Display Driver Integration and Timing Measurement

Vincent Martin
TUID: 913012274
ECE 2613
Lab #: 7 (10/15/2012)

Introduction:
The objective of this lab is to use instantiation to integrate the seven segment decoder module that we created in our previous lab into the display driver module that we have created during our class lectures. In addition to this the clock on the hardware will be divided to time a module that will allow more than one LED display digit to be used at a time.

This will be accomplished by using a concept known as instantiation. Instantiation allows us to both integrate modules together and also easily resuse previously written verilog code for one module many times.

For example if we coded a module called 5_bit_counter for a project that required three of them, we would think that perhaps we would have to rewrite this module three times. Instantiation makes this not true, we could reuse the code we wrote by instantiating the module as three different names, a module of the type 5_bit_counter called A, B and C.

Instantiation Applied to Our Design
In Lab 7 we will not be using instantiation in order to create multiple instances. Instead we will be using it in order to use one instance of each of our modules, the dsp_drvr and the svn_seg_decoder module.

The top level of our design will consist of the sw_core which inside will include an instantiation of the dsp_drvr which we will label as udd. Inside of the udd instance we will find yet another instantiation, this time called U1 of the svn_seg_decoder variety.

Applying the Theory to Verilog:
In order to transfer our understanding of theory to verilog code we will have to understand to use the following syntax to instantiate each module. Below I will include the two lines which I had to add, however, for a more detailed understanding of how this fits into the design you will have to read all of the verilog source that will be included in this report.

In our sw_core module we will use the following line to instantiate the udd module as udd:

```
dsp_drvr udd(.digit0(4'b0001), .digit1(4'b0010), .digit2(4'b0011),.digit3(4'b0100),
                .rst(m_rst), .clk(m_clk), .cath(cathode), .an(anode));
```

In our udd module we will use the following line to instantiate the svn_seg_decoder module as U1.

svn_seg_decoder U1(.display_on(1), .bcd_in(mux_digit), .seg_out(seg_out));

Additionally we will want to implement a testing module. This scheme will utilize a .txt file, which will allow us to test all of our expected outcomes for our design.

Procedures:

Create the Modules in the Project

1. Connect to the design server using no machine.
2. Execute XISE
3. Open the Lab7 project in *~/xilinx/lab7*
4. Create a new source for module sw_top with the following parameters
    a. input run
    b. input m_rst
    c. input m_clk
    d. output 7 bit bus cathode
    e. output 4 bit bus anode
    f. output m_sec
5. Import a copy of the svn_seg_decoder.v file from the previous lab.
6. Instantiate udd of the dsp_drvr type in the sw_core.
7. Instantiate U1 of the svn_seg_decoder type in the udd module.

Test the Design with iSim

1. Switch to Simulation mode by clicking on
    a. View:Simulation
        i. Xc3s500e-4fg320.
        ii. tb_sw_core
2. Run iSim simulator by clicking on
    a. iSim Simulator
    b. Right click Simulate Behavioral Model and then run.
3. Once iSim runs, verify that the Mismatch—index messages match what you are expecting in your test bench text file.
4. If the results are not what you expect either edit your module code or your test bench code and then attempt to test again.
5. In the console window execute the command *run all* and verify that you receive output that matches your expected input digits to the dsp_drvr module.

Alter the Digits to be Displayed to Verify Proper Function of the Display

1. Alter the sw_core.v file so that it is passing a different set of values to the udd instance of the dsp_dvr to display an alternative series of values. I used 9999.

2. Use the *run all* command inside of the console window of iSim to determine if this caused the new values were displayed.

Results:
Below is iSim output from an input of 4321 and 9999 which will display these numbers via the dsp_drvr and an oscilliscope image representing our timing.

Display 4321:

Console

```
ISim O.76xd (signature 0x8ddf5b5d)
This is a full version of ISim.
WARNING:  For instance udd/U1/, width 1 of formal port display_on is not equal to width 32 of actual constant.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
Anodes start - time = 20 ns
anodes: 0111 seg_out: 1001111
ISim> run all
Anodes changed - time = 1000010 ns
anodes: 1011 seg_out: 1001111
Anodes changed - time = 2000010 ns
anodes: 1101 seg_out: 0100100
Anodes changed - time = 3000010 ns
anodes: 1110 seg_out: 0000110
Anodes changed - time = 4000010 ns
anodes: 0111 seg_out: 0001011
Anodes changed - time = 5000010 ns
anodes: 1011 seg_out: 1001111
Simulation complete!!!
Stopped at time : 5000010 ns : File "/home/students/tuc56100/xilinx/lab7/tb_sw_core.v" Line 60
ISim>
```

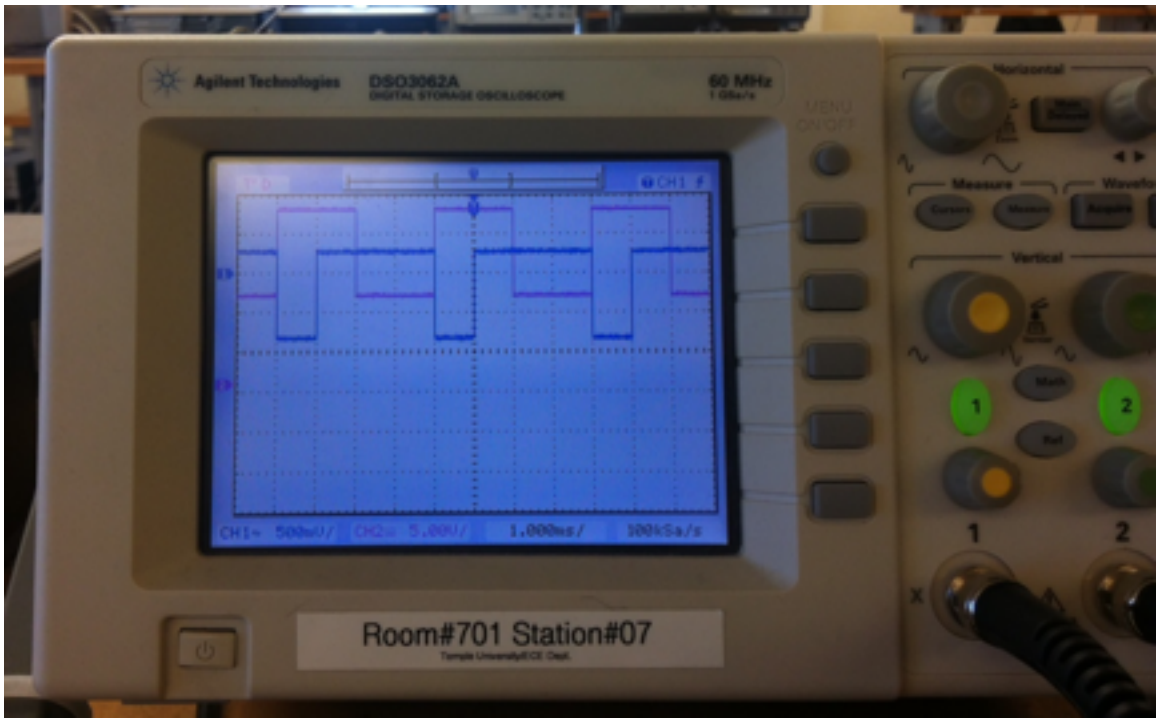
Display 9999:

Console

```
ISim O.76xd (signature 0x8ddf5b5d)
This is a full version of ISim.
WARNING:  For instance udd/U1/, width 1 of formal port display_on is not equal to width 32 of actual constant.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
Anodes start - time = 20 ns
anodes: 0111 seg_out: 0000010
ISim> run all
Anodes changed - time = 1000010 ns
anodes: 1011 seg_out: 0000010
Anodes changed - time = 2000010 ns
anodes: 1101 seg_out: 0000010
Anodes changed - time = 3000010 ns
anodes: 1110 seg_out: 0000010
Anodes changed - time = 4000010 ns
anodes: 0111 seg_out: 0000010
Anodes changed - time = 5000010 ns
anodes: 1011 seg_out: 0000010
Simulation complete!!!
Stopped at time : 5000010 ns : File "/home/students/tuc56100/xilinx/lab7/tb_sw_core.v" Line 60
ISim>
```

Photo from Oscilliscope:
In addition to our results we can also use an oscilliscope to view different anode timing. The way to set it up and the results of the setup can be seen below.

The cables on the left are hooked to ground. On the right each is attached to an anode.



In this image we can see the delay between two different anodes.

Discussion:
This lab was a good introduction on dividing timing. As well as tying different modules together. It is the begining of the creation of our final project which will be a stop watch. This project has concepts above mentioned that will be used in the future project.

In regards to the hold switch. It appears that when enabled it will cause only the 0 anode to be lit instead of cycling through all of them. So in our case it would be a 1 on the display. Below is the code that supports that that was found in dsp_drvr.v

```
// anode control counter logic
    always @(anreg, rst, m_sec) begin
        // defaults
        next_anreg = anreg;     // hold count
        // regular logic
        if (m_sec == 1) next_anreg = anreg + 1;
        // priority logic
        if (rst == 1) next_anreg = 0;
    end     // end of anode combinational logic
```

During the instantiation of my modules I had some problems. I spent a few hours dealing with errors below.

ERROR:HDLCompilers:27 - "dsp_drvr.v" line 30 Illegal redeclaration of 'mux_digit'
ERROR:HDLCompilers:27 - "dsp_drvr.v" line 32 Illegal redeclaration of 'seg_out'

This was caused due to the placement in the dsp_drvr.v file that I attempted to instantiate my svn_seg_decoder module. Interestingly enough this worked fine during the simulation but began to fail once I tried to put the created binary on the board.

After some struggling I found that I had to move the following line below to a lower part of the source code. This will be the primary code that is included, however, this report will also include the dsp_drvr.v file that was causing me problems. In this file you will find the line that caused the problem in **bold**.

```
// Instantiation of our svn_seg_decoder module as U1.
svn_seg_decoder U1(.display_on(1), .bcd_in(mux_digit), .seg_out(seg_out));
```

This problem leaves me with a question that I have yet to get answered and must put some thought into. Why does it matter where I attempt to instantiate this? It was my understanding that everything is to be run in parallel, ergo the order of these statements should in theory not matter. The simulation proves this, however real application to hardware causes problems. Perhaps it does technically work, but the xISE package has some limitations in regards to my hardware.

Source Code:

sw_core.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:14:01 10/12/2012
// Design Name:
// Module Name:    sw_core
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module sw_core(
    input run,
    input m_rst,
    input m_clk,
    output [6:0] cathode,
    output [3:0] anode,
    output m_sec
    );

//Instantiate the dsp_driver module as udd with kin of crazy format
//let us see that the spaces do not matter in this language.
dsp_drvr udd(.digit0(4'b0001),
                                        .digit1(4'b0010),
                                        .digit2(4'b0011),
                                        .digit3(4'b0100),
                                        .rst(m_rst),
                                        .clk(m_clk),
                                        .cath(cathode),
                                        .an(anode));

endmodule
```

dsp_drvr.v

```
// lab7 : version 10/09/2012
//
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////////////////////////
module dsp_drvr(
    input clk,
    input [3:0] digit0,
    input [3:0] digit1,
    input [3:0] digit2,
    input [3:0] digit3,
    input rst,
    output reg [3:0] an,
    output [6:0] cath,
    output reg m_sec
            );

            // register for dividing by 50000
            reg [15:0] count, next_count;
            // register for anode control
            reg [1:0] anreg, next_anreg;
            // for multiplexed input into the decoder
            reg [3:0] mux_digit;
            // output from the decoder
            wire [6:0] seg_out;

            // sequential block
            always @(posedge clk) begin
                        count <= next_count;
                        anreg <= next_anreg;
            end       // end of always block

            // combinational logic for count
            always @(count, rst) begin
                        // take care of defaults
                        next_count = count - 1;
                        m_sec = 0;

                        // normal logic
                        if (count == 0) begin
                                    next_count = 49999;
                                    m_sec = 1;
                        end
                        // priority logic
                        if (rst == 1) next_count = 49999;
            end       // end of count combinational logic

            // anode control counter logic
            always @(anreg, rst, m_sec) begin
                        // defaults
                        next_anreg = anreg;       // hold count
                        // regular logic
                        if (m_sec == 1) next_anreg = anreg + 1;
                        // priority logic
                        if (rst == 1) next_anreg = 0;
```

```
        end     // end of anode combinational logic

        // now do two muxes: digit mux and an signals
        always @(anreg,digit0,digit1,digit2,digit3) begin
                an = 4'b1111;   // default
                case (anreg)
                        0: begin
                                mux_digit = digit0;
                                an[3] = 0;
                        end
                        1: begin
                                mux_digit = digit1;
                                an[2] = 0;
                        end
                        2: begin
                                mux_digit = digit2;
                                an[1] = 0;
                        end
                        3: begin
                                mux_digit = digit3;
                                an[0] = 0;
                        end
                endcase
        end     // end of combinational mux

        // finally instantiate the decoder here
        svn_seg_decoder U1(.display_on(1), .bcd_in(mux_digit), .seg_out(seg_out));
        assign cath = ~seg_out; // invert

endmodule
```

svn_seg_decoder.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:16:21 09/11/2012
// Design Name:            seven segment decoder module
// Module Name:    svn_seg_decoder
// Project Name:   lab 03 seven segment decoder
// Target Devices:  xilinx board
// Tool versions:
// Description: Take in 4 bits as a descriptor of the number you want to show and also
// 1bit as an on/off switch and then output the appropriate signal to drive
// the seven segment display.
//
// Dependencies:
//
// Revision: 2 (now using case statement)
// Revision 0.01 - File Created
```

```
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module svn_seg_decoder(
    input [3:0] bcd_in,
    input display_on,
    output reg [6:0] seg_out
    );



// My always logic to replace old code
always @(display_on, bcd_in[3] or bcd_in[2] or bcd_in[1] or bcd_in[0]) begin



        case({display_on,bcd_in[3],bcd_in[2],bcd_in[1],bcd_in[0]})


        // Takes into account all combinations with the display turned off

                //Number 0000 Decimal Val: 0 Display: on *
                5'b10000: seg_out = 7'b0111111;

                //Number 0001 Decimal Val: 1 Display: on *
                5'b10001: seg_out = 7'b0110000;

                //Number 0010 Decimal Val: 2 Display: on *
                5'b10010: seg_out = 7'b1011011;

                //Number 0011 Decimal Val: 3 Display: on
                5'b10011: seg_out = 7'b1111001;

                //Number 0100 Decimal Val: 4 Display: on
                5'b10100: seg_out = 7'b1110100;

                //Number 0101 Decimal Val: 5 Display: on
                5'b10101: seg_out = 7'b1101101;

                //Number 0110 Decimal Val: 6 Display: on
                5'b10110: seg_out = 7'b1101111;

                //Number 0111 Decimal Val: 7 Display: on
                5'b10111: seg_out = 7'b0111000;

                //Number 1000 Decimal Val: 8 Display: on
                5'b11000: seg_out = 7'b1111111;

                //Number 1001 Decimal Val: 9 Display: on
                5'b11001: seg_out = 7'b1111101;
```

```
                default: seg_out = 7'b0000000;

        endcase


end

endmodule
```

<Broken as seen in Discussion Section> dsp_drvr.v

```
//
// lab7 : version 10/09/2012
//
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
module dsp_drvr(
    input clk,
    input [3:0] digit0,
    input [3:0] digit1,
    input [3:0] digit2,
    input [3:0] digit3,
    input rst,
    output reg [3:0] an,
    output [6:0] cath,
    output reg m_sec
        );


        // Instantiation of our svn_seg_decoder module as u1.
        svn_seg_decoder U1(.display_on(1), .bcd_in(mux_digit), .seg_out(seg_out));


        // register for dividing by 50000
        reg [15:0] count, next_count;
        // register for anode control
        reg [1:0] anreg, next_anreg;
        // for multiplexed input into the decoder
        reg [3:0] mux_digit;
        // output from the decoder
        wire [6:0] seg_out;

        // sequential block
        always @(posedge clk) begin
                count <= next_count;
                anreg <= next_anreg;
        end     // end of always block

        // combinational logic for count
        always @(count, rst) begin
```

```
		// take care of defaults
		next_count = count - 1;
		m_sec = 0;

		// normal logic
		if (count == 0) begin
			next_count = 49999;
			m_sec = 1;
		end
		// priority logic
		if (rst == 1) next_count = 49999;
	end	// end of count combinational logic

	// anode control counter logic
	always @(anreg, rst, m_sec) begin
		// defaults
		next_anreg = anreg;	// hold count
		// regular logic
		if (m_sec == 1) next_anreg = anreg + 1;
		// priority logic
		if (rst == 1) next_anreg = 0;
	end	// end of anode combinational logic

	// now do two muxes: digit mux and an signals
	always @(anreg,digit0,digit1,digit2,digit3) begin
		an = 4'b1111;	// default
		case (anreg)
			0: begin
				mux_digit = digit0;
				an[3] = 0;
			end
			1: begin
				mux_digit = digit1;
				an[2] = 0;
			end
			2: begin
				mux_digit = digit2;
				an[1] = 0;
			end
			3: begin
				mux_digit = digit3;
				an[0] = 0;
			end
		endcase
	end	// end of combinational mux

	// finally instantiate the decoder here

	assign cath = ~seg_out; // invert

endmodule
```