FPGA Simulation

A SystemVerilog Primer for VHDL Coders

Ray Salemi

Please feel free to share this primer.

You can comment on this primer at www.fpgasimulation.com/forums.

To receive updated versions, sign up for the email newsletter at www.fpgasimulation.com.

Version 1.0

June 16, 2009

The debate between Verilog and VHDL has raged for twenty years, and as an AE for Mentor Graphics I still get questions about which is the better language. The answer to this question has become clear: both languages work for RTL design.

I say this because if there were a clear winner, the market would have shaken it out by now. But because both languages still exist for RTL design, one has to conclude that either can be used to design logic successfully. The same cannot be said in the world of verification. When it comes to writing test benches and simulating complex designs, the clear winner is SystemVerilog.

SystemVerilog is an extension of Verilog. It delivers features such as randomization, functional coverage, and assertions, which allow engineers to create powerful test benches.[1] SystemVerilog allows users to create objects and implement object-oriented libraries. Object-oriented programming allows engineers to encapsulate complex behaviors in easy-to-use objects and reuse them. The Open Verification Methodology (OVM) is one such library.

SystemVerilog is an open language that is replacing the proprietary language *e*, which was the granddaddy of all verification languages. Engineers learning how to do modern verification are learning how to write in SystemVerilog.

This primer teaches VHDL users about SystemVerilog in the terms of VHDL. People learn faster when they can anchor new concepts to existing concepts. It's easier to say that "a softball is like a baseball, only bigger," than to describe the specifications of a softball from scratch. Likewise, it's much easier for a VHDL engineer to learn SystemVerilog if the latter is described in terms of VHDL.

This primer is not a complete description of SystemVerilog. (The *Language Reference Manual for SystemVerilog* is over 1400 pages long.) Instead, it describes the basic features of SystemVerilog and provides enough SystemVerilog information so that a VHDL engineer could create test benches like those in *FPGA Simulation: A Complete Step-by-Step Guide.*

You'll probably have more questions about SystemVerilog after reading this primer than you had before reading it. The website www.fpgasimulation.com hosts forums where you can ask those questions and learn more about SystemVerilog. Also, members of the *FPGA Simulation* newsletter will receive new versions of this primer as I update it based on feedback.

1. Creating those test benches is the subject of my book *FPGA Simulation: A Complete Step-by-Step Guide*. You can also learn more about test bench creation at www.fpgasimulation.com.

1 A Matter of Philosophy

The biggest controversy between Verilog and VHDL has been a matter of philosophy. Specifically, how much should the compiler protect the programmer from stupid errors? VHDL represents one side of the argument. The VHDL (and ADA and Pascal) approach says that a language should capture as much of the designer's intent in the language's syntax and semantics. The VHDL philosophy requires designers to be very clear when describing the data in the design, as well as the way that data gets moved around.

Verilog represents the other side of the argument. The Verilog (and C and Perl) approach says that a language should make common assumptions about intent and that the designer is responsible for understanding those assumptions so that the code can be kept relatively sparse, or “terse.”

Let’s look at an example of these philosophies by implementing an adder in both languages and comparing the code. The adder looks like this in the Precision RTL synthesis tool from Mentor Graphics:

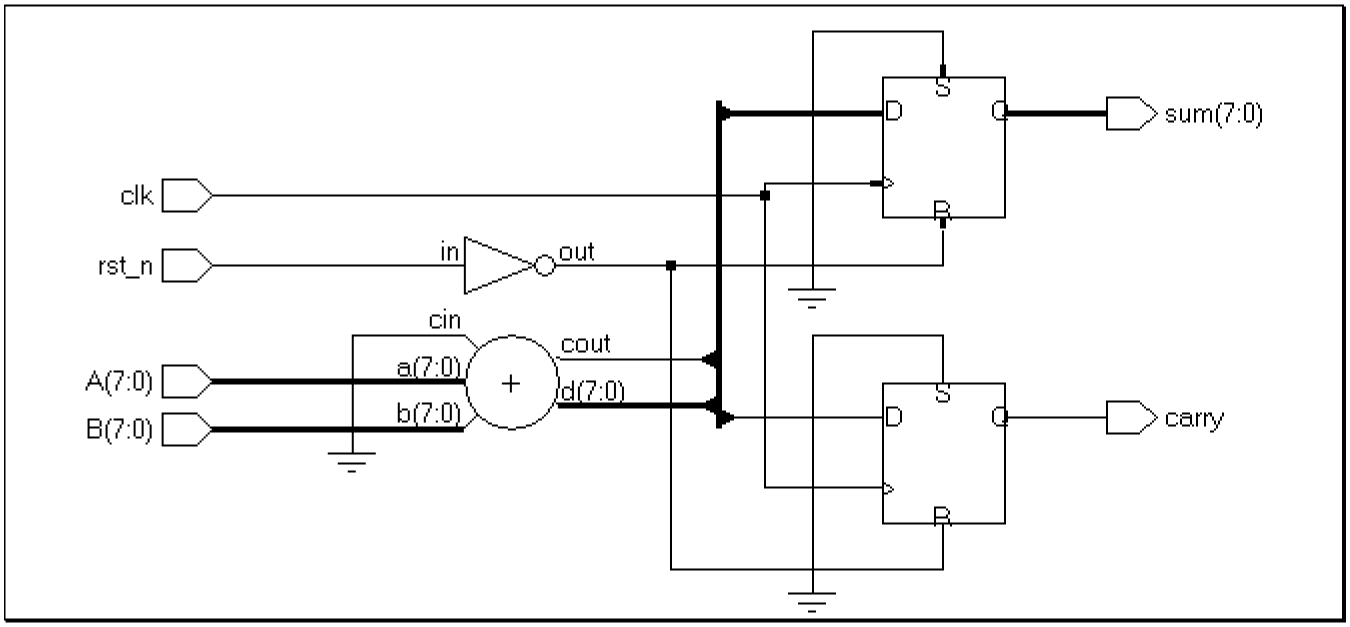


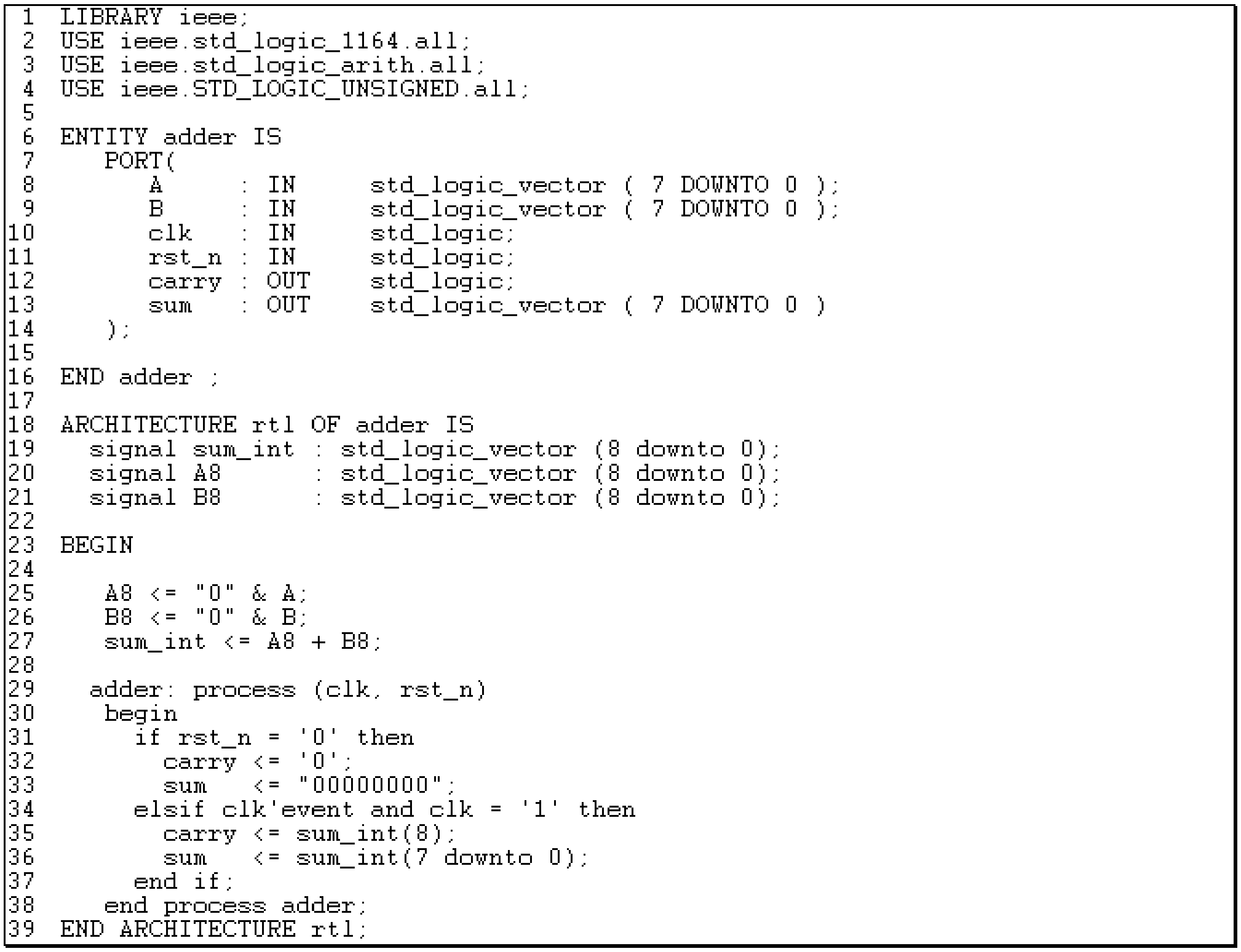
FIGURE 1. **A Simple Adder**

This adder takes two 8-bit inputs and adds them together on the positive edge of the clock. It produces an 8-bit sum and a carry bit. It is reset asynchronously when the reset is low.

Let’s see how we implement this in VHDL and Verilog.

1.1 VHDL Adder

Here is the code for the VHDL Adder:



```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  USE ieee.std_logic_arith.all;
 4  USE ieee.STD_LOGIC_UNSIGNED.all;
 5
 6  ENTITY adder IS
 7     PORT(
 8        A     : IN     std_logic_vector ( 7 DOWNTO 0 );
 9        B     : IN     std_logic_vector ( 7 DOWNTO 0 );
10        clk   : IN     std_logic;
11        rst_n : IN     std_logic;
12        carry : OUT    std_logic;
13        sum   : OUT    std_logic_vector ( 7 DOWNTO 0 )
14     );
15
16  END adder ;
17
18  ARCHITECTURE rtl OF adder IS
19    signal sum_int : std_logic_vector (8 downto 0);
20    signal A8      : std_logic_vector (8 downto 0);
21    signal B8      : std_logic_vector (8 downto 0);
22
23  BEGIN
24
25     A8 <= "0" & A;
26     B8 <= "0" & B;
27     sum_int <= A8 + B8;
28
29    adder: process (clk, rst_n)
30     begin
31       if rst_n = '0' then
32         carry <= '0';
33         sum   <= "00000000";
34       elsif clk'event and clk = '1' then
35         carry <= sum_int(8);
36         sum   <= sum_int(7 downto 0);
37       end if;
38     end process adder;
39  END ARCHITECTURE rtl;
```

FIGURE 2. **The VHDL Adder**

This VHDL code demonstrates something important about VHDL: nothing is left to be handled by the simulator. Every aspect of this adder is clearly defined in the code.

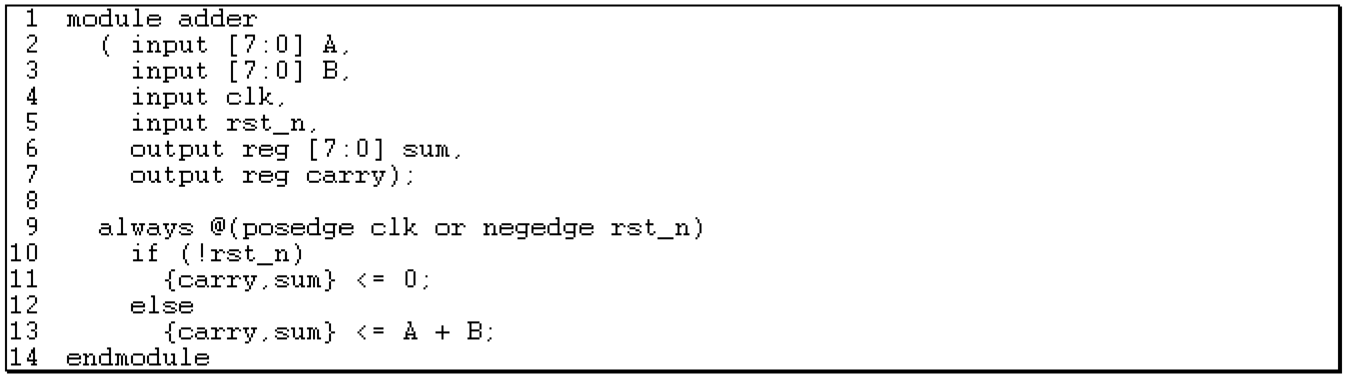
First, the IEEE libraries we are using are clearly defined. You can look up their definitions and fully understand what the `std_logic` and `std_logic_vector` data types do and how they work. You can even look up what the `+` and `&` operators mean.

We can see another sign of clarity in VHDL when we look at lines 19–27. These lines handle the fact that we are taking two 8-bit operands and creating a 9-bit result. Line 27 does the actual summation, and `sum_int` is a 9-bit number. This means we need to extend the 8-bit A and B inputs so their data can be used in a 9-bit addition.

Notice also that VHDL separates a component's interface (the entity) from its implementation (the architecture). This allows us to change architectures without changing the interface to the block. Finally, notice that we explicitly set the carry and sum outputs when the clock goes high. Let's compare this to a SystemVerilog module.

1.2 The SystemVerilog Adder

Here is the code for the SystemVerilog Adder:



```
 1  module adder
 2    ( input [7:0] A,
 3      input [7:0] B,
 4      input clk,
 5      input rst_n,
 6      output reg [7:0] sum,
 7      output reg carry);
 8
 9    always @(posedge clk or negedge rst_n)
10      if (!rst_n)
11        {carry,sum} <= 0;
12      else
13        {carry,sum} <= A + B;
14  endmodule
```

FIGURE 3. **SystemVerilog Adder**

The first thing we notice about the SystemVerilog description is its terseness. Where the VHDL adder takes 39 lines, the SystemVerilog adder takes 14. This is because SystemVerilog assumes that the user is engaged in a specific task (describing hardware) and has shared assumptions about how the language works.

The first assumption is that there is no separation between the interface to the module and its implementation. The port list on lines 2–7 apply only to the code below. If we wanted a different implementation of the adder (for example, at the gate level), we would need to compile a different version of the adder.

The biggest difference between the descriptions is that SystemVerilog makes assumptions about the size of the values in this design and they way they are extended and truncated. The addition on line 13 uses the concatenation operators "{}" to create a 9-bit output. Then it adds two 8-bit numbers and places the result in the 9-bit output.

SystemVerilog relies upon the user to understand how the + operator works and whether it will create a 9-bit output. Of course, it works fine in this case, because that is exactly how addition works. If A and B were being multiplied, then the result would get truncated to fit in 9 bits.

This is the crux of the debate over using VHDL or Verilog for RTL design. VHDL designers are horrified by Verilog's cavalier attitude about its data types and how they are used. They don't like relying upon the simulator to "do the right thing." Instead, they want to see what's happening and control it.

The Verilog designers are horrified by the amount of code VHDL takes to do "simple" tasks. They want to use the fact that they understand the simulator's behavior in order to write shorter code that (they feel) does not obscure the design's "intent."

As we can see, this is mostly an argument over taste and personal preference. Both approaches obviously work. The one that appeals to you depends upon factors such as your upbringing, your personality type, and which language you learned first.

Now that we've seen the basic philosophical differences, let's look at the details of SystemVerilog, by using VHDL as a springboard.

2 SystemVerilog is Case Sensitive

It's worth giving a page to this fact. SystemVerilog is case sensitive, so `CLK` is different from `clk`.

All keywords must be in lower case.

Variables with different cases are different variables.

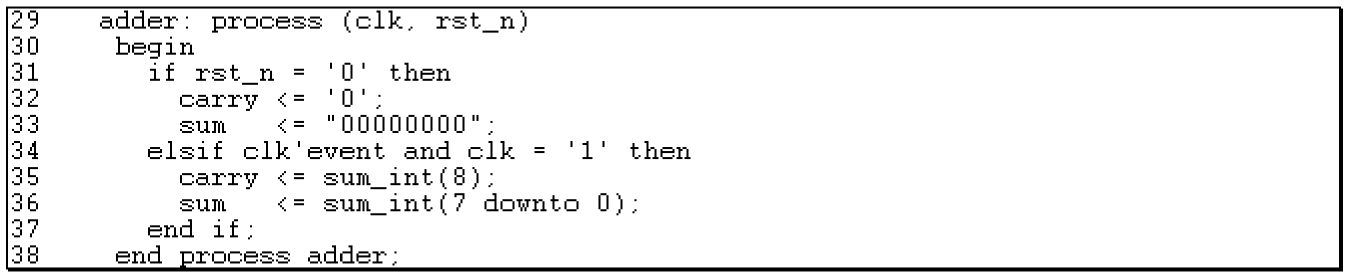
This is, of course, “opposite the case” with VHDL, which is case insensitive.

3 Processes and Sensitivity Lists

VHDL and SystemVerilog create threads to simulate simultaneous behavior. VHDL does this by using the `process` block. Verilog uses `always` blocks and `initial` blocks. In both languages, the simulator launches the blocks in a random order at time zero.

3.1 A VHDL Process Block

For example, here is the primary process in the VHDL adder:



```
adder: process (clk, rst_n)
 begin
   if rst_n = '0' then
     carry <= '0';
     sum   <= "00000000";
   elsif clk'event and clk = '1' then
     carry <= sum_int(8);
     sum   <= sum_int(7 downto 0);
   end if;
 end process adder;
```

FIGURE 4. **A VHDL Process Block**

This process demonstrates the basics of VHDL processes. It does the following:

- Loops continuously.
- Pauses at each loop and waits for a signal in the sensitivity list to change.
- Uses conditionals to decide how to respond to the events in the sensitivity list.

Because `process` blocks loop continually, we need to use a `wait` statement to force them to execute only once.

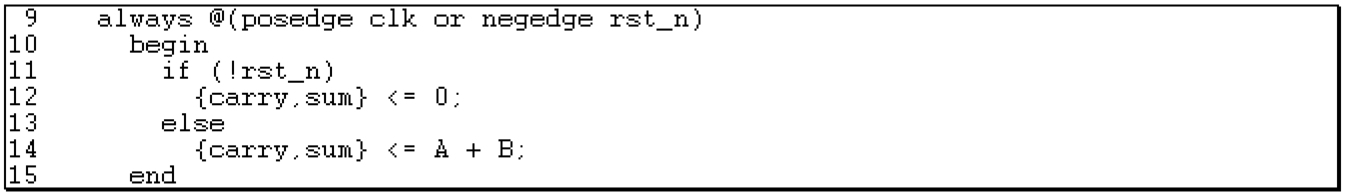
3.2 SystemVerilog `always` Blocks and `initial` Blocks

The first difference between Verilog and VHDL is that Verilog has two flavors of processes:

- **`always`** Blocks—SystemVerilog `always` blocks work just like VHDL processes. They start at time zero and loop continuously.
- **`initial`** Blocks—SystemVerilog `initial` blocks execute just once. They are exactly like a process block with a `wait` statement at the end.

Both `initial` and `always` blocks use `begin` and `end` statements to contain multiple statements. SystemVerilog allows you to put a single statement into a block without the `begin` and `end`.

Here is the `always` block from the SystemVerilog adder:



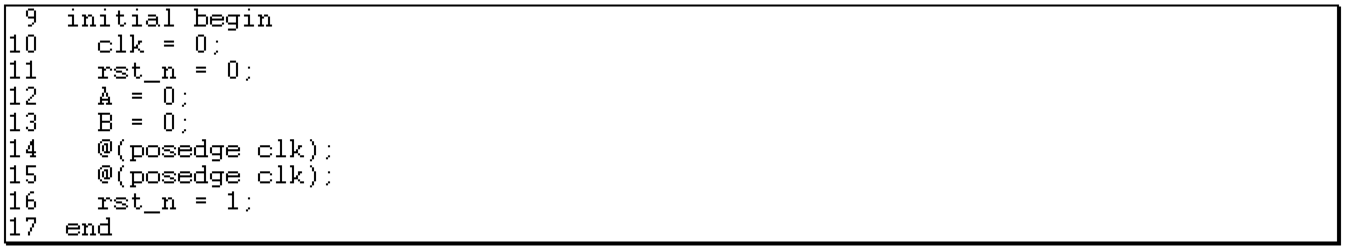
```
 9    always @(posedge clk or negedge rst_n)
10      begin
11        if (!rst_n)
12          {carry,sum} <= 0;
13        else
14          {carry,sum} <= A + B;
15      end
```

FIGURE 5. **SystemVerilog `always` Block**

This `always` block acts exactly like the VHDL one in Figure 4. It loops continuously and waits for the sensitivity list in order to start running. We'll discuss the sensitivity list in the next subsection.

The reader may notice that this example has a `begin` on line 10 and an `end` on line 15. Because the `if` statement is considered a single statement, the `begin` and `end` were not strictly necessary in this block; however, if you use them you will not run into syntax errors.

Like an `always` block, an `initial` block starts at time zero, but the latter executes only once.[1] Here is the `initial` block from the module that tests the adders:



```
 9  initial begin
10    clk = 0;
11    rst_n = 0;
12    A = 0;
13    B = 0;
14    @(posedge clk);
15    @(posedge clk);
16    rst_n = 1;
17  end
```

FIGURE 6. **SystemVerilog Initial Block**

This block starts at time zero and sets the `clk` and `rst_n` signals to `0` along with the input data. Then it waits for the positive edge of the clock to arrive twice, raises the `rst_n` signal, and exits. Once it is finished, it is removed from the simulation. This is just like a VHDL process with an infinite `wait` statement at the end of it.

Lines 14 and 15 bring us to the issue of waiting and sensitivity lists. This is the topic of our next section.

1. I've wished that they could have been called "once" blocks. Then we would have "always" blocks and "once" blocks for a nice symmetry.

4 Waiting for Time and Events

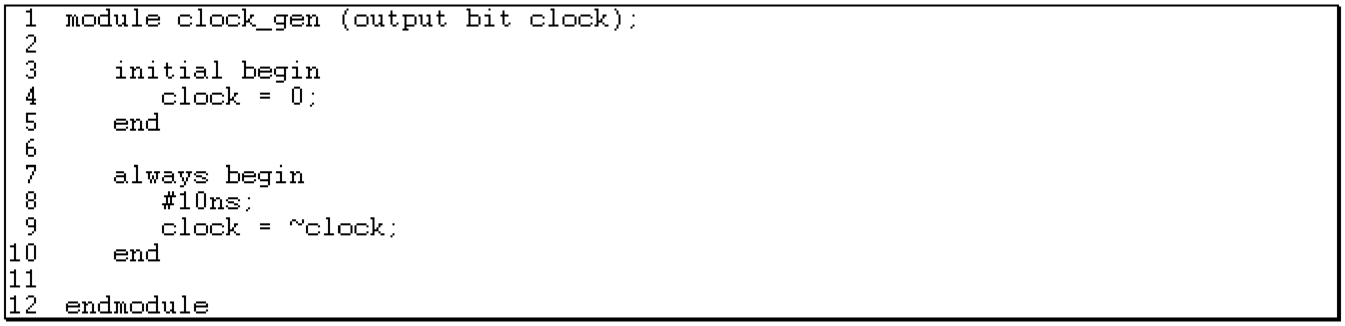
VHDL has a `wait` statement that we can use to wait for signals to achieve a value, or for a certain amount of time to pass. SystemVerilog also allows you to wait for things to happen. There are three types of delay in SystemVerilog:

- **Time Delays**—The `#` symbol tells SystemVerilog to wait for a period of time.
- **Event Delays**—The `@` symbol tells SystemVerilog to wait for an event to happen. Events are instantaneous changes in a signal's value.
- **Level Delays**—The `wait` statement tells SystemVerilog to wait for a condition to become true.

All three of these approaches can control looping through an always block, or control execution in an initial block. We'll look at each in turn.

4.1 Time Delays

SystemVerilog uses the `#` operator to specify time. Here is a simple example that creates a clock:



```
 1  module clock_gen (output bit clock);
 2
 3     initial begin
 4        clock = 0;
 5     end
 6
 7     always begin
 8        #10ns;
 9        clock = ~clock;
10     end
11
12  endmodule
```

FIGURE 7. **SystemVerilog Clock Generator**

- **Line 3**—This `initial` block sets `clock` to 0 at the beginning of the simulation.
- **Line 7**—This `always` block waits 10 ns and then flips the value of `clock`.[1] Then, because it's an `always` block, it goes back to the top and waits another 10 ns before flipping the value of `clock` again.
- **Line 8**—This is the equivalent of the VHDL wait statement with a time argument.

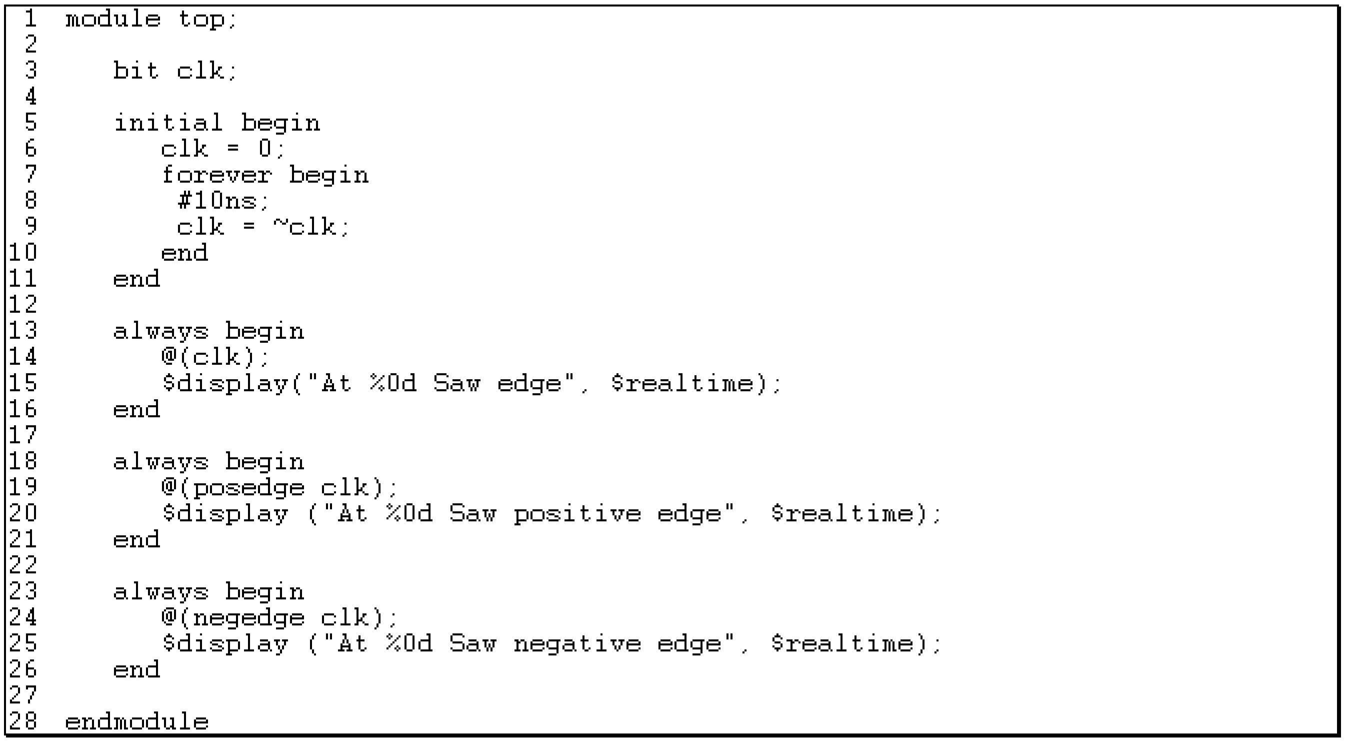
This example demonstrates basic time control in SystemVerilog. There are additional controls that manage the simulation precision. Those are discussed at www.fpgasimulation.com.

1. The ~ is the logical negation operator.

4.2 Event Delays

SystemVerilog events take zero time to execute, as they just represent changes to a signal's value. Signals can change in two ways: they can go up, generating a positive edge event, or go down, generating a negative edge event.

We use the `@(<event>)` construct to wait for events. Here is a piece of SystemVerilog code that demonstrates event delays:



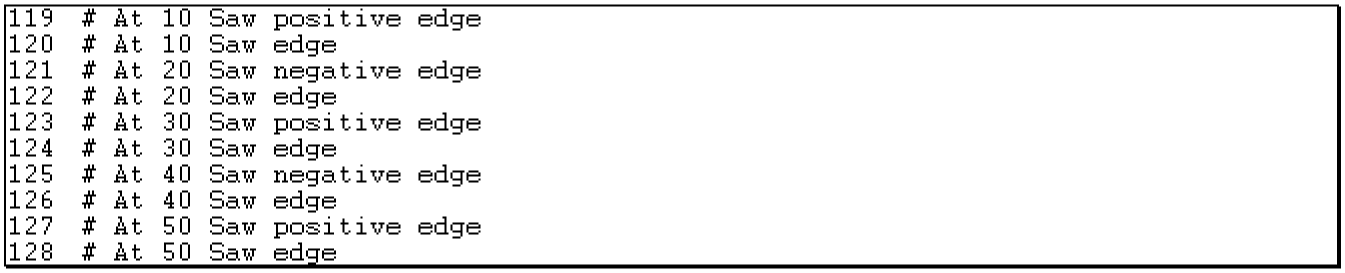
```
 1  module top;
 2
 3     bit clk;
 4
 5     initial begin
 6        clk = 0;
 7        forever begin
 8         #10ns;
 9         clk = ~clk;
10        end
11     end
12
13     always begin
14        @(clk);
15        $display("At %0d Saw edge", $realtime);
16     end
17
18     always begin
19        @(posedge clk);
20        $display ("At %0d Saw positive edge", $realtime);
21     end
22
23     always begin
24        @(negedge clk);
25        $display ("At %0d Saw negative edge", $realtime);
26     end
27
28  endmodule
```

FIGURE 8. **Event Delays in SystemVerilog**

- **Lines 5-11**—This code creates a clock that toggles every ten nanoseconds.
- **Lines 13-16**—This code will respond to every change on `clk`, such as a VHDL process. (It has no sensitivity list, a topic we'll examine in a later section.)
- **Line 14**—This event delay responds to every change to `clk`.
- **Line 19**—This event delay responds only to the positive edges of `clk`.
- **Line 24**—This event delay responds only to the negative edges of `clk`.

Event delays such as these can go anywhere in your procedural code. They are useful for creating monitors that watch busses and respond to protocol signals.

When we run the code we see the following:



```
# At 10 Saw positive edge
# At 10 Saw edge
# At 20 Saw negative edge
# At 20 Saw edge
# At 30 Saw positive edge
# At 30 Saw edge
# At 40 Saw negative edge
# At 40 Saw edge
# At 50 Saw positive edge
# At 50 Saw edge
```

FIGURE 9. **Watching Events in SystemVerilog**

We can see here that the `@(clk) always` block triggers with every clock edge. The other two `always` blocks are limited to triggering on the positive or negative edge. Notice that we can't control which `always` block will trigger first. In this example, the `posedge` and `negedge` blocks triggered first, but that might not always be the case with a different simulator or a different version of this simulator.

You can combine various events with the `or` keyword. This is the same as using a comma in a VHDL sensitivity list.

4.3 Boolean Condition Delays

The SystemVerilog `wait` statement is equivalent to the VHDL `wait until` statement. The SystemVerilog `wait` blocks execution until its argument is nonzero, and then continues. SystemVerilog does not strongly separate boolean operations from conditional operations. Consequently, you can use any statement in the test as long as it returns a zero or nonzero value. For example:

```
 1  module top;
 2
 3     integer x;
 4
 5     initial begin
 6        x = 10;
 7        forever begin
 8           #1ns;
 9           x--;
10        end
11     end
12
13     initial begin
14        wait (!x);
15        $display("At %0d: x is 0", $realtime);
16        $finish;
17     end
18  endmodule // top
```

FIGURE 10. Waiting for a 0

- **Lines 5-11**—This code sets `x` to `10` and then counts `x` down to `0`, decrementing on each nanosecond.
- **Line 14**—This line waits until the expression `!x` returns a `1`. The exclamation point (called a “bang”) reverses the logical sense of the variable. So if `x` is zero then bang-`x` (`!x`) is nonzero, and `!x` will return a true value once `x` reaches `0`.

When we run the code it looks like this:

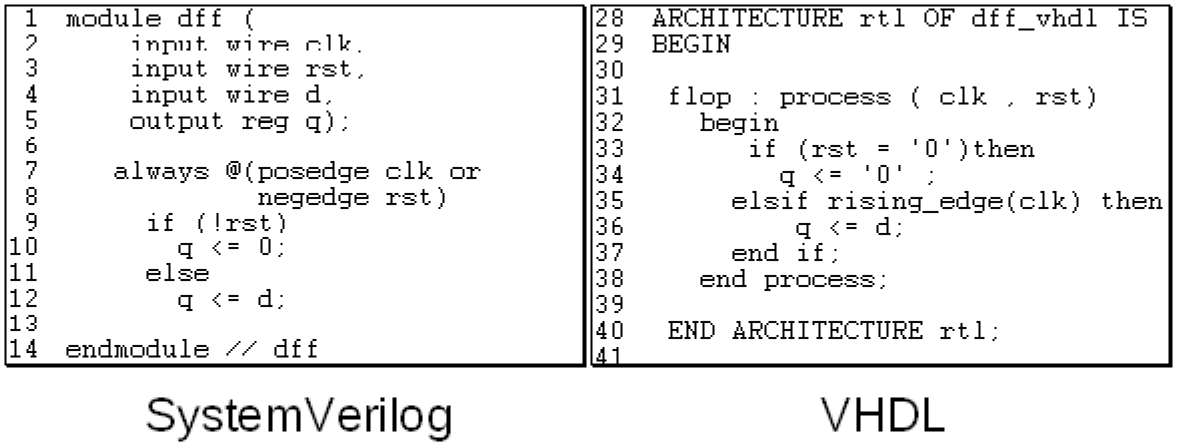
```
19  # Loading work.top
20  # At 10: x is 0
21  # ** Note: $finish    : wait_example.sv(16)
22  #    Time: 10 ns  Iteration: 0  Instance: /top
```

FIGURE 11. Waiting for `x` to reach `0`.

Now that we’ve studied the different ways to wait in SystemVerilog, we can look at sensitivity lists.

5 Sensitivity Lists in SystemVerilog

SystemVerilog and VHDL have similar ways of handling sensitivity lists, but with one big difference. VHDL sensitivity lists respond to every change on their signals, and then the programmer uses `if` statements to sort out the clock edge. SystemVerilog sensitivity lists can limit their responses to a particular edge. Here is a simple D flip-flop in both VHDL and Verilog:



```
module dff (
    input wire clk,
    input wire rst,
    input wire d,
    output reg q);

  always @(posedge clk or
          negedge rst)
    if (!rst)
      q <= 0;
    else
      q <= d;

endmodule // dff
```

```
ARCHITECTURE rtl OF dff_vhdl IS
BEGIN

 flop : process ( clk , rst)
   begin
     if (rst = '0')then
       q <= '0' ;
     elsif rising_edge(clk) then
         q <= d;
     end if;
   end process;

 END ARCHITECTURE rtl;
```

FIGURE 12. **Figure 12: Sensitivity Models in SystemVerilog and VHDL**

Both of these models create the same design, a simple D flip-flop. Each model has a process that has two signals in the sensitivity list.

The first thing to notice is that SystemVerilog `always` and `initial` blocks allow you to place a sensitivity list right after the `always` or `initial` keyword. In fact, an `always` block with no delay or sensitivity list will hang the simulator, as it causes it to spin infinitely.

The VHDL design has a process with `clk` and `rst` on the sensitivity list. This means that this process will suspend until `clk` or `rst` changes. Then the process will use `if` statements to check whether the `rst` is low or `clk` has just risen.

The SystemVerilog sensitivity list works differently. It uses the edge of the clock as part of the sensitivity. Consequently, it will start running only at the positive edge of `clk` or the negative edge of `rst`. It still needs to check whether `rst` low; however, after that it can assume that `clk` has just risen, because of the `posedge` keyword.

This additional qualification on the sensitivity list is the primary difference between VHDL and SystemVerilog. Notice also that SystemVerilog uses the `or` keyword to create a list of events on a sensitivity list, whereas VHDL uses a comma.

6 SystemVerilog Data Values

Now that we can create processes in SystemVerilog, we are ready to talk about data types. Here too, SystemVerilog and VHDL have very different philosophies.

VHDL was written to allow future users to create a wide variety of descriptions. Therefore, it allows its users to define anything they want as data types. You could have a data type in VHDL with values such as "red" and "yellow." You could even create a "+" operator for these values, so that

```
color:= red + yellow
```

is a legal statement, and `color` can take on the value of `orange`.

This means that if you want to use VHDL for hardware design, you need to create a set of data types that describe hardware. Fortunately, because it would be bad for each developer to create a different set of values, the IEEE stepped in and created `std_logic`, `std_logic_vector`, and all the other libraries.[1]

As with most things, SystemVerilog took a different approach. SystemVerilog assumes that you are designing or describing hardware, so it has built in four data values that can be used for hardware design:

- `0`—This is either logic zero or a false condition.
- `1`—This is either logic one or a true condition.
- `X`—This is an unknown logic value.
- `Z`—This is a high-impedance value.

(Notice that there are no quotes around these numbers.)

All SystemVerilog values are a combination of these values on a bit-wise basis. SystemVerilog converts every integer to a set of these values.

6.1 Multi-Bit Values

VHDL has `std_logic` for single-bit values and `std_logic_vector` for multi-bit values, where a single-bit value looks like `'0'` and a multi-bit value looks like `"00000000"`. (Both of these values are assumed to be in a binary radix.)

SystemVerilog allows you to combine the radix, the width, and the value into a single string. Constants in SystemVerilog look like this:

```
<width>'<radix><number>
```

where

1. This is why a good linter will complain if you use non-IEEE data types in your design. They could represent anything, or their representation could be machine dependent.

- `<width>`—The number of bits in the number. If you ignore this width, then the number indicates the width.
- `<radix>`—This can be `b`, `d`, `o`, or `h` for binary, decimal, octal, or hexadecimal. It could also be `B`, `D`, `O` or `H`, because this is a rare case where SystemVerilog is case insensitive. *If you don't specify the width or the radix, the simulator assumes that the number is a 32-bit decimal.*
- `<number>`—This is a string of numbers in the radix just selected.

Here are some examples of legal SystemVerilog numbers:

TABLE 1-1. SystemVerilog Values

String	Interpretation
`32'hFEDCBA90`	32-bit hexadecimal number
`'h837FFF`	24-bit hexadecimal number
`48`	Same as 'd48. A decimal integer, 32-bits wide.
`4F`	**Illegal.** Numbers with no radix designator must be decimal.
`101`	One hundred and one in decimal
`3'b101`	A 3-bit binary number representing the value 5.
`8'b00011000`	"00011000" as a VHDL `std_logic_vector`.
`1'b1`	'1' in VHDL `std_logic`
`1`	This is a 32-bit value with 31 zeros and a single bit set. You can put it into a single-bit value, as we'll see below, so it acts like '1'.
`8'o232`	An 8-bit octal number (8'b1001_1010). Notice that the underscore is legal in a number in SystemVerilog
`3'b012`	Illegal[a] binary value.
`4'b10x0`	Binary number with one bit unknown.
`16'hZZZZ`	Driving high-impedance to a 16-bit bus.

a. Remember Futurama's Frye comforting Bender the robot after Bender had a nightmare?
Bender: "Ones and zeroes everywhere...[shudder] and I thought I saw a two."
Fry: "It was just a dream, Bender. There's no such thing as two."

Notice that unlike `std_logic_vector`, or `std_logic`, these numbers are not strings of bits. There is no need to encase them in quotes, because they are just numbers in the program.

Of course, you can't just write programs with numbers. You need variables, and that brings us to the SystemVerilog data types.

7 SystemVerilog Data Types

Creating hardware, or programs that interact with hardware, requires a four-state data model. You need to be able to handle the values `1`, `0`, `X`, and `Z`. VHDL implements this with the IEEE data types such as `std_logic` and `std_logic_vector`. SystemVerilog inherently delivers these data types, so there is no need for additional libraries.

There are two basic kinds of data types. There are 2-state types that can only hold `0`s and `1`s, and there are four-state types that can hold `0`s,`1`s, `X`s, and `Z`s.

7.1 4-State Data Types

The 4-state data types are similar to the data types you'd use for a VHDL signal, usually a `std_logic` or `std_logic_vector` type. There are four of these types:

- **`logic` and `reg`**—These data types are identical. The `reg` type is a holdover from Verilog, but the developers of SystemVerilog felt that it implied the creation of a register where there really didn't need to be one, so they created a synonym called `logic`. If you don't declare a width for these data types, they are assumed to be one bit wide.
- **`integer`**—By default an integer is assumed to be 32-bits wide. It can be longer if you are simulating on a computer with a wider data word. This type is different than the `int` data type which is a 2-state type.
- **`time`**—This is a 64-bit variable. It is called "time" because you can store the simulation time in it.

Most designs you see will use the `logic` and `reg` data types to store data in procedural SystemVerilog.

7.1.1 4-State Data Types

The `logic`, `reg`, `integer`, and `time` data types can store values in procedural code within `always` blocks. The bits in these data types can take on the values `0`, `1`, `X`, and `Z`.

The `wire`, `logic`, and `reg` data types can also be used to connect modules. Variables that connect modules vary their value based on how they are driven from their module blocks. These types take on the value of `X` if they are not driven.

If two or more modules drive the same signal, then the simulator figures out the value. In the default case of logic, reg, or wire the following table applies:

TABLE 1-2. Default Conflict Resolution in SystemVerilog

	0	**1**	**X**	**Z**
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

The default conflict in SystemVerilog acts as you would expect a wire to act. SystemVerilog has other resolution behaviors that you can control with different wire types such as `wor`, and `wand`, and with different drive strengths. But these are beyond the scope of this primer.

7.1.2 2-State Data Types

The 4-state data types can handle the unknown values of X and Z, but they take more memory and run more slowly than 2-state data types. The 2-state data types can only handle 1 and 0, but they need fewer resources, so some people prefer to use them if it isn't necessary to handle unknowns.

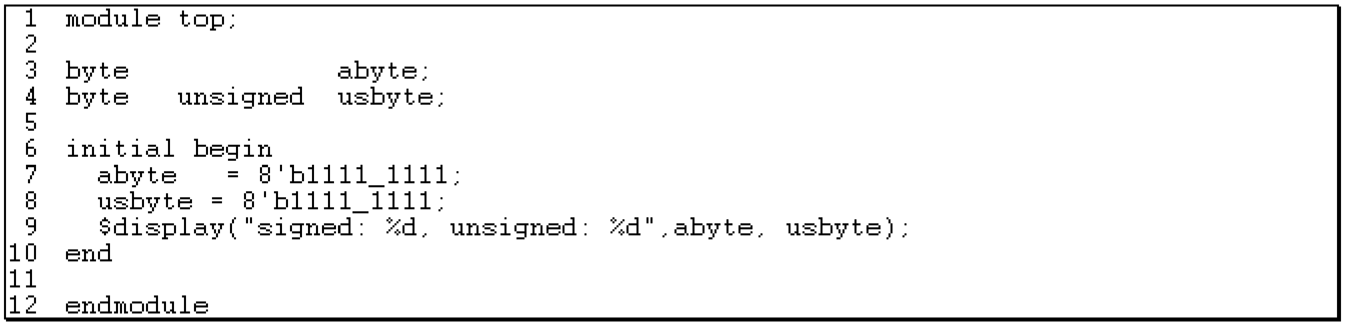
The 2-state data types can be further broken down in integral data types and floating-point data types. Here are the integral data types:

- **`shortint`**—A 16-bit signed integer.
- **`int`**—A 32-bit signed integer
- **`longint`**—A 64-bit signed integer
- **`byte`**—An 8-bit signed integer
- **`bit`**—A 2-state variable in which the user can define the width. It is 1-bit by default, and it is unsigned.

7.1.3 Signed and Unsigned

By default, all the 2-state variables (except `bit`) are signed. So a byte variable set to `8'b11111111` is equal to -1, not 255. A signed 8-bit variable can take the values of -127 to 128, while an unsigned 8-bit value can take the values from 0 to 255.

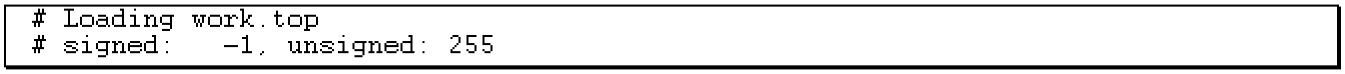
We change our signed variables into unsigned variables with the `unsigned` keyword, like this:



```
module top;

byte             abyte;
byte   unsigned  usbyte;

initial begin
  abyte   = 8'b1111_1111;
  usbyte = 8'b1111_1111;
  $display("signed: %d, unsigned: %d",abyte, usbyte);
end

endmodule
```

FIGURE 13. **Defining a Variable as Unsigned**

This code gives the following result when we run it:



```
# Loading work.top
# signed:   -1, unsigned: 255
```

FIGURE 14. **Results of `signed` and `unsigned`**

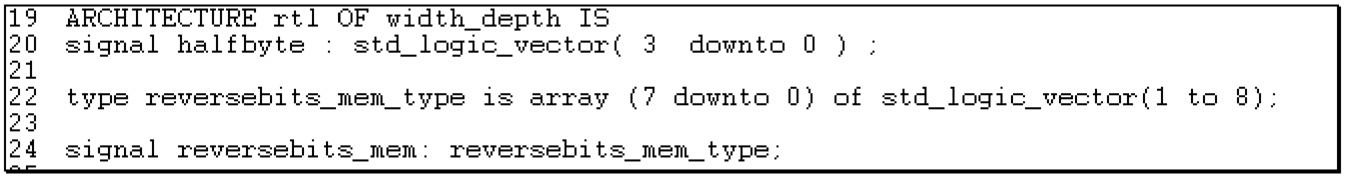
7.2 Defining Data Widths and Depths

The `logic` (and `reg`) data types are one bit wide by default. Of course, there are very few one-bit registers in a design, and we have memories, so we need to be able to define width and depth when we define data.

As an example let's define a register and a memory:

- **`halfbyte`**—Four bits indexed where the MSB is number 3 and the LSB is number 0.
- **`reversebits_mem`**—A memory where each data word has an MSB of 1 and an LSB of 8, and the address run from 7 at the top of the memory to 0.

Here are these signals defined in VHDL:

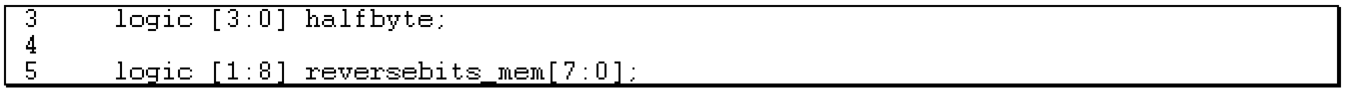


```
19  ARCHITECTURE rtl OF width_depth IS
20  signal halfbyte : std_logic_vector( 3  downto 0 ) ;
21
22  type reversebits_mem_type is array (7 downto 0) of std_logic_vector(1 to 8);
23
24  signal reversebits_mem: reversebits_mem_type;
```

FIGURE 15. **Declaring Registers and Memories in VHDL**

- **Line 20**—We define halfbyte to be a `std_logic_vector` with four bits with the MSB at 3 and the LSB at 0.
- **Line 22**—VHDL uses a type to create an array. Each word in this array starts at bit 1 and goes up to bit 8. Then the array elements are numbered from 7 at the top word down to 0.
- **Line 24**—Declare a signal of the array type.

This little example shows how VHDL declares registers and arrays. Here's how SystemVerilog declares the same register and memory:



```
3     logic [3:0] halfbyte;
4
5     logic [1:8] reversebits_mem[7:0];
```

FIGURE 16. **Declaring Registers and Memories in SystemVerilog**

- **Line 3**—When we create a register in SystemVerilog, we put the bit addresses immediately after the type name in square brackets. Then every variable declared on this line has the same width. Notice also that SystemVerilog has no `to` or `downto` keywords. You use the colon for both directions[1].
- **Line 5**—You declare an array in SystemVerilog by placing the bounds of the array after the variable name within square brackets. You can declare the width of the variable's data word, and the depth of the array in the same line. Notice again, the colon serves double duty.

1. This type of variable, with the indices on the left of the variable name is sometimes called a *packed array* in SystemVerilog error messages. Whe you get an error complaining that you are trying to put an array into a packed array, it means you are trying to stuff an entire memory into a register.

You access the data bits and the memory words in SystemVerilog using the square brackets as indices. In this example we will load up the memory and assign four bits of it to our register:

```
 1  module top;
 2
 3     logic [3:0] halfbyte;
 4
 5     logic [1:8] reversebits_mem[7:0];
 6
 7     int         address;
 8
 9     initial begin
10        for (address = 0; address <= 7; address++)
11          reversebits_mem[address] = address;
12
13        halfbyte = reversebits_mem[6][5:8];
14
15        $display("reversebits_mem[6]: %b",reversebits_mem[6]);
16        $display("halfbyte = reversebits_mem[6][5:8];");
17        $display("halfbyte: %b", halfbyte);
18        $display("two bits - halfbyte[2:1]: %b", halfbyte[2:1]);
19     end
20
21  endmodule // top
```

FIGURE 17. **Bit Select Example**

- **Lines 10–11**—We initialize the memory by storing the address in each memory location. Notice that we did not need `begin` and `end` because there is only one statement in this loop.
- **Line 13**—We select four bits of the memory and place them into half byte. These are the bottom four bits of memory location 6. Notice that we use the square brackets to access the bits. Also notice that when we have multiple indices that we access the memory location first and then the bits.
- **Lines 15–18**—Display the information.

Here is what happens when we run this example:

```
# reversebits_mem[6]: 00000110
# halfbyte = reversebits_mem[6][5:8];
# halfbyte: 0110
# two bits - halfbyte[2:1]: 11
```

FIGURE 18. **Bit Select in Action**

8 Variable Assignments

Now it's time to take these variables and start moving them around in programs. While both VHDL and Verilog have similar assignment approaches, their philosophy around legal assignments, and what the simulator will do for you, are worlds apart. In this section we'll discuss three topics:

- Type conversion and bit width management
- Concurrent assignments
- Procedural assignments, both blocking and nonblocking.

We'll start by talking about type conversion, because this affects all other assignments.

8.1 Type Conversion and Width Management

SystemVerilog was designed to create hardware, and so the simulator automatically implements type conversion and width management policies that match the needs of hardware developers. This is disconcerting to VHDL writers, because a lot of what happens in a SystemVerilog program consists of hidden magic, whereas VHDL requires developers to define the conversions explicitly.

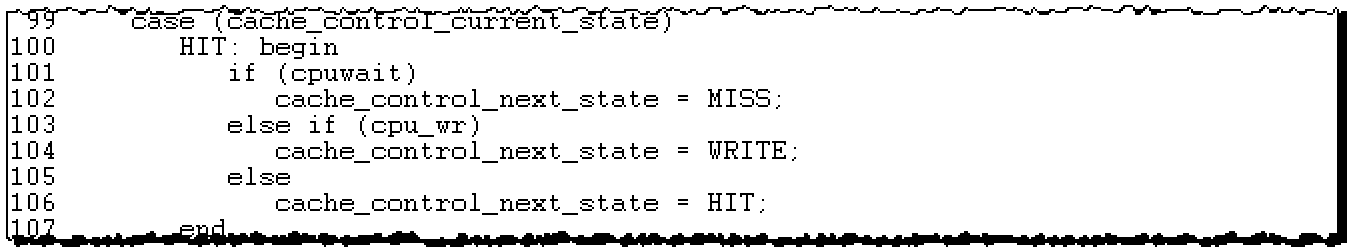
This discussion focuses on the integral data types. Engineers use these types to describe logic or create test benches. Because all these data types describe strings of bits, SystemVerilog simply allows you to assign them to each other. There is no need to explicitly cast an `int` to a `reg`, for example. SystemVerilog simply "does the right thing" here by moving the bits over.

8.1.1 Boolean Operations and Conditionals

SystemVerilog has no explicit boolean data type. Instead, all conditionals follow the same rule:

- Zero means false.
- Nonzero means true.

This means that I can write code such as the following. This is a snippet from a cache:



```
    case (cache_control_current_state)
       HIT: begin
          if (cpuwait)
             cache_control_next_state = MISS;
          else if (cpu_wr)
             cache_control_next_state = WRITE;
          else
             cache_control_next_state = HIT;
       end
```

FIGURE 19. **Boolean Conversions in Action**

This snippet of code is part of a cache's state machine. The cache checks for two signals:

- `cpuwait`—This indicates that there was a cache miss.
- `cpu_wr`—This is a control signal from the CPU, indicating that this is a write operation.

Neither of these signals was declared as a boolean; in fact, they are both wire types. SystemVerilog treats these signals as boolean operators in this context; they are true if they are nonzero.

In VHDL we would have explicitly written `(“cpuwait='1')` or `(cpu_wr='1')` to implement the same behavior.

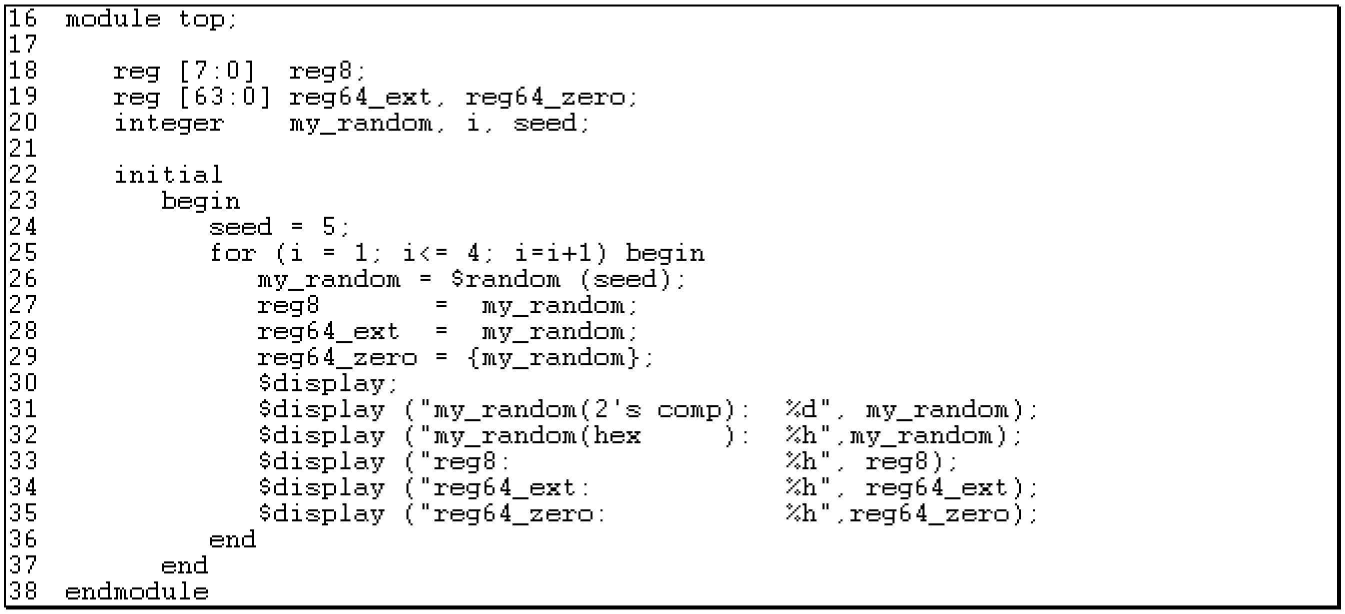
8.1.2 Width Conversion

SystemVerilog allows you to make assignments where the number of bits on the right-hand side (RHS) of the assignment is different from the number of bits on the left-hand side (LHS) of the assignment. The simulator automatically manages the width differences by following these rules:

- If the LHS is shorter than the RHS, then SystemVerilog truncates the larger value by getting rid of the most significant bits.
- If the LHS is wider than the RHS, then SystemVerilog copies the most significant bit of the RHS into the extra space, unless the RHS is unsigned. In that case, SystemVerilog puts zero into the extra space.

This automatic width management can create some surprises for the programmer. For example, if you try to put the constant `4'h8` into a 3-bit value, you'll get `3'h0` instead, because the top bit will be cut off.

Here is an example from the book *FPGA Simulation: A Complete Step-by-Step Guide* that demonstrates truncation and extension:

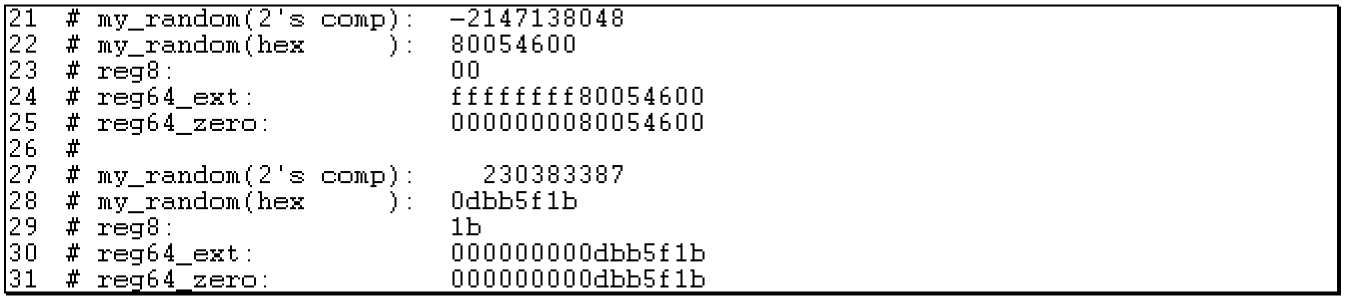


```
16  module top;
17
18     reg [7:0]  reg8;
19     reg [63:0] reg64_ext, reg64_zero;
20     integer    my_random, i, seed;
21
22     initial
23        begin
24           seed = 5;
25           for (i = 1; i<= 4; i=i+1) begin
26              my_random = $random (seed);
27              reg8       =  my_random;
28              reg64_ext  =  my_random;
29              reg64_zero = {my_random};
30              $display;
31              $display ("my_random(2's comp):  %d", my_random);
32              $display ("my_random(hex     ):  %h",my_random);
33              $display ("reg8:                 %h", reg8);
34              $display ("reg64_ext:            %h", reg64_ext);
35              $display ("reg64_zero:           %h",reg64_zero);
36           end
37        end
38  endmodule
```

FIGURE 20. **Truncation and Extension**

- **Line 18**—The variable `reg8` is an 8-bit 4-state variable.
- **Line 19**—`reg64_ext` and `reg64_zero` are 64-bit variables.
- **Line 20**—The variables `my_random`, `i`, and `seed` are 32-bit variables.
- **Line 26**—The system call `$random` returns 32 random bits. These go into `my_random`.
- **Line 27**—`reg8` gets the bottom 8 bits of `my_random`.
- **Line 28**—`reg64_ext` gets the 32-bits from `my_random`. SystemVerilog sign-extends the MSB in `my_random` because an integer is a signed data type.
- **Line 29**—The {}'s implement a trick. They concatenate `my_random` with nothing, but because concatenation returns an unsigned value, SystemVerilog will not sign-extend the MSB in `my_random`.
- **Lines 31–35**—Display our values. Notice the C-like formatting in `$display`.

When we run this example we get the following results:



```
21  # my_random(2's comp):   -2147138048
22  # my_random(hex     ):   80054600
23  # reg8:                  00
24  # reg64_ext:             ffffffff80054600
25  # reg64_zero:            0000000080054600
26  #
27  # my_random(2's comp):     230383387
28  # my_random(hex     ):   0dbb5f1b
29  # reg8:                  1b
30  # reg64_ext:             000000000dbb5f1b
31  # reg64_zero:            000000000dbb5f1b
```

FIGURE 21. **Truncation and Extension in Action**

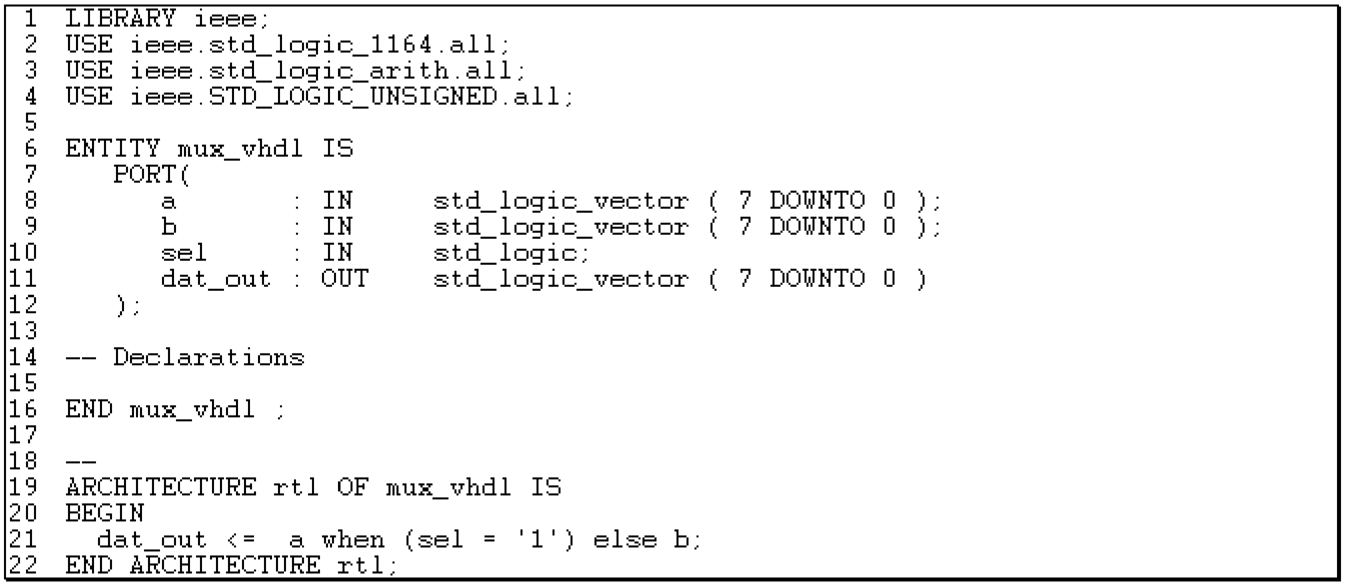
- **Line 21**—We generate a random set of 32 bits. In this case the most-significant bit is set, so the signed view of the number is negative.
- **Line 22**—The hex view shows the MSB set.
- **Line 23**—Assigned the 32-bit number to an 8-bit variable, so the top 24 bits are gone.
- **Line 24**—Assigned the 32-bit signed number to a 64-bit variable, so the top bit was extended.
- **Line 25**—We used the concatenation trick to convert the signed number to an unsigned number. Then we assigned a 32-bit unsigned number to a 64-bit number and see zeros in the top 32 bits.
- **Lines27–31**—We did the same things with a different random number, but in this case the most significant bit was zero.

The biggest difference between SystemVerilog and VHDL in the area of assignments is that Verilog tries to do the right thing (and usually succeeds) when you assign data, and VHDL makes you connect all the dots syntactically.

8.2 Concurrent Assignments

Both VHDL and SystemVerilog allow you to define expressions that the simulator evaluates continuously. We call these *concurrent assignments* in VHDL, and we call them *continuous assignments* in SystemVerilog. SystemVerilog uses the `assign` keyword to define continuous assignments.

For example, here is a MUX defined in VHDL with a concurrent assignment:



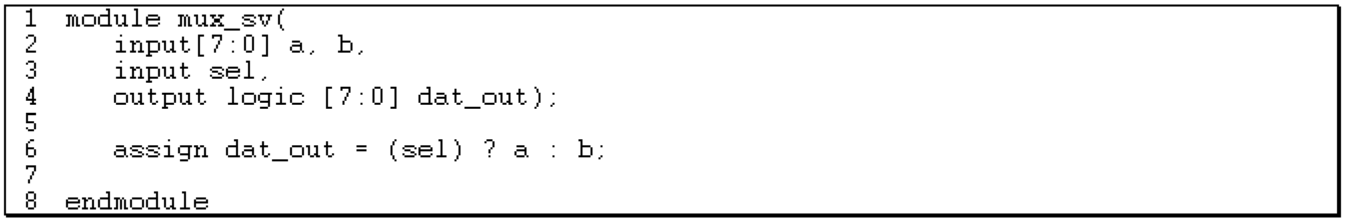
```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  USE ieee.std_logic_arith.all;
 4  USE ieee.STD_LOGIC_UNSIGNED.all;
 5
 6  ENTITY mux_vhdl IS
 7     PORT(
 8        a       : IN     std_logic_vector ( 7 DOWNTO 0 );
 9        b       : IN     std_logic_vector ( 7 DOWNTO 0 );
10        sel     : IN     std_logic;
11        dat_out : OUT    std_logic_vector ( 7 DOWNTO 0 )
12     );
13
14  -- Declarations
15
16  END mux_vhdl ;
17
18  --
19  ARCHITECTURE rtl OF mux_vhdl IS
20  BEGIN
21    dat_out <=  a when (sel = '1') else b;
22  END ARCHITECTURE rtl;
```

FIGURE 22. **VHDL MUX with Concurrent Assignment**

- **Line 21**—This concurrent assignment implements the MUX.

We see that the architecture consists only of the concurrent assignment. We use the conditional assignment to implement the MUX functionality.

Here is the same design in SystemVerilog:



```
1  module mux_sv(
2     input[7:0] a, b,
3     input sel,
4     output logic [7:0] dat_out);
5
6     assign dat_out = (sel) ? a : b;
7
8  endmodule
```

FIGURE 23. **SystemVerilog MUX with Continuous Assignment**

- **Line 6**—This continuous assignment implements the MUX using a C-like conditional assignment operator.

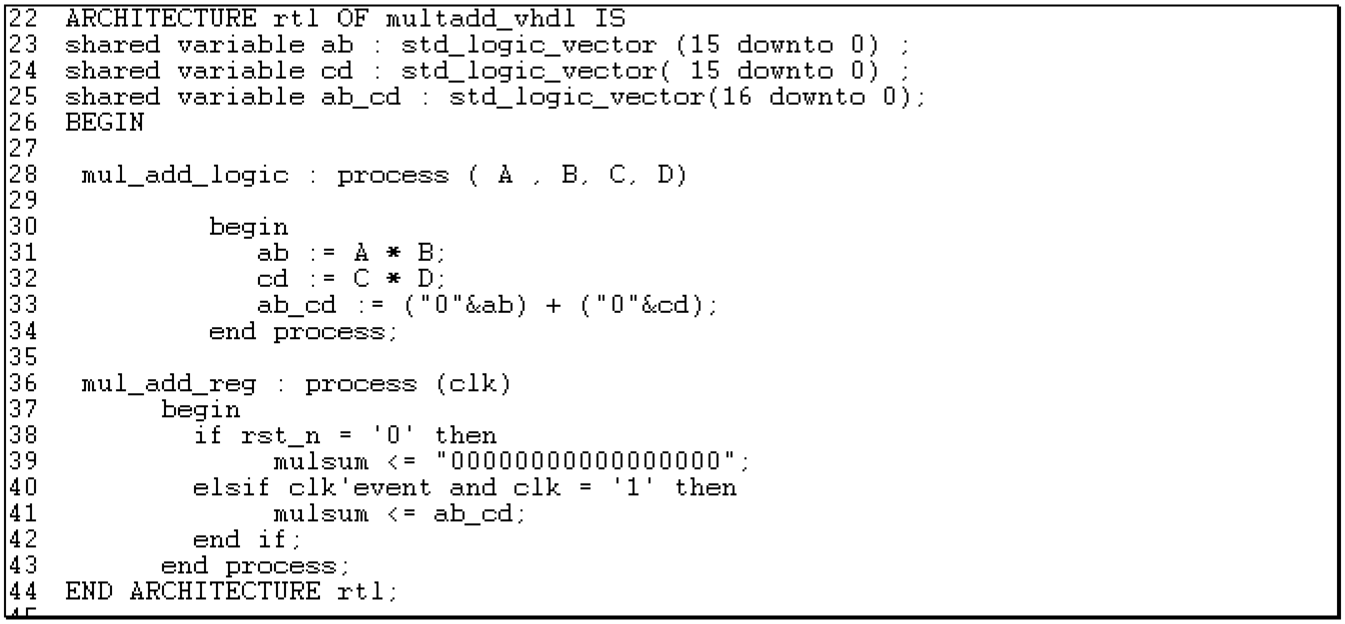
We can see Verilog's C heritage in the conditional assignment operator. This continuous assignment, with the `assign` keyword, is just like the concurrent assignment in VHDL.

8.3 Blocking and Nonblocking Procedural Assignments

Both VHDL and Verilog allow you to create assignments that either (1) happen all together in the same clock edge, to simulate assigning values to a register (nonblocking); or (2) happen sequentially, to implement an algorithm (blocking).

Both languages use the <= operator to indicate a nonblocking assignment. The difference is that SystemVerilog can use any type for nonblocking assignments, whereas VHDL requires that you use the `variable` type for nonblocking assignments.

Here is an example of a multiply/ADD circuit with blocking and nonblocking assignments in VHDL:



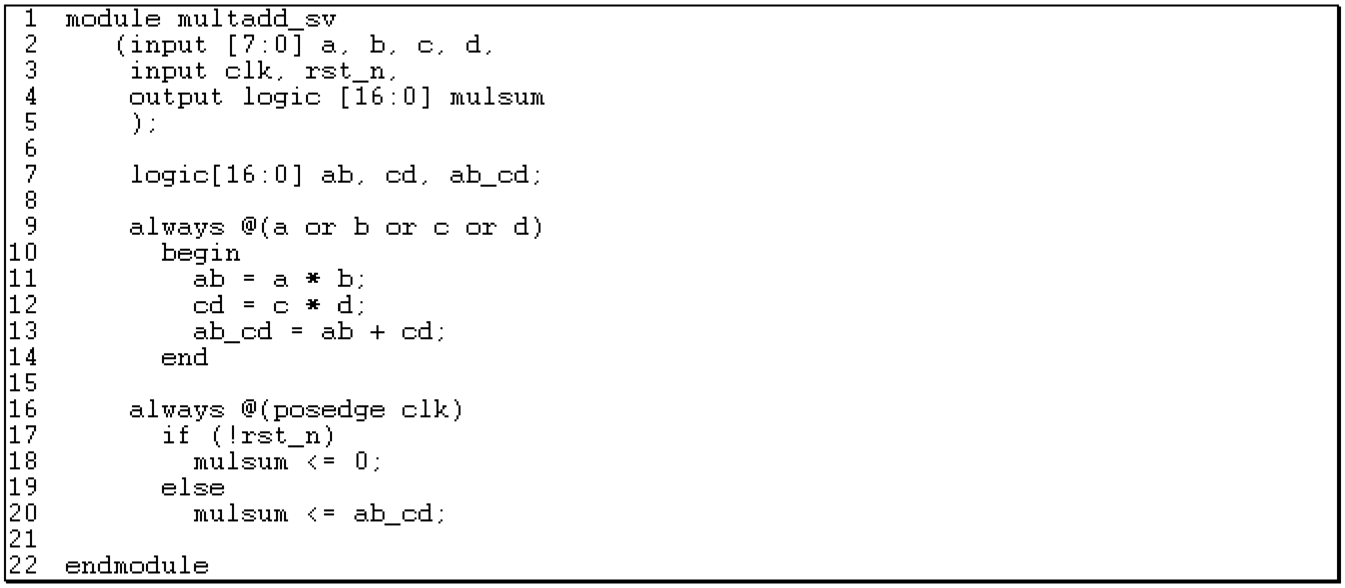
```
22  ARCHITECTURE rtl OF multadd_vhdl IS
23  shared variable ab : std_logic_vector (15 downto 0) ;
24  shared variable cd : std_logic_vector( 15 downto 0) ;
25  shared variable ab_cd : std_logic_vector(16 downto 0);
26  BEGIN
27
28   mul_add_logic : process ( A , B, C, D)
29
30           begin
31              ab := A * B;
32              cd := C * D;
33              ab_cd := ("0"&ab) + ("0"&cd);
34           end process;
35
36   mul_add_reg : process (clk)
37        begin
38          if rst_n = '0' then
39               mulsum <= "00000000000000000";
40          elsif clk'event and clk = '1' then
41               mulsum <= ab_cd;
42          end if;
43        end process;
44  END ARCHITECTURE rtl;
```

FIGURE 24. **Multiply Add with VHDL**

- **Lines 23–25—**We declare variables to use in our blocking assignment. We have a signal (`mulsum`) for the nonblocking assignment.
- **Line 31–33**—These are blocking assignments in VHDL. We know that `ab` and `cd` get calculated before they are added to create `ab_cd`.
- **Line 41**—This is a nonblocking assignment. It happens right at the clock edge and then moves on. If there were multiple nonblocking assignments, they would all happen right at the clock edge in no guaranteed order. We declared `mulsum` to be a signal in the entity for our multiply add block.

This VHDL design demonstrated both blocking and nonblocking assignments. We used blocking assignments, implemented with the := operator, in a combinatorial process to create the multiply/add value. Then we used nonblocking assignments, implemented with the <= operator, in a clocked process to assign the value to a register. This is a good coding style that avoids race conditions.

Now let's implement the same code in SystemVerilog:



```
 1  module multadd_sv
 2     (input [7:0] a, b, c, d,
 3      input clk, rst_n,
 4      output logic [16:0] mulsum
 5      );
 6
 7      logic[16:0] ab, cd, ab_cd;
 8
 9      always @(a or b or c or d)
10        begin
11          ab = a * b;
12          cd = c * d;
13          ab_cd = ab + cd;
14        end
15
16      always @(posedge clk)
17        if (!rst_n)
18          mulsum <= 0;
19        else
20          mulsum <= ab_cd;
21
22  endmodule
```

FIGURE 25. **Multiply/Add in SystemVerilog**

- **Lines 2–4**—The inputs are all implied to be wires and the output is declared to be `logic`. The types have nothing to do with whether the variables are used in blocking or nonblocking assignments.
- **Line 7**—We've declared the intermediate values to be 17-bit `logic` variables. This allows us to handle the maximum result size easily. Notice that there is no distinction between `signal` and `variable` in SystemVerilog. Both can do blocking and nonblocking assignments.
- **Lines 9–13**—This combinatorial `always` block has blocking assignments, implemented with the = operator. You should always use blocking assignments when implementing combinatorial logic.
- **Lines 16–20**—This clocked block either resets the output register or sets it equal to the result of the combinatorial logic. It uses a single nonblocking assignment by means of the <= operator.

In this section we learned how to implement blocking and nonblocking behavior in SystemVerilog. We saw that SystemVerilog controls this behavior entirely through the operators = (blocking) and <= (nonblocking).

Now we'll put all this information together in a test bench example.

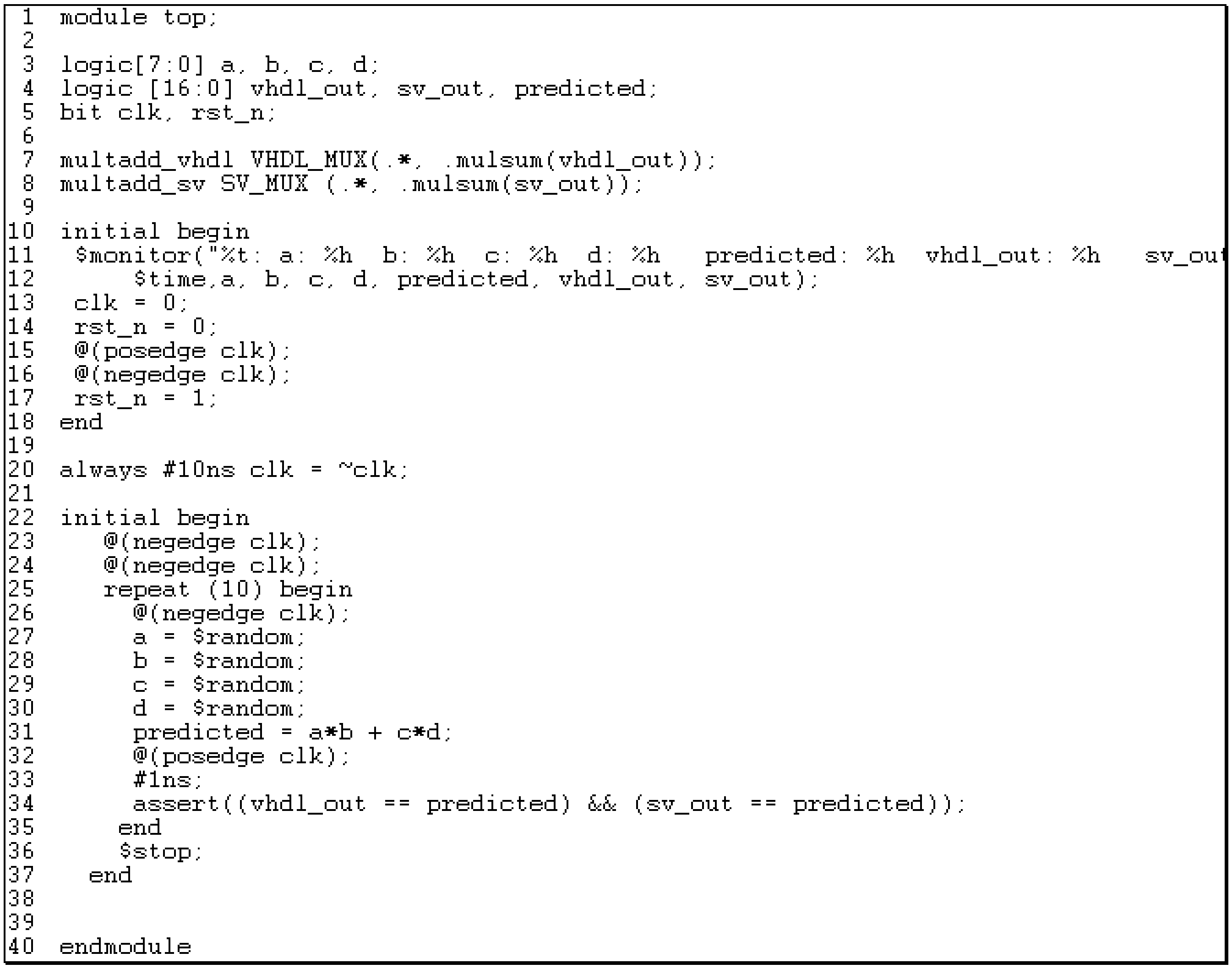
9 Example Test Bench

As we've discussed, SystemVerilog is a new language targeted at creating test benches. In this primer, we saw how SystemVerilog works by using the concepts from VHDL. We'll now put those concepts together in an sample test bench. We are going to test our multiplier from Chapter 8. You can find this code in the folder `systemverilog_primer/assignments/blocking_non_blocking`.

We will create a test bench that ensures that the VHDL and SystemVerilog multipliers give the same results, and that those results are correct. You could use a similar test bench to check whether your Verilog gate-level simulation gives the same results as your VHDL RTL.[1]

This test bench demonstrates two features of advanced test benches, even though it is relatively simple. It generates its own stimulus, and it checks its own results.

1. Verilog gate-level simulations are inherently 3 to 5 times faster than VHDL gate-level simulations. This is because of the assumptions built into the Verilog simulators, which must be defined explicitly in VHDL.



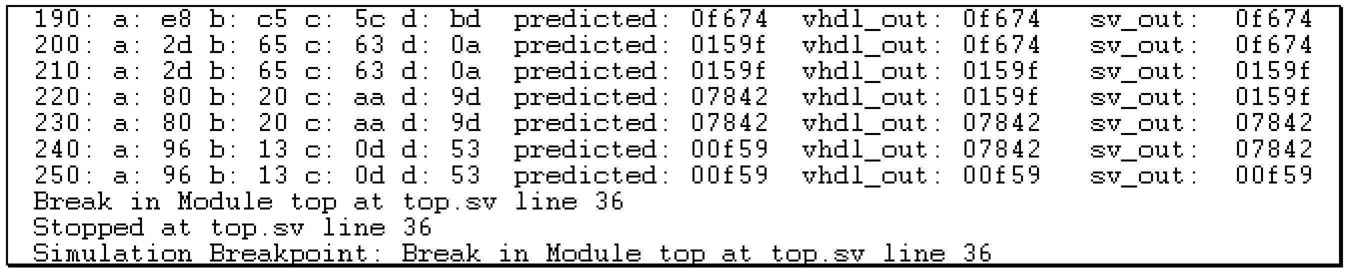
```
 1  module top;
 2
 3  logic[7:0] a, b, c, d;
 4  logic [16:0] vhdl_out, sv_out, predicted;
 5  bit clk, rst_n;
 6
 7  multadd_vhdl VHDL_MUX(.*, .mulsum(vhdl_out));
 8  multadd_sv SV_MUX (.*, .mulsum(sv_out));
 9
10  initial begin
11   $monitor("%t: a: %h  b: %h  c: %h  d: %h   predicted: %h  vhdl_out: %h   sv_ou
12       $time,a, b, c, d, predicted, vhdl_out, sv_out);
13   clk = 0;
14   rst_n = 0;
15   @(posedge clk);
16   @(negedge clk);
17   rst_n = 1;
18  end
19
20  always #10ns clk = ~clk;
21
22  initial begin
23     @(negedge clk);
24     @(negedge clk);
25     repeat (10) begin
26       @(negedge clk);
27       a = $random;
28       b = $random;
29       c = $random;
30       d = $random;
31       predicted = a*b + c*d;
32       @(posedge clk);
33       #1ns;
34       assert((vhdl_out == predicted) && (sv_out == predicted));
35      end
36      $stop;
37    end
38
39
40  endmodule
```

FIGURE 26. **SystemVerilog Test Bench for Multiply/Add**

- **Line 3**—We will use these variables to create inputs randomly. We use lowercase here to conform with variable names in VHDL.
- **Line 4**—Here we see how `logic` can be used both in procedural code (in a process) or as a wire. The variables `vhdl_out` and `sv_out` will act as wires, while `predicted` will be calculated procedurally.
- **Line 5**—Our test bench needs to generate the reset and the clock.
- **Line 7**—We instantiate the VHDL version of the DUT. Notice that SystemVerilog allows you to declare the port and signal connection in instantiation. The connection `.mulsum(sv_out)` is the same as `mulsum=>sv_out` in VHDL.
- **Line 8**—We instantiate the SystemVerilog DUT. Notice this trick in SystemVerilog. If you use the wildcard expression `.*`, then the simulator will find ports in the component that have the same name as variables in this level and hook them up. As you can see in Figure 25 on page 28, the multiplier has ports named `a`, `b`, `c`, `d`, `clk`, and `rst`. These match the ports in the test bench. We had to connect the port `mulsum` explicitly.
- **Line 10**—This `initial` block lives up to its name. It initializes the test bench.

- **Line 11**—The `$monitor` system call prints to the screen any time on of its inputs changes. In this case we are printing all the inputs and outputs from the blocks.
- **Line 13**—Set the `clk` to `0` to start the simulation.
- **Line 14**—Set the `rst` to `0` to start the simulation.
- **Lines 15 and 16**—Wait for a clock and a half before raising the reset.
- **Line 20**—This block generates the clock. It waits 10 ns before inverting the `clk` signal. Because the `initial` block set `clk` to `0` at time `0`, this block will create the first positive edge of `clk` at time `10`. This is the positive edge that the `initial` block sees on line 15.
- **Line 22**—This `initial` block implements the test.
- **Lines 23 and 24**—Wait for a couple of clocks for the reset to take effect.
- **Line 25**—We will execute 10 tests. The `repeat` loop is the simplest looping tool in SystemVerilog. SystemVerilog also has `for` loops and `while` loops. You can also say `repeat forever`, but we don't want that here.
- **Line 26**—We will change the input data to the DUT on the negative edge of the clock.
- **Lines 27–30**—The `$random` system call returns a 32-bit string of random bits. The simulator truncates these to fit `a`, `b`, `c`, and `d`. It would have been fancier to say `{a,b,c,d} = $random`, because the operands are each a byte wide and there are four of them, making 32 bits—but that would just be showing off. This approach is much clearer.
- **Line 31**—We calculate the expected result.
- **Line 32**—Wait for the positive edge of the clock so that the DUTs calculate the value.
- **Line 33**—Wait for 1 ns to avoid a race condition between the result being generated and checked.
- **Line 34**—The `assert` statement will fire if the condition inside is not true. The expression says that both the VHDL and SystemVerilog DUT outputs must match the predicted value.
- **Line 35**—The `$stop` system call stops the simulation but does not quit out of the simulator. You want `$finish` for that.

When we run the simulation we get the following result:



```
190: a: e8 b: c5 c: 5c d: bd  predicted: 0f674  vhdl_out: 0f674   sv_out:  0f674
200: a: 2d b: 65 c: 63 d: 0a  predicted: 0159f  vhdl_out: 0f674   sv_out:  0f674
210: a: 2d b: 65 c: 63 d: 0a  predicted: 0159f  vhdl_out: 0159f   sv_out:  0159f
220: a: 80 b: 20 c: aa d: 9d  predicted: 07842  vhdl_out: 0159f   sv_out:  0159f
230: a: 80 b: 20 c: aa d: 9d  predicted: 07842  vhdl_out: 07842   sv_out:  07842
240: a: 96 b: 13 c: 0d d: 53  predicted: 00f59  vhdl_out: 07842   sv_out:  07842
250: a: 96 b: 13 c: 0d d: 53  predicted: 00f59  vhdl_out: 00f59   sv_out:  00f59
Break in Module top at top.sv line 36
Stopped at top.sv line 36
Simulation Breakpoint: Break in Module top at top.sv line 36
```

FIGURE 27. **Results of Multiply/Add Test Bench**

You can see the `$monitor` statement in action here. It prints twice for each test value, once when the inputs change, and again when the outputs change. `$monitor` always runs at the end of a time step. Notice that the prediction happens in the same timestep in which the inputs change, and then the outputs follow the prediction.

10 Why Learn SystemVerilog?

This primer has covered the basic structure of SystemVerilog in the context of VHDL. While it is far from a complete description of SystemVerilog (the *SystemVerilog Language Reference Manual* is over 1400 pages long), it does give you enough to get started with SystemVerilog. It also brings us to the question of why you should bother to learn SystemVerilog.

SystemVerilog is worth learning because the industry, that amorphous group of all of us that makes a bandwagon, has decided it is the future of RTL verification. While VHDL can do many of the things that are possible in SystemVerilog (though not some crucial ones such as covergroups), VHDL does not have the industry behind it, and so VHDL does not have free class libraries designed to support RTL verification.

While different EDA companies tout different class libraries (Synopsys supports the VMM, while Cadence and Mentor support the OVM), all three companies support SystemVerilog as their language of choice for verification.

This means that verification libraries and verification IP will soon appear exclusively in SystemVerilog. This is already happening. The Open Verification Library (OVL), which supplies assertions to engineers, has 50 assertions written in SystemVerilog but only eight that are written in VHDL. This trend continues in other areas. This industry support makes it well worth learning SystemVerilog.

I hope that this primer has helped you get started down the path to learning SystemVerilog. If you have any questions about the topics in this primer, please post them at www.fpgasimulation.com/forums. I will answer them there, and use them to create future versions of this primer. I'll notify you of updates on the FPGA Simulation email newsletter.

11 Revisions

TABLE 1-3. SystemVerilog Primer Revisions

Revision	**Changes**
0.1 Beta	Test release.
0.2	Revisions from Michael Meyer
1.0	Initial release