Verifying Large-Scale Multiprocessors Using an Abstract Verification Environment

Dennis Abts Mike Roberts

Silicon Graphics Inc.
Vector Systems Division
P.O. Box 4000, Chippewa Falls, WI 54729-0078
{dabts, roberts}@sgi.com

Abstract

The complexity of large-scale multiprocessors has burdened the design and verification process making complexity-effective functional verification an elusive goal. We propose a solution to the verification of complex systems by introducing an abstracted verification environment called Raven. We show how Raven uses standard C/C++ to extend the capability of contemporary discrete-event logic simulators. We introduce new data types and a diagnostic programming interface (DPI) that provide the basis for Raven. Finally, we show results from an interconnect router ASIC used in a large-scale multiprocessor.

1 Introduction

Modern scalable multiprocessors may have tens of millions of gates in a single node (Figure 1). Consequently, a minimal system simulation consisting of several nodes yields a very bloated and unmanageable verification environment. Historically we have used a special-purpose testbench constructed in a hardware design language (HDL) and simulated along with the original design to provide stimulus for the design under test (DUT). Unfortunately, this method creates a heavyweight logic simulation and, moreover, the size and complexity of the testbench can rival the logic design being verified.

The increasing complexity, and sheer enormity, of large-scale multiprocessors is impeding the design and verification process. In the past we sought innovative solutions to manage this complexity, including special-purpose cycle-based logic simulators. More recently, we have turned to new technologies such as formal verification (both model checking and equivalence) to mitigate this complexity. However, to be successful we need to exploit more traditional logic simulators in conjunction with other methods, such as formal analysis and verification techniques.

To address future verification challenges we propose using *Raven* as a verification environment suitable for verifying large-scale multiprocessors. We developed Raven to provide an extensible, abstract verification framework that augments traditional logic simulators, and is built upon standard C/C++ and a uniform diagnostic programming interface (DPI).

Related Prior Work

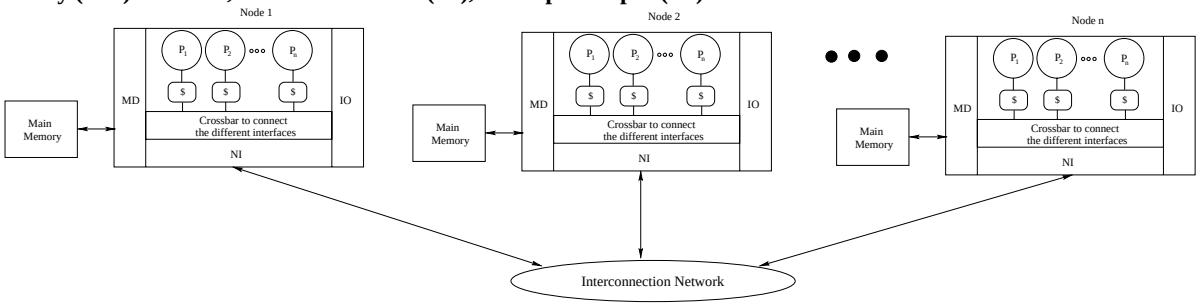
Managing complexity in the verification process is not a new idea in the hardware design community. Contemporary multiprocessors such as the SGI Origin2000 [1] have encountered design challenges, and addressed these obstacles with innovative methodologies [2] that include formal verification and traditional simulation. The verification demands of the Origin2000 project inspired the Keen-Michelson Language (KML) [3] which was developed to address the verification problem of a scalable node controller. However, KML is a special-purpose language requiring a large effort to adapt the verification environment to other design projects.

Several commercial tools available from Verisity Design and System Science Inc. extend the capability of current logic simulators to address the ASIC/IC verification problem. Verisity's *Specman* [4] provides an automated functional test generation environment from rules embodied in the design specification. System Science's *VERA* [5] hardware verification language (HVL) provides an abstract test development environment that replaces the traditional HDL-based testbenches with a special-purpose language to describe self-checking diagnostics.

Prior large-scale projects, such as the Cray T3E and SN1 (the follow-on to SGI Origin2000) involved constructing very complex testbenches around logic modules, typically single chips. For example the SN1 router chip testbench was written using an HDL (a combination of behavioral, RTL, and structural) with additional C code to support self-checking diagnostics. Directed and random diagnostics were written using this testbench; however, this heterogeneous verification environment, although effective, was unnecessarily complex and cumbersome.

Jones and Privitera use a formal specification to automatically generate functional test vectors for Rambus designs [6]. The formal specification completely describes the correct behavior of the device, from which they generate random and directed diagnostics using the RS language. The RS description expresses an abstracted operational interface of the design as opposed to the logical structure of the device. Since the RS specification is not suitable for simulation it must be transformed into Verilog and simulated with a netlist of the device. Although this approach proved valuable for verifying Rambus DRAMs it is not likely to be effective for more general-purpose designs that don't exhibit a high degree of regularity.

Figure 1: A typical node organization of a distributed shared memory multiprocessor, consisting of one or more processors, memory and directory (MD) interface, network interface (NI), and input/output (IO) interface.



2 Motivation

The verification problem is rooted in the exponential growth of IC technology. As design complexity and integration increase, the verification challenges are compounded. In the logic design process we use various types of *abstraction* to manage this complexity, for example hierarchical design methods. In much the same way, we want to employ abstraction to effectively mask complexity and implementation details during the verification phase.

To provide *complexity-effective functional verification* we need to leverage current discrete event logic simulators, without burdening the verification engineer who is writing the diagnostic. To accomplish this we established several goals for a verification environment.

scalability must allow verification at various levels, including: sub-chip level, single-chip, and multi-chip system simulations. As the logic design matures, the verification environment must grow also. It is unproductive to create "disposable testbenches" at each phase of the logic design.

standard language must provide for a familiar and contemporary high-level language for diagnostic development, such as C/C++. These programming skills are commonplace in most engineering teams.

decoupled diagnostic/simulator the diagnostic should not be bound to the simulator. By this we mean diagnostics can be quickly re-compiled without rebuilding the simulator.

uniform diagnostic programming interface the diagnostic must interact with the device under test (DUT) through a well-defined Diagnostic Programming Interface (DPI) to enhance diagnostic portability and ease-of-use.

automatic verification the verification environment must provide automatic validation of functional and temporal properties. That is, stimuli are injected into the logic design and the response is automatically validated against some expected results. An error condition causes the diagnostic to immediately terminate.

A hardware verification environment that encompasses the above attributes will provide a better framework for validating the correctness of future large-scale multiprocessors.

3 Design and Operation

The Raven verification environment consists of two major parts: a hardware simulator, and a diagnostic (test). The two parts are independent processes which can execute concurrently on a multiprocessor or time-share on a uniprocessor system, with synchronization and communication via sockets (Figure 2). By decoupling the diagnostic and the logic simulator we essentially create a client-server relationship between the diagnostic and simulator, where the interprocess communication (IPC) mechanism (sockets) delivers the stimulus and returns the results from the hardware simulator.

3.1 Architecture

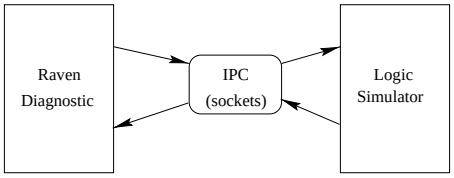
The Raven verification environment (Figure 3) is designed upon several layers of abstraction to mask the complexity and details of the logic design. This use of abstraction is analogous to an application program being abstracted from the actual physical hardware by several layers of operating system (API, user and system level calls, device drivers), or similarly a network application abstracted from the communication media by a communication protocol stack (e.g. layers of the OSI networking model). The logic design (usually RTL or gate-level HDL) of the DUT is the nucleus of the Raven environment. The *wrapper* consists of a shell around the logic that serves as an interface to the Raven kernel. The logic and wrapper are compiled into an object file, which is linked with the Raven kernel routines to create an executable logic simulator. The interprocess communication (IPC) layer provides the client-server interface between the *diagnostic program* and the logic simulator. The Raven kernel consists of a *diagnostic kernel* and a *simulation kernel* that lie on either side of the IPC layer. The Raven kernel provides very generic, low-level operations that are leveraged by the upper layers. The diagnostic programming interface (DPI) layer uses the kernel routines to provide a very well-defined and uniform interface to the diagnostic (test) developer. Diagnostic programs are written in C/C++ using the DPI.

The primary data object used by Raven is an *event*. An event is a very generic term used to describe two types of activities: signal transition (*wiggle*), and a sequence of signal transitions (*bundle*). A wiggle corresponds to a singular signal transition, whereas a bundle refers to an aggregate of transitions.

3.2 Raven Data Types

Raven extends the ANSI C/C++ basic data types (`char`, `int`, `long`, `float`, `double`) with data types needed to support hardware designs with two-state and four-state values that are common in most HDLs. In addition, several compiler macros are used to make the new types platform independent.

Figure 2: A diagnostic and logic simulator running under the Raven verification environment.



Portable integers

Heterogenous computing environments are very common resources, unfortunately the standard C++ integer types `int`, `short`, and `long` can vary from one compiler and hardware platform to the next. On most workstation or server-class machines, a `short` is 16-bits and an `int` and `long` are 32 and 64-bits wide, respectively. Similarly, on SGI machines using a 64-bit compiler a `short` is 16-bits and an `int` is 32-bits long. However, on Cray machines a `short`, `int` and `long` are all 64-bits. To prevent any compatibility problems Raven introduces a `big` data type which is defined as a 64-bit integer that is independent of the hardware platform.

Two-state Variables

Often in hardware designs we come across signals that are larger than the standard integer types. Raven simplifies the use of large data values by providing an arbitrarily large `num` type that closely follows the rules of the standard `int` type except that none of the standard modifiers like `unsigned`, `signed`, `const`, and `volatile` can be used in conjunction with a `num`. Raven provides the `NUM` macro for specifying `num` literals. A variable of type `num` is declared as follows:

```
num a;
num b = 1;
num c = NUM(0xdeadbeef);
```

Four-state Variables

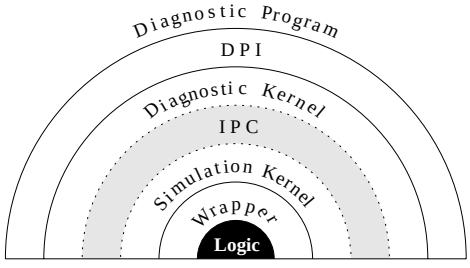
The `reg` data type provides a portable (machine independent) 256-bit four-state value with a *comparison mask*. The `reg` type is the standard type for holding simulation values. The `reg` type also has a `REG` macro that takes a four-state *literal*. The literal uses the standard format prefixes of either C++ (`0x` for hexadecimal, `0` for octal), or Verilog HDL (`'h` for hexadecimal, `'o` for octal, `'d` for decimal, `'b` for binary). The literal can be composed of valid digits for the base, along with unknowns (`x` or `X`) and high-impedance (`z` or `Z`) values. For clarity, the underscore (`_`) character can be included in the literal as a separator. A variable of type `reg` is declared as follows:

```
reg a;
reg b = "'b1111_0000_xxxx_zzzz";
reg c = REG(0x123456789abcdef);
```

All the arithmetic, logical, bit-wise, and relational operators work the same on `reg` types as they do on standard integers. For instance, it is perfectly legal to perform (given the above variable declarations): `a = b + c;`. Some operations, however, may not make sense on four-state variables, such as greater-than and less-than relational operators. If either variable contains `x`'s or `z`'s, the result would be undefined. In such cases we followed the convention of producing the same results as Verilog would.

Each `reg` also has a *mask* field to identify significant bits used when applying stimulus or validating results. The mask field is logically and-ed with the data before being applied or verified, effectively ignoring the non-significant portions. The user must set this mask accordingly to prevent *false error detection*; that is, a diagnostic fails because we are comparing bits of a register that are uninteresting. By default, all of the 256-bits of a `reg` value are considered significant, unless instructed otherwise by an assignment to the `mask` field of a `reg` type variable.

Figure 3: The Raven verification environment.



Signal Transition Events

The `wiggle` data type contains information to describe a signal transition. The `wiggle` type is the atomic unit for building more abstract types and is defined as follows:

```
class wiggle : public reg {
public :
    big delay;
    big time;
};
```

The semantics of the *delay* field depend on whether the signal is an input or an output of the simulator. If the signal is being applied as stimulus, then the delay field contains the minimum number of clock periods the signal will be delayed before being applied to the simulation. Otherwise, for a simulator output signal, the delay field is the number of clock periods that passed since the last output from the simulator. The *time* field is assigned by the Raven kernel as a time-stamp of the simulation time when the event is applied to or captured from the simulator.

Message Events

The `bundle` data type is an aggregate of `wiggle`s, and encapsulates all the necessary information to describe a *message*. The `bundle` abstraction can be thought of as a dynamically sized array of `wiggle`s. A message then, is a sequence of wiggles occurring on some communication channel.

Packet Events

The `packet` type is an abstraction of the `bundle` type, and can be viewed as a logical sequence of flow control units (flits) on a communication channel. For instance, we can define a packet, P as:

$$P = \{head, body^*\}$$

where $head$ is a flit describing the start of a packet P, $body$ is zero or more flits of payload for the packet. The packet format is governed by a communication protocol which the user defines.

3.3 Diagnostic Programming Interface

The diagnostic programming interface (DPI) provides the verification engineer with a high-level abstraction of the hardware design. The DPI makes extensive use of events as either arguments or return values of DPI primitives.

We formally define an *event* using the 5-tuple:

$$E = (node, location, name, event\text{-}id, type)$$

where $node$ is the node-id of a node in a multi-node system simulation (defaults to node 0 for unit-level or chip-level simulations), the *location* specifies the interface in the simulation at which the event is associated, *name* is a string to identify the event, *event-id* is a unique event number which is assigned by the Raven kernel to identify an event, and *type* is either *wiggle*, *bundle*, or *packet*.

Generating Stimulus

The `apply` primitive is used to create an event for delivery to the simulation. The `apply` primitive will submit an event to the Raven kernel for simulation. The basic form for `apply` is:

event-id = `apply(`*event*`);`

where *event* is the event to be applied as stimulus to the simulator, and *event-id* is a unique numeric identifier of the event. The `apply` operation is non-blocking, so diagnostic execution continues with the next sequential instruction in the diagnostic program.

Validating Simulation Results

The `verify` primitive is used as the `apply` counterpart to validate results from the logic simulator. The `verify` operation will attempt to match the *expected* event with the *actual* event received from the simulation. If a mismatch is detected during automatic verification, the diagnostic will display the error trace and terminate. The general form for `verify` is:

event-id = `verify(`*event*`);`

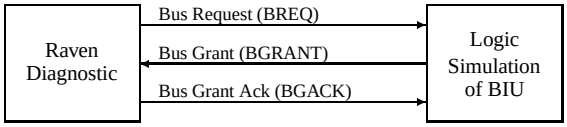
The `verify` operation returns a unique numeric identifier of the expected event, which can subsequently be used for event synchronization (`verify`, like the `apply` primitive, is a non-blocking operation).

Note, an unexpected event (one for which there is no outstanding `verify`) is also treated as an error condition. Raven also provides a user-definable *timeout* value to detect errors caused by indefinite postponement.

Explicit Synchronization

Often a hardware design will behave according to some protocol or interface that is reasonably well defined. For instance, consider a bus interface unit (BIU) of a microprocessor (Figure 4). The BIU is responsible for implementing the bus-level interface to a microprocessor. This interface requires some handshaking protocol for devices that wish to use the bus. To request the bus a device will issue a *bus request* (BREQ), and wait for a *bus grant* (BGRANT) response from the microprocessor. After receiving a BGRANT, the device will issue a *bus grant acknowledge* (BGACK) to inform the microprocessor that it successfully received control of the bus. A simple Raven diagnostic to validate this behavior is given below.

Figure 4: A diagnostic and logic simulator for a bus interface unit (BIU) of a microprocessor.




```
apply(BREQ);
verify(BGRANT);
await(BGRANT);
apply(BGACK);
```

Each of the events in this simple example would be a signal transition event (wiggle). Since `apply` and `verify` primitives are non-blocking it is necessary for the user to specify explicit synchronization points to resolve any dependencies among events. The `await` operation provides a barrier-like synchronization among events. In our example, it is necessary for the diagnostic to `await` the arrival of the BGRANT event prior to `apply`ing a BGACK. The `await` primitive is a blocking operation that causes diagnostic the thread of execution to halt until the event of interest has occurred.

Handling Spurious Events

During verification it is often essential to capture *speculative* events that may occur during the course of the simulation. These spurious events can be very difficult to detect using traditional simulation techniques. However, Raven provides a very flexible mechanism for handling these occurrences. The `trap` operation allows a user to define a *handler routine* which is invoked if an exception occurs. For example, after reset is performed by a chip it is undesirable for another reset to occur unexpectedly. The following Raven diagnostic will detect this anomaly.

```
/* preamble */
...
apply(TOGGLE_RESET);
verify(RESET_COMPLETED);
await(RESET_COMPLETED);
trap(RESET_ASSERTED, reset_error());
...
/* continue with diagnostic */
```

In this simple example, the TOGGLE_RESET event will cause the appropriate reset input pins to be asserted. Then we validate the reset operation actually occurs, by `verify`ing and `await`ing the RESET_COMPLETED event. Finally, we use the `trap` operation to catch an unexpected reset and invoke the user's reset_error routine if detected.

It is important to note that the diagnostic doesn't terminate when an event triggers the `trap` primitive. Raven merely notifies the diagnostic of the occurrence, similar to how the Unix operating system notifies a user process using *signals*.

Thread-level Parallelism

Instances of *identical replicated components* are common in modern hardware designs, for example, a communication switch with four identical input/output ports. Then, it is desirable to write a single diagnostic routine and apply it to each of the component instances in parallel. Moreover, the ability to concurrently apply stimuli to multiple components accurately models real-world applications.

The Verilog HDL [7] provides `fork` and `join` operations to create the appearance of parallel threads of execution within the simulation. However, using these operations to write modestly complex diagnostics is non-trivial. POSIX threads (pthreads) [8] offer another method of parallel execution. Unfortunately, pthreads are limited to a relatively small number of child processes. Moreover, use of pthreads would make a logic simulation environment *non-reproducible*. That is, if we did discover an error in the design using a pthreads approach it would be difficult, if not impossible, to recreate the circumstances that exposed the error.

Raven provides a novel approach to diagnostic thread-level parallelism using `parallel` and `merge` primitives. The general form for the `parallel` and `merge` operations is as follows:

```
parallel(user routine, args) ;
merge() ;
```

The `parallel` operation creates a new diagnostic thread beginning execution with the first line of code in the *user routine*. Any number of threads may execute concurrently. The Raven kernel manages the execution of all diagnostic threads providing *cooperative multitasking* among the parallel diagnostic threads. The `merge` operation blocks execution of the *parent* diagnostic thread until all its *child* parallel threads have completed. For example, consider a four port communication switch and the following diagnostic.

```
void send_msg(int port_num)
{
  for(i=0; i < NUM_MESSAGES_TO_SEND; i++)
  {
    ...
    MESSAGE.location = port_num ;
    apply(MESSAGE) ;
    verify(MESSAGE_COMPLETE) ;
    await(MESSAGE_COMPLETE) ;
    ...
  }
return 0 ;
} /* end of send_msg routine */


/* parent diagnostic */
parallel(send_msg, port0) ;
parallel(send_msg, port1) ;
parallel(send_msg, port2) ;
parallel(send_msg, port3) ;
merge() ;
/* wait for all threads to complete */
```

The above diagnostic will create four identical diagnostic threads, one for each communication port. The send_msg routine will send a predetermined number of messages to a specific port. The `merge` operator in the parent diagnostic blocks execution until all parallel diagnostic threads have completed.

Validating Nondeterministic Behavior

We are accustomed to designing very deterministic, predictable, well-behaved logic designs. However, there are occasions when *nondeterministic* behavior is preferred. Consider a four port communication switch that uses adaptive routing to circumvent a congested path. In such a device, a packet arrives at an input port and the communication switch will arbitrate among several possible output ports to route the outgoing packet. This behavior is particularly difficult to validate, since we may not really *know* which output port will be chosen. All we know is that *one* will be chosen.

To validate nondeterministic behavior, Raven provides a special form of the `verify(`*event*`)` and `await(`*event*`)` DPI primitives. When we expect the DUT to produce one out of n possible outcomes, we `verify` each of the n possibilities providing the *same event name* to each of the events being verified. Then, we can `await` the nondeterministic outcome from the design. The `await` primitive will return the event that was the outcome of the nondeterministic behavior. This idea can be best illustrated with a simple example. Consider a diagnostic to validate the four port communication switch with adaptive routing that we described earlier.

```
packet IN_PACKET ;
packet OUT_PACKET ;
packet CHOSEN ;

IN_PACKET.location = port0 ;
apply(IN_PACKET) ;
OUT_PACKET.location = port1 ;
verify("output", OUT_PACKET) ;
OUT_PACKET.location = port2 ;
verify("output", OUT_PACKET) ;
OUT_PACKET.location = port3 ;
verify("output", OUT_PACKET) ;
CHOSEN = await("output") ;

switch(CHOSEN.location)
{
 case 'port1':  ...
 case 'port2':  ...
 case 'port3':  ...
}
...
```

The above diagnostic will inject a packet into `port0` of the communication switch. Then, it expects the output to occur on one of the three output ports (`port1`, `port2`, or `port3`). We `verify` each of the OUT_PACKET events at each of the possible port locations. The named event "output" binds the three OUT_PACKET events. Then, we `await` the results from the design. The `await` operation returns the event which satisfied the nondeterminism, in this case, which output port was chosen.

Odds and Ends

Raven provides several other DPI primitives for such things as:

- setting the timeout value for a diagnostic with the scope being the entire diagnostic, an individual event, or an interface (port) on the design (`timeout`),
- controlling the verbosity of debugging and error trace messages (`debug`),
- reading and setting a value of a signal anywhere within the logic design (`sample` and `deposit`, respectively) which could be used for polling a bit within a register, for example,
- writing and reading the contents of a memory (`fill` and `extract`, respectively), and
- logging output to a file (`log`).

Although these miscellaneous DPI primitives are very useful, further exposition is unwarranted since the semantics of these operations are very intuitive and common to many HDLs.

4 Summary and Conclusions

Raven is currently being used to verify several large ASICs for next-generation multiprocessors. Our initial results are very encouraging. As a test case, we used Raven on a 1.1 million gate ASIC that routes packetized messages on the interconnection network of a scalable multiprocessor. Compared to an equivalent Verilog testbench, we expereinced a $\approx$10% performance penalty using Raven. This overhead is mostly attributed to the overhead in the interprocess communication (IPC) mechanism. However, this communication latency is easily overcome when the diagnostic contains a sufficient amount of computation that can be overlapped with the execution time of the logic simulator.

One consequence of developing diagnostics in C/C++ is the loss of visibility into lower-levels of the design under test. For instance, it is not intuitively obvious what logic is being exercised by a high-level operation such as `apply(`*event*`)`. To bridge this gap we use code coverage analysis tools, like Summit Design's *HDLScore* [10], to get a more in-depth look at how the design is being exercised by a diagnostic suite. The use of code coverage analysis tools in hardware design verification is analagous to the use of profiling tools used in software development.

Acknowledgements

We would like to express our gratitude to others who assisted in this work both directly and indirectly. Foremost we must thank Chris Koopmans for his contribution to the DPI. A special thanks to Tom Court, Abdulla Bataineh, Steve Scott, Greg Faanes, Michael Woodacre, Paul Frank, Scott Schroeder, and Galen Flunker for their insightful comments and support of the project.

References

[1] James Laudon and Daniel Lenoski, "The SGI Origin: A cc-NUMA Highly Scalable Server," Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97), p. 241–251.

[2] Ásgeir Th. Eiríksson, John Keen, Alex Silbey, Swami Venkataraman, and Michael Woodacre, "Origin System Design Methodology and Experience: 1M-gate ASICs and Beyond," COMPCON-97.

[3] John Keen and Jon Michelson, "How to Use the KML Language," SGI Internal Report.

[4] "Spec-based Verification: A New Methodology for Functional Verification of Systems/ASICs," white paper, Verisity Design web page: www.verisity.com

[5] Mehdi Mohtashemi, "High-Performance Functional Validation," white paper, System Science Inc company web page: www.systems.com/products/vera/vera.htm

[6] K.D. Jones and J.P. Privitera, "The Automatic Generation of Functional Test Vectors for Rambus Designs," Proceedings of the 33rd Annual Design Automation Conference, June 1996, p. 415-420.

[7] "The Verilog-XL Reference Manual," Cadence Design Systems, 1991.

[8] K. Robbins and S. Robbins, "Practical UNIX Programming," Prentice Hall, 1996. p. 347-364.

[9] "Synopsys VCS Reference Manual," Synopsys, Inc., July, 1997.

[10] Summit Design, Inc. web page: http://www.sd.com