# Tutorial C

# Introduction to Verilog and Quartus II

## Section I: Overview

Verilog is a *hardware description language* (HDL) that allows us to describe circuits. Unlike other programming languages, which typically execute commands in sequence, Verilog has been designed to describe hardware, and since different hardware components all operate in parallel at the same time, every statement you write in Verilog can be executed concurrently. This is an important distinction, which both gives Verilog significant power in what it can design and can make the language confusing at first for people with prior programming background.

This tutorial will show you how to get started with Verilog for our labs. We will be using the Altera Quartus II program to write and compile our Verilog. It is required that you have completed *Tutorial B: Installing Quartus II* before beginning this tutorial.

## Section II: Creating Your First Circuit

### A. Basic Verilog Syntax

Before you start writing your first circuit, we will cover some basic Verilog syntax. Read this section carefully, as you will be using Verilog for the rest of the semester. You can also read Chapter 4 of the textbook, which describes SystemVerilog, an extended version of Verilog.

#### *Defining a Circuit*

Verilog circuits are composed of a set of *modules*. You can think of a module as a black box circuit. As shown in Figure 1, a module has a set of inputs and a set of outputs which are visible to the outside world. Inside the module, these inputs are connected using logic functions to generate the outputs. We can wire modules together using *wires*. A module can contain several smaller modules inside it, which allows us to build a circuit out of smaller sub-components.

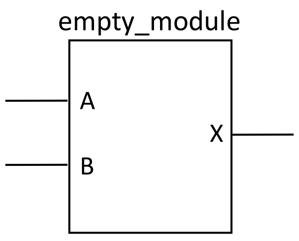


**Figure 1.** Example of a Verilog block.

Code Block I shows how a very basic Verilog module is structured. Here, we have a circuit called `empty_module`. After the module name, inside parentheses, we need to provide a list of inputs and outputs separated by commas, much like a function call in C or Java would (Line 1). Note, though, that we do not define the type yet; we do that afterwards (Lines 2-5). We must explicitly declare all of the connections (which, unless otherwise specified, are *wires*) from Line 1 as either inputs or outputs, using the format `<input/output> <pin_name>`. You can also declare a bus by adding the width in square brackets beforehand. For example, wire **B** in Code Block I (Line 3) is a 4-bit bus, whose individual wires are named **B[0]**, **B[1]**, **B[2]**, and **B[3]**, where **B[3]** is the *most-significant bit* (MSB). Note that the order of the bus indices is important. For our purposes, we will stick to the convention of always writing the higher number first.

Each of these inputs and outputs must be separated by a semicolon. After the input declaration, logic rules can be written to create the output values needed. Once this is complete, the keyword `endmodule` (with no punctuation) must be placed, to let the compiler know that there is no more code for the `empty_module` object. If you look at Code Block I, you will see that all of the code in between `module` and `endmodule` is indented for clarity – you should do this in your code as well.

**Code Block I.** Example of basic Verilog module declaration.

```
/*  1 */  module empty_module(A, B, X);
/*  2 */    input       A;
/*  3 */    input [3:0] B;
/*  4 */
/*  5 */    output      X;
/*  6 */
/*  7 */    /* Add Verilog for logic functions here. */
/*  8 */
/*  9 */  endmodule
```

**NOTE 1**

You **must** place a semicolon at the end of the parentheses next to the module name, as seen in Code Block 1. Leaving out the semicolon could lead to some strange-looking compiler errors. Also, do not place a comma at the end of the input/output list, or it will also throw an error.

## *Commenting Your Code*

Verilog uses C-style comments. Any text placed between `/*` and `*/`, as is done in Code Block I, is treated as a comment block, and is ignored by the compiler. You can use comment blocks to write long comments that may take up several lines. For shorter comments, you can use two forward slashes – anything to the right of `//` is ignored until the end of the line. Comments are a helpful way to describe to a reader what function is implemented and how this was accomplished. You should get in the habit of commenting now, as it will be very helpful for the larger projects later in the semester.

## *Assigning Values*

The Verilog keyword `assign` is used to set values for output wires. Code Block II shows several examples of how this assignment can be done. We can either assign a *constant* (i.e., a value that always outputs a logic '1' or '0') or a logic statement using Boolean algebra. Inside Code Block II, lines 11 and 12 show how we assign constants. We first type in the **number of bits to assign** (and *not* the number of digits), then enter an apostrophe, followed by a letter to indicate the type of data (`b` for binary, `d` for decimal, `h` for hexadecimal, or `o` for octal) and the value to assign. For example, `6'b110010` equals $110010_2$, `6'd37` equals $100101_2$, and `8'hA9` equals $10101001_2$. All assign statements must end with a semicolon.

**Code Block II.** A Verilog module with several assignments.

```
/*  1 */  module not_so_empty_module(input_one, input_two, input_three,
/*  2 */          OutW, OutX, OutY, OutZ);
/*  3 */    input         input_one;
/*  4 */    input         input_two;
/*  5 */    input   [1:0] input_three;
/*  6 */    output        OutW;
/*  7 */    output  [1:0] OutX;
/*  8 */    output  [1:0] OutY;
/*  9 */    output  [2:0] OutZ;
/* 10 */
/* 11 */    assign OutW = 1'b1;
/* 12 */    assign OutZ = 3'b101;
/* 13 */    assign OutX = input_three;
/* 14 */    assign OutY[0] = input_one & input_two;
/* 15 */    assign OutY[1] = ((~input_one) & input_two)
/* 16 */                    | (input_one & (~input_two);
/* 17 */
/* 18 */  endmodule
```

Lines 13 – 16 of Code Block II show how we can assign combinational logic to the outputs. In this example, we set **OutX** to be equal to **input_three**, bit 0 of **OutY** to **input_one** AND **input_two**, and bit 1 of **OutY** to **input_one** XOR **input_two** (here, we manually create the XOR using two inverters, two AND gates, and an OR gate). Table A shows the different logical operators that we can assign. Notice in Lines 15 and 16 that we can use parentheses to enforce the ordering of the logic, just as we have done when writing algebra.

| LOGIC OPERATION | SYNTAX |
|---|---|
| Bitwise AND | `A & B` |
| Bitwise OR | `A | B` |
| Bitwise NOT | `~A` |
| Bitwise XOR | `A ^ B` |
| Bitwise NAND | `~ (A & B)` |
| Bitwise NOR | `~ (A | B)` |

**Table A.** Verilog logical operators.

**NOTE 2**

Do **not** use + or * to perform OR and AND, respectively, when you are writing Verilog. These are used to implement other functions, which we will discuss in a later lab. If you try to use these to implement sum-of-product equations, **your circuit will produce incorrect values**.

Another point to note from Code Block II is that for multi-bit buses, we can either assign a multi-bit value to the entire output (such as in Lines 12 and 13), or we can set each individual bit individually (as in Lines 14 – 16).

## B. Writing Your First Circuit in Quartus

For all of our labs, we will provide you with already-started projects that contain all of our files and settings. These are convenient to save, close, and re-open at a later date without having to set up everything once more. A project called ***project_name*** will consist of two basic files:

***project_name*.qpf**, which is the file you will use to open the project, and ***project_name*.qsf**, which contains all of the settings associated with the project. These files will typically require a *top-level* Verilog file named ***project_name_top*.v**, which will contain your circuit itself.

**ASSIGNMENT 1** Download **lab2.zip** from the Lab 3 assignment. Unzip this onto your computer, into a folder called **lab2**.

**NOTE 3** If you are using a virtual machine (such as VirtualBox), make sure to download the ZIP file *from inside Ubuntu*. If you don't download it in the virtual machine, Quartus may be unable to see the files.

**NOTE 4** For our labs, you cannot simply open a compressed folder (a ZIP file) without unzipping it. Quartus (and most software) cannot see inside the ZIP file. In Windows/Ubuntu, you can unzip by opening the ZIP file and clicking on *Extract all files*.

**ASSIGNMENT 2** Open the **comb** project.

1. Click on *File* → *Open Project…* Go to the **lab2** folder that you created, and then click on the **comb** file. Click the *Open* button.
2. On the left, you should see a box called *Project Navigator*, with three tabs. If the project opened correctly, you will see **comb** listed under the *Hierarchy* tab.

**ASSIGNMENT 3** Open the **comb.v** Verilog file.

1. Inside the *Project Navigator* box, click on the *Files* tab.
2. Double-click on the file **comb.v**. This should open a Verilog file.

Inside this file, we will now create and test out a boolean function with four variables $W$, $X$, $Y$, and $Z$. The logic both in SOP form and using NAND gates only, is:

$$A = W + X'Z' + XZ + YZ \qquad\qquad A = (W'(X'Z')'(XZ)'(YZ)')'$$

Inside **comb.v**, you will see that we have already provided the inputs and outputs for you. You only need to add the `assign` statements to implement the logic itself. The 1-bit output **OUT** will have the logic for the SOP implemented, while the 1-bit output **OUTNAND** should use the logic using only NAND gates. (Hint: to write a NAND gate, use the syntax `~(A & B)`, as there is no single NAND operator.)

**ASSIGNMENT 4** Complete the **comb** module. Use Code Block I and Code Block II as examples, and use only the Boolean operators from Table A.

**ASSIGNMENT 5** Compile your design to check for errors.

1. Click on *Processing* → *Start Compilation*.
2. Wait until the messages box at the bottom of the Quartus window says *Quartus II Full compilation was successful*. If it shows errors, there is a mistake in your design – go back and verify that your circuit is typed up properly. Otherwise, your syntax is fine, and you can click on *OK*. (Note: while your syntax may be fine, your logic may not be correct – we will check this later).
3. Close the *Compilation Report* tab.

**NOTE 5** Quartus II has an odd quirk where sometimes the names are case-sensitive, while other times they aren't. **Make sure you keep the casing and format of the names exactly as typed in the handouts, for both symbols and file/project names.** Otherwise, your circuit may not work.

Now that we have successfully compiled the circuit, we must verify that it works the way we have intended it to. In order to do this, we will use *test benches*, which are a Verilog files that contains the combinations of inputs we want to verify. For now, we will provide complete test benches for you.

**ASSIGNMENT 6** Before beginning simulation, read through the test bench file so you understand what it's trying to do.

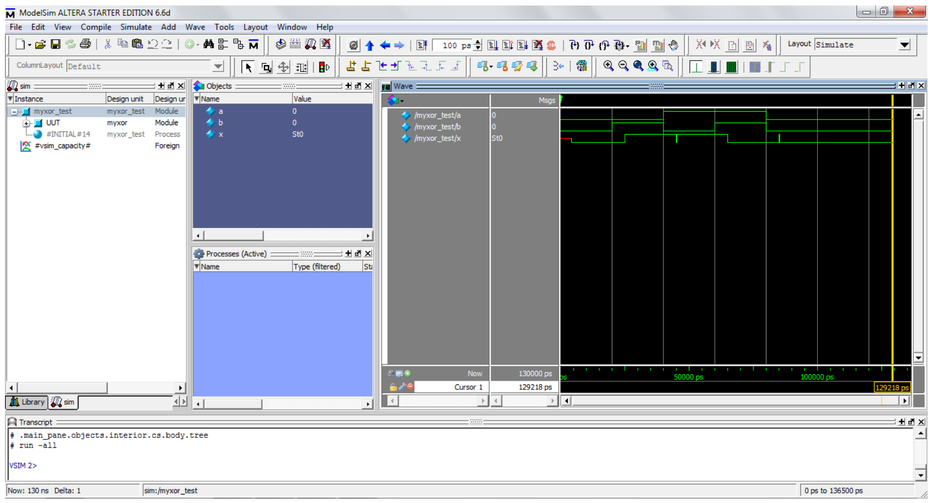
1. Click *File* → *Open…* Go to the **lab2** folder, and open the file **comb_test.v**.
2. Read through the code. The test bench has been commented to give you a description of what each part of the code is doing. Do not make any modifications to this file.
3. When finished, close the test bench.

**ASSIGNMENT 7** Simulate your circuit and verify the results.

1. Click on *Tools* → *Run Simulation Tool* → *RTL Simulation*.
2. Wait for the simulation window to appear (Figure 3). You will see green lines on the right hand side when it is finished. This is the *waveform*, which plots the input and output values of the circuit over time. If the waves do not show up, make sure you have the Wave window shown (check *View* → *Wave*), and that there are no errors in the Transcript window (at bottom).
3. Under the *View* menu, make sure that the following are checked: *Library*, *Objects*, *Transcript*, and *Wave*.
4. Zoom out to see what the waveform is doing, using the magnifying glass buttons at the top of the toolbar. (The filled-in magnifying glass

fits the entire simulation into the current window, though this may be zoomed too far out in some cases.)

5. You can verify this waveform against the truth table for Segment A. We have also written the test bench to verify the truth table on its own. Go to the Transcript window at the bottom of ModelSim. If you scroll up and down, you will see statements that start with MSIM – these are printouts that we have provided. You can easily see if you have made a mistake using these outputs. If one of your outputs is listed as `z`, that means it has not been connected properly. If an output is listed as `x`, then multiple values have been written to a single output at the same time. Both of these cases should be corrected in your Verilog file.


ModelSim ALTERA STARTER EDITION 6.6d
File Edit View Compile Simulate Add Wave Tools Layout Window Help
Objects
Wave
Transcript
/myxor_test/a
/myxor_test/b
/myxor_test/x
Now
Cursor 1
Library
sim

```
# .main_pane.objects.interior.cs.body.tree
# run -all

VSIM 2>
```

Now: 130 ns Delta: 1    sim:/myxor_test

**Figure 3.** ModelSim-Altera window once simulation has completed.

6. Close the ModelSim-Altera window when you are done.
7. If you need to correct your Verilog, close ModelSim first, and then edit and re-compile **comb.v**, and redo Steps 1-6. If you don't close ModelSim, you will receive weird error messages. **To receive full credit on all prelabs, your code must pass all of the test bench cases.**

## Section III: More on combinational logic

The circuit we created in Section II was a simple combinational circuit (and already, we can see how much easier it was to implement and test when compared to using actual ICs and wires). Now we look at how we can build more complex combinational circuits.

### A. Verilog Syntax

#### *Conditional Assignment*

Code Block II showed us how to assign certain logic functions to output wires using sum-of-product notation. However, sometimes we want to assign an output only if a certain condition is met. The *conditional assignment* operator allows us to choose one of two values to assign, based on the value of a *condition* variable. Think of it as a 2-to-1 multiplexer – based on the one-bit select input, we will choose one of two values to output. (For those familiar with coding, this is akin to an if/else statement; the crucial difference is that *everything is evaluated in parallel here*.)

**Code Block III.** Using conditional assignment in Verilog.

```
/*  1 */  module silly_module(A, B, C, D, E, X, Y);
/*  2 */    input A;
/*  3 */    input B;
/*  4 */    input C;
/*  5 */    input [2:0] D;
/*  6 */    input [1:0] E;
/*  7 */
/*  8 */    output X;
/*  9 */    output [1:0] Y;
/* 10 */
/* 11 */    assign X = A ? B : C;
/* 12 */    assign Y[0] = E[0] ? D[2] : (E[1] ? D[0] : D[1]);
/* 13 */    assign Y[1] = A ? (B ? 1'b1 : D[1]) : (B ? C : 1'b0); // 4:1 mux
/* 14 */
/* 15 */  endmodule
```

Code Block III illustrates how to use a conditional assignment operator. On Line 11, we first check the value of our condition variable **A**. If **A** is equal to 1, then **X** will be set to the input **B**. If A is equal to 0, then **X** will be set to **C**. Note that the condition variable can only be one bit wide – conditional assignment operators cannot implement a multiplexer larger than 2-to-1. In order to emulate the behavior of larger multiplexers, we can nest a conditional assignment operator inside another conditional assignment operator, as seen in Lines 12 and 13. Code Block IV illustrates a more explicit example of a multiplexer.

**Code Block IV.** Implementing a 4-to-1 6-bit multiplexer in Verilog.

```
module mux4to1(MUXIN0, MUXIN1, MUXIN2, MUXIN3, SEL, MUXOUT);
  input  [5:0] MUXIN0;
  input  [5:0] MUXIN1;
  input  [5:0] MUXIN2;
  input  [5:0] MUXIN3;
  input  [1:0] SEL;

  output [5:0] MUXOUT;

  assign MUXOUT = SEL[1] ? (SEL[0] ? MUXIN3 : MUXIN2) :
                       (SEL[0] ? MUXIN1 : MUXIN0);

endmodule
```

**NOTE 6** Whenever you declare a bus, you should make sure to put the higher number first inside the brackets (i.e., `[7:0]` instead of `[0:7]`). This corresponds to the order in which we read binary numbers, and it also makes sure that we perform operations and assign values in a consistent order.

## *Creating Internal Variables*

While all we do inside a Verilog module is assign some Boolean operation on inputs to an output, this is often cumbersome when we have a lot of logic to implement. To reduce some of the burden, we can use *internal variables* to do a partial Boolean operation, saving this intermediate result for use in another partial operation. This allows us to break down complex modules into more manageable pieces inside the module itself. We can see an example of this in Code Block V, where the wire `partial_logic` holds some intermediate value that we can later incorporate in our outputs.

**Code Block V.** A Verilog module with internal variables.

```
module complex_module(A, B, C, X, Y);
  input  A;
  input  B;
  input  C;

  output X;
  output [1:0] Y;

  wire partial_logic;

  assign partial_logic = (A | B) ^ (B & (~A));
  assign X = ~(partial_logic | C);
  assign Y[0] = partial_logic & A;
  assign Y[1] = (~partial_logic) | B;

endmodule
```

## *Defining Registers in Verilog*

Previously, we have used the `wire` type to interconnect different Verilog modules (which is the implied type for all inputs and outputs). When using `assign`, we must assign to wires. However, for logic that must be triggered (i.e., only done at a certain time), we need to use the `reg` type. The difference is that while `wire` simply transmits the data directly between two points, `reg` can potentially store the value of the signal. Code Block VI shows how `reg` can be used as either an output port (Lines 6-8) or as an intermediate storage component (Lines 10-11). Importantly, when we treat an output port as a `reg`, you can see that we declare it twice – first as an `output`, and then as a `reg`.

**Code Block VI.** A Verilog module with registers.

```
/*  1 */  module harmless_circuit(a, b, C, D);
/*  2 */    input  a;
/*  3 */    input  b;
/*  4 */
/*  5 */    output C;
/*  6 */    output [3:0] D;
/*  7 */
/*  8 */    reg [3:0] D;
/*  9 */
/* 10 */    reg F;
/* 11 */    reg [31:0] myMemory;
/* 12 */
/* 13 */    /* To do: implement Skynet */
/* 14 */
/* 15 */  endmodule
```

### *Building a combinational circuit using always blocks*

Verilog can be told when to do the register write by the `always` block. We consider the 4-to-1 multiplexer example in Code Block IV and build the same using always blocks. Code Block IV shows an example of the `always` block on Line 11. The `always` block includes a *sensitivity list* – after the `@` symbol, in parentheses, we write the names of variables that, when they change, will cause the lines inside the block to be evaluated. These variables are separated by commas. For example, the block on Lines 12-22 will be executed only whenever the value of SEL changes. Note that these blocks are delineated by the keywords `begin` and `end`.

**Code Block VII.** Implementing a 4-to-1 6-bit multiplexer in Verilog using always block.

```
/*  1 */  module mux4to1(MUXIN0, MUXIN1, MUXIN2, MUXIN3, SEL, MUXOUT);
/*  2 */    input  [5:0] MUXIN0;
/*  3 */    input  [5:0] MUXIN1;
/*  4 */    input  [5:0] MUXIN2;
/*  5 */    input  [5:0] MUXIN3;
/*  6 */    input  [1:0] SEL;
/*  7 */
/*  8 */    output [5:0] MUXOUT;
/*  9 */    reg temp;
/* 10 */
/* 11 */    always@(SEL)
/* 12 */    begin
/* 13 */        if (SEL == 2b'0) begin
/* 14 */           temp = MUXIN0;
/* 15 */        end else if (SEL == 2b'01) begin
/* 16 */           temp = MUXIN1;
/* 17 */        end else if (SEL == 2b'10) begin
/* 18 */           temp = MUXIN2;
/* 19 */        end else if (SEL == 2b'11) begin
/* 20 */           temp = MUXIN3;
/* 21 */       end
/* 22 */   end
/* 23 */   assign MUXOUT = temp
/* 24 */
/* 25 */  endmodule
```

The sensitivity list allows us to write values to registers when the variables we are monitoring change. Sometimes, we would like to perform this evaluation only when a variable is set to a certain value, or when a Boolean algebra expression is true. We can use an `if/else` statement to do this, as shown on Lines 13-21 of Code Block VII. Again, we use the `begin` and `end` keywords to put all of these statements in a block. If you do not need the `else` part of the statement, you can leave it out entirely. As you can see, a single `always` block can contain multiple `if/else` blocks.

Notice that, just as we indented the contents of a module, we indent the contents of the `always` block even further in. By having this kind of indentation, your code will be much more legible. The same is true for the contents of each `if/else` block. As you can see in Code Block VII, the indentation makes it very easy for us to identify what's going on, and how logic is nested.

In Code Block VII, we use always block to cleanly write the complex combinational part which was earlier written using conditional statements in Code Block IV. At the end, we assign `temp` to output wire `MUXOUT` giving us the same functionality as Code Block IV.

As a rule of thumb, any output or variable being set (i.e., on the left hand side of the statement) by an assign should be a `wire`, and any output or variable set inside an always block should be a `reg`.

Now, we will build the comb module from Assignment 4 above using always blocks.

**ASSIGNMENT 8** Comment the code you added in Assignment 4 in comb module and now complete the **comb** module using always blocks. Use Code Block VI and Code Block VII as examples.

**ASSIGNMENT 9** Compile your design similar to Assignment 5.

**ASSIGNMENT 10** Simulate and verify your design by repeating Assignment 7 again with the new code for comb module.

## Section IV: Building sequential circuits

When we are writing Verilog for sequential circuits, we usually want to write to these registers at a particular time. This could be when a previous value changes or on the rising edge of a *clock*, for example. Code Block VIII shows an example of such an `always` block on Line 22.

**Code Block VIII.** A simple sequential circuit in Verilog.

```
/*  1 */  module sequencer(clk, A, B, C, X, Y, Z);
/*  2 */    input  clk;
/*  3 */    input  A;
/*  4 */    input  B;
/*  5 */    input  C;
/*  6 */
/*  7 */    output X;
/*  8 */    output [2:0] Y;
/*  9 */    output Z;
/* 10 */
/* 11 */    reg    X;
/* 12 */    reg    [2:0] Y;
/* 13 */    reg    Z;
/* 14 */
/* 15 */    reg W;
/* 16 */
/* 17 */    always @(A, B) begin
/* 18 */      W = A ^ B;
/* 19 */      X = (~W) ^ C;
/* 20 */    end
/* 21 */
/* 22 */    always @(posedge clk) begin
/* 23 */      Y[0] <= A | ~B;
/* 24 */
/* 25 */      if(X == 1'b1) begin
/* 26 */          Y[1] <= 1'b0;
/* 27 */          Z <= 1'b0;
/* 28 */      end
/* 29 */      else begin
/* 30 */          Y[1] <= W;
/* 31 */          Z <= 1'b1;
/* 32 */      end
/* 33 */
/* 34 */      if(~C) begin
/* 35 */          Y[2] <= ~A;
/* 36 */      end
/* 37 */    end
/* 38 */
/* 39 */  endmodule
```

Oftentimes, for a signal like a clock, we only want to run the `always` block when the signal transitions from low to high (the *rising edge*) or from high to low (the *falling edge*). This can be written in the sensitivity list by adding `posedge` (rising edge) or `negedge` (falling edge) before the name of the signal, as seen in Line 22. In Line 23, we set the value of **Y[0]** once the clock signal has a rising edge. At all other times, the value of **Y[0]** remains at whatever it was previously set to, which is the behavior we expect when using a register.

## *Blocking and Non-Blocking Statements*

If you look closely at Code Block VIII, you will notice that for some of the always blocks, we use the <= symbol (e.g., Line 26) to set a value instead of a = symbol, as we have traditionally used. In sequential logic, since timing is extremely important, it matters whether we perform the different lines of code within a block all in parallel, or if we do them one at a time. If we want to do them in parallel, we use the <= symbol to say that the statement is *non-blocking*. Instead, if we want the next statement to be evaluated only after the previous one finishes (which means we must wait for the delay of the circuit), we use the = symbol to tell Verilog that the statement is *blocking*. In a nutshell, use blocking statements only when the order of the statements matters, and use non-blocking statements at all other times.

## Section V: Inferred Latches

The modules that you have coded using `always` blocks have implemented flip-flops. Flip-flops are automatically created by Quartus whenever the sensitivity list (i.e., the triggers for the always block) are *edge-trigged* (i.e., `posedge` or `negedge`). However, it is also possible to implement logic inside an `always` block that does not create a flip-flop, such as Lines 17-20 in Code Block VIII. Since these blocks change as a function of the actual value of the variables, and not of the edge, these are called *level-triggered* gates.

We have to be careful with level-triggered `always` blocks. While we want them to represent some form of combinational logic, we forget to assign an output in certain cases. For example, with an `if/else` statement, we may leave out the output assignment in part of the statement, as seen in Code Block IX (the assignment of `y` is missing when `reset` is high, on Lines 12-14). If we forget to assign the output for certain cases, Quartus will need to find some way of saving the value of the output until the next change. In order to do this, Quartus will *infer* from your code that it needs to add a *latch* (hence the term inferred latch).

**Code Block IX.** A module that contains an inferred latch.

```
/*  1 */  module lean_green_inferring_machine(reset, a, x, y);
/*  2 */    input  reset;
/*  3 */    input  a;
/*  4 */
/*  5 */    output x;
/*  6 */    output y;
/*  7 */
/*  8 */    reg    x;
/*  9 */    reg    y;
/* 10 */
/* 11 */    always @(reset, a) begin
/* 12 */      if(reset) begin
/* 13 */        x <= 1'b0;
/* 14 */      end
/* 15 */      else begin
/* 16 */        x <= 1'b1;
/* 17 */        y <= ~a;
/* 18 */      end
/* 19 */    end
/* 20 */
/* 21 */  endmodule
```

For an FPGA, inferred latches are bad. They are often the result of an accidental omission, and can result in unexpected timing assumptions. Worse yet, since the FPGA is designed to be *synchronous* (changes are coordinated with the clock), yet latches are *asynchronous* (changes occur at any time), circuits with inferred latches can cause very unusual behavior. Fortunately, whenever Quartus infers a latch, it adds a comment inside the Warnings tab when you compile your circuit. After every compilation, check the Warnings tab to see if any latches were inferred. You should go back through your code, figure out which cases you are forgetting to assign an output for, and then add the output assignment in. For example, Code Block X shows the corrected code from Code Block IX with the inferred latch removed. **For all code submitted for class, you must remove all inferred latches from your design.**

**Code Block X.** The module from Code Block IX, but with the inferred latch problem fixed.

```
module infer_this(reset, a, x, y);
  input  reset;
  input  a;

  output x;
  output y;

  reg    x;
  reg    y;

  always @(reset, a) begin
    if(reset) begin
      x <= 1'b0;
      y <= 1'b0;
    end
    else begin
      x <= 1'b1;
      y <= ~a;
    end
  end

endmodule
```

## For More Information

If you want to learn more about Verilog, or are interested in finding out additional syntax, read the Quick Reference Guide for Verilog, which is available online at:

http://fpgacpu.ca/fpga/hdl/verilog_2001_ref_guide.pdf