An Introduction to Verilog

If you are new to programming FPGAs and CPLDs or looking for a new design language, Kareem has the solution for you. In this article, he introduces you to Verilog. Although the hardware description language has been used in the ASIC industry for years, it has all the tools to help you implement complex designs, such as a creating a VGA interface or writing to an Ethernet controller.

By Kareem Matariyeh

Editor's Note: This article first appeared in Circuit Cellar 221, 2008.

Programmable logic has been around for well over two decades. Today, due to larger and cheaper devices on the market, FPGAs and CPLDs are finding their way into a wide array of projects, and there is a plethora of languages to choose from. VHDL is the popular choice outside of the U.S. It is preferred if you need a strong typed language. However, the focus of this article will be on another popular language called Verilog, which is a hardware description language that is similar to the C language.

Typically, Verilog is used in the ASIC design industry. Companies such as Sun Microsystems, Advanced Micro Devices, and NVIDIA use Verilog to verify and test new processor architectures before committing to physical silicon and post-fab verification. However, Verilog can be used in other ways, including implementing complex designs such as a VGA interface. Another complex design such as an Ethernet controller can also be written in Verilog and implemented in a programmable device.

This article is mostly tailored to engineers who need to learn Verilog and do not know or know little about the language. Those who know VHDL will benefit from reading this article as well and should be able to pick up Verilog fairly quickly after reviewing the example listings and referring to the Resources at the end of the article. This article does not go over hardware, but I have included some links that will help you learn more about how the hardware interacts with this language at the end.

THE VERILOG LANGUAGE

First, it is best to know what variable types are available in Verilog. The basic types available are: binary, integer, and real. Other types are available but they are not used as often as these three. Keep everything in the binary number system as much as possible because type casting can cause post-implementation issues, but not all writers are the same. Binary and integer types have the ability to use other values such as “z” (high impedance) and “x” (don't care). Both are

Type	Length	Notes
Binary	1-Bit to n-bit	No type declaration needed just variable name (unsigned)
Wire	1-Bit to n-bit	A connection that does not store its value (typically used in combinational logic and module connections, some derivatives have tristate logic) (unsigned).
Reg	1-Bit to n-bit	Similar to wire but needs to be used in sequential designs. It retains its value until updated (unsigned)
Integer	32-bit	Thirty-two-bit integer (still a binary word can be signed)
Real	64-bit	Double precision word, when used its value is truncated to a whole number (signed)

TABLE 1
These are basic variable types frequently used in Verilog.

Operator	Symbol	HW Equivalent	Notes
Addition	A + B	Binary adder	(Binary and integer result)
Subtraction	A – B	Binary subtraction	(Binary and integer result)
Shift left	M << N	Logical binary shifter	Shift M left N times and appends zeros at LSB. N is treated as an integer.
Shift right	M >> N	Logical binary shifter	Shift M right N times and append zeros at MSB. (Does not sign extend.)
Relational	A > B A < B A >= B A <= B	Logical comparator	If an input value is "X" or "Z" the result is automatically false, this is for all relational operators
Equality	A == B A !=B	Logical comparator	If an input value is "X" or "Z" the result is "X"
Concatenate	{M, N, …}	Group words	Groups words into a larger word with M being the most significant (binary result)

TABLE 2
These are commonly used operators in Verilog.

nice to have around when you want a shared bus between designs or a bus to the outside world. Binary types can be assigned by giving an integer value. However, there are times when you want to assign or look at a specific bit. Some of the listings use this notation. In case you are curious, it looks like this: X'wY, where X is the word size, w is the number base—b for binary, h for hex—and Y is the value. Any value without this is considered an integer by default. Keeping everything in binary, however, can become a pain in the neck especially when dealing with numbers larger than 8 bits. **Table 1** shows some of the variable types that are available in Verilog. Integer is probably the most useful one to have around because it's 32 bits long and helps you keep track of numbers easily. Note that integer is a signed type but can also be set with all "z" or "x." Real is not used that much, when it is used the number is truncated to an integer. It is best to keep this in mind when using the real type, granted it is the least popular compared to binary and integer. When any design is initialized in a simulator, the initial values of a binary and integer are all "x." Real, on the other hand, is 0.0 because it cannot use "x." There are other types that are used when interconnecting within and outside of a design. They are included in the table, but won't be introduced until later.

Some, but not all, operators from C are in Verilog. Some of the operators available in Verilog are in Table 2. It isn't a complete list, but it contains most of the more commonly used operators. Like C, Verilog can understand operations and perform implicit casting (i.e., adding an integer with a 4-bit word and storing it into a binary register or even a real); typically this is frowned on mostly due to the fact that implicit casting in Verilog can open a new can of worms and cause issues when running the code in hardware. As long as casting does not give any erroneous results during an operation, there should be no show-stoppers in a design. Signed operation happens only if integers and real types are used in arithmetic (add, subtract, multiply) operations.

VERILOG MODULES

In Verilog, designs are called modules. A module defines its ports and contains the implementation code. If you think of the design as a black box, Verilog code typically looks like a black box with the top missing. Languages like Verilog and VHDL encourage black box usage because it can make code more readable, make debugging easier, and encourage code reuse. In Verilog, multiple code implementations cannot have the same module name. This is in stark contrast to VHDL, where architectures can share the same entity name. The only way to get around this in Verilog is to copy a module and rename it.

In **Listing 1**, a fairly standard shift register inserts a binary value at the end of a byte every clock cycle. If you're experienced with VHDL, you can see that there aren't any library declarations. This is mainly due to the fact that Verilog originated from an interpretive foundation. However, there are include directives that can be used to add external modules and features. Obviously, the first lines after the module statement are defining the modules' port directions and type with the reserved words input and output. There is another declaration

LISTING 1
This is an example of a shift register modeled in Verilog.

```
module shiftr (clk, inbit, outbyte); // module name and its
connectors

// direction definitions
// if there is no type to the dedition its automatically binary
input clk;
input inbit;

output [7:0] outbyte;

// bucket to hold the output
reg [7:0] outbyte;

// implementation
always @ (posedge clk)
begin // block of code that triggers when the clk has a positive
edge

   outbyte <= outbyte << 1;
   outbyte[0] <= inbit;

end

endmodule  // this is the end of the module
```

LISTING 2
This is an example of using functions to perform simple math.

```
module listing2(itema, itemb, resultc, opcode);
input [31:0] itema;
input [31:0] itemb;
input [1:0] opcode;

output [7:0] resultc;// function

reg [31:0] resultc;

// function
function integer myalu;
input integer a; // arguments
input integer b;
input [1:0] opc;
begin // function block

  case(opc) // case statement
      // (if X'bY did not exist, we would have to use 0, 1, 2, and 3)
    2'b00: myalu = a + b; // binary add
    2'b01: myalu = a - b; // binary subtract
    2'b10: myalu = a << b;// shift a left b times
    2'b11: myalu = a & b; // logical and
    default: myalu = 0;
  endcase
end

endfunction

assign resultc = myalu(itema, itemb, opcode);

endmodule
```

called inout, which is bidirectional but not in the listing. A module's input and output ports can use integer and real, but binary is recommended if it is a top-level module.

The reg statement essentially acts like a storage unit. Because it has the same name as the output port it acts like one item. Using reg this way is helpful because its storage ability allows the output to remain constant while system inputs change between clock cycles. There is another kind of statement called wire. It is used to tie more than one module together or drive combinational designs. It will appear in later listings.

The next line of code is the always statement or block. You want to have a begin and end statement for it. If you know VHDL, this is the same as the process statement and works in the same fashion. If you are completely new to programmable logic in general, it works like this: "For every action X that happens on signals indicated in the sensitivity list, follow these instructions." In some modules, there is usually a begin and an end statement. This is the equivalent of curly braces seen in C/C++. It's best to use these with decision structures (i.e., always, if, and case) as much as possible.

Finally, the last statement is a logical left shift operation. Verilog bitwise operators in some instances need the keyword assign for the operation to happen. The compiler will tell you if an assign statement is missing. From there, the code does its insertion operation

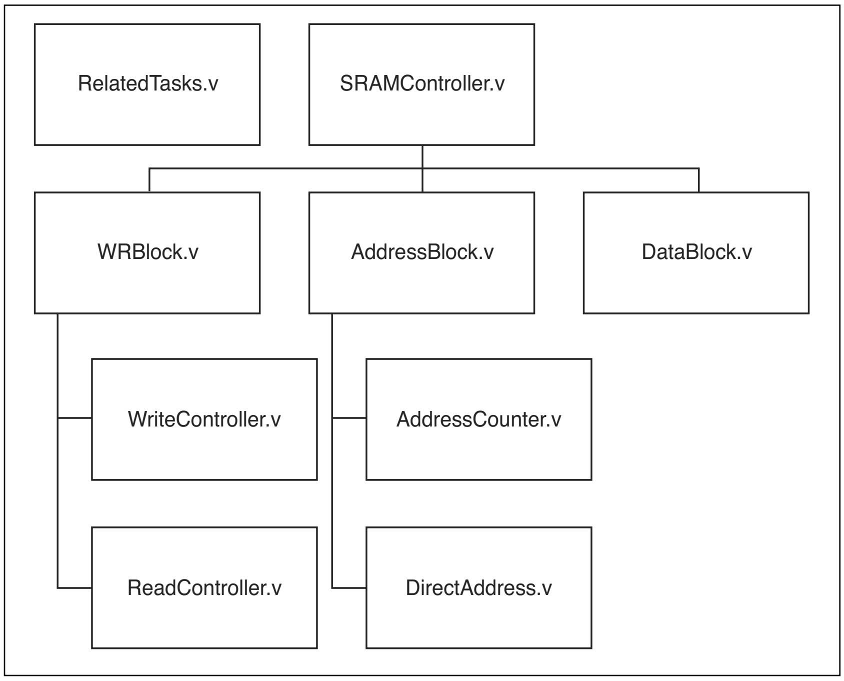
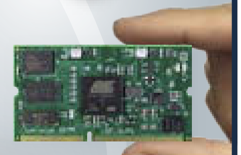


FIGURE 1
This is a top-level design view of a Verilog project.

Industrial IoT

Qty 100 Starts at $89

- Embedded Systems
- Engineering
- Prototyping
- Manufacturing
- Application Development



SoM-9G25

- Atmel ARM9
- AT91SAM9x25 400MHz
- Up to 128 MB DDR2
- 8 MB Serial Data Flash
- 256 MB NAND
- Ethernet
- A/D, SPI, I2C, I2S
- PWM, GPIO, CAN
- 6x Serial Ports, SDIO Port
- Wide Temp -40 to +85

EMAC OEM

EMAC, inc.
EQUIPMENT MONITOR AND CONTROL
Our Products Make Your Products Better™

30 YEARS OF EMBEDDED SOLUTIONS

618-529-4525 • info@emacinc.com • www.emacinc.com

The Best Test Equipment at the Lowest Prices

Scopes

Signal Generators

EMC Test Equipment

- Free Technical Support
- Excellent Customer Service

Saelig.com unique electronics
585-385-1750

```
module listing3(tank1, tank2, pump1, pump2);

input [7:0] tank1;
input [7:0] tank2;

output pump1;
output pump2;

`include “tankcheck.v”

always begin
  check(tank1, tank2, pump1, pump2);
end

endmodule

// tankcheck.v contents
task check;
input [7:0] vol1;
input [7:0] vol2;
output switch1;
output switch2;

if(vol1 == vol2) begin
  switch1 = 0;
  switch2 = 0;
end
else if(vol1 > vol2) begin
  switch1 = 0;
  switch2 = 1;
end
else  begin // vol2 > vol1
  switch2 = 0;
  switch1 = 1;
end

endtask
```

LISTING 3
This is an example of using tasks to keep tank contents equal.

and then waits for the next positive edge of the clock. This was a pretty straightforward example; unfortunately, it doesn’t do much. The best way to get around that is to add more features using functions, tying-in more modules, or using parameters to increase flexibility.

TASKS & FUNCTIONS

Tasks and functions make module implementation clearer. Both are best used when redundant code or complex actions need to be split up from the main source. There are some differences between tasks and functions.

A task can call other tasks and functions, while a function can call only other functions. A task does not return a value; it modifies a variable that is passed to it as an output. Passing items to a task is also optional. Functions, on the other hand, must return one and only one value and must have at least one value passed to them to be valid. Tasks are well-suited for test benches because they can hold delay and control statements. Functions, however, have to be able to run within one time unit to work. This means functions should not be used for test benches or simulations that require delays or use sequential designs. Experimenting is a good thing because these constructs are helpful.

There is one cardinal rule to follow when using a function or task. They have to be defined within the module, unlike VHDL where functions are defined in a package to get maximum flexibility. Tasks and functions can be defined in a separate file and then attached to a module with an include statement. This enables you to reuse code in a project or across multiple projects. Both tasks and functions can use types other than binary for their input and output ports, giving you even more flexibility.

Listing 2 contains a function that essentially acts like a basic ALU. Depending on what is passed to the function, the function will process the information and return the calculated integer value. Tasks work in the same way, but the structure is a little different when dealing with inputs and outputs. As I said before, one of the major differences between a task and a function is that the former can have multiple outputs, rather than just one. This gives you the ability to make a task more complicated internally, if need be.

Listing 3 is an example of a task in action with more than one output. Note how it is implemented the same way as a function. It has to be defined and called within the module in order to work. But rather than define the task explicitly within the module, the task is defined in a separate file and an include directive is added in the module code just to show how functions and tasks can be defined outside of a module and available for other modules to use.

BUILDING WITH MODULES

If too much is added to a module, it can become so large that debugging and editing become a chore. Doing this also minimizes code reuse to the point where new counters and state machines are being recreated when just using small modules/functions from a previous project is more than adequate. A good way to get around these issues is by making

ABOUT THE AUTHOR

Kareem Matariyeh (asm2750@gmail.com) graduated with a BS in Computer Engineering from the University of Nevada, Las Vegas, where he focused primarily on programmable logic, verification and testing, and embedded systems. His final design project involved implementing an HTDV pattern generator using a CPLD and HDMI transmitter. Kareem’s interests include RC aircraft and PC modding.

multiple modules in the same file or across multiple files and creating an instantiation of that module within an upper-level module to use its abilities. Multiple modules are good to have for a pipelined system. This enables you to use the same kind of module over multiple areas of a system. Older modules can also be used this way so less time is used on constant recreation.

That is the idea of code reuse in a nutshell. Now I will discuss an example of code reuse and multiple modules. The shift register from Listing 1 is having its data go into an even parity generator and the result from both modules is output through the top-level module in **Listing 4**. All of this is done across multiple files in one listing for easier reading. In all modular designs, there is always a module called a top-level entity, where all of the inputs and outputs of a system connect to the physical world. It is also where lower-level entities are spawned. Subordinates can spawn entities below themselves as well (see **Figure 1**). Think of it as a large black box with smaller black boxes connected with wires and those small black boxes have either stuff or even smaller black boxes. Pretty neat, but it can get annoying. Imagine a situation where a memory controller for 10-bit addressing is created and then the address length needs to be extended to 16 bits. That can be a lot of files to go through to change 10 to 16. However, with parameters all that needs to be changed is one value in one file and it's all done.

PARAMETERS

Parameters are great to have around in Verilog and can make code reuse even more attractive. Parameters allow words to take the place of a numerical value like #define in C, but with some extra features such as overriding. Parameters can be put in length descriptors, making it easy to change the size of an output, input, or variable. For example, if a VGA generator had a color depth of 8 bits but needed to be changed to 32-bit color depth, then instead of changing the locations where the value occurs, only the value of the parameter would be changed and when the module was recompiled it would be able to display 32-bit color. The same can be done for memory controllers and other modules that have ports, wires, or registers with 1 bit or more in size. Parameters can also be overridden. This is performed just before or when a module is instantiated. This is helpful if the module needs to be the same all the time across separate projects that are using the same source, but needs to be a little different for another project. Parameters can also be used in functions and tasks as

```
module listing4 (clk,
        inbit, out9word);

// input words
input clk;
input inbit;

// output words
output [8:0] out9word;

// wires
wire [7:0] shiftByte;
wire paritybit;


// instances of modules
// format: module_name   unique_inst_name   port_map
shiftr u0 (clk, inbit, shiftByte);
epg u1 (shiftByte, paritybit);

assign out9word = {paritybit, shiftByte[7:0]};

endmodule

// file epg.v
module epg(inbyte,
   outparity);

input [7:0] inbyte;
output outparity;

wire outparity;

assign outparity = (inbyte[0] ^ inbyte[1] ^
         inbyte[2] ^ inbyte[3] ^
         inbyte[4] ^ inbyte[5] ^
         inbyte[6] ^ inbyte[7]);
endmodule

// file shiftr.v
module shiftr (clk, inbit, outbyte);

// port defs
input clk;
input inbit;

output [7:0] outbyte;

// bucket to hold output
reg [7:0] outbyte;

// implementation
always @ (posedge clk)begin
  outbyte <= outbyte << 1;
  outbyte[0] <= inbit;
end

endmodule
```

LISTING 4
This is a shift register and parity generator tied in different modules.

```
module dummy(in1, out1);

// local parameters
parameter x = 1;
parameter bus1w = 10;
parameter bus2w = 8;
parameter intbus = 1 << (bus1w + bus2w);

input [bus1w-1:0] in1;
output [bus2w-1:0] out1;

// rest of dummy implementation here

endmodule

module listing5 (inbusa, inbusb, outbusa, outbusb);

// local parameters
parameter a = 16;
parameter b = 10;

input [a-1:0] inbusa;
input [b-1:0] inbusb;

output [b-1:0] outbusa;
output [a-1:0] outbusb;

wire [b-1:0] outbusa;
wire [a-1:0] outbusb;

// defparam method
defparam I0.x = 2;
defparam I0.bus1w = a;
defparam I0.bus2w = b;

dummy I0 (inbusa, outbusa);

// newer method
dummy  #(.x(3),
  .bus1w(b),
  .bus2w(a)) I1(inbusb, outbusb);

endmodule
```

LISTING 5
In this instance, parameters are overloading two modeled memories.

circuitcellar.com/ccmaterials

RESOURCES

ASIC World, A Great Source of Verilog tutorials, www.asic-world.com.

Information About Large Projects, www.opencores.com.

W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches*, Prentice Hall, Upper Saddle River, NJ, 2005.

A Programmable Logic Community Website, www.fpga4fun.com.

long as the parameter is in the same file the implementation code is in. Parameters with functions and tasks give Verilog the flexibility of a VHDL package, granted it really isn't a package, because the implementation is located in a module and not in a separate construct.

There are many ways to override parameters. One way is by using the defparam keyword, which explicitly changes the value of the parameter in the instantiated module before it is invoked. Another way is by overriding the parameter when the module is being invoked. **Listing 5** shows how both are done with dummy modules that already have defined parameters. The defparam method is from an older version of the language, so depending on the version of Verilog being used, make sure to pick the right method.

TEST BENCHES

Any system, regardless of whether it's a simple multiplexer or a large complex memory controller, should be run through a test bench for testing and simulation before moving into a hardware medium for physical verification. A test bench is really just a normal Verilog module but with unsynthesizable code such as delays, system directives, and loops. Some test bench code can be put directly into a normal module like delays. When the compiler synthesizes, it will just ignore the delays and continue onward to fitting and timing analysis. Still, it is best to try and keep all of this kind of code in the test bench module and not in the unit under test.

Designing a test bench is like making a new top-level module, this time with no ports defined, and with an instance of the main project module inside. From there, test code can be added, comprising keywords like initial, which sets the initial state of the test bench when started. System commands like $monitor, $display, and $finish, which can be used to display values of signals when they change, display signal values when called, and then stop the simulation after so much time has passed respectively are also good to have in a test bench. Another directive that is useful is timescale. Timescale defines how big a time unit is within a test bench. There is no default value for a time unit and it needs to be defined in the test bench for a simulation to occur.

Listing 6 is an example of a test bench for an 8-bit up-down counter with async reset. It runs through a simulation for 200 time units (with a time unit equal to 1 ns). The $monitor command prints any changes that happen to the input and output signals. This is a common test bench used in Verilog. It's usually best to pick up a book on test benches to learn how to write more elaborate and thorough test benches. In industry, verification and testing of designs alone account for a huge amount of work. Most people on a design team will do verification work to make sure the architecture designed operates as intended and does not lock into any invalid states. For small or hobbyist projects, a simple simulation should suffice, unless the design runs into errors that lock up the system. If that happens, run the code through a test bench to find where the issue is located in code.

MANAGING LARGE PROJECTS

Regardless of what is done in a programmable logic project, it is going to get huge. Although design abstraction at such a high level makes code more understandable to a human, it still needs to be verbose enough for the compiler to synthesize. VHDL tends to excel at large designs due to having actual packages, configuration statements, and generics. Verilog has some of these features. Initially, the language did not because it was made for modeling lower-level constructs. But, as time progressed, newer versions were released that made higher level abstraction available. When managing large projects in Verilog, keeping track of all the functions/tasks, modules, and parameters used gets hard. IDEs like Xilinx ISE and Altera Quartus II help manage project files and module levels. They should be used when writing Verilog projects because they help keep a project on track during development.

Other good ideas include keeping most functions and tasks in a separate file, and trying to keep parameters in a separate file as well. This way, when system-wide changes are needed or modules need more abilities, all that has to be added to the code are include statements. This is a bit crude when compared to VHDL and packages, but it gets the job done. Don't worry if the function file has 100 functions. The compiler will synthesize only the number of function instantiations used in the actual implementation code.

Take care of the order of compilation. Keeping track of this will ensure simulation and test results stay the same and do not change radically. It is best to ensure the type of top-level module and make sure all other modules are in their correct levels in the file hierarchy. If these guidelines are followed, there shouldn't be any issues when writing a Verilog project.

Designing with a programmable logic controller is probably the most fun and rewarding experience an engineer can have. If you have any questions, or if you want to start practicing the language, refer to the Resources section of this article.

```
module listing6tb ();

reg clktb;
reg resettb;
wire [7:0] counttb;

// create an instance of the module
listing6 mycounter(.clk (clktb),
                   .reset (resettb),
                   .count (counttb));
// setup the initial state of the system
initial
begin
$display (“Time\tClk\tReset\tCount”);

$monitor(“%d\t%b\t%b\t%b”, $time, clktb, resettb,
counttb);

clktb = 0;
resettb = 0;
counttb = 0;

#20 // delay for 20 units

// test reset condition
#20 resettb = 1;
#20 resettb = 0;

// stop testing at 200 time units
#200 $finish;
end

// clock generation
always begin
  #10 clktb = !clktb; // every 10 time units invert
clocks value
end

endmodule

// counter module
module listing6 (clk, reset, count);

// inputs
//input clk;
input reset;
input clk;

// outputs
output [7:0] count;

// buffers
reg [7:0] count;

// implementation
always @(posedge clk, posedge reset)
begin

  if(reset) begin
  count <= 8’b0; // reset counter
  end
  else begin
  count <= count + 1; // inc counter
    end
end

endmodule
```

LISTING 6
Test benches in Verilog are modules with no ports and have an instance of the top level-module within.