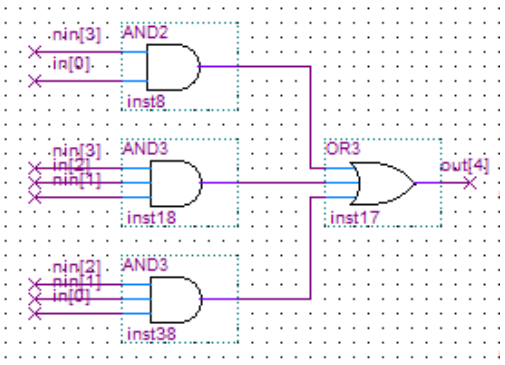
Experiment 1

Saturday, November 18, 2017 15:36

This experiment is quite simple so I haven't really included many notes.

Essentially, we just completed the "partially" completed" version of the 7 Segment Decoder by adding the schematic for circuit to produce **out[4]**. This is shown in the figure below.



We then created the bitstream (i.e. the .sof file) and used the programmer to transfer it to the FPGA to make sure that out design was working properly.

Next, we used the "TimeQuest Timing Analyzer" to analyze the timing constraints of the design for different temperatures.

Propagation Delay

	Input Port	Output Port	RR	RF	FR	FF
1	SW[0]	HEX0[0]	8.469	8.618	8.981	9.138
2	SW[0]	HEX0[1]	8.669	9.017	9.181	9.583
3	SW[0]	HEX0[2]		8.896	9.197	
4	SW[0]	HEX0[3]	8.113	8.212	8.621	8.774
5	SW[0]	HEX0[4]	8.886			9.712
6	SW[0]	HEX0[5]	8.771			9.718
7	SW[0]	HEX0[6]	8.209	8.226	8.720	8.745
8	SW[1]	HEX0[0]	8.127	8.297	8.669	8.821
9	SW[1]	HEX0[1]	8.422	8.719	8.970	9.244
10	SW[1]	HEX0[2]	8.335			9.095
11	SW[1]	HEX0[3]	7.910	7.958	8.461	8.486
12	SW[1]	HEX0[4]		8.871	9.086	
13	SW[1]	HEX0[5]	8.530	8.861	9.079	9.387
14	SW[1]	HEX0[6]	7.867	7.905	8.409	8.429
15	SW[2]	HEX0[0]	8.669	8.893	9.202	9.479
16	SW[2]	HEX0[1]	9.144			9.960
17	SW[2]	HEX0[2]	8.889	9.177	9.422	9.763
18	SW[2]	HEX0[3]	8.585	8.610	9.118	9.151
19	SW[2]	HEX0[4]	9.086	9.467	9.619	10.053
20	SW[2]	HEX0[5]	9.246	9.554	9.779	10.095
21	SW[2]	HEX0[6]	8.409	8.501	8.942	9.087
22	SW[3]	HEX0[0]	8.277	8.488	8.746	8.905
23	SW[3]	HEX0[1]	8.572	8.917	9.186	9.448
24	SW[3]	HEX0[2]	8.505	8.780	8.975	9.198
25	SW[3]	HEX0[3]	8.012	8.108	8.626	8.639
26	SW[3]	HEX0[4]		9.068	9.169	
27	SW[3]	HEX0[5]	8.673	9.052	9.287	9.583
28	SW[3]	HEX0[6]	8.013	8.092	8.483	8.510

Propagation Delay

	Input Port	Output Port	RR	RF	FR	FF
1	SW[0]	HEX0[0]	8.919	9.101	9.334	9.522
2	SW[0]	HEX0[1]	9.183	9.545	9.606	10.002
3	SW[0]	HEX0[2]		9.427	9.595	
4	SW[0]	HEX0[3]	8.574	8.686	8.996	9.142
5	SW[0]	HEX0[4]	9.369			10.131
6	SW[0]	HEX0[5]	9.275			10.136
7	SW[0]	HEX0[6]	8.648	8.690	9.064	9.112
8	SW[1]	HEX0[0]	8.592	8.789	9.053	9.236
9	SW[1]	HEX0[1]	8.925	9.240	9.387	9.683
10	SW[1]	HEX0[2]	8.845			9.557
11	SW[1]	HEX0[3]	8.369	8.434	8.835	8.882
12	SW[1]	HEX0[4]		9.398	9.504	
13	SW[1]	HEX0[5]	9.024	9.382	9.487	9.827
14	SW[1]	HEX0[6]	8.322	8.379	8.784	8.827
15	SW[2]	HEX0[0]	9.172	9.418	9.605	9.884
16	SW[2]	HEX0[1]	9.666			10.398
17	SW[2]	HEX0[2]	9.437	9.750	9.870	10.216
18	SW[2]	HEX0[3]	9.054	9.103	9.482	9.536
19	SW[2]	HEX0[4]	9.622	10.026	10.055	10.492
20	SW[2]	HEX0[5]	9.757	10.099	10.186	10.533
21	SW[2]	HEX0[6]	8.902	9.008	9.334	9.473
22	SW[3]	HEX0[0]	8.783	9.015	9.151	9.333
23	SW[3]	HEX0[1]	9.121	9.478	9.629	9.910
24	SW[3]	HEX0[2]	9.057	9.356	9.427	9.676
25	SW[3]	HEX0[3]	8.508	8.615	9.016	9.047
26	SW[3]	HEX0[4]		9.630	9.608	
27	SW[3]	HEX0[5]	9.213	9.613	9.720	10.044
28	SW[3]	HEX0[6]	8.508	8.600	8.877	8.919

This first data sheet shows the timings for a "Slow 1100mV 0C Model". The second data sheet shows the timings for a "Slow 1100mV 85C Model". There are several interesting points to mention here:

1. The "Slow" essentially tells the software to give us the worst case delay between the cause and the effect.
2. For each combination of input/output there are 4 timings (i.e. RR, RF, FR, FF) these show the time delay between a RISE/FALL in the input signal to a RISE/FALL in the output signal.
3. For some combinations of signals some data is missing (e.g. **SW[0]** and **HEX0[2]** do not have data for RR). There is no data for these combination of signals because a RISE/FALL in the input never causes a RISE/FALL in the output
4. As we increase the temperature, the time delay increases. This is what we would expect since at higher temperatures transistors electron mobility decreases due to an increase in

resistance and since the voltage is kept constant, more time is required to gather this energy.

Finally, we confirmed that the design was using the expected amount of resources (i.e. 4 ALMs and 11 I/O pins). This can be seen in the Figure below.

Fitter Status	Successful - Sat Nov 18 15:57:34 2017
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	ex1_top
Top-level Entity Name	ex1_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	4 / 32,070 (< 1 %)
Total registers	0
Total pins	11 / 457 (2 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total RAM Blocks	0 / 397 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Overall, this experiment helped us realize the limits of designing hardware with schematic diagrams.

Experiment 2

Saturday, November 18, 2017 16:16

This experiment is again quite simple since we are essentially just designing the same circuits but with a hardware description language instead of a schematic diagram.

Essentially, we just copied the code representing the decoder can compiled it. The code is shown in the Figure bellow.

```
module hex_to_7seg (out, in);

 output [6:0] out;
 input [3:0] in;

 reg [6:0]out;

 always @ (*)
   case(in)
     4'h0:out = 7'b1000000;
     4'h1:out = 7'b1111001;
     4'h2:out = 7'b0100100;
     4'h3:out = 7'b0110000;
     4'h4:out = 7'b0011001;
     4'h5:out = 7'b0010010;
     4'h6:out = 7'b0000010;
     4'h7:out = 7'b1111000;
     4'h8:out = 7'b0000000;
     4'h9:out = 7'b0011000;
     4'ha:out = 7'b0001000;
     4'hb:out = 7'b0000011;
     4'hc:out = 7'b1000110;
     4'hd:out = 7'b0100001;
     4'he:out = 7'b0000110;
     4'hf:out = 7'b0001110;
     endcase
endmodule
```

We then downloaded the **pin_assignment.txt** file and copied the contents in to our Quartus Setting File (i.e. **ex2_top.qsf**).

Finally, we compiled the bitstream for our designed made sure it worked on the FPGA. We copied our **hex_to_7seg.v** file into a folder called **mylib** for future use.

Overall, this experiment allowed us the see the benefits of designing a digital system with a Hardware Description Language such as Verilog rather than a schematic diagram.

Experiment 3

Saturday, November 18, 2017 17:11

For this design we created a Top-Level entity called **ex3_top.v** which used the 7 segment decoder that we created in the previous experiment. The Verilog code for the Top-Level entity is shown below.

```
1   module ex3_top(SW, HEX0, HEX1, HEX2);
2
3    input [9:0] SW;
4    output [6:0] HEX0;
5    output [6:0] HEX1;
6    output [6:0]HEX2;
7
8
9    hex_to_7seg SEG0(HEX0, SW[3:0]);
10   hex_to_7seg SEG1(HEX1, SW[7:4]);
11   hex_to_7seg SEG2(HEX2, {2'b00,SW[9:8]});
12
13   endmodule
14
```

Effectively, we create 3 instances of the **hex_to_7seg** module. However, in the last instance we have to concatenate 2 zero bits in order to create the 4 bit input signal. The design will also compile and work without the concatenation; however, we though that it was best to include it for clarity.

We compiled the designed and tested it and it worked as expected.

Overall, this experiment underscored the power of using an HDL language such as Verilog since your design can easily be extended by adding a couple of lines of code.

Experiment 4

Saturday, November 18, 2017 17:20

This was an optional experiment; however, we decided to do it as well.

We wrote the **bin2bcd_10.v** which is shown in the Figure below.

```
module bin2bcd_10(B, BCD_0, BCD_1, BCD_2, BCD_3);

    input [9:0] B;
    output [3:0] BCD_0, BCD_1, BCD_2, BCD_3;

    wire [3:0] w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12;

    wire[3:0] a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12;

    add3_ge5 A1 (w1,a1);
    add3_ge5 A2 (w2,a2);
    add3_ge5 A3 (w3,a3);
    add3_ge5 A4 (w4,a4);
    add3_ge5 A5 (w5,a5);
    add3_ge5 A6 (w6,a6);
    add3_ge5 A7 (w7,a7);
    add3_ge5 A8 (w8,a8);
    add3_ge5 A9 (w9,a9);
    add3_ge5 A10 (w10,a10);
    add3_ge5 A11 (w11,a11);
    add3_ge5 A12 (w12,a12);


    assign w1 = {1'b0, B[9:7]};
    assign w2 = {a1[2:0], B[6]};
    assign w3 = {a2[2:0], B[5]};
    assign w4 = {1'b0, a1[3], a2[3], a3[3]};
    assign w5 = {a3[2:0], B[4]};
    assign w6 = {a4[2:0], a5[3]};
    assign w7 = {a5[2:0], B[3]};
    assign w8 = {a6[2:0], a7[3]};
    assign w9 = {a7[2:0], B[2]};
    assign w10 = {1'b000, a4[3], a6[3], a8[3]};
    assign w11 = {a8[2:0], a9[3]};
    assign w12 = {a9[2:0], B[1]};

    assign BCD_0 = {a12[2:0], B[0]};
    assign BCD_1 = {a11[2:0], a12[3]};
    assign BCD_2 = {w10[2:0], a11[3]};
    assign BCD_3 = {(3'b000), a10[3]};

endmodule
```

We then create a top level design (i.e. **ex4_top.v**) which initially used the **bin2bcd_10.v** to convert the 10bit binary input to the decimal output and we recorded the resources that it used.

Flow Status	Successful - Sun Nov 19 12:56:48 2017
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	ex4_top
Top-level Entity Name	ex4_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	37 / 32,070 (< 1 %)
Total registers	0
Total pins	45 / 457 (10 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

The top level design was:

```
1   module ex4_top(SW, HEX0, HEX1, HEX2, HEX3, HEX4);
2
3      input [9:0] SW;
4      output [6:0] HEX0;
5      output [6:0] HEX1;
6      output [6:0] HEX2;
7      output [6:0] HEX3;
8      output [6:0] HEX4;
9
10     wire [3:0] BCD0;
11     wire [3:0] BCD1;
12     wire [3:0] BCD2;
13     wire [3:0] BCD3;
14     //wire [3:0] BCD4;
15
16
17
18     bin2bcd_10 SEG0(SW[9:0], BCD0[3:0], BCD1[3:0], BCD2[3:0], BCD3[3:0]);
19
20     hex_to_7seg SEG1(HEX0[6:0], BCD0[3:0]);
21     hex_to_7seg SEG2(HEX1[6:0], BCD1[3:0]);
22     hex_to_7seg SEG3(HEX2[6:0], BCD2[3:0]);
23     hex_to_7seg SEG4(HEX3[6:0], BCD3[3:0]);
24     hex_to_7seg SEG5(HEX4[6:0], 4'b0);
25
26  endmodule
```

Next, we did the same thing using the **bin2bcd_16.v** to verify that Quartus will use the same resources due to optimizations.

The top level design was:

```
 1    module ex4_top(SW, HEX0, HEX1, HEX2, HEX3, HEX4);
 2
 3       input [9:0] SW;
 4       output [6:0] HEX0;
 5       output [6:0] HEX1;
 6       output [6:0] HEX2;
 7       output [6:0] HEX3;
 8       output [6:0] HEX4;
 9
10       wire [3:0] BCD0;
11       wire [3:0] BCD1;
12       wire [3:0] BCD2;
13       wire [3:0] BCD3;
14       wire [3:0] BCD4;
15
16
17
18       bin2bcd_16 SEG0(SW[9:0], BCD0[3:0], BCD1[3:0], BCD2[3:0], BCD3[3:0], BCD4[3:0]);
19
20       hex_to_7seg SEG1(HEX0[6:0], BCD0[3:0]);
21       hex_to_7seg SEG2(HEX1[6:0], BCD1[3:0]);
22       hex_to_7seg SEG3(HEX2[6:0], BCD2[3:0]);
23       hex_to_7seg SEG4(HEX3[6:0], BCD3[3:0]);
24       hex_to_7seg SEG5(HEX4[6:0], BCD4[3:0]);
25
26    endmodule
```

After compiling the new design, we verified that it used the same resources.

Flow Status	Successful - Sun Nov 19 12:36:46 2017
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	ex4_top
Top-level Entity Name	ex4_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	37 / 32,070 (< 1 %)
Total registers	0
Total pins	45 / 457 (10 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Effectively, the optimization that Quartus performs allows us to keep the highest bit module since we can use it with any number of bits that is lower than out initial design.

Experiment 5

Sunday, November 19, 2017 13:05

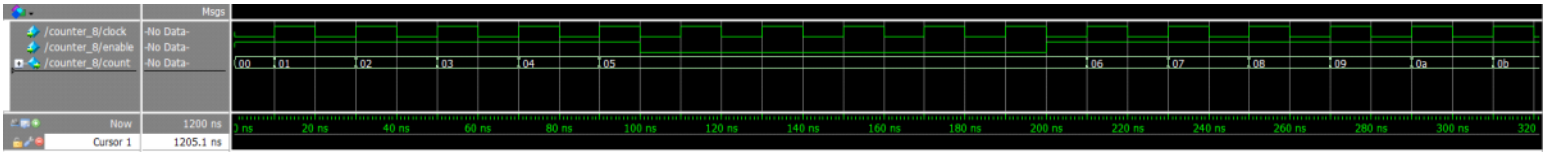
We copied the **counter_8.v** code from the experiment handout and set **counter_8.v** as out top level design. The Verilog code is shown below.

```
`timescale 1ns / 100ps  //unit time is 1ns, resolutions 100ps

module counter_8(
   clock,
   enable,
   count
);

   parameter BIT_SZ = 8;
   input clock;
   input enable;
   output [BIT_SZ-1:0] count;

   reg [BIT_SZ-1:0] count;

   initial count = 0;

   always @ (posedge clock)
      if (enable == 1'b1)
      count <= count + 1'b1;

endmodule
```

We then ran ModelSim and created the do file (i.e. **tb_counter.do**) in order to test out design. The **tb_counter.do** code is shown below.

```
add wave clock enable
add wave -hexadecimal count
force clock 0 0, 1 10ns -repeat 20ns
force enable 1
run 100ns
force enable 0
run 100ns
force enable 1
run 1000ns
```

The part waveform from the simulations is shown below:



From the waveform we can deduce that the counter seems to be working!

Overall, this experiment helped us understand how to use ModelSim and the "step" command in order to debug and test out designs.

Experiment 6

Sunday, November 19, 2017 13:55

We modified the **counter_8.v** to extend it to a 16-bit counter. The code for **counter_16.v** is shown below.

```
1   `timescale 1ns / 100ps  //unit time is 1ns, resolutions 100ps
2
3   module counter_16(
4       clock,
5       enable,
6       reset,
7       count
8   );
9
10      parameter BIT_SZ = 16;
11      input clock;
12      input enable;
13      input reset;
14      output [BIT_SZ-1:0] count;
15
16      reg [BIT_SZ-1:0] count;
17
18      initial count = 0;
19
20      always @ (posedge clock)
21      begin
22          if (reset == 1'b1)
23              count <= 1'b0;
24          else if (enable == 1'b1)
25              count <= count + 1'b1;
26      end
27
28  endmodule
```

Next, we added all the modules that our top level design will depend on to the project. The top level design (i.e. **ex6_top.v**) is shown below.

```
module ex6_top(
    KEY,
    CLOCK_50,
    HEX0,
    HEX1,
    HEX2,
    HEX3,
    HEX4
);

    input [3:0] KEY;
    input CLOCK_50;

    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;

    wire [15:0] count;

    wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;


    counter_16 COUNTER(CLOCK_50, ~KEY[0], ~KEY[1], count[15:0]);

    bin2bcd_16 BIN2BCD(count, BCD0, BCD1, BCD2, BCD3, BCD4);

    hex_to_7seg HEX_TO_7SEG1(HEX0, BCD0);
    hex_to_7seg HEX_TO_7SEG2(HEX1, BCD1);
    hex_to_7seg HEX_TO_7SEG3(HEX2, BCD2);
    hex_to_7seg HEX_TO_7SEG4(HEX3, BCD3);
    hex_to_7seg HEX_TO_7SEG5(HEX4, BCD4);

endmodule
```

Next, we create the Synopsis Delay Constraint file (i.e. **ex6_top.sdc**) which defines the clock frequency that our design uses (i.e. 50MHz). The SDC file is shown below.

```
create_clock -name "CLOCK_50" -period 20.000ns [get_ports    {CLOCK_50}]
```

After generating the TImeQuest analysis, we can find the FMAX for LOW and HIGHT temperatures (i.e. 0C, 85C). The maximum frequencies are shown below.

Slow 1100mV 0C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	422.12 MHz	422.12 MHz	CLOCK_50	

Slow 1100mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	444.64 MHz	444.64 MHz	CLOCK_50	

What is interesting here, is that the Fmax for a higher temperature is higher than the Fmax for a lower temperature. According to our conclusions form Exercise 1, we would expect the opposite to happen.

In fact, there are 2 factors that affect the Fmax. One is the Resistance of the circuit which will increase with temperature causing the overall electron mobility to decrease. Another is the threshold voltage of

the transistor which also varies with temperature.

In this case, we can say that the overall effect of the threshold voltage difference between the 2 temperatures had a greater effect on the Fmax than the increase in resistance which leads to these results.

From this report, we can also see the slack times for both the Setup and Hold times. Since neither of the slack times are zero, we can conclude that the "bottleneck" of the circuit is the actual clock.

Slow 1100mV 85C Model Setup Summary

	Clock	Slack	End Point TNS
1	CLOCK_50	17.751	0.000

Slow 1100mV 85C Model Hold Summary

	Clock	Slack	End Point TNS
1	CLOCK_50	0.368	0.000

Slow 1100mV 0C Model Setup Summary

	Clock	Slack	End Point TNS
1	CLOCK_50	17.631	0.000

Slow 1100mV 0C Model Hold Summary

	Clock	Slack	End Point TNS
1	CLOCK_50	0.369	0.000

The TimeQuest report is red because we haven't provided constraints for the various inputs and outputs to the circuit (i.e. the KEYS and the HEX display). As we can see from the figure bellow, TimeQuest does not have any information for these inputs/outputs.

Unconstrained Input Ports

	Input Port	Comment
1	KEY[0]	No input delay, min/max delays, false-path exceptions, or max skew assignments found
2	KEY[1]	No input delay, min/max delays, false-path exceptions, or max skew assignments found

Out design uses the following resources:

Flow Summary	
Flow Status	Successful - Tue Nov 21 10:10:42 2017
Quartus Prime Version	16.0.0 Build 211 04/27/2016 SJ Standard Edition
Revision Name	ex6_top
Top-level Entity Name	ex6_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	99 / 32,070 (< 1 %)
Total registers	16
Total pins	38 / 457 (8 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

As we can see from the figure above, we are using 16 registers because of the 16 bit counter and 38 pins because we have 2 Keys as inputs and 5 7-segment displays (i.e. 2 + 7 * 5 = 37)

Furthermore, if we go to **tools > Technology Map Viewer (Post-Mapping)** we could theoretically understand where exactly out design uses all the ALMs.

We compiled the .sof for the Cascade Counter. It worked! :D

The **clktick_16.v** and the **ex6_top.v** are shown below.

```
module clktick_16(
clkin,
tick);

    parameter N_BIT = 16;
    parameter N = 49999;

    input clkin;
    output tick;

    reg [N_BIT-1:0] count;
    reg tick;

    initial tick = 1'b0;
    initial count = 16'b0;

    always @ (posedge clkin)
    begin
       if (count == 0)
       begin
          tick <= 1'b1;
          count <= N;
       end
       else
       begin
          tick <= 1'b0;
          count  <= count + 1'b1;
       end
    end

endmodule
```

```
module ex6_top(
    KEY,
    CLOCK_50,
    HEX0,
    HEX1,
    HEX2,
    HEX3,
    HEX4
);

    input [1:0] KEY;
    input CLOCK_50;

    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;

    wire [15:0] count;
    wire tick, enable;

    wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;


    counter_16 COUNTER(CLOCK_50, enable, ~KEY[1], count[15:0]);

    clktick_16 TICKER(CLOCK_50, tick);

    bin2bcd_16 BIN2BCD(count, BCD0, BCD1, BCD2, BCD3, BCD4);

    hex_to_7seg HEX_TO_7SEG1(HEX0, BCD0);
    hex_to_7seg HEX_TO_7SEG2(HEX1, BCD1);
    hex_to_7seg HEX_TO_7SEG3(HEX2, BCD2);
    hex_to_7seg HEX_TO_7SEG4(HEX3, BCD3);
    hex_to_7seg HEX_TO_7SEG5(HEX4, BCD4);

    assign enable = (tick & ~KEY[0]);

endmodule
```

Experiment 7

Sunday, November 19, 2017 15:54

Here is the code:

```
1   //polynomial 1 + X + X^7
2
3   module LFSR_7(data_out, clk);
4
5
6      input clk;
7      output [7:1] data_out;
8
9      reg [7:1] sreg;
10
11     initial sreg = 7'b1;
12
13     always @ (posedge clk)
14     begin
15        sreg <= {sreg[6:1], sreg[1] ^ sreg[7]};
16     end
17
18     assign data_out = sreg;
19
20  endmodule
```

```
1   module ex7_top(
2   KEY,
3   HEX0,
4   HEX1
5   );
6
7      input [3:0] KEY;
8      output [6:0] HEX0, HEX1;
9
10     wire [6:0] data_out;
11
12     LFSR_7 LFSR(data_out, ~KEY[3]);
13
14     hex_to_7seg SEG0(HEX0, data_out[3:0]);
15     hex_to_7seg SEG1(HEX1, {1'b0, data_out[6:4]});
16
17  endmodule
```

It worked!

Experiment 8

Friday, December 1, 2017 10:26

This is the code:

ex8_top.v | delay.v

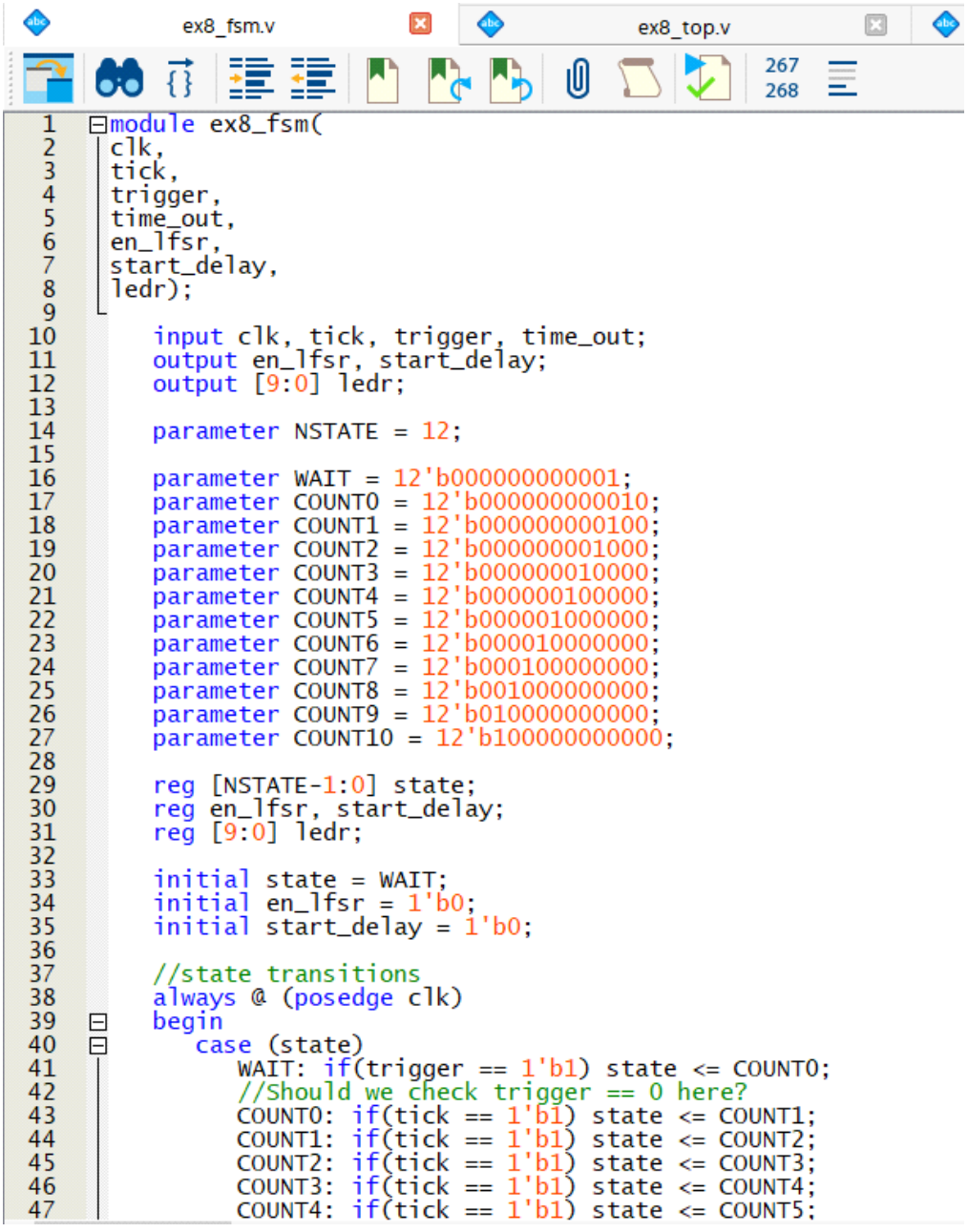
```
module ex8_top(
CLOCK_50,
KEY,
HEX0,
HEX1,
HEX2,
HEX3,
HEX4,
LEDR);

    input CLOCK_50;
    input [3:0] KEY;

    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;
    output [9:0] LEDR;

    wire CLOCK_1MS, TICK_500MS;
    wire time_out;
    wire en_lfsr, start_delay;
    wire [5:0] RANDOM;
    wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;

    clkdiv_16 DIV0(CLOCK_50, CLOCK_1MS, 16'd50000, 1'b1);
    clktick_16 TICK0(CLOCK_1MS, TICK_500MS, 16'd500, 1'b1);
    LFSR_6 LFSR(en_lfsr, CLOCK_1MS, RANDOM);

    ex8_fsm FSM(CLOCK_1MS, TICK_500MS, ~KEY[3], time_out, en_lfsr, start_delay, LEDR[9:0]);

    delay D0((RANDOM * 250), CLOCK_1MS, start_delay, time_out);

    bin2bcd_16 BIN2BCD((RANDOM * 250), BCD0, BCD1, BCD2, BCD3, BCD4);
    hex_to_7seg SG0(HEX0, BCD0);
    hex_to_7seg SG1(HEX1, BCD1);
    hex_to_7seg SG2(HEX2, BCD2);
    hex_to_7seg SG3(HEX3, BCD3);
    hex_to_7seg SG4(HEX4, BCD4);

endmodule
```

ex8_top.v | delay.v

```
1  //polynomial 1 + X + X^6
2
3  module LFSR_6(en, clk, prbs);
4
5
6     input clk;
7     input en;
8     output [6:1] prbs;
9
10    reg [6:1] sreg;
11
12    initial sreg = 7'b1;
13
14    always @ (posedge clk)
15    begin
16       if(en == 1'b1)
17          sreg <= {sreg[5:1], sreg[1] ^ sreg[6]};
18    end
19
20    assign prbs = sreg;
21
22 endmodule
23
```

ex8_top.v | delay.v

```
1  module delay(N, clk, trigger, time_out);
2
3     input [15:0] N;
4     input clk;
5     input trigger;
6
7     output time_out;
8
9     reg [15:0] count;
10    reg time_out;
11
12    initial count = 16'b0;
13    initial time_out = 1'b0;
14
15    always @ (posedge clk)
16    begin
17       if(trigger == 1'b1)
18       begin
19          if(count == N)
20             time_out <= 1'b1;
21          else
22             count <= count + 1'b1;
23       end
24       else
25       begin
26          count <= 0;
27          time_out <= 0;
28       end
29    end
30
31 endmodule
```




```
1  module ex8_fsm(
2  clk,
3  tick,
4  trigger,
5  time_out,
6  en_lfsr,
7  start_delay,
8  ledr);
9
10    input clk, tick, trigger, time_out;
11    output en_lfsr, start_delay;
12    output [9:0] ledr;
13
14    parameter NSTATE = 12;
15
16    parameter WAIT = 12'b000000000001;
17    parameter COUNT0 = 12'b000000000010;
18    parameter COUNT1 = 12'b000000000100;
19    parameter COUNT2 = 12'b000000001000;
20    parameter COUNT3 = 12'b000000010000;
21    parameter COUNT4 = 12'b000000100000;
22    parameter COUNT5 = 12'b000001000000;
23    parameter COUNT6 = 12'b000010000000;
24    parameter COUNT7 = 12'b000100000000;
25    parameter COUNT8 = 12'b001000000000;
26    parameter COUNT9 = 12'b010000000000;
27    parameter COUNT10 = 12'b100000000000;
28
29    reg [NSTATE-1:0] state;
30    reg en_lfsr, start_delay;
31    reg [9:0] ledr;
32
33    initial state = WAIT;
34    initial en_lfsr = 1'b0;
35    initial start_delay = 1'b0;
36
37    //state transitions
38    always @ (posedge clk)
39    begin
40       case (state)
41          WAIT: if(trigger == 1'b1) state <= COUNT0;
42          //Should we check trigger == 0 here?
43          COUNT0: if(tick == 1'b1) state <= COUNT1;
44          COUNT1: if(tick == 1'b1) state <= COUNT2;
45          COUNT2: if(tick == 1'b1) state <= COUNT3;
46          COUNT3: if(tick == 1'b1) state <= COUNT4;
47          COUNT4: if(tick == 1'b1) state <= COUNT5;
```

```
48          COUNT5: if(tick == 1'b1) state <= COUNT6;
49          COUNT6: if(tick == 1'b1) state <= COUNT7;
50          COUNT7: if(tick == 1'b1) state <= COUNT8;
51          COUNT8: if(tick == 1'b1) state <= COUNT9;
52          COUNT9: if(tick == 1'b1) state <= COUNT10;
53          COUNT10: if(time_out == 1'b1) state <= WAIT
54          default: ; //do nothing
55       endcase
56    end
57
58    //ouput combinational logic
59    always @ (*)
60    begin
61       case (state)
62          WAIT: begin
63                en_lfsr <= 1'b1;
64                start_delay <= 1'b0;
65                ledr <= 10'b0000000000;
66                end
67          COUNT0:  begin
68                   en_lfsr <= 1'b0;
69                   start_delay <= 1'b0;
70                   ledr <= 10'b0000000000;
71                   end
72          COUNT1:  begin
73                   en_lfsr <= 1'b0;
74                   start_delay <= 1'b0;
75                   ledr <= 10'b1000000000;
76                   end
77          COUNT2:  begin
78                   en_lfsr <= 1'b0;
79                   start_delay <= 1'b0;
80                   ledr <= 10'b1100000000;
81                   end
82          COUNT3:  begin
83                   en_lfsr <= 1'b0;
84                   start_delay <= 1'b0;
85                   ledr <= 10'b1110000000;
86                   end
87          COUNT4:  begin
88                   en_lfsr <= 1'b0;
89                   start_delay <= 1'b0;
90                   ledr <= 10'b1111000000;
91                   end
92          COUNT5:  begin
93                   en_lfsr <= 1'b0;
94                   start_delay <= 1'b0;
95                   ledr <= 10'b1111100000;
96                   end
97          COUNT6:  begin
98                   en_lfsr <= 1'b0;
99                   start_delay <= 1'b0;
100                  ledr <= 10'b1111110000;
101                  end
102         COUNT7:  begin
103                  en_lfsr <= 1'b0;
104                  start_delay <= 1'b0;
105                  ledr <= 10'b1111111000;
106                  end
107         COUNT8:  begin
108                  en_lfsr <= 1'b0;
109                  start_delay <= 1'b0;
110                  ledr <= 10'b1111111100;
111                  end
112         COUNT9:  begin
113                  en_lfsr <= 1'b0;
114                  start_delay <= 1'b0;
115                  ledr <= 10'b1111111110;
116                  end
117         COUNT10: begin
118                  en_lfsr <= 1'b0;
119                  start_delay <= 1'b1;
120                  ledr <= 10'b1111111111;
121                  end
122         default: ; //do nothing
123      endcase
124   end
```

```
121                 end
122                 default: ; //do nothing
123             endcase
124         end
125
126     endmodule
```

I used one hot encoding because the FPGA is a register rich architecture. Furthermore, I used a 6 bit LFSR since we effectively need $\frac{16}{0.25} = 64$ different values. Effectivelly, we only have 63 different values since we we will never have 0. As a result, the maximum delay will only go up to 15.75 seconds. We could add another bit and use a 7 bit LFSR to fix this; however, I didn't really think it was that big of a deal.

Delay is effectively 16 bit, we could have easily used a 7 bit counter but I don't think it matters that much since Quartus will just simplify the design.

Another issue is that in the top level design (i.e. **ex8_top**) we simply multiply the **RANDOM** 6 bit number by 250. A better solution would be to specifically use IP multiplier provided by Altera in order to multiply the numbers which would allow Quartus to use the space much more effectively.

Furthermore, the **clocktick_16** module has been modified so that its data input is reduced by 1. Thus for the tick 500 module the data input is 500 instead of 499.

Experiment 9

Friday, December 1, 2017 11:10

This is the code

ex9_top.v*

```
1  module ex9_top(
2  CLOCK_50,
3  KEY,
4  HEX0,
5  HEX1,
6  HEX2,
7  HEX3,
8  HEX4,
9  LEDR);
10
11     input CLOCK_50;
12     input [3:0] KEY;
13
14     output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;
15     output [9:0] LEDR;
16
17     wire CLOCK_1MS, TICK_500MS;
18     wire time_out;
19     wire en_lfsr, start_delay, en_counter, reset_counter;
20     wire [5:0] RANDOM;
21     wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;
22     wire [15:0] COUNT;
23
24     clkdiv_16 DIV0(CLOCK_50, CLOCK_1MS, 16'd50000, 1'b1);
25     clktick_16 TICK0(CLOCK_1MS, TICK_500MS, 16'd500, 1'b1);
26     LFSR_6 LFSR(en_lfsr, CLOCK_1MS, RANDOM);
27
28     ex9_fsm FSM(CLOCK_1MS, TICK_500MS, ~KEY[3], time_out, en_lfsr, start_delay, LEDR[9:0], ~KEY[0], en_counter,
29     reset_counter);
30
31     delay D0((RANDOM * 250), CLOCK_1MS, start_delay, time_out);
32     counter_16(CLOCK_1MS, en_counter, reset_counter, COUNT);
33
34     bin2bcd_16 BIN2BCD(COUNT, BCD0, BCD1, BCD2, BCD3, BCD4);
35     hex_to_7seg SG0(HEX0, BCD0);
36     hex_to_7seg SG1(HEX1, BCD1);
37     hex_to_7seg SG2(HEX2, BCD2);
38     hex_to_7seg SG3(HEX3, BCD3);
39     hex_to_7seg SG4(HEX4, BCD4);
40
41 endmodule
```

ex9_fsm.v

ex9_top.v*

```
module ex9_fsm(
clk,
tick,
trigger,
time_out,
en_lfsr,
start_delay,
ledr,
reaction,
en_counter,
reset_counter);

    input clk, tick, trigger, time_out, reaction;
    output en_lfsr, start_delay, en_counter, reset_counter;
    output [9:0] ledr;

    parameter NSTATE = 13;

    parameter WAIT = 13'b0000000000001;
    parameter COUNT0 = 13'b0000000000010;
    parameter COUNT1 = 13'b0000000000100;
    parameter COUNT2 = 13'b0000000001000;
    parameter COUNT3 = 13'b0000000010000;
    parameter COUNT4 = 13'b0000000100000;
    parameter COUNT5 = 13'b0000001000000;
    parameter COUNT6 = 13'b0000010000000;
    parameter COUNT7 = 13'b0000100000000;
    parameter COUNT8 = 13'b0001000000000;
    parameter COUNT9 = 13'b0010000000000;
    parameter COUNT10 = 13'b0100000000000;
    parameter WAIT_REACTION = 13'b1000000000000;

    reg [NSTATE-1:0] state;
    reg en_lfsr, start_delay, en_counter, reset_counter;
    reg [9:0] ledr;

    initial state = WAIT;
    initial en_lfsr = 1'b0;
    initial start_delay = 1'b0;

    //state transitions
    always @ (posedge clk)
    begin
        case (state)
            WAIT: if(trigger == 1'b1) state <= COUNT0;
            //Should we check trigger == 0 here?
            COUNT0: if(tick == 1'b1) state <= COUNT1;
```

```
48          COUNT1: if(tick == 1'b1) state <= COUNT2;
49          COUNT2: if(tick == 1'b1) state <= COUNT3;
50          COUNT3: if(tick == 1'b1) state <= COUNT4;
51          COUNT4: if(tick == 1'b1) state <= COUNT5;
52          COUNT5: if(tick == 1'b1) state <= COUNT6;
53          COUNT6: if(tick == 1'b1) state <= COUNT7;
54          COUNT7: if(tick == 1'b1) state <= COUNT8;
55          COUNT8: if(tick == 1'b1) state <= COUNT9;
56          COUNT9: if(tick == 1'b1) state <= COUNT10;
57          COUNT10: if(time_out == 1'b1) state <= WAIT_REACTION;
58          WAIT_REACTION: if(reaction == 1'b1) state <= WAIT;
59          default: ; //do nothing
60       endcase
61    end
62
63    //ouput combinational logic
64    always @ (*)
65    begin
66       case (state)
67          WAIT: begin
68                en_lfsr <= 1'b1;
69                start_delay <= 1'b0;
70                ledr <= 10'b0000000000;
71                en_counter <= 1'b0;
72                reset_counter <= 1'b0;
73                end
74          COUNT0:  begin
75                   en_lfsr <= 1'b0;
76                   start_delay <= 1'b0;
77                   ledr <= 10'b0000000000;
78                   en_counter <= 1'b0;
79                   reset_counter <= 1'b0;
80                   end
81          COUNT1:  begin
82                   en_lfsr <= 1'b0;
83                   start_delay <= 1'b0;
84                   ledr <= 10'b1000000000;
85                   en_counter <= 1'b0;
86                   reset_counter <= 1'b0;
87                   end
88          COUNT2:  begin
89                   en_lfsr <= 1'b0;
90                   start_delay <= 1'b0;
91                   ledr <= 10'b1100000000;
92                   en_counter <= 1'b0;
93                   reset_counter <= 1'b0;
94                   end
95          COUNT3:  begin
96                   en_lfsr <= 1'b0;
97                   start_delay <= 1'b0;
98                   ledr <= 10'b1110000000;
99                   en_counter <= 1'b0;
100                  reset_counter <= 1'b0;
101                  end
102         COUNT4:  begin
103                  en_lfsr <= 1'b0;
104                  start_delay <= 1'b0;
105                  ledr <= 10'b1111000000;
106                  en_counter <= 1'b0;
107                  reset_counter <= 1'b0;
108                  end
109         COUNT5:  begin
110                  en_lfsr <= 1'b0;
111                  start_delay <= 1'b0;
112                  ledr <= 10'b1111100000;
113                  en_counter <= 1'b0;
114                  reset_counter <= 1'b0;
115                  end
116         COUNT6:  begin
117                  en_lfsr <= 1'b0;
118                  start_delay <= 1'b0;
119                  ledr <= 10'b1111110000;
120                  en_counter <= 1'b0;
121                  reset_counter <= 1'b0;
122                  end
123         COUNT7:  begin
124                  en_lfsr <= 1'b0;
125                  start_delay <= 1'b0;
126                  ledr <= 10'b1111111000;
127                  en_counter <= 1'b0;
128                  reset_counter <= 1'b0;
129                  end
130         COUNT8:  begin
131                  en_lfsr <= 1'b0;
132                  start_delay <= 1'b0;
133                  ledr <= 10'b1111111100;
134                  en_counter <= 1'b0;
135                  reset_counter <= 1'b0;
136                  end
```

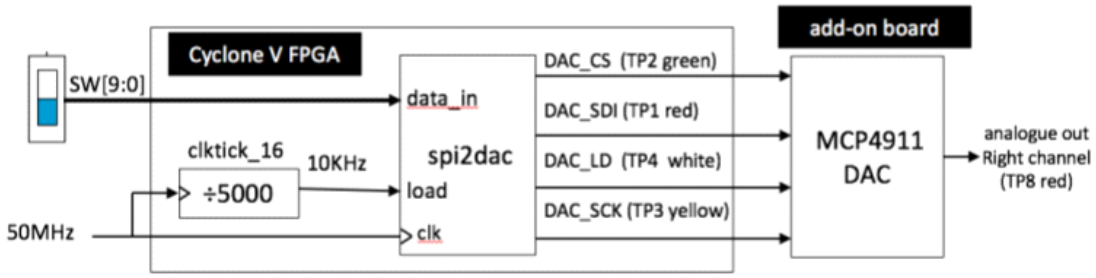
```
137             COUNT9:  begin
138                      en_lfsr <= 1'b0;
139                      start_delay <= 1'b0;
140                      ledr <= 10'b1111111110;
141                      en_counter <= 1'b0;
142                      reset_counter <= 1'b0;
143                      end
144             COUNT10: begin
145                      en_lfsr <= 1'b0;
146                      start_delay <= 1'b1;
147                      ledr <= 10'b1111111111;
148                      en_counter <= 1'b0;
149                      reset_counter <= 1'b1;
150                      end
151             WAIT_REACTION: begin
152                            en_lfsr <= 1'b0;
153                            start_delay <= 1'b0;
154                            ledr <= 10'b0000000000;
155                            reset_counter <= 1'b0;
156                            en_counter <= 1'b1;
157                            end
158             default: ; //do nothing
159         endcase
160     end
161
162 endmodule
```

Effectively, the only thing that changes is the top_level desing and the fsm. We also added a counter to count the number of ms until the user responds.

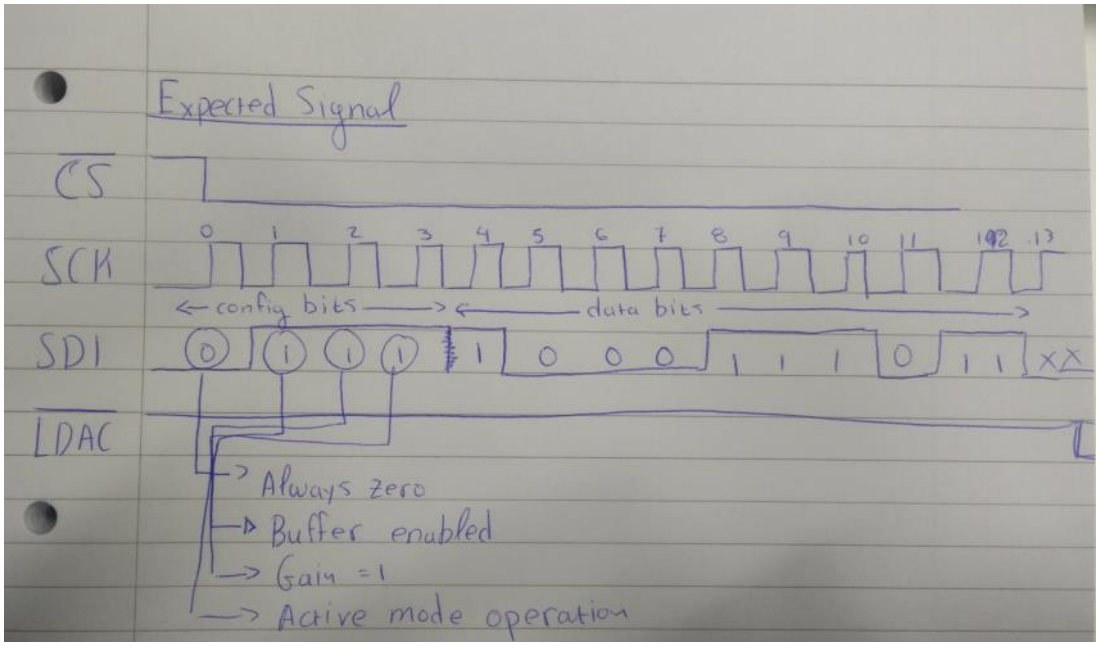
Experiment 10

Saturday, December 2, 2017 17:53

The Diagram is shown below.



Below you can see the expected timing diagram of the SPI interface signals when a word 10'h23b is sent to the DAC.



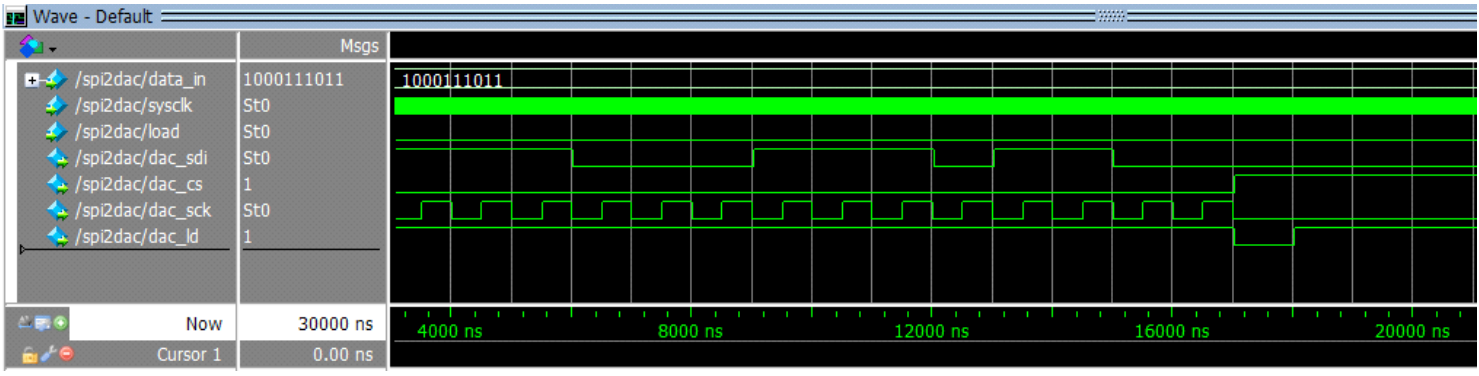
We can confirm this through the use of ModelSim.

We created a do file shown below

H:/VERI/PART3/ex10/simulation/modelsim/spi2dac.do - Default

```
Ln#
1  add wave data_in sysclk load dac_sdi dac_cs dac_sck dac_ld
2  force data_in 001000111011
3  force sysclk 0 0, 1 10ns -repeat 20ns
4  force load 0 0, 1 40ns, 0 80ns
5  run 30000ns
```

After running the do file, we generated the following diagram .

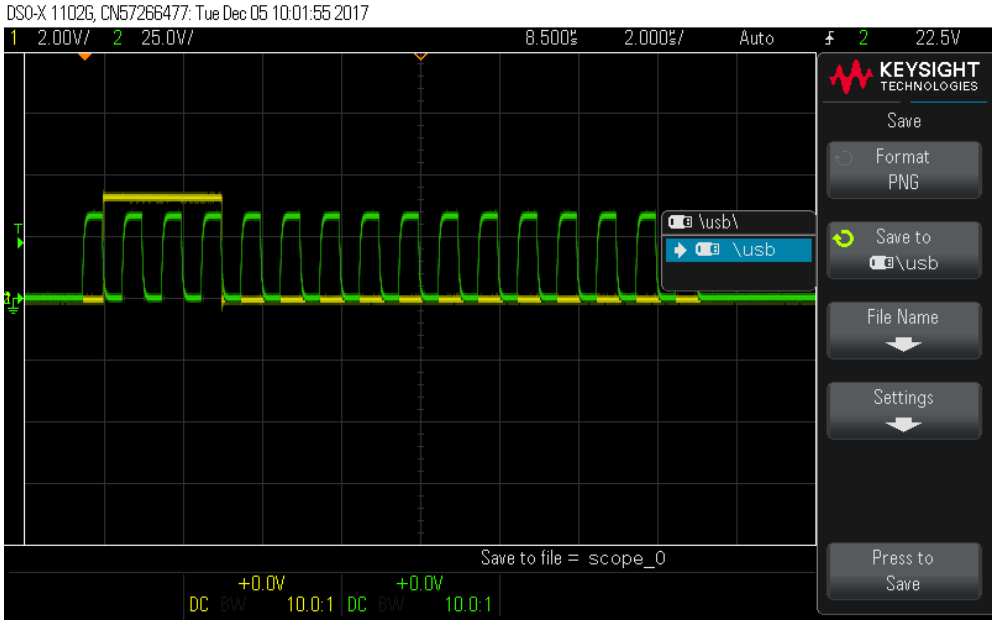


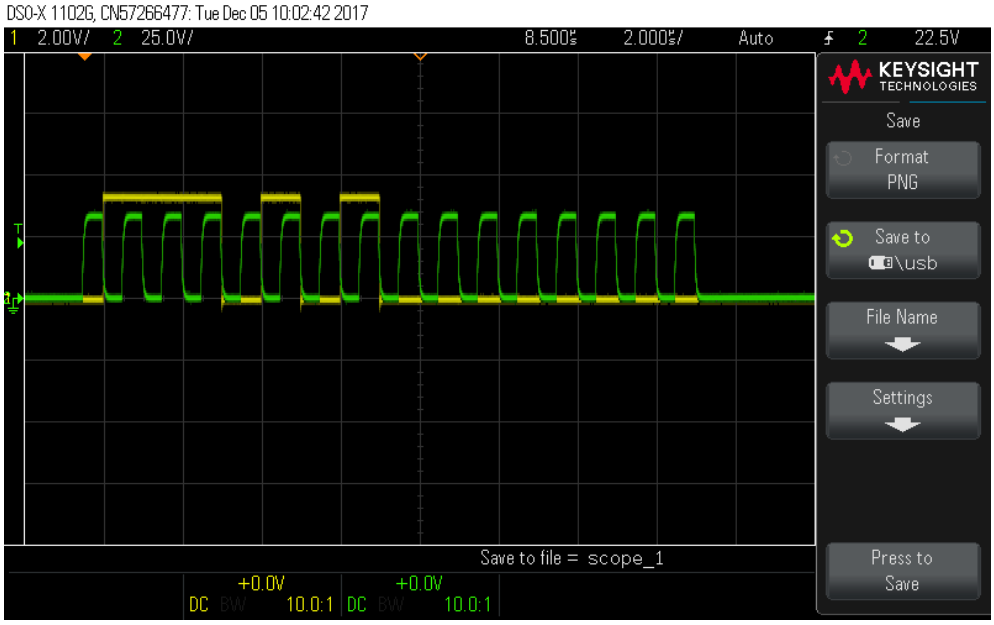
Which confirms our original sketch of the output waveforms.

We created the **ex_10_top.v** shown below.

```
module ex_10_top(SW, CLOCK_50, DAC_CS, DAC_SDI, DAC_LD, DAC_SCK
);

    input [9:0] SW;
    input CLOCK_50;
    output DAC_CS, DAC_SDI, DAC_LD, DAC_SCK;

    wire TICK_10KHz;

    clktick_16 OBJ1(CLOCK_50, TICK_10KHz, 16'd5000, 1'b1);

    spi2dac OBJ2(CLOCK_50, SW, TICK_10KHz, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);

endmodule
```

We can also confirm the ModelSim output through the use of the oscilloscope.





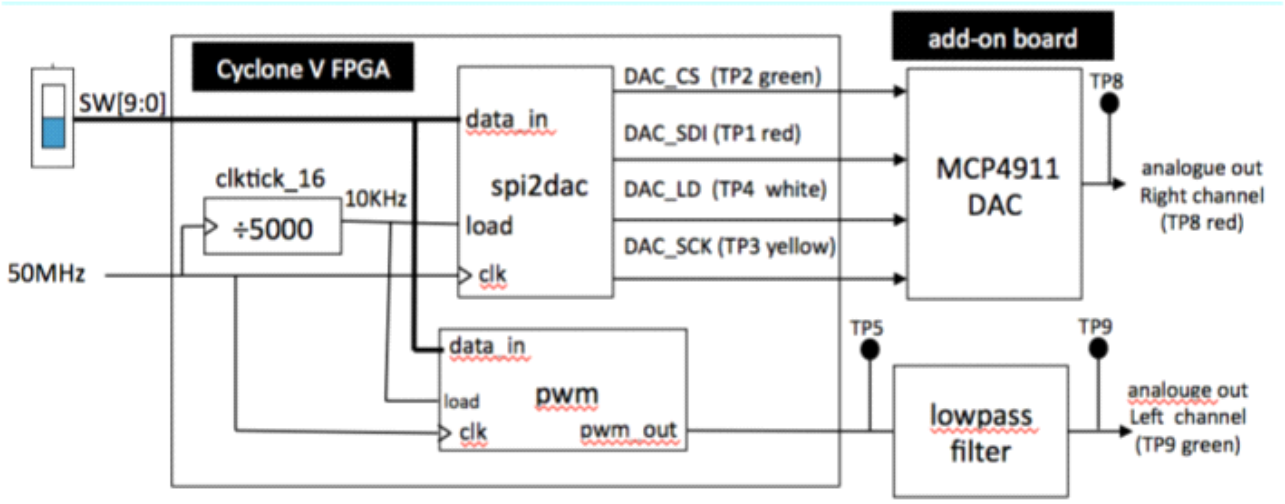
The green signal is the DAC_SCK and the yellow signal is the DAC_SDI. The first Image shows the output for a **SW[9:0] = 10'b0** and the second Image shows the output for a **SW[9:0] = 10'b0101000000**. These results confirm our expectations.

Looks Gucci.

Experiment 11

28 November 2017 11:15

The top level design is shown below.



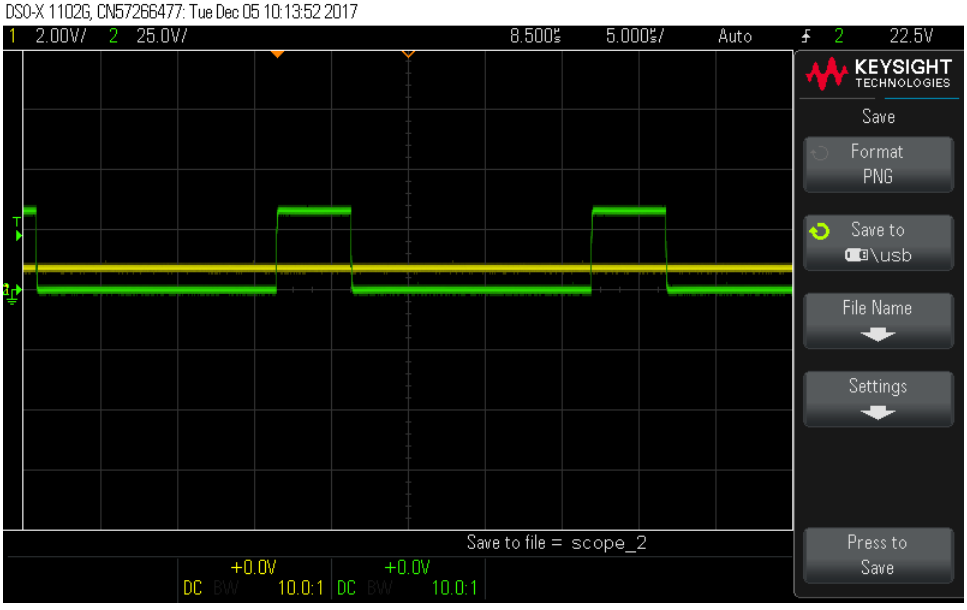

```
module ex11_top(SW, CLOCK_50, DAC_CS, DAC_SDI, DAC_LD, DAC_SCK, PWM_OUT
);

    input [9:0] SW;
    input CLOCK_50;
    output DAC_CS, DAC_SDI, DAC_LD, DAC_SCK;
    output PWM_OUT;

    wire TICK_10KHz;

    clktick_16 OBJ1(CLOCK_50, TICK_10KHz, 16'd5000, 1'b1);

    pwm OBJ2(CLOCK_50, SW, TICK_10KHz, PWM_OUT);

    spi2dac OBJ3(CLOCK_50, SW, TICK_10KHz, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);

endmodule
```

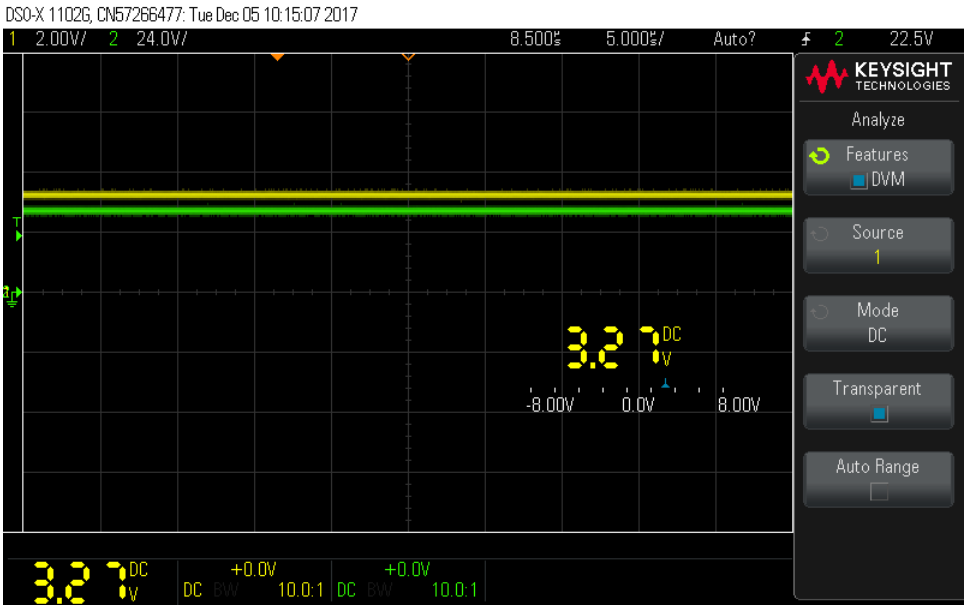
The code for the pwm module is shown below.

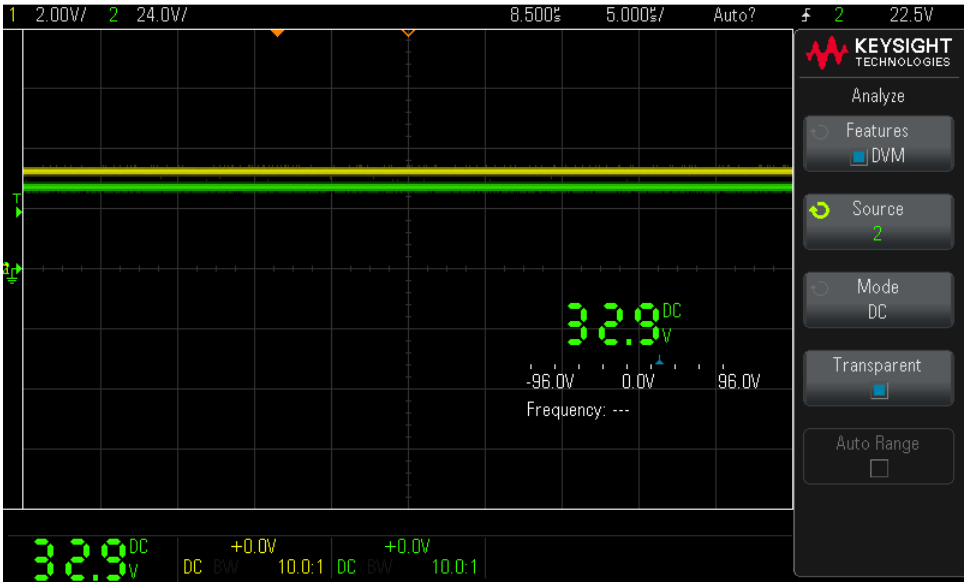
```
1  module pwm(clk, data_in, load, pwm_out);
2
3      input clk;
4      input [9:0] data_in;
5      input load;
6      output pwm_out;
7
8      reg [9:0] d;
9      reg [9:0] count;
10     reg pwm_out;
11
12     initial count = 10'b0;
13
14     always @ (posedge clk)
15        if (load == 1'b1) d <= data_in;
16
17     always @ (posedge clk) begin
18        count <= count + 1'b1;
19        if (count > d)
20           pwm_out <= 1'b0;
21        else
22           pwm_out <= 1'b1;
23        end
24
25 endmodule
```

The switches **SW[9:0]** effectively allow us to vary the duty cycle of the pulse width modulator. Which increases the voltage at **TP9** (i.e. after it pulse is passed though the integrator). The voltage at **TP9** varies from 0V to 3.3V.

We can confirm our expectations through the use of the oscilloscope.







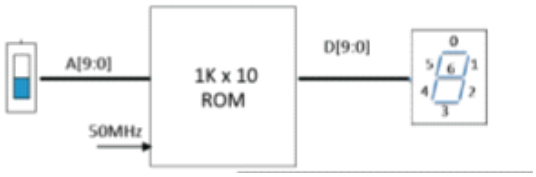
The first Image has tp5 (i.e. the duty cycle) on the green signal and TP8 (i.e. the analogue output from the DAC) on the yellow channel.

The second and third Image show TP9 (i.e. the low pass filter analogue output) on the green channel and TP8 (i.e. the analogue output from the DAC) on the yellow channel. As you can see from the 2 images, the max output from the low-pass filter is slightly higher than the max output from the DAC). However, effectively, the two signals will follow each other.

Experiment 12

Saturday, December 2, 2017 20:41

The Diagram is shown below.



The Code for the top level design is shown below.

```
module ex12_top(
CLOCK_50,
SW,HEX0,HEX1,HEX2,HEX3,HEX4
);
    input CLOCK_50;
    input [9:0] SW;
    output [6:0] HEX0;
    output [6:0] HEX1;
    output [6:0] HEX2, HEX3, HEX4;

    wire [9:0] ROM_OUT;
    wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;

    ROM OBJ1(SW, CLOCK_50, ROM_OUT);

    bin2bcd_16 OBJ2({6'b0,ROM_OUT}, BCD0, BCD1, BCD2, BCD3, BCD4);

    hex_to_7seg OBJ3(HEX0, BCD0);
    hex_to_7seg OBJ4(HEX1, BCD1);
    hex_to_7seg OBJ5(HEX2, BCD2);
    hex_to_7seg OBJ6(HEX3, BCD3);
    hex_to_7seg OBJ7(HEX4, BCD4);

endmodule
```

The Code for the ROM is shown below.

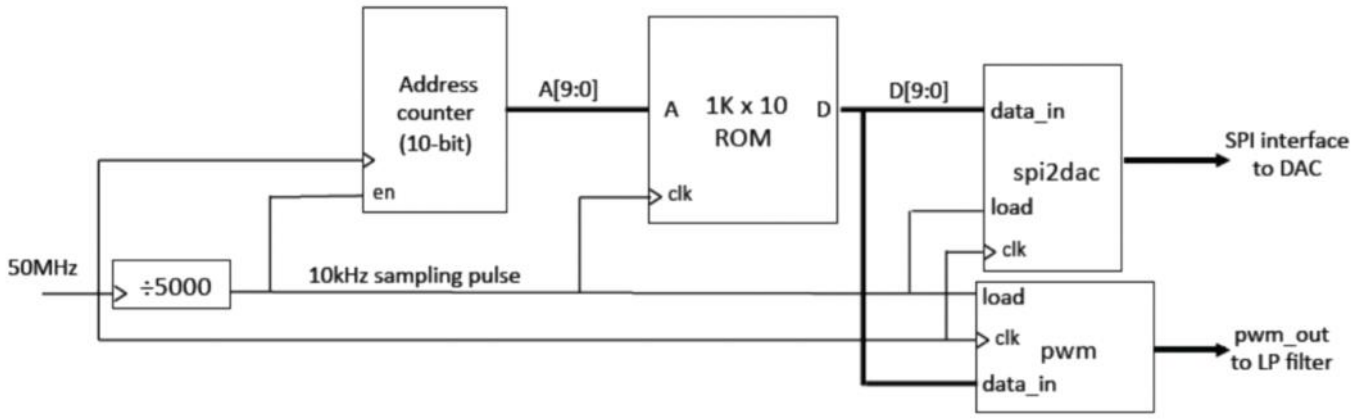
```
40  module ROM (
41     address,
42     clock,
43     q);
44
45     input [9:0]  address;
46     input   clock;
47     output   [9:0]  q;
48  `ifndef ALTERA_RESERVED_QIS
49  // synopsys translate_off
50  `endif
51     tri1    clock;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_on
54  `endif
55
56     wire [9:0] sub_wire0;
57     wire [9:0] q = sub_wire0[9:0];
58
59     altsyncram  altsyncram_component (
60               .address_a (address),
61               .clock0 (clock),
62               .q_a (sub_wire0),
63               .aclr0 (1'b0),
64               .aclr1 (1'b0),
65               .address_b (1'b1),
66               .addressstall_a (1'b0),
67               .addressstall_b (1'b0),
68               .byteena_a (1'b1),
69               .byteena_b (1'b1),
70               .clock1 (1'b1),
71               .clocken0 (1'b1),
72               .clocken1 (1'b1),
73               .clocken2 (1'b1),
74               .clocken3 (1'b1),
75               .data_a ({10{1'b1}}),
76               .data_b (1'b1),
77               .eccstatus (),
78               .q_b (),
79               .rden_a (1'b1),
80               .rden_b (1'b1),
81               .wren_a (1'b0),
82               .wren_b (1'b0));
83     defparam
84        altsyncram_component.address_aclr_a = "NONE",
85        altsyncram_component.clock_enable_input_a = "BYPASS",
86        altsyncram_component.clock_enable_output_a = "BYPASS",
87        altsyncram_component.init_file = "./rom_data.mif/rom_data.mif",
88        altsyncram_component.intended_device_family = "Cyclone V",
89        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
90        altsyncram_component.lpm_type = "altsyncram",
91        altsyncram_component.numwords_a = 1024,
92        altsyncram_component.operation_mode = "ROM",
93        altsyncram_component.outdata_aclr_a = "NONE",
94        altsyncram_component.outdata_reg_a = "CLOCK0",
95        altsyncram_component.widthad_a = 10,
96        altsyncram_component.width_a = 10,
97        altsyncram_component.width_byteena_a = 1;
98
99
100 endmodule
```

Overall, the switches generate a binary number which is used as an address that corresponds to a value of the sine wave.

Experiment 13

Tuesday, December 5, 2017 09:44

The Diagram is shown below.



The Code is shown below

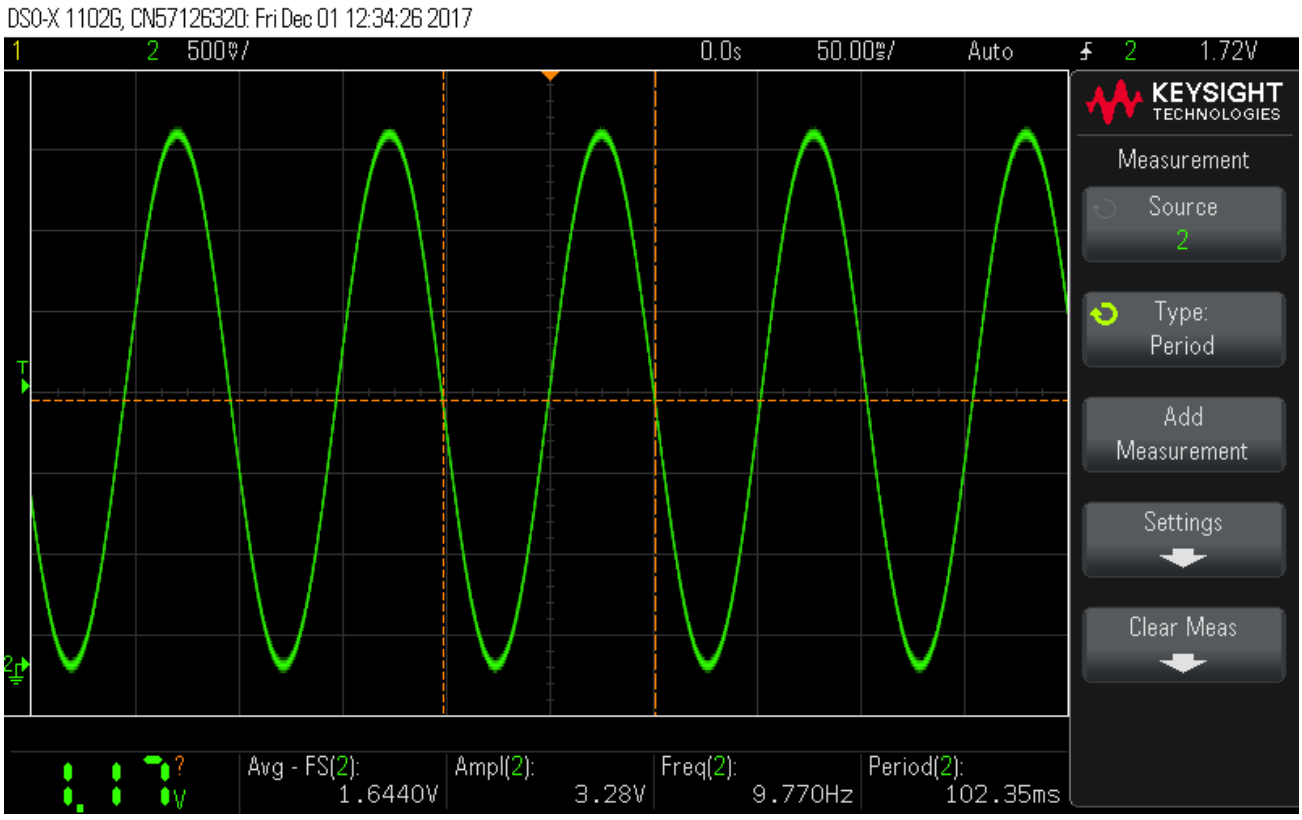
```
module ex13_top(CLOCK_50, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD, PWM_OUT);

   input CLOCK_50;
   output DAC_SDI, DAC_CS, DAC_SCK, DAC_LD, PWM_OUT;

   wire TICK_10KHz;
   wire [9:0] DATA;
   wire [9:0] count;

   reg reset_counter;

   initial reset_counter = 1'b0;

   clktick_16 OBJ1(CLOCK_50, TICK_10KHz, 16'd5000, 1'b1);

   counter_16 OBJ2(CLOCK_50, TICK_10KHz, reset_counter, count);
   ROM OBJ3(count, TICK_10KHz, DATA);

   spi2dac OBJ4(CLOCK_50, DATA, TICK_10KHz, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);
   pwm OBJ5(CLOCK_50, DATA, TICK_10KHz, PWM_OUT);

   always @ (posedge CLOCK_50)
   begin
      if(count == 16'd1023)
         reset_counter <= 1'b1;
      else
         reset_counter <= 1'b0;
   end

   endmodule
```

Overall, the switches generate a binary number which is used as an address that corresponds to a value of the sine wave.

The frequency of the sine wave will be:

$$\frac{50 \times 10^6}{5000 \times 1024} \approx 9.77Hz$$

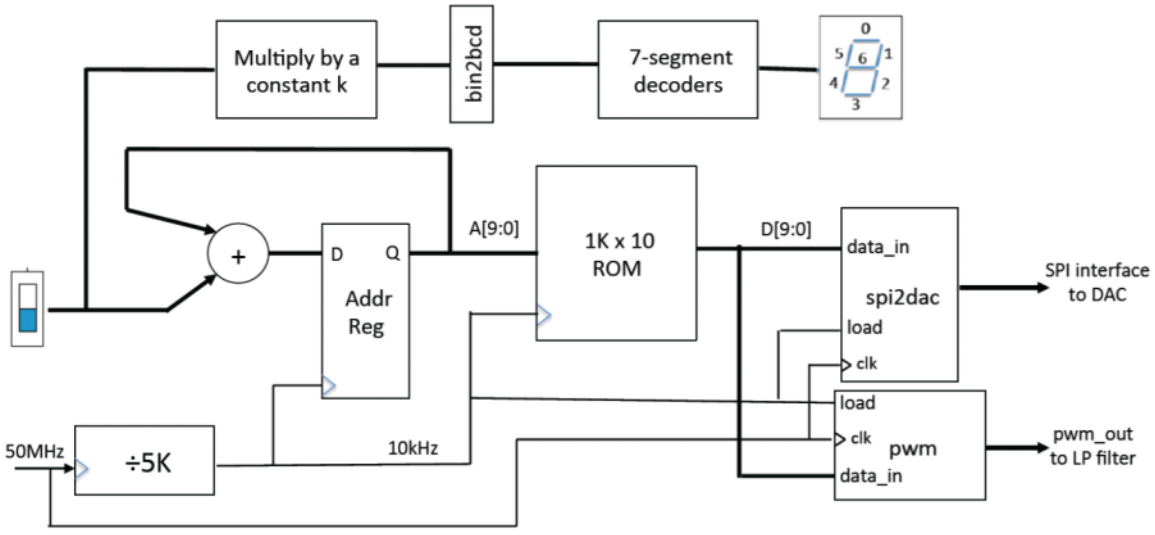
We can confirm this through the use of the oscilloscope



Experiment 14

Saturday, December 2, 2017 20:52

The Diagram is shown below.



The code for the top level design file (i.e. **ex14_top.v**) is shown below.

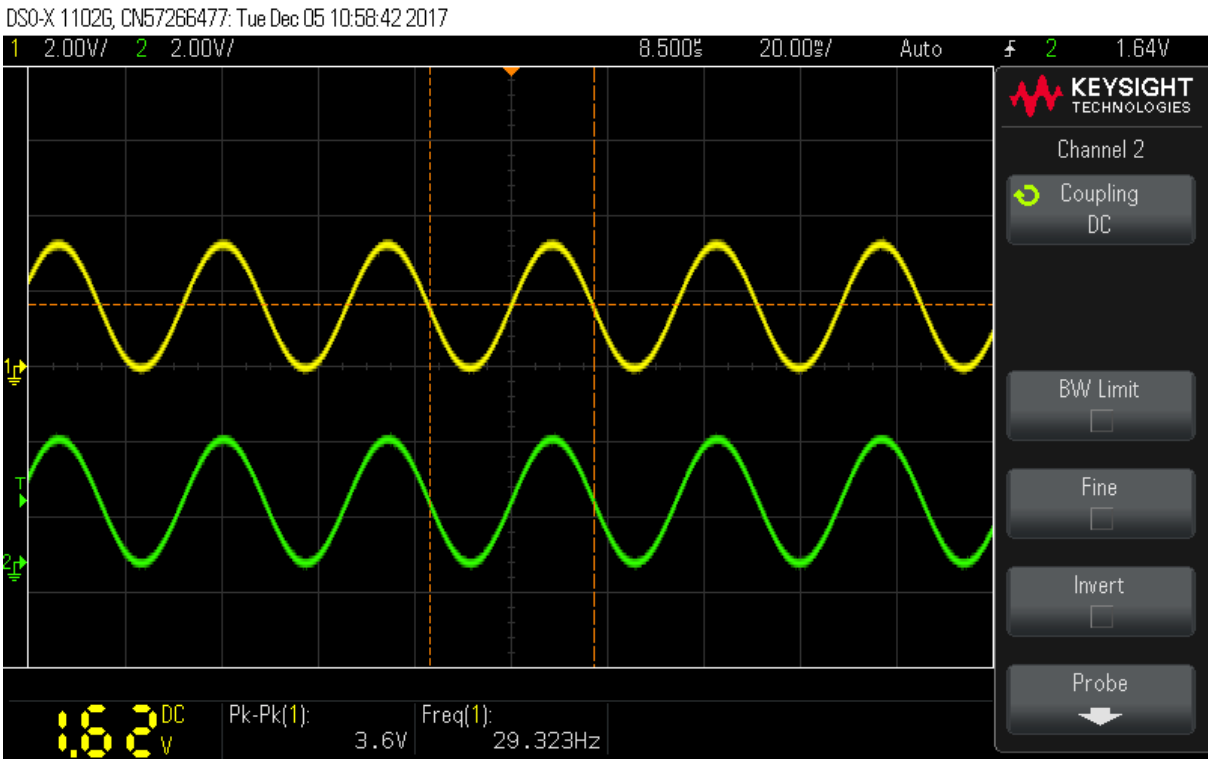
```
module ex14_top(
CLOCK_50,
SW,
HEX0, HEX1, HEX2, HEX3, HEX4,
DAC_SDI, DAC_CS, DAC_SCK, DAC_LD,
PWM_OUT);

    input CLOCK_50;
    input [9:0] SW;

    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;
    output DAC_SDI, DAC_CS, DAC_SCK, DAC_LD, PWM_OUT;

    wire TICK_10KHz;
    wire [9:0] DATA;
    wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;
    wire [23:0] MULT_RES;

    reg [9:0] COUNT;

    initial COUNT = 10'b0;

    clktick_16 OBJ1(CLOCK_50, TICK_10KHz, 16'd5000, 1'b1);

    ROM OBJ3(COUNT, TICK_10KHz, DATA);

    spi2dac OBJ4(CLOCK_50, DATA, TICK_10KHz, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);

    pwm OBJ5(CLOCK_50, DATA, TICK_10KHz, PWM_OUT);

    MULT OBJ2(SW,MULT_RES);

    bin2bcd_16 OBJ6({2'b0, MULT_RES[23:10]}, BCD0, BCD1, BCD2, BCD3, BCD4);

    hex_to_7seg OBJ7(HEX0, BCD0);
    hex_to_7seg OBJ8(HEX1, BCD1);
    hex_to_7seg OBJ9(HEX2, BCD2);
    hex_to_7seg OBJ10(HEX3, BCD3);
    hex_to_7seg OBJ11(HEX4, BCD4);

    always @ (posedge TICK_10KHz)
       COUNT <= COUNT + SW;

    endmodule
```

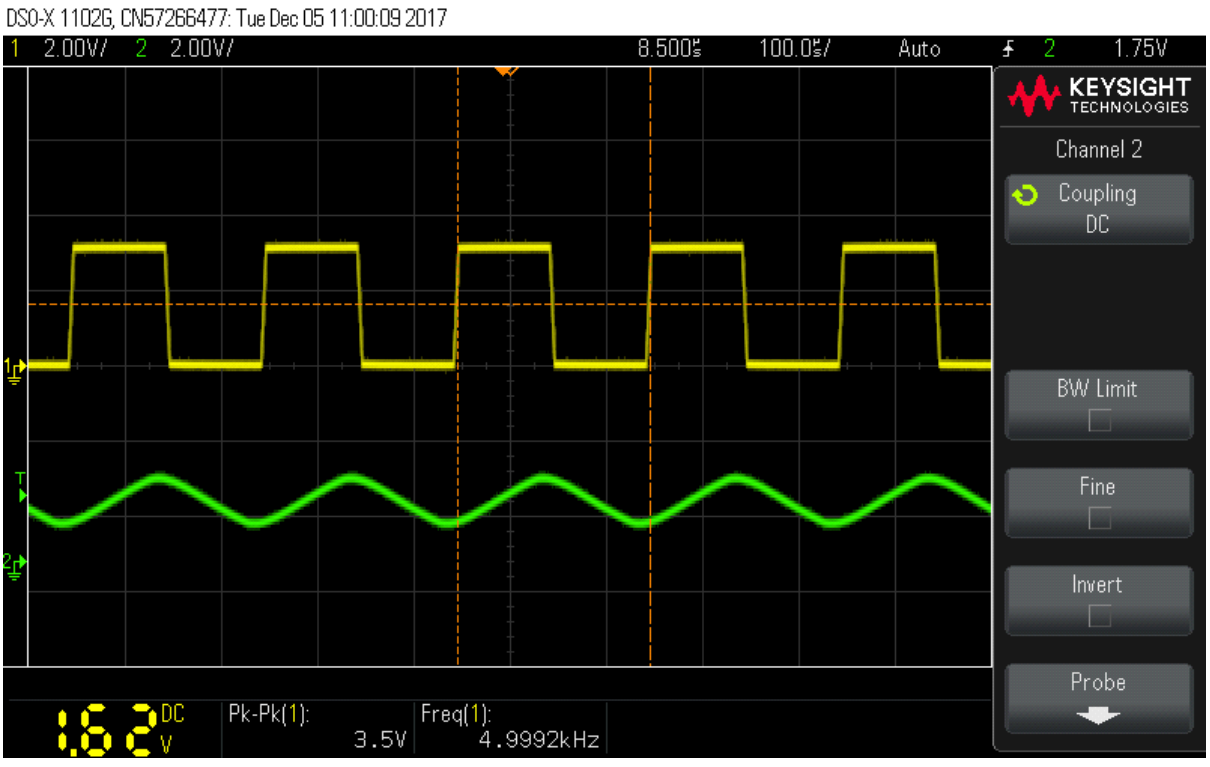
The **MULT** module was created through the Quartus IP Catalog (i.e. **IP Catalog > LPM_MULT**) and simply fill in the form given.

The frequency of the wave generates is going to be:

$$\frac{SW \times 50 \times 10^6}{5000 \times 1024} = \frac{SW \times 10000}{1024}$$

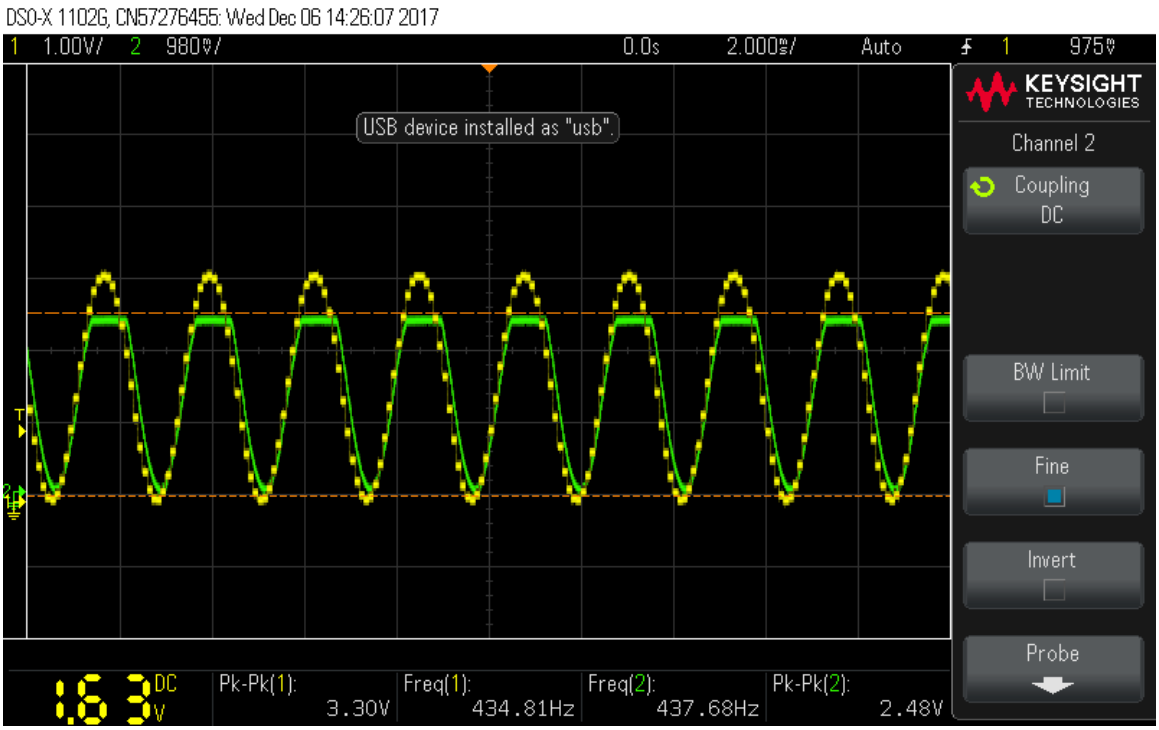
Thus, in order to display the actual frequency of the sine wave generated we first have to multiply the switch value by a constant (i.e. **14'd10000**) and then divide by 1024 (i.e. 2^{10}) which is equivalent to shifting our binary number by 10 bits to the right. Therefore, we only take the top 14 bits of the 24 bit result of the multiply and diregard the last 10 bits. Effectivelly, a very clever was of dividing by a constant (the fact that it is a power of 2 is what allows us to do this). This saves space on the FPGA.





However, as shown in the Images above (Yellow = TP8, Green = TP9) as we increase the step we skip (i.e. the value of **SW[9:0]**) the "resolution" of the sine wave will decrease.

Using the switches, a 439Hz sine wave was produced.



//Could Add some more comments here

Experiment 15

05 December 2017 15:15

The code is shown below.

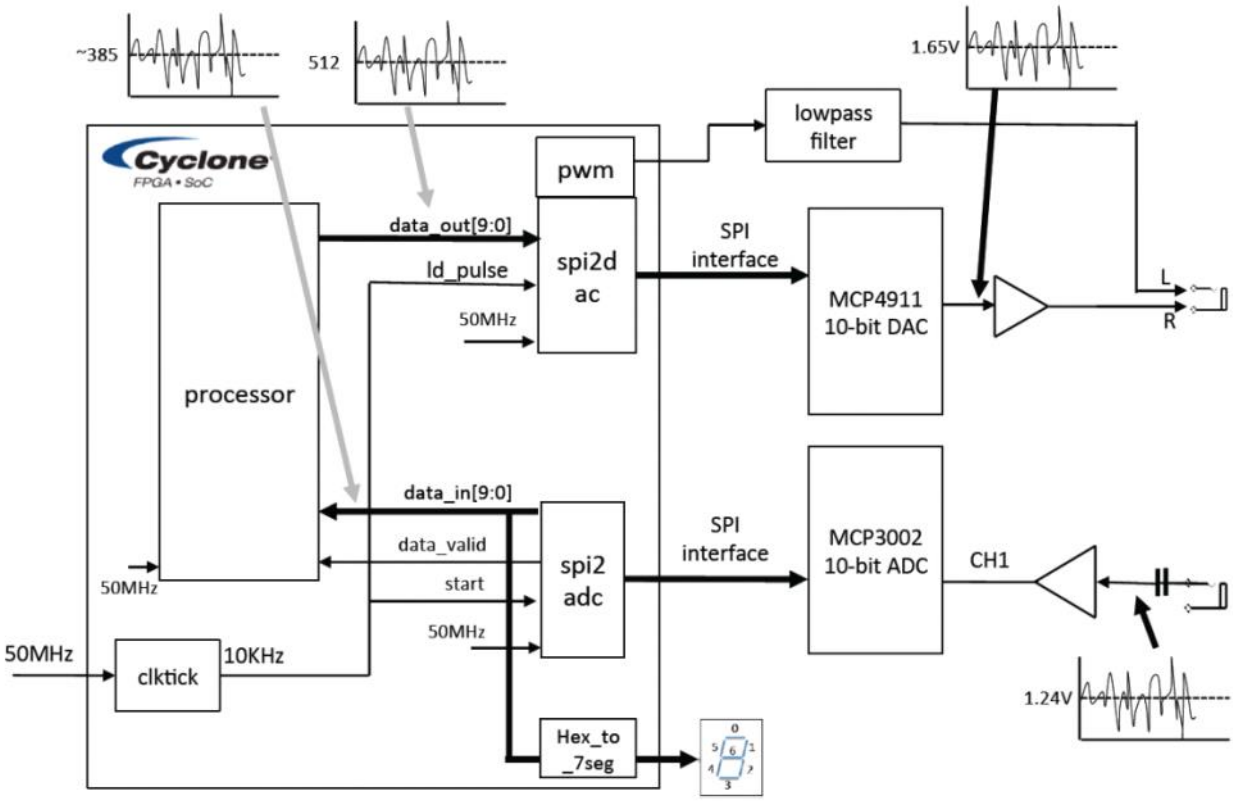
```
module ex15_top(
CLOCK_50,
HEX0, HEX1, HEX2, HEX3, HEX4,
DAC_SDI, DAC_CS, DAC_SCK, DAC_LD,
ADC_SDI, ADC_CS, ADC_SCK, ADC_SDO,
PWM_OUT);

   parameter channel = 1'b0;

   input CLOCK_50;
   input ADC_SDO;

   output ADC_CS, ADC_SCK, ADC_SDI;
   output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;
   output DAC_SDI, DAC_CS, DAC_SCK, DAC_LD, PWM_OUT;


   wire TICK_10KHz;
   wire [9:0] DATA;
   wire [9:0] a2d_data;
   wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;
   wire [23:0] MULT_RES;

   reg [9:0] COUNT;

   initial COUNT = 10'b0;

   clktick_16 OBJ1(CLOCK_50, TICK_10KHz, 16'd5000, 1'b1);

   ROM OBJ3(COUNT, TICK_10KHz, DATA);

   spi2dac OBJ4(CLOCK_50, DATA, TICK_10KHz, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);

   spi2adc SPI_ADC(
      .sysclk (CLOCK_50),
      .channel (1'b0),                                  //set Channel 0 for the potentiometer
      .start (TICK_10KHz),
      .data_from_adc (a2d_data),
      .data_valid (data_valid),                         //not used
      .sdata_to_adc (ADC_SDI),
      .adc_cs (ADC_CS),
      .adc_sck (ADC_SCK),
      .sdata_from_adc (ADC_SDO));

   pwm OBJ5(CLOCK_50, DATA, TICK_10KHz, PWM_OUT);

   mult_24 OBJ2(DATA,MULT_RES);

   bin2bcd_16 OBJ6({2'b0, MULT_RES[23:10]}, BCD0, BCD1, BCD2, BCD3, BCD4);

   hex_to_7seg OBJ7(HEX0, BCD0);
   hex_to_7seg OBJ8(HEX1, BCD1);
   hex_to_7seg OBJ9(HEX2, BCD2);
   hex_to_7seg OBJ10(HEX3, BCD3);
   hex_to_7seg OBJ11(HEX4, BCD4);

   always @ (posedge TICK_10KHz)
      COUNT <= COUNT + a;

   endmodule
```

It worked.

Experiment 16

05 December 2017 10:18

The Diagram for the Experiment shown below.



The code for the experiment is shown below.

```
//-------------------------------
// Module name: ex16_top
// Function: top level module - pass audio input to output directly
// Creator:  Peter Cheung
// Version:  2.0
// Date:     10 Nov 2016
//-------------------------------

module ex16_top (CLOCK_50, SW, HEX0, HEX1, HEX2,
                 DAC_SDI, DAC_SCK, DAC_CS, DAC_LD,
                 ADC_SDI, ADC_SCK, ADC_CS, ADC_SDO, PWM_OUT);

   input       CLOCK_50;      // DE0 50MHz system clock
   input [9:0] SW;            // 10 slide switches to specify address to ROM
   output [6:0] HEX0, HEX1, HEX2;
   output      DAC_SDI;       //Serial data out to SDI of the DAC
   output      DAC_SCK;          //Serial clock signal to both DAC and ADC
   output      DAC_CS;        //Chip select to the DAC, low active
   output      DAC_LD;        //Load new data to DAC, low active
   output      ADC_SDI;       //Serial data out to SDI of the ADC
   output      ADC_SCK;    // ADC Clock signal
   output      ADC_CS;        //Chip select to the ADC, low active
   input       ADC_SDO;       //Converted serial data from ADC
   output      PWM_OUT;       // PWM output to R channel

   wire        tick_10k;      // internal clock at 10kHz
   wire [9:0]  data_in;    // converted data from ADC
   wire [9:0]  data_out;   // processed data to DAC
   wire        data_valid;
   wire        DAC_SCK, ADC_SCK;

   clktick_16  GEN_10K (CLOCK_50, 1'b1, 16'd4999, tick_10k);   // generate 10KHz sampling clock ticks
   spi2dac SPI_DAC (CLOCK_50, data_out, tick_10k,      // send processed sample to DAC
                 DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);    // order of signals matter
   pwm PWM_DC(CLOCK_50, data_out, tick_10k, PWM_OUT);    // output via PWM - R-channel

   spi2adc SPI_ADC (                                   // perform a A-to-D conversion
      .sysclk (CLOCK_50),                              // order of parameters do not matter
      .channel (1'b1),                                 // use only CH1
      .start (tick_10k),
      .data_from_adc (data_in),
      .data_valid (data_valid),
      .sdata_to_adc (ADC_SDI),
      .adc_cs (ADC_CS),
      .adc_sck (ADC_SCK),
      .sdata_from_adc (ADC_SDO));

   processor   ALLPASS (CLOCK_50, data_in, data_out); // do some processing on the data

   hex_to_7seg    SEG0 (HEX0, data_in[3:0]);
   hex_to_7seg    SEG1 (HEX1, data_in[7:4]);
   hex_to_7seg    SEG2 (HEX2, {2'b0,data_in[9:8]});

endmodule
```

```
module processor (sysclk, data_in, data_out);

   input           sysclk;     // system clock
   input [9:0]     data_in;    // 10-bit input data
   output [9:0]    data_out;   // 10-bit output data

   wire            sysclk;
   wire [9:0]      data_in;
   reg [9:0]       data_out;
   wire [9:0]      x,y;

   parameter       ADC_OFFSET = 10'h181;
   parameter       DAC_OFFSET = 10'h200;

   assign x = data_in[9:0] - ADC_OFFSET;      // x is input in 2's complement

   // This part should include your own processing hardware
   // ... that takes x to produce y
   // ... In this case, it is ALL PASS.


   assign y = (x << 2);

   //  Now clock y output with system clock
   always @(posedge sysclk)
      data_out <=  y + DAC_OFFSET;

endmodule

```

We tested the solution and confirmed that the output sound was louder than the input sound.

We also notice that, we turned the volume up, the output sound would get distorted since the x << 2 will overflow.

Experiment 17

05 December 2017 10:29

The code for the experiment is shown below.

```
module ex17_top (CLOCK_50, HEX0, HEX1, HEX2,
                 DAC_SDI, DAC_SCK, DAC_CS, DAC_LD,
                    ADC_SDI, ADC_SCK, ADC_CS, ADC_SDO, PWM_OUT);

    input       CLOCK_50;       // DE0 50MHz system clock
    output [6:0] HEX0, HEX1, HEX2;
    output      DAC_SDI;        //Serial data out to SDI of the DAC
    output      DAC_SCK;           //Serial clock signal to both DAC and ADC
    output      DAC_CS;         //Chip select to the DAC, low active
    output      DAC_LD;         //Load new data to DAC, low active
    output      ADC_SDI;        //Serial data out to SDI of the ADC
    output      ADC_SCK;     // ADC Clock signal
    output      ADC_CS;         //Chip select to the ADC, low active
    input       ADC_SDO;        //Converted serial data from ADC
    output      PWM_OUT;        // PWM output to R channel

    wire        tick_10k;       // internal clock at 10kHz
    wire [9:0]  data_in;     // converted data from ADC
    wire [9:0]  data_out;    // processed data to DAC
    wire        data_valid;
    wire        DAC_SCK, ADC_SCK;

    clktick_16  GEN_10K (CLOCK_50, tick_10k, 16'd5000, 1'b1);   //Changed this module to our clktick_16
    spi2dac SPI_DAC (CLOCK_50, data_out, tick_10k,      // send processed sample to DAC
                DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);      // order of signals matter
    pwm PWM_DC(CLOCK_50, data_out, tick_10k, PWM_OUT);     // output via PWM - R-channel

    spi2adc SPI_ADC (                                   // perform a A-to-D conversion
       .sysclk (CLOCK_50),                              // order of parameters do not matter
       .channel (1'b1),                                 // use only CH1
       .start (tick_10k),
       .data_from_adc (data_in),
       .data_valid (data_valid),
       .sdata_to_adc (ADC_SDI),
       .adc_cs (ADC_CS),
       .adc_sck (ADC_SCK),
       .sdata_from_adc (ADC_SDO));

    processor   ALLPASS (CLOCK_50, tick_10k, data_in, data_out);   // do some processing on the data

    hex_to_7seg    SEG0 (HEX0, data_in[3:0]);
    hex_to_7seg    SEG1 (HEX1, data_in[7:4]);
    hex_to_7seg    SEG2 (HEX2, {2'b0,data_in[9:8]});

endmodule
```

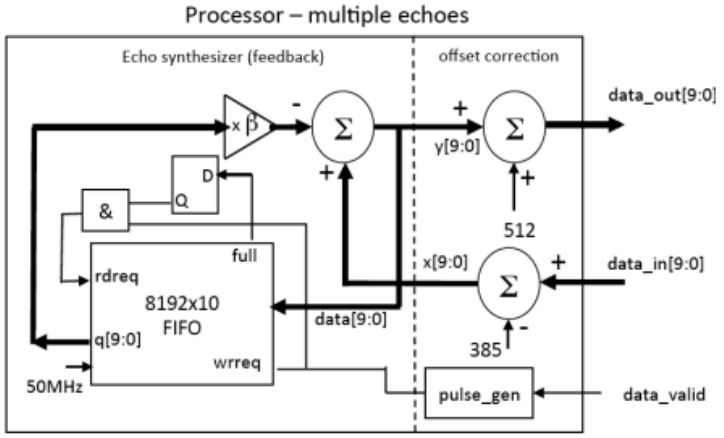
```
module processor (sysclk, data_valid, data_in, data_out);

   input          sysclk;     // system clock
   input          data_valid;
   input [9:0]    data_in;    // 10-bit input data
   output [9:0]   data_out;   // 10-bit output data

   wire           sysclk;
   wire [9:0]     data_in;
   reg [9:0]      data_out;
   wire [9:0]     x,y;
   wire[9:0] echo;
   wire full;
   wire wrreq;

   parameter      ADC_OFFSET = 10'h181;
   parameter      DAC_OFFSET = 10'h200;

   reg rdreq;

   initial rdreq = 1'b0;

   assign x = data_in[9:0] - ADC_OFFSET;     // x is input in 2's complement

   // This part should include your own processing hardware
   // ... that takes x to produce y
   // ... In this case, it is ALL PASS.

   FIFO fifo(sysclk, x, rdreq, wrreq, full, echo);
   pulse_gen pulseGen(sysclk, data_valid, wrreq);

   always @ (posedge sysclk)
   begin
      if(full == 1'b1 && wrreq == 1'b1)
         rdreq <= 1'b1;
      else
         rdreq <= 1'b0;
   end

   assign y = x + {echo[9], echo[9], echo[9:2]}; //Divide by 4 but keep the sign bit;

   //  Now clock y output with system clock
   always @(posedge sysclk)
      data_out <=  y + DAC_OFFSET;

endmodule
```

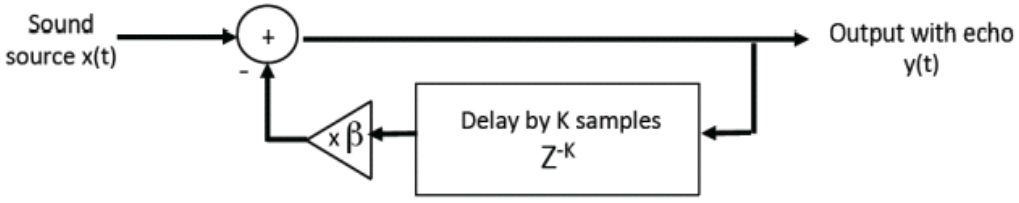
Tested it and it worked.

Experiment 18

05 December 2017 16:38

The Diagram for the experiment is shown below.





The code for the experiment is shown below.

```
module ex18_top (CLOCK_50, HEX0, HEX1, HEX2,
                 DAC_SDI, DAC_SCK, DAC_CS, DAC_LD,
                    ADC_SDI, ADC_SCK, ADC_CS, ADC_SDO, PWM_OUT);

    input       CLOCK_50;       // DE0 50MHz system clock
    output [6:0] HEX0, HEX1, HEX2;
    output      DAC_SDI;        //Serial data out to SDI of the DAC
    output      DAC_SCK;           //Serial clock signal to both DAC and ADC
    output      DAC_CS;         //Chip select to the DAC, low active
    output      DAC_LD;         //Load new data to DAC, low active
    output      ADC_SDI;        //Serial data out to SDI of the ADC
    output      ADC_SCK;    // ADC Clock signal
    output      ADC_CS;         //Chip select to the ADC, low active
    input       ADC_SDO;        //Converted serial data from ADC
    output      PWM_OUT;        // PWM output to R channel

    wire        tick_10k;       // internal clock at 10kHz
    wire [9:0]  data_in;    // converted data from ADC
    wire [9:0]  data_out;   // processed data to DAC
    wire        data_valid;
    wire        DAC_SCK, ADC_SCK;

    clktick_16  GEN_10K (CLOCK_50, tick_10k, 16'd5000, 1'b1);   //Changed this module to our clktick_16
    spi2dac SPI_DAC (CLOCK_50, data_out, tick_10k,      // send processed sample to DAC
                DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);      // order of signals matter
    pwm PWM_DC(CLOCK_50, data_out, tick_10k, PWM_OUT);     // output via PWM - R-channel

    spi2adc SPI_ADC (                                   // perform a A-to-D conversion
       .sysclk (CLOCK_50),                              // order of parameters do not matter
       .channel (1'b1),                                 // use only CH1
       .start (tick_10k),
       .data_from_adc (data_in),
       .data_valid (data_valid),
       .sdata_to_adc (ADC_SDI),
       .adc_cs (ADC_CS),
       .adc_sck (ADC_SCK),
       .sdata_from_adc (ADC_SDO));

    processor   ALLPASS (CLOCK_50, data_valid, data_in, data_out); // do some processing on the data

    hex_to_7seg     SEG0 (HEX0, data_in[3:0]);
```

```
37          .sdata_from_adc (ADC_SDO));
38
39      processor   ALLPASS (CLOCK_50, data_valid, data_in, data_out); // do some processing on the data
40
41      hex_to_7seg     SEG0 (HEX0, data_in[3:0]);
42      hex_to_7seg     SEG1 (HEX1, data_in[7:4]);
43      hex_to_7seg     SEG2 (HEX2, {2'b0,data_in[9:8]});
44
45  endmodule
```

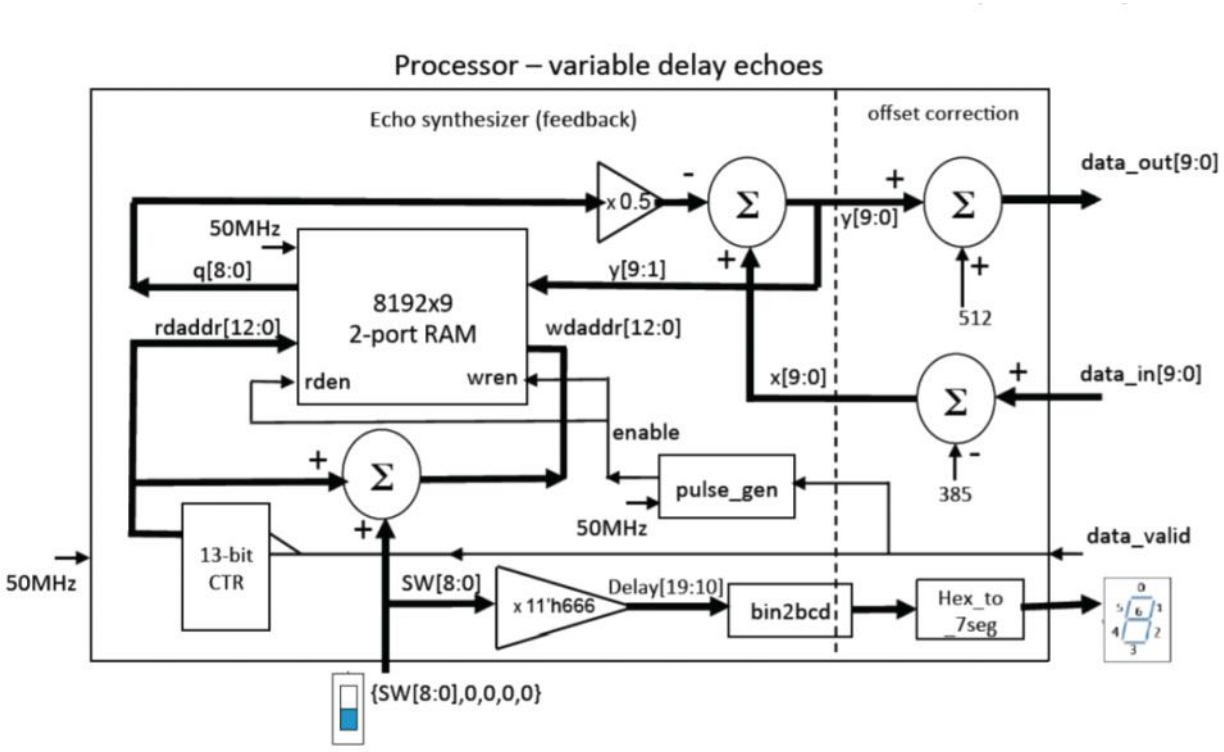
```
 1  module processor (sysclk, data_valid, data_in, data_out);
 2
 3      input           sysclk;     // system clock
 4      input           data_valid;
 5      input [9:0]     data_in;    // 10-bit input data
 6      output [9:0]    data_out;   // 10-bit output data
 7
 8      wire            sysclk;
 9      wire [9:0]      data_in;
10      reg [9:0]       data_out;
11      wire [9:0]      x,y;
12      wire[9:0] echo;
13      wire full, wrreq;
14
15      parameter       ADC_OFFSET = 10'h181;
16      parameter       DAC_OFFSET = 10'h200;
17
18      reg rdreq;
19
20      initial rdreq = 1'b0;
21
22      assign x = data_in[9:0] - ADC_OFFSET;     // x is input in 2's complement
23
24      // This part should include your own processing hardware
25      // ... that takes x to produce y
26      // ... In this case, it is ALL PASS.
27
28      pulse_gen PulseGen(sysclk, data_valid, wrreq);
29      FIFO fifo(sysclk, y, rdreq, wrreq, full, echo);
30
31      always @ (posedge sysclk)
32      begin
33          if(full == 1'b1 && wrreq == 1'b1)
34              rdreq <= 1'b1;
35          else
36              rdreq <= 1'b0;
37      end
38
39      assign y = x - {echo[9], echo[9], echo[9:2]}; //Divide by 4 but keep the sign bit;
40
41      //  Now clock y output with system clock
42      always @(posedge sysclk)
43          data_out <=  y + DAC_OFFSET;
44
45  endmodule
```

It is important to subtract the attenuated signal (i.e. **y(t)**) from the input signal (i.e. **x(t)**) in order to have negative feedback. If we added them, the system would have positive feedback and thus become unstable. Humans aren't sensitive to changes in phase of audio signals and thus there is no noticeable difference when listening to the echo. By subtracting the signal instead of adding it we are essentially shifting the phase of the added signal by 180 degrees.

Experiment 19

Tuesday, December 5, 2017 19:33

The Diagram for the experiment is shown below.



The code for the experiment is shown below.

```
module ex19_top (SW, CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4,
                 DAC_SDI, DAC_SCK, DAC_CS, DAC_LD,
                    ADC_SDI, ADC_SCK, ADC_CS, ADC_SDO, PWM_OUT);

    input [8:0] SW;
    input       CLOCK_50;       // DE0 50MHz system clock
    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;
    output      DAC_SDI;        //Serial data out to SDI of the DAC
    output      DAC_SCK;          //Serial clock signal to both DAC and ADC
    output      DAC_CS;         //Chip select to the DAC, low active
    output      DAC_LD;         //Load new data to DAC, low active
    output      ADC_SDI;        //Serial data out to SDI of the ADC
    output      ADC_SCK;     // ADC clock signal
    output      ADC_CS;         //Chip select to the ADC, low active
    input       ADC_SDO;        //Converted serial data from ADC
    output      PWM_OUT;        // PWM output to R channel

    wire        tick_10k;       // internal clock at 10kHz
    wire [9:0]  data_in;    // converted data from ADC
    wire [9:0]  data_out;   // processed data to DAC
    wire        data_valid;
    wire        DAC_SCK, ADC_SCK;
    wire [19:0] display;
    wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;



    clktick_16  GEN_10K (CLOCK_50, tick_10k, 16'd5000, 1'b1);   //Changed this module to our clktick_16
    spi2dac SPI_DAC (CLOCK_50, data_out, tick_10k,     // send processed sample to DAC
                DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);     // order of signals matter
    pwm PWM_DC(CLOCK_50, data_out, tick_10k, PWM_OUT);     // output via PWM - R-channel

    spi2adc SPI_ADC (                                  // perform a A-to-D conversion
       .sysclk (CLOCK_50),                             // order of parameters do not matter
       .channel (1'b1),                                // use only CH1
       .start (tick_10k),
       .data_from_adc (data_in),
       .data_valid (data_valid),
       .sdata_to_adc (ADC_SDI),
       .adc_cs (ADC_CS),
       .adc_sck (ADC_SCK),
       .sdata_from_adc (ADC_SDO));

    processor   ALLPASS (SW, CLOCK_50, data_valid, data_in, data_out);   // do some processing on the data

    mult_const mult(SW, display);

    bin2bcd_16 bin_to_bcd({6'b0, display[19:10]}, BCD0, BCD1, BCD2, BCD3, BCD4);

    hex_to_7seg    SEG0 (HEX0, BCD0);
    hex_to_7seg    SEG1 (HEX1, BCD1);
    hex_to_7seg    SEG2 (HEX2, BCD2);
    hex_to_7seg    SEG3 (HEX3, BCD3);
    hex_to_7seg    SEG4 (HEX4, BCD4);

endmodule
```

```
module processor (sw, sysclk, data_valid, data_in, data_out);

   input [8:0]    sw;
   input          sysclk;     // system clock
   input          data_valid;
   input [9:0]    data_in;    // 10-bit input data
   output [9:0]   data_out;   // 10-bit output data

   wire           sysclk;
   wire [9:0]     data_in;
   reg [9:0]      data_out;

   wire [9:0]     x,y;
   wire [8:0]     echo;
   wire           enwrrd;

   wire [12:0]    readaddr, writeaddr;

   parameter      ADC_OFFSET = 10'h181;
   parameter      DAC_OFFSET = 10'h200;

   assign x = data_in[9:0] - ADC_OFFSET;     // x is input in 2's complement

   // This part should include your own processing hardware
   // ... that takes x to produce y
   // ... In this case, it is ALL PASS.

   assign writeaddr = readaddr + {sw, 4'b0};

   pulse_gen PulseGen(sysclk, data_valid, enwrrd);

   ram_2port RAM(sysclk, y[9:1], readaddr, enwrrd, writeaddr, enwrrd, echo);

   counter_13 counter(data_valid, 1'b1, 1'b0, readaddr);


   assign y = x - {echo[8], echo[8:1]}; //Divide by 2 but keep the sign bit;

   //  Now clock y output with system clock
   always @(posedge sysclk)
      data_out <=  y + DAC_OFFSET;

endmodule
```

Tested it and it worked.

We need the addresses to be 13 bits because $2^{13} = 8192$

The delay in milliseconds is: $\boldsymbol{SW[8:0] \times 16 \times 0.1\ msec}$

In order to calculate this while being hardware efficient we multiply by 1638 and then divide by 1024
(i.e. removing the last 10 bits).

$\frac{1638}{1024} \approx 1.6.$

Thus it is a pretty close approximation to the actual delay.