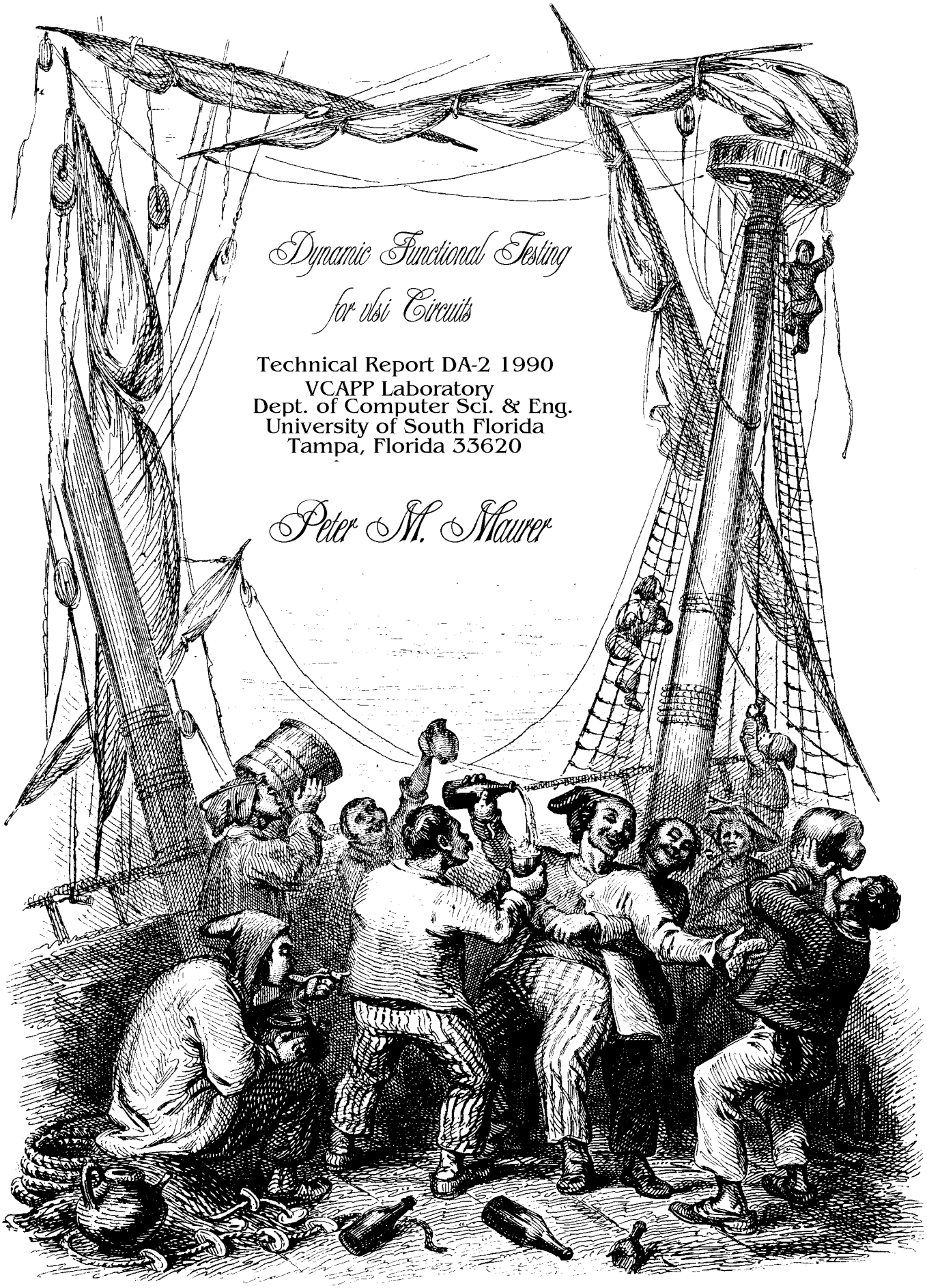

Dynamic Functional Testing for vlsi Circuits

Technical Report DA-2 1990

VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620

Peter M. Maurer

DYNAMIC FUNCTIONAL TESTING FOR VLSI CIRCUITS

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

Dynamic testing is the process of creating test-vectors during simulation and using the output of the simulator to guide the vector generation process. The two main problems of dynamic testing are the design of a high-level vector-generation language, and the design of the interface between the vector-generator and the simulator. Solutions to these two problems are presented. The paper discusses guidelines for designing a high-level vector generation language, and presents several examples written in the FHDL driver language which was designed according to these guidelines. The examples illustrate how dynamic testing can be used to simplify the verification of circuits at the functional level. The paper presents a solution to the interface problem which is designed around a special interface data structure. This data structure supports several different styles of vector generators and also supports the interactive debugging of circuits. The interface data structure also supports the independent simulation of subcircuit instances and the dynamic creation and simulation of new circuit instances.

DYNAMIC FUNCTIONAL TESTING FOR VLSI CIRCUITS*

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

1. Introduction

Creating a VLSI circuit is a complex process that requires careful design procedures at many different levels[1-5]. At the lowest level the circuit must function correctly as an electrical device, at higher levels the arithmetic and control circuits must conform with the laws of mathematics and with the over-all objectives of the design, while at still higher levels the circuit must perform a sequence of steps that accomplishes some over-all objective. To produce a correct design, it is necessary to simulate and test the chip at each of these levels. Much effort has been focused on the problem of simulating and testing the electrical properties of a circuit, and this remains one of the most fruitful areas of VLSI research. However as circuits have become larger, the problem of verifying that a circuit performs its logical and algorithmic functions correctly has become increasingly difficult. In fact, the search for techniques that facilitate the functional verification of circuits is rapidly becoming an important area of VLSI research[6-11]. Tools and techniques that once were considered to be applicable only to software verification are now being applied to the problem of functional verification of VLSI circuits[12,13].

Although considerable progress has been made in proof techniques for certain types of circuits[14], the most widely used method of functional verification is functional testing. (In this paper, the term "functional testing" is used in its broadest sense to include all forms of testing that are not concerned with timing or other electrical parameters of the circuit.) The main problem with functional testing is that an enormous number of tests are needed, and conventional measures of test quality, such as classical fault coverage, are of little help in determining the quality of functional tests. (If fault-coverage is low the tests are certainly bad, but if fault coverage is high they are not necessarily good.) Unfortunately for many VLSI circuits, creating even a small number of tests may be a difficult task, especially at the

---

* This research was supported by the Defense Advanced Research Projects Agency under grant 2114-033-LO and by the University of South Florida Center for Microelectronics.

logic and switch levels. This is particularly true for highly sequential circuits that must conform to complex bus protocols such as that of the MC68020 microprocessor[15].

Many tools have been developed to simplify the process of creating functional tests[16], although most of these are special purpose tools created with a specific circuit or a specific type of circuit in mind. General purpose vector-generation languages allow one to create a test generator that accepts a small set of operands, and transforms this set of operands into a (much larger) set of test vectors. Simulation begins only after all vectors have been generated. Although these tools greatly reduce the amount of time necessary to create tests, construction of the test generator sometimes requires an intimate knowledge of the circuit under test. For example, construction of a test-generator for a microprogrammed divider requires one to know the number of cycles the circuit will consume while performing a division. This can be particularly troublesome when the number of cycles is data dependent. The difficulty stems from the fact that vector-generation is a static process that does not interact with the simulation. Clearly if the test generator could interact with the simulator while it was running, feedback signals from the circuit under test could be used to dynamically determine the number of test vectors required for a particular set of operands.

Another problem with existing vector-generation methods is that the high-level commands used to generate tests are not available for interactive debugging. Interactive debugging is a powerful technique that has been used to debug both hardware and software systems. Although many simulators provide some sort of interactive interface([17] for example), they generally do not provide the powerful command-set available in a vector-generation language. A vector generation language that could be used interactively would greatly enhance the power of such an interface.

Dynamic testing is the process of creating test-vectors *while the simulation is running* and using the output of the simulator to guide the vector generation process. The vectors may be created interactively through a user interface, or automatically by a vector-generation program. The two most important issues are the design of a dynamic vector generation language and the design of the interface between the vector-generator and the simulator. Sections 2 and 3 discuss the principles of such a language and illustrate how it can be used to simplify the process of functional testing. Section 4 describes a versatile interface that is used to pass data between the dynamic vector-generator and the simulator. Section 5 draws conclusions.

2. The Basics of Dynamic Testing

The most obvious advantage of dynamic testing is that it can be used to simplify the process of creating test vectors for circuits such as that pictured in Figure 1. This circuit is a 16-bit microprogrammed divider that accepts two inputs "divisor" and "dividend" and produces one output "quotient." The "start" input is used to signal the circuit that a new divisor and dividend are available, and the "done" output is used to signal the availability of the quotient. The "clk" input is a two-phase clock.

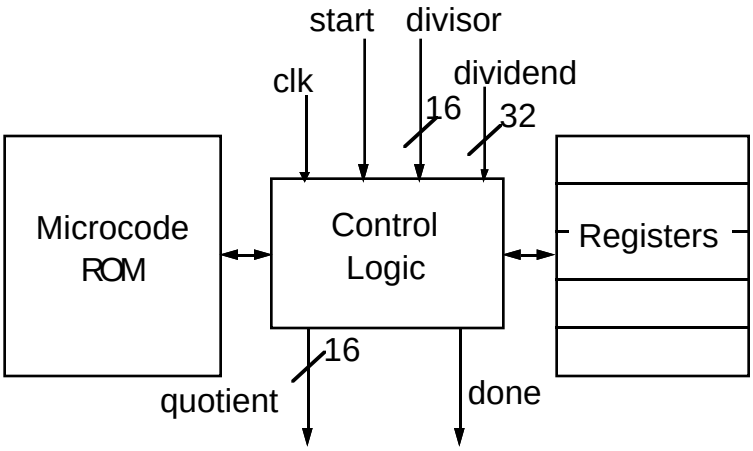

Figure 1. A Microprogrammed Divider.

Assuming that a phase-by-phase logical simulator has been constructed for the circuit, one would test it by performing the following steps.

1. Specify a set of divisor/dividend pairs that will be used to test the circuit.
2. Embed each divisor/dividend pair in a set of test vectors that will supply the divisor and dividend at the proper time and will also provide the proper stimuli to cause the circuit to perform a division.
3. Provide the simulator with the appropriate parameters to cause it to print the value of the "quotient" and "done" signals for each phase.
4. Run the simulator using the test vectors created in step two, and print the output.
5. Examine the output looking for the "done" signal and write down the quotient computed by the circuit for each dividend/divisor pair.
6. Compare the quotients computed by the circuit with quotients computed by some other method.
7. If any tests fail, rerun that part of the simulation to obtain a more detailed trace of the internal logic.

Dynamic testing can eliminate most of this work. First, consider the problem of creating the test-vectors for the divisor/dividend pairs. Static tools can be used to create these vectors as long as the number of vectors required for each pair is known. But this can be difficult if the number of cycles consumed by the algorithm is data-dependent or if there are significant changes to the algorithm during debugging. Of course one could force all tests to be the same length by padding the shorter tests with extra vectors, but this wastes resources on useless simulation. Furthermore, it may be necessary to test whether the circuit is capable of responding to a "start" signal in the cycle following the assertion of the "done" signal. In this case, the extra vectors would prevent one from testing the circuit properly. Dynamic test-generation can solve these problems by using the "done" signal to determine when a sufficient number of vectors have been generated.

Next, consider the problem of analyzing test output. Because the test generator runs concurrently with the simulator, it can be used to filter the output of the simulation. Specifically, the test generator can be designed to print the value of the quotient only when the "done" signal is active. Furthermore, the output does not have to be a phase-by-phase dump of the outputs of the circuit. Data from several different phases can be combined to create a test report that is easy to analyze. Combining these ideas we obtain the test generator illustrated in Figure 2.

```
clock   clk,0,1
read    test_file,divisor,dividend
while   not eof(test_file)
        1->start
        go; go
        0->start
        while   done=0
            go; go
        endwhile
        display     divisor,dividend,quotient
        read    test_file,divisor,dividend
endwhile
```

Figure 2. A Dynamic Test Generator.

The example of Figure 2 is constructed using the *driver language* provided with the Florida Hardware Design Language (FHDL). (The term "driver" denotes a dynamic vector-generation program.) The input to the vector-generator is a set of divisor/dividend

pairs, which are read directly into the circuit's inputs by the "read test_file" statement. The "go" statement causes one test vector to be passed to the simulator. When a vector is passed to the simulator, the circuit is simulated for one clock phase, one output vector is produced, and control is returned to the driver. (The semicolon is used to place more than one statement on a line.) The "clock" statement causes the "clk" input to cycle between 0 and 1 every time a "go" statement is executed.

When a test is read, the "start" input is asserted for one cycle. Then the driver provides a clock signal until the "done" output is asserted. When "done" is asserted, the driver displays the value of the quotient along with the values of the divisor and the dividend. The uncluttered output may easily be read and checked against a set of expected results. Note that the driver language resembles a high-level programming language in which the primary inputs and outputs of a circuit play the role of variables.

Figure 2 illustrates some of the principles that governed the design of the driver language. The most important principle is that the language must provide the full power of a high-level programming language, in the sense that it must provide conditionals, loops, variables, assignments, expressions, and some method of constructing complex commands out of simpler ones. Another important principle is that the primary inputs, primary outputs, and internal nodes of a circuit should be treated as if they were variables. They may be assigned values, and their values may be used in expressions. Any operation that can be performed on a variable must also be legal when performed on a circuit element, and vice versa. All circuit elements, including register and flip-flop states, must be accessible and must bear the name assigned to them in the circuit description. No additional declaration of these names should be necessary. Finally, the driver language must provide I/O facilities that are oriented toward reading and writing vectors, and it must have some method of interfacing with the simulator.

These principles have allowed us to design a reasonably powerful language for generating test-vectors and controlling simulations. We can use the power of this language to simplify the testing process even further. For example we could insert the following code after the "display" statement in Figure 2 to automatically diagnose the results of a test.

```
if   quotient≠divisor/dividend
     print    "test failed"
else
     print    "test ok"
endif
```

But we can do better than this. When a test fails we can automatically rerun the test and provide a detailed trace of the internal logic for debugging purposes, as the example of Figure 3 illustrates.

```
if   quotient≠divisor/dividend
     print    "test failed"
     monitor     clk,rom.address,add_subtract_control,...
     1->start
     go; go
     0->start
     while   done=0
          go; go
     endwhile
     demonitor  clk,rom.address,add_subtract_control,...
else
     print    "test ok"
endif
```

Figure 3. Providing a Detailed Trace of Failed Tests.

In Figure 3 the "monitor" statement causes the values of the specified signals to be printed every time a "go" statement is executed. The "demonitor" statement reverses the effect of the "monitor" statement. As with the previous example, the code of Figure 3 will be inserted after the display statement of Figure 2. With this modification we have eliminated all but the first step of the procedure described above.

Although the language features illustrated so far are sufficient for creating simple test-generators, the creation of more complex drivers will be difficult without some method of creating subroutines. Driver-language macros can be used for this purpose. The following is an example of a driver-language macro definition.

```
cycle:  macro
        go; go
        endmacro
```

Once this macro has been defined, it can be used in place of the two commands "go; go." As with most macro facilities, driver language macros may have parameters and local variables. Figure 4 gives an example of a macro that performs a "read" bus transaction for a complex bus protocol. This macro makes use of the "cycle" macro defined above.

```
busread:  macro     address,value
        0->write_strobe; 1->address_strobe; address->address_bus
        cycle
        1->data_strobe
        while   acknowledge=0
            cycle
        endwhile
        data_bus->value
        cycle
        0->data_strobe; 0->address_strobe
        while   acknowledge=1
            cycle
        endwhile
        endmacro
```

Figure 4. A Complex Macro Definition.

The macro of Figure 4 would be invoked using a command similar to the following.

```
busread     input_address,result
```

In terms of its implementation, this command is more similar to a high-level-language function than to an assembly-language macro call. The macro facility allows one to perform the essential task of partitioning a large complex test generator into more manageable pieces.

3. Asynchronous Event Handlers

To realize the full potential of dynamic testing, it is necessary to have a convenient method for responding to asynchronous events that occur in the simulator output. For

example, suppose we add an "overflow" output to the circuit of Figure 1, and specify that the "overflow" output can be asserted only when the "done" output is active. In the driver of Figure 2, one could enforce this condition by placing the appropriate tests after every "go" statement, but this could hardly be called *convenient*. To simplify the process of responding to asynchronous conditions, the driver language provides asynchronous event handlers. An event handler is declared as follows.

```
on  <condition>
    <statements>
endon
```

An event handler declaration causes the specified condition to be tested after every "go" statement and if the condition is true, causes the specified statements to be executed. The following event handler could be added to the driver of Figure 2 to verify the correct functioning of the "overflow" signal.

```
on  overflow=1&done=0
    print    "overflow asserted illegally"
endon
```

This event handler is reasonably simple because the specified condition is always illegal. However there are conditions that are legal at some points in the simulation and illegal at others. Named event handlers, such as that illustrated below, can be used to simplify the handling of such conditions.

```
a:  on  x=y
        print    "x is equal to y"
    endon
```

The advantage of named event handlers is that they can be explicitly activated and deactivated. When an event handler is deactivated, its statements are never executed, regardless of the state of its condition. Named event handlers can be used to verify the correct sequencing of events such as those that occur in complex bus transactions. For example, suppose that a chip has two output signals "cycle_start" and "address_strobe." All bus transactions begin with the "cycle_start" signal, followed by the "address_strobe" signal one cycle later. Both signals must coincide with the high phase of the clock. Figure 5 illustrates a set of named event handlers that can be used to issue error messages if this protocol is violated.

```
cstart:  on  cycle_start=1
             if   clk=0
                  print "cycle starts on wrong phase"
             else
                  deactivate   cstart
                  activate     csend
             endif
         endon
adrbad:      on  address_strobe=1
             print    "Address Strobe Too Early"
         endon
csend:   on  clk=0
             deactivate   adrbad,csend
             activate     adrstart
         endon
adrstart:    on  address_strobe=1
             if   clk=0
                  print    "Address Strobe on wrong phase"
             else
                  deactivate   adrstart
                  activate ...
             endif
         endif
```

Figure 5. Event Handlers for Protocol Verification.

Before a bus-cycle starts, the two on conditions "cstart" and "adrbad" are activated, while "csend" and "adrstart" are deactivated. The event handler "adrbad" will print an error message if "address_strobe" appears before "cycle_start." When "cycle_start" is asserted on the correct phase, "cstart" is deactivated and "csend" is activated. After the clock advances one phase, "csend" deactivates both itself and "adrbad" and activates "adrstart" to indicate that "address_strobe" is now legal. Because named event handlers can be used to implement arbitrarily complex state machines, they can be used to verify highly complex protocols.

4. The Simulator Interface

Up to this point no method has been specified for executing driver-language commands or communicating data to the simulator. Although it has been stated that the driver statements are executed simultaneously with the simulator, this could be accomplished in several different ways The commands could be typed interactively and executed as they are entered. The commands could be stored in a file which is then passed to a command interpreter. Finally, the command file could be compiled into an object program and either linked with the simulator, or run as a separate process that communicates with the simulator via message passing. Each of these methods has its advantages. An interactive command interpreter is a necessity for interactive debugging, but is cumbersome for high-volume testing. Interpreted command files provide versatility, but execute slower than compiled command files. Oddly enough, our implementation strategy for the driver language is "all of the above." We have implemented a driver-language compiler to provide fast execution for frequently executed command files. We have also provided an interpreter for interpreting command files and executing commands interactively. Since FHDL is a compiled simulation language, compiled command files are normally linked with the simulator as illustrated in Figure 6. It is also possible to link the driver with an interface routine that communicates with a non-compiled simulator running as a separate process.

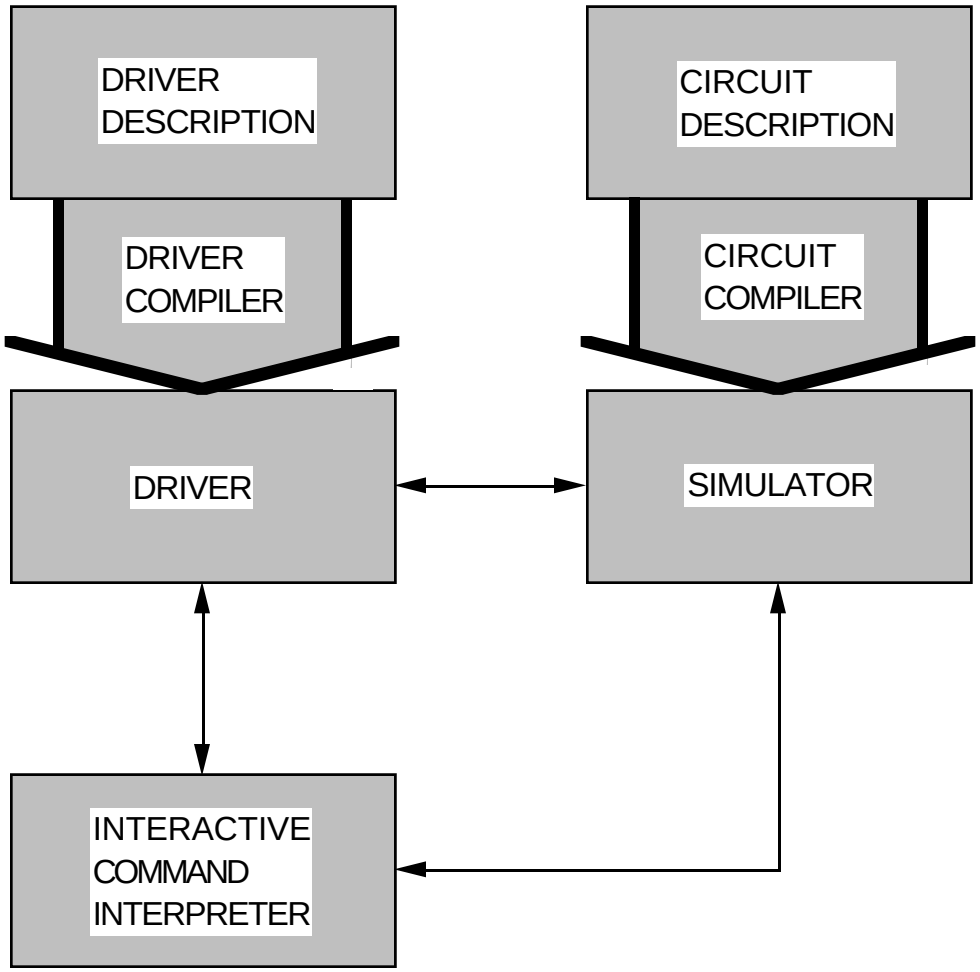

Figure 6. The Relationship of The Test Generator and The Simulator

As Figure 6 illustrates, every simulation must have a compiled driver, although in the default case it may be nothing more than a call to the interpreter. The driver language provides commands for interpreting command files and for invoking the interactive command interpreter. The compiled and interpreted versions of the driver language are identical.

Communication between the driver and the simulator is done via a combination of an interface routine and an interface data structure. In most cases the simulator itself plays the role of the interface routine. The interface data structure, which is illustrated in Figure 7, is passed as an argument to the interface routine every time a "go" statement is executed. The data structure is created by the interface routine during driver initialization. (When the interface routine is called with a "NULL" argument, it creates and returns a new interface data structure. This data structure is used as an argument for all subsequent calls.)

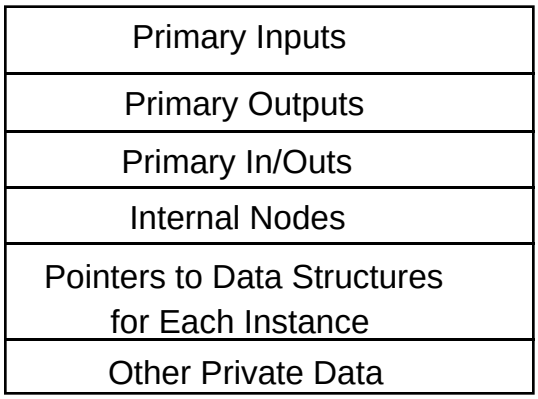

Figure 7. The Interface Data Structure.

An interface data structure represents one instance of a circuit. As Figure 7 illustrates, the data structure contains the values of all primary inputs and outputs, as well as the values of all internal nodes. Input vectors are passed to the simulator by assigning values to the primary-input portion of the data structure. Output vectors are placed in the primary-output section by the interface routine, which also performs any required transformation or transmission of the data. If a circuit contains instances of another circuit, the interface data structure will contain pointers to subordinate interface data structures for the instances. Figure 8 illustrates the data structure for a circuit A which contains two instances of circuit B, which in turn contains two instances of circuit C.

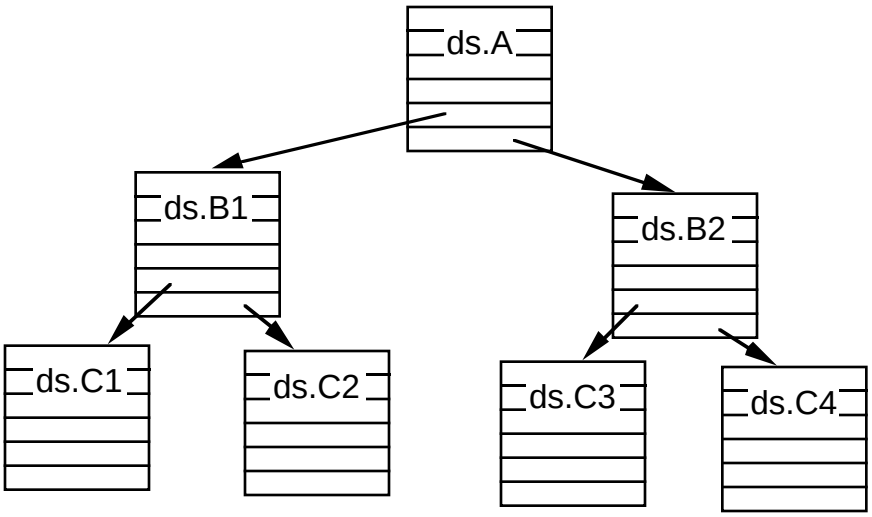

Figure 8. The Interface Data Structure for a Hierarchical Circuit.

All of the data structures pictured in Figure 8 are of the form illustrated in Figure 7. Because of the uniformity of these data structures, there is no real difference between an

instance of the primary circuit and an instance of a subcircuit. See [18] for a complete discussion of the interface data structure and its use.

The interface data structure provides a method for accessing primary inputs and outputs of a circuit, as well as all of its internal nodes. However, since the driver code references nodes by name rather than by position, it is necessary to create a dictionary that relates each name to a position in the data structure. The problem of creating the dictionary is complicated by the fact that the FHDL language allows subcircuits to be compiled independently, with a separate dictionary being created for each independently compiled circuit. A dictionary contains entries for primary inputs and outputs, internal nodes, and subcircuit instances. Each entry contains the name of the item, its type, and its position in the interface data structure. The dictionaries for the individual circuits must be linked to create a master dictionary for the entire circuit. Figure 9 illustrates the linking process.

Before Linking:

A's Dictionary	B's Dictionary	C's Dictionary
…	…	…
B1,instance-of-B,7	C1,instance-of-C,9	X,internal-node,3

After Linking:

A's Dictionary
…
B1.C1.X,internal-node,(7,9,3)

Figure 9. An Example of the Dictionary Linking Process.

Although it is not strictly necessary, the linking process retains the entries for subcircuit instances and for the primary inputs and outputs of each instance. This is done to facilitate the independent testing of subcircuits. The compilation process will supply an interface routine for each independently compiled circuit, but during normal simulation these routines are called only by the simulator for the main circuit. However, since the subcircuit interface is identical to the interface for the main circuit, independent simulation of subcircuit instances is a trivial extension of the technique for simulating the main circuit. Primary inputs and outputs are qualified with instance names just as are internal nodes. Thus if circuit C in Figure 9 has a primary input A, then instance B1.C1 of C has primary input B1.C1.A. The "simulate" command is used to simulate the instance without

simulating the circuit containing it. For example, to simulate the instance B1.C1, the following command would be used.

```
simulate    B1.C1
```

The simulate command is similar to the "go" command except that it uses the interface data-structure and routine for the instance rather than those for the main circuit. It is also possible to create new instances of a subcircuit that are independent of the main circuit. This allows subcircuits to be simulated independently without damaging the simulation of the main circuit. The following command creates a new instance of circuit C.

```
instance    newC,C
```

Internally this has the effect of creating a new interface data structure for C, and saving it in the instance-variable "newC." The circuit elements of the new instance are accessed using names of the form "newC.A." Although independent testing of subcircuits appears to be a powerful technique that will be quite useful for interactive debugging, we have not yet had enough experience to determine how effective it will be.

5. Conclusion.

Dynamic functional testing is based on the concept of vector generation with the added proviso that the vector generator must run simultaneously with the simulator and must be able to use the output of the simulator to guide the vector-generation process. The two main problems are the design of a language for vector generation and the design of the interface between the vector-generator and the simulator. Several guiding principles were used in design of the language, the most important of which is that the language should be similar to a high-level programming language, with added features for reading and writing vectors and for controlling the simulation. A sophisticated interface is used to pass data between the driver and the simulator. This interface is capable of supporting both interpreted and complied test generators, and can be used to interface with both compiled simulators and conventional interpreted simulators. The interface also supports the independent simulation of subcircuit instances in a reasonably natural way. The dynamic testing software described in this paper is currently being adapted to support the functional verification of WSI circuits, and has already been instrumental in debugging several simpler circuits. Dynamic testing appears to be a powerful tool that should significantly enhance the ability of designers to debug their circuits at the functional level.

REFERENCES

1. K. Adhoot, R. Alvarodias, L. Crawley, "IBM FSD Chip Design Methodology," *Proceedings of the 20th Design Automation Conference*, 1983, pp. 39-45.

2. C. J. Rimkus, M. R. Wayne, D.D. Cheng, F. J. Magistro, "Structured Design Verification: Function and Timing," *Proceedings of the 20th Design Automation Conference*, 1983, pp. 246-252.

3. K. Tham, R. Willoner, D. Wimp, "Functional Design Verification by Multi-Level Simulation," *Proceedings of the 21st Design Automation Conference*, 1984, pp. 473-478.

4. R. L. Wadsack, "Design Verification and Testing of the WE 32100 CPUs," *IEEE Design and Test of Computers*, Aug. 1984, pp. 66-75.

5. P. M. Maurer "The Design Verification of the WE 32106 Math Accelerator Unit," *IEEE Design and Test of Computers*, June, 1988, pp. 11-21.

6. D. Weise, "Functional Verification of MOS Circuits," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 265-270.

7. S. Devadas, H. T. Ma, A. R. Newton, "On the Verification of Sequential Machines at Different Levels of Abstraction," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 271-276.

8. J. D. Mastroianni, "A Verification System for Cell-Based Design Styles", *Proceedings of ICCAD*, 1983, pp. 211-212.

9. T. Lin, "The S-Algorithm: A Promising Solution for Systematic Functional Test Generation," *Proceedings of ICCAD*, 1984, pp. 134-136.

10. A. Kolodny, R. Friedman, T. Ben-Tzur, "Rule-Based Static Debugger and Simulation Compiler for VLSI Schematics," *Proceedings of ICCAD*, 1985, pp. 150-152.

11. H. T. Ma, S. Devadas, R. Wei, A. S. Vincentelli, "Logic Verification Algorithms and Their Parallel Implementation", *IEEE Transactions on CAD*, Vol. 8, No. 2, Feb. 1989, pp. 181-189.

12. A. S. Wojcik, "Formal Design Verification of Digital Systems," *Proceedings of the 20th Design Automation Conference*, 1983, pp. 246-252.

13 S. Bapat, G. Venkatesh, "Reasoning About Digital Systems Using Temporal Logic," *Proceedings of the 23rd Design Automation Conference*, 1986, pp. 215-219.

14. V. Stavridou, H. Barringer, D. A. Edwards, "Formal Specification and Verification of Hardware: A Comparative Study," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 197-204.

15. *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

16. K. Lai, "Test Program Compiler - A High Level Test Program Specification Language," *Proceedings of ICCAD*, 1983, pp. 30-31.

17. R. E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 9-16.

18. P. Maurer, Z. Wang, C. Morency, "Techniques for Multi-Level Compiled Simulation," Department of Computer Science and Engineering, University of South Florida, Tampa, FL, Technical Report CSE-89-04 (submitted for publication).