TECHNICAL PUBLICATION

Implementing SystemVerilog for FPGA Design

Ehab Mohsen, Technical Marketing Engineer
Design Creation and Synthesis Division
Mentor Graphics Corporation
June 2008

Mentor Graphics®

INTRODUCTION

Since its ratification in 2005, the SystemVerilog IEEE-1800 standard has come a long way in terms of adoption — though this adoption has not been even-handed. The specification can be roughly broken into several categories: assertions, testbench constructs, DPI and API interface, and design enhancements. The verification constructs were the first to get industry-wide attention, as shown with the high profile methodologies such as the Verification Methodology Manual (VMM), the Advanced Verification Methodology (AVM), and Open Verification Methodology (OVM) — all to address the verification bottleneck.

And yet the design implementation bottleneck deserves equal attention. One study on FPGA design reports that RTL logic design consumes roughly half of the design cycle time, with verification consuming the other half. In addition, roughly 60% of projects are delayed due to changes in the design specification. In other words, RTL coding methods are far from perfect and significant in schedule impact. Recent enhancements to SystemVerilog are intended to address these issues.

With the ability to design at higher levels of abstraction, compactness of code, and the unification of design and verification environments, the new SystemVerilog extensions offer an improved method of logic implementation, relevant at nearly all RTL designers.

ADOPTING SYSTEMVERILOG FOR DESIGN

But engineers are wary of change, particularly when it involves adopting a new design language. Changing to a new language not only implies starting from scratch, but convincing others in the design team to start from scratch. Changing also means waiting for EDA tools to provide adequate support of the language. Most designers do not have the bandwidth, freedom, or comfort level to take this leap.

The truth is, however, adopting SystemVerilog for design is not a leap at all. SystemVerilog can (and should) be thought of as an extension to the existing Verilog standard with the same basic syntax and backward compatibility. In addition, there are synthesis tools available that support the synthesizable constructs of the language. In fact, Precision® Synthesis has long supported SystemVerilog for design with customers already implementing designs into real hardware. Adapting current design methods to make use of SystemVerilog is not the risky revamp or makeover most engineers assume it to be.

In a real sense, the risk is sticking with the same design methodology while chip size and complexity continue to increase. For example, average gate count from one FPGA project to the next increase on average 25 percent. Managing this increase in complexity was among the goals of the authors of the SystemVerilog specification. In fact, the charter for the new standard was to "extend Verilog IEEE 2001 to higher abstraction levels for architectural and algorithmic design, and advanced verification."

THE NEW CONSTRUCTS

This paper reviews a few of these constructs added to make the language more design friendly. While the language has a variety of extensions, the sub-set examined here focuses on those that allow higher levels of abstraction for RTL design. Each construct builds on the other, with the degree of abstraction and complexity left to the designer.

User-Defined Types

As a starting point, SystemVerilog allows the definition of new data types based on existing types with the **typedef** keyword, similar to C (example shown below). This construct is a fundamental building block in modeling a complex design at abstract levels while still being accurate and synthesizable. Once it is defined, the user-defined type can be used as with any built-in data type, making data structures more readable:

```
typedef  logic [1:0] opcode_t;
opcode_t [1:0] op1;
```

Note: *The example above follows the naming convention of ending a user-defined type with the characters "_t". This convention is done to improve readability and maintainability, since the user-defined type may have been defined elsewhere in the code or in another file.*

The examples in this paper use the **logic** data type — a replacement to the **reg** data type, as the latter misleadingly implies a register is to be inferred. In fact, **reg** is a variable and may be implemented as sequential or combinational logic, depending on its use. The use of **logic** over **reg** is recommended to avoid designer confusion. There are other differences between the **reg** data type from Verilog-2001 and the **logic** type of SystemVerilog, but they are beyond the scope of this paper.

Enumerated Types

Similar to VHDL's enumeration data type, enumerated types in SystemVerilog provide the ability to declare a variable with a specific list of valid values:

```
enum {ADD, SUB, MULT} opcode
```

The variable **opcode** is given a set of potential user-define names "ADD", "SUB", and "MULT" (also known as labels). An example of this use is in the following case statement (assume a, b, and c have been defined as variables):

```
...
 case (opcode)
   ADD:  c = a + b;
   SUB:  c = a - b;
   MULT: c = a * b;
 endcase
...
```

The code shown above is readable and intuitive. One might argue that the same can be achieved in Verilog with use of the **`define** macro or parameter constants to define a set of names with specific values:

```
'define ADD 2'b00
'define SUB 2'b01
'define MULT 2'b10
...
reg [1:0] opcode;
...
 case (opcode)
   'ADD:  c = a + b;
   'SUB:  c = a - b;
   'MULT: c = a * b;
 endcase
```

In the Verilog example above, the variables are defined as 2-bit vectors; therefore, they can legally be assigned the value 2'b11 — a value not accounted for in the above code — forcing designers to be aware of this potential error. So in addition to being more concise, enumerated types not only define a list of user-defined labels for a variable but also limit the variable to the set of valid values specified. By offering stronger type checking in the language compiler, the code accurately captures designer intent.

Enumerated types can be used with the **typedef** keyword to create user-defined types with a set of valid values. This technique is useful when needing to make declarations in many places:

```
typedef enum {ADD, SUB, MULT} opcode_t;
opcode_t op1, op2;
```

Figure 1 shows how using strong typing can catch potential errors.

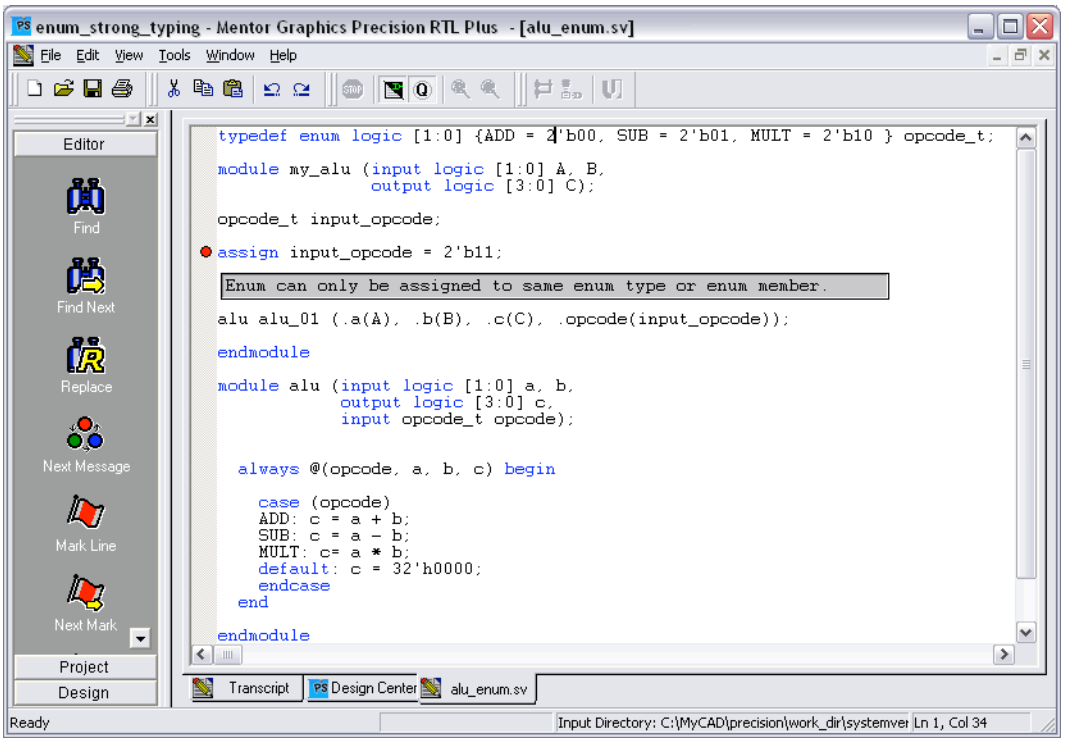

Figure 1: *Strong Typing Using Enum Data-Type*

Structures

SystemVerilog offers structures as the next level of abstraction. Structures are a means of aggregating variables or data fields under a common name, usually those conceptually related in some way. With the **struct** keyword, users can intuitively describe data structures and mask lower-level complexity as needed. Common applications for this include instruction registers, network packets, and bus packets.

```
struct {
logic [1:0] opcode;
int a, b;
logic [23:0] addr;
} instruction;
```

This construct is similar to C and VHDL's **record** construct. In Verilog, the members of a structure have to be defined as separate variables and managed individually, forcing the engineer to track how all the pieces fit together. By aggregating elements, data structures can be represented in the HDL as they are understood conceptually.

In the example above, **instruction** is the structure that aggregates the data elements **opcode**, **a**, **b**, and **addr**. Any element of the structure can be accessed and manipulated individually, for example:

```
instruction.opcode = 2'b00
```

An entire **struct** can be a user-defined type to be re-used throughout in the code — effectively creating higher-level building blocks for design implementation.

```
typedef struct {
  opcode_t opcode;
  int a, b ;
  logic [23:0] addr;
} instruction_t;
instruction_t instr1, instr2;
```

Structures can also be passed through tasks, functions, and module ports, allowing abstraction to be maintained through data flow:

```
assign instr1 = instr2;
```

In the example above, the user-defined type **opcode_t** is a member of the user defined type **instruction_t** — an example of how multiple layers of abstraction can be created in the RTL. Implementing such multiple levels of abstraction in the design does not significantly impact performance or area during synthesis. Structs are ultimately the bits and bytes described in their original user definitions, and the synthesis tool will implement them as such. There is no loss of specificity when compared to a Verilog or VHDL description.

Figure 2 and **Figure 3** show a simple ALU implemented in both Verilog and SystemVerilog using structures. This simple example demonstrates that the RTL descriptions are equivalent.

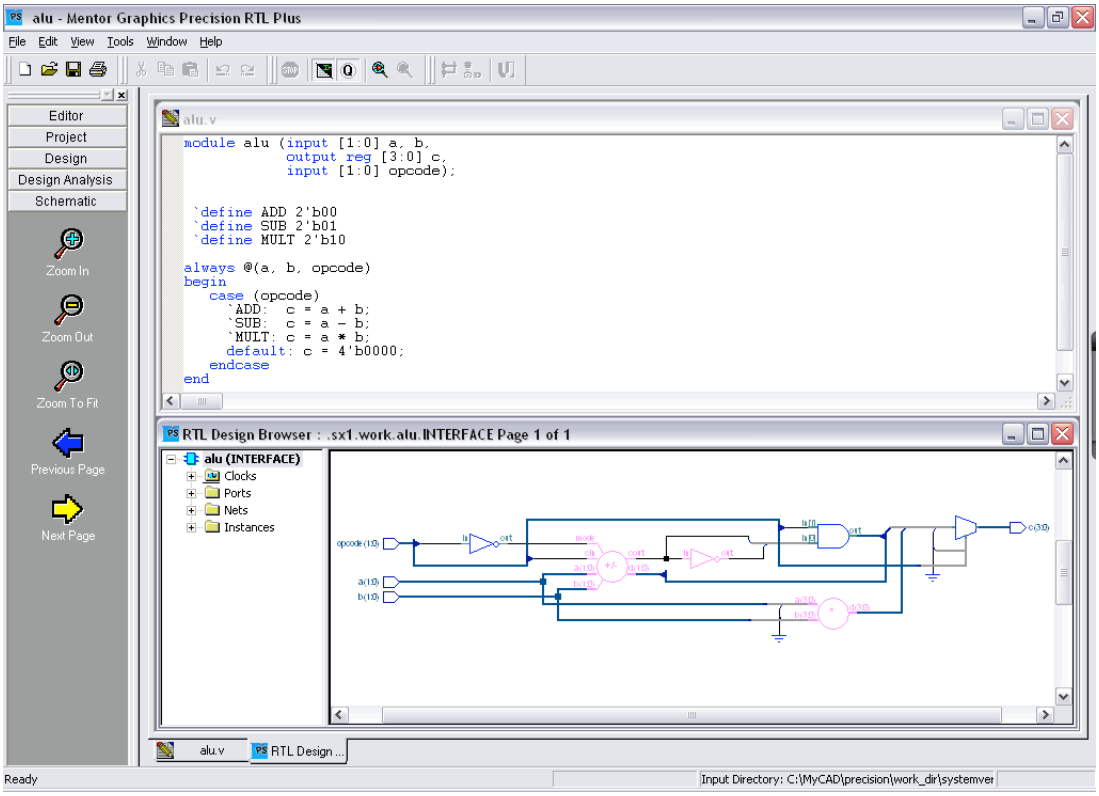

Figure 2: Simple ALU implemented in Verilog

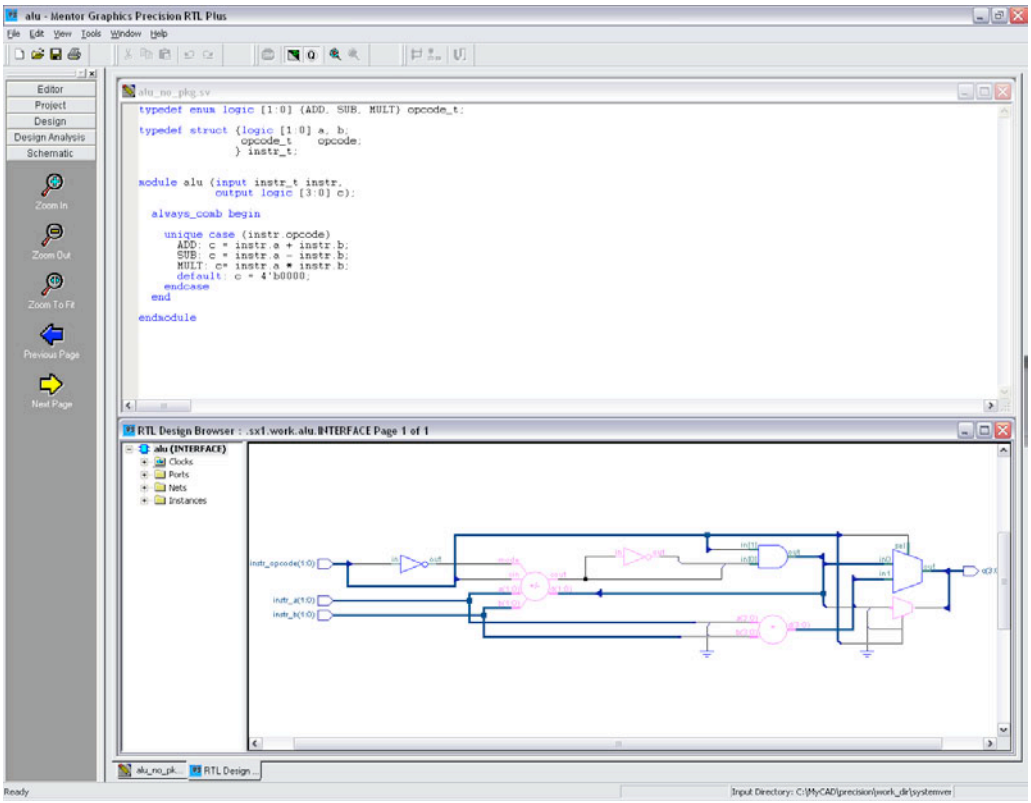

Figure 3: Simple ALU implemented with SystemVerilog Structures

Unions

Unions enhance **structs** or any data model by allowing multiple definitions for a single storage element. Each definition has to have the same storage space (e.g., vector length, memory size), but different data types can be used. In the following example **[Ref 6]**, the same storage space is defined as the struct **data** with three members (**source_address**, **destination_address**, and **data**) and in another definition as a two-dimensional array **bytes**. Both of these definitions represent a 64-bit storage space identified as the union **data_reg**. Elements of the union are assigned values depending on the definition used (**data** or **bytes**).

```
union packed{
struct packed {
bit [15:0] source_address;
bit [15:0] destination_address;
bit [31:0] data;
} data;

bit [7:0][7:0] bytes;

} data_reg;

data_reg.data = data_in; //assumes
data_in is a 32-bit vector

dest_low_byte = data_reg.bytes[4];
```

The example above briefly introduces the concept of *packed* structures and unions. Packed arrays refer to all elements being represented as contiguous bits, i.e., as a single vector.

Interfaces

While user-defined types, structures, and unions allow for abstract data *modeling*, they do not provide a complete mechanism for the abstraction of data *flow*. The **interface** construct solves this problem by encapsulating port connectivity and functionality related to module-to-module communication. More than just the high-level modeling of data structures, interfaces raise the level of communication protocols within a design, reducing the size of code and improving readability and maintainability.

Figure 4 shows a system-on-chip (SoC) design **[Ref 7]** with processor (CPU), UART, and parallel input/output (PIO) interface, all communicating through a bus interface following the wishbone specification.

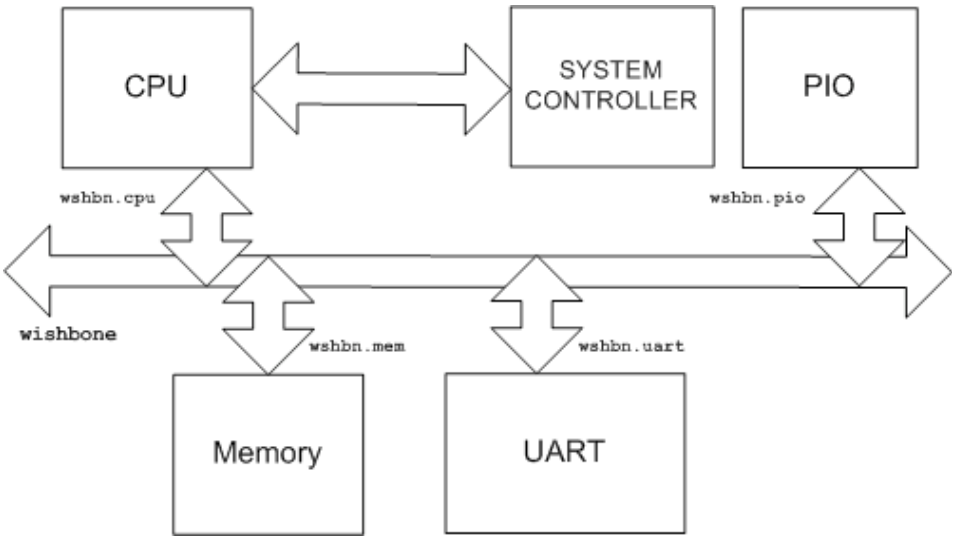

Figure 4: *SoC Design from OpenCores.org*

An interface is defined similarly to a module but with the **interface** keyword (**Figure 5**). The set of signals to be bundled in the interface are declared without input or output direction. Since each block in the SoC sees a port differently (as input, output, or bi-directional), the subset of signals and port directions for each design block is defined in **modports** — each modport having a unique name. Interfaces support parameters, constants, tasks, functions, procedural blocks, program blocks, assertions, as well as any user-defined type.

When declaring a module in the design, an instance of the interface is used as a single module port, using the name of the interface as the port type. Signal details are masked with this single port declaration, as is all functionality embedded in the interface. In the UART module (**Figure 5**), individual signals are referenced within the module as needed.

One benefit of using interfaces is the bundling of wires in one location, thereby masking inter-connectivity, reducing lines of code, and improving readability. The more significant benefit is being able to localize communication logic between design blocks. The protocol details do not need to be duplicated in multiple modules. As a result, when the inevitable change of the interface specification is made, modification are minimized. These changes may include the addition or omission of a signal, a correction to a protocol error, or an entire revamp of the bus specification itself.

If logic is actually described in the interface (which is actually the case for this design, though not shown above), the interface appears as an actual logic block in Precision Synthesis (**Figure 6**), allowing cross-probing between schematic and HDL or traversing the hierarchy as with any other design module.

```
interface wishbone(input logic CLK, RST);
  logic[31:0]  DAT;
  logic[3:0]   SEL;
  logic        WE;
  logic[31:0]  ADR;
  .. .. ..
//modports
  modport cpu (output ADR, input DAT, ..)
  modport uart( input WE, input SEL, ...)
  .. .. ..
  //embedded functionaliy...
    always @(ADR) begin
      if (ADR[17] == 1'b0)
  .. .. ..
endinterface
```

```
module cpu (input CLK, input RST,…,
            wishbone.cpu wshbn_cpu);
.....
```

```
module uart (input CLK, input RST,…,
             wishbone.cpu wshbn_uart);
.....
  if ((wshbn_uart.WE == 1'b1) && ....
```

```
module top( input CLK, input RST, .....);
  .....
  wishbone wishbone(CLK, RST)

  cpu CPU(CLK, RST, .wshbn_cpu(wishbone.cpu));

  uart UART(CLK, RST,.uart_bus(bus.uart));
  .....
```

Figure 5: *Example of the Interface Construct*

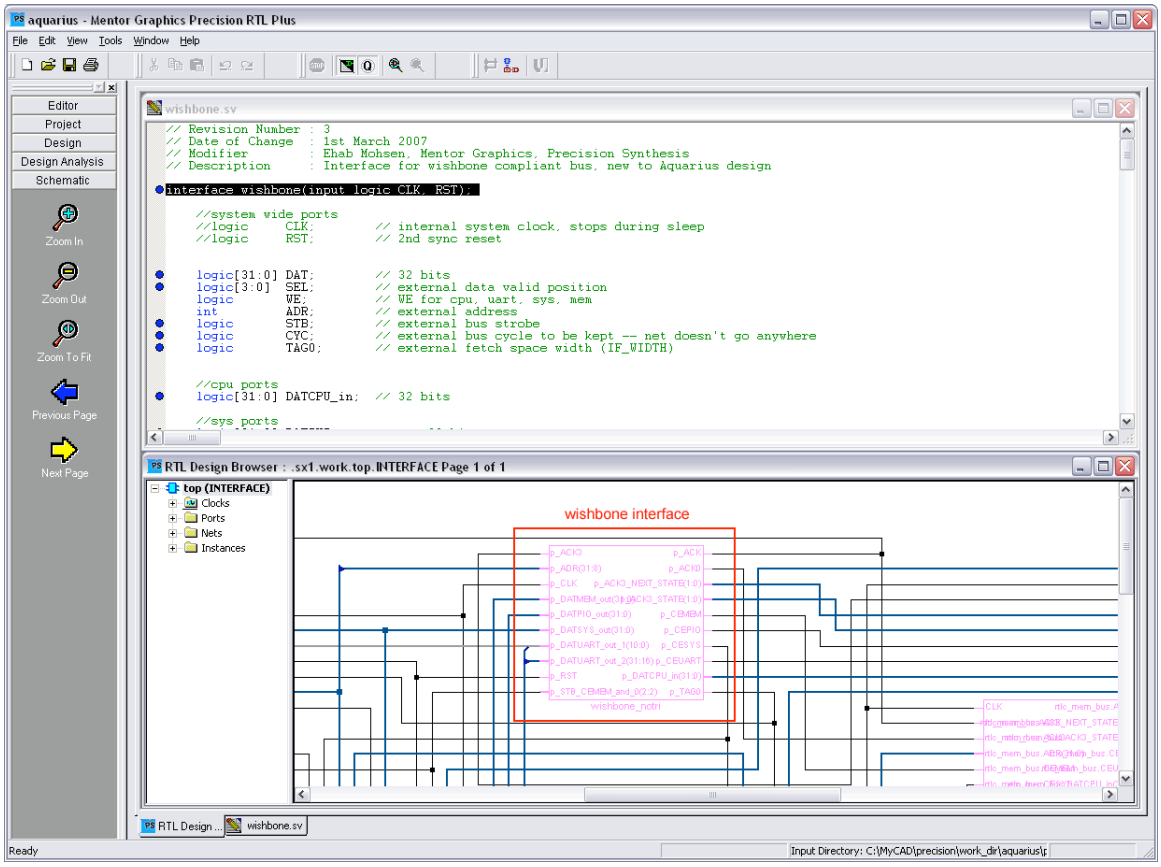

Figure 6: *Interface HDL and Schematic in Precision System*

CONCLUSION

The constructs covered here are enough to justify a fair and close look at SystemVerilog for design, but the enhancements do not stop here. The standard provides a wide range of other extensions including improvements to tasks, functions, loops, procedural blocks, and the addition of new operators, jump statements, and packages.

SystemVerilog is more of an evolution than revolution in RTL design and does not require a major paradigm shift or complete abandonment of existing RTL design methodologies. It is common for companies to look at the constructs one-by-one and deploy them incrementally as a more conservative approach. Even with this method, designers can realize the standard's benefits.

REFERENCES

1. EE Times *Electronic Design Automation Branding Study,* FPGA Design
2. *SystemVerilog enhancements for all chip designers*, Stuart Sutherland, EE Times, 2/26/2004
3. DAC 2003 Accellera SystemVerilog Workshop
4. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*, Stuart Sutherland, Kluwer, Academic Publishers, Boston, MA, 2004, 0-4020-7530-8.
5. IEEE Std. 1800-2005 IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language, IEEE, 3 Park Avenue, NY, 2005.
6. *Getting Ready for SystemVerilog at DAC [2004]*. Presentation, Stuart Sutherland, Sutherland HDL, Inc.
7. OPENCORES Project, opencores.org

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000

Silicon Valley
Mentor Graphics Corporation
1001 Ridder Park Drive
San Jose, California 95131 USA
Phone: 408.436.1500
Fax: 408.436.1501

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
Room 1001, 10F
International Trade Building
No. 333, Section 1, Keelung Road
Taipei, Taiwan, ROC
Phone: 886.2.87252000
Fax: 886.2.27576027

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Hills
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140
Japan
Phone: 81.3.5488.3033
Fax: 81.3.5488.3021

Printed on Recycled Paper

05-08-BB 5262: 080529