Introduction to Verilog ESC-246, ESC-266

Bob Zeidman
President
Zeidman Consulting
Bob@ZeidmanConsulting.com
www. ZeidmanConsulting.com

1. Why do you need HDLs?

I remember talking a few years ago to another engineer at a startup company where we were both working. He was older than me - he had been one of the engineers that developed the original IBM 360 computer. He was now complaining about the schematic capture tools we were learning. □I hate these tools,□ he said. □They make it so tough. It used to be we would draw our schematics on paper with pencil. Make a mistake, simply erase it or start over. We didn□t have to make sure each wire was straight or the right width. We didn□t have to line up signal names or draw boxes with all these attributes. We didn□t have to learn a whole new computer system. We just gave it to the draftsmen who cleaned them up and put everything in order. I spend too much time drawing and not enough time designing.□

This surprised me. I had worked one summer at a company where draftsmen cleaned up the schematics that I drew on paper. After that, every job I had, I used schematic capture tools. And loved them. To me, it took the power out of other people□s hands and put it into mine. It may have taken me longer in the short term, but in the long term I didn□t have to continually review schematics and look for miscommunications and mistakes. It meant that I could work my own hours without having to wait for a technician with whom I would have to review pages of schematics.

The other advantage of schematic capture was that suddenly, I could simulate my design. Something that was painful, if at all possible, previously. I could run automatic design rule checks. Modifications to existing designs were much easier. And a minimal level of documentation was standardized and almost automatic.

What the other engineer saw was the learning curve that he had to surmount rather than the eventual benefits. So now, many engineers may be inclined to react the same way to Hardware Description Languages (HDLs), the newest technology for designing circuits. But the biggest motivation for learning about HDLs is job security. In order to survive in the future, engineers will need to know HDLs. Industry analyst Dataquest (San Jose, CA) forecasts that most engineers will be switching to HDL-based tools over the next few years. About 12,000 design engineers used HDLs for some part of their designs in 1996. That number is predicted to grow to 67,000 by 1999.

The primary push for this trend is that ASICs and FPGAs are growing in complexity, which can be better handled by HDLs. As more designers incorporate these devices into their designs, they are forced to change their design methods. The major advantages of HDLs are shown below.

1. Ability to handle large, complex designs
2. Different levels of abstraction
3. Reusability
4. Concurrency
5. Timing
6. Optimization
7. Standards

8. Documentation

2. What are HDLs?

Hardware Description Languages use statements, like programming language statements, in order to define, simulate, synthesize, and layout hardware. The two main HDLs are Verilog and VHDL. There are other, limited capability, languages such as ABEL, CUPL, and PALASM that are tailored specifically for designing PALs and CPLDs. They are not robust enough to cover the complexity required for most FPGAs and ASICs. However, both Verilog and VHDL can be used to design anything from the most complex ASIC to the least complex PAL. This paper will use examples in Verilog, and will focus on that language, but the general attributes of that language can be ascribed also to VHDL.

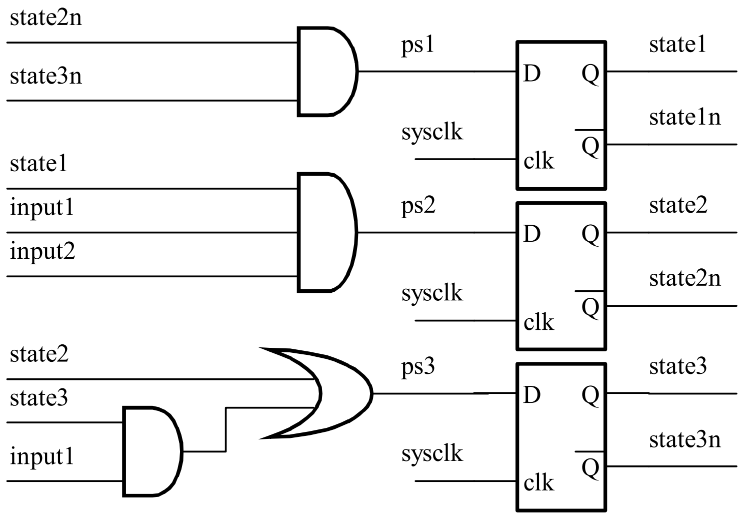


Figure 2-1 State machine schematic

```
// This module is used to implement the memory control state machine
module state_machine(sysclk, input1, input2, state3);

    /* INPUTS */
    input sysclk;          // system clock
    input input1, input2;  // inputs from the adder

    /* OUTPUTS */
    output state3;         // this can be used as the write signal

    /* DECLARATIONS */
    wire ps1;              // input to state1 flip-flop
    wire ps2;              // input to state2 flip-flop
    wire ps3;              // input to state3 flip-flop
    reg  state1, state2, state3; // state bits

    assign ps1 = ~state2 & ~state3;
    assign ps2 = state1 & input1 & input2;
    assign ps3 = state2 | (state3 & input1);

    initial begin                // initialize the state machine
       state1 = 0;
```

```
         state2 = 0;
         state3 = 0;
      end

      always @(posedge sysclk) begin      // clock in the new state on
   the
         state1 <= #3 ps1;                // rising edge of sysclk
         state2 <= #3 ps2;
         state3 <= #3 ps3;
      end
   endmodule
```

Figure 2-2 State machine code

A schematic for a simple state machine is shown in Figure 2-1. The equivalent Verilog code is shown in Figure 2-2. One important difference to note is that schematic capture is limited to just the design aspect that involves physically selecting and connecting devices. HDLs, on the other hand, are used for all stages of the design as is explained further below, where the advantages of HDLs are enumerated.

2.1 Different levels of abstraction

A hardware description language can be used to design at any level of abstraction from high-level architectural models to low-level switch models. These levels, from least amount of detail to most amount of detail are:

- Algorithmic
- Architectural
- Register Transfer Logic (RTL)
- Gate Level
- Switch Level

The top two levels use what are called Behavioral Models[1], while the lower three levels use what are called Structural Models. Behavioral models consist of code that represents the behavior of the hardware without respect to its actual implementation. Behavioral models don□t include timing numbers. Buses don□t need to be broken down into their individual signals. Adders can simply add two or more numbers without specifying registers or gates or transistors. Algorithmic models represent simply algorithms that act on data. No hardware implementation is implied in an algorithmic model. Architectural models specify the blocks that implement the algorithms. Architectural models may be divided into blocks representing PC boards, ASICs, FPGAs, or other major hardware components of the system, but they do not specify how the algorithms are implemented in each particular block.

Structural models consist of code that represents specific pieces of hardware. RTL specifies the logic on a register level. In other words, the simplest RTL code specifies register logic. Actual gates are avoided, although RTL code may use Boolean functions that can be implemented in gates. Gate level modeling consists of code that specifies gates such as NAND and NOR gates. Finally, the lowest level is that of

[1] I use my own nomenclature here for describing the various levels of abstraction. These levels do not have well-defined boundaries, and different individuals or organizations may have different, but equally valid, definitions.

switch-level models, which specifies the actual transistor switches that are combined to make gates.

The advantage to HDLs is that all of these different levels of modeling can be done with the same language. This makes all the stages of design very convenient to implement. You don□t need to learn different tools. You can easily simulate the design at a behavioral level, and then substitute various behavioral code modules with structural code modules. For system simulation, this allows you to analyze your entire project using the same set of tools. First, the algorithms can be tested and optimized. Next, the behavioral models can be used to partition the hardware into boards, ASIC, and FPGAs. The RTL code can then be written and substituted for behavioral blocks one at a time to easily test the functionality of each block. From that, the design can be synthesized into gate and switch level blocks that can be resimulated with timing numbers to get actual performance measurements. Finally this low level code can be used to generate a netlist for layout. All stages of the design have been performed using the same basic tool.

2.2 Reusability

Reusability is a big advantage of HDLs. Code written in one HDL can be used in any system that supports that HDL. Schematics, on the other hand, are only useful in a particular schematic capture software tool. Even using the same tool, portability can be difficult if a module does not physically fit into the new design. A behavioral level HDL model, and most RTL level models can be easily used over and over again on multiple designs.

2.3 Concurrency

Concurrency is an advantage that HDLs offer over normal software languages which can also be used to simulate hardware. With a normal software language, statements are executed sequentially. With an HDL, on the other hand, provisions have been added to support concurrent execution of statements. This is an absolute necessity since in a hardware design, many events occur simultaneously. For example, in a synchronous design, all flip-flops on a particular clock line must be evaluated simultaneously. While normal software languages can be used to model simultaneous events, it is up to the programmer to add the mechanism for handling this. With an HDL, the mechanism is built into the language.

2.4 Timing

```
module state_machine(sysclk, input1, input2, state3);
   . . .
   // Output delays are specified with the # symbol
    assign ps1 = #10 ~state1 & ~state2;
    assign ps2 = #5 state1 & input1 & input2;
    assign ps3 = #12 state2 | (state3 & input1);

    initial begin           // initialize the state machine
       #13;                 // wait 13 time units before initializing
       state1 = 0;
       state2 = 0;
       state3 = 0;
   end

   always @(posedge sysclk) begin
       state1 <= #3 ps1;    // output delay = 3 time units
```

```
            state2 <= #3 ps2;
            state3 <= #3 ps3;
         end
      endmodule
```

Figure 2-3 Code with timing

With schematic capture tools, when it comes time to simulate a design, the timing numbers are embedded in the netlist that is generated from the schematic. These timing numbers are based on parameters supplied by the vendors whose chips are being used. The user has some limited ability to change these numbers and some of the parameters. With HDLs, the timing numbers are explicitly stated in the code as shown in Figure 2-3. Nothing is hidden from the user, and the user has complete control over these numbers. This makes it much easier to control and optimize the timing of your design.

2.5 Optimization

HDLs are particularly useful for optimization. The most common method of designing ASICs and FPGAs involves writing an RTL level design and then using a software tool to □ synthesize□ your design. The synthesis process involves generating an equivalent gate level HDL description. Using various methods, the resulting description can be optimized to suit the particular target hardware. FPGAs, for example, typically have a very well-defined, large grain architecture. Mapping a gate level design to an FPGA would normally be difficult. However, you can write an RTL level model and synthesize it specifically for a particular FPGA. That same RTL description can be used in the future and synthesized for a particular ASIC technology.

2.6 Standards

Both major HDLs, Verilog and VHDL, are public standards. Verilog was initially a privately developed language, which was released to the public in 1990 and is now maintained by the Open Verilog International (OVI), a consortium of companies that sustain and improve the language. It has been adopted by the Institute of Electrical and Electronic Engineers (IEEE) as the standard IEEE-STD-1364. VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, and is supported by the U.S. Department of Defense. It was adopted by the IEEE as the standard IEEE-STD-1076. Because these languages are public standards, it ensures that a design written in the language can be accepted by every software tool that supports the language. It also means that you are not tied in to purchasing the tools of any one company, but can choose from tools offered by many vendors.

2.7 Documentation

HDLs, being text-based, programming-type languages, lend themselves easily to documentation. This is not to say that the code is self-documenting. However, the code is relatively easy to read, and a code module shows much about the functionality, the architecture, and the timing of the hardware. In addition, as with any programming language, statements can be organized in ways that give more information about the design, and comments can be included which explain the various sections of code. It is up to the designer and the project leader to determine how to document the code, but the nature of HDLs certainly encourages this type of documentation.

2.8 Large, Complex Designs

Large, complex designs require all of the above features. Because of this, HDLs are much better suited to large, complex designs than schematic capture or any other methods of design currently available.

3. How do I use HDLs (specifically Verilog) ?

This section describes the basics of programming in the Verilog language. Many of the concepts discussed below also apply to VHDL, although the syntax of the statements used to implement them is different.

3.1 Basic Verilog Syntax

The following subsections explain various parts of the syntax of Verilog statements.

3.1.1 Comments

Comments use the same format as C++. Single line comments begin with a // and end at the end of the line. An alternative way of writing comments is to enclose the comments between /* and */. Multiple lines comments may be constructed in this way.

3.1.2 Integer and Real Number Constants

The format for representing an integer constant is

```
<size>'<base><value>
```

where

`<size>` is the number of bits. If left out, the default value of 32 bits is used.

`<base>` is b, o, d, or h for binary, octal, decimal, or hexadecimal. If left out, the default is decimal.

`<value>` is any legal number for the specified base. A □z□ or a □?□ instead of a digit represents a high impedance signal in an integer. An □x□ instead of a digit in an integer represents an undefined signal.

Examples of integer constants are:

- `171` - size = 32, base = 10
- `'h2f3` - size = 32, base = 16
- `12'b1111_0000_1011` - size = 12, base = 2, underscores can be used as place holders and are ignored.
- `3'b11?` - size = 3, base = 2, LSB is high impedance
- `16'O101342` - size = 16, base = 8
- `8'hZ` - size = 8, base = 16, (x and z values are extended to the full size)

The format for representing a real number constant is to use normal decimal notation or scientific notation. Examples or real number constants are:

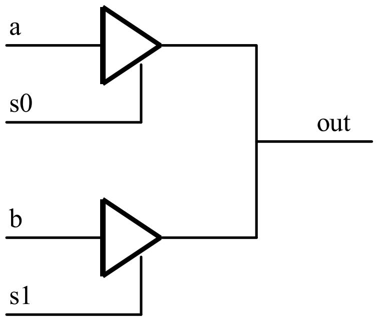
`10.5`	- decimal notation
`3.1415926`	- decimal notation
`9.3e-3`	- scientific notation
`7.7e12`	- scientific notation

3.1.3 String Constants

String constants are used to generate output from the simulator. They are enclosed in double quotes and must be on a single line. Verilog uses the same escape characters as the C programming language, which are:

`\t`	tab
`\n`	newline
`\\`	backslash
`\"`	double quote
`%%`	percent sign

3.1.4 Logic Values



a	b	s0	s1	out
x	x	0	0	z
0	x	1	0	0
x	0	0	x	L (0 or z)
x	1	0	x	H (1 or z)
0	1	1	1	x

Figure 3-1 The three unknown logic values

In Verilog there are four logic values that a signal can have. They are 0, 1, x, and z. A z represents a high impedance value on a signal. An x represents an undefined value. An undefined output from a simulation can also be an undefined of type x, L, or H. An x value means the signal is completely unknown. An L value means the signal can be a 0 or z but not a 1. An H value means the signal can be a 1 or z but not a 0. Figure 3-1shows a schematic and truth table that demonstrates these undefined outputs.

3.1.5 Identifiers

Identifiers are used to provide names for Verilog objects. Normal identifiers must begin with an alphabetical character (a-z, A-Z) or an underscore (_), and they can contain any alphanumeric character, dollar sign ($), or underscore. Escaped identifiers begin with a backslash and end with white space (space, tab, or newline), and can include any printable ASCII characters in between. Identifiers can be up to 1023 characters long.

Legal identifiers include:

• `john_smith`	
• `JohnSmith1`	
• `_john2john`	
• `\@my_house#3`	Escaped identifier (must end with white space)

Illegal identifiers include:

• `2JohnSmith`	Does not begin with an alphabetic character or underscore
• `John*2`	Includes a non-alphanumeric character

Note that Verilog is case sensitive. This means that the variable **TEMP** is not the same as **temp**. All keywords are in lower case letters. Some software allows you to turn off case sensitivity, but this must be done carefully and for good reason, since it can potentially introduce errors by connecting separate signals to each other.

3.1.6 Special Tokens

Verilog uses tokens to specify built-in functions. System tasks and functions that are part of the Verilog language are identified by a leading dollar sign ($). The pound sign (#) is used to specify delay values for simulation. Special directives for the compiler are indicated with a grave accent (`).

3.2 Register, Net, and Parameter Data Types

There are three data types in Verilog. They are registers, nets, and parameters, each of which represents different aspects of the hardware. Nets represent physical connections between devices. Registers represent storage devices. Parameters are constants.

3.2.1 Net Data Types

Nets can be thought of as wires connected to outputs of devices. Like a wire, when the output changes, the value of the net immediately changes. Figure 3-2 shows a circuit with the nets labeled. When the output of a device changes, the values of all nets that are connected to that output are immediately changed.

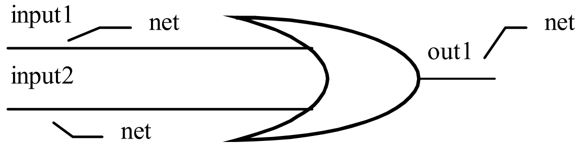


Figure 3-2 Net data types

The following net types are used to model different design-specific and technology-specific functions.

• wire, tri	standard interconnect wires
• wor, trior	for connecting multiple drivers in a Wire-OR
• wand, triand	for connecting multiple drivers in a Wire-AND
• trireg	for nets with capacitive storage
• tr1	for nets which are pulled up when not driven
• tr0	for nets which are pulled down when not driven
• supply1	for power rails
• supply0	for ground rails

Note that some wire types have two names. This allows the designer to select one or the other for better documentation purposes.

Declarations of net data types are shown below. Note that the default net width is 1. In other words, a net is assumed to be one signal unless stated otherwise.

```
wire        reset;      // This is a single signal
tr1 [4:0]   address;    // This is a bus consisting of 5 signals
                        // that are pulled up
```

3.2.2 Register Data Types

Registers hold their values until a new value is explicitly assigned to them. Registers are used extensively in behavioral models and simulations because they can model hardware without the need for defining circuitry. There are five register data types as shown below.

• reg	unsigned integer variable of any specified width
• integer	signed integer variable, 32 bits wide.
• real	signed floating-point variable, double precision
• event	Boolean event
• time	unsigned integer variable, 64 bits wide

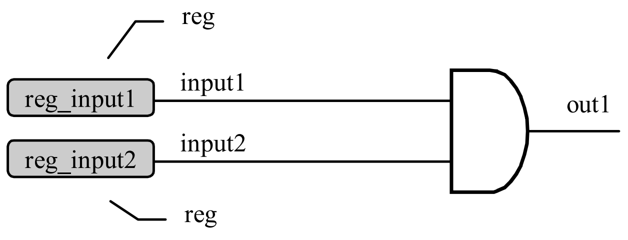


Figure 3-3 Register data type

Figure 3-3 shows registers driving a circuit for simulation. Examples of declarations of variables of each register data type are shown below.

```
integer     i, j, count;
real        x, y, val;
```

```
event        trigger, flag_set;
time         t_setup, t_hold;
reg [15:0]   data;
```

3.2.2.1 Integer and Real Data Types

The integer and real data types are exactly like the C programming language definitions. They can be manipulated just like with any other programming language. They are used for computing values that are not part of the hardware, but are more commonly used in behavioral models and simulations. Examples of the use of these data types are shown below.

```
integer      cnt, num;           // Integers
real         x, y, zCount;       // Real numbers

cnt = cnt + 1;
num = cnt/2;                     // Result will be truncated to an integer
x = 157.3;
y = x/2;
zCount = 1.2e-5;
```

3.2.2.2 Time Data Types

A time data type refers to a timing number. Time data types are used to control the timing of execution of code. In Verilog, in order to model hardware, the relative timing of signal changes is very important. Verilog keeps track of the timing of events with respect to a time scale that is set by the user. It is important to remember that the Verilog software is running sequentially, and executing statements sequentially. But it is keeping track of an internal time counter. It is this internal counter to which time data types refer. A time variable is declared using the following format:

```
time  time_variable;
```

Time data types are discussed in more detail in the section on Execution Control Statements.

3.2.2.3 Event Data Types

An event is an occurrence at one time during simulation. Code can be written such that an event is signaled due to something occurring in the hardware. Based on when this event happens, other code can execute. An event is declared using the following format:

```
event event_variable;
```

An event is triggered by the following expression:

```
->event_variable;
```

This causes the event variable to be activated. Events are discussed in more detail in the section on Execution Control Statements.

3.2.3 Parameters

Parameters are simply compile-time constants. The format of a parameter declaration is shown in Figure 3-4. Parameters can be used anywhere that a constant can be used.

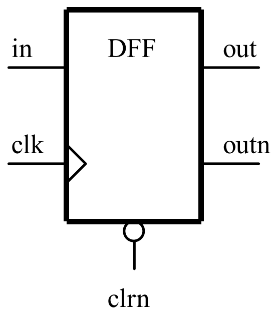
Parameters are useful for defining constants that are used throughout your code - for example, state machine states. By defining them as parameters, any change to the value of the constant needs to be done once, rather than searching through the code to find each instance of that value.

```
module temp(in1, in2, out);
    . . .
    parameter     p1 = 5;
    . . .
    wire [p1:0] signal1;   // A wire declaration using parameter
    reg      store1, store2;
    . . .
    store1 = p1 & store2;  // An equation using a parameter
    . . .
endmodule
```

Figure 3-4 Parameters

3.3 Modules

The essential building blocks for modeling hardware, is the Verilog module. A module simply represents a small, self-contained device within the design. This device may be a state machine within an ASIC, it may be an entire ASIC, or it may be a complete system. A simple module and its equivalent schematic are shown in Figure 3-5.



```
module DFF(in, clk, clrn,
           out, outn);
   input in, clk, clrn;
   output out, outn;
   . . .
   . . .
   // module body
   . . .
   . . .
endmodule
```

Figure 3-5 Module example

Note that the module has what are called module ports. These ports are equivalent to the I/O pins of the hardware device. The ports must be declared as either input, output, or inout (bi-directional) ports in the body of the module. These ports, and all internal signals, must be declared using the appropriate data type. Figure 3-6 shows a diagram that is useful for understanding how to declare signals on a module boundary. The rules to follow are:

- An input port can be driven by a net or a register, but it can only drive a net.
- An output port can be driven by a net or a register, but it can only drive a net.
- An inout port can be driven by a net, and it can only drive a net.

- If a signal is assigned a value by a procedural block, then it is a register.

The last rule means that if you want a signal to change when a certain piece of code is executed, regardless of what may be driving it, then that signal must be a register. A wire always takes the value of whatever is driving it.

As shown in Figure 3-7, modules may be embedded within modules. In this way, large devices are built upon smaller devices. There is an instance name associated with each module so that a single module can be used multiple times within a larger module. In this case, the DFF module is used four times within the REG4 module, with instance names ff0, ff1, ff2, and ff3. Remember that these modules exist simultaneously as separate devices operating concurrently with each other.

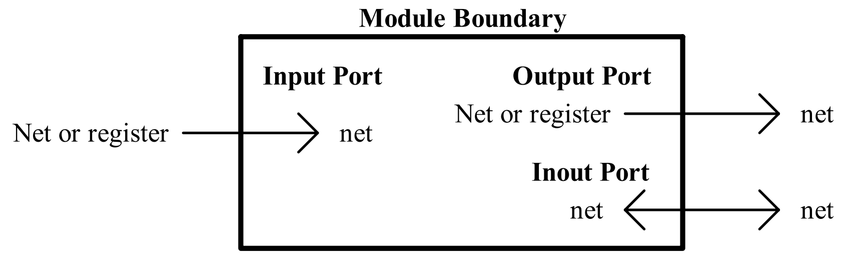
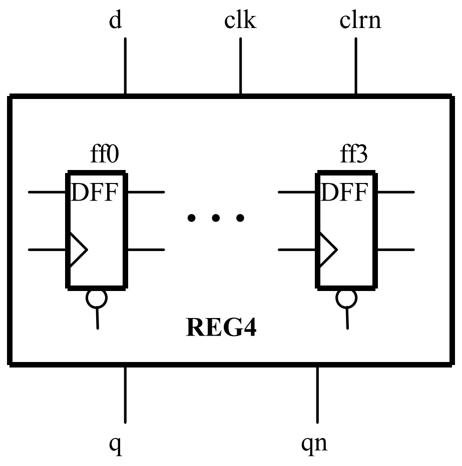


Figure 3-6 Module data types



```
module REG4(d, clk, clrn, q, qn);
   input [3:0] d;
   input clk, clrn;
   output [3:0] q, qn;
   DFF ff0(d[0], clk, clrn, q[0],
   qn[0]);
   DFF ff1(d[1], clk, clrn, q[1],
   qn[1]);
   DFF ff2(.in(d[2]), .clk(clk),
         .clrn(clrn), .out(q[2]),
         .outn(qn[2]));
   DFF ff3(.in(d[3]), .clk(clk),
         .clrn(clrn), .out(q[3]),
         .outn(qn[3]));
endmodule
```

Figure 3-7 Nested modules

There are also modules that are built into the Verilog language for convenience. These modules are known as primitives, and are shown in Table 3-1.

Primitive	Function	Expandable?
not(out, in)	inverter	expandable outputs
buf(out, in)	buffer	expandable outputs
and(out, in1, in2)	logical AND	expandable inputs
or(out, in1, in2)	logical OR	expandable inputs
xor(out, in1, in2)	logical exclusive OR	expandable inputs
nand(out, in1, in2)	logical NAND	expandable inputs
nor(out, in1, in2)	logical NOR	expandable inputs
xnor(out, in1, in2)	logical exclusive NOR	expandable inputs
bufif1(out, a, e)	tri-state buffer enabled by e = 1	not expandable
bufif0(out, a, e)	tri-state buffer enabled by e = 0	not expandable
notif1(out, a, e)	tri-state inverter enabled by e = 1	not expandable
notif0(out, a, e)	tri-state inverter enabled by e = 0	not expandable

Table 3-1 Verilog Primitives

One unique characteristic of some of the built-in modules is that the number of pins is expandable, as shown in Figure 3-8. Gates that have more than one input can have any number of inputs with a single output. The **not** and **buf** gates must have a single input but can have any number of outputs. All outputs change with zero delay with respect to the inputs, unless a delay number is explicitly stated as shown in Figure 3-9. If one delay is given, it is used for all output changes. Three delays can be given, separated by commas, to specify rise, fall, and turn-off delays respectively. Three delays separated by colons are used to specify minimum, typical, and maximum delays respectively.

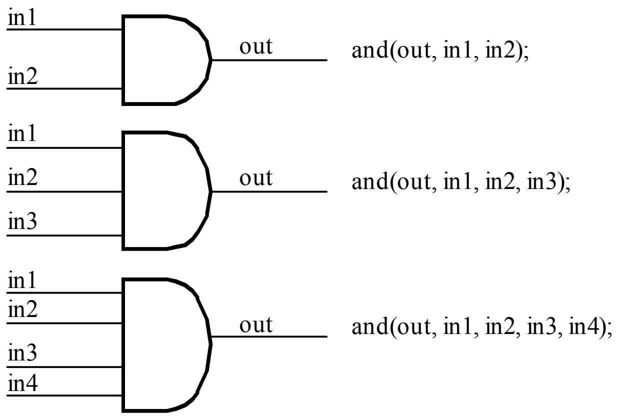


Figure 3-8 Expandable pins on primitives

```
bufif0 #(3,3,7) (out, in, ctrl);          // rise, fall, turn-off times
and #(2,3) (out, in1, in2);               // rise, fall
or #(3.2:4.0:6.3) o1(out, in1, in2);      // min:typ:max
nand #(1:2:3, 2:3:4) n1(out, in1, in2);   // rise min:typ:max,
                                          // fall min:typ:max
```

Figure 3-9 Delay specifications in primitives

When instantiating modules, there are two ways to connect signals to the ports of the module. Using the first method, the signals are matched in the same order as the ports in the declaration of the module. The second method uses the following syntax to match ports to signals:

```
.actual_port_name(net_connected_to_port)
```

This second method is the preferred method since it does not rely on knowing the order of the ports in the module declaration. Using this method, ports can easily be added or removed from the module declaration and the instantiation without accidentally shorting nets.

3.4 Operators and Expressions

The operators used in Verilog are similar to those used in the C programming language with some extra operators specifically for hardware design. They are shown in the table below.

Symbol	Operator type
+ - * /	arithmetic
> >= < <=	relational
! && ||	logical
== !=	logical equality
?:	conditional
%	modulus
=== !==	case equality
~ & | ^ ~^	bit-wise
& ~& | ~| ^ ~^	unary reduction
<< >>	shift

Table 3-2 Verilog operators

Because of the use of undefined values and high impedance values, Verilog uses two kinds of tests for equality - logical equality and case equality. Logical equality tests the equality of two well-defined signals. If one signal is not defined as a 1 or 0, then the result of the equality is undefined. Case equality is needed for simulation and debugging, where you may need to know whether a signal is undefined or high impedance. The logic tables for these operators are shown below.

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Table 3-3 Logical equality

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

Table 3-4 Case equality

The conditional operator is used to assign one of two results based on the result of an expression. The following conditional sets a to 10 if x equals 5, otherwise it sets a to 9.

```
a = (x == 5) ? 10 : 9;
```

The order of precedence for the operators is shown in the table below.

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

Table 3-5 Operator precedence

There is also a concatenate and replicate operator which acts like parentheses, except they use curly brackets to concatenate nets into a wider net. The following three statements are equivalent:

```
a = {2'b11, 4'h6};        // concatenation
```

```
                  a = {2{3'b110}};        // replication
                  a = 6'b110110;
```

A constant that appears in a concatenation must have an explicit width. The size of the concatenation is the sum of all of the widths of the concatenated values.

Unary reduction operators take a bus and operate on all bits to reduce it to a single bit. For example, the **XNOR** unary reduction operator produces the following result by **XNOR**ing all bits of a word together.

```
                  ~^4'b0110 is reduced to 1'b1
```

3.5 Continuous Assignments

Continuous assignments are used to assign values to nets. A continuous assignment consists of an expression that is continuously being evaluated, so that when one variable on the right side of the equation changes, the left side also changes. In this way, continuous assignments can be used to model combinatorial logic. Some continuous assignments are shown below.

```
      wire out1;                    // out1 is declared as a wire
      assign out1 = in1 & ~in2;     // continuous assignment - whenever
                                    // in1 or in2 changes, out1 will
                                    // also change

      wire [7:0] net1 = in1 ^ in2;  // declaration and assign statement

      assign #5 out1 = in1 | in2;   // assignment with delay - out1
                                    // changes 5 time units after in1 or
                                    // in2 change
```

There is one aspect of assignment statements that you need to be aware of. Since your Verilog simulator will run on a sequential computer, even though assigned nets should be updated immediately, the computer cannot really update all assign statements concurrently. There will be an order to them. The code in Figure 3-10 illustrates this problem.

```
      module test;
          reg  in1, in2;            // in1, in2 are declared as registers
          reg  temp;                // temporary register

          wire out1;                // out1 is declared as a wire
          assign out1 = in1 & ~in2; // continuous assignment

          initial begin             // initialize the values
            in1 = 1;
            in2 = 0;

            #1 temp = out1;         // put the value of out1 into temp
            $display("At time %0d, out1 = %1b, temp = %1b",
                     $time, out1, temp);
            #1 in1 = 0;             // change the value of in1
            temp = out1;            // put the value of out1 into temp
            $display("At time %0d, out1 = %1b, temp = %1b",
                $time, out1, temp);
        #1;
        $display("At time %0d, out1 = %1b, temp = %1b",
                $time, out1, temp);
    end
endmodule
```

Figure 3-10 Continuous assignments

The output of the code will look like:

```
At time 1, out1 = 1, temp = 1
At time 2, out1 = 1, temp = 1
At time 3, out1 = 0, temp = 1
```

The reason that out1 does not change immediately at time 2 is that the Verilog simulator, in this case, executes each statement in the module until there is a delay statement. Then it immediately goes back to reevaluate each continuous assign statement. After the delay, the value of out1 is updated, but the value of temp is not updated since temp is a register and is only updated when its assignment statement is executed.

3.6 Execution Control Statements

There are a number of ways of controlling the flow of execution of the Verilog code. Controlling the flow is essential for creating models of real hardware. Execution control allows you to design state machines, pipelines, and essentially any synchronous or conditional logic.

3.6.1 Timing Control Statements

One form of execution control is timing control. This involves delaying code from being executed by a certain amount of time, or waiting for an event to occur before executing code. The three forms of timing control statements are simple delay, event-triggered timing, and level-triggered timing.

3.6.1.1 Simple Delay

Simple delay involves waiting a certain amount of time before executing code. This is accomplished with the use of the pound (#) symbol. This symbol tells the Verilog simulator to wait for the specified number of time units before executing the following code.

```
module hello_world;

time  t1, t2;

initial begin
    t1 = 10;
    t2 = 20;
end

initial begin
    #t1 $display("%0d HELLO", $time);
    #t2 $display("%0d world!", $time);
end
```

```
   initial begin
       #t2 $display("%0d there", $time);
   end

   endmodule
```

Figure 3-11 Simple delay

The code in Figure 3-11 above produces the following results:

```
10 HELLO
20 there
30 world!
```

3.6.1.2 Event-Triggered Delay

An event is an occurrence at one time during simulation. Based on when this event happens, other code can execute. The @ symbol is used to tell Verilog to hold off executing the following statement until the specified event has occurred. Events are useful for modeling interrupts. They can also be used for high level modeling of handshaking between two pieces of hardware. The code in Figure 3-12 produces the same output as the code in Figure 3-11above.

```
module hello_world;
   event e1, e2, e3;
   initial begin
      #10 ->e1;
   end
   initial @e1 begin
      $display("%0d HELLO", $time);
      #10 ->e2;
   end
   initial @e2 begin
      $display("%0d there", $time);
      #10 ->e3;
   end
   initial @e3 $display("%0d world!", $time);
endmodule
```

Figure 3-12 Event-triggered delay

3.6.1.3 Level-Triggered Delay

Level-triggered delay is similar to event-triggered delay, except that any condition can be used to begin executing the following code. A wait statement is used along with an expression in parentheses. Once the expression becomes true, the code executes. The code executes only once, even if the expression remains true.

```
module hello_world;
reg a;
initial begin
    a = 0;
    #10 a = 1;
    #10 a = 2;
    #10 a = 3;
end
initial begin
    wait(a === 1) $display("%0d HELLO", $time);
    wait(a === 2) $display("%0d there", $time);
    wait(a === 3) $display("%0d world!", $time);
end
endmodule
```

Figure 3-13 Level-triggered delay

3.6.1.4 Intra-Assignment Timing Control

There are two different potential problems that can occur with simple delays, and Verilog has means of solving both of them. The first can be demonstrated by two signals that swap values at a given time. If the code is written with a simple delay at the beginning of the line, as in Figure 3-14, a race condition occurs where it is unclear what the final values will be. By placing the delay inside the assignment statement, Verilog evaluates the right hand side of the equation immediately and stores the value. When the specified time arrives it assigns that value. There is no race condition, as shown in Figure 3-15.

```
module race(clock);
    input clock;
    reg a, b;

    initial begin
      a = 0;
      b = 1;
    end

    // At 5 time units after the clock edge, there is a race
    // between a and b. We don't know what the final value will
    // be, or there may be an infinite loop.
    always @(posedge clock) begin
      #5 a = b;
    end
    always @(posedge clock) begin
      #5 b = a;
    end
endmodule
```

Figure 3-14 Simple delay with race condition

```
module no_race(clock);
    input clock;
    reg a, b;

    initial begin
      a = 0;
      b = 1;
    end

    // At the clock edge, the future values of a and b are
    // determined and stored. At 5 time units after the clock
    // edge, a and b are assigned the stored values. There is no
    // race condition.
    always @(posedge clock) begin
      a = #5 b;
    end
    always @(posedge clock) begin
      b = #5 a;
    end
endmodule
```

Figure 3-15 Intra-assignment timing control - no race condition

3.6.1.5 Blocking and Non-blocking Assignments

The other issue with simple delays is that it can be extremely useful to set up an output to change at a particular time and then continue to execute the remaining code. The kind of delayed assignments that we have seen so far are called blocking assignments because they block the remaining code from executing until the delay has passed. Non-blocking assignments set up the output to change at a future time, and continue executing the next statements. Non-blocking assignments simply use <= rather than = as shown in Figure 3-16.

```
module test_mod(clock);
    input clock;
    wire clock;
    reg ff1, ff2, ff3, ff4;         // models of flip-flops
    initial begin
      ff1 = 0;
      ff2 = 0;
      ff3 = 0;
      ff4 = 0;
    end

    // Blocking assignments
    always @(posedge clock) begin
      ff1 = #3 ff2;                 // ff1 changes at time 3
      ff2 = #3 ~ff1;                // ff2 changes at time 6
    end

    // Non-blocking assignments
    always @(posedge clock) begin
      ff1 <= #3 ff2;    // Both flip-flops change simultaneously
      ff2 <= #3 ~ff1;   // at time 3 with no race conditions.
    end
```

```
endmodule
```

Figure 3-16 Non-blocking assignments

3.6.2 Conditional Statements

Conditional statements have several different sections of code. Which section of code is actually executed depends on the value of the specified condition.

3.6.2.1 If and If-Else Statement

With the **if** and **if-else** statements, a single condition is given. If the condition is true, the statement after the **if** statement is executed. If it is false and there is an **else** condition, the statement after the **else** is executed. Each **else** statement is associated with the nearest **if** statement preceding it. The code in Figure 3-17 demonstrates nested **if-else** statements.

```
if (a > b) begin        // beginning of outer if
                        // begin-end allows multiple statements
                        // to be executed by the if statement
    b = b + 1;
    if (a == b)         // beginning of inner if
      flag = 1;
    else begin          // else belongs to inner if
                        // begin-end allows multiple statements
                        // to be executed by the else statement
      flag = 0;
      count = count + 1;
    end
end
```

Figure 3-17 If-Else statement

3.6.2.2 Case Statement

With the **case** statement, a single expression is given and multiple possible statements can be executed depending on the value of the expression. An example is given in Figure 3-18.

```
case (addr)             // beginning of case statement
    2’b00 : data = x;
    2’b01 : data = y*2;
    2’b0x : x = 3;
    2’b0z : y = 1;
    default :           // execute this statement if none of the
                        // other cases are true (optional)
      begin
          data = 0;
          x = 0;
          y = 0;
      end
endcase                 // this ends the case statement
```

Figure 3-18 Case statement

Note that the **case** statement does a bit-by-bit comparison. The default statement is optional. If it is used, then when none of the other cases is true, the default statement will be executed. If it is not used and all of the cases are false, then no statements will be executed. If more than one case is true, the first true case is selected and the specified statement is executed. Then execution continues after the **endcase** statement. The second case is not executed.

3.6.3 Looping Statements

Looping statements are used to execute code several times. The three types of loops are **repeat** loops, **while** loops, and **for** loops.

3.6.3.1 Repeat Loop

The **repeat** loop simply executes the following statement a fixed number of times. As in previous examples, the **begin-end** construct may be used to repeat sections of code.

```
repeat(100) $display("I will not chew gum in class!");
```

3.6.3.2 While Loop

The **while** loop allows code to be executed as long as the specified condition is true.

```
count = 10;                         // initialize count
while (count > 0) count = count - 1;// stupid way to set
                                    // count to zero
while (count < 10) begin            // execute code while count<10
      count = count + 1;            // increment count
      $display("Count = %0d", count);
      #10;                          // delay 10 time units
end
```

3.6.3.3 For Loop

The **for** loop allows code to be executed as long as the specified condition is true. It also sets up initial conditions and conditions to be executed on each loop. The following **for** loop executes exactly the same as the above **while** loop.

```
// execute code while count < 10
for (count = 0; count < 10; count = count + 1) begin
    $display("Count = %0d", count);
    #10;
end
```

3.7 Functions and Tasks

Functions and tasks are small sections of code that are used frequently throughout a design. There are a few very important differences between a task and a function. A task can contain any kind of code and any number of inputs and outputs. It can contain delays, making it useful for state machines or other types of sequential hardware. Functions, on the other hand, cannot include any delay information. Functions are executed in zero time (simulation time). Functions must have at least one input and must have exactly one output. Figure 3-19 and Figure 3-20 show examples of a function and a task, respectively.

```
module test_module;
    reg [7:0]     input;
    reg [7:0]     filter;
    reg     output;
    . . .
    output = test_function(input, filter);
    . . .

function test_function;        // this is the function declaration
    input [7:0] data;          // these are the inputs
    input [7:0] mask;

    reg temp;                  // this is an internal register

    begin                      // begin the function logic
      temp = data & mask;
      if (temp > 16)
            test_function = 1;      // this is the output
      else
            test_function = 0;      // this is the output
    end
endfunction

endmodule
```

Figure 3-19 Function in a module

```
module test_module;
    reg [7:0]     input;
    reg           out1;
    reg [7:0]     out2;
    . . .
    test_task(input, out1, out2);
    . . .

task test_task;                // this is the task declaration
    input [7:0] in_data;       // these are the inputs
    output      out_data1;     // these are the outputs
    output [7:0] out_data2;
    out_data1 = #3 ^in_data;
    out_data2 = #2 in_data & 8'h7E;
endtask

endmodule
```

Figure 3-20 Task in a module

3.8 Procedural Blocks

Finally, the various sections of code must be put into procedural blocks, which specify how to execute each section. There are two types of procedural blocks - the *initial* block and the *always* block. The initial block begins with the keyword **initial**, and it executes exactly once, immediately when the simulation begins at time 0. The always block begins with the keyword **always**, and executes each time the condition following the **always** keyword is met. The procedural block can be a single statement, but is more usually an entire set of statements that are grouped together by a **begin-end** construct or a **fork-join** construct. The difference between these two constructs is that a **begin-end** construct surrounds code that is to be executed in sequence. A **fork-join** construct surrounds code that is executed in parallel. In other words, each statement inside a **fork-join** construct is executed simultaneously. Examples of procedural blocks are given in Figure 3-21.

```
module procedures;

    reg a, b, c, d;

    // initial block with a begin-end construct
    initial begin
      a = 0;
      b = 0;
      c = 0;
      d = 0;
    end

    // always block with begin-end construct
    always @(posedge clock) begin
      a = #3 ~b;              // a changes at time 3
      b = #3 ~c;              // b changes at time 6
    end

    always @(posedge clock) fork
      c = #3 ~b;              // both c and d change
                              // simultaneously at time 3
      d = #3 ~a;              // (watch for race conditions).
    join

endmodule
```

Figure 3-21 Procedural blocks

3.9 Compiler Directives

The Verilog compiler has directives that allow the user flexibility in writing code. Some of the more commonly used directives are given in the table below.

Directive	Use
`timescale	sets the time unit and its precision
`include	include other files
`define	defines compile-time constants
`ifdef `else `endif	selectively compile code

Table 3-6 Common compiler directives

3.9.1 Timescale Directive

The syntax of the **`timescale** directive is

```
            `timescale <time_unit> / <time_precision>
```

where <time_unit> specifies the unit of measurement for delays and timing, and <time_precision> specifies how to round delay numbers before using them in simulation. Both of these are represented by the number 1, 10, or 100 followed by s for seconds, ms for milliseconds, us for microseconds, ns for nanoseconds, ps for picoseconds, or fs for femtoseconds. The <time_precision> must be smaller than or equal to the <time_unit>. It is allowed to specify multiple modules running at different time scales, but the simulation runs using the smallest time precision value in order to be accurate for the smallest precision specified in the design. Accuracy is improved but simulation speed is reduced for a greater difference between the two values.

3.9.2 Include Directive

The **`includ**e directive simply includes a text file into the current file. This is useful for including common definitions and declarations that are used in multiple code files. The syntax of this directive is

```
            `include "<file name>"
```

The file name can be absolute, including the full path, or relative to the current path. Examples of this directive are:

```
            `include "global.v"
            `include "/usr/bobz/project/defs.vh"
            `include "../../library/adder.v"
```

3.9.3 Define Directive

The syntax of the **`define** directive is

```
            `define <name> <macro_text>
```

This tells the compiler to substitute <macro_text> wherever the construct `<name> is used in the code. The following example illustrates this.

```
      `define COUNT_inc count + 1
      ‘define out_delay #4
      . . .
      count = `out_delay `COUNT_inc;       // This increments the counter
                                           // with a specific delay,
                                           // using a simple text
                                           // substitution
```

Note that a command line option can also be given to override definitions that are specified with a compiler directive.

3.9.4 Ifdef-Else-Endif Directive

The **`ifdef**, **`else** and **`endif** directives are used to selectively compile code using conditions that are determined at compile time. The formats of these directives are:

```
`ifdef <definition_name>
. . .
`else                   // optional else
. . .
`endif
```

If <definition_name> has been defined, either by a **`define** directive or by a command line option, then the code following the **`ifdef** is compiled and the code following the **`else** is ignored. If <definition_name> has not been defined, then the code following the **`ifdef** is ignored while the code following the **`else** is compiled. Note that the **`else** directive is optional. An example is given below.

```
`define HELLO 1
. . .
`ifdef HELLO
. . .                   // this code is compiled
`else
. . .                   // this code is ignored
`endif
`ifdef GOODBYE
. . .                   // this code is ignored
`endif
```

3.10 System Tasks and Functions

There are a number of system tasks and functions that are included in Verilog that provide functionality that is needed or helpful for running and debugging code. Several of the more commonly used ones are given in the table below.

Task or Function	Use
$time	a variable that holds the current simulation time
$display	creates formatted output to the screen
$monitor	output variables only when they change
$stop	stop simulating and enter debug mode
$finish	finish the simulation

Table 3-7 Common system functions and tasks

An example of the use of these system functions and tasks are shown in Figure 3-22.

```
// the following code provides simulation stimulus to a design
integer cnt;

initial begin
    cnt = 0;                        // initialize cnt
    // display any of these values whenever they change (except $time)
    // system task is not monitored)
    $monitor($time, a, b, c);

    // this monitor statement uses an optional format specifier
    $monitor($time, “%b %h %d %o”, a, b, c);
end
always @(posedge clk) begin
    cnt = cnt + 1;      // increment cnt on each rising clock

    // the display outputs each time it is executed -
    // display the time and the value of cnt on each rising clock
    $display(“time = %0d, cnt = %0d”, $time, cnt);

    if (cnt == 10)
      $finish;                 // end simulation after 10 clocks
    else if (cnt > 10)
      $stop;                   // this point shouldn’t be reached.
                               // if it gets reached, stop the
end                            // simulation and enter the debugger
```

Figure 3-22 System functions and tasks

4. Where do I get more information?

The following references are useful for obtaining more information about HDLs in general and Verilog in specific.

4.1 Books

1. Zeidman, Bob, *Verilog Designer's Library,* Prentice-Hall, 1999
2. Zeidman, Bob, *Introduction to Verilog,* IEEE Press, 2000
3. Sutherland, Stuart, *The Verilog Pli Handbook : A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface*, Kluwer Academic Publishers, 1999
4. Sternheim, E., Singh, R., Trivedi Y., *Digital Design with Verilog HDL*, Automata Publishing Company, 1990
5. Palnitkar, Samir, *Verilog HDL: A Guide to Digital Design and Synthesis*, Prentice-Hall, 1996
6. Navabi, Zainalabedin, *Verilog Digital System Design*, McGraw Hill Text, 1999
7. Moorby, P. R., and Thomas, D. E., *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1998
8. Bhasker, J., *Verilog HDL Synthesis, A Practical Primer*, Star Galaxy Press, 1998
9. Bhasker, J., *A Verilog HDL Primer*, Star Galaxy Press, 1998

4.2 Internet

1. www.chalknet.com - courses on engineering from The Chalkboard Network
2. cmp.lang.vhdl - VHDL usenet newsgroup
3. vhdl.org - VHDL International Users□ Forum (VIUF) web site
4. www.vhdl.org/vhdl_intl/ - VHDL International home page
5. www.doulos.co.uk/hegv/index.htm - *A Hardware Engineer□s Guide to VHDL*, Doulos Company

6. www.acc-eda.com/h_intro.htm - *An Introduction to VHDL for Synthesis and Simulation*, David Pellerin, President, Accolade Design Automation, Inc
7. src.doc.ic.ac.uk/public/usenet/news-FAQS/comp.answers/verilog-faq - information about Verilog
8. www.teleport.com/~celiac/verilog-manual.html - Hyde, D., *Bucknell Handbook on Verilog HDL*
9. www.doulos.co.uk/hdgv/index.htm - *A Hardware Designer□s Guide to Verilog*, Doulos Company
10. www.angelfire.com/in/verilogfaq/index.html - *Alternate Verilog FAQ*, Rajesh Bawankule
11. www.angelfire.com/in/rajesh52/verilog.html - *Rajesh Bawankule's Verilog and EDA page*, Rajesh Bawankule

About the Author

Bob Zeidman is the president of Zeidman Consulting (www.ZeidmanConsulting.com), a contract research and development firm. Since 1983, he has designed ASICs, FPGAs, and PC boards for RISC-based parallel processor systems, laser printers, network switches and routers, and other real time systems. His clients have included Apple Computer, Cisco Systems, Intel, Quickturn Design Systems, and Texas Instruments. Previously, Bob was the president of The Chalkboard Network (www.chalknet.com), an e-learning company for high tech professionals. Among his publications are technical papers on hardware and software design methods as well as three textbooks □ *Designing with FPGAs and CPLDs*, *Verilog Designer's Library*, and *Introduction to Verilog*. He has taught courses at engineering conferences throughout the world. Bob earned bachelor's degrees in physics and electrical engineering at Cornell University and a master's degree in electrical engineering at Stanford University.