An Introduction to Verilog

With examples for the Altera DE1

By: Andrew Tuline

Date: May 30, 2013

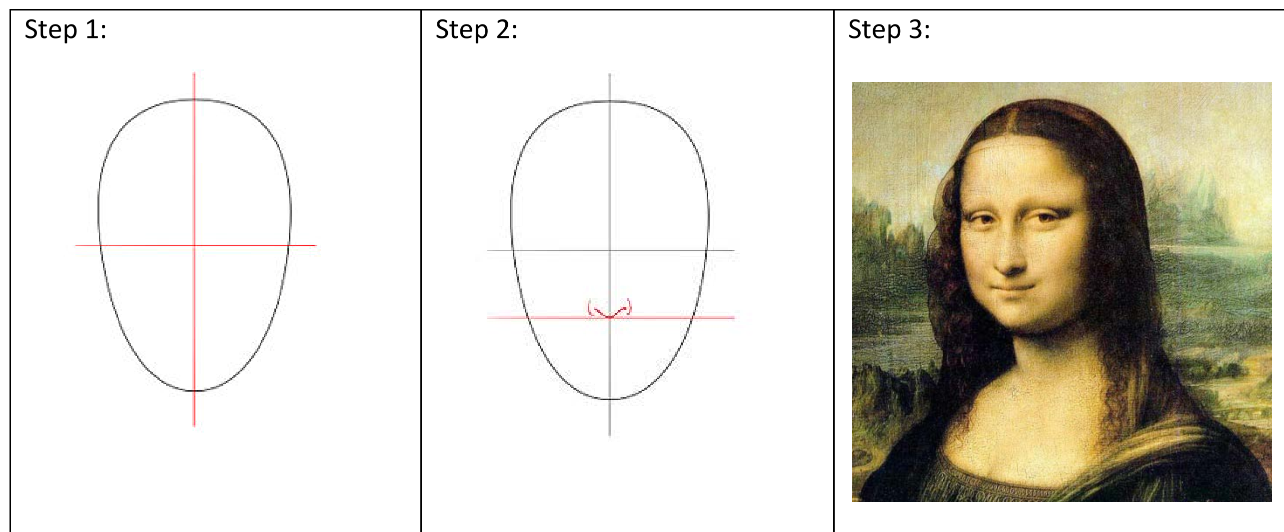
Feel free to send constructive feedback to atuline@gmail.com

This is STILL an early work in progress. . .

Introduction

Whether it's computers or art, it never ceases to amaze me how many so called 'Introductory' books start out with simple concepts but then take a huge leap to the finished product. For instance . . .

How to draw a face:



I can only shake my head at the gargantuan jump in complexity and then move onto the next book.

This document aims to build upon basic elements of digital logic written in Verilog and slowly build upon them. It is not a reference manual, but rather a simple guide with reasonable steps between each section. I suggest supplementing this with other materials that explain various concepts in more depth.

Platform and Pre-Requisites

Before continuing, you should have a basic understanding of digital logic, including:

- Truth tables
- And, Or, Xor, Not gates
- Latches, flip flops
- Karnaugh mapping and simple combinational logic
- Simple multiplexers, decoders and counters

You should have already installed a Verilog platform of some form. In my case, I am using the Altera DE1 FPGA development board. The DE1 comes equipped with a clock, several switches and LED's which we'll use to provide inputs and outputs for our circuits. I am using version 13.0 of Altera's Quartus II software along with the version 10.1 of the Modelsim simulator. In addition, you should already know how to get out of the starting gate with your platform of choice. For the DE1, I went through the Digital Logic Tutorials at:

http://www.altera.com/education/univ/materials/digital_logic/tutorials/unv-tutorials.html

The ones I did were the Schematic and Verilog tutorials at:

ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Tutorials/Schematic/Quartus_II_Introduction.pdf

ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Tutorials/Verilog/Quartus_II_Introduction.pdf

The above tutorials get you started with the DE1 (or other DEx boards) and shows you how to:

- Create a project with your development board of choice
- Create a schematic or Verilog file
- Create a simple design
- Use the assignment editor to match circuit inputs/outputs to devices on the board
- Compile the design
- Program your board with the compiled design (using the .sof file)
- Display the resultant circuit with the RTL viewer
- Toggle the switches and watch the LED's blink

Altera also provides some labs, (but these come without much supporting educational material or answers, so their value may be limited to those of us not in university. They are located at:

http://www.altera.com/education/univ/materials/digital_logic/labs/unv-labs.html

Combinational Logic

As beginner’s we’re going to create some designs using Verilog and then program them on the development board. Once we have a reasonable grasp of things, we will graduate onto using a simulator to test our designs. This allows for improved debugging and timing analysis.

AND Gate

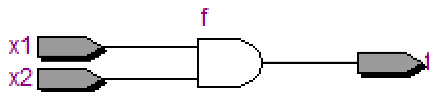
Go ahead and create a simple AND gate using Verilog. This should be a backwards step as you should have already created an XOR gate with Altera’s Verilog tutorial. This first example uses Verilog 1995 to define the ports.

```
// And gate
//
module andgate (x1, x2, f);
      input x1, x2;
      output f;

      assign f = x1 & x2;

endmodule
```

Note: To get this to compile, I made sure that the name of the Verilog file was called ‘**andgate.v**’.

RTL Viewer

Once this compiled properly, you can view the output by selecting ‘**Tools | Netlist Viewers | RTL Viewer**’:



Assignment Editor

The assignments from your circuit to the FPGA location will vary depending on the board you are using. If these weren’t already assigned, you’ll need to re-compile your design before programming the DE1.

<<new>> ▾ ☑ Filter on node names: *

	tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✔		in x1	Location	PIN_L22	Yes			
2	✔		in x2	Location	PIN_L21	Yes			
3	✔		out f	Location	PIN_U22	Yes			
4		<<new>>	<<new>>	<<new>>					

On the DE1, PIN_L22 maps to SW[0] on the board (which we call x1), while PIN_L21 maps to SW[1] (which we call x2) and PIN_U22 maps to LEDG[0] (which we call f). Once you have compiled and programmed the board, you should be able to toggle the switches and see the expected outputs.

QSF files

If you haven't already done so, you should download and import the DE1 (or appropriate assignment file to your project. For the DE1, it's available at:

ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE1/DE1.qsf

Once the .qsf file has been imported by selecting '**Assignment | Import Assignments**', your assignment editor should look something along the lines of:

<<new>> Filter on node names: *

	tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	?		x1	Location	PIN_L22	Yes			
2	?		x2	Location	PIN_L21	Yes			
3	?		f	Location	PIN_U22	Yes			
4	?		SW[0]	Location	PIN_L22	Yes			
5	?		SW[1]	Location	PIN_L21	Yes			
6	?		SW[2]	Location	PIN_M22	Yes			
7	?		SW[3]	Location	PIN_V12	Yes			
8	?		SW[4]	Location	PIN_W12	Yes			

These should all turn black once the circuit has been compiled. Here's the AND gate re-written with Altera's qsf definitions. It uses 'bus' notation, which allows us to define several switches with a single entry. We can then use them in our logic as SW[1] and SW[0], which were defined in the .qsf file. For example:

```
// And gate
//
module andgate (SW, LEDG);
      input [1:0]SW;               // We'll use 2 switches
      output [0:0]LEDG;            // And a single green LED

      assign LEDG[0] = SW[1] & SW[0];

endmodule
```

When defining an array of inputs or outputs, you put the brackets with the size of the bus **prior** to the name. For example [0:0] is 1 bit wide, while both [3:0] and [7:4] are 4 bits wide.

Here's another way to write this more concisely in Verilog 2005 format:

```
// And gate
//
module andgate (
      input [1:0] SW,
      output [0:0] LEDG
);

      assign LEDG[0] = SW[1] & SW[0];

endmodule
```

We'll try and remember to use this notation with our remaining examples.

'Wire'

We often need to connect gates together and we'll use a command called '**wire**' to do so.

A '**wire**' is a passive connection. You cannot assign a specific value to a '**wire**', so it's typically used between combinational logic elements. For a good reference, see:

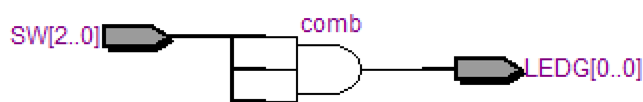
http://www.asic-world.com/tidbits/wire_reg.html

3 Input AND Gate

We're going to use a couple of 2 input AND gates to create a 3 input AND gate. We'll need a wire to connect the output of one of the AND gates to the other.

```
// AND3 gate from two AND2 gates
//
module and3 (
      input [2:0]SW,
      output [0:0]LEDG
      );

      wire s1;              // We need a wire for interconnection

      assign s1 = SW[1] & SW[0];
      assign LEDG[0] = s1 & SW[2];

endmodule
```

Here is how it appears in RTL:



Alternatively, we could have used a single assign statement without requiring a '**wire**' such as:

assign LEDG[0] = SW[0] & SW[1] & SW[2];

or even:

assign LEDG[0] = &SW;

If you continue to have any significant problems up to this point, please make sure that you have gone through the initial study materials.

Buffers

As shown earlier, you can define an array and use it as a bus. Here's an 8 bit wide set of buffers:

```
// Buffers
//
module buffers (
      input [7:0] SW,
      output [7:0] LEDG
);

      assign LEDG = SW;           // Could use '=~SW' for inverters

endmodule
```

Five Wide Xor

In this example, the LED will be on when an odd number of switches are high.

```
// Xor5
//
module xor5 (
      input [4:0] SW,
      output [0:0] LEDG
);

      assign LEDG = ^SW;

endmodule
```

Suggestion: If you're looking to design a circuit in Verilog, try Googling for it. It's amazing what you can find.

A 4:1 Multiplexer

Apparently, there's lots of ways you can design this, so let's look at some. The first example is my favourite.

We will want to give our SW's different names as they're going to be used in arrays for different purposes. Let's use the assignment editor and duplicate the SW settings to other variables:

- select [1:0] will use the same pins as SW[1:0]
- dat[3:0] will use the same pins as SW[7:4]

In this case, select[1] would be PIN_L21, while dat[3] would be PIN_M2 and so on.

```
// 4 to 1 mux
//
module mux4to1 (
      input [1:0] select,
      input [3:0] dat,
      output [0:0] LEDG
);

assign LEDG[0] = dat[select];    // 'select' is a 2 bit value

endmodule
```

Another method is as follows for a 2 to 1 mux:

```
// 2 to 1 mux
//
module mux2to1 (
      input [1:0] select,
      input [1:0] dat,
      output [0:0] LEDG
);

assign LEDG[0] = select ? dat[1] : dat[0];

endmodule
```

One way to combine single variables with an array would be as follows:

```
// 2 to 1 mux
//
module mux2to1 (
      input [1:0] select,
      input [1:0] dat1, dat0,
      output [0:0] LEDG
);

assign LEDG[0] = select ? dat1 : dat0;
endmodule
```

You could use a similar technique for a 4 to 1 multiplexer as well, but it starts to get ugly as shown below. I'll take the first version thank you.

```
// 4 to 1 mux
//
module mux4to1 (
      input [1:0] select,
      input [3:0] dat,
      output [0:0] LEDG
);

assign LEDG[0] = select[1] ? (select[0] ? dat[3] : dat[2])
                            : (select[0] ? dat[1] : dat[0]);
endmodule
```

By now, you should be comfortable with assigning inputs and outputs to switches and LED's, so unless something different comes up like a clock, I'll assume you're OK.

Module Instantiation (i.e. subroutines)

We can use define and use modules and include them in our main code.

When passing parameters between the top module and lower level modules, you can use either:

- Implicit (declarations are in the same order)
- Named (both declarations are listed and linked

In addition, the name for including a module is called 'instantiation'. Finally, each instantiation can have a unique name.

For example:

```
module top (
      input in1, in2, in3, in4,
      output out1, out2
);

      // Implicit instantiation - must be in same order
      dostuff first (in1, in2, out1);

      // Named instantiation - they don’t need to be in order
      dostuff second (.stin1(in3), .stin2(in4), .stout(out2));

endmodule

module dostuff (
      input stin1, stin2,
      output stout
);
      assign stout = stin1 & stin2;

endmodule
```

Simple Half Adder

We’ll define a half adder and then use that to create a full adder.

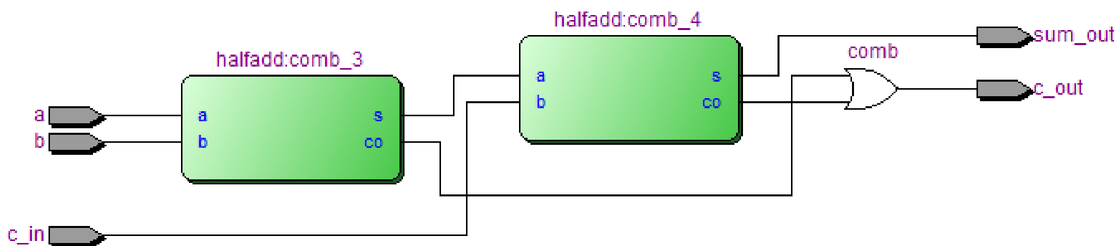
```
// Half Adder
//
module halfadd (
      input a, b,
      output s, co
);

      assign s = a ^ b;
      assign co = a & b;

endmodule
```

Full Adder

We will need to use some wires and a couple of instantiations to make this happen.

```
// Full Adder from Half Adders
//
module fulladd (
      input a, b, c_in,
      output sum_out, c_out
);

      wire s1, c1, c2;

      halfadd (a, b, s1, c1);
      halfadd (s1, c_in, sum_out, c2);
      assign c_out = c1 | c2;

endmodule


module halfadd (
      input a, b,
      output s, co
);

      assign s = a ^ b;
      assign co = a & b;

endmodule
```

Here’s the rtl output:



Hex Display

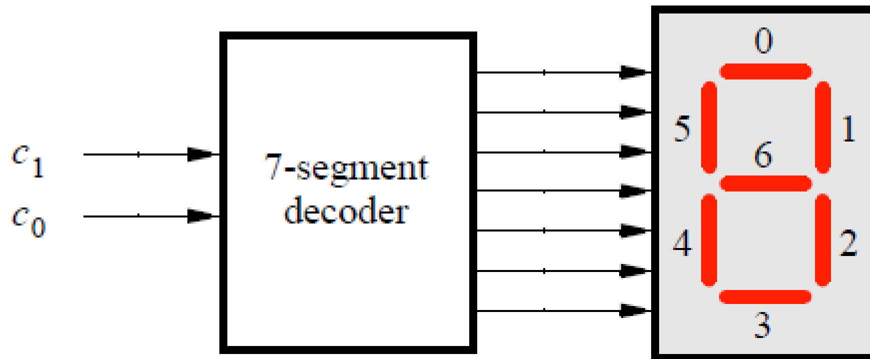


Figure 6. A 7-segment decoder.

Note: For HEX0, when a bit is ‘1’, that LED segment is OFF.

KEY[2]	KEY[1]	KEY[0]	HEX0[0]	HEX0[1]	HEX0[2]	HEX0[3]	HEX0[4]	HEX0[5]	HEX0[6]
0	0	0	0	0	0	0	0	0	1
0	0	1	1	0	0	1	1	1	1
0	1	0	0	0	1	0	0	1	0
0	1	1	0	0	0	0	1	1	0
1	0	0	1	0	0	1	1	0	0
1	0	1	0	1	0	0	1	0	0
1	1	0	0	1	0	0	0	0	0
1	1	1	0	0	0	1	1	1	1

Karnaugh Mapping

HEX0[0] Karnaugh Map

	[2]/[1]			
[0]	00	01	11	10
0	0	0	0	1
1	1	0	0	0

HEX0[0] = (~SW[2] & ~SW[1] & SW[0]) | (SW[2] & ~SW[1] & ~SW[0])

HEX0[1] Karnaugh Map

	[2]/[1]			
[0]	00	01	11	10
0	0	0	1	0
1	0	0	0	1

HEX0[1] = (SW[2] & SW[1] & ~SW[0]) | (SW[2] & ~SW[1] & SW[0])

HEX0[2] Karnaugh Map

	[2]/[1]			
[0]	00	01	11	10
0	0	1	0	0
1	0	0	0	0

HEX0[2] = ~SW[2] & SW[1] & ~SW[0]

HEX0[3] Karnaugh Map

	[2]/[1]			
[0]	00	01	11	10
0	0	0	0	1
1	1	0	1	0

HEX0[3] = (~SW[2] & ~SW[1] & SW[0]) | (SW[2] & SW[1] & SW[0]) | (SW[2] & ~SW[1] & ~SW[0])

HEX0[4] Karnaugh Map

	[2]/[1]			
[0]	00	01	11	10
0	0	0	0	1
1	1	1	1	1

HEX0[4] = SW[0] | (SW[2] & ~SW[1])

HEX0[5] Karnaugh Map

	[2]/[1]			
[0]	00	01	11	10
0	0	1	0	0
1	1	1	1	0

HEX0[5] = (~SW[2] & SW[0]) | (~SW[2] & SW[1]) | (SW[1] & SW[0])

HEX0[6] Karnaugh Map

	[2]/[1]			
[0]	00	01	11	10
0	1	0	0	0
1	1	0	1	0

HEX0[6] = (~SW[2] & ~SW[1]) | (SW[2] & SW[1] & SW[0])

```
// Single 7 Segment LED Display
//
module seg7 (
      input [2:0] SW,
      output [6:0] HEX0,
      output [0:0] LEDG
);

assign HEX0[0] = (~SW[2] & ~SW[1] & SW[0]) | (SW[2] & ~SW[1] &
~SW[0]);
assign HEX0[1] = (SW[2] & SW[1] & ~SW[0]) | (SW[2] & ~SW[1] & SW[0]);
assign HEX0[2] = ~SW[2] & SW[1] & ~SW[0];
assign HEX0[3] = (~SW[2] & ~SW[1] & SW[0]) | (SW[2] & SW[1] & SW[0]) |
(SW[2] & ~SW[1] & ~SW[0]);
assign HEX0[4] = SW[0] | (SW[2] & ~SW[1]);
assign HEX0[5] = (~SW[2] & SW[0]) | (~SW[2] & SW[1]) | (SW[1] &
SW[0]);
assign HEX0[6] = (~SW[2] & ~SW[1]) | (SW[2] & SW[1] & SW[0]);

endmodule
```

Alternatively, we could use a 3 to 8 decoder with a conditional operator:

```
// Single 7 Segment LED Display
//
module seg7 (
      input [2:0] SW,
      output [6:0] HEX0
);

      assign HEX0 =
      (SW == 3'b000) ? 7'b100_0000 :
      (SW == 3'b001) ? 7'b111_1001 :
      (SW == 3'b010) ? 7'b010_0100 :
      (SW == 3'b011) ? 7'b011_0000 :
      (SW == 3'b100) ? 7'b001_1001 :
      (SW == 3'b101) ? 7'b001_0010 :
      (SW == 3'b110) ? 7'b000_0010 :
      (SW == 3'b111) ? 7'b111_1000 : 7'b100_0000 ;

endmodule
```

Sequential Logic

In order to define devices like latches, flip-flops and counters, we need to save state information. In order to do so, we'll be using additional components of Verilog

Reg & Wire

http://www.asic-world.com/tidbits/wire_reg.html

Wire = A connection. You cannot assign a value to a wire, ie a clock or a connection from a gate to another gate. Primarily used to connect combinational logic.

Reg = something you can assign a value to. Good for combinational and sequential logic.

Reg can be combinational in an 'Always' statement. This is good if you want to test the reg value.

Reg can be sequential, however you must test for it with posedge in an 'Always' statement.

Initial & Always

These are procedural blocks in Verilog.

Initial - Is executed only once when a circuit starts up.

Always - Is executed continuously.

Begin & End

These are used when the Initial or Always segments are longer than a single statement. A 'case' with all its' conditionals counts as a single statement.

For example, here's our 7 segment using 'always' with a case statement. Note that since we are using an output inside the 'always', it must be declared as a 'reg' within the original definition:

```
// Single 7 Segment LED Display via 3:8 decoder and 'always' case
//
module seg7 (
      input [2:0] SW,
      output reg [6:0] HEX0
);

      always
            case (SW)
                  3'b000 : HEX0 = 7'b100_0000;
                  3'b001 : HEX0 = 7'b111_1001;
                  3'b010 : HEX0 = 7'b010_0100;
                  3'b011 : HEX0 = 7'b011_0000;
                  3'b100 : HEX0 = 7'b001_1001;
                  3'b101 : HEX0 = 7'b001_0010;
                  3'b110 : HEX0 = 7'b000_0010;
                  3'b111 : HEX0 = 7'b111_1000;
                  default: HEX0 = 7'b100_0000;
            endcase

endmodule
```

Blocking and Non-blocking Assignments

Within an 'Always' procedural block, you can assign values either in sequence or in parallel. These are also called blocking and non-blocking. For example:

```
always begin
      a=b;        // blocking assignment, b is copied to a
      b=a;        // b has now been copied to both a and b
end
```

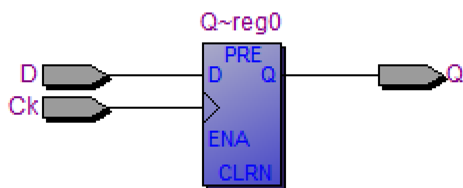
```
always begin
      a<=b;       // non-blocking assignment
      b<=a;       // values of a and b have been swapped
end
```

D Flip Flop

This device is triggered on the edge of the clock, aka edge triggered.

```
// D flip flop
//
module dff (
      input D, Ck,
      output reg Q
);

      always @(posedge Ck)
            Q = D;

endmodule
```

Here's the RTL result:

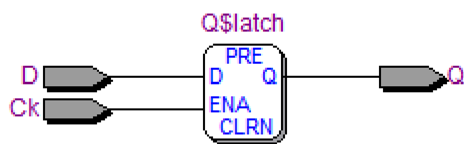


D Latch

Interestingly enough, a D latch is more difficult to program than a D flip flop. It is level as opposed to edge triggered on the clock.

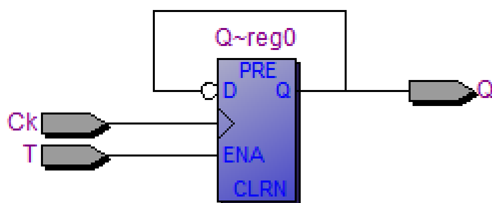
```
// D latch
//
module dlatch (
      input D, Ck,
      output reg Q
);

      always @(D, Ck)
            if (Ck) Q = D;

endmodule
```

Again, the RTL result:



T Flip Flop

```
// T flip flop
//
module tff (
      input T, Ck,
      output reg Q
);

      always @(posedge Ck)
            if (T) Q = ~Q;
endmodule
```



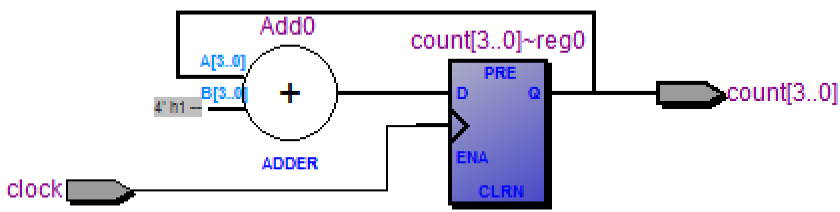
An 8 Bit Register

It's easy to extent a D flip-flop to a full blown register in Verilog. Let's also add some more control as well.

```
// 8 bit register
//
module register8 (
      input [7:0] data,
      input clock, set, reset,
      output reg [7:0] q
);

      always @(posedge clock) begin
            if (set)
                  q = 8'b1111111;
            else if (reset)
                  q = 8'b0;
            else
                  q = data;
            end

endmodule
```

A 4 Bit Counter

```
// A 4 bit counter
//
module counter4 (
      input clock,
      output reg [3:0] count
);

initial
      count = 0;

always @(posedge clock)
      count = count + 1;

endmodule
```



A Blinking LED With Instantiated Counter

CLOCK_50 is the 50Mhz clock from the .qsf file. You should map ‘LED’ to an LED obviously.

The slow clock frequency is (5*(10^6))/(2^26) = .745 Hz

We could have counted and compared to a specific decimal number in order to get a 1Hz slow clock, but this makes for simpler logic.

```
// Blinking LED with implicit instantiated 50MHz clock
//
module blinky (
           input CLOCK_50,
           output [0:0]LEDG
     );

     wire SlowClock;                        // Slow clock signal

   SlowIt(CLOCK_50, SlowClock);             // Implicit instantiation
   assign LEDG[0] = SlowClock;
endmodule


module SlowIt(input FastClock, output reg SlowClock);
     reg [25:0]R;

     always @(posedge FastClock) begin
           R = R + 1;
           SlowClock = R[25];         //(50*10^6)/(2^26) or .745 Hz
     end

endmodule
```

```
// Blinking multi-LED with 50MHz clock
//
module ledcounter(
     input CLOCK_50,
     output reg [7:0] LEDG
);

  reg [27:0] count1;

always @(posedge CLOCK_50) begin
  LEDG <= count1[27:20];              // Green LED's are counting
  count1 <= count1 + 1;               // Non-blocking assignments
end

endmodule
```

Using a Simulator (Modsim)

By this point, we've spent a lot of time creating circuits, using the assignment editor and programming the DE1. Let's move on to a simulator, so we can see a greater level of detail within the circuit via waveforms, as opposed to just blinky LED's. Before moving on, please go through the tutorial at:

http://ecee.colorado.edu/~ecen2350/AlteraSoftware/ModelSim_Session01.html

This is a relatively simple example, and it includes a separate file (also called testbench) that provides a clock/counter as input for the design.

Alternatively, there's chapters 1-6 of the Modsim tutorial at:

http://www.usna.edu/EE/ee362/LABS/modelsim_tut.pdf

My First Modsim Example

When we compiled a blinky LED for programming on the DE1, we used the CLOCK_50 pin to get our clock signal. Since this isn't available in the simulator, we'll use a different method. First, we set the overall timescale of the clock with **`timescale 1ns/1ns**. Then we enter a delay to create our clock cycles with **#5**. In this case, that would be 5ns.

You can also use a #value to set propagation delay on other gates as well.

Once you have completed the first tutorial, try out the following (simpler) code:

```
`timescale 1ns/1ns

module blinky (output reg myclk, output LED);

      assign LED = myclk;

      initial begin
            myclk = 0;
      end

      always begin
      #5 myclk = ~myclk;
end

endmodule
```

You should be able to:

1. Create a new project directory
2. Run Modsim and Create a new project in this directory
3. Give the Library a name
4. Add one or more verilog files to your project
5. Add content to the files (i.e. above example)
6. Compile it

7. Add waves for the input and output
8. Run the simulation and watch the output.

Separate the Testbench and the device under test

Let's separate the testing component of the program from the program itself. Our testbench will provide the inputs for the device under test, also called the DUT.

Here's the Truth Table for our new device.

Inputs			Output	???
a	**b**	**c**	**y**	
0	0	0	0	1
0	0	1	x	0
0	1	0	0	0
0	1	1	1	0
1	0	0	x	1
1	0	1	1	1
1	1	0	1	0
1	1	1	0	0

The Karnaugh Map

	a/b			
c	00	01	11	10
0	0	0	1	x
1	x	1	0	1

y = (~a.c) + (a.~c) + (a.~b)

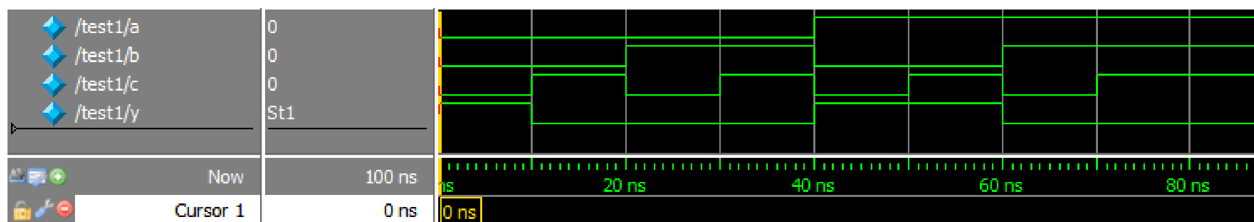
The code in '**verilog.v**' will look like this:

```
module verilog (
  input a, b, c,
  output y
);

assign y = ~a & c |
           a & ~c |
           a & ~b;

endmodule
```

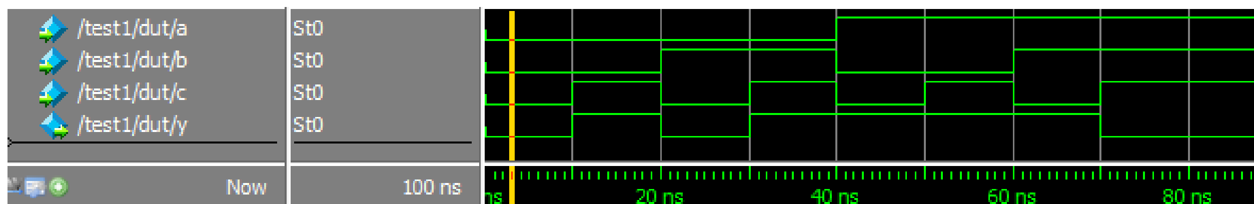
Let's test this with a file called test1.v as follows:

```
`timescale 1ns/1ns

module test1 ();

  reg a, b, c;
  wire y;

verilog dut (a, b, c, y);

initial begin
  a = 0; b = 0; c = 0; #10;
  c = 1;               #10;
  b = 1; c = 0;        #10;
  c = 1;               #10;
  a = 1; b = 0; c = 0; #10;
  c = 1;               #10;
  b = 1; c = 0;        #10;
  c = 1;               #10;
end

endmodule
```

Waveform output should NOT look like this:



You should have selected the dut for the waveform and show this:

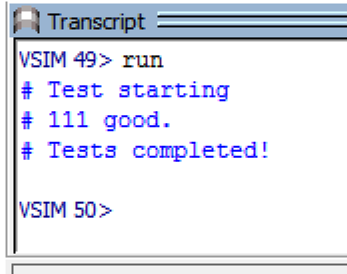


By running your cursor along the timescale, you should see that the **/test1/dut/y** output will match the truth table for each of the a/b/c values of 000 through 111.

Add some 'IF' testing

```
`timescale 1ns/1ns

module test2 ();

  reg a, b, c;
  wire y;

  verilog dut (a, b, c, y);

  initial begin

  $display ("Test starting");

  a = 0; b = 0; c = 0; #10;
  if (y !== 0) $display ("000 failed.");

  c = 1;               #10;
//  if (y !== 0) $display ("001 failed.");  // This is a don't care

  b = 1; c = 0;        #10;
  if (y !== 0) $display ("010 failed.");

  c = 1;               #10;
  if (y !== 1) $display ("011 failed.");

  a = 1; b = 0; c = 0; #10;
//  if (y !== 1) $display ("100 failed.");  // Another don't care

  c = 1;               #10;
  if (y !== 1) $display ("101 failed.");

  b = 1; c = 0;        #10;
  if (y !== 1) $display ("110 failed.");

  c = 1;               #10;
  if (y !== 0) $display ("111 failed.");
  if (y == 0) $display ("111 good.");

  $display ("Tests completed!");
end

endmodule
```

When running the test2 scenario, in addition to the waveform, you'll see the results of your testing in the Transcript window as follows:



References

http://www.ee.ed.ac.uk/~gerard/Teach/Verilog/manual/index.html

http://vol.verilog.com/VOL/main.htm

http://www.asic-world.com/verilog/veritut.html

http://electrosofts.com/verilog/

http://sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guid...

http://www.verilogwiki.info/wiki/index.php/Tutorials

http://www.swarthmore.edu/NatSci/echeeve1/Class/e15/QQS_V/QuickQuartusVerilog.html#Adding_A_Predefined_Circuit_Element

http://www.cc.gatech.edu/~milos/Teaching/CS3220F2011/Slides

Books

Digital Design and Computer Architecture by David Money Harris & Sarah L. Harris

FSM in Verilog

http://inst.eecs.berkeley.edu/~cs150/sp12/resources/FSM.pdf

http://www.asic-world.com/tidbits/verilog_fsm.html