transaction illustrates how lower-level transactions can be assembled to form a higher-level one: here, each bit of the byte being transmitted is output separately. The first bit has a unique type of transaction defined for it, rather than the general-purpose "`output_bit`" transaction, as it has to synchronize to the transmit-clock (at line 36 of transaction "`outputStartBit`" on page 147). The following bits use separate instances of one transaction. `Output_byte` also shows how individual bits of a binary value can be picked out by the "`get_bit`" function (in lines 27 through 36). This function is not part of the language, but a useful function that might well be part of the FTSL library.

`Test_singly` and `test_continuous` are similar, each consisting of nested loops that get data from the data generators and save it; but one performs `load_byte` and `output_byte` in sequence, while the other performs them concurrently.

Finally, the main test program sets out the overall test very clearly; start the transmit-clock and, concurrently, perform the test sequence. The test sequence performs, in turn; the external reset, the single-byte-at-a-time test, and the continuous-stream-of-bytes test.

Two transactions from the receive side have been added: the `serial_data` transaction shows how a "current-time" can be incremented in a loop, while forcing a stream of values on the SI pin; and `verify_data` again shows how FTSL can be used to program one transaction being synchronized with another.

These examples have shown that functional test programs written in FTSL match the vocabulary and syntax of hardware designers and test engineers, and match the usual specification methods for digital circuits. This fulfills the original intent of this thesis.

However, defining a high-level language for digital functional testing suggests that extensions could be made to the language, for other forms of testing.

One particular problem could be solved by a small extension to FTSL: programming the testing of hybrid (analog-digital) circuits. This type of circuit, which includes modems, codecs and analog/digital converters, needs digital,

analog and mixed-mode testing. The available languages though are generally either oriented towards analog testing by controlling individual instruments, or towards digital functional testing by creating vector patterns. Neither type of language is particularly good at the opposite function, and codec test programs have tended to be hand-crafted for each type of circuit to be tested.

The language for the analog side is frequently based on Pascal or Basic, and so FTSL could be used for programming the whole test; the process of compilation would create both a digital test pattern and a control sequence for the analog instruments.

As a specific example of a possible extension to the language, consider the generation of a "digital sine-wave" (see [Hum81]). A PCM (pulse-code-modulated) signal typically consists of sample values at a rate of 8000 samples/second, each sample being processed to form an 8-bit byte. One specific test of the CCITT (Committee Consultative International pour Telegraphie et Telephone) calls for a digital signal corresponding to a sine-wave of specific amplitude and a frequency of 1004Hz to be transmitted to the circuit under test. A change to the TESTDATA section of FTSL would permit the necessary sequence of byte

values to be generated and stored: the change is to allow the data specification to include external functions.

For example, as well as

```
TESTDATA
  Number_of_bits := 5 to 8;
```

the extension would allow

```
TESTDATA
  PCMdata := (PCMcode(n), n = 1, 4096);
```

"PCMcode" would be a library function defined externally, which would convert the parameter *n* to a phase angle, compute the sine of the angle, encode that value into an 8-bit integer, and return the integer.

The data-generator would be used in the body of the test program in a loop such as:

```
forever
  at 1.0/8000 seconds
    force(SI, GetData(PCMdata));
```

Even without language extension though there are exciting possibilities for future work from FTSL.

Obviously, there is the task of implementing a compiler and translator to ATE language. Methods of doing this have been discussed. But there is a much smaller implementation task that would provide useful results and provide a more

rigorous test of the suitability of FTSL: the task is to create timing diagrams from the transaction descriptions in FTSL language. The timing diagram would provide feedback to the person writing test programs in FTSL, and the program to create the timing diagrams (as paper plots, or in PostScript language for laser printers) would detect errors and inconsistencies.

For most timing diagrams, all the necessary data is contained within the transaction description; the exception is transactions which wait on other events. A dummy "force" would be needed, at a suitable time value, to trigger such dependencies.

In summary, FTSL is a language and programming system that would allow a circuit designer, not a specialist in test programming, to write efficient functional test patterns. It is based on a high-level language that most test engineers and circuit designers are familiar with.

The features which set FTSL apart from previous test programming languages are:

- The ability to create parallel streams of transactions, and to synchronize these to each other.

- The ability to re-partition the pins of the DUT between streams of transactions, so that different sections of the overall test program may more easily be created.

This may be illustrated graphically: in the following figure each box represents one transaction. In the first part of the test program, pins E through P are in the lower stream of transactions; but in the second part, they are in the upper stream.

- The ability to encapsulate timing values (minimum, typical, and maximum) within a transaction definition, and to specify, in the program which uses the packaged

definition, which of these is to be used, or to override the timing definition entirely.

The current language definition is however only the start; extensions to the language could make it suitable for other types of test specification, and this raises the possibility of a common language suitable for functional specification, stimulus/response generation for simulation, test generation, and the automatic generation of documentation.

REFERENCES

- [Accu78] -, VLSI Test System Users' Programming and Operation Manual, Accutest Corporation, Chelmsford, Ma.
- [Arno85] J.E.Arnold, "The Knowledge-based Test Assistant's Wave/Signal Editor:", Proc. 2nd International Conference on Artificial Intelligence applications, 1985 pp. 130-136
- [Bori87] G.Borriello, R.H.Katz, "Synthesizing Transducers from Interface Specifications", Proc. International Conference on VLSI (VLSI87).
- [Brin77] Per Brinch Hansen, "The Architecture of Concurrent Programs", Prentice-Hall, Inc. 1977
- [Chan87] Ji.M. Chang, W.T.Krakow, G.Kedem, "A Generic Program Translator (GPTP) for Tester-per-Pin ATE", Report TR87-12, Microelectronics Center of North Carolina, Research Triangle Park, NC.
- [Chom86] R.S.Chomiczewski, "VIVED - A Visual Vector Editor", Proc. 1986 IEEE International Test Conference pp. 621-625
- [DenB87] W.E.Den Beste, "Intent Capture Concepts for Device Simulation and Test", Proc. 1987 IEEE International Test Conference.
- [Fair78] -, Master FACTOR Programming Language Manual, Fairchild Test Systems Group, San Jose, Ca.
- [Gelm82] S.J.Gelman, "VEEP - A VVector Editor and Preparer", Proc. 19th Design Automation Conference, June 1982 pp. 771-776
- [GENR85] -, Software Reference Guide, Genrad Semiconductor Test Division, Milpitas, Ca.
- [Have86] C.D.Havener, "Issues That Arise in Translating VLSI Test Programs Between Testers", Proc. 1986 IEEE International Test Conference pp. 875-885
- [Hood78] D. Hood, R.A. Hum, L.L. Williams, "Developing a Versatile Integrated Circuit Tester", Telesis October 1978, Bell-Northern Research, Ottawa pp. 336-342

- [Hum81] R.A. Hum, D.L. Williams, "An Integrated Analog/Digital Test System", Proc. 1981 Electronics Components Conference pp. 202-206
- [IBM81] Pascal/VS Language Reference Manual, SH20-6162, IBM Corporation, April 1981, p. 62
- [IEEE78] -, IEEE Standard ATLAS Test Language, ANSI/IEEE Standard 416-1978
- [ILOG79] ILOGIC Digital Simulator Reference Manual, I/Logic Inc. 1979
- [Lips86] R. Lipsett, E. Marschner, M. Shahdad, "VHDL - The Language", IEEE Design and Test of Computers, April 1986 pp. 28-41
- [Lowe88] A. Lowenstein, Prospective Computer Analysts, Inc. Roslyn, NY 11576: "Working Paper for a Proposed IEEE Standard Test Vector Language", April 1988
- [LSIL84] -, Design Methodology Manual; LSI Logic Inc.
- [Marx87] E. Marx, H. Switzer, M. Waters, "Use EDIF to Create Test Vectors", EDN, February 1987 pp. 185-192
- [Nash86] J.D. Nash, L.F. Saunders, "VHDL Critique", IEEE Design and Test of Computers, April 1986
- [Peac87] J. Peacock, Cadence Computer Corp: private communication
- [Rohr87] K. Rohr, Schlumberger, Billerica, MA. 01821, letter to chairman of IEEE TVA committee, 1987-11-19.
- [Shou81] B. Snoultten, J. Peacock, "ANGEL - Algorithmic Pattern Generator System", Proc. 1981 IEEE International Test Conference pp. 484-488
- [Tek76] S3250 Automated Test System Description, Tektronix Inc., Beaverton, Oregon, 1976
- [TSSI87] TDS (Test Development Series) software descriptions, Test Systems Strategies, Inc. Beaverton, Oregon
- [VGEN87] -, VGEN User's Manual, Source III, Inc. Los Gatos, Ca

- [Wats82] I.M.Watson, J.A.Newkirk, R.Mathews, D.B.Boyle,
"ICTEST: A Unified System for Functional Testing and
Simulation of Digital ICs", Proc. 1982 IEEE International
Test Conference pp. 499-502
- [Will85] D.L.Williams, "HOLIXZ User Guide", internal report,
Bell-Northern Research.

APPENDIX A. FTSL

The Functional Test Specification Language (FTSL) is a language, loosely based upon PASCAL, for describing functional tests for digital circuits. It is based on PASCAL because most modern manufacturers of Automatic Test Equipment (ATE) use PASCAL as the control language for test equipment.

In comparison with PASCAL, there are additional data types, additional intrinsic procedures, and changes to the structure. The following description concentrates on these differences, rather than providing a complete description of the language.

A.1 TRANSACTIONS VS. PROCEDURES

In PASCAL, the elements of the structure of a program are procedures and functions. In FTSL, a third element has been added: the transaction.

Syntax:

transaction-heading:

——>transaction——>{id}——>{formal-parameters}——>

A transaction is a type of procedure in which the values of signals or registers are set or verified. While a transaction could be concerned with any amount of interaction between signals, it is convenient to form a separate transaction for each unit of interaction between part of a circuit and the rest of the world. Thus a transaction could correspond to concepts such as "handshaking" or "data transfer", or more generally, to "timing diagram", or possibly "timing diagram fragment".

Any statement that is legal within a procedure is also legal within a transaction; but in addition, force and expect statements may be used within a transaction, but not in a procedure nor in a function.

ables of the transaction may have values assigned to them for the duration of the transaction call, in this manner. Any time variable that does not appear in the list will retain its default value, set, in the time section.

The words `min`, `max`, and `typ` may be used in expressions that form part of assignments in this list: those words will be replaced by corresponding values from the time section. Use of a `min`, `max` or `typ` that has not been defined is an error.

A.2 DECLARATIVE SECTIONS

In PASCAL, there may be declarative sections for VAR, LABEL and TYPE; and there may be other sections such as CONST in specific implementations of the language. FTSL adds the following sections: PIN, REGISTER, TIME, and TESTDATA (note that although these are shown here in upper case, FTSL, like PASCAL, is not case sensitive: "PIN" has the same meaning as "pin" and "Pin").

- 0 The low, or "zero" state
- 1 The high, or "one" state
- X The "don't care" state: the actual state may be a 0 or a 1, but which of those two is not specified, and is immaterial.
- Z The floating, undriven state, known variously as "tristate" or "high-impedance".

A pin of one of these types, once declared, may take part in the following operations:

1. Its current value may be used in an expression:
e.g. `buffer_empty := TBMT;`
2. At a specified time, it may be set to a given state:
e.g. `make (TBMT, X)`
3. An instruction may be generated to verify its value at a specified time: e.g. `expect (TBMT, 1).`

Pins of types pwr and gnd may not take part in any operations, or have their states changed or tested. Such pins are enumerated for completeness, and so that specific columns may be assigned to them in any test patterns which may be generated.

Example:

PIN

```
vcc: pwr 1;  
vdd: pwr 2;  
gnd: gnd 3;  
rde: i 4;  
rd8: o 5;  
rd7: o 6;  
rd6: o 7;  
rd5: o 8;  
rd4: o 9;  
rd3: o 10;  
rd2: o 11;  
rd1: o 12;
```

A.2.2 REGISTER

Syntax:

reg-dcl:

—> register —> {reg-spec} —> {pin-id} —> ; —>

reg-spec:

—> {reg-id} —> [—> {low-bit} —> .. —> {high-bit} —>]: —>

low_bit:

—> {unsigned-integer} —>

—> {symbolic-constant} —>

high_bit:

—> {unsigned-integer} —>

—> {symbolic-constant} —>

The register declaration allows groups of pins to be treated as a single register for computation, assignment and manipulation. A pin may appear in more than one register declaration, but such multiple declarations must not cause conflict in setting registers or signals to specific values

(for instance, use of a signal as part of one register could cause it to be given the value "1"; if the signal is used in another register, the second usage must not cause it simultaneously to be set to "0").

Example:

REGISTER

```
DB[1..8]:  db1 db2 db3 db4 db5 db6 db7 db8;
```

```
NB[1..2]:  nb1 nb2;
```

The register limits may be specified as symbolic constants. Any such constant must be defined before any register definition (such as a library element) uses it.

A.2.3 TIME

This section is used both to declare the names of the timing parameters of the circuit to be tested, and to spec-

ify the minimum allowed, typical, and maximum allowed values (or one specific value) for each parameter.

Each procedure that defines a transaction (i.e. that is a textual representation of a timing diagram) must have a TIME section in which are declared all timing parameters used in the timing diagram.

Only the relevant values need be specified: this means that in most cases, only one or two values will be needed for each parameter, and only rarely will all three appear.

Example:

```
TIME
  Tsetup:    1ns/12ns/ 3us;
  Tpropdelay: / 8ns/20ns; { prop delay will typically }
                        { be 8ns, but will never be }
                        { longer than 20ns. }
  Twidth:    100ns;        { The value 100ns is to be }
                        { used in test generation }

```

Notes:

- Parameters relating to input pins:
 - For a parameter such as Tsetup: the minimum value means that as long as the test equipment provides a valid signal at this time, then the circuit is re-

quired to function correctly. A maximum value has no meaning.

- For a parameter such as clock-width: the maximum and minimum values mean that as long as the test equipment provides a pulse whose width (high or low, as specified) is greater than or equal to the minimum, and less than or equal to the maximum, then the circuit is required to function correctly.
- Parameters relating to output pins:
 - For a parameter such as Tpropdelay: the test equipment should verify that the signal value is as predicted, after the minimum time, and before the maximum time.
- Any symbolic constant must be an expression, defined in the cont section, computable at compile time.

Example:

TESTDATA

```

number_of_stops := 1, 2;
Nparity := 1, 2, 3;
Number_of_bits:= 5 to 8;
xmit_data:= 55h, 00h, 0AAh, 0FFh, 1 to 255 by 3;

```

A data generator provides a stream of data values for exercising a circuit or specifying how that circuit should behave..

In the first example line above, "number-of-stops" is a stream of two values, 1 and 2.

In the third example, "Number-of-bits" has four values, 5, 6, 7 and 8.

In the final example, "xmit-data" has the values 55 hexadecimal, 0, AA hex, FF hex, then 1, 4, 7, 10, etc. by 3 to 255: 89 values in all.

The ordinary facilities of Pascal could also be used to specify data values. However, that method would tend to spread the data values through the program, while the data-generator specification clusters all data values at the head of a test program or suite of test programs.

The usage and manipulation of data generators is described in "Functions and Procedures: Data control" on page 132.

A.2.5 VAR: Additional data types

This section shows predefined data types that are available to the FTSL programmer, in addition to those available in standard Pascal.

A.2.5.1 The "flag" data type

Variables of type flag are used for synchronization between transactions. They may be regarded as a form of pseudo-pin, and like a pin, can take on the values 0 or 1 (but not X nor Z). The prime purpose is to allow a stream of events to await some other event, such as a data-strobe waiting for data-available.

A flag may be set to 0 or 1 with the force statement (see "Force statement" on page 128), and, like a pin, may be used in a waituntil (see "Waituntil statement" on page 121).

Example:

```
var
  dataready: flag; {for synchronization}
```

Transaction A;

```
  .
  .
  .
  force dataready, 1;
  .
  .
  .
  end;
```

Transaction B;

```
  .
  .
  .
  waituntil dataready rises;
  force dataready, 0;
  .
  .
  end;
```

A.3 TIMING-RELATIONSHIP-STATEMENTS

Timing-relationship-statements govern the relative timing of other statements or transactions.

A.3.1 Concurrent statement

All statements within a concurrency statement (that is, a concurrent-endconcurrent pair) will begin execution simultaneously; the start-time of each is with reference to the time the concurrent starts.

The concurrency statement will not finish until the finish of the longest-running statement within it. During the time between the end of any constituent statement and the end of the overall concurrency statement, any signals controlled by the statement will retain the value to which they were last forced (see the force statement).

If any of the constituent statements are forever statements, their termination is handled somewhat differently (see "Forever statement" on page 126).

Example:

```
concurrent
  while byte-waiting do output_byte;
  while not eod(number_of_stops) do load_byte;
endconcurrent;
```

In this example, as long as end-of-data had not been reached, "load-byte" would load one byte of data; and as long as there were a byte waiting to be output, "output_byte" would send it.

A.3.2 Waituntil statement

Execution of the current stream is suspended until the event occurs or the condition is satisfied.

In the first form of the `waituntil`, the event is an action: thus if, for example, signal `STRB` were high, then "`waituntil STRB rises`" could wait forever.

In the second form, if `STRB` were high, then "`waituntil STRB is 1`" would not cause a wait.

Notes

1. The "simple-expression" is evaluated when the `waituntil` statement is encountered, and it is this value that is used for testing whether the event occurs or the condition is satisfied.
2. Because of the limitations of ATE (Automatic Test Equipment), the only pins that may appear in a `waituntil` statement, are input pins, or i/o pins in an input mode.

However, a variable of type `flag` may be used to "shadow" an output pin, if needed for a `waituntil` statement.

A.3.3 At statement

Execution of the current stream is suspended until the time given by "time-expression" or "time-const", relative to

the reference time of the statement. The reference time is normally the time at which the "at-statement" was encountered, but may be set to the time of any label within the current block of the program structure by using the wrt clause. In that clause, "wrt" should be read as "with respect to".

The "time-const" may be any expression computable at compile time, and may include terms defined in the const section of the transaction or test program.

Example:

```
at -Tsetup force (DATABUS, 55h);
```

which will force a specific value on to the bus at Tsetup before the "current" time.

In general, the time-expression must evaluate to a quantity greater than or equal to zero; but a negative time may be used provided that it would not cause any waituntil statement to be triggered. This condition can be checked when the test program is translated to ATE language.

Another example:

```
waituntil DS rises;
concurrent
  at -Tsetup force (DATABUS, datavalue);
  at Thold force (DATABUS, not datavalue);
endconcurrent;
```

which will set the databus to a specific value at Tsetup before DS rises, and then set it to the complement of the value at Thold after DS rises (the "not" performs a 1's complement operation on the value to be forced on the bus).

This example could also be written as:

```
waituntil DS rises;
```

```
DSrisen:
```

```
  at -Tsetup wrt DSrisen  
    force (DATABUS, datavalue);
```

```
  at Thold wrt DSrisen  
    force (DATABUS, not datavalue);
```

A.3.4 Forever statement

The forever statement allows continuous signals such as clocks to be succinctly described.

At the end of execution of a forever statement, if any non-forever concurrent process (a process specified in the same concurrent statement as the forever statement) is still running, then the forever statement is re-started at its beginning; otherwise, it is deemed to have completed.

The effect is that any clock pulses defined in forever procedures will repeat until there is no parallel process.

For example:

```

concurrent;
  forever xmit_clock;
  forever rcvr_clock;
  lengthy_xmitter_test;
  lengthy_receiver_test;
endconcurrent;
    
```

A.4 PIN-VALUE-STATEMENTS

Signal-value statements either set pins or registers to specific values, or specify what values are expected to appear on the specified pins.

A.4.1 Force statement

The specified pins, flags or registers are set to the given values.

In the "pin-force" and "flag-force" statements, the "simple-expression" must evaluate to a 0 or a 1.

In the syntax diagram above, the definitions of "flags" and "regs" are analogous to the definition of "pins".

Examples:

```
force RST, 1;
```

```
force (base-register, index-register), 0;
```


This statement specifies that the pins or registers are to be tested to ensure that they do have the specified values. The action to be taken, should the values prove incorrect, depends on the particular ATE (Automatic Test Equipment) on which the tests are being conducted.

A.5 FUNCTIONS AND PROCEDURES: DATA CONTROL

The actual values of test data to be applied to the circuit under test, or to be expected from that circuit, are described in "TESTDATA" on page 113. The sequencing of those values (when to step from one value to the next, detecting when the end of the list of data values has been reached) is performed by the following functions and procedures:

- GetData
- ResetData
- LastData
- EOD

These function and procedure names are not case-sensitive.

A.5.1 GetData function

Obtain the next data item from the specified data generator.

Definition:

```
function GetData (D: DataGenerator): integer;
```

where:

D is the name of a data generator previously specified in a TESTDATA section.

When executed, the GetData function will

1. Step the specified data-generator to the next value in the list of values specified (in the declaration in the TESTDATA section);
2. Return that value to the caller.

Initial and final values:

- Before a particular data generator has been used, or if it has been reset (see "ResetData procedure" on page

136), its internal state is pointing previous to the first value; thus the first GetData on that data-generator will return the first value in the list.

- The GetData that obtains the last item in the data list will also set the internal LastData flag for that generator (see "LastData function" on page 137).
- Any subsequent GetData will continue to return the final value in the list, but will also raise the "EOD" (end of data) flag for that generator.

Usage notes:

- Each GetData will step the generator to the next value: so if a value is to be used at more than one place in a test procedure, then one call should be made to GetData, and the value should be stored in a temporary variable.

Examples:

(1) while not LastData(xmit_data) do
 TransmitTest(GetData(xmit_data));

(2) VAR
 temp: integer;

 BEGIN
 while not LastData(xmit_data) do begin
 temp := GetData(xmit_data);
 ReceiveTest(temp);
 TransmitTest(temp);
 end;
 END.

A.5.2 ResetData procedure

Reset the specified data generator to its original condition.

Definition:

```
procedure ResetData (D: DataGenerator);
```

where:

D is the name of a data generator previously specified in a TESTDATA section.

When executed, the ResetData procedure will reset the internal pointer of the specified data generator so that the next subsequent call to GetData will return the first data item in the list that was specified (in the TESTDATA section of the test program).

ResetData will also clear (set false) the LastData and EOD flags, so that subsequent calls to LastData and EOD functions will return false.

An example is shown on page 138.

A.5.3 LastData function.

Return the value of the internal LastData flag from the specified data generator.

Definition:

```
function LastData (D: DataGenerator): Boolean;
```

where:

D is the name of a data generator previously specified in a TESTDATA section.

When executed, the LastData function will return

true if the value returned on the last previous call to GetData for this generator returned the last value in its list of data values, and the data-generator was not subsequently reset with ResetData;

false otherwise, i.e. if the data-generator has not been used, has just been reset with ResetData, or if the last call to GetData returned a value that was not the last value in the list.

Note: The LastData function does not affect the value of the internal LastData flag.

Example:

```
TESTDATA
  primary := 1, 2, 4, 8, 16, 32, 64, 128;
  secondary := 1, 2, 4, 8, 16, 32, 64, 128;

BEGIN
  while not LastData(primary) do begin
    Emit(GetData(primary));
    ResetData(chorus);
    while not LastData(secondary) do begin
      Emit(GetData(secondary));
    end;
  end;
END.
```

A.5.4 EOD function

Return the value of the internal EOD flag from the specified data generator.

Definition:

function EOD (D: DataGenerator): Boolean;

where:

D is the name of a data generator previously specified in a TESTDATA section.

When executed, the EOD function will return

true if the LastData flag was already true when the last previous call to GetData for the specified data generator was made, and that data generator had not been subsequently reset with ResetData.

false otherwise.

Notes:

- The internal EOD flag is not affected by the use of the EOD function;
- The internal EOD flag is reset by ResetData.
- This function performs analogously to the EOF function in standard Pascal, and is provided for those who are used to that programming style.

Example:

```
VAR
  temp: integer;

BEGIN
  while true do begin
    temp := GetData(xmit_data);
    if EOD(xmit_data) then begin
      ResetData(xmit_data);
      temp := GetData(xmit_data);
    end {if};
    ReceiveTest(temp);
    TransmitTest(temp);
  end {while};
END.
```

APPENDIX B. EXAMPLE: TESTING A UART

```

PROGRAM UARTXMIT;
{ This demonstrates a functional test of a UART, for all
  the valid combinations of parameters (number of bits,
  number of stop bits, even-parity/odd-parity/no-parity)
  and for both individual bytes and a stream of bytes. }
PIN  vcc: pwr 1;
      vdd: pwr 2;
      gnd: gnd 3;
      rde: i 4;
      rd8: o 5;
      rd7: o 6;
      rd6: o 7;
      rd5: o 8;
      rd4: o 9;
      rd3: o 10;
      rd2: o 11;
      rd1: o 12;
      pe: o 13;
      fe: o 14;
      or: o 15;
      swe: i 16;
      rcp: i 17;
      rdav: i 18;
      dav: o 19;
      si: i 20;
      xr: i 21;
      tbmt: o 22;
      ds: i 23;
      eoc: o 24;
      so: o 25;
      db1: i 26;
      db2: i 27;
      db3: i 28;
      db4: i 29;
      db5: i 30;
      db6: i 31;
      db7: i 32;
      db8: i 33;

```

```

cs:    i    34;    {PIN section continued}
np:    i    35;
tsb:   i    36;
nb2:   i    37;
nb1:   i    38;
eps:   i    39;
tcp:   i    40;

```

REGISTER

```

RD[1..8]:  rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8;
DB[1..8]:  db1 db2 db3 db4 db5 db6 db7 db8;
NB[1..2]:  nb1 nb2;

```

VAR

```

saved_data:    integer; {saved value of xmit-data}
NoParityBit:   integer; {saved No-parity}
EvenParityBit: integer; {saved Even Parity Select}
nStopBits:     integer; {saved Number of Stop bits}
nDataBits:     integer; {saved Number of data bits}

```

```

TBMTflag: flag;

```

TIME

```

Tsetup: 1ns/12ns/20ns;

```

TESTDATA

```

NoParity      := 0, 1;
EvenParity    := 0, 1;
NumberOfStops := 1, 2;
NumberOfBits  := 5 to 8;
xmit_data     := 55h, 00h, 0AAh, 0FFh, 1 to 255 by 3;

```

```

1 TRANSACTION xmit_clock;
2
3 { This produces the transmit-clock TCP, according
4   to the timing diagram, with minimum timing values
5   from the chip specification.
6
7   Only one pulse need be programmed, as xmit_clock
8   will be used in a "forever" construct, and will
9   be restarted as often as needed.
10
11
12
13   TCP
14
15   <-TCPlow->|
16
17   <-----TCPperiod----->|
18 }
19
20
21 TIME
22   TCPperiod: 1.6us/ / ;           { min from spec }
23   TCPlow:    750ns/ /TCPperiod-750ns;
24
25 BEGIN
26   {This uses "concurrent" so that both edges can be
27     specified w.r.t. T0 (start time of transaction)
28   }
29   concurrent
30     at TCPlow    force  TCP, 1;
31     at TCPperiod force  TCP, 0;
32   endconcurrent;
33 END;

```

```

1 TRANSACTION control_strobe;
2 {
3     t0
4
5
6 CS      <-CSwidth->
7
8
9 CSsetup-> --> <---- <-CShold
10
11
12 NP etc.
13
14
15 }
16
17 TIME
18     CSsetup    0s / /;
19     CShold     0s / /;
20     CSwidth    300ns / /;
21
22 BEGIN
23     SetStatusBits: at 0ns
24         begin
25             force NP, NoParityBit;
26             force EPS, EvenParityBit;
27             force TSB, nStopBits;
28             force NB, nDataBits; { NB is register }
29         end;
30
31     SetCS: at CSsetup wrt SetStatusBits
32         force CS, 1;
33
34     CSfall: at CSwidth wrt SetCS
35         force CS, 0;
36
37     EndStatus: at CShold wrt CSfall
38         begin
39             force NP, not NoParityBit;
40             force EPS, not EvenParityBit;
41             force TSB, not nStopBits;
42             force NB, not nDataBits;
43         end;
44 END;

```

```

1 Transaction load_byte
2   (const DBdata: integer; var saved_DBdata: integer);
3   { All times with respect to T0, (time when buffer becomes
4     empty: if already empty, T0 is start of the transaction).
5     It is separate from "StartBit", so that it may be used
6     in both test_singly and test_continuous.
7     t0
8     |
9
10  TBMT  TBMTdelay--> | <--TBMTdelay
11
12
13  DS    <-- <--DSwidth
14
15
16  DBsetup|<----->|
17
18  DB    DBdata      X
19
20      |<--|<-DBhold      }
21  TIME
22    DSwidth 190ns/    / ;
23    DBsetup  0ns/    / ;
24    DBhold   0ns/    / ;
25    TBMTdelay    /50ns/ ;
26  BEGIN
27    waituntil TBMTflag is 1; { Wait for buffer empty }
28    BufferEmpty: { It may already be empty }
29    BufferEmpty: { Note TBMTflag is not TBMT }
30    at -DBsetup wrt BufferEmpty begin
31      saved_DBdata := DBdata; { save the value NOW }
32      force DB, DBdata;
33    end;
34
35    at -DSwidth wrt BufferEmpty begin
36      force DS, 0;
37      expect TBMT, 1;
38    end;
39
40    at 0 wrt BufferEmpty begin
41      force DS, 1;
42      force TBMTflag, 0;
43    end;
44
45    at DBhold wrt BufferEmpty force DB, X;
46
47    at TBMTdelay wrt BufferEmpty expect TBMT, 0;
48  END;

```

```

1 TRANSACTION output_bit(const bit_value:integer);
2
3 {
4   This is the "generic" transaction to output
5   one bit of the byte.
6
7
8   TCP
9
10
11   --> || <--Pdelay
12
13   SO
14   bit_value
15
16   <----- 16 * TCPperiod ----->
17
18 }
19
20 TIME
21   Pdelay: 0ns / / ;
22
23 VAR
24   j: integer;
25
26 BEGIN
27
28   { Synchronize on the next clock tick }
29   waituntil TCP falls;
30
31   at Pdelay expect SO, bit_value;
32   for j := 1 to 16 do waituntil TCP falls;
33   { We expect SO to retain its latest value until
34     some later "expect" affects it.
35   }
36 END;

```



```

1 TRANSACTION outputStartBit;
2 { This is specific to the first bit
3   of the byte, to verify EOC.
4
5
6 TCP
7
8
9
10
11 SO      X      bit_value i.e. 1
12
13
14
15
16
17
18 EOC
19
20
21 TBMT
22
23
24
25
26 }
27 TIME
28     Pdelay: 0ns/ / ;
29     EOCdelay: 1us/1us/ ;
30
31 VAR
32     j: integer;
33
34 BEGIN
35     waituntil TCP falls;
36     at Pdelay expect SO, 1;
37
38     waituntil TCP rises;
39     at EOCdelay expect EOC, 0;
40
41     for j := 2 to 6 do waituntil TCP falls;
42
43     expect TBMT, 1;
44     force TBMTflag, 1;
45
46     for j := 7 to 16 do waituntil TCP falls;
47 END;

```

1 Transaction output_byte(const saved_DBdata:integer);
2
3 { If transmitter-buffer is not empty, then the process
4 starts at once; otherwise, it waits for TBMTflag
5 to become 0.
6
7
8
9 TBMTflag

10
11
12
13
14
15 output-
16 byte

Output start bit	Out put_ bit [0]	Out put_ bit [1]	Out put_ bit [2]	Out put_ bit [3]	Out put_ bit [4]	etc.
------------------------	---------------------------	---------------------------	---------------------------	---------------------------	---------------------------	------

17
18
19 }
20
21 VAR byte_data: integer;
22
23 BEGIN
24 waituntil TBMTflag is 0;
25 byte_data := save_DBdata; {grab the value NOW }
26 OutputStartbit;
27 output_bit(get_bit(byte_data,0));
28 output_bit(get_bit(byte_data,1));
29 output_bit(get_bit(byte_data,2));
30 output_bit(get_bit(byte_data,3));
31 output_bit(get_bit(byte_data,4));
32 {In the next statements, note that data-bits are }
33 {numbered 0 thru 7, but NdataBits goes from 5 thru 8.}
34 if (nDataBits > 5) then output_bit(get_bit(byte_data,5));
35 if (nDataBits > 6) then output_bit(get_bit(byte_data,6));
36 if (nDataBits > 7) then output_bit(get_bit(byte_data,7));
37 if not NoParityBit
38 then if EvenParityBit
39 then output_bit(even_parity(byte_data))
40 else output_bit (odd_parity(byte_data));
41 first_stopbit;
42 if (nStopBits = 2)
43 then output_bit(1);
44 expect EOC, 1;
45 END;

```

1 TRANSACTION test_singly;
2
3 { This section tests the UART's ability to output bytes
4   singly; that is, the transmitter goes to the inactive
5   state between each byte, and must re-synchronize
6   itself to the TCP clock.
7 }
8
9 BEGIN
10   ResetData(NoParity);
11   force TBMTflag, 1;
12
13   while not LastData(NoParity) do begin .
14     NoParityBit := GetData(NoParity);
15     ResetData(EvenParity);
16     while not LastData(EvenParity) do begin
17       EvenParityBit := GetData(EvenParity);
18       ResetData(NumberOfStops);
19       while not LastData(NumberOfStops) do begin
20         nStopBits := GetData(NumberOfStops);
21         ResetData(NumberOfBits);
22         while not LastData(NumberOfBits) do begin
23           nDataBits := GetData(NumberOfBits);
24           ControlStrobe;
25           ResetData(xmit_data);
26
27           while not LastData(xmit_data) do begin
28             Load_Byte(GetData(xmit_data), saved_data);
29             Output_Byte(saved_data);
30           end {while xmit_data};
31
32         end {while NumberOfBits};
33       end {while NumberOfStops};
34     end {while EvenParity};
35   end {while NoParity};
36 END;

```

```

1 TRANSACTION test_continuous;
2 {
3   This tests that the UART can correctly transmit a
4   continuous stream of bytes, with the last stop bit
5   of one byte being immediately followed by the start
6   bit of the succeeding byte.
7 }
8
9 BEGIN
10   ResetData(NoParity);
11   force TBMTflag, 1;
12
13   while not LastData(NoParity) do begin
14     NoParityBit := GetData(NoParity);
15     ResetData(EvenParity);
16     while not LastData(EvenParity) do begin
17       EvenParityBit := GetData(EvenParity);
18       ResetData(NumberOfStops);
19       while not LastData(NumberOfStops) do begin
20         nStopBits := GetData(NumberOfStops);
21         ResetData(NumberOfBits);
22         while not LastData(NumberOfBits) do begin
23           nDataBits := GetData(NumberOfBits);
24           ControlStrobe;
25           ResetData(xmit_data);
26
27           concurrent;
28             while not LastData(xmit_data) do
29               Load_Byte(xmit_data,saved_data);
30             Output_Byte(saved_data);
31           endconcurrent;
32
33           end {while NumberOfBits};
34         end {while NumberOfStops};
35       end {while EvenParity};
36     end {while NoParity};
37 END;
```

152

L

```

1 TRANSACTION serial_data
2     (const value:integer; var value_sent:integer);
3
4     { This transforms "value" into a bit stream,      }
5     { according to the number of bits, parity, and    }
6     { number of stop bits specified by the controls.  }
7     { Note that the statements are executed sequentially }
8     { Furthermore, we need to set a flag to give the time }
9     { at which we expect that the received word will be  }
10    { ready to be strobed out (see timing diagram).      }
11
12    {value          is the value of the byte to be sent    }
13    {value_sent     is the value of only the actual bits sent}
14
15    VAR bitnum: integer; {loop counter for each bit of byte}
16
17    BEGIN
18        at 0 force SI, 0; {The "at 0" is not needed here!}
19
20        value_sent := 0;
21
22        for bitnum := 0 to 7 do begin
23            if bitnum < nDataBits then begin
24                at 16*RCPperiod begin
25                    bit_value := get_bit(value, bitnum)
26                    force SI, bit_value;
27                    put_bit (value_sent, bit_value, bitnum);
28                end {at};
29            end {if};
30        end {for};
31        StartOfBit7:
32            if not NoParityBit then
33                if EvenParityBit
34                    then at 16*RCPperiod wrt StartOfBit7
35                        force SI, evenparity(value_sent)
36                    else at 16*RCPperiod wrt StartOfBit7
37                        force SI, oddparity(value_sent);
38        StartOfParitybit:
39            { first stop bit }
40            at 16*RCPperiod wrt StartOfParitybit force SI, 1;
41        StartOfStop:
42            { halfway through stop, data becomes available }
43            at 8*RCPperiod wrt StartOfStop force DAVflag, 1;
44        HalfStop:
45            if nStopBits = 1
46                then at 8*RCPperiod wrt HalfStop force SI, X
47                else at 24*RCPperiod wrt HalfStop force SI, X;
48    END;

```

```

1 TRANSACTION verify_data(const value_sent: integer);
2
3   { This verifies that the correct value is received }
4   { and made available on the RD bus.  It also checks }
5   { that DAV has risen, and that PE, FE and OR are 0. }
6
7   VAR  actual_data: integer; { value saved when DAVflag=1 }
8
9   BEGIN
10      waituntil DAVflag is 1;
11
12      actual_data := value_sent;
13
14
15      { No "at" construct, so each of the following      }
16      { "forces" is at time 0 wrt start of transaction }
17      force DAVflag, 0;
18      force RDE, 0;
19      force SWE, 0;
20
21      { No "wrt" in the "at" so each one is implied }
22      { with respect to the end of the previous one. }
23      { i.e. the delay times are cumulative.          }
24      at 100ns begin
25         expect PE, 0;
26         expect FE, 0;
27         expect OR, 0;
28         expect DAV, 1;
29         expect RD, actual_data;
30      end;
31
32      at 200ns force RDAV, 0;
33
34      at 250ns force RDAV, 1;
35
36 END;
```


Pin No.	Name (Symbol)	Function															
1	V _{cc} Power Supply (V _{cc})	+5V Supply															
2	V _{cc} Power Supply (V _{cc})	+12V Supply (Not connected for AY-3-1014A/1015)															
3	Ground (V _{ss})	Ground															
4	Received Data Enable (RDE)	A logic "0" on the receiver enable line places the received data onto the output lines															
5-12	Received Data Bits (RD0-RD7)	These are the 8 data output lines. Received characters are right justified. The LSB always appears on RD0. These lines have tri-state outputs, i.e., they have the normal TTL output characteristics when RDE is "0" and a high impedance state when RDE is "1". Thus, the data output lines can be bus structure oriented.															
13	Parity Error (PE)	This line goes to a logic "1" if the received character parity does not agree with the selected parity. Tri-state															
14	Framing Error (FE)	This line goes to a logic "1" if the received character has no valid stop bit. Tri-state															
15	Over-Run (OR)	This line goes to a logic "1" if the previously received character is not read (DAV line not reset) before the present character is transferred to the receiver holding register. Tri-state															
16	Status Word Enable (SWE)	A logic "0" on this line places the status word bits (PE, FE, OR, DAV, TBMT) onto the output lines. Tri-state															
17	Receiver Clock (RCP)	This line will contain a clock whose frequency is 16 times (16X) the desired receiver baud.															
18	Reset Data Available (RDAV)	A logic "0" will reset the DAV line. The DAV/F is only thing that is reset.															
19	Data Available (DAV)	This line goes to a logic "1" when an entire character has been received and transferred to the receiver holding register. Tri-state. Fig 12,34															
20	Serial Input (SI)	This line accepts the serial bit input stream. A Marking (logic "1") to spacing (logic "0") transition is required for initiation of data reception. Fig 11,12,33,34															
21	External Reset (XR)	Resets all registers (except that the received data register is not reset in the AY-5-1013/1013A and AY-6-1013). Sets SO, EOC, and TBMT to a logic "1". Resets DAV, and error flags to "0". Clears input data buffer. Must be tied to logic "0" when not in use.															
22	Transmitter Buffer Empty (TBMT)	The transmitter buffer empty flag goes to a logic "1" when the data bits holding register may be loaded with another character. Tri-state. See Fig 16,20,40,42															
23	Data Strobe (DS)	A strobe on this line will enter the data bits into the data bits holding register. Initial data transmission is initiated by the rising edge of DS. Data must be stable during entire strobe.															
24	End of Character (EOC)	This line goes to a logic "1" each time a full character is transmitted. It remains at this level until the start of transmission of the next character. See Fig 17,19,38,41															
25	Serial Output (SO)	This line will serially, by bit, provide the entire transmitted character. It will remain at a logic "1" when no data is being transmitted. See Fig 16.															
26-33	Data Bit Inputs (DB0-DB7)	There are up to 8 data bit input lines available.															
34	Control Strobe (CS)	A logic "1" on this lead will enter the control bits (EPS, NB1, NB2, TSB, NP) into the control bits holding register. This line can be strobed or hard wired to a logic "1" level.															
35	No Parity (NP)	A logic "1" on this lead will eliminate the parity bit from the transmitted and received character (no PE indication). The stop bit(s) will immediately follow the last data bit. If not used, this lead must be tied to a logic "0".															
36	Number of Stop Bits (TSB)	This lead will select the number of stop bits, 1 or 2, to be appended immediately after the parity bit. A logic "0" will insert 1 stop bit and a logic "1" will insert 2 stop bits. For the AY-3-1014A/1015, the combined selection of 2 stop bits and 5 bits/character will produce 1 1/2 stop bits.															
37-38	Number of Bits/Character (NB2, NB1)	These two leads will be internally decoded to select either 5, 6, 7 or 8 data bits/character. <table> <tr> <th>NB2</th><th>NB1</th><th>Bits/Character</th></tr> <tr> <td>0</td><td>0</td><td>5</td></tr> <tr> <td>0</td><td>1</td><td>6</td></tr> <tr> <td>1</td><td>0</td><td>7</td></tr> <tr> <td>1</td><td>1</td><td>8</td></tr> </table>	NB2	NB1	Bits/Character	0	0	5	0	1	6	1	0	7	1	1	8
NB2	NB1	Bits/Character															
0	0	5															
0	1	6															
1	0	7															
1	1	8															
39	Odd/Even Parity Select (EPS)	The logic level on this pin selects the type of parity which will be appended immediately after the data bits. It also determines the parity that will be checked by the receiver. A logic "0" will insert odd parity and a logic "1" will insert even parity.															
40	Transmitter Clock (TCP)	This line will contain a clock whose frequency is 16 times (16X) the desired transmitter baud.															

ELECTRICAL CHARACTERISTICS

Maximum Ratings*

 V_{IL} (with respect to V_{IL})Clock and logic input voltages (with respect to V_{IL})

Storage Temperature

Lead Temperature (soldering 10 seconds)

-20 to +0.3V

-20 to +0.3V

-65°C to +150°C

+330°C

*Exceeding these ratings could cause permanent damage. Functional operation of these devices at these conditions is not implied — operating ranges are specified below

Standard Conditions (unless otherwise noted)

 V_{IL} = -12V $\pm$ 5% V_{IL} = +5V $\pm$ 5%Temperature (T_A) = 0°C to +70°C (AY-5-1013A)

-40°C to +85°C (AY-6-1013 Plastic Package)

-55°C to +125°C (AY-6-1013 Ceramic Package)

Characteristic	Min	Typ**	Max	Units	Conditions
DC CHARACTERISTICS					
Input Logic Levels					
Logic 0	0	—	0.8	Volts	(I_{IL} = 1.6mA max.)
Logic 1	V_{CC} -1.5	—	V_{CC} +0.3	Volts	Unit has internal pullup resistors
Input Capacitance					
All Inputs	—	—	20	pF	0 volts bias f = 1MHz
Leakage Currents					
Three State Outputs	—	—	1.0	μ A	0 volts
Data Output Levels					
Logic 0	—	—	+0.4	Volts	I_{OL} = 1.6mA (sink)
Logic 1	V_{CC} -1.0	—	—	Volts	I_{OH} = 3mA (source)
Output Capacitance	—	10	15	pF	
Short Ckt Current	—	—	—	—	See Fig 25
Power Supply Current					
I_{CC}	—	14	16	mA	AY-5-1013/1013A - See Fig 25
I_{CC} } 25°C, all inputs +5V	—	17	19	mA	AY-6-1013 - See Fig 25
	—	18	20	mA	AY-5-1013/1013A - See Fig 26
	—	21	23	mA	AY-6-1013
AC CHARACTERISTICS					
					T_A = 25°C, output load capacitance 50pF max
Clock Frequency	DC	—	640	KHz	AY-5-1013A
	DC	—	360	KHz	AY-6-1013
Baud	0	—	40	Kbaud	AY-5-1013A
	0	—	22.5	Kbaud	AY-6-1013
Pulse Width					
Clock Pulse	750	—	—	ns	AY-5-1013A - See Fig 9
	1.5	—	—	μ s	AY-6-1013 - See Fig 9
Control Strobe	300	—	—	ns	AY-5-1013A - See Fig 15
	600	—	—	ns	AY-6-1013
Data Strobe	190	—	—	ns	AY-5-1013A - See Fig 14
	250	—	—	ns	AY-6-1013
External Reset	500	—	—	ns	AY-5-1013A - See Fig 13
	1.0	—	—	μ s	AY-6-1013
Status Word Enable	500	—	—	ns	AY-5-1013A - See Fig 21
	600	—	—	ns	AY-6-1013 - See Fig 21
Reset Data Available	250	—	—	ns	AY-5-1013A - See Fig 22
	350	—	—	ns	AY-6-1013 - See Fig 22
Received Data Enable	500	—	—	ns	AY-5-1013A - See Fig 21
	600	—	—	ns	AY-6-1013 - See Fig 21
Set Up & Hold Time					
Input Data Bits	0	—	—	ns	See Fig 14
Input Control Bits	0	—	—	ns	See Fig 15
Output Propagation Delay					
TPD0	—	—	500	ns	AY-5-1013A - See Fig 21 & 24
	—	—	650	ns	AY-6-1013 - See Fig 21 & 24
TPD1	—	—	500	ns	AY-5-1013A - See Fig 21 & 24
	—	—	650	ns	AY-6-1013 - See Fig 21 & 24

**Typical values are at +25°C and nominal voltages

TIMING DIAGRAMS

Fig.5 UART/T TRANSMITTER TIMING

Fig.6 TRANSMITTER AT START BIT

Fig.7 TRANSMITTER AT START BIT

Fig.8 ALLOWABLE POINTS TO USE CONTROL STROBE

Fig.9 ALLOWABLE TCP, RCP

TIMING DIAGRAMS

Fig. 10 UART RECEIVER TIMING

Fig. 11

Fig. 12 RECEIVER DURING 1st STOP BIT

WHEN NOT IN USE, XR MUST BE HELD AT GND.
XR RESETS EVERY REGISTER EXCEPT CONTROL REGISTER AND RECEIVED DATA. SO, TSMY, EOC ARE RESET TO 5V. ALL OTHER OUTPUTS RESET TO 0V.

Fig. 13 XR PULSE

Fig. 14 DS

CONTROL BITS MUST BE STABLE FOR LAST 300ns OF CS

Fig. 15a CS

CONTROL STROBE AND CONTROL BITS MUST BE 300ns MINIMUM

Fig. 15b CS

LEADING EDGE OF DATA IS NOT CRITICAL AS LONG AS TRAILING EDGE AND PULSE WIDTH SPECS ARE OBSERVED

Fig. 15c CS

Fig. 16

TIMING DIAGRAMS

Fig.17 EOC TURN-ON

Fig.18 TBMT TURN-OFF

Fig.19 EOC TURN-OFF

Fig.20 TBMT TURN-ON

Fig.21 RDE, SWE

Fig.22 RDAV

TYPICAL CHARACTERISTIC CURVES

Fig.23 SHORT CIRCUIT OUTPUT CURRENT

Fig.24 RE1, RD8, PE, FE, OR, TBMT, DAV

Fig.25 -12 VOLT SUPPLY CURRENT

Fig.26 +5 VOLT SUPPLY CURRENT

APPENDIX D. CREATING AN EVENT-LIST FROM AN FTSL PROGRAM

A test program in FTSL is a set of co-operating parallel processes, and so a computer program to execute FTSL will have some resemblance to a multiprocessing operating system: there must be facilities for creating, scheduling and destroying processes. This program then is a simulator of the "ideal" tester, but instead of operating real ATE facilities, it writes out the equivalent instructions as events, into an event list file, at the same time recording "structural events" (start or end of procedures or transactions) in the file.

The ATE simulator has two phases:

1. Load the whole test program into memory, representing the test program as a static data structure.
2. Execute the test program from the static data structure, adding a dynamic data structure to it to represent the course of the execution.

D.1 CREATING THE STATIC DATA STRUCTURE

Scanning the text of the test program, breaking it into tokens, converting these to symbol references, and assigning a symbol type to each reference are performed according to established techniques for compiler-writing. Creating an executable data structure for an interpretive computer programming language is again according to known techniques.

The result of this initial phase is a linked list of records, each of which corresponds to a statement or expression evaluation in the source language, plus supporting data structures such as a symbol table, data element descriptors, and the nucleus of an execution stack.

The records described below are proposals for some elements of the data structure that would not be found in a computer programming language.

D.2 THE STATEMENT/EXPRESSION CHAIN

Each record in the statement/expression chain has the same two fields at the start of the record:

where

StmntPtr is the pointer to the next statement/expression record. Although the records do follow each other consecutively, it is convenient to have a pointer because not all records will have the same length, and some records (e.g. a CALL) do not have a fixed length.

code is the code that identifies the type of record:
such as Expression, Assignment, Call, Return, ...

and the unique fields are those that are specific to the type of record.

Records for the additional statement types in FTSL (statements that are not in standard Pascal) are:

D.2.1 Concurrent

where

ProcPtr1 points to the chain of execution elements for the first concurrent process specified in this concurrent statement.

ProcPtr1 points to the chain of execution elements for the second concurrent process specified in this concurrent statement.

ProcPtrn points to the chain of execution elements for the nth concurrent process specified in this concurrent statement.

D.2.2 At

where

TimePtr points to either a time-constant, or to a temporary variable holding the result of a time-expression.

D.2.3 Waituntil

The `waituntil` generates one of two types of record: one type is "waituntil event" and the other, "waituntil condition".

where

WUcode is the code for the type of "waituntil" record: either "waituntil event" or "waituntil condition".

PinFlgRegPtr points to the record for the pin, flag or register named in the `waituntil` statement.

ValuePtr points to an integer word that is the value specified in the "waituntil" statement (but see "Mission" below). Notice that "waituntil....rises" becomes

"waituntil....becomes 1", and "waituntil....falls" becomes "waituntil....becomes 0".

Mission For "waituntil event", this field has one of the values "becomes_X", "becomes_Z", or "becomes_value". When it is "becomes_value", then the value will be found in the word specified by the ValuePtr field.

For "waituntil condition", this field has one of the values "is" (i.e. "=") or "isnot" (i.e. "!="), which specifies a comparison between the specified pin, flag or register and the value referred to by ValuePtr.

D.2.4 Forever

D.2.5 Force and Expect

The force statement can be considered as three types of statement: force value on a pin, force value on a register, force value on a flag.

Similarly, the **expect** statement can be considered as two types of statement: expect pin to have a value, expect register to have a value.

The generic record shown here is then the prototype for five types of record: Force Pin, Force Register, Force Flag, Expect Pin, and Expect register.

where

FEcode is the code for the type of statement: of record: Force Pin, Force Register, Force Flag, Expect Pin, or Expect register.

PinFlgRegPtr points to the entity whose value is to be forced or queried: a pin, a register or a flag.

Since these records only allow for one **PinFlgRegPtr** operand, a **force** or **expect** statement that refers to more than one pin, register or flag will translate into one record for each pin, register or flag.

ValuePtr points to an integer word that is the value to be forced or expected.

VXZ is a code to specify what is to be forced or expected: either X, Z, or the actual value (V) in the value field.

D.3 PIN RECORDS

As each pin description is encountered, a "pin-entry" record, as shown in "Pin records", will be created. This record will have null pointers in the PinRegPtr and DemonPtr fields, and its initial state will be X, meaning "unknown" or "don't care".

Pin information is stored in a linked list, with one record (one entry) for each pin defined in the FTSL test program.

D.3.1 Pin record and field descriptions

In the Pin record,

PinPtr points to the next record in the linked list of pin-records. The records are not linked in any

particular order. The PinPtr field in the last record of the chain has a null pointer.

PinRegPtr points to the start of a chain of PinRegister records, if the pin has been specified in any register definitions; otherwise, this field is null.

DemonPtr points to the chain of demons that are watching for activity on this pin. This is a dynamic chain, varying as execution progresses; if there are no demons watching this pin, the DemonPtr field is null.

PinNum holds the pin-number of this pin.

Name holds the name of this pin (either literally, or as a pointer to a name record).

State is a dynamic field which at all times holds the current state of the pin: 0, 1, X, or Z.

Type is the type of pin: i (input), o (output), io (input-output), pwr (power), or gnd (ground).

There is a linked list of PinRegister records for every pin that appears in a register definition. Each PinRegister record contains three fields.

PinRegPtr is a pointer to the next PinRegister record for this pin; thus a pin may appear in more than one register definition.

RegPtr points to a register whose definition includes this pin;

PinMask contains a mask that specifies the position of this pin in the register definition. It contains a "1" at the bit position corresponding to this pin, and a "0" in every other bit position. This method of specification allows the register's value to be updating efficiently when the pin's value changes.

D.4 REGISTER RECORDS

A register is an alias for a group of pins, so that the pins may be referenced as a group in force, expect, and waituntil statements.

For each register descriptor in the test program, a "reg-entry" record must be created. The DemonPtr field should be null, and the whole register (and hence each individual pin) will have a state of X.

For each pin in the register, a PinRegPtr record must be created and linked to that pin's PinRegPtr chain.

Finally,

- the RegPtr field must be set to point to the corresponding "reg entry" record.
- the PinMask field must be loaded with a word that is zero, except for one "1" in the bit that corresponds to the pin's position in the register.
- a RegPinPtr record must be created and linked to the register's RegPinPtr field.

- the PinPtr field of this record must point to the pin's record.
- the PinMask field must be loaded with the same mask word as described above.

N.B. It would be possible to have only one copy of the mask, in the PinRegPtr record, and to access it from the Register record by following pointers and scanning for the correct record in the PinRegPtr chain. However, a copy of the mask is also kept in the RegPinPre record, purely for execution efficiency.

D.4.1 Register record and field descriptions

There is one register record for each register that is defined in the FTSL test program; these form a linked list, in no particular order. In the register record,

RegPtr points to the next record in the linked list.

RegPinPtr points to a chain of register-to-pin records, one for each pin named in the register definition.

DemonPtr points to the chain of demons that are watching for activity on this register. This is a dynamic chain, varying as execution progresses; if there are no demons watching this register, the **DemonPtr** field is null. (Even though there may be no demons watching the register as a whole, there may be one or more demons watching individual pins of the register).

Name is the name of the register, either literally, or as a pointer to a text string.

State is the current state of the register: the values which may appear in this field are: X, Z, data, mixed. The "mixed" value means that one or more individual pins of the register have a state that is at variance with other pins in the register. This can arise if, for example, the register as a whole is set to the Z state, but subsequently one pin is forced to a 0 or 1.

Value is the value contained in the register when the state field has the value "data".

The register-to-pin records are used to form linked lists, one chain for each register, pointed to by the RegPinPtr of the register record. In each chain, there will be one record for each pin named in the definition of the register. The records are linked in bit order within the register: low bit (2^0) to high bit. In each register-to-pin record:

RegPinPtr points to the next record in the chain, with the last record in the chain having a null pointer.

PinPtr points to a pin record (see "Pin records" on page 170) corresponding to a pin that was named in the register definition.

PinMask specifies the position of the pin within the register. This field contains a "1" in the bit-position corresponding to the pin, and "0" in every other bit position.

D.5 RECORDS FOR "FLAG" DATA TYPE

The flag data type is for use in inter-process synchronization where there is no pin suitable for this purpose: the flag record thus has many fields in common with the pin-record.

D.5.1 Flag record descriptions

In the Flag record,

Flagptr points to the next record in the linked list of flag-records. The records are not linked in any particular order. The FlagPtr field in the last record of the chain has a null pointer.

DemonPtr points to the chain of demons that are watching for activity on this flag. This is a dynamic chain, varying as execution progresses; if there are no demons watching this flag, the DemonPtr field is null.

Name holds the name of this flag (either literally, or as a pointer to a name record).

State is the current state of the flag: the values which may appear in this field are: X, Z, data.

Value is the value contained in the register when the state field has the value "data".

D.6 THE DYNAMIC DATA STRUCTURE: EXECUTION

As execution progresses through the statement/expression chain created from the FTSL test program, various data structures are created and destroyed. These structures are composed of process and demon records (see "Process" on page 184 and "Demon" on page 188). The actual operations on these records are triggered by encountering specific statement records during the execution.

Initially, there will be one process, corresponding to the TestProgram statement. As new processes are created by "concurrent" statements, they are linked into the chain of process records. A process runs until it creates an event (and sets its own current-time to the time of that event), or until it encounters a "waituntil" statement. At this point it relinquishes control, and all other processes are scanned to see whether any is available to run. If more than one is available, the one with the lowest current-time is given control.

Processes are destroyed at "endconcurrent" statements, following the rules given below, and finally one process is left to end the test program.

At that point, all events will have been created, and stored in the event-file.

The description that follows is limited to describing actions that result from the FTSL features that differ from standard Pascal. On encountering

concurrent Make a process-entry record for each process specified in the statement record (See Figure 13 on page 164). Link these records into a chain, and set the "child" field of the currently-executing process to point to the head of this chain. Set the "parent" field of each record to point to the record for the currently-executing chain.

transaction Stack return address, and parameter values or parameter addresses, depending on type of parameter, and transfer control to called transaction.

end (transaction) Unstack, transfer control to return address

endconcurrent Control only arrives here when one of the controlled concurrent processes reaches its end.

3

of/de

3

MINI-DESIGN-512
MICRO

- If the process has no "forever" tag, terminate it.
- If every remaining process in the concurrent group has a "forever" tag, then terminate all processes in this concurrent group.
- Else, if the current process has a "forever" tag, restart it from the top.

at Create an event at the specified time, update the current-time of the owning process.

waituntil Create a demon (see "Demon" on page 188).

force Follow the pin-record pointer to find whether this pin (or flag) has a demon: if so, then if the process enslaved by the demon has become free to run, then set its "run" field accordingly.

D.7 PROCESS

Process records are created and destroyed dynamically, as execution of the FTSL test program progresses. At the start, there is one process, and one process record: others are created and linked on encountering `waituntil`, `concurrent`, `forever`, and `at` statements.

The process records form a branching chain; a branch occurs for each concurrent statement. In each record, the fields are:

ProcPtr points to the next concurrency record in the current chain, which initially is the main chain.

StartPtr points to the start of the executable code for this process.

StackPtr points to the first record of the stack of execution records for this process.

Time is the current simulated time for this process.

Forever specifies whether this process is to terminate when the end of the executable code is reached, or

is to re-start at the beginning and re-execute until all concurrent processes have stopped (as in clocks), or forever processes.

Run specifies whether the process is available to run, or is blocked, awaiting the action of a demon, or the termination of other processes, to make it available to run once more.

Child is null except where the current process has encountered a concurrent statement, has created and started concurrent subsidiary processes, and has suspended itself. In this case, the Child field points to the first of the subsidiary processes, which are linked by their ProcPtr fields.

Parent points to the process which created the current one, or, in the case of the initial (top) process, is null.

nChildren the number of concurrent subsidiary processes.

This number is set when the subsidiary processes are created, and will decrement as each subsidiary process terminates.

nForever the number of subsidiary processes that are tagged as "forever".

The **StackPtr** in the process record points to a linked list of Stack records. This stack represents the nesting of procedures during execution of the process. In each record, the fields are:

StackPtr points to the next stack record in the stack. In the last stack record, this field is null.

PC points to the current execution point in the code, for this process. In fact, it points to the start of the next statement to be executed, and is analogous to the Program Counter (PC) of a computer.

ParamPtr Procedure calls may have parameters, and so each record in the stack must store the parameter value that pertain to that call. Conceptually, these values are stored in the stack, but they may in practice be in a separate record or records.

D.8 DEMON

The "demon" is the mechanism used to implement the waituntil feature of FTSL. When a waituntil is encountered during execution, a demon is created and linked to the pin, register or flag named in the waituntil. When a pin, register or flag is named in a force or expect statement, it is checked for demons: and if one or more are found, they are processed to find whether the waiting processes may be allowed to continue.

from Figure 18 on page 171.

> to next demon
watching this pin

> to the waiting process:
see Figure 21 on page 185.

Figure 22. Demon records

In the demon record,

DemonPtr points to the next record in the linked list of records specifying demons for this pin, register or flag (the field is null if there no demons for this pin).

DemonType is the type of the demon:

- event-demon
- condition-demon

ProcPtr (Process Pointer) is a pointer to the process waiting on this demon.

Mission specifies the sort of happening that the demon is looking for.

For an event-demon: this field will have one of the following values: rises, falls, becomes_X, becomes_Z, becomes_value.

For a condition-demon: this field will have one of the values <, <=, =, >=, >, not=.

Value contains the actual value to be used in the comparison operation by a condition-demon, or by an event-demon whose Mission is "becomes_value".

END

17|07|89|

FIN