Bucknell Handbook on Verilog HDL

Dr. Daniel C. Hyde
Computer Science Department
Bucknell University
Lewisburg, PA 17837
August 25, 1995

Table of Contents

1. Introduction

Verilog HDL is a **Hardware Description Language (HDL)**. A Hardware Description Language is a language used to describe a digital system, for example, a computer or a component of a computer. One may describe a digital system at several levels. For example, an HDL might describe the layout of the wires, resistors and transistors on an **Integrated Circuit (IC)** chip, i. e., the **switch level**. Or, it might describe the logical gates and flip flops in a digital system, i. e., the **gate level**. An even higher level describes the registers and the transfers of vectors of information between registers. This is called the **Register Transfer Level (RTL)**. Verilog supports all of these levels. However, this handout focuses on only the portions of Verilog which support the RTL level.

1.1 What is Verilog?

Verilog is one of the two major Hardware Description Languages (HDL) used by hardware designers in industry and academia. VHDL is the other one. The industry is currently split on which is better. Many feel that Verilog is easier to learn and use than VHDL. As one hardware designer puts it, "I hope the competition uses VHDL." VHDL was made an IEEE Standard in 1987, while Verilog is still in the IEEE standardization process. Verilog is very C-like and liked by electrical and computer engineers as most learn the C language in college. VHDL is very Ada-like and most engineers have no experience with Ada.

Verilog was introduced in 1985 by Gateway Design System Corporation, now a part of Cadence Design Systems, Inc.'s Systems Division. Until May, 1990, with the formation of Open Verilog International (OVI), Verilog HDL was a proprietary language of Cadence. Cadence was motivated to open the language to the Public Domain with the expectation that the market for Verilog HDL-related software products would grow more rapidly with broader acceptance of the language. Cadence realized that Verilog HDL users wanted other software and service companies to embrace the language and develop Verilog-supported design tools.

Verilog HDL allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (i. e. , gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication. A primary use of HDLs is the simulation of designs before the designer must commit to fabrication. This handout does not cover all of Verilog HDL but focuses on the use of

Verilog HDL at the architectural or behavioral levels. The handout emphasizes design at the Register Transfer Level (RTL).

1.2 What is VeriWell?

VeriWell is a comprehensive implementation of Verilog HDL from Wellspring Solutions, Inc. VeriWell supports the Verilog language as specified by the OVI language Reference Manual. VeriWell was first introduced in December, 1992, and was written to be compatible with both the OVI standard and with Cadence's Verilog-XL.

Wellspring offers free versions of their VeriWell product available from `ftp://iii.net/pub/pub-site/wellspring`. Wellspring offers free versions for DOS, Sparc and Linux. The free versions are the same as the industrial versions except they are restricted to a maximum of 1000 lines of HDL code.

1.3 Why Use Verilog HDL?

Digital systems are highly complex. At their most detailed level, they may consists of millions of elements, i. e., transistors or logic gates. Therefore, for large digital systems, gate-level design is dead. For many decades, logic schematics served as the *lingua franca* of logic design, but not any more. Today, hardware complexity has grown to such a degree that a schematic with logic gates is almost useless as it shows only a web of connectivity and not the functionality of design. Since the 1970s, Computer engineers and electrical engineers have moved toward hardware description languages (HDLs). The most prominent modern HDLs in industry are Verilog and VHDL. Verilog is the top HDL used by over 10,000 designers at such hardware vendors as Sun Microsystems, Apple Computer and Motorola. Industrial designers like Verilog. It works.

The Verilog language provides the digital designer with a means of describing a digital system at a wide range of levels of abstraction, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels.

Verilog allows hardware designers to express their design with **behavioral constructs**, deterring the details of implementation to a later stage of design in the design. An abstract representation helps the designer explore architectural alternatives through **simulations** and to detect design bottlenecks before detailed design begins.

Though the behavioral level of Verilog is a high level description of a digital system, it is still a precise notation. Computer aided design tools, i. e., programs, exist which will "compile" programs in the Verilog notation to the level of circuits consisting of logic gates and flip flops. One could then go to the lab and wire up the logical circuits and have a functioning system. And, other tools can "compile" programs in Verilog notation to a description of the integrated circuit masks for **very large scale integration** (VLSI). Therefore, with the proper automated tools, one can create a VLSI description of a design in Verilog and send the VLSI description via electronic mail to a **silicon foundry** in California and receive the integrated chip in a few weeks by way of snail mail. Verilog also allows the designer to specific designs at the logical gate level using **gate constructs** and the transistor level using **switch constructs**.

Our goal in the course is not to create VLSI chips but to use Verilog to precisely describe the *functionality* of *any* digital system, for example, a computer. However, a VLSI chip designed by way of Verilog's behavioral constructs will be rather slow and be wasteful of chip area. The lower levels in Verilog allow engineers to optimize the logical circuits and VLSI layouts to maximize speed and minimize area of the VLSI chip.

2. The Verilog Language

There is no attempt in this handout to describe the complete Verilog language. It describes only the portions of the language needed to allow students to explore the architectural aspects of computers. In fact, this handout covers only a small fraction of the language. For the complete description of the Verilog HDL, consult the references at the end of the handout.

We begin our study of the Verilog language by looking at a simple Verilog program. Looking at the assignment statements,

we notice that the language is very C-like. Comments have a C++ flavor, i e., they are shown by "//" to the end of the line. The Verilog language describes a digital system as a set of **modules**, but here we have only a single module called "simple".

2.1 A First Verilog Program

```
//By Dan Hyde; August 9, 1995
//A first digital model in Verilog

module simple;
// Simple Register Transfer Level (RTL) example to demo Verilog.
// The register A is incremented by one.  Then first four bits of B is
// set to "not" of the last four bits of A.  C is the "and" reduction
// of the last two bits of A.

//declare registers and flip-flops
reg [0:7] A, B;
reg       C;

// The two "initial"s and "always" will run concurrently
initial begin: stop_at
   // Will stop the execution after 20 simulation units.
   #20; $stop;
end

// These statements done at simulation time 0 (since no #k)
initial begin: Init
    // Initialize the register A.  The other registers have values of "x"
    A = 0;

    // Display a header
    $display("Time   A         B    C");

    // Prints the values anytime a value of A, B or C changes
    $monitor("  %0d %b %b %b", $time, A, B, C);
end

//main_process will loop until simulation is over
always begin: main_process

    // #1 means do after one unit of simulation time
    #1 A = A + 1;
    #1 B[0:3] = ~A[4:7];  // ~ is bitwise "not" operator
    #1 C = &A[6:7];       // bitwise "and" reduction of last two bits
of A

end

endmodule
```

In **`module simple`**, we declared **A** and **B** as 8-bit registers and **C** a 1-bit register or flip-flop. Inside of the module, the one "**`always`**" and two "**`initial`**" constructs describe three **threads of control,** i. e., they run at the same time or

concurrently. Within the **`initial`** construct, statements are executed sequentially much like in C or other traditional imperative programming languages. The **`always`** construct is the same as the **`initial`** construct except that it loops forever as long as the simulation runs.

The notation **`#1`** means to execute the statement after delay of one unit of simulated time. Therefore, the thread of control caused by the first **`initial`** construct will delay for 20 time units before calling the system task **`$stop`** and stop the simulation.

The **`$display`** system task allows the designer to print a message much like **`printf`** does in the language C. Every time unit that one of the listed variables' value changes, the **`$monitor`** system task prints a message. The system function **`$time`** returns the current value of simulated time.

Below is the output of the VeriWell Simulator: (See Section 3 on how to use the VeriWell simulator.)

```
Time   A         B    C
  0 00000000 xxxxxxxx x
  1 00000001 xxxxxxxx x
  2 00000001 1110xxxx x
  3 00000001 1110xxxx 0
  4 00000010 1110xxxx 0
  5 00000010 1101xxxx 0
  7 00000011 1101xxxx 0
  8 00000011 1100xxxx 0
  9 00000011 1100xxxx 1
  10 00000100 1100xxxx 1
  11 00000100 1011xxxx 1
  12 00000100 1011xxxx 0
  13 00000101 1011xxxx 0
  14 00000101 1010xxxx 0
  16 00000110 1010xxxx 0
  17 00000110 1001xxxx 0
  19 00000111 1001xxxx 0
Stop at simulation time 20
```

You should carefully study the program and its output before going on. The structure of the program is typical of the Verilog programs you will write for this course, i. e., an **`initial`** construct to specify the length of the simulation, another **`initial`** construct to initialize registers and specify which registers to monitor and an **`always`** construct for the digital system you are modeling. Notice that all the statements in the second **`initial`** are done at time = 0, since there are no delay statements, i. e., #<**integer**>.

2.2 Lexical Conventions

The lexical conventions are close to the programming language C++. Comments are designated by **`//`** to the end of a line or by **`/*`** to **`*/`** across several lines. Keywords, e. g., **`module,`** are reserved and in all lower case letters. The language is case sensitive, meaning upper and lower case letters are different. Spaces are important in that they delimit tokens in the language.

Numbers are specified in the traditional form of a series of digits with or without a sign but also in the following form:

```
<size><base format><number>
```

where **<size>** contains *decimal* digits that specify the size of the constant in the number of *bits*. The **<size>** is optional. The **<base format>** is the single character ' followed by one of the following characters **b**, **d**, **o** and **h**, which stand for binary, decimal, octal and hex, respectively. The **<number>** part contains digits which are legal for the **<base format>**. Some examples:

```
549     // decimal number
'h 8FF  // hex number
'o765   // octal number
4'b11   // 4-bit binary number 0011
3'b10x  // 3-bit binary number with least significant bit unknown
5'd3    // 5-bit decimal number
-4'b11  // 4-bit two's complement of 0011 or 1101
```

The **<number>** part may *not* contain a sign. Any sign must go on the front.

A string is a sequence of characters enclosed in double quotes.

```
"this is a string"
```

Operators are one, two or three characters and are used in expressions. See Section 2.5 for the operators.

An identifier is specified by a letter or underscore followed by zero or more letters, digits, dollar signs and underscores. Identifiers can be up to 1024 characters.

2.3 Program Structure

The Verilog language describes a digital system as a set of modules. Each of these modules has an interface to other modules to describe how they are interconnected. Usually we place one module per file but that is not a requirement. The modules may run concurrently, but usually we have one top level module which specifies a closed system containing both test data and hardware models. The top level module invokes instances of other modules.

Modules can represent bits of hardware ranging from simple gates to complete systems, e. g., a microprocessor. Modules can either be specified behaviorally or structurally (or a combination of the two). A **behavioral specification** defines the behavior of a digital system (module) using traditional programming language constructs, e. g., **if**s, assignment statements. A **structural specification** expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules. At the bottom of the hierarchy the components must be primitives or specified behaviorally. Verilog primitives include gates, e. g., nand, as well as pass transistors (switches).

The structure of a module is the following:

```
module <module name> (<port list>);
<declares>
<module items>
endmodule
```

The **<module name>** is an identifier that uniquely names the module. The **<port list>** is a list of input, inout and output ports which are used to connect to other modules. The **<declares>** section specifies data objects as registers, memories and wires as wells as procedural constructs such as **function**s and **task**s.

The <**module items**> may be **`initial`** constructs, **`always`** constructs, continuous assignments or instances of modules.

Here is a behavior specification of a module **NAND**. The output **out** is the **not** of the **and** of the inputs **in1** and **in2**.

```
// Behavioral Model of a Nand gate
// By Dan Hyde, August 9, 1995
module NAND(in1, in2, out);

  input in1, in2;
  output out;
                // continuous assign statement
  assign out = ~(in1 & in2);

endmodule
```

The ports **in1**, i**n2** and **out** are labels on wires. The continuous assignment **assign** continuously watches for changes to variables in its right hand side and whenever that happens the right hand side is re-evaluated and the result immediately propagated to the left hand side (**out**).

The continuous assignment statement is used to model **combinational circuits** where the outputs change when one wiggles the input.

Here is a structural specification of a module **AND** obtained by connecting the output of one **NAND** to both inputs of another one.

```
module AND(in1, in2, out);
// Structural model of AND gate from two NANDS
  input in1, in2;
  output out;
  wire w1;
                // two instances of the module NAND
  NAND NAND1(in1, in2, w1);
  NAND NAND2(w1, w1, out);

endmodule
```

This module has two instances of the **NAND** module called **NAND1** and **NAND2** connected together by an internal wire **w1**.

The general form to invoke an instance of a module is :

```
<module name>  <parameter list> <instance name> (<port list>);
```

where <**parameter list**> are values of parameters passed to the instance. An example parameter passed would be the delay for a gate.

The following module is a high level module which sets some test data and sets up the monitoring of variables.

```
module test_AND;
// High level module to test the two other modules
```

```
  reg a, b;
  wire out1, out2;

  initial begin  // Test data
     a = 0;  b = 0;
     #1 a = 1;
     #1 b = 1;
     #1 a = 0;
  end

  initial begin // Set up monitoring
    $monitor("Time=%0d a=%b b=%b out1=%b out2=%b",
              $time, a, b, out1, out2);
  end
                // Instances of modules AND and NAND
  AND  gate1(a, b, out2);
  NAND gate2(a, b, out1);

endmodule
```

Notice that we need to hold the values **a** and b over time. Therefore, we had to use 1-bit registers. **reg** variables store the last value that was *procedurally assigned* to them (just like variables in traditional imperative programming languages). **wire**s have no storage capacity. They can be continuously driven, e. g., with a continuous **`assign`** statement or by the output of a module, or if input wires are left unconnected, they get the special value of **x** for unknown.

Continuous assignments use the keyword **assign** whereas procedural assignments have the form **<reg variable> = <expression>** where the **<reg variable>** must be a register or memory. Procedural assignment may only appear in **`initial`** and **`always`** constructs.

The statements in the block of the first **`initial`** construct will be executed sequentially, some of which are delayed by **#1**, i. e., one unit of simulated time. The **`always`** construct behaves the same as the **`initial`**construct except that it loops forever (until the simulation stops). The **`initial`** and **`always`** constructs are used to model **sequential logic** (i. e., **finite state automata**).

Verilog makes an important distinction between procedural assignment and the continuous assignment **`assign`** . Procedural assignment changes the state of a register, i. e., *sequential logic*, whereas the continuous statement is used to model **combinational logic**. Continuous assignments drive **`wire`** variables and are evaluated and updated whenever an input operand changes value. It is important to understand and remember the difference.

We place all three modules in a file and run the simulator to produce the following output.

```
Time=0 a=0 b=0 out1=1 out2=0
Time=1 a=1 b=0 out1=1 out2=0
Time=2 a=1 b=1 out1=0 out2=1
Time=3 a=0 b=1 out1=1 out2=0
```

Since the simulator ran out of events, I didn't need to explicit stop the simulation.

2.4 Data Types

2.4.1 Physical Data Types

Since the purpose of Verilog HDL is to model digital hardware, the primary data types are for modeling registers (**reg**) and wires (**wire**). The **reg** variables store the last value that was procedurally assigned to them whereas the **wire** variables represent physical connections between structural entities such as gates. A **wire** does not store a value. A **wire** variable is really only a label on a wire. (Note that the **wire** data type is only one of several **net** data types in Verilog HDL which include wired and (**wand**), wired or (**wor**) and tristate bus (**tri**). This handout is restricted to only the **wire** data type.)

The **reg** and **wire** data objects may have the following possible values:

```
0       logical zero or false
1       logical one or true
x       unknown logical value
z       high impedance of tristate gate
```

The **reg** variables are initialized to **x** at the start of the simulation. Any **wire** variable not connected to something has the **x** value.

You may specify the size of a register or wire in the declaration For example, the declarations

```
reg [0:7] A, B;
wire [0:3] Dataout;
reg [7:0] C;
```

specify registers A and **B** to be 8-bit wide with the most significant bit the zeroth bit, whereas the most significant bit of register **C** is bit seven. The wire **Dataout** is 4 bits wide.

The bits in a register or wire can be referenced by the notation **[<start-bit>:<end-bit>]**.

For example, in the second procedural assignment statement

```
initial begin: int1
        A = 8'b01011010;
        B = {A[0:3] | A[4:7], 4'b0000};
end
```

B is set to the first four bits of **A** bitwise or-ed with the last four bits of **A** and then concatenated with 0000. **B** now holds a value of 11110000. The **{}** brackets means the bits of the two or more arguments separated by commas are concatenated together.

The range referencing in an expression **must** have constant expression indices. However, a single bit may be referenced by a variable. For example:

```
reg [0:7] A, B;
B = 3;
A[0: B] = 3'b111; // ILLEGAL - indices MUST be constant!!
A[B] = 1'b1;      // A single bit reference is LEGAL
```

Why such a strict requirement of constant indices in register references? Since we are describing hardware, we want only expressions which are realizable.

Memories are specified as vectors of registers. For example, **Mem** is 1K words each 32-bits.

```
reg [31:0] Mem [0:1023];
```

The notation **Mem[0]** references the zeroth word of memory. The array index for memory (register vector) may be a register. Notice that one can *not* reference at the bit-level of a memory in Verilog HDL. If you want a specific range of bits in a word of memory, you must first transfer the data in the word to a temporary register.

2.4.2 Abstract Data Types

In addition to modeling hardware, there are other uses for variables in a hardware model. For example, the designer might want to use an **integer** variable to count the number of times an event occurs. For the convenience of the designer, Verilog HDL has several data types which do not have a corresponding hardware realization. These data types include **integer**, **real** and **time**. The data types **integer** and **real** behave pretty much as in other languages, e. g., C. Be warned that a **reg** variable is unsigned and that an **integer** variable is a signed 32-bit integer. This has important consequences when you subtract.

time variables hold 64-bit quantities and are used in conjunction with the **$time** system function. Arrays of **integer** and **time** variables (but *not* reals) are allowed. Multiple dimensional arrays are *not* allowed in Verilog HDL. Some examples:

```
integer Count;     // simple 32-bit integer
integer K[1:64];   // an array of 64 integers
time Start, Stop;  // Two 64-bit time variables
```

2.5 Operators

2.5.1 Binary Arithmetic Operators

Binary arithmetic operators operate on two operands. Register and net (wire) operands are treated as unsigned. However, real and integer operands may be signed. If any bit is unknown ('**x**') then result is unknown.

Operator	Name	Comments
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	Divide by zero produces an **x**.
%	Modulus	

2.5.2 Unary Arithmetic Operators

Operator	Name	Comments
-	Unary Minus	Changes sign of its operand.

2.5.3 Relational Operators

Relational operators compare two operands and return a logical value, i. e., TRUE(1) or FALSE(0). If any bit is unknown, the relation is ambiguous and the result is unknown.

Operator	Name	Comments
>	Greater than	
>=	Greater than or equal	
<	Less than	
<=	Less than or equal	
==	Logical equality	
!=	Logical inequality	

2.5.4 Logical Operators

Logical operators operate on logical operands and return a logical value, i. e., TRUE(1) or FALSE(0). Used typically in **if** and **while** statements. Do not confuse logical operators with the bitwise Boolean operators. For example , ! is a logical NOT and ~ is a bitwise NOT. The first negates, e. g., !(5 == 6) is TRUE. The second complements the bits, e. g., ~{1,0,1,1} is 0100.

Operator	Name	Comments
!	Logical negation	
&&	Logical AND	
||	Logical OR	

2.5.5 Bitwise Operators

Bitwise operators operate on the bits of the operand or operands. For example, the result of A & B is the AND of each corresponding bit of A with B. Operating on an unknown (**x**) bit results in the expected value. For example, the AND of an **x** with a FALSE is an **x**. The OR of an **x** with a TRUE is a TRUE.

Operator	Name	Comments
~	Bitwise negation	
&	Bitwise AND	
|	Bitwise OR	
^	Bitwise XOR	
~&	Bitwise NAND	
~|	Bitwise NOR	
~^ or ^~	Equivalence	Bitwise NOT XOR

2.5.6 Unary Reduction Operators

Unary reduction operators produce a single bit result from applying the operator to all of the bits of the operand. For example, **&A** will **AND** all the bits of **A**.

Operator	Name	Comments
&	AND reduction	
|	OR reduction	
^	XOR reduction	
~&	NAND reduction	
~|	NOR reduction	
~^	XNOR reduction	

2.5.7 Other Operators

The conditional operator operates much like in the language C.

Operator	Name	Comments
===	Case equality	The bitwise comparison includes comparison of **x** and **z** values. All bits must match for equality. Returns TRUE or FALSE.
!==	Case inequality	The bitwise comparison includes comparison of **x** and **z** values. Any bit difference produces inequality. Returns TRUE or FALSE.
{ , }	Concatenation	Joins bits together with 2 or more comma-separated expressions, e, g. {A[0], B[1:7]} concatenates the zeroth bit of A to bits 1 to 7 of B.
<<	Shift left	Vacated bit positions are filled with zeros, e. g., **A = A << 2;** shifts A two bits to left with zero fill.
>>	Shift right	Vacated bit positions are filled with zeros.
?:	Conditional	Assigns one of two values depending on the conditional expression. E. g., A = C>D ? B+3 : B-2 means if C greater than D, the value of A is B+3 otherwise B-2.

2.5.8 Operator Precedence

The precedence of operators is shown below. The top of the table is the highest precedence and the bottom is the lowest. Operators on the same line have the same precedence and associate left to right in an expression. Parentheses can be used to change the precedence or clarify the situation. We strongly urge you to use parentheses to improve readability.

```
unary operators: !  &  ~&  |  ~|  ^  ~^  +  -   (highest precedence)
                                 *  /  %
                                   +  -
                                  <<  >>
                                <  <=  >  >+
                              ==  !=  ===  ~==
                                &  ~&  ^  ~^
                                    |  ~|
                                     &&
                                     ||
                                     ?:
```

2.6 Control Constructs

Verilog HDL has a rich collection of control statements which can used in the procedural sections of code, i. e., within an **`initial`** or **`always`** block. Most of them will be familiar to the programmer of traditional programming languages like C. The main difference is instead of C's **{ }** brackets, Verilog HDL uses **`begin`** and **`end`**. In Verilog, the **{ }** brackets are used for concatenation of bit strings. Since most users are familiar with C, the following subsections typically show only an example of each construct.

2.6.1 Selection - `if` and `case` Statements

The **if** statement is easy to use.

```
if (A == 4)
   begin
     B = 2;
   end
else
   begin
     B = 4;
   end
```

Unlike the **case** statement in C, the first **<value>** that matches the value of the **<expression>** is selected and the associated statement is executed then control is transferred to after the **endcase**, i. e., no **break** statements are needed as in C.

```
case (<expression>)
  <value1>: <statement>
  <value2>: <statement>
  default: <statement>
endcase
```

The following example checks a 1-bit signal for its value.

```
case (sig)
  1'bz: $display("Signal is floating");
  1'bx: $display("Signal is unknown");
  default: $display("Signal is %b", sig);
endcase
```

2.6.2 Repetition - `for`, `while` and `repeat` Statements

The **for** statement is very close to C's **for** statement except that the ++ and -- operators do not exist in Verilog. Therefore, we need to use **i = i + 1**.

```
for(i = 0; i < 10; i = i + 1)
  begin
    $display("i= %0d", i);
  end
```

The **while** statement acts in the normal fashion.

```
i = 0;
while(i < 10)
   begin
     $display("i= %0d", i);
     i = i + 1;
   end
```

The **repeat** statement repeats the following block a fixed number of times, in this example, five times.

```
repeat (5)
```

```
      begin
        $display("i= %0d", i);
        i = i + 1;
      end
```

2.7 Other Statements

2.7.1 parameter Statement

The parameter statement allows the designer to give a constant a name. Typical uses are to specify width of registers and delays. For example, the following allows the designer to parameterized the declarations of a model.

```
parameter byte_size = 8;

reg [byte_size - 1:0] A, B;
```

2.7.2 Continuous Assignment

Continuous assignments drive **`wire`** variables and are evaluated and updated whenever an input operand changes value. The following **and**s the values on the wires **in1** and **in2** and drives the wire **out**. The keyword **`assign`** is used to distinguish the continuous assignment from the procedural assignment. See Section 2.3 for more discussion on continuous assignment.

```
assign out = ~(in1 & in2);
```

2.7.3 Blocking and Non-blocking Procedural Assignments

The Verilog language has two forms of the procedural assignment statement: blocking and non-blocking. The two are distinguished by the **=** and **<=** assignment operators. The blocking assignment statement (= operator) acts much like in traditional programming languages. The whole statement is done before control passes on to the next statement. The non-blocking (<= operator) evaluates all the right-hand sides *for the current time unit* and assigns the left-hand sides at the end of the time unit. For example, the following Verilog program

```
// testing blocking and non-blocking assignment
module blocking;
reg [0:7] A, B;
initial begin: init1
        A = 3;
        #1 A = A + 1;   // blocking procedural assignment
           B = A + 1;
           $display("Blocking:     A= %b B= %b", A, B );

        A = 3;
        #1 A <= A + 1;  // non-blocking procedural assignment
           B <= A + 1;

        #1 $display("Non-blocking: A= %b B= %b", A, B );
end

endmodule
```

produces the following output:

```
Blocking:     A= 00000100 B= 00000101
Non-blocking: A= 00000100 B= 00000100
```

The effect is for all the non-blocking assignments to use the old values of the variables at the beginning of the current time unit and to assign the registers new values at the end of the current time unit. This reflects how register transfers occur in some hardware systems.

2.8 Tasks and Functions

Tasks are like procedures in other programming languages, e. g., tasks may have zero or more arguments and do not return a value. Functions act like function subprograms in other languages. Except:

1. A Verilog function must execute during one simulation time unit. That is, no time controlling statements, i. e., no delay control (#), no event control (**@**) or **`wait`** statements, allowed. A task can contain time controlled statements.

2. A Verilog function can *not* invoke (call, enable) a task; whereas a task may call other tasks and functions.

The definition of a task is the following:

```
task <task name>;   // Notice: no list inside ()s
       <argument ports>
       <declarations>
       <statements>
endtask
```

An invocation of a task is of the following form:

```
<name of task> (<port list>);
```

where <**port list**> is a list of expressions which correspond to the <**argument ports**> of the definition. Port arguments in the definition may be **`input`**, **`inout`** or **`output`**. Since the <**argument ports**> in the task definition look like declarations, the programmer must be careful in adding declares at the beginning of a task.

```
// Testing tasks and functions
// Dan Hyde, Aug 28, 1995
module tasks;

task add;      // task definition
 input a, b;   // two input argument ports
 output c;     // one output argument port
 reg R;        // register declaration
 begin
   R = 1;
   if (a == b)
     c = 1 & R;
   else
     c = 0;
 end
```

```
endtask

initial begin: init1
   reg p;
   add(1, 0, p);  // invocation of task with 3 arguments
   $display("p= %b", p);
end

endmodule
```

input and **inout** parameters are passed by value to the task and **output** and **inout** parameters are passed back to invocation by value on return. Call by reference is not available.

Allocation of all variables is static. Therefore, a task may call itself but each invocation of the task uses the same storage, i. e., the local variables are *not* pushed on a stack. Since concurrent threads may invoke the same task, the programmer must be aware of the static nature of storage and avoid unwanted overwriting of shared storage space.

The purpose of a function is to return a value that is to be used in an expression. A function definition must contain at least one **input** argument. The passing of arguments in functions is the same as with tasks (see above). The definition of a function is the following:

```
        function <range or type> <function name>; // Notice: no list inside ()s
                <argument ports>
                <declarations>
                <statements>
        endfunction
```

where **<range or type>** is the type of the results passed back to the expression where the function was called. Inside the function, one must assign the function name a value. Below is a function which is similar to the task above.

```
// Testing functions
// Dan Hyde, Aug 28, 1995
module functions;

function [1:1] add2; // function definition
 input a, b;         // two input argument ports
 reg R;              // register declaration
 begin
   R = 1;
   if (a == b)
     add2 = 1 & R;
   else
     add2 = 0;
 end
endfunction

initial begin: init1
   reg p;
   p = add2(1, 0);  // invocation of function with 2 arguments
   $display("p= %b", p);
end
```

```
endmodule
```

2.9 Timing Control

The Verilog language provides two types of explicit timing control over when simulation time procedural statements are to occur. The first type is a **delay control** in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The second type of timing control is the **event expression**, which allows statement execution. The third subsection describes the **`wait`** statement which waits for a specific variable to change.

Verilog is a **discrete event time simulator**, i. e., events are scheduled for discrete times and placed on an ordered-by-time wait queue. The earliest events are at the front of the wait queue and the later events are behind them. The simulator removes all the events for the current simulation time and processes them. During the processing, more events may be created and placed in the proper place in the queue for later processing. When all the events of the current time have been processed, the simulator advances time and processes the next events at the front of the queue.

If there is no timing control, simulation time does not advance. Simulated time can *only* progress by one of the following:

```
1. gate or wire delay, if specified.

2. a delay control, introduced by the # symbol.

3. an event control, introduced by the @ symbol.

4. the wait statement.
```

The order of execution of events in the same clock time may not be predictable.

2.9.1 Delay Control (#)

A **delay control** expression specifies the time duration between initially encountering the statement and when the statement actually executes. For example:

```
#10 A = A + 1;
```

specifies to delay 10 time units before executing the procedural assignment statement. The # may be followed by an expression with variables.

2.9.2 Events

The execution of a procedural statement can be triggered with a value change on a wire or register, or the occurrence of a named event. Some examples:

```
@r begin                        // controlled by any value change in
        A = B&C;                // the register r
end
```

```
@(posedge clock2) A = B&C;   // controlled by positive edge of clock2

@(negedge clock3) A = B&C;   // controlled by negative edge of clock3

forever @(negedge clock)     // controlled by negative edge
begin
        A = B&C;
end
```

In the forms using **posedge** and **negedge,** they must be followed by a 1-bit expression, typically a clock. A **negedge** is detected on the transition from 1 to 0 (or unknown). A **posedge** is detected on the transition from 0 to 1 (or unknown).

Verilog also provides features to name an event and then to trigger the occurrence of that event. We must first declare the event:

```
event event6;
```

To trigger the event, we use the -> symbol :

```
-> event6;
```

To control a block of code, we use the @ symbol as shown:

```
@(event6)  begin
        <some procedural code>
end
```

We assume that the event occurs in one thread of control, i. e., concurrently, and the controlled code is in another thread. Several events may to **or**-ed inside the parentheses.

2.9.3 Wait Statement

The **wait** statement allows a procedural statement or a block to be delayed until a condition becomes true.

```
wait (A == 3)
begin
    A = B&C;
end
```

The difference between the behavior of a **wait** statement and an event is that the **wait** statement is **level sensitive** whereas **@(posedge clock);** is triggered by a **signal transition** or is **edge sensitive**.

2.10 Traffic Light Example

To demonstrate tasks as well as events, we will show a hardware model of a traffic light.

```
// Digital model of a traffic light
// By Dan Hyde August 10, 1995
```

```
module traffic;
parameter on = 1, off = 0, red_tics = 35,
          amber_tics = 3, green_tics = 20;
reg clock, red, amber, green;

// will stop the simulation after 1000 time units
initial begin: stop_at
   #1000; $stop;
end

// initialize the lights and set up monitoring of registers
initial begin: Init
    red = off; amber = off; green = off;
    $display("                 Time green amber red");
    $monitor("%3d    %b     %b    %b", $time, green, amber, red);
end

// task to wait for 'tics' positive edge clocks
// before turning light off
task light;
   output color;
   input [31:0] tics;
   begin
      repeat(tics)  // wait to detect tics positive edges on clock
         @(posedge clock);
      color = off;
   end
endtask

// waveform for clock period of 2 time units
always begin: clock_wave
  #1 clock = 0;
  #1 clock = 1;
end

always begin: main_process
   red = on;
   light(red, red_tics);  // call task to wait
   green = on;
   light(green, green_tics);
   amber = on;
   light(amber, amber_tics);
end

endmodule
```

The output of the traffic light simulator is the following:

```
                Time green amber red
                  0    0     0    1
                 70    1     0    0
```

```
                    110    0     1    0
                    116    0     0    1
                    186    1     0    0
                    226    0     1    0
                    232    0     0    1
                    302    1     0    0
                    342    0     1    0
                    348    0     0    1
                    418    1     0    0
                    458    0     1    0
                    464    0     0    1
                    534    1     0    0
                    574    0     1    0
                    580    0     0    1
                    650    1     0    0
                    690    0     1    0
                    696    0     0    1
                    766    1     0    0
                    806    0     1    0
                    812    0     0    1
                    882    1     0    0
                    922    0     1    0
                    928    0     0    1
                    998    1     0    0
Stop at simulation time 1000
```

3. Using the VeriWell Simulator

3.1 Creating the Model File

Enter the Verilog code using your favorite editor. We recommend that you use "**.v**" as the extension on the source file.

3.2 Starting the Simulator

VeriWell is run from the UNIX shell window. Type "veriwell" followed by the names of the files containing the models and the options. The options can appear in any order and anywhere on the command line. For example:

```
host-name% veriwell cpu.v bus.v top.v -s
```

This will load each of the files into memory, compile them, and enter interactive mode. Removing the "**-s**" option would cause the simulation to begin immediately. Options are processed in the order that they appear on the command line. Files are processed in the order that they appear after the options are processed.

3.3 How to Exit the Simulator?

To exit the simulator, you can type **`$finish;`** or press **Control-d**.

To stop the simulation, you press **Control-c**. Executing a **`$stop`** system task in the code will also stop the simulation.

3.4 Simulator Options

Commonly used options typed on the command line are shown below. One should consult the *VeriWell User's Guide* for the others.

`-i <inputfilename>`

Specifies a file that contains interactive commands to be executed as soon as interactive command mode is entered. This option should be used with the "**-s**" option. This can be used to initialize variables and set time limits on the simulation.

`-s`

Causes interactive mode to be entered before the simulation begins.

`-t`

Causes all statements to be traced. Trace mode may be disabled with the **`$cleartrace`** system task.

3.5 Debugging Using VeriWell's Interactive Mode

VeriWell is interactive. Once invoked, the simulation can be controlled with simple commands. Also, VeriWell accepts any Verilog statement (but new modules or declarations cannot be added).

Interactive mode is entered in one of three ways:

1). When the "**-s**" option is used on the command line (or in a command file), interactive mode is entered before the simulation begins,

2). When the simulation encounters the **`$stop`** system task, or,

3). When the user types **Control-c** during simulation (but not during compilation).

Interactive Commands

```
Continue ('.') [period]
        Resume execution from the current location.

Single-step with trace (',') [comma]
        Execute a single statement and display the trace for that statement.

Single-step without trace (';') [semicolon]
        Execute a single statement without trace.

Current location (':') [colon]
        Display the current location.

Control-d or $finish;
        Exit VeriWell simulator.
```

Typically, the kinds of Verilog statements executed interactively are used for debugging and information-gathering. **`$display`** and **`$showvars`** can be typed at the interactive prompt to show the values of variables. Notice the complete

system task statement must be typed including parameters and semicolon. **$scope(<name>);** and **$showscopes;** can be typed to traverse the model hierarchy. **$settrace;** and **$cleartrace;** will enter and exit trace mode. Typing "**#100; $stop;**" will stop the execution after 100 simulation units.

4. System Tasks and Functions

System tasks are not part of the Verilog language but are build-in tasks contained in a library. A few of the more commonly used one are described below. The Verilog Language Reference Manual has many more.

4.1 $cleartrace

The **$cleartrace** system task turns off the trace. See **$settrace** system task to set the trace.

```
$cleartrace;
```

4.2 $display

Displays text to the screen much like the **printf** statement from the language C. The general form is

```
$display(<parameter>, <parameter>, ... <parameter>);
```

where **<parameter>** may be a quoted string, an expression that returns a value or a null parameter. For example, the following displays a header.

```
$display("Registers:   A   B   C");
```

The special character % indicates that the next character is a format specification. For each % character that appears in the string, a corresponding expression must be supplied after the string. For example, the following prints the value of A in binary, octal, decimal and hex.

```
$display("A=%b binary %o octal %d decimal %h hex",A,A,A,A);
```

produces the following output

```
A=00001111 binary 017 octal  15 decimal 0f hex
```

The commonly used format specifiers are

```
%b      display in binary format
%c      display in ASCII character format
%d      display in decimal format
%h      display in hex format
%o      display in octal format
%s      display in string format
```

A 0 between the % and format specifier allocates the exact number of characters required to display the expression result, instead of the expression's largest possible value (the default). For example, this is useful for displaying the time as shown by

the difference between the following two **$display** statements.

```
$display("Time = %d", $time);
$display("Time = %0d", $time);
```

produces the following output

```
Time =                    1
Time = 1
```

Escape sequences may be included in a string. The commonly used escape sequences are the following:

```
\n      the newline character
\t      the tab character
\\      the \ character
\"      the " character
%%      the percent sign
```

A null parameter produces a single space character in the display. A null parameter is characterized by two adjacent commas in the parameter list.

Note that **$display** automatically adds a newline character to the end of its output. See **$write** in Verilog Language Reference Manual if you don't want a newline.

4.3 $finish

The **$finish** system task exits the simulator to the host operating system. Don't forget to type the semicolon while in interactive mode.

```
$finish;
```

4.4 $monitor

The **$monitor** system task provides the ability to monitor and display the values of any variable or expression specified as parameters to the task. The parameters are specified in exactly the same manner as the **$display** system task. When you invoke the **$monitor** task, the simulator sets up a mechanism whereby each time a variable or an expression in the parameter list changes value, with the exception of **$time**, the entire parameter list is displayed at the end of the time step as if reported by the **$display** task. If two or more parameters change values at the same time, however, only one display is produced. For example, the following will display a line anytime one of the registers A, B or C changes.

```
$monitor("  %0d %b %b "%b, $time, A, B, C);
```

Only one **$monitor** statement may be active at any one time. The monitoring may be turned off and on by the following:

```
$monitoroff;
<some code>
$monitoron;
```

4.5 $scope

The **$scope** system task lets the user assign a particular level of hierarchy as the interactive scope for identifying objects. **$scope** is useful during debugging as the user may change the scope to inspect the values of variables in different modules, tasks and functions.

```
$scope(<name>);
```

The <**name**> parameter must be the complete hierarchical name of a module, task, function or named block. See **$showscopes** system task to display the names.

4.6 $settrace

The **$settrace** system task enables tracing of simulation activity. The trace consists of various information, including the current simulation time, the line number, the file name, module and any results from executing the statement.

```
$settrace;
```

You can turn off the trace using the **$cleartrace** system task.

4.7 $showscopes

The **$showscopes** system task displays a complete lists of all the modules, tasks, functions and named blocks that are defined *at the current scope level.*

```
$showscopes;
```

4.8 $showvars

The **$showvars** system task produces status information for register and net (wires) variables, both scalar and vector. When invoked without parameters, **$showvars** displays the status of all variables in the current scope. When invoked with a list of variables, it shows only the status of the specified variables.

```
$showvars;
$showvars(<list of variables>);
```

4.9 $stop

The **$stop** system task puts the simulator into
a halt mode, issues an interactive command prompt and passes control to the user. See Section 3.5 on using VeriWell's interactive mode.

```
$stop;
```

4.10
$time

The **$time** system function returns the current simulation time as a 64-bit integer. **$time** must be used in an expression.

References

1. Cadence Design Systems, Inc., *Verilog-XL Reference Manual.*

2. Open Verilog International (OVI), *Verilog HDL Language Reference Manual (LRM)*, 15466 Los Gatos Boulevard, Suite 109-071, Los Gatos, CA 95032; Tel: (408)353-8899, Fax: (408) 353-8869, Email: OVI@netcom.com, $100.

3. Sternheim, E. , R. Singh, Y. Trivedi, R. Madhaven and W. Stapleton, *Digital Design and Synthesis with Verilog HDL*, published by Automata Publishing Co., Cupertino, CA, 1993, ISBN 0-9627488-2-X, $65.

4. Thomas, Donald E., and Philip R. Moorby, *The Verilog Hardware Description*

Language, second edition, published by Kluwer Academic Publishers, Norwell MA, 1994, ISBN 0-7923-9523-9, $98, includes DOS version of VeriWell simulator and programs on diskette.

5. Wellspring Solutions, Inc., *VeriWell User's Guide 1.2*, August, 1994, part of free distribution of VeriWell, available online.

6. World Wide Web Pages:

FAQ for comp.lang.verilog - http://www.cray.com/verilog/verilog-faq.html

comp.lang.verilog archives - http://www.cray.com/verilog/archive.html

Cadence Design Systems, Inc. - http://www.cadence.com/

Wellspring Solutions, Inc. - ftp://iii.net/pub/pub-site/wellspring

Verilog research at Cambridge, England - http://www.cl.cam.uk/users/mjcg/Verilog/

Page maintained by Dan Hyde, hyde@bucknell.edu Last update September 10, 1995

Back to Computer Science Home Page.