Superlog, a Unified Design Language for System-on-chip

Peter L. Flake and Simon J. Davidmann

Co-Design Automation, Inc.
San Jose, CA 95113-1295
Tel : 408-718-1678
Fax : 408-273-6025
e-mail : flake@co-design.com,
simond@co-design.com

Abstract - The design of systems consisting of custom software controlling custom digital hardware is easier if a single language can be used for system specification, software development, hardware design and hardware verification. Superlog takes features of existing languages for software development and hardware design, adds features for system specification and hardware verification, and blends them into a single, coherent language .

I. Introduction

Currently we have system specification languages such as SDL [1], we have hardware description languages such as Verilog [2] and VHDL [3], we have testbench languages such as Vera [4] and we have programming languages such as C. This mixture means that the design intent is often re-coded in different languages, making verification and maintenance difficult and introducing bugs.

A single language for all four purposes would reduce this re-coding and make it easier to re-use code from one part of the design flow in another part. This would speed up the design process and reduce the number of bugs.

There has been much talk about specifying hardware in C. If it were easy to describe hardware in C, hardware description languages would not have become popular. It is common to code testbenches in C, yet testbench languages are selling. Many system architects write C models of their system, yet there is also a market for more specialized languages.

II. Related Work

There is a long history of adapting imperative programming languages to describe hardware. Nowadays the favorite programming languages are C, C++ and Java [5]. Derivatives of C include Hardware C [6], Handel C [7], Spec C+ [8] from UC Irvine, and Esterel C [9]. C++ has the advantage that a class library can almost create a new language by changing the meaning of operators, but this can make it hard to understand. The Scenic [10] environment, which has developed into SystemC from Synopsys, is an example.

In spite of the attempts to adapt programming languages to hardware description, most hardware design is done using hardware description languages, which are dedicated to the purpose. In the 1970s DDL and ISP [11] were just academic languages. In the early 1980s, gate arrays created the need for logic simulation, and the simulators usually had four languages (or formats) - netlist, stimulus, modeling and simulator control. Hardware description languages initially combined the netlist and modeling languages but kept different stimulus and control languages e.g. HILO [12]. VHDL combined the netlist, modeling and stimulus languages, and Verilog included the simulator control language as well.

With the advent of synthesis there were attempts to move back to languages which only described hardware and hence were synthesizable, e.g. ELLA, UDL/I. This approach was overtaken by the synthesizable subset of a simulation-based language, and now Verilog or VHDL are used for the majority of large designs.

The complexity of designs, and hence testbenches, has increased. Now specialized testbench languages have appeared with 'software-like' features. Indeed many people use the programming language interface (PLI) to Verilog simulators to write testbenches in C or PERL, sometimes with the help of a threads package.

There are hardware verification languages such as Vera from Synopsys and 'e' from Verisity. These are currently proprietary but Vera's reported features include dynamic process creation and sequence checking.

This is partially because Verilog is poor as a general-purpose programming language. It has no dynamic memory allocation (not even a stack let alone a heap), poor built-in

input, and text processing is extremely difficult.

VHDL has more programming functionality than Verilog, but it is hampered by three problems: the strict and complex data type system, the limited inter-process communication and the fixed number of processes. Some of these problems are being tackled by proposals to extend VHDL, but they tend to complicate an already complex language.

There are also system design languages. Some of these are standards in particular industries, such as SDL in telecom. Some are proprietary, often hidden behind graphical interfaces. Most are not suitable for designing hardware or for writing general embedded software.

III. Requirements

A new language should cover system specification, software development, hardware design, and hardware verification. In the embedded software development field, C is the most widely used language, although there is some use of C++, Ada and Java, and still some use of assembly code. In the hardware design field Verilog and VHDL are both popular. Looking at the language styles, the natural pairings are C with Verilog and Ada with VHDL. In the system specification and hardware verification fields, there are various languages in particular market segments, but no overall dominance greater than C.

The minimum new language, therefore, should contain Verilog and C functionality. However, these languages have limitations in system and testbench modeling, which is why testbench languages have appeared. The following features are examples:

(a) Create and destroy processes like an operating system
(b) Check that changes occur in a particular sequence.
(c) Apply behavioral code to various I/O ports

In addition, there are some general hardware design features that would be useful in Verilog, some of which are being considered by the Verilog 2000 committee:

(d) A 'generate' statement like in VHDL
(e) Avoiding the repetition of ports in hierarchical designs

There are also improvements that could be made to C for general programming purposes as well as for system modeling:

(f) Dynamic arrays, support for queues
(g) String handling

Thus, a single language that can meet all these requirements should give a substantial improvement in productivity.

IV. Overview

As its name implies, at first sight Superlog looks like Verilog, as shown in Fig. 1. Braces '{}' are used for data but not for code. The 'begin .... end' or 'fork .... join' from Verilog is used to emphasize the sequential or concurrent execution of statements.

```
// A tree search routine showing software
features

typedef struct {string s;
  ref node left, right;} node;

// global data
ref node n, root;// pointers to nodes
int visited = 0;//number of nodes visited

function ref node find(
  string str, ref node parent);

  if (parent == null) return null;
  visited++;
  if (str == parent->s) return parent;
  if (str < parent->s) return find(
    str, parent->left);
  else return find(str, parent->right);
endfunction

//  state  machine  showing  new  hardware
features

module  FSM4(input  logic  serial,  clock,
reset);

state {S0, S1, S2} currentState;

always_ff @(posedge clock iff !reset)
  transition (currentState)
    S0:if (serial == 1) ->>S2;
    S2:if (serial == 0) ->> S1;
      else ->> S0;
    S1: ->> S0 n = find("a", root);
  endtransition

endmodule
```

Fig. 1 Superlog code fragment

Superlog is not a strict superset of Verilog. Some little-used features such as quasi-continuous assignments have been removed, as have the switch level features.

Superlog assumes an event-driven model. The basic data communication mechanism between processes is the shared variable, like a Verilog 'reg', but resolution functions can be used to model wire behavior.

V. Data Types and Declarations

Superlog contains both C and Verilog built-in data types. Like Java, it has a byte data type, which is guaranteed to be 8 bits, whereas char is not, and a long data type, which is 64 bits. Like VHDL it has a bit data type which is 0 or 1, as

distinct from the logic data type which is 0, 1, X or Z as in Verilog. These are called 'unmasked' and 'masked' data types respectively.

Like C and VHDL, Superlog has user-defined data types, which can be enumerations, structures, pointers or arrays. Like C, it also has unions. User-defined data types are introduced by 'typedef' as shown in Fig. 1.

Data declarations follow the C syntax of qualifier, type and instances with optional initializers, as shown in Fig. 1

Arrays are of five types: packed, unpacked (fixed and variable length), sparse and associative. Packed arrays are like Verilog registers and can be written or read in a single action. Slices (part selects) can also be written or read. Variable length arrays are useful for modeling queues.

A new kind of 'state' declaration is provided for defining state machines, as shown in Fig. 1. This is more than just an enumerated type, because the state names have variable values.

VI. Operators and Expressions

These essentially follow the rules of Verilog and C, which are similar in most cases. In both languages, the type and size of the operands are fixed, and hence the operator is of a fixed type and size. This allows efficient code generation, and Superlog supports it.

Unlike VHDL, Superlog allows data types to be easily converted. When a masked value is converted to an unmasked value, the X or Z is converted to 0. An operator that has a masked input is a masked operator.

VII. Procedural Statements and Control Flow

Superlog has the C and Verilog control constructs. A new 'transition' construct is provided for synchronous state machines, to facilitate their recognition by synthesis and verification tools. This is like a Verilog 'case' statement but the cases are limited to state names belonging to the state machine identified in the transition statement. The transitions to a new state are indicated by '->>', as shown in Fig. 1.

The example also shows a conditional event expression, with the 'iff' keyword, which in this simple case provides shorthand for:

```
do @(posedge clock); while (reset);
```

Conditional event expressions can be combined with 'or' to provide complex event control.

In addition, Superlog has an assertion that an expression is true like in VHDL. It also has a construct to check a sequence. This enables a simulator to automatically trap an illegal sequence error, and enables a stimulus generation tool to keep within the constraints of a protocol.

VIII. Functions and Tasks

The Verilog distinction between functions, which cannot take simulation time, and tasks, which can, is preserved in Superlog. Note that VHDL has a similar distinction, which allows expressions to be guaranteed instantaneous evaluation. Because of its inclusion of C data types, it is easy for Superlog to call C functions.

In Verilog, tasks cannot have event expressions containing arguments, nor can they write directly to arguments during execution. Superlog overcomes these limitations, allowing arguments of mode 'port'.

To communicate with foreign languages such as C, there are 'import' and 'export' statements for functions and tasks.

IX. Processes

Superlog provides specialization of the 'always' construct to indicate processes that are synthesizable to combinatorial logic, sequential logic with latches, or sequential logic with flip-flops. The Verilog fork-join construct allows the structured creation and destruction of processes, but this is inadequate for modeling an operating system, or even a hardware pipeline. Superlog therefore allows unstructured creation and destruction of processes.

X. Modules and Interfaces

Superlog provides enhanced hierarchical structures compared with Verilog. Module declarations can be nested, providing better control of name space. To encapsulate connectivity and communications, a new entity called an interface can connect module instances. The simplest example of an interface is a bundle of wires. Interfaces can also contain variables, functions and tasks to model a bus at a more abstract level.

Data, functions and tasks can also be defined outside modules and interfaces, which makes them global in scope.

For Verilog compatibility and performance, there are built-in module types (gates) and built-in interface types (wires).

XI. File Structure

Superlog follows the Verilog file structure, where all source files are concatenated and compiler directives persist across file boundaries. The usual directives are supported, including macros with parameters.

XII. Concluding Remarks

The Superlog language contains the functionality of:
- Verilog for hardware design
- C for software
- Direct calling of C for libraries and third party software

And new constructs:
- Interfaces to encapsulate communication
- Sequence checking for protocols
- State machines for designing control logic
- Dynamic processes for modeling real time software

Thus Superlog contains much more functionality than VHDL, but with a simpler type system to simplify the language overall.

The mixture of languages currently used (Verilog, C, VHDL, etc) makes the design of complex systems on a chip much more painful than it need be. Superlog streamlines the design flow by providing one unified language that can specify systems, hardware, testbenches and software. The benefit is higher productivity and quality due to more re-use of code through the various stages of the design flow.

Acknowledgements

The authors wish to thanks the members of Co-Design's Technical Advisory Board, Phil Moorby, Don Thomas and Mike McNamara, for their comments during the development of Superlog, and the other members of the company who have contributed to its specification and implementation.

References

[1]A. Olsen et al., *Systems Engineering using SDL-92*, Elsevier 1994.
[2] *IEEE Std 1394 - Hardware Description Language Based on the Verilog Hardware Description Language*, IEEE 1996.
[3] *IEEE Std 1076 - 1993 IEEE Standard VHDL Language Reference Manual*, IEEE 1994.
[4] S. Al-Ashari, "System Verification from the Ground Up" *Integrated Systems Design*, January 1999.
[5] R. Helaihel, K. Olukotun, "Java as a Specification Language for Hardware-Software Systems", *Proc. IEEE/ACM ICCAD*, Santa Clara, 1997.
[6] *Hardware C - A Language for Hardware Design* CSL Technical Report CSL-TR-90-419, Stanford University, April 1990.
[7] M. Aubrury and M. Sauer, *Hardware/Software Co-simulation for a Rapid Prototyping Environment*, Hardware Compilation Group, Oxford University Computing Laboratory, UK, August 1997.
[8] J. Zhu, R. Domer, D. D. Gajski, *Syntax and Semantics of the Spec C+ Language*, Technical Report ICS-97-16, Dept of Information and Computer Science, UC Irvine, April 1997.
[9] L. Lavagno, E. Sentovich “ECL: A Specification Environment for System-Level Design”, *Proc. ACM/IEEE DAC*, New Orleans, LA, 1999.
[10] S. Liao, S. Tjang, Rajesh Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment" *Proc. ACM/IEEE DAC*, Anaheim, CA, 1997
[11] S. J. Davidmann, *Multi Level Simulation in Digital System Design*, MSc Dissertation, 1980, University of Essex, UK
[12] P. L. Flake, P. R. Moorby, G. Musgrave, "HILO Mark 2 Hardware Description Language" *Proc. ACM/IEEE CHDL*, Kaiserslautern, Germany, 1981.