CHAPTER 1

Verilog Design in the Real World

The challenges facing digital design engineers in the Real World have changed as technology has advanced. Designs are faster, use larger numbers of gates, and are physically smaller. Packages have many fine-pitch pins. However, the underlying design concerns have not changed, nor will they change in the future. The designer must create designs that:

- are understandable to others who will work on the design later.
- are logically correct. The design must actually implement the specified logic correctly. The designer collects user specifications, device parameters, and design entry rules, then creates a design that meets the needs of the end user.
- perform under worst-case conditions of temperature and process variation. As devices age and are exposed to changes in temperature, the performance of the circuit elements change. Temperature changes can be self-generated or caused by external heat sources. No two devices are exactly equivalent, particularly

devices that were manufactured at different times, perhaps at different foundries and perhaps with different design rules. Variations in the device timing specifications, including clock skew, register setup and hold times, propagation delay times, and output rise/fall times must be accounted for.

- are reliable. The end design cannot exceed the package power dissipation limits. Each device has an operational temperature range. For example, a device rated for commercial operation has a temperature rating of 0 to 70 degrees C (32 to 160 degrees F). The device temperature includes the ambient temperature (the temperature of the air surrounding the product when it is in use), temperature increases due to heat-generating sources inside the product, and heat generated by the devices of the design itself. Internally generated temperature rises are proportional to the number of gates and the speed at which they are changing states.
- do not generate more EMI/RFI than necessary to accomplish the job and meet EMI/RFI specifications.
- are testable and can be proven to meet the specifications.
- do not exceed the power consumption goals (for example, in a battery-operated circuit).

These requirements exist regardless of the final form of the design and regardless of the engineering tools used to create and test the design.

SYNTHESIS: the translation of a high-level design description to target hardware. For the purposes of this book, synthesis represents all the processes that convert Verilog code into a netlist that can be implemented in hardware.

The job of the digital designer includes writing HDL code intended for synthesis. This HDL code will be implemented in the target hardware and defines the operation of the shippable product. The designer also writes code intended to stimulate and test the output of the design. The designer writes code in a language that is easy for humans to understand. This code must be translated by a compiler into a form appropriate for the final hardware implementation.

Why HDL?

There are other methods of creating a digital design, for example: using a schematic. A schematic has some advantages: it's easy to create a design more tailored to the FPGA, and a more compact and faster design can be created. However, a schematic is not portable and schematics become unmanageable when a design contains more than 10 or 20 sheets. For large and portable designs, HDL is the best method.

As a contrast between a Verilog design found in other books and a Real World design, consider the code fragments in Listings 1-1 and 1-2.

Listing 1-1 Non- Real World Example

```
// Transfer the content of register b to register a.
    a      <=     b;
```

Listing 1-2 Real World Example

```
/* Signal b must transfer to signal a in less than 7.3 nsec in a
–3 speed grade device as part of a much larger design that must
draw less than 80 uA while in standby and 800 uA while operating.
The whole design must cost less than $1.47, pass CE testing, and
take less than two months to be written, debugged, integrated,
documented, and shipped to the customer. Signal a must be
synchronized to the 75 MHz system clock and reset by the global
system reset. The signal b input should be located at or near pin
79 on the 208-pin package in order to help meet the setup and
hold requirement of register a.*/

    a      <=     b;
```

To illustrate the design process, let's follow a trivial example from concept to delivery and examine the issues that the designer confronts when implementing the design. Don't worry if the Verilog language elements are unfamiliar; they will be covered in detail later in this chapter.

TRIVIAL OVERHEAT DETECTOR EXAMPLE

Sarah, the Engineering Manager, writes the following email to Sam, the digital designer.

To: sam@engineering
From: sarah@management
Subject: Hot Design Project.

```
The customer wants a red light that turns on and stays on if a
button is pressed and if their machine is overheating. They
want it yesterday, it needs to be battery operated, and has to
have a final build cost of $0.02 so the company can make money
when they sell it for $9.95.
```

First Sam estimates the scope of the design. From experience, she determines that this circuit is very similar to a design she did last year. She counts the gates of the previous design, factors in the differences between the two designs, and decides the design is approximately 20 gates. She considers the speed that the design must run at and any other complicating factors she can think of, including the error she made in estimating complexity of the previous design and the fact that she's already purchased airline tickets for a week of vacation. She knows that, overall, including design, test, integration, and documentation, she can design 2000 gates a month without working significant overtime. She counts the number of pins (the specification lists a pushbutton input, an overheat input, and an overheat output, but Sarah realizes that she’ll need to add at least a reset and clock input). From the gate-count estimate and the pin estimate she can select a device. She picks a device that has more pins than she needs because she knows the design will grow as features are added. She picks an FPGA package from a family that has larger and faster parts available so she is not stuck if she needs more logic or faster speed. Now she sends a preliminary schedule and part selection to her boss and starts working on the design. Her boss will thank her for her thorough work on the cost and schedule estimates, but will insist that the job be done faster to be ready for an important trade show and cheaper to satisfy the marketing department.

Keep in mind that rarely will your estimates be low. Even when we know better, engineers are eternally optimistic. Unless you are very smart and very lucky, your estimate will not allow enough contingency to cover growth of the design (feature-creep) the hassles associated with fitting a high-speed design into a part that is too small, and the other 1001 things that can go wrong. These estimating errors result in overtime hours and increased project cost.

Now that Sam has taken care of the up-front project-related chores, she can start working on the design. Sam recognizes that a simple flipflop circuit will perform this function. She also recognizes, because of the problems she had with an earlier project, that a synchronous digital design is the right approach to solving this problem. Sam creates a Verilog design that looks like Listing 1-3.

Listing 1-3 Overheat Detector Design Example

```
module overheat (clock, reset, overheat_in, pushbutton_in,
overheat_out);
input      clock, reset, overheat_in, pushbutton_in;
output     overheat_out;
reg        overheat_out;
reg        pushbutton_sync1, pushbutton_sync2;
reg        overheat_in_sync1, overheat_in_sync2;

// Always synchronize inputs that are not phase related to
//  the system clock.
// Use double-synchronizing flipflops for external signals
//  to minimize metastability problems.
// Even better would be some type of filtering and latching
//  for poorly behaving external signals that will bounce
//  and have slow rise/fall times.

always @ (posedge clock or posedge reset)
begin
    if (reset)
          begin
          pushbutton_sync1   <=    1'b0;
          pushbutton_sync2   <=    1'b0;
          overheat_in_sync1  <=    1'b0;
          overheat_in_sync2  <=    1'b0;
          end
    else  begin
          pushbutton_sync1   <=    pushbutton_in;
          pushbutton_sync2   <=    pushbutton_sync1;
          overheat_in_sync1  <=    overheat_in;
          overheat_in_sync2  <=    overheat_in_sync1;
          end
end

// Latch the overheat output signal when overheat is
//  asserted and the user presses the pushbutton.
always @ (posedge clock or posedge reset)
begin
    if (reset)
          overheat_out <=    1'b0;

// Overheat_out is held forever (or until reset).
    else if (overheat_in_sync2 && pushbutton_sync2)
          overheat_out <=    1'b1;
end

endmodule
```

This seems like a lot of typing for such a simple circuit, doesn't it? The first always element appears to do nothing and looks like it could be deleted. In a previous design, Sam had problems (which will be discussed in Chapter 2) with erratic logic behavior, so she always double-synchronizes inputs from the Real World. The second always block asserts pushbutton_out when overheat_in_sync and pushbutton_sync are asserted.

A useful method estimating the size of a design is to count the semicolons. A utility called *metric*, which counts semicolons in a module, is included on the Real World FPGA Design with Verilog CD-ROM. This method should be used informally to avoid designers developing a semicolon-rich coding style :^).

Sam has done the fun part of the design: the actual designing of the code. She quickly runs her compiler, simulator, or Lint program to make sure there are no typographical or syntax errors. Next, because writing test vectors is almost as much fun as designing the code, Sam does a test fixture and checks out the behavior of her design. Her test fixture looks something like Listing 1-4.

Listing 1-4 Overheat Detector Test Fixture

```
// Overheat detector test fixture.
// Created by Sam Stephens

`timescale 1ns / 1ns

module oheat_tf;

reg clock, system_reset, overheat_in, pushbutton_in;

parameter clk_period          =      33.333;

overheat u1 (clock, system_reset, overheat_in, pushbutton_in,
overheat_out);

always     begin
    #clk_period clock   =      ~clock; // Generate system clock.
          end

initial
begin
          clock               =      0;
          system_reset        =      1; // Assert reset.
          overheat_in         =      0;
          pushbutton_in       =      0;
#75       system_reset        =      0;
end

// Toggle the input and see if overheat_out gets asserted.
always
    begin
#200      overheat_in         =      1;
#100      pushbutton_in       =      1;
#100      pushbutton_in       =      0;
#200      overheat_in         =      0;
#100       $finish;
    end

endmodule
```

Sam invokes her favorite simulation tool and examines the output waveforms to make sure the output is logically correct. The output waveform looks like Figure 1-1 and appears okay. Generally Sam will write and run an automated test-fixture program (as described in Chapter 5), but the design is simple and the boss has ordered her to quit being such a fussbudget and get on with it.

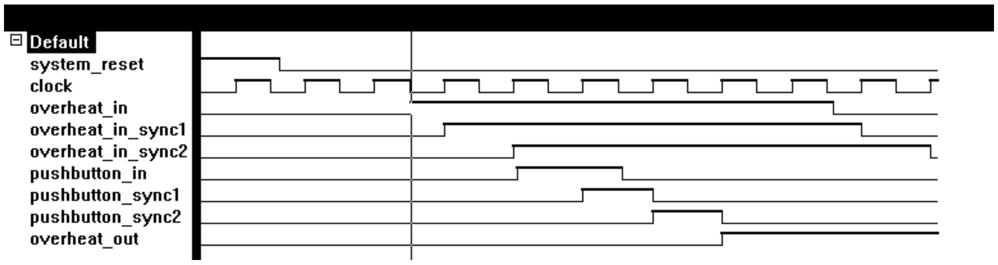

Figure 1-1 Overheat Detector Design Output Waveforms

Sam assigns input/output pins and defines timing constraints for her design. She knows that the system does not have to run fast, so she selects the lowest available crystal oscillator to drive the clock input. This gives the lowest current consumption to maximize the life of the battery. Sam submits the design to her FPGA compiler and gets a report back that tells her that the design fits into the device she chose and that timing constraints are met. From experience, she knows that a design running this slowly will not have temperature or RFI emission problems. She checks the design into the revision control system, sends an email to her boss to tell her the job is complete, and takes the rest of the day off to go rollerblading.

This probably seems like a lot of work to complete a job that consists of six flipflops, but Sam was lucky. The design fit into the device she chose, the design ran at the right speed, the design did not have temperature/EMI/RFI problems, the specifications didn’t change halfway through the design, the software tools and her workstation didn’t crash, and she avoided the 1001 other hazards that exist in the Real World.

ENGINEERING SCHEDULE: too often, a management tool for browbeating an engineer into working free overtime. Engineers, even when they should really know better, are generally too optimistic when creating schedules, thus, they are almost always late.

We have to be mature about this subject: without a deadline, nothing would ever get finished. Still, most jobs should be completed with little overtime.

Some problems can be avoided by doing thorough design work up front. Sam was careful not to start coding until she completely understood the requirements of the design.

GIGO (GARBAGE IN, GARBAGE OUT)

There is a great temptation to start coding before the product is well understood. After all, to an engineer, coding is fun and planning is not.

I don't care how much fun the job is, don't start coding the design until you know what the end result is supposed to be.

This book emphasizes design approaches that minimize problems and unpleasant surprises.

SYNTHESIZABLE VERILOG ELEMENTS

Verilog was designed as a simulation language and many of its elements do not translate to hardware. Verilog is a large and complete simulation language. Only about 10% of it is synthesizable. This chapter covers the fundamental properties of the 10% that the FPGA designer needs.

Exactly which Verilog elements are considered synthesizable is a design problem faced by the synthesis vendor. Generally, an "unofficial" subset of the Verilog language elements will be supported by all vendors, but the current Verilog specification does not contain any minimum list synthesizable language elements. An IEEE working group is writing a specification called IEEE Std 1364.1 RTL Synthesis Subset to define a minimum subset of synthesizable Verilog language elements. Whether this specification is ever released – and, once released, is embraced by users and synthesis tool vendors - remains to be seen at this writing.

Verilog looks similar to the C programming language, but keep in mind that C defines sequential processes (after all, only one line of code can be executed by a processor at a time), whereas Verilog can define both sequential and parallel processes. Listing 1-5 presents some sample code with common synthesizable Verilog elements.

Listing 1-5 Example Verilog Program

```
module hello (in1, in2, in3, out1, out2, clk, rst, bidir_signal,
output_enable);// See note 1.
/* See note 2.
Comments that span multiple lines can be identified like this.
*/
input      in1, in2, in3, clk, rst, output_enable;// See note 3.
output     out1, out2;
```

```
inout      bidir_signal;
reg        out2;                    // See note 4
wire       out1;

assign out1         = in1 & in2;    // See note 5.
assign bidir_signal = output_enable ? out2 : 1'bz; // See note 6.

always @ (posedge clk or posedge rst)       // See note 7.
    begin                                   // See note 8.
    if (rst) out2      <=    1'b0;          // See note 9.
    else out2          <=    (in3 & bidir_signal);
    end
endmodule
```

Note 1: The first element of a module is the module name. Modules are the building blocks of a Verilog design. In this book, the module name will be the same as the file name (with a .v extension added) and each file will contain a single module. This is not required but helps keep the design structure intelligible.

The port list follows the module/file name. This list contains the signals that connect this module to other modules and to the outside world. Signals used in the module that are not in the port list are local to the module and will not be connected to other modules. Note the use of a semicolon as a separator to isolate Verilog elements. One confusing aspect of Verilog is that not all lines end with a semicolon, particularly the compiler instructions (always statements, if statements, case statements, etc.). It takes the Verilog newbie some time to get comfortable with Verilog syntax.

Note 2: Comments follow double forward slashes or can be enclosed within a /* Comment here */ pair. The latter type of comment delimiting can't be nested. The detection of a /* following another /* will be flagged as an error.

Note 3: The port direction list follows the module port list. This list defines whether the signals are inputs, outputs, or inouts (bidirectional) ports of the module. All port list signals are wires. A wire is simply a net similar to an interconnection on a printed circuit card.

Note 4: Signals are either wires (interconnects similar to traces and pads on a circuit board) or registers (a signal storage element like a latch or a flipflop). Wires can be driven by a register or by combinational assignments. It is illegal to connect two registers together inside a module. Verilog assumes that a signal, unless otherwise defined in the code, is a one-bit-wide wire. This can be a problem, the synthesis tool will not test vector width. This is one good reason for using a Verilog Lint tool.

Note 5: The assign statement is a continuous (combinational) logic assignment.

Note 6: The assignment of the bidir_signal uses a conditional assignment; if output_enable is true, bidir_signal is assigned the value of out2, otherwise it's assigned the tri-state value z.

Note 7: Always blocks are sequential blocks. The signal list following the @ and inside the parenthesis is called the event sensitivity list, and the synthesis tool will extract block control signals from this list. The requirement of a sensitivity list comes from Verilog's simulation heritage. The simulator keeps a list of monitored signals to reduce the complexity of the simulation model; the logic is evaluated only when signals on the sensitivity list change. This allows simulation time to pass quickly when signals are not changing. This list doesn't mean much to the synthesis tool, except, that by convention, when certain signals are extracted for control, these input signals must appear on the sensitivity list. The compiler will issue a warning if the sensitivity list is not complete. These warnings should be resolved to assure that the synthesis result matches simulation.

The sensitivity list can be a list of signals (in which case, any change on any listed signal is detected and acted upon), posedge (rising-edge triggered), or negedge (falling-edge triggered). Posedge and negedge triggers can be mixed, but if posedge or negedge is used for one control, posedge or negedge must be used for ALL controls for this block.

Note 8: The begin/end command isolates code fragments. If the code can be expressed using a single semicolon, the begin/end pair is optional.

Note 9: We're using nonblocking assignments (<=) in the always block. If blocking assignments are used, the order of the instructions may cause unwanted latches to be synthesized so that a value can be held while earlier variables are updated. Generally, the designer wants all elements in the sequential (always) block updated simultaneously, hence the use of the nonblocking assignment, which emulates the clock-to-Q delay. The clock-to-Q delay assures that cascaded flipflops (like a shift register) operate as expected. They are called nonblocking because updating an earlier variable will not block the updating of a later variable.

The rst input, when coded in this manner (i.e., a nonsynchronous signal used in a synchronous module), is interpreted as asynchronous reset. This is not Verilog requirement per IEEE Std 1364 but is an accepted convention.

Verilog language elements are case sensitive (X and x are not equivalent, for example). Like the C programming language, Verilog is tolerant of white space. The designer uses white space to assist legibility. It's legal to combine lines as so:

```
  a = b&c; d = e&f; g = h | i; j = k^m; n = o&p;
```

but designers who write hard-to-read code like this are subject to the loss of their free sodas.

PORTABLE VERILOG CODE: It is desirable to write code that can be compiled by any vendor's compiler and can be implemented in any hardware technology with identical results. Unfortunately, to write high-performance (where the design runs at high speed) and efficient (where the design uses minimum hardware resources by targeting architecture-specific features) code, the designer often must use architecture- and compiler-specific commands and constructs. Portability is often not a practical or achievable design requirement. It's a great goal, even if we never reach it.

We're not going to cover operator precedence. If you have a required precedence, then use parenthesis to be explicit about that precedence. The reader should be able to read the precedence in the source code, not be forced to memorize or look up the built-in language precedence(s). Don't create complicated structures; use the simplest and clearest coding style possible. Listings 1-6 and 1-7 illustrate equivalent coding structures with implicit and explicit 'don't-cares'.

Listing 1-6 Casex (Implicit Don't Care) Code Fragment

```
// Indexing example with implicit 'don't cares'.
reg        [7:0] test_vector;
Casex (test_vector)
8'bxxxx0001:
     begin
// Insert code here.
// This coding style results in a parallel case structure (MUX).
     end
endcase
```

Listing 1-7 Explicit Don't Care Code Fragment

```
// Indexing example with explicit 'don't cares'.
reg        [7:0] test_vector;
if (test_vector[3:0] == 4'b0001)
     begin
     // Insert code here.
     // This coding style results in priority encoded logic.
     end
```

One feature of Verilog the designer must conquer is whether a priority-encoded (deep and slow) structure or a MUX (wide and fast) structure is desired. Nested if-then statements tend to create priority-encoded logic. Case statements tend to create MUX logic elements. There will be more discussion of this topic later.

Do not assume a Verilog register is a flipflop of some type. In Verilog, a register is simply a memory storage element. This is one of the first of the features (or quirks) the Verilog designer grapples with. A register might synthesize to a flipflop (which is a digital construct) or a latch (which is an analog construct), a wire, or might be absorbed during optimization. Verilog assumes that a variable not explicitly changed should hold its value. This is a handy feature (compared to Altera's AHDL, which assumes that a variable not mentioned gets cleared). Verilog, with merciless glee, will instantiate latches to hold a variable's state. The designer must structure the code so that the intended hardware construct is synthesized and must be constantly alert to the possibility that the latches may be synthesized. Verilog does not include instructions that *require* the synthesizer to use a certain construct. By using conventions defined by the synthesis vendor, and making sure all input conditions are completely defined, the proper *interpretation* will be made by the synthesizer.

VERILOG HIERARCHY

A Verilog design consists of a top-level module and one or many lower-level modules. The top-level module can be instantiated by a simulation module that applies test stimulus to the device pins. The top-level device module generally contains the list of ports that connect to the outside world (device pins) and interconnect between lower-level modules and multiplexing logic for control of bidirectional I/O pins or tristate device pins. The exact way the design is structured depends on designer preference.

Module instances are defined as follows:

```
module_name instance_name (port list);
```

For example, the code in Listing 1-8 creates four instances of assorted primitive gates and the post-synthesis schematic for this design is shown in Figure 1-2.

Listing 1-8 Structural Example

```
module gates (in1,in2,in3,out4);
input     in1, in2, in3;
output    out4;
wire      in1, in2, in3, out1, out2, out3, out4;

and u1 (out1, in1, in2); // Structural (schematic-like)
or u2 (out2, out1, in3); //  constructs.
xor u3 (out3, out1, out2);
not u4 (out4, out3);
endmodule
```

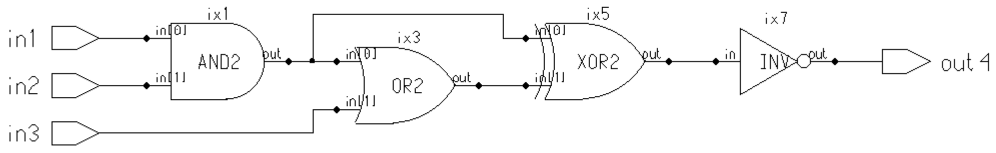

Figure 1-2 Gates Example Schematic

This example uses positional assignment. Signals are connected in the same order that they are listed in the instantiated module port list(s). Generally, the designer will cut and paste the port list to assure they are identical. A requirement for a primitive port listing is that the output(s) occur first on the port list followed by the input(s).

The module port list can also use named assignments (exception: primitives require positional assignment), in which case the order of the signals in the port list is arbitrary. For named assignments, the format is .lower-level signal name (higher-level module signal name). The module of Listing 1-9 includes examples of both named and positional assignments.

Listing 1-9 Named and Positional Assignment Example

```
module and_top;
wire       test_in1, test_in2, test_in3;
wire       test_out1, test_out2;

// Named assignment where the port order doesn't matter.
user_and u1 (.out1(test_out1), .in1(test_in1), .in2(test_in2));

// Positional assignment.
user_and u2 (test_out2, test_in2), (test_in3));
endmodule

module user_and (out1, in1, in2);
input      in1, in2;
output     out1;

assign     out1  =     (in1 & in2);
endmodule
```

BUILT-IN LOGIC PRIMITIVES

Tables 1-1 through 1-12 describe Verilog two-input functions. The input combinations are read down and across. Verilog primitives are not limited to two inputs, and the logic for primitives with more inputs can be extrapolated from these tables.

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Table 1-1 AND Gate Logic

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	z
x	1	x	x	z
z	1	x	x	z

Table 1-2 NAND Gate Logic

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Table 1-3 OR Gate Logic

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

Table 1-4 NOR Gate Logic

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Table 1-5 XOR Gate Logic

xnor	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 1-6 XNOR (Equivalence) Gate Logic

input	output
0	0
1	1
x	x
z	x

Table 1-7 buf (buffer) Gate Logic

input	output
0	1
1	0
x	x
z	x

Table 1-8 not (inverting buffer) Gate Logic

bufif0	control = 0	control = 1	control = x	control = z
data = 0	0	z	x	x
data = 1	1	z	x	x
data = x	x	z	x	x
data = z	x	z	x	x

Table 1-9 bufif0 (tristate buffer, low enable) Gate Logic

bufif0	control = 0	control = 1	control = x	control = z
data = 0	z	0	0 or z	0 or z
data = 1	z	1	1 or z	1 or z
data = x	z	x	x	x
data = z	z	x	x	x

Table 1-10 bufif1 (tristate buffer, high enable) Gate Logic

notif0	control = 0	control = 1	control = x	control = z
data = 0	1	z	1 or z	1 or z
data = 1	0	z	0 or z	0 or z
data = x	x	z	x	x
data = z	x	z	x	x

Table 1-11 notif0 (tristate inverting buffer, low enable) Gate Logic

notif1	control = 0	control = 1	control = x	control = z
data = 0	z	1	1 or z	1 or z
Data = 1	z	0	0 or z	x or z
Data = x	z	x	x	x
Data = z	z	x	x	x

Table 1-12 notif1 (tristate inverting buffer, high enable) Gate Logic

The code fragment in Listing 1-10 illustrates the use of these buffers and Figure 1-3 is the schematic extracted from the synthesized logic.

Listing 1-10 Example of Instantiating Structural Gates

```
module struct1 (out1, out2, out3, out4, in1, in2, in3, in4, in5,
in6, buf_control);
output     out1, out2, out3, out4;
input      in1, in2, in3, in4, in5, in6, buf_control;
bufif0 buf1(out1, in1, buf_control);
and and1(out2, in2, in3);
nor nor1(out3, in4, in5);
not not1(out4, in6);
endmodule
```

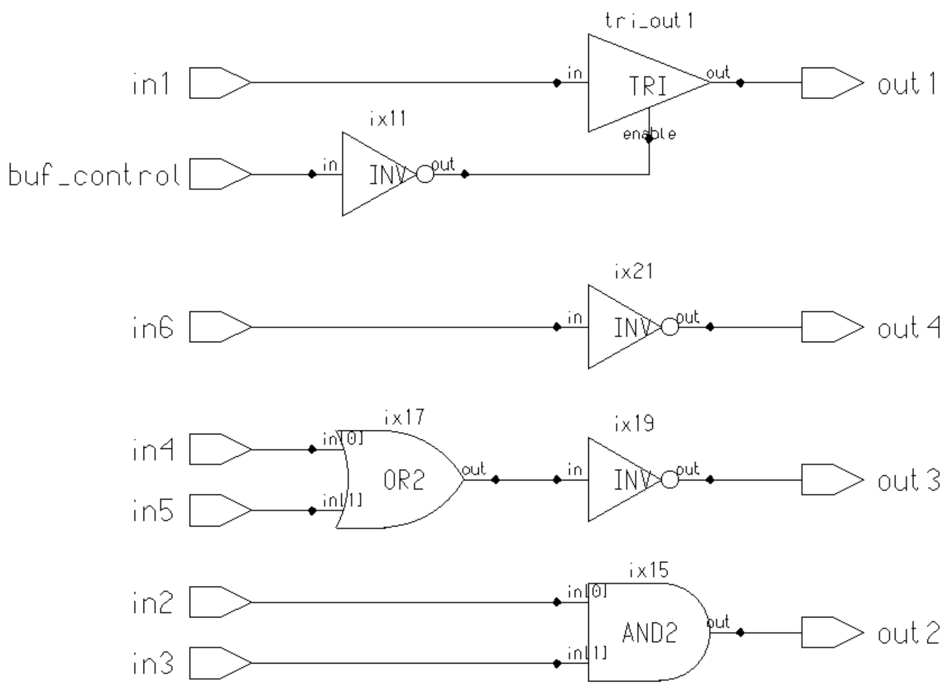

Figure 1-3 Schematic of Structural Gates

LATCHES AND FLIPFLOPS

Technically, a flipflop is defined as a bistable multivibrator. Not a very helpful definition, is it? A multivibrator is an analog circuit with two or more outputs, where, if one output is on, the other(s) will be off. Bistable means an output is binary or digital and has two output states: high or low. We will extend the output states to include tristate (z).

There are various flavors of flipflops, but generally we will be discussing the clocked D flipflop in which the output follows the D input after a clock edge. Table 1-13 shows the function table of a common edge-triggered D flipflop. Note that Table 1-13 Set and Reset inputs are active-low.

/Set	/Reset	Clock	Data	Q	/Q	
0	1	x	x	1	0	
1	0	x	x	0	1	
0	0	x	x	1	1	Note 1
1	1	r	1	1	0	
1	1	r	0	0	1	
1	1	0	x	n	n	
1	1	1	x	n	n	

Note 1: This condition is not stable and is illegal. The problem is, if the /Set and /Reset inputs are removed simultaneously, the output state will be unknown.
x = don't care (doesn't matter).
r = rising edge of clock signal.
n = no change, previous state is held.

Table 1-13 Logic description of a 7474-style D flipflop

The typical FPGA logic element design allows the use of either an asynchronous Set or Reset, but not both together, so we won't have to worry about the illegal input condition where both are asserted. **This book is going to strongly emphasize synchronous design techniques, so we discourage any connection to a flipflop asynchronous Set or Reset input except for power-up initialization control.** Even in this case, a synchronous Set/Reset might be more appropriate.

A latch is more of an analog function. It's helpful to bear in mind that all the underlying circuits that make up our digital logic are analog! There is no magic flipflop element. Flipflops are made with transistors and positive feedback: latches.

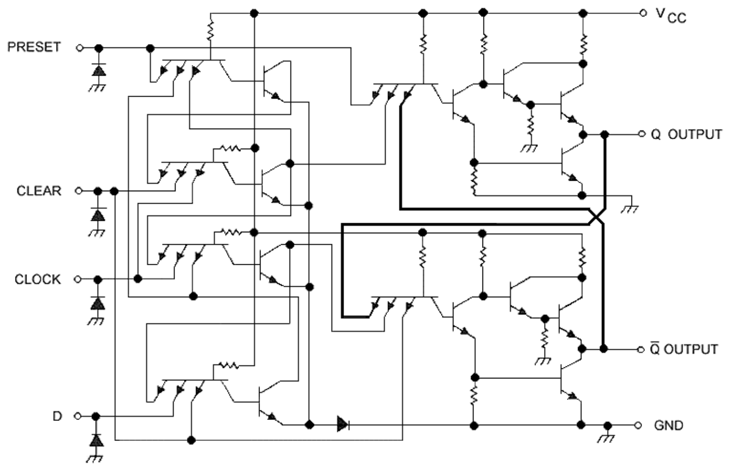

Figure 1-4 Schematic of a Typical CMOS D Flipflop Implementation

Even if you're the kind of person whose eyes glaze over when you see transistors on a schematic, you should still notice two things about Figure 1-4. The first thing is that this D flipflop is made with linear devices, i.e., transistors. If you can always keep the idea in the back of your head that all digital circuits are built from analog elements that have gain, impedance, offsets, leakages, and other analog nasties, then you are on the road to being an excellent digital designer. The second thing to notice is feedback (see highlighted signals) from the Q and /Q outputs back into the circuit. Feedback is what causes the flipflop to hold its state.

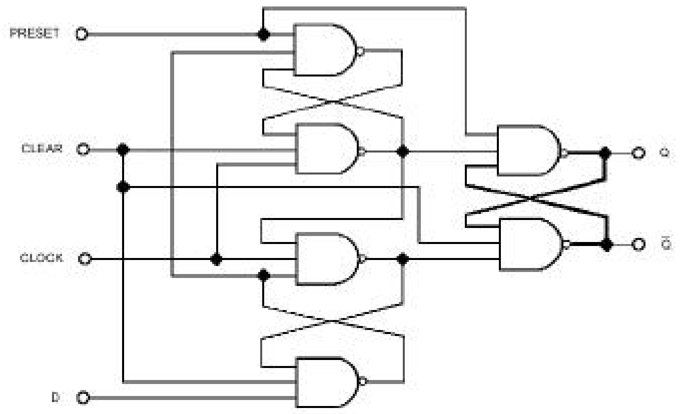

Figure 1-5 Schematic of a Typical CMOS D Flipflop Implementation (Gates)

If you are more comfortable with gates, a different view of the same D flipflop is shown in Figure 1-5. This is a higher level of abstraction; the transistors and resistors are hidden. Those pesky transistors are still there! Again, note highlighted feedback path.

Listing 1-11 shows a Verilog version of a latch, and Figure 1-6 shows the schematic extracted from this Verilog design. The underlying circuit that implements RS Latch (LATRS) is a circuit functionally similar to Figure 1-5. It's not a digital circuit!

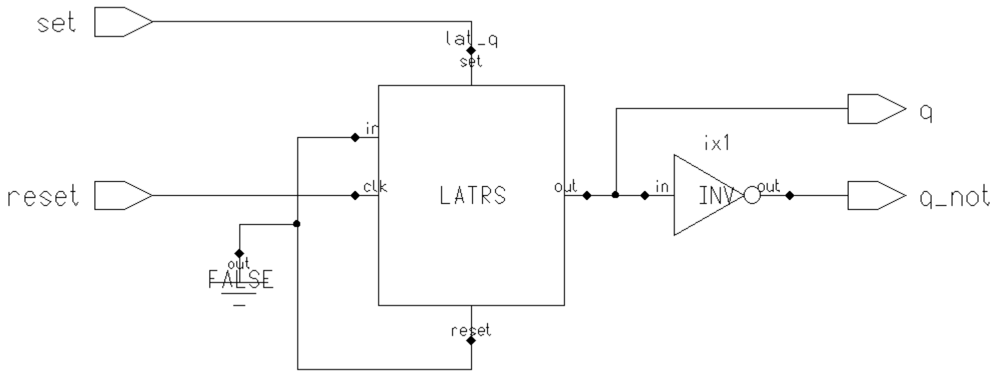

Figure 1-6 Schematic of Latch Flipflop Implementation

Listing 1-11 Latch Verilog Code

```
// Your Basic Latch.
// This is a bad coding style: do not create latches this way!

module latch(q, q_not, set, reset);
output     q, q_not;
input      set, reset;
reg        q;

wire       set, reset;

assign     q_not  =     ~q;

    always @ (set or reset)
    begin
           if (set)
           q     =     1;
           else if (reset)
           q     =     0;
    end
endmodule
```

The latch uses feedback to hold a state: this feedback is implied in Listing 1-11 by not defining q for all combinations of input conditions. For undefined inputs, q will hold its previous state. The logic that determines a latch output state may include a clock signal but typically does not and is therefore a level-sensitive rather than an edge-triggered construct.

Listing 1-12 Verilog Code That Creates a Latch

```
module lev_lat(test_in1, enable_input, test_out1);
input      test_in1, enable_input;
output     test_out1;
reg        test_out1;

always @ (test_in1 or enable_input)
if (enable_input) begin
    test_out1    <=    test_in1;
    end
endmodule
```

In the example of Listing 1-12, test_out1 will change only while enable_input is high, then test_out1 will follow test_in1. This will synthesize to a combinational latch as illustrated in Figure 1-7. We'll discourage this type of coding style unless the latch is driven by a synchronous circuit and drives a synchronous circuit, resulting in a pseudosynchronous design.

Is a latch a good design construct? That depends on the designer's intent. If the designer intended to create a latch construct, then a synthesized latch is good. If the designer did not intend to create a latch construct (which Verilog is very inclined to create), then a latch is bad. In general, we will scrutinize all synthesized latches suspiciously, because they are, at best, pseudosynchronous constructs.

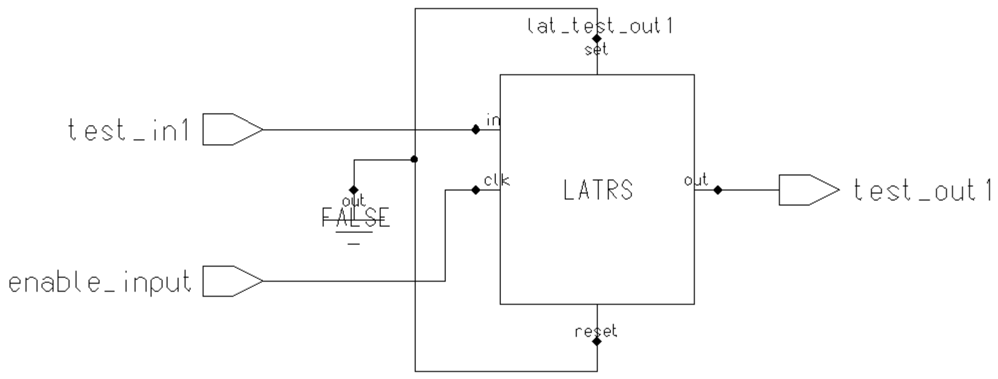

Figure 1-7 Latch Circuit Schematic (Reset-Set Latch)

A better design infers a clocked flipflop structure, as in Listing 1-13, with the respective schematic shown in Figure 1-8.

Listing 1-13 Cascaded Flipflops with Synchronous Reset

```
module edge_lat (clk, rst, test_in1, enable_input, test_out2);
input      clk, rst, test_in1, enable_input;
reg        test_out1, test_out2;
output     test_out2;

always @ (posedge clk or posedge rst)
begin
    if (rst) test_out1  <=    0;
    else if (enable_input) begin
           test_out2    <=    test_out1;
           test_out1    <=    test_in1;
    end
end
endmodule
```

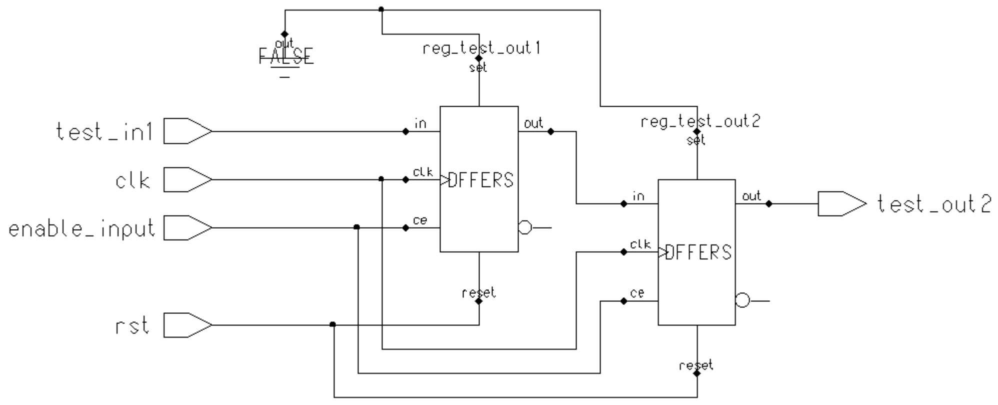

Figure 1-8 Schematic for Cascaded Flipflops with Synchronous Reset

Listing 1-13 demonstrates a flipflop with synchronous reset where the reset input is evaluated only on clock edges. If the target hardware does not support a synchronous reset, logic will be added to set the D input low when reset is asserted as shown in Figure 1-9. Listing 1-14 illustrates a flipflop with asynchronous reset where the **rst** signal is "evaluated" on a continuous basis. Notice that the dedicated global set/reset (GSR) resource of the flipflops are not used. It would be much more efficient to synthesize a synchronous reset signal and connect it to the GSR. This type of assignment is covered in Chapter 5.

Listing 1-14 Verilog Flipflop with Asynchronous Reset

```
module edgetrig (clk, rst, test_in1, enable_input, test_out2);
input      clk, rst, test_in1, enable_input;
reg        test_out1, test_out1;
output     test_out2;

always @ (posedge clk)
begin
    if (rst)
          test_out1    <=    0;
    else if (enable_input)   begin
          test_out2    <=    test_out1;
          test_out1    <=    test_in1;
    end
end
endmodule
```

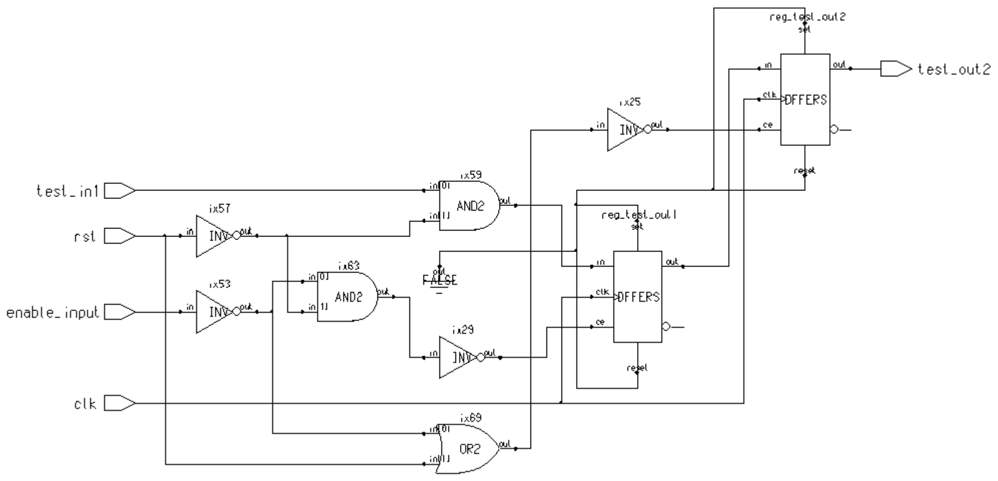

Figure 1-9 Schematic for Cascaded Flipflops with Synchronous Reset

BLOCKING AND NONBLOCKING ASSIGNMENTS

So far, we've used only nonblocking assignments (<=). A blocking assignment, when the variable is defined outside the **always** statement where it is used, holds off future assignments until the previous assignment is complete. How can synthesized hardware hold off an assignment? By storing an old value in a latch, that's how. This means that blocking assignments are order sensitive; they are executed in the begin/end sequential block in the order in which they are encountered by the compiler (top to bottom).

Listing 1-15 Blocking Statement Example 1

```
/* The blocking statement of the first blocking assignment must
be completed before any later assignments will be performed. In
this example, two sets of flipflops will be created (see Figure
1-10) because an intermediate value is required to create
data_out. */

module blocking(clock, reset, data_in, data_out);
input      clock, reset;
input      data_in;
```

```
reg        data_temp;
output     data_out;
reg        data_out;

    always @ (posedge clock or posedge reset)
    if (reset)
    begin
        data_out       =     0;
        data_temp      =     0;
    end
    else
    begin
        data_out       =  data_temp;
        data_temp      =  data_in;
    end
endmodule
```

The synthesized logic for Listing 1-15, shown in Figure 1-10, illustrates the blocking assignment of data_temp and data_out: a flipflop is synthesized to create the intermediate (pipelined) data_temp variable.

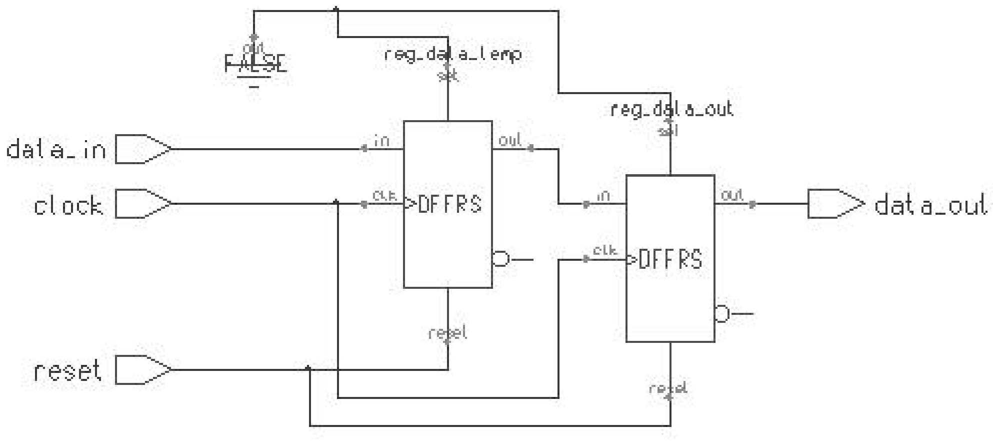

Figure 1-11 Blocking Statement Example 1

In Listing 1-16, the blocking statements are reversed. Notice how the resulting logic, as illustrated in Figure 1-11, is different from the logic of Figure 1-10.

Listing 1-16 Blocking Statement Example 2

```
/* The blocking statements are reversed, making the data_temp
variable redundant, so data_temp gets optimized out. One set of
flipflops is created because the intermediate value is 'blocked'
and not needed to create data_out. */

module block2(clock, reset, data_in, data_out);
    input  clock, reset, data_in;
    reg    data_temp;
    output data_out;
    reg    data_out;

    always @ (posedge clock or posedge reset)
    if (reset)   begin
        data_out       =     0;
        data_temp      =     0;
    end
    else   begin
        data_temp      =  data_in;  // Switch order.
        data_out       =  data_temp;
    end
endmodule
```

In a set of blocking assignments that appear in the same **always** block, the order in which the statements are evaluated is significant. The use of nonblocking assignments avoids order sensitivity and tends to create flipflops: this is generally what the designer intends.

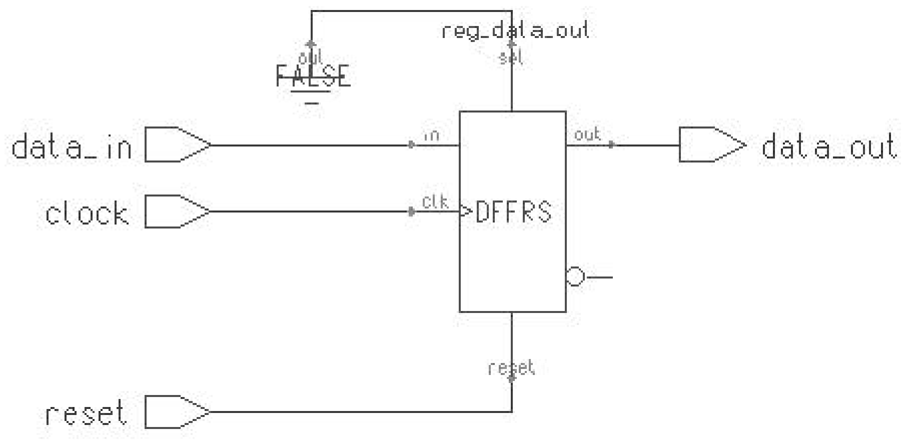

Figure 1-11 Blocking Assignment Example 2

If we replace the blocking assignments with nonblocking assignments, the order of the sequential instructions no longer matters. All right-hand values are evaluated at the positive edge of the clock, and all assignments are made at the same time. The synthesized logic for Listing 1-17, shown in Figure 1-12, illustrates the nonblocking assignments of data_temp and data_out and resulting synthesized design which is equivalent to the logic of Listing 1-16.

Listing 1-17 Non-Blocking Assignment Example

```
// Nonblocking Logic Example
// The order of the nonblocking assignments is not significant.

module nonblock(clock, reset, data_in, data_out);
input      clock, reset;
input      data_in;
reg        data_temp;
output     data_out;
reg        data_out;

    always @ (posedge clock or posedge reset)
    if (reset)
    begin
        data_out       <=    0;
        data_temp      <=    0;
    end
    else
    begin
        data_out       <=    data_temp;
        data_temp      <=    data_in;
    end
endmodule
```

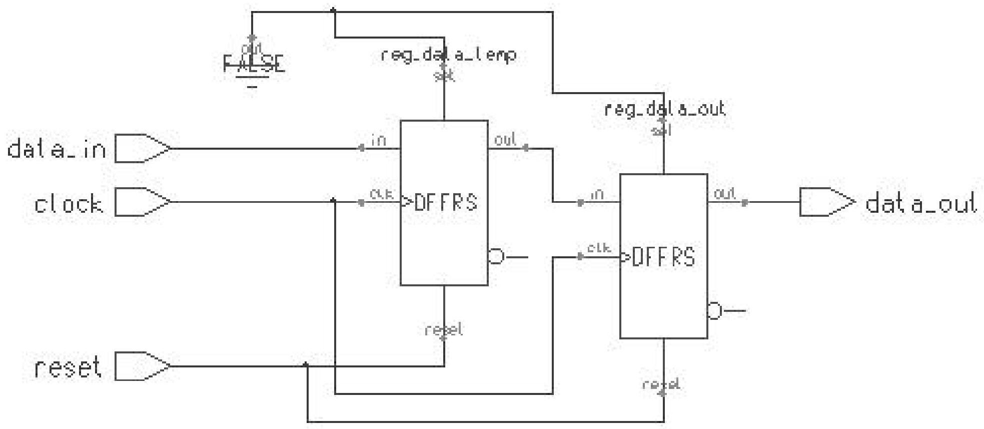

Figure 1-12 Nonblocking Assignment Example

MISCELLANEOUS VERILOG SYNTAX ITEMS

Numbers

Unless defined otherwise by the designer, a Verilog number is 32 bits wide. The format of a Verilog integer is **size'base value**. The **'** is the apostrophe or tick, not to be confused with **`** (accent grave or back tick) which is used to identify constants or text substitutions. Both tick and back tick are used in Verilog, which will frustrate a newbie. Underscores are legal in a number to aid in readability. All numbers are padded to the left with zeros, x's, or z's (if the leftmost defined value is x or z) as necessary. If the number is unsized, the assumed size is just large enough to hold the defined value when the value gets used for comparison or assignment. X or x is undefined, Z or z is high impedance. Verilog allows the use of ? in place of z. Numbers without an explicit base are assumed to be decimal.

Number examples:

```
1'b0         // A single bit, zero value.
'b0          // 32 bits, all zeros.
```

```
32'b0        // 32 bits, all zeros,
             //  0000_0000_0000_0000_0000_0000_0000_0000).
4'ha         // A 4-bit hex number (1010).
5'h5         // A 5-bit hex number (00101).
4'hz         // zzzz.
4'h?ZZ?      // zzzz; ? is an alternate form of z.
4'bx         // xxxx.
9            // A 32-bit number (it’s padded to the left
             //  with 28 zeroes).
a            // An illegal number.
```

Verilog is a loosely typed language. For example, it will accept what looks like an 8-bit value like 4'hab without complaint (the number will be recognized as 1011 or b and the upper nibble will be ignored). The use of a Lint program like Verilint will flag problems like this. Otherwise, the Verilog designer must stay alert to guard against such errors.

Forms of Negation

! is logical negation; the result is a single bit value, true (1) or false (0). ~ (tilde) is bitwise negation. We can use a ! (sometimes called a bang) to invert a single bit value, and the result is the same as using a ~ (tilde), but this is a bad habit! As soon as someone comes in and changes the single bit to a multibit vector, the two operators are no longer equivalent, and this can be a difficult problem to track down (see Listing 1-18).

Listing 1-18 Negation example

```
module negation (clk, resetn);
input        clk, resetn;
reg [3:0]    c, d, e;

always @ (posedge clk or negedge resetn)
begin
      if (~resetn)    // Active low asynchronous reset.
      begin

      c      <=     5; // Bad form to async set a value like
                       //  this. This is called a magic
                       //  number and should be a parameter.
      d      <=     0;
      e      <=     0;
      end
      else   begin
      d      <=     !c;    // d gets assigned value of 0;
      e      <=     ~c;    // e gets assigned value of 1010.
      end
end
endmodule
```

Forms of AND/OR

& is the symbol for the AND operation. & is a bitwise AND, && is a logical (true/false) AND. As illustrated in Listing 1-19, these two forms are not functionally equivalent.

Listing 1-19 Logical and Bitwise AND Examples

```
a    =      4’b1000 &    4’b0001;     // a = 4’b0000;
b    =      4’b1000 &&   4’b0001;     // b = 1’b0.
```

| (pipe) is the symbol for the OR operation where | is a bitwise OR and || is a logical OR. As illustrated in Listing 1-20, these two forms are not functionally equivalent.

Listing 1-20 Logical and Bitwise OR Examples

```
a    =      4’b1000 |    4’b0001;     // a = 4’b1001;
b    =      4’b1000 ||   4’b0001;     // b = 1’b1.
```

Listing 1-21 AND/OR Examples

```
module and_or (clk, resetn, and_test, or_test);
    input        clk, resetn, and_test, or_test;
    reg          a;
    reg [3:0]    b;
    reg [3:0]    c;
    reg [3:0]    d;
    reg [3:0]    e;
    reg [3:0]    g;

    always @ (posedge clk or negedge resetn)
    begin
          if (~resetn) // Active low asynchronous reset.
          begin
          a      <=    0;
          b      <=    4’d4;  // Bad form to async set values
                              //  like this, should be a
                              //  parameter.
          c      <=    4’d5;
          d      <=    0;
          e      <=    0;
          g      <=    0;
          end
```

```
        else if (and_test)
        begin

d       <=    (c && !a);   // d gets assigned value of 0.
e       <=    (c & !a);    // e gets assigned value of
                           //  1010.
g       <=    (b & c);     // g gets assigned the value
                           //  0100.
        end
else if (or_test == 1)     // Equivalent to simply (or_test).
        begin
e       <=    (c | !a);    // e gets assigned value of all
                           //  1's (1111).
g       <=    (b | c);     // g gets assigned the value 0101.
        end
        else
        begin
d       <=    0;     // Assign default values to avoid
                     //  unwanted latches.
e       <=    0;
g       <=    0;
        end

end
endmodule
```

In Listing 1-21, the final **else** condition bears some comment. We did not cover all input conditions in the logic above the final **else** condition. For example, what output do we want if neither and_test or or_test is asserted? Without the final **else** defined, Verilog interprets a change from a defined condition to an undefined condition as a hold condition (if outputs are not commanded, the last value gets held). This causes latches to be created. Generally, this is not what the designer intends; thus we need to make sure that all conditions are defined.

Equality Operators

== === are logical operators, the result is either true or false except that the == (called logical equality) version will have an unknown (x) result if any of the compared bits are x or z. The === (called case equality) version looks for exact match of bits including x's and z's and returns only a true or false. Prepending a ! (bang) means "is not equal." In the equality examples of Listing 1-22, there are several if statements that will evaluate to true. As the block is examined from top to bottom, only the first true condition will be accepted. The later ones will not be evaluated. This is called priority encoding, and, like instantiating latches, Verilog has a natural tendency to use this structure. It can result in many levels of cascaded logic! Pay close attention. The alternative option is more of a MUXstyle of structure where inputs are evaluated in parallel, which may be what you intend. We'll talk more about this later.

Listing 1-22 Equality Examples

```
    module eq_test (clk, resetn, and_test, or_test);
    input        clk, resetn, and_test, or_test;
    reg          result;
    reg [3:0]    b;
    reg [3:0]    c;
    reg [3:0]    d;
    reg [3:0]    e;
    reg [4:0]    g;
    reg [3:0]    h, i, j;

    always @ (posedge clk or negedge resetn)
    begin
          if (~resetn) // Active low asynchronous reset.
          begin

          result <=     0;     // We'll use this register to
                               //  mirror the equality result.
          b      <=     4'b1x00; // Bad form to async set values
                                 //  like this; should be a
                                 //  parameter.
          c      <=     4'b1z00;
          d      <=     4'b1000;
          e      <=     4'b1001;
          g      <=     4'b01001;
          h      <=     4'b1z00;
          i      <=     4'b0110;
          j      <=     4'b011x;
          end

// The following test fails.

    else if ((b == d) == 1)
          result <=     1'bx;

    else if (b == d)    // This test is the same as previous
                        //  line. Fails.
          result <=     1'bx;

    else if ((b == d) == 0)   // This test fails because of the
                              //  x value in b.
          result <=     1'bx;

    else if ((b != d) == 1)   // This test is the same as in the
                              //  previous line. Fails.
          result <=     1'b0;
```

```
else if ((b == d) == 1'bx)// This test passes because the
                          //  b value is x.
      result <=    1'b1;  // All following true conditions
                          //  will be ignored.

else if (c === d)         // This test fails.
      result <=    1'b0;

else if (e == g)        // This test passes because e is
                        //  padded with 0's to become equal
                        //  in size to g.
      result <=    1'b0;  // Be careful when variables sizes
                          //  don't match.

else if (b == c)        // This test fails (returns false).
      result <=    1'b0;

else if (b != c)        // This test passes (returns true).
      result <=    1'b1;

else if ( d == e)       // This test fails (returns false).
      result <=    1'b0;

else if (b !== c)       // This test passes (returns true).
      result <=    1'b1;

else if (c == h)        // This test fails (returns x).
      result <=    1'bx;

else if (c===h)         // This test passes (returns true).
      result <=    1'b1;

else if (e == !i)   // This test passes (returns true).
      result <=    1'b1;

else if (e != j)    // This test fails (returns x).
                    //  An inverted x (unknown) is
                    //  still an unknown.
      result <=    1'bx;

end
endmodule
```

The designer can choose between the following **if** statement forms:

if (~resetn) ...

if (resetn == 1`b0)

Both are equivalent. Which is easy to read and easier to understand? That's a matter of opinion. Note the use of an 'n' suffix to indicate an active low (asserted when low or low-true are other ways to describe this) signal. There are various ways of identifying active low signals- for example, reset_not, resetl, or reset*, or resetN. It helps to identify the assertion sense as part of the label; the main thing is to be consistent when selecting labels.

Other equalities are supported, including greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

Shift Operators

>> n and << n identify right-shift (divide by 2n) and left-shift (multiply by 2n) operations. This operation will fill left and right values with zeros as necessary to fill the register. Operating on a value which contains an x or a z gives an x result in all bit positions. Some examples of using the shift operators are presented in Listing 1-23.

Listing 1-23 Shift Operator Examples

```
module shifter (clk, resetn, shift_right_test, shift_left_test);
input           clk, resetn;
input           shift_right_test;
input           shift_left_test;
reg [3:0]  a;
reg [3:0]  b;
reg [3:0]  c;
reg        d;
reg [3:0]  e, f;

    always @ (posedge clk or negedge resetn)
    begin
          if (~resetn) // Active low asynchronous reset.
          begin

          a      <=     'b1001;
          b      <=     0;    // It’s bad form to async set
                              //  values like this.
          c      <=     0;
          d      <=     0;
          e      <=     'bx000;
          end

          else if (shift_right_test)
          begin

          c      <=     a >> 2;       // c gets assigned value of
                                      //  0010.
          d      <=     a >> 5;       // Regardless of the value
                                      //  of a, d will always get
```

```
                                     //  assigned to 0.
                                     // Verilog will not complain
                                     //  about this; use caution.
             f      <=    e >> 1;    // Result is xxxx because of
                                     //  x in e.
             end

             else if (shift_left_test)
             begin

             c      <=    a << 2;    // c gets assigned value of
                                     //  0100.
             d      <=    a << 5;    // Regardless of the value
                                     //  of a, d will always get
                                     //  assigned to 0.
                                     // Verilog will not complain
                                     //  about this; use caution.
             f      <=    e << 1;    // Result is xxxx because of
                                     //  of x in e.
             end

             else
             begin
             d      <=    0;     // Assign values default to avoid
                                 //  unwanted latches.
             e      <=    0;
             f      <=    0;
             end

      end
      endmodule
```

Conditional Operator

A shorthand method of doing a conditional uses a ternary form (which means arranged in order by threes).

```
    output_assignment <= expression ? true_assignment :
  false_assignment;
```

This is a common way of defining a MUX. If the expression being evaluated resolves to x or z, the output_bus is evaluated bit-by-bit, and Verilog will try to resolve the output values. If both input bits are 1 (which means the input condition doesn't matter), then the output bit is a 1. Same for both input bits being 0. Any bits that can't be resolved are assigned an x value. If the true_assignment or the false_assignment register width is not wide enough to fill the output_assignment, the output_assignment bits are left-filled with zeros. See Listing 1-24.

Listing 1-24 Conditional Example

```
    module cond_tst (clk, resetn, tristate_control, input_bus,
output_bus);
    input        clk, resetn;
    input        tristate_control;
    input  [7:0] input_bus;
    output       output_bus;
    reg    [7:0] output_bus;

    always @ (posedge clk or negedge resetn)
    begin
          if (!resetn) // Active low asynchronous reset.
          output_bus   <=    8'bz;

          else

// Assign output_bus = input_bus if tristate_control is
//  true and assign output_bus = high impedance if
//  tristate_control is false.
    output_bus   <=    tristate_control ? input_bus : 8'bz;

    end
    endmodule
```

Math Operators

Verilog supports a small set of math operators including addition (+), subtraction (-) , multiplication (*), division (/), and modulus (%); however, the synthesis tool probably limits the usage of multiplication and division to constant powers of two (in other words, a left shifter or right shifter will be synthesized) and may not support modulus. The + and - math operators will instantiate preoptimized adders. Verilog assumes all reg and wire variables are unsigned.

Parameters

Parameters are a useful way of making constants values more readable in the code. Parameters are used only in the modules where they are defined, but they can be changed by higher-level modules. Parameters cannot be changed at run time, but they can be changed at compile time. This is useful in cases where a parameter changes the defined number of signals or the number of instances some construct is used. Not all parameters have to be assigned, but if there is a positional assignment list, parameters can't be skipped.

A parameter can also be defined in terms of other constants or parameters. To aid in reading the code, some people use upper-case characters for parameters.

Listings 1-25 and 1-26 demonstrate Verilog hierarchy, where a module list descends into the hierarchy, starting at the top, and with module names separated by periods.

Listing 1-25 Parameter Example, Top Module

```
module top;
reg          clk, resetn;
parameter    byte_width   = 8;
defparam
      u1.reg_width       = 16;  // This parameter will
                                //  replace the first
                                //  parameter found in
                                //  the u1 instantiation
                                //  of reg_width.
defparam
      u2.reg_width       =      byte_width * 2;
parm_tst u1 (clk, reset, output_bus);  // Create a version
                                       //  of parm_tst with
                                       //  reg_width = 16.
parm_tst u2 (clk, reset, output_bus);  // This version of
                                       //  parm_tst also has
                                       //  reg_width of 16.
parm_tst u3 (clk, reset, output_bus);  // This version of
                                       //  parm_tst has a
                                       //  reg_width of 8.
endmodule
```

Listing 1-26 Parameter Example, Lower Module

```
module       parm_tst (clk, resetn, output_bus);
input        clk, resetn;
parameter    reg_width    = 8;   // This constant can be
                                 // overridden by a parameter
                                 // value passed into the
                                 // module.
parameter    byte_signal        = 8'd99;
parameter    byte_signal_true   = 8'hff;
parameter    byte_signal_false  = 8'h00;
output       [reg_width - 1 : 0] output_bus;
reg          [reg_width - 1 : 0] output_bus;
reg          [7:0]  byte_count;

always @ (posedge clk or negedge resetn)
begin
      if (~resetn) // Active low asynchronous reset.
      begin
      output_bus   <=    8'b0;
      byte_count   <=    8’b0;
      end
```

```
            else if (byte_count == byte_signal)
            output_bus   <=    byte_signal_true;
            else
            begin
            output_bus   <=    byte_signal_false;
            byte_count   <=    byte_count + 1;
            end
      end
      endmodule
```

Concatenations

Concatenations are groupings of signals or values and are enclosed in curly brackets {} with commas separating the concatenated expressions, as shown in Listing 1-27. All concatenated values must be sized. Note the use of [ ] to identify the bit select or register index. It's legal to define a register like backwards_reg, but, regardless of the numbers used, the leftmost definition is always the most significant bit. Usually, you'll see the largest number occurring on the left side of the colon (:) unless a one-dimensional array of variables (like a RAM) is being created.

Listing 1-27 Concatenation Example

```
module backward;
reg [0:2] backwards_reg;
reg [2:0] test;
/*  {1'b0, test, 8'h55} is the same as:

{1'b0, test[2], test[1], test[0], 1'b0, 1'b1, 1'b0, 1'b1, 1'b0,
1'b1, 1'b0, 1'b1} */

always @ (test)
    begin

    test         =     backwards_reg;
// The assignment above is equivalent to the assignments below:
    test[2]      =     backwards_reg[0];
    test[1]      =     backwards_reg[1];
    test[0]      =     backwards_reg[2];
    end
endmodule
```