Digital Electronics 2: Experiment Verilog

Laboratory Book

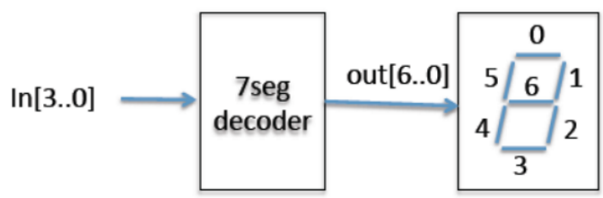
Omar Sharif, *EIE Undergraduate*
Raphael Bijaoui, *EIE Undergraduate*
EEE Department, Imperial College

Experiment 1: Schematic capture using Quartus 7-s display

Sunday, December 8, 2019 1:48 PM

In Experiment 1, we were tasked with designing a 7-segment decoder using 'schematic-entr method (as opposed to using Verilog).

The decoder output out[6:0] drives the 7 segments on the HEX displays. The LED segments a low active which means that the LED will light up (ON) only if the corresponding digital signa 0V.

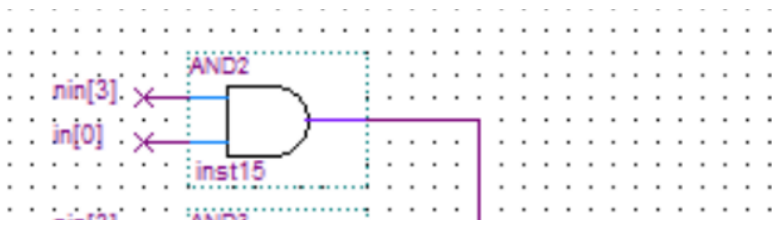


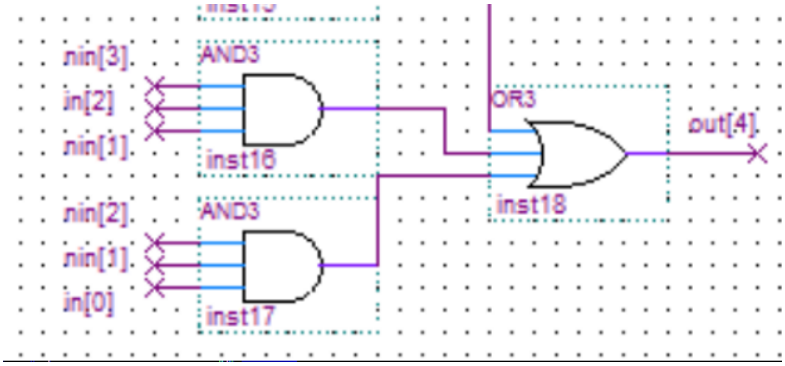
in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7

in[3..0]	out[6:0]	Digit
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

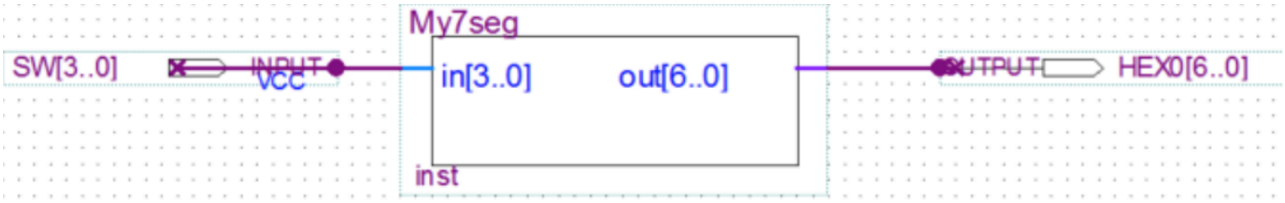
Using the truth table, it is possible to using K-maps to minimalize the logic required the logic required for the decoder – however Quartus performs the logic minimalization and is better taking into account the architecture of the FPGA chip than us!

We therefore downloaded the incomplete 7 segment decoder circuit and implemented the equation out4 = /in3*in0 + /in3*in2*/in1 + /in2*/in1*in0 using the schematic editor (as sho below) to complete the design.





We then encapsulated the circuit into a module (entity), setting this module as the top-level design schematic (as shown below).



To run the module, we need to associate the design with the physical pins of the Cyclone V FPGA board. The top level input/output pins are shown as a list below and the I/O standard (i.e. interface voltages) were set“3.3V LVTTL".

Signal Name	Pin Location
HEX0[6]	PIN_AH28
HEX0[5]	PIN_AG28
HEX0[4]	PIN_AF28
HEX0[3]	PIN_AG27
HEX0[2]	PIN_AE28
HEX0[1]	PIN_AE27
HEX0[0]	PIN_AE26
SW[3]	PIN_AF10
SW[2]	PIN_AF9
SW[1]	PIN_AC12
SW[0]	PIN_AB12

We then examined how the propagation delay from inputs to outputs varies with temperature for (Slow 1100mV) with

- RR (CLK rising - Output rising)
- RF (CLK rising - Output falling)
- FR (CLK falling - Output rising)
- FF (CLK falling - Output falling)

Slow Model is the worst case delay & Fast is the best case delay

0 Degree Model (ns)

Propagation Delay

	Input Port	Output Port	RR	RF	FR	FF
1	SW[0]	HEX0[0]	8.320	8.547	8.579	8.814
2	SW[0]	HEX0[1]	8.520	8.946	8.779	9.259
3	SW[0]	HEX0[2]		8.825	8.795	
4	SW[0]	HEX0[3]	7.964	8.141	8.219	8.450
5	SW[0]	HEX0[4]	8.737			9.388
6	SW[0]	HEX0[5]	8.622			9.394
7	SW[0]	HEX0[6]	8.060	8.155	8.318	8.421
8	SW[1]	HEX0[0]	7.978	8.226	8.267	8.497
9	SW[1]	HEX0[1]	8.273	8.648	8.568	8.920
10	SW[1]	HEX0[2]	8.186			8.771
11	SW[1]	HEX0[3]	7.761	7.887	8.059	8.162
12	SW[1]	HEX0[4]		8.800	8.684	
13	SW[1]	HEX0[5]	8.381	8.790	8.677	9.063
14	SW[1]	HEX0[6]	7.718	7.834	8.007	8.105
15	SW[2]	HEX0[0]	8.520	8.822	8.800	9.155
16	SW[2]	HEX0[1]	8.995			9.636
17	SW[2]	HEX0[2]	8.740	9.106	9.020	9.439
18	SW[2]	HEX0[3]	8.436	8.539	8.716	8.827
19	SW[2]	HEX0[4]	8.937	9.396	9.217	9.729
20	SW[2]	HEX0[5]	9.097	9.483	9.377	9.771
21	SW[2]	HEX0[6]	8.260	8.430	8.540	8.763
22	SW[3]	HEX0[0]	8.128	8.417	8.344	8.581
23	SW[3]	HEX0[1]	8.423	8.846	8.784	9.124
24	SW[3]	HEX0[2]	8.356	8.709	8.573	8.874
25	SW[3]	HEX0[3]	7.863	8.037	8.224	8.315
26	SW[3]	HEX0[4]		8.997	8.767	
27	SW[3]	HEX0[5]	8.524	8.981	8.885	9.259
28	SW[3]	HEX0[6]	7.864	8.021	8.081	8.186

85 Degree Model (ns)

Propagation Delay

	Input Port	Output Port	RR	RF	FR	FF
1	SW[0]	HEX0[0]	8.716	8.921	8.901	9.112
2	SW[0]	HEX0[1]	8.980	9.365	9.173	9.592
3	SW[0]	HEX0[2]		9.247	9.163	
4	SW[0]	HEX0[3]	8.371	8.506	8.563	8.732
5	SW[0]	HEX0[4]	9.167			9.721
6	SW[0]	HEX0[5]	9.072			9.726

7	SW[0]	HEX0[6]	8.446	8.510	8.632	8.702
8	SW[1]	HEX0[0]	8.389	8.609	8.620	8.826
9	SW[1]	HEX0[1]	8.722	9.060	8.954	9.273
10	SW[1]	HEX0[2]	8.643			9.147
11	SW[1]	HEX0[3]	8.166	8.254	8.402	8.472
12	SW[1]	HEX0[4]		9.218	9.072	
13	SW[1]	HEX0[5]	8.821	9.202	9.054	9.417
14	SW[1]	HEX0[6]	8.120	8.199	8.352	8.417
15	SW[2]	HEX0[0]	8.969	9.238	9.172	9.474
16	SW[2]	HEX0[1]	9.463			9.988
17	SW[2]	HEX0[2]	9.235	9.570	9.438	9.806
18	SW[2]	HEX0[3]	8.851	8.923	9.049	9.126
19	SW[2]	HEX0[4]	9.420	9.846	9.623	10.082
20	SW[2]	HEX0[5]	9.554	9.919	9.753	10.123
21	SW[2]	HEX0[6]	8.700	8.828	8.902	9.063
22	SW[3]	HEX0[0]	8.580	8.835	8.718	8.923
23	SW[3]	HEX0[1]	8.918	9.298	9.196	9.500
24	SW[3]	HEX0[2]	8.855	9.176	8.995	9.266
25	SW[3]	HEX0[3]	8.305	8.435	8.583	8.637
26	SW[3]	HEX0[4]		9.450	9.176	
27	SW[3]	HEX0[5]	9.010	9.433	9.287	9.634
28	SW[3]	HEX0[6]	8.306	8.420	8.445	8.509

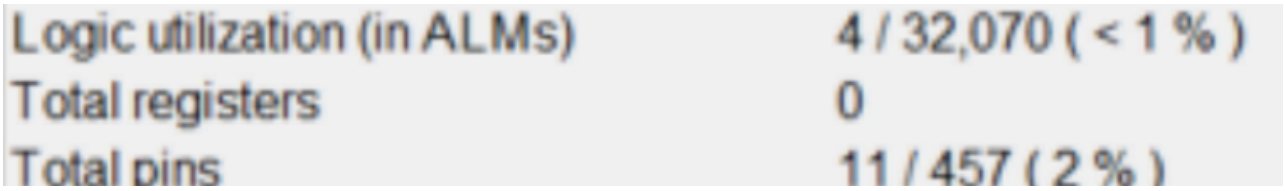
RR & FF gaps vs. RF & FR gaps -> these are representative of the cases when there a RF in the input never causes a FR in the output

We got 2 key takeaways from this experiment:

1. the ***0 degree model had lower propagation times than the 85 degree model***. This can be explained because at lower temperatures there is less resistance in the wires within the FPGA. The effect of increased heat on the atomic structure of a wire to make the atoms vibrate, making it more difficult for signals to propagate.
2. It was very tedious to make the block diagram! The need for code clarity is real.

Resources

Finally, we examined the compilation report for our resources used. We found that this design used only 4 out of 32,060 ALMs (Adaptive Logic Modules), 11 of the 457 I/O pins and none of the other resources.



Experiment 2: 7-segment Decoder in Verilog HDL

Sunday, December 8, 2019 1:49 PM

We found the process of designing the 7-segment decoder using schematic design to be limiting and time consuming. In reality designers would use a hardware description language to specify design.

For Experiment 2, we designed a module for the 7-segment decoder

```
module hex_to_7seg   (out,in);

   output   [6:0] out;     // low-active out
   input [3:0] in;       // 4-bit binary inp

   reg      [6:0] out;  // make out a varia

   always @ (*)
     case (in)
      4'h0: out = 7'b1000000;
      4'h1: out = 7'b1111001;     // --0 --
      4'h2: out = 7'b0100100;     // |     |
      4'h3: out = 7'b0110000;     // 5     1
      4'h4: out = 7'b0011001;     // |     |
      4'h5: out = 7'b0010010;     // --6 --
      4'h6: out = 7'b0000010;     // |     |
      4'h7: out = 7'b1111000;     // 4     2
      4'h8: out = 7'b0000000;     // |     |
      4'h9: out = 7'b0011000;     // --3 --
      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 specified the top-level design module in Verilog - analogous to encapsulating our schematic design in Experiment 1.

```
module  ex2_top (
      SW,                  // input switches
      HEX0                 // Hex output on 7 segment display
);
   input    [3:0] SW;    // declare input/output ports
   output   [6:0] HEX0;

   hex_to_7seg    SEG0 (HEX0, SW);

endmodule
```

PIN ASSIGNMENT

In Experiment 1, we used the pin assignment editor to associate pins – this is also tedious. Instead we used the generated qsf file along with the provided pin_assignment.txt to assign pins. In the qsf file we find the following statements:

```
set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to HEX0[4]
set_location_assignment PIN_AF28 -to HEX0[4]
```

1. The first line defines the voltage standard used by the HEX0[4] signal (3.3V logic).
2. The second line defines the physical pin location of HEX0[4] is PIN_AF28.

Experiment 3: 10-bit binary switch values on three 7-segment displays

Sunday, December 8, 2019 1:50 PM

In Experiment 3, we were tasked to create our own design to display all 10-bit sliding switches as hexadecimal values on three 7 segment LED displays. Our top module is shown below:

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

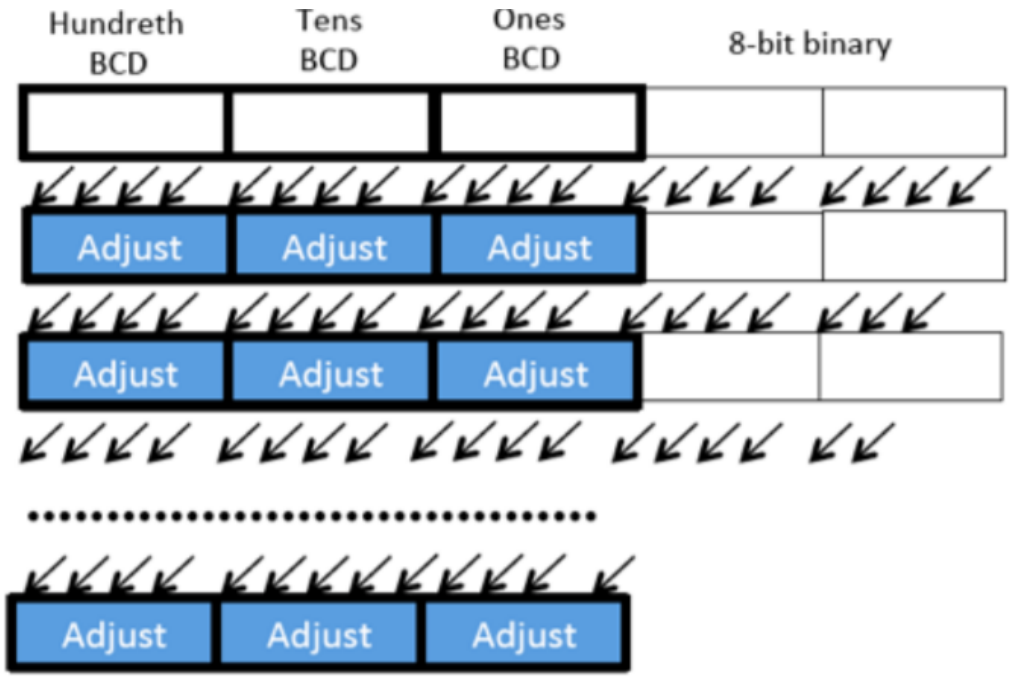
1. This code in the top-level specification for 10-bit binary switch values on three 7-segment displays
2. It takes as input the 10 switch value (0 -> off & 1 -> on).
3. This is then relayed into three hex_to_7seg function calls, each taking as input the required switch values.
4. These values are specified using SW[3:0] which takes SW0, SW1, SW2, etc.
5. Since there are only 2 switches SW[9:8] specifying one hex values, we have the maximum values as being **10'h03FFFFFF**

Experiment 4 (Optional): Displaying 10-bit binary as displays

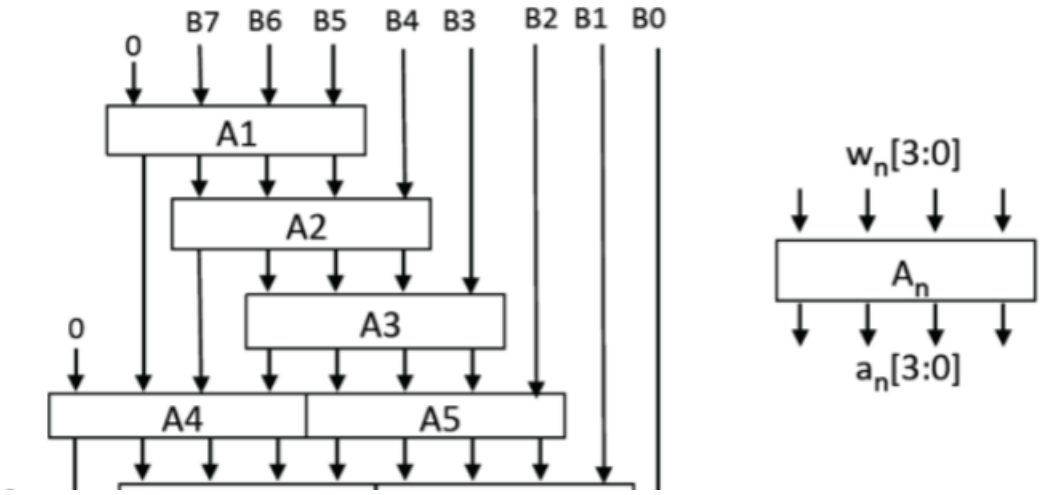
Sunday, December 8, 2019 1:50 PM

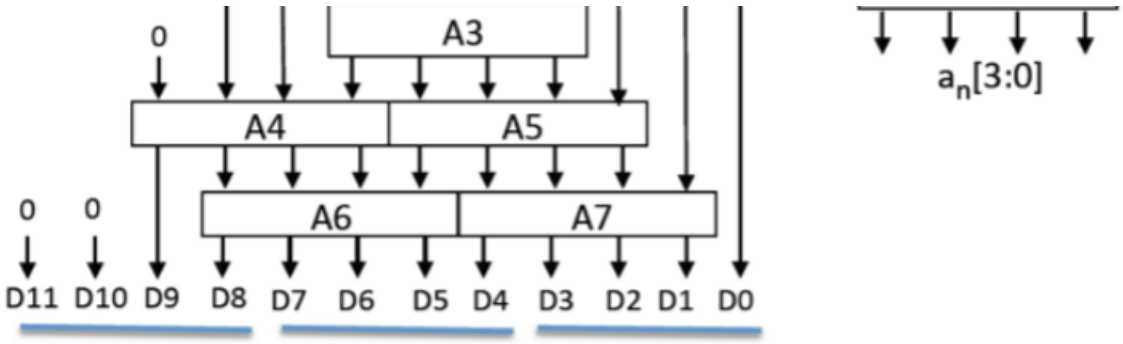
In Experiment 4, we were tasked to build upon our own design to display all 10-bit sliding sv

To do this, we started off with the "Shift and Add 3" Algorithm which was presented to us in applying all the arithmetic/logic blindly we took the wiring approach to perform the binary t shifting in hardware is simple. It does not require gates, only connecting signals with one bit

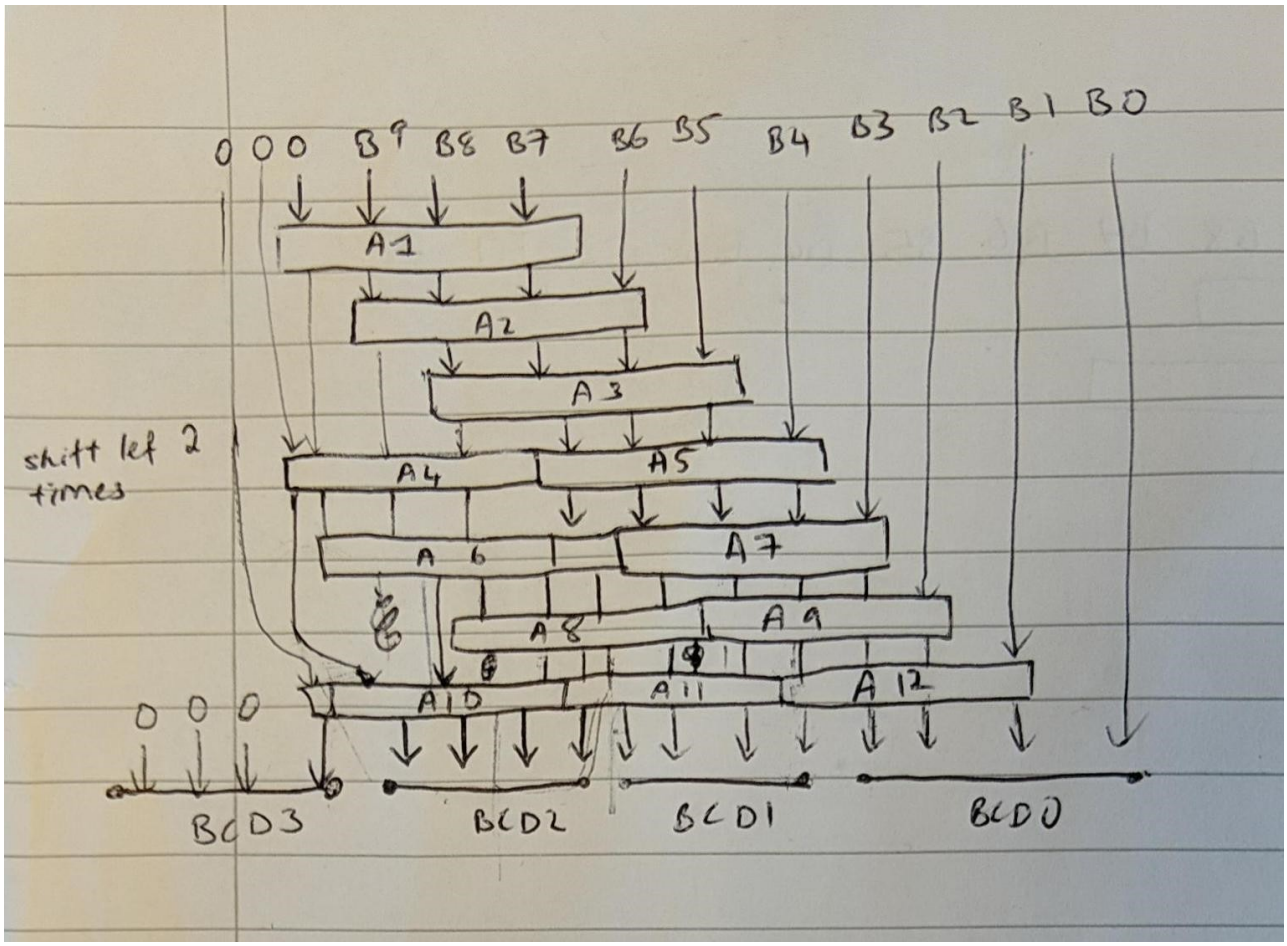


Using the example given to us in Lecture 10 for an 8 bit converter, we used the same princip propagation of binary bits into the system to isolate where the add3_ge5 modules will be us





Below it the diagram we constructed to help us to design the module.



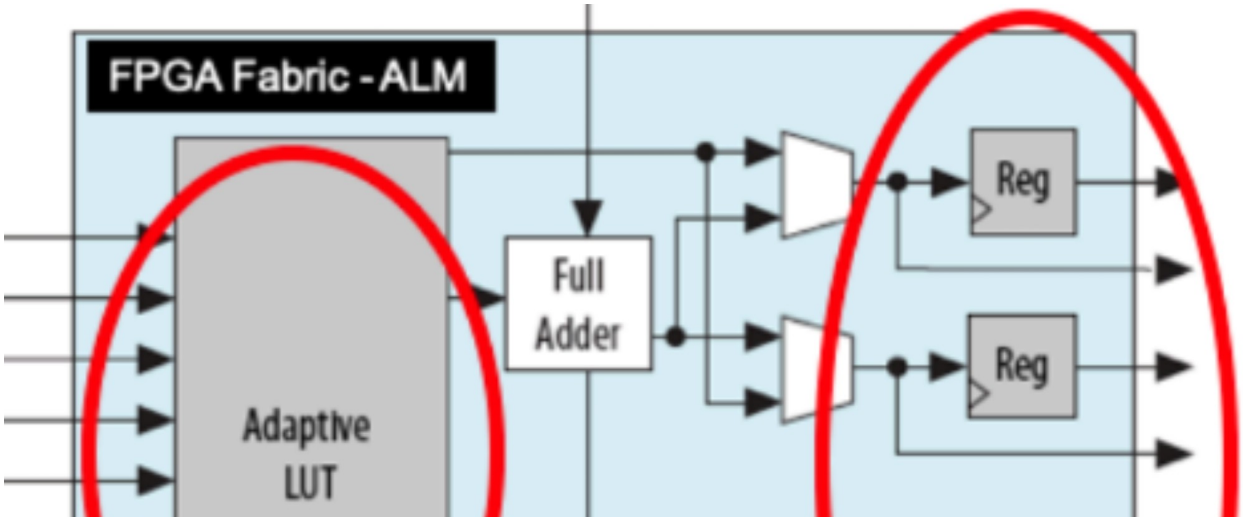
We then specified our top level design module passing the converted BCD values as wires into our hex_to_7seg modules to output our results in 4 displays. Our top-level design module can be seen below:

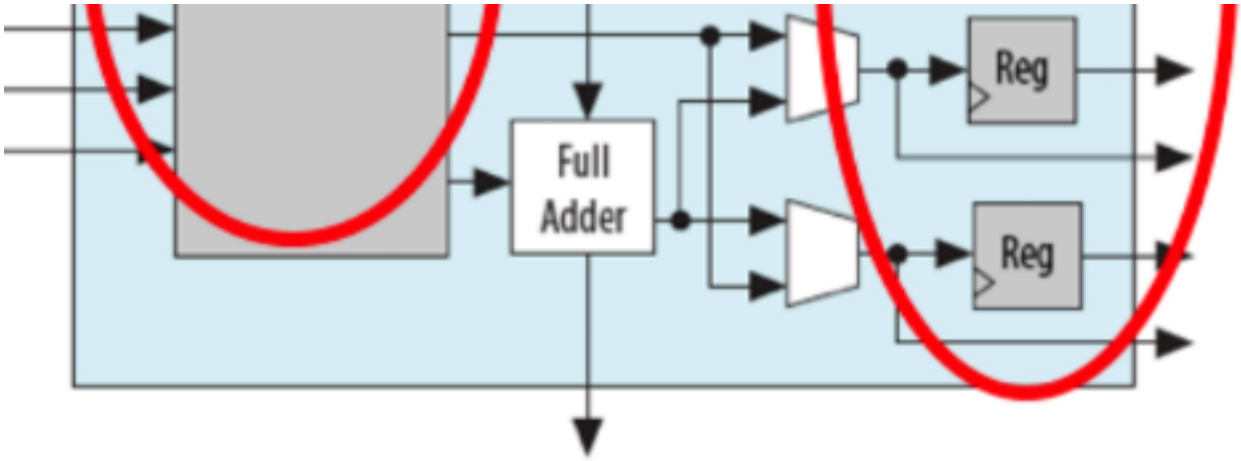
```
module ex4_top(SW, HEX0,HEX1,HEX2,HEX3);

    input [9:0] SW;
    output [6:0] HEX0;
    output [6:0] HEX1;
    output [6:0] HEX2;
    output [6:0] HEX3;
```

```
    wire [3:0] BCD_0, BCD_1, BCD_2, BCD_3;

    binary_to_bcd_10 bin2bcd10(SW, BCD_0, BCD_1, BCD_2, BCD_3);

    hex_to_7seg SEG0(HEX0, BCD_0);
    hex_to_7seg SEG1(HEX1, BCD_1);
    hex_to_7seg SEG2(HEX2, BCD_2);
    hex_to_7seg SEG3(HEX3, BCD_3);

endmodule
```

Wiring up of the binary to bcd

```
//------------------------------
//   For more explanation of how this work, see
//     ... instructions on wwww.ee.ic.ac.uk/pcheung/teaching/E2_experiment

module binary_to_bcd_10 (B, BCD_0, BCD_1, BCD_2,BCD_3);

   input [9:0] B;     // binary input number
   output [3:0]   BCD_0, BCD_1, BCD_2,BCD_3;   // BCD digit LSD to MSD

   wire [3:0]  w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12; //Check how many wires needed- do diagram
   wire [3:0]  a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12;

   // Instantiate a tree of add3-if-greater than or equal to 5 cells
   //  ... input is w_n, and output is a_n
   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);

   // wire the tree of add3 modules together
   assign  w1 = {1'b0, B[9:7]};      // wn is the input port to module An
   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'b0,a4[3],a6[3],a8[3]};
   assign  w11 = {a8[2:0], a9[3]};
   assign  w12 = {a9[2:0],B[1]};


   // connect up to four BCD digit outputs
   assign BCD_0 = {a12[2:0],B[0]};
   assign BCD_1 = {a11[2:0],a12[3]};
   assign BCD_2 = {a10[2:0],a11[3]};
   assign BCD_3 = {3'b0,a10[3]};

endmodule
```




Full
Adder
Reg
Reg

Experiment 5: Counters & FSMs

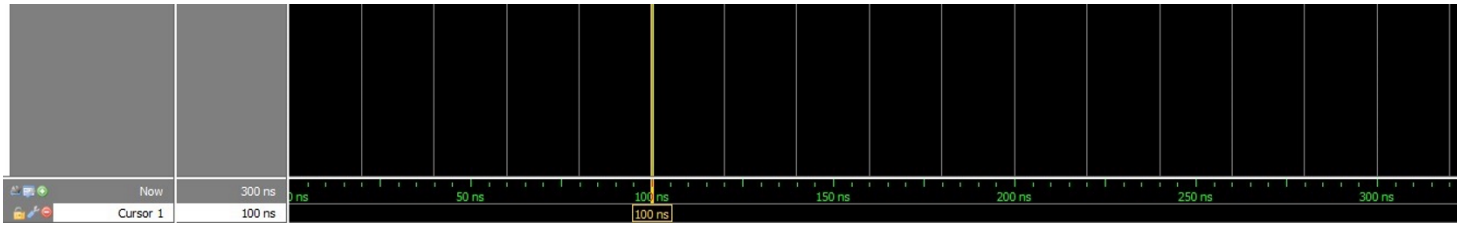
Sunday, December 8, 2019 1:51 PM

In Experiment 5, we were tasked with designing an 8-bit counter as shown below:

```
`timescale 1ns / 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
```

After we ensured our design was correct, we looked to simulate our design using the **Modelsim** environment. In the transcript window we ran the commands to add the *clock* and *count* signals as waveforms (showing count as hexadecimal) before driving the clock with a 50MHz signal and running the simulator for 5 clock cycles.

The timing-diagram below was created using sequential instructions.



We then created a do-file to run as a testbench and simulate our design:

DO-file

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 100ns

Experiment 6: Implementing a 16-bit counter on

Sunday, December 8, 2019 1:52 PM

In Experiment 6, we were tasked with building a 16-bit Cascade Counter having previously standard 16-bit counter, The problem here was that counter was moving at 20MHz which observe the changes. A divide-by-50000 circuit was needed to generate a 1 cycle high puls the output signal tick provides one enable pulse every millisecond

COUNTER

To build these, the first step was to extend our 8 bit counter to allow it to count to 16 bits [BIT_SZ-1:0], we simply set BIT_SZ = 16 in our code from Experiment 5. Then we wired the hex_to_7seg modules

Compilation Report

Frequency

After compiling the design, we investigated the *maximum* frequencies of the project at dif

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	438.02 MHz	438.02 MHz	CLOCK_50	

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	460.19 MHz	460.19 MHz	CLOCK_50	

Here, the Fmax for a higher temperature is higher than the Fmax for a lower temperature learnt in Exercise 1, where we expected the opposite to happen as things got hotter.

We found out that there are 2 factors that affect the Fmax. One is the Resistance of the ci temperature causing the overall electron mobility to decrease. Another is the threshold v also varies with temperature.

temperature- this goes against what we learnt in Exercise 1, where we expected the opposite to happen as things got hotter.

We found out that 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.

We could also see our slack times for both Setup and Hold:

SETUP

Slow 1100mV 0C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	CLOCK_50	-1.283	-15.305

Slow 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	CLOCK_50	-1.173	-14.055

HOLD

Slow 1100mV 0C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	CLOCK_50	0.369	0.000

Slow 1100mV 85C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	CLOCK_50	0.367	0.000

All of our slack times are non-zero, which shows there is some form of bottleneck in our system.

Our negative slack shows that we have not met the timing constraints given by Quartus for our given design.

To fix this, we could either relax the constraints set up, or come up with another design with a better functionality to meet the constraints.

Our hold time shows that the signal can get from the startpoint to the endpoint of the timing path fast enough for the circuit to operate correctly. It is the difference between the required time and the arrival time for the timing path.

To calculate the maximum frequency, we use the equation:

Fmax = 1/(t{c-q} + t{p} + t{setup})

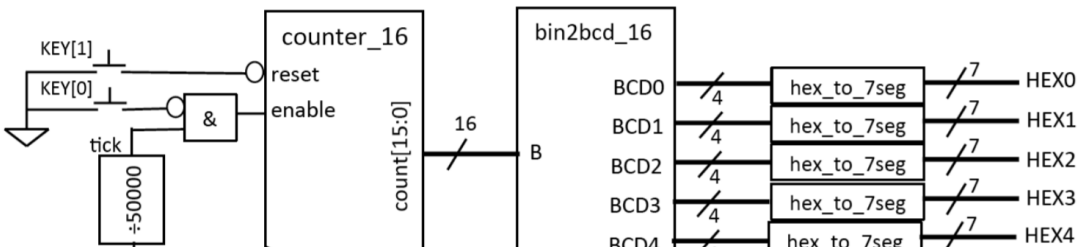
t{c-q} = is delay from the 'clock' to the output Q
t{p} = is the propagation delay for device X
t{setup} = is the setup time

Timing Analyzer

- SETUP - DATA must reach its new value at least ***ts (setup slack)*** before the positive edge of the clock: ***ts = required time – arrival time***
- HOLD – DATA must be held constant for at least ***th (hold time)*** after the positive edge of the clock: ***th= arrival time – required time***

(note, the setup and hold times are measured with respect to the active clock edge only)

CASCADE COUNTER



To extend this to a cascade counter, the next step was to design a tick module to allow the count to change every millisecond. The way this module works is to output high **(tick = 1'b1)** every 50,000 clock cycles as the enable signal into our 16-bit counter. Below is the code for out clock tick

```
module clk_16_tick( //DIVIDE BY 50 000
   clkin,
   tick);

   parameter N_BIT = 16;

   input clkin;
   output tick;

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


   initial tick = 1'b0;
   initial count = 0;

   always @ (posedge clkin)
   begin
         if (count == 16'b1100001101010000) // 16'b1100001101010000 = 50,0000 IN DECIMAL
         begin
            tick <= 1'b1;
            count <= 0;
         end
         else
         begin
            count <= count + 1'b1;
            tick <= 1'b0;
         end
   end

endmodule
```

After creating the clk_16_tick module, the next step was to wire together the modules in the top-level specification.

```
module ex6_top(
      CLOCK_50,
      KEY,
      HEX0,HEX1,HEX2,HEX3,HEX4,
);
   input CLOCK_50;
   input  [1:0] KEY;
   output [6:0] HEX0;
   output [6:0] HEX1;
   output [6:0] HEX2;
   output [6:0] HEX3;
   output [6:0] HEX4;

   //Top level module only has KEY,CLOCK and HEX as in/outputs

   //output [3:0] BCD0;
   //output [3:0] BCD1;
   //output [3:0] BCD2;
   //output [3:0] BCD3;
   //output [3:0] BCD4;
   //output [15:0] count;

   //input and outputs are only for the inputs into the module and outputs
   //wires are used to connect input and output ports of a module instantiation

   wire [3:0] BCD0;
   wire [3:0] BCD1;
```

```
    wire [3:0] BCD2;
    wire [3:0] BCD3;
    wire [3:0] BCD4;
    wire [15:0] count;
    wire tick, enable;

    clk_16_tick    TICK (CLOCK_50, tick);//count up to 50000

    assign enable = (tick & ~KEY[0]); //Key[0] is active low

    counter_16     CNTR (CLOCK_50,enable,~KEY[1],count);
    bin2bcd_16     CVRT (count,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
```

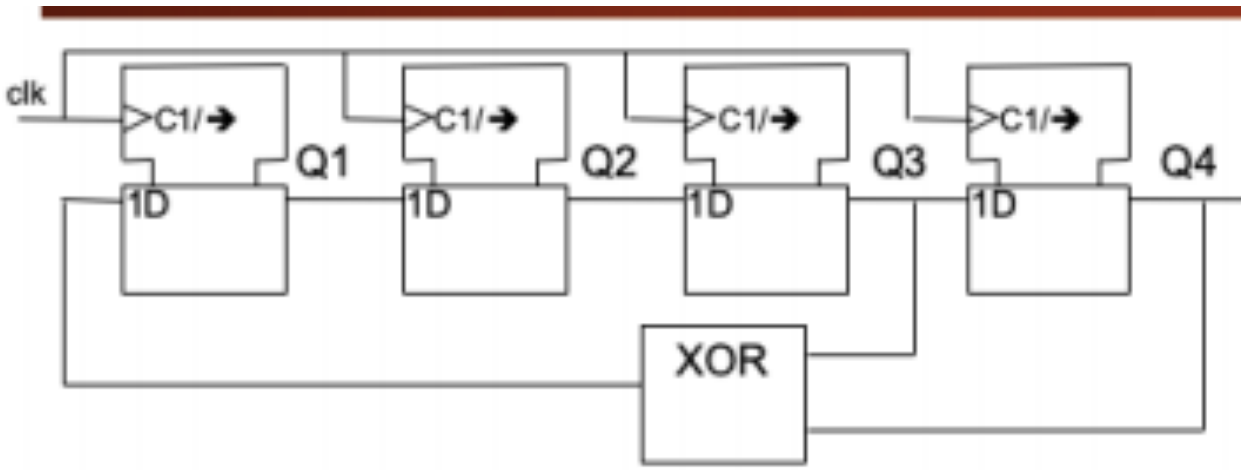
We had the problem of instantiating individual module inputs/outputs as **input/output** instead of wires but after understanding the purpose of **wire,** the implementation of the top-module in Verilog became simple. We also had pay attention to the fact **KEY** is active low so we needed to negate (~) it when assigning enable.

Experiment 7: Linear Feedback Shift Register (LFSR) and PRBS

Sunday, December 8, 2019 1:52 PM

In Experiment 7, we were required to design a 7 bit LFSR implementing the polynomial 1 + X + X^7, initializing the shift register to 7'b1 to generate a pseudo random binary sequence (PRBS).

We first recall the polynomial we came across in Lecture 5, for 1 + X^3 + X^4



It is important to note that he shift register must be initialized to 1 as beginning at 0 causes the count to remain stuck at 0. In our current case, this will constantly shift 0s into the LSB and never change the value of SREG[1] ^ SREG[7].

We understood that to implement the polynomial, we had to XOR the bits we required together.

The first step was to build a module to perform the LFSR:

```
module lfsr_7(
    enable,
    clock,
    out
);

    input clock;
    input enable;
    output [7:1] out;

    reg[7:1] sreg;
```

```
    initial sreg = 7'b1;

    always @ (posedge clock)
       begin
       if (enable==1'b1)
          sreg<= {sreg[6:1], sreg[7] ^ sreg[1]};
       end
    assign out = sreg;

endmodule
```

Afterwards the only thing that was left was to wire up the top-level design specification.

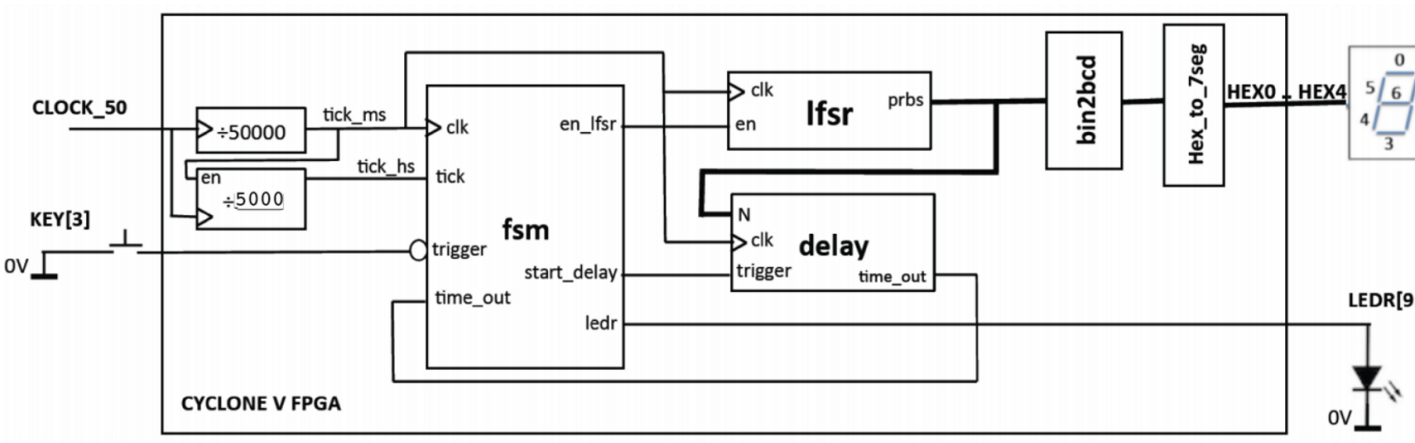
```
module ex7_top(
    CLOCK_50,
    HEX0,     //Hex output on  segment display
    HEX1,
    KEY
);
    input CLOCK_50;
    input [3:3]KEY;
    output [6:0]HEX0;
    output [6:0]HEX1;

    wire [6:0] sreg_out;
    wire enable;

    assign enable = ~KEY[3]; //Key[3] is active low


    lfsr_7 shift(enable, CLOCK_50, sreg_out);

    hex_to_7seg    SEG0(HEX0,sreg_out[3:0]);
    hex_to_7seg    SEG1(HEX1,sreg_out[6:4]);

endmodule
```

It is also important to note that although the shift register is defined as having 7 output bits, it actually has 8 bits with the 0^{th} bit being simulated by the SREG[1] ^ SREG[7] output. Hence why the bits of the bits of the output **out** (which has been assigned to SREG) must be refined as [7:1] instead of [6:0].

Experiment 8 (Optional): Starting line delay circuit

Sunday, December 8, 2019 1:53 PM

For the optional Experiment 8, we were tasked with designing a starting line delay circuit. T
below:



- The circuit is triggered by pressing KEY[3]
- The 10 LEDs will then light up from left to right at 0.5 second intervals until all the LED
- The circuit then waits for a random period (as specified by the LFSR) for between 0.25
  LEDs turn off
- Then the display will show the random delay period in milliseconds on five 7 segment

Design decisions

Before embarking on the Experiment, several design decisions needed to be made:

1. **How many bits LFSR is required?**

We decided on 6 bits, for two reasons:

   a. We needed to start at 0.25 seconds, and end at 16. If we step each time by 0.25
      16/0.25 = 64 different values, which is the size of 6 bits.
   b. This also allowed us to make use of code modularity, by taking our previous wo

2. **How many bits should we use in the delay module?**

The range of timing delay is from 0.25 to 16,000 milliseconds. Hence we need 14 bits which
of 16,384ms ~ 16,000ms

The important modules for this experiment which have not been introduced to us yet throu
Delay modules:

TOP FUNCTION

The schematic is shown

Os are ON
5 and 16 seconds before all

displays

5, we would therefore need

rk on the polynomial 1

allows us to get a maximum

ugh the Labs are the FSM and

The important modules for this experiment which have not been introduced to us yet through the Labs are the FSM and Delay modules:

TOP FUNCTION

```
1  module ex8_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
16     wire TICK_1MS, TICK_500MS;
17     wire time_out;
18     wire en_lfsr, start_delay;
19     wire [6:0] prbs;
20     wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;
21
22     clk_16_tick TICK1(CLOCK_50, TICK_1MS);
23     clk500 TICK500(TICK_1MS,TICK_500MS);
24     lfsr_7 RDMGEN(en_lfsr, TICK_1MS, prbs);
25
26     fsm_light FSM(TICK_1MS, TICK_500MS, ~KEY[3], time_out, en_lfsr, start_delay, LEDR[9:0]);
27
28     //Random delay generation
29     delay DELAY(TICK_1MS, start_delay,(prbs*125),time_out);
30
31     //Number conversions for 7 seg display
32     bin2bcd_16((prbs*125),BCD0, BCD1, BCD2, BCD3, BCD4);
33     hex_to_7seg SEG0(HEX0,BCD0);
34     hex_to_7seg SEG1(HEX1,BCD1);
35     hex_to_7seg SEG2(HEX2,BCD2);
36     hex_to_7seg SEG3(HEX3,BCD3);
37     hex_to_7seg SEG4(HEX4,BCD4);
38
39  endmodule
40
41
42
```

LFSR

```
module lfsr_7(
    enable,
    clock,
    out
);

    input clock;
    input enable;
    output [7:1] out;

    reg[7:1] sreg;

    initial sreg = 7'b1;

    always @ (posedge clock)
       begin
```

```
        if (enable==1'b1)
            sreg<= {sreg[6:1], sreg[7] ^ sreg[1]};
        end
    assign out = sreg;

endmodule
```

Delay

```
module delay(
sysclk,
trigger,
n,
time_out);

    //Define number of bits in delay counter
    parameter BIT_SZ = 16;

    //Ports
    input sysclk, trigger;
    input [BIT_SZ-1:0] n;
    output time_out;

    //reg declaration
    reg[BIT_SZ-1:0] count;
    reg time_out;

    always @ (posedge sysclk)
    begin
        if(trigger==1'b1)
        begin
            if(count==n)
                time_out <= 1'b1;
            else
                count <= count + 1'b1;
        end
        else
        begin
                count <= 0;
                time_out <= 0;
        end
    end


endmodule //end of module
```

FSM

Designing the FSM proved to be the most difficult section of this task because it involves many different steps

```
  input clk, tick, trigger,time_out;
```

```
output en_lfsr, start_delay;
output [9:0] ledr;

reg [9:0] ledr;
reg en_lfsr, start_delay;

//Defining each of our states using one-hot encoding
parameter IDLE = 11'b00000000001;
parameter LIGHT0 = 11'b00000000010;
parameter LIGHT1 = 11'b00000000100;
parameter LIGHT2 = 11'b00000001000;
parameter LIGHT3 = 11'b00000010000;
parameter LIGHT4 = 11'b00000100000;
parameter LIGHT5 = 11'b00001000000;
parameter LIGHT6 = 11'b00010000000;
parameter LIGHT7 = 11'b00100000000;
parameter LIGHT8 = 11'b01000000000;
parameter LIGHT9 = 11'b10000000000;

//setting up initial state for switch case
reg [10:0] state;
initial state = IDLE;
initial en_lfsr = 1'b0;
initial start_delay = 1'b0;

//transitions
always @ (posedge tick)
begin
        case(state)
           IDLE: if(trigger == 1'b1) state<= LIGHT0;
           LIGHT0: if(tick == 1'b1) state <= LIGHT1;
           LIGHT1: if(tick == 1'b1) state <= LIGHT2;
           LIGHT2: if(tick == 1'b1) state <= LIGHT3;
           LIGHT3: if(tick == 1'b1) state <= LIGHT4;
           LIGHT4: if(tick == 1'b1) state <= LIGHT5;
           LIGHT5: if(tick == 1'b1) state <= LIGHT6;
           LIGHT6: if(tick == 1'b1) state <= LIGHT7;
           LIGHT7: if(tick == 1'b1) state <= LIGHT8;
           LIGHT8: if(tick == 1'b1) state <= LIGHT9;
           LIGHT9: if(time_out == 1'b1) state <= IDLE;
           default: ;
        endcase
end

 //Output for each state
 always @ (*)
 begin
        case(state)
           IDLE: begin //no lights
                       ledr <= 10'b0000000000;
                       en_lfsr <= 1'b1; //request random number
                       start_delay <= 1'b0;
                       end
           LIGHT0: begin //start lights
                       ledr <= 10'b1000000000; //LED9
                       en_lfsr <= 1'b0;
                       start_delay <= 1'b0;
                       end
           LIGHT1: begin
                       ledr <= 10'b1100000000; //LED9-8
                       en_lfsr <= 1'b0;
                 start_delay <= 1'b0;
                       end
```

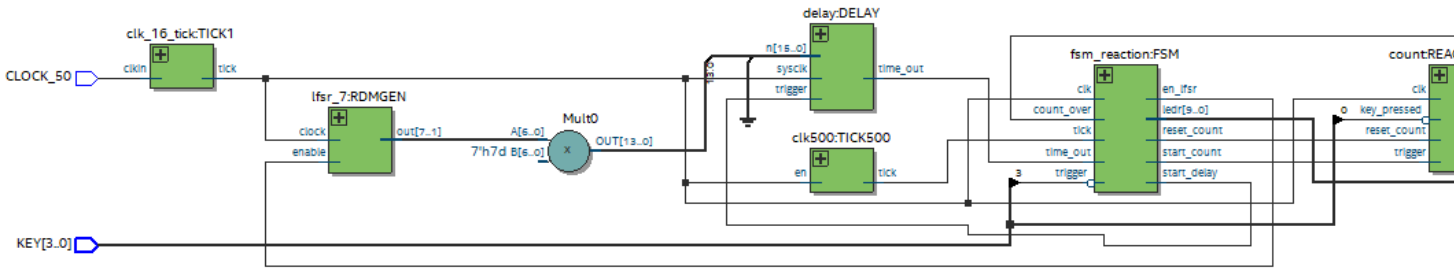
```
                    end
                LIGHT2: begin
                            ledr <= 10'b1110000000; //LED9-7
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b0;
                            end
                LIGHT3: begin
                            ledr <= 10'b1111000000; //LED9-6
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b0;
                            end
                LIGHT4: begin
                            ledr <= 10'b1111100000; //LED9-5
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b0;
                            end
                LIGHT5: begin
                            ledr <= 10'b1111110000; //LED9-4
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b0;
                            end
                LIGHT6: begin
                            ledr <= 10'b1111111000; //LED9-3
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b0;
                            end
                LIGHT7: begin
                            ledr <= 10'b1111111100; //LED9-2
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b0;
                            end
                LIGHT8: begin
                            ledr <= 10'b1111111110; //LED9-1
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b0;
                            end
                LIGHT9: begin
                            ledr <= 10'b1111111111; //LED9-0
                            en_lfsr <= 1'b0;
                            start_delay <= 1'b1;
                            end

                   default: ;
               endcase
    end

endmodule
```

Experiment 9 (Optional): A reaction meter

Sunday, December 8, 2019 1:53 PM

For Experiment 9, we were tasked with incorporating a Reaction Meter into out design from should count the time between all the LEDs turning OFF and the user pressing KEY[0]. The re random delay is then output on the 7 segment displays.

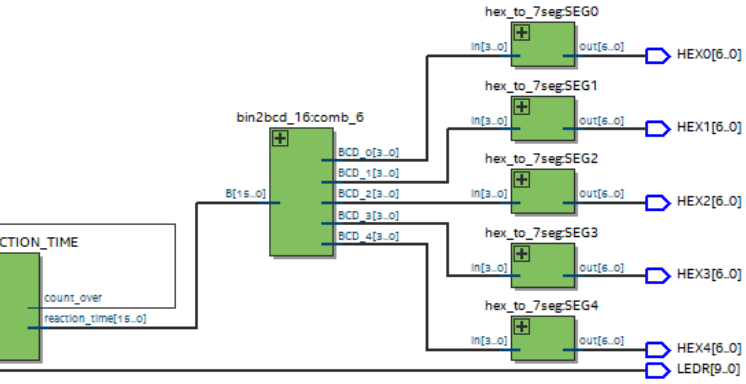
After analyzing the task, we decided that a **count** module to measure the reaction time of th schematic of our final design can be seen below:



This meant that we required extra outputs from the FSM to control the enable and reset of additional trigger signal for KEY[0]. The first step was to design the count module:

COUNT

```
module count(clk, key_pressed, trigger, reset_count, reaction_time, count_over);

    input clk;      // input clk is tick_1ms
    input key_pressed;  //  input [3:0] KEY;
    input trigger;
    input reset_count;
    output  count_over;
```


hex_to_7seg:SEG0
in[3..0]
out[6..0]
HEX0[6..0]
hex_to_7seg:SEG1
in[3..0]
out[6..0]
HEX1[6..0]
bin2bcd_16:comb_6
BCD_0[3..0]
BCD_1[3..0]
BCD_2[3..0]
BCD_3[3..0]
BCD_4[3..0]
B[15..0]
hex_to_7seg:SEG2
in[3..0]
out[6..0]
HEX2[6..0]
hex_to_7seg:SEG3
in[3..0]
out[6..0]
HEX3[6..0]
CTION_TIME
count_over
reaction_time[15..0]
hex_to_7seg:SEG4
in[3..0]
out[6..0]
HEX4[6..0]
LEDR[9..0]

```
    input reset_count;
    output  count_over;

    parameter BIT_SZ = 16;
    output reg [BIT_SZ-1:0] reaction_time;


    initial reaction_time = 0;

    reg count_over;

    always @ (posedge clk)
          if (reset_count == 1'b1)  // key not pressed
             reaction_time <= 0;
          else if (trigger == 1'b1 && key_pressed == 0)  // key not pressed
          begin
             reaction_time <= reaction_time + 1'b1;
             count_over <= 0;
          end
          else if (trigger == 1'b1 && key_pressed == 1'b1) // key pressed
             count_over <=  1;

endmodule
```

The next step was to edit the FSM to incorporate the addition of the count module.

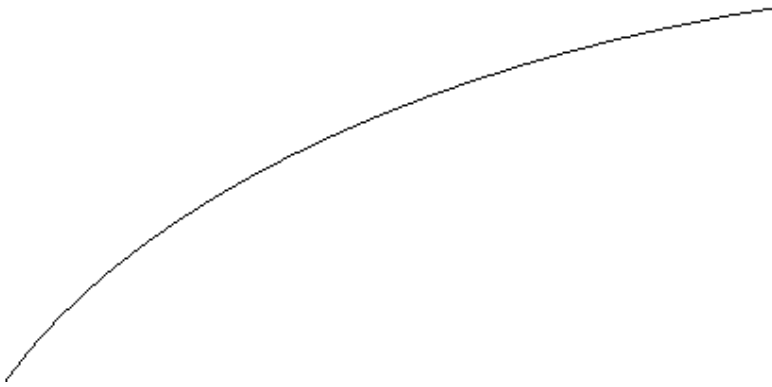
FSM

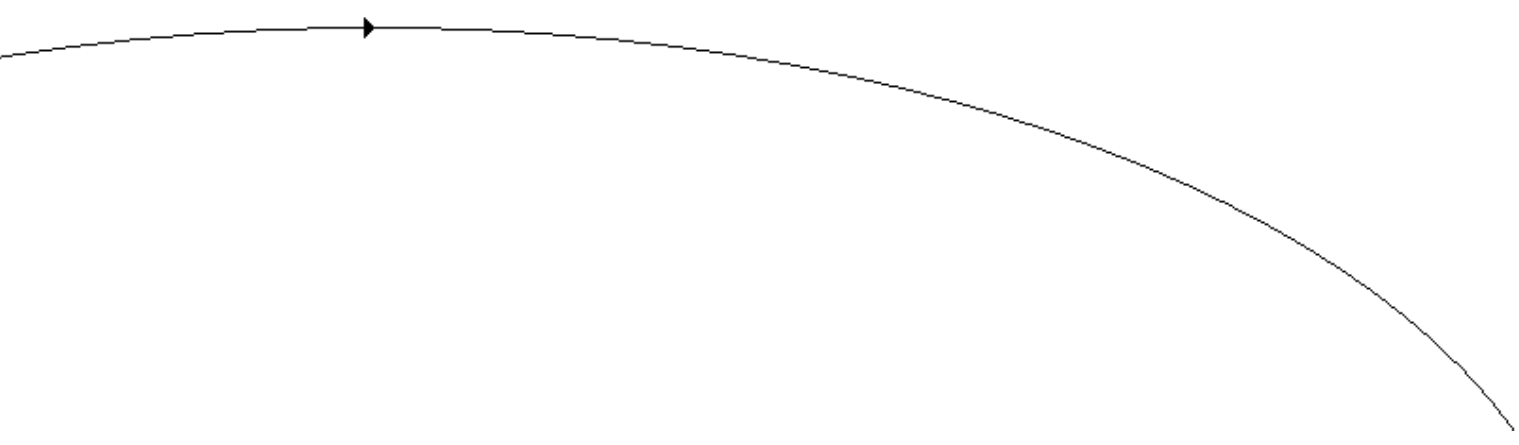
```
  input clk, tick, trigger,time_out;
  output en_lfsr, start_delay;
  output [9:0] ledr;


  input count_over;
  output reg start_count;
  output reg reset_count;


  reg [9:0] ledr;
  reg en_lfsr, start_delay;

  //Defining each of our states using one-hot encoding
  parameter IDLE = 12'b000000000001;
  parameter LIGHT0 = 12'b000000000010;
  parameter LIGHT1 = 12'b000000000100;
  parameter LIGHT2 = 12'b000000001000;
  parameter LIGHT3 = 12'b000000010000;
  parameter LIGHT4 = 12'b000000100000;
  parameter LIGHT5 = 12'b000001000000;
  parameter LIGHT6 = 12'b000010000000;
  parameter LIGHT7 = 12'b000100000000;
  parameter LIGHT8 = 12'b001000000000;
  parameter LIGHT9 = 12'b010000000000;
  parameter REACTION = 12'b100000000000;

  //setting up initial state for switch case
  reg [11:0] state;
  initial state = IDLE;
  initial en_lfsr = 1'b0;
```

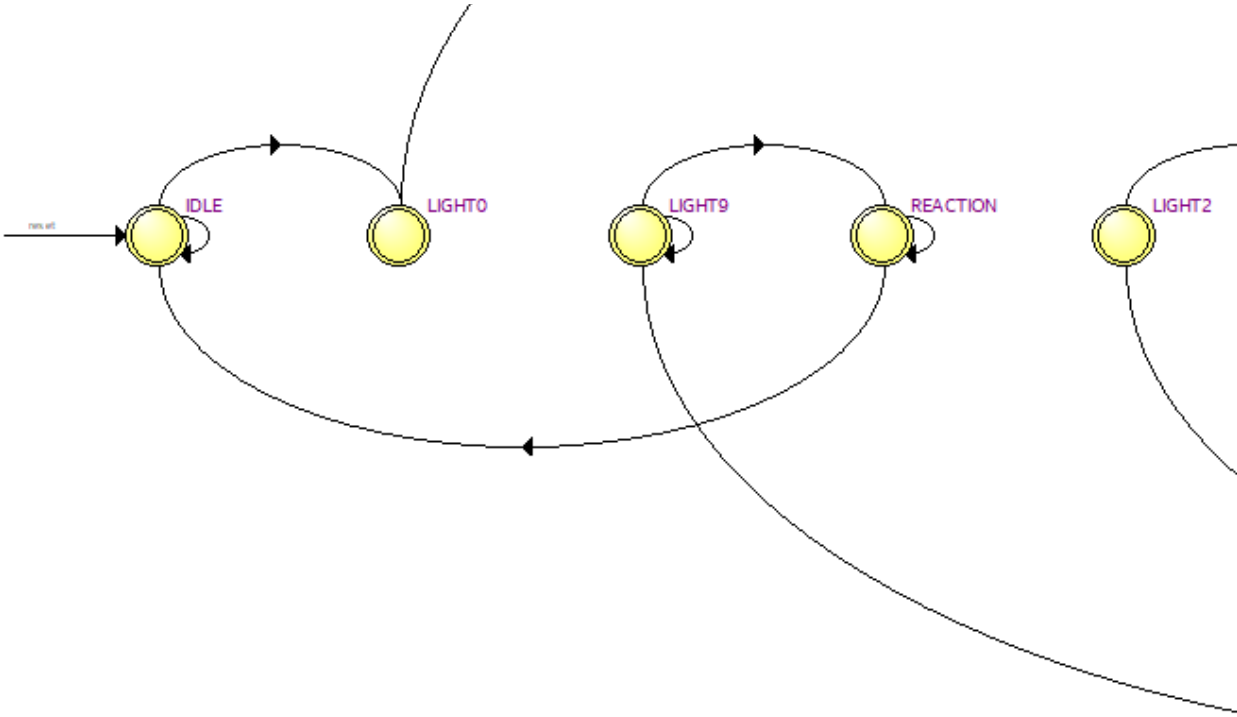
```
initial start_delay = 1'b0;
initial start_count = 1'b0;
initial reset_count = 1'b0;

//transitions
always @ (posedge tick)
begin
        case(state)
           IDLE: if(trigger == 1'b1) state<= LIGHT0;
           LIGHT0: if(tick == 1'b1) state <= LIGHT1;
           LIGHT1: if(tick == 1'b1) state <= LIGHT2;
           LIGHT2: if(tick == 1'b1) state <= LIGHT3;
           LIGHT3: if(tick == 1'b1) state <= LIGHT4;
           LIGHT4: if(tick == 1'b1) state <= LIGHT5;
           LIGHT5: if(tick == 1'b1) state <= LIGHT6;
           LIGHT6: if(tick == 1'b1) state <= LIGHT7;
           LIGHT7: if(tick == 1'b1) state <= LIGHT8;
           LIGHT8: if(tick == 1'b1) state <= LIGHT9;
           LIGHT9: if(time_out == 1'b1) state <= REACTION;
           REACTION: if(count_over == 1'b1) state <= IDLE;
           default: ;
        endcase
end

//Output for each state
always @ (*)
begin
        case(state)
           IDLE: begin //no lights
                        ledr <= 10'b0000000000;
                        en_lfsr <= 1'b1; //request random number
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
           LIGHT0: begin //start lights
                        ledr <= 10'b1000000000; //LED9
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b1;
                        end
           LIGHT1: begin
                        ledr <= 10'b1100000000; //LED9-8
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
           LIGHT2: begin
                        ledr <= 10'b1110000000; //LED9-7
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
           LIGHT3: begin
                        ledr <= 10'b1111000000; //LED9-6
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
           LIGHT4: begin
                        ledr <= 10'b1111100000; //LED9-5
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
```

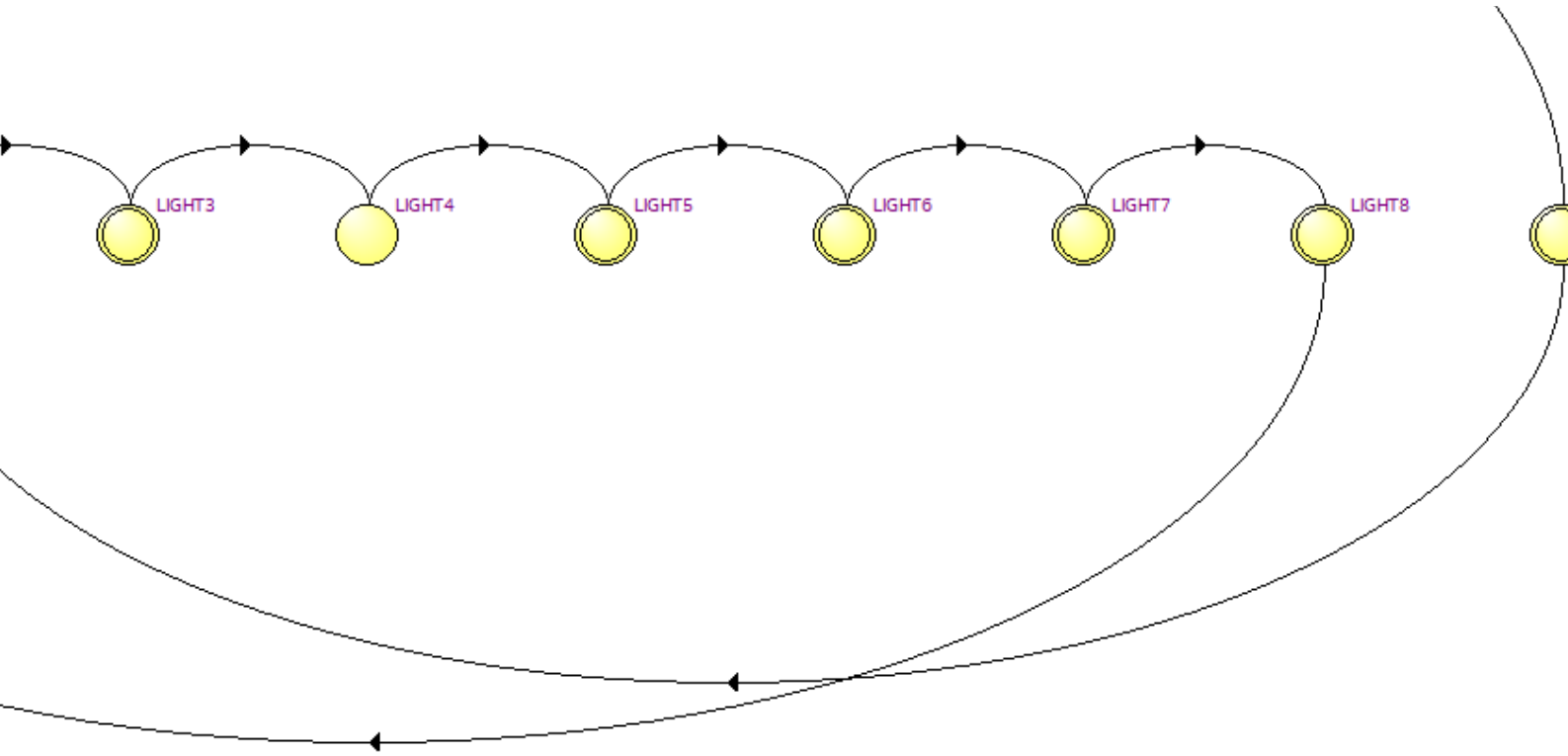
```
                end
            LIGHT5: begin
                        ledr <= 10'b1111110000; //LED9-4
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
            LIGHT6: begin
                        ledr <= 10'b1111111000; //LED9-3
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
             LIGHT7: begin
                        ledr <= 10'b1111111100; //LED9-2
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
             LIGHT8: begin
                        ledr <= 10'b1111111110; //LED9-1
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
             LIGHT9: begin
                        ledr <= 10'b1111111111; //LED9-0
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b1;
                        start_count  <= 1'b0;
                        reset_count <= 1'b0;
                        end
             REACTION: begin
                        ledr <= 10'b0000000000; //LED OFF
                        en_lfsr <= 1'b0;
                        start_delay <= 1'b0;
                        start_count  <= 1'b1;
                        reset_count <= 1'b0;
                        end

                default: ;
            endcase
    end

endmodule
```

This provided us with the following state diagram:






IDLE
LIGHT0
LIGHT9
REACTION
LIGHT2


LIGHT3
LIGHT4
LIGHT5
LIGHT6
LIGHT7
LIGHT8


LIGHT1

Experiment 10: Interface with the MCP4911 Digital Converter

Sunday, December 8, 2019 1:53 PM

We began this experiment looking at the add-on card to the DE1 board. It held a 10-bit ADC sockets for each.

Having our own add-on card unique to the course was extremely beneficial, as it gave us an interface with the peripherals of our FPGA using the SPI.

The focus of this experiment was thus to introduce us to:

- Understanding and verifying the **operation of the Serial-to-Parallel Interface** (SPI) of converter (DAC) MCP4911 using Modelsim
- Testing the DAC and **measured its output voltage range**

The focus was on understanding our MCP4911 Digital-to-Analogue Converter. As with all for understand it's behavior before proceeding. For that reason, we started by looking at our da

1.0 ELECTRICAL CHARACTERISTICS

Absolute Maximum Ratings †

V_{DD} 6.5V
All inputs and outputs w.r.t ..... V_{SS} –0.3V to V_{DD}+0.3V
Current at Input Pins ..... ±2 mA
Current at Supply Pins ..... ±50 mA
Current at Output Pins ..... ±25 mA
Storage temperature ..... -65°C to +150°C
Ambient temp. with power applied ..... -55°C to +125°C
ESD protection on all pins ..... ≥ 4 kV (HBM), ≥ 400V (MM)
Maximum Junction Temperature (T_J) ..... +150°C

† **Notice:** Stresses above those listed under "Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at those or any other conditions above those indicated in the operational listings of this specification is not implied. Exposure to maximum rating conditions for extended periods may affect device reliability.

ELECTRICAL CHARACTERISTICS

Electrical Specifications: Unless otherwise indicated, V_{DD} = 5V, V_{SS} = 0V, V_{REF} = 2.048V, Output Buffer Gain (G) = 2x, R_L = 5 kΩ to GND, C_L = 100 pF T_A = -40 to +85°C. Typical values are at +25°C.

Parameters	Sym	Min	Typ	Max	Units	Conditions
Power Requirements						
Operating Voltage	V_{DD}	2.7	—	5.5		
Supply Current	I_{DD}	—	175	350	µA	V_{DD} = 5V

-to-Analogue

and 10-bit DAC, along with

insight into how we can

the digital-to-analogue

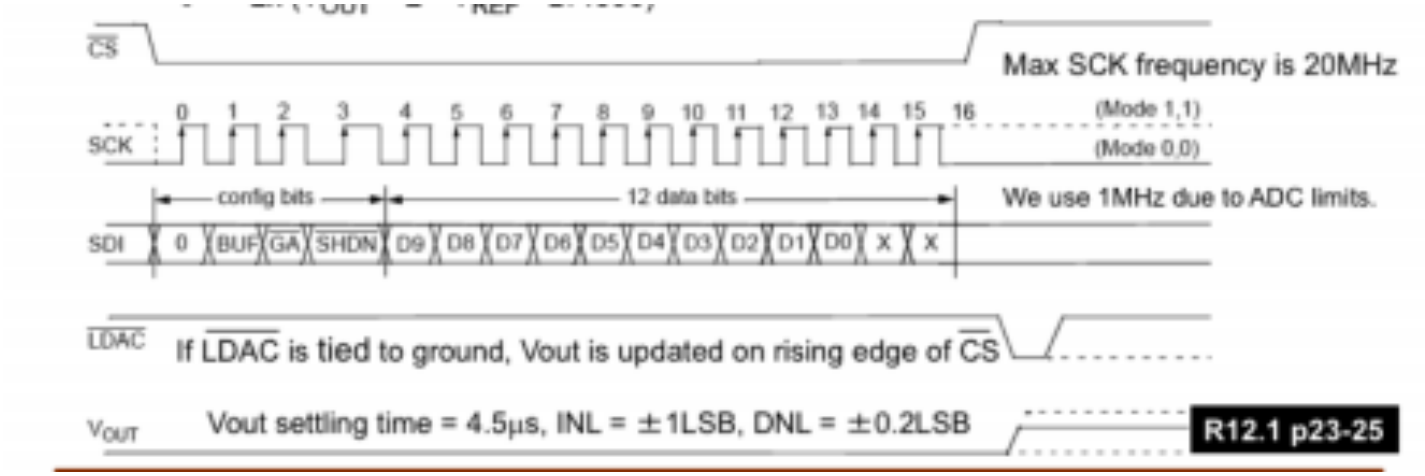
reign devices, we should first
atasheet.

		—	125	250	µA	V_{DD} = 3V V_{REF} input is unbuffered, all digital inputs are grounded, all analog outputs (V_{OUT}) are unloaded. Code = 0x000h

1. **Understanding datasheet**

We were able to synthesize the datasheet information as follows:

- PURPOSE OF DAC PINS
  - Vdd, power on.
  - Vss, power reset
  - Chip Select (CS) signal goes low, indicating start of data
  - SDI, serial data input, the stream of 15 bits that are sampled
  - By the clock, SCK
- Information sent to the DAC using SDI pin:



- DAC configuration: first bit set to zero, config bits until clock 3.
  - Remaining 12 bits used for data, which is to be converted to analog information
  - BUF: set to 1 for transfer of voltage
  - GA set to 0 as Electrical specifcation states it is Output Buffer Gain of 2x
  - Bit 12: SHDN, set to 1 as we are using the DAC.

2. **Timing Diagram**

Having understood the general functions of our datasheet, we were now concerned with ou
and its interaction with said DAC.

Understanding that the spi2dac module takes a 10-bit number in parallel, and generates ser
we drew a timing diagram to assert our understanding of this for the word 10'h23b.

We proceed to developing the timing diagram, in the hopes of understanding the flow of ou

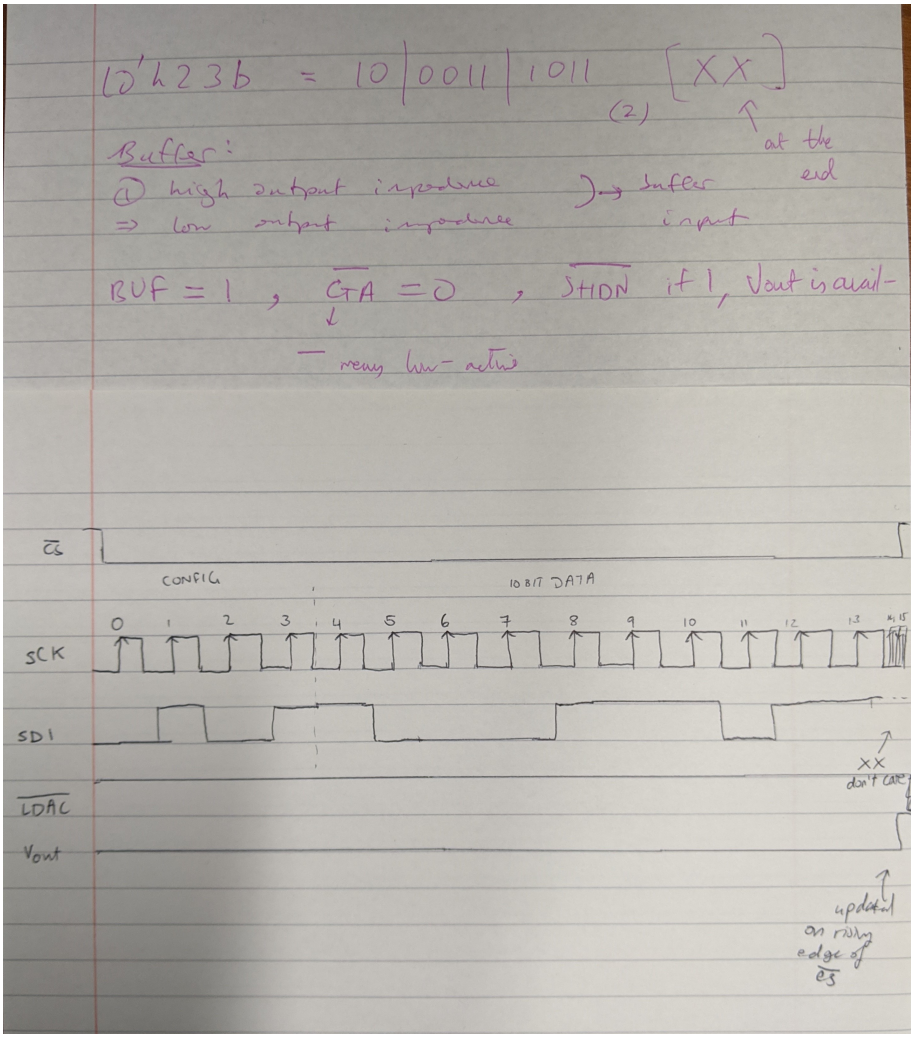
n

ur Verilog module spi2dac

rial signals to drive the DAC,

ur module.

Timing diagram for sending of 10'h23b word

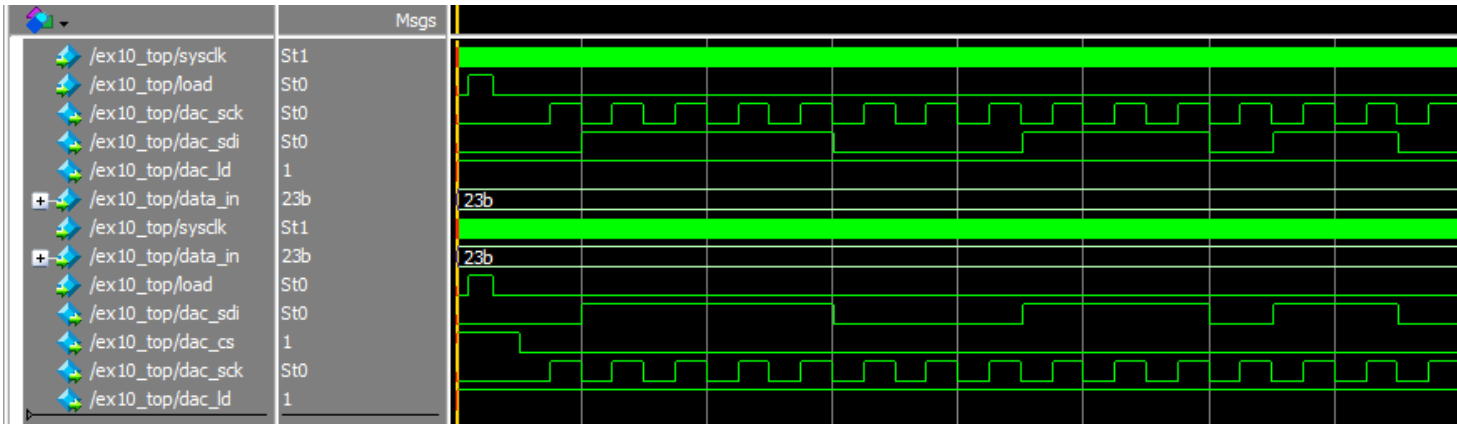


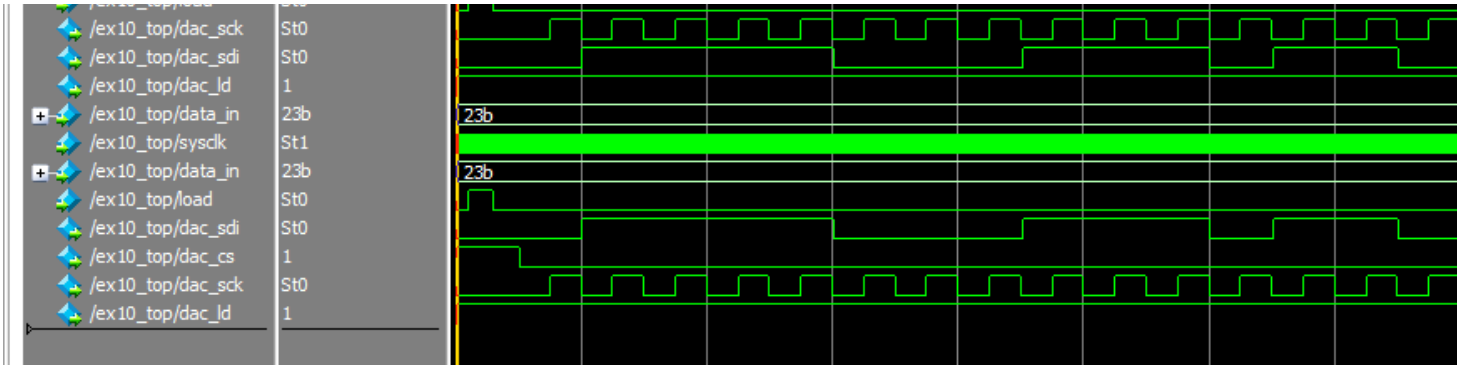
We felt we had some sort of intuition, but now really wanted to see that we were getting a

3. **Timing diagram on Modelsim**

To first generate a timing diagram, we had to set the parameters for our testbench. This wa generated our clock and other signal values as follows.

INSERT DO FILE





Our timing diagram was correct for the most part, bar our 3rd config bit- ~GA. We thought to follow what was said in the datasheet: "2x unless otherwise stated". However we understood that in fact ~GA meant it should be set to 0 if we want to shutdown our device to conserve power, and should be one if we are using the DAC.

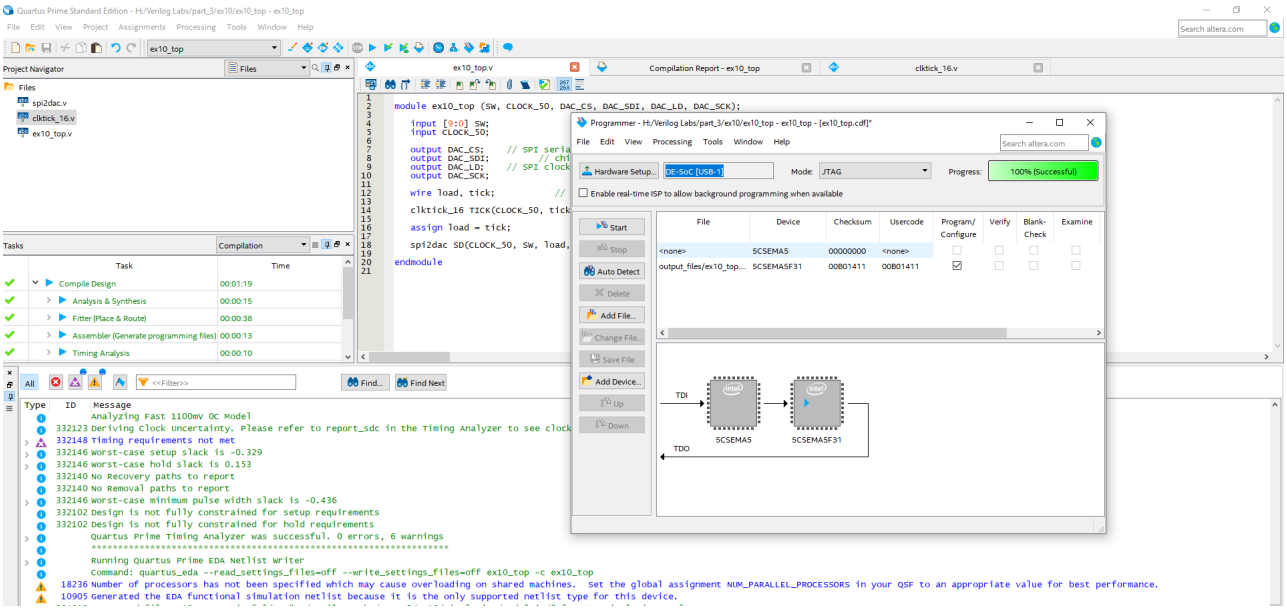
From the datasheet, we believed that this bit should be 1 for MCP4911. It was only after seeing it in reality with our spi2dac module did we understand that in our timing diagram it should have been 1. This displayed a real strength of Modelsim to visualize the behavior of our program.

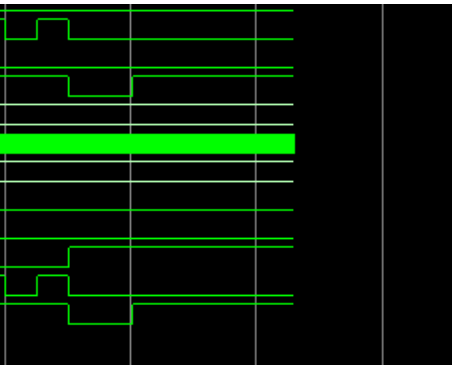
After understanding our data signals and viewing their correct waveforms, we needed to test our DAC on DE1.

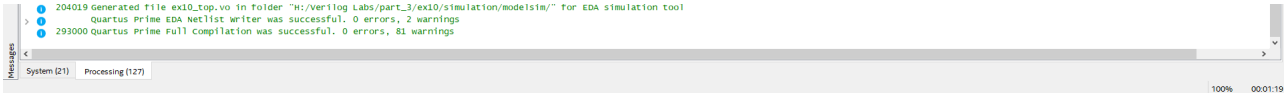
4. Testing DAC on DE1

After completing our pin assignment and assigning our given top-level entity, we proceeded to place & route our design.

Successful Place & Route





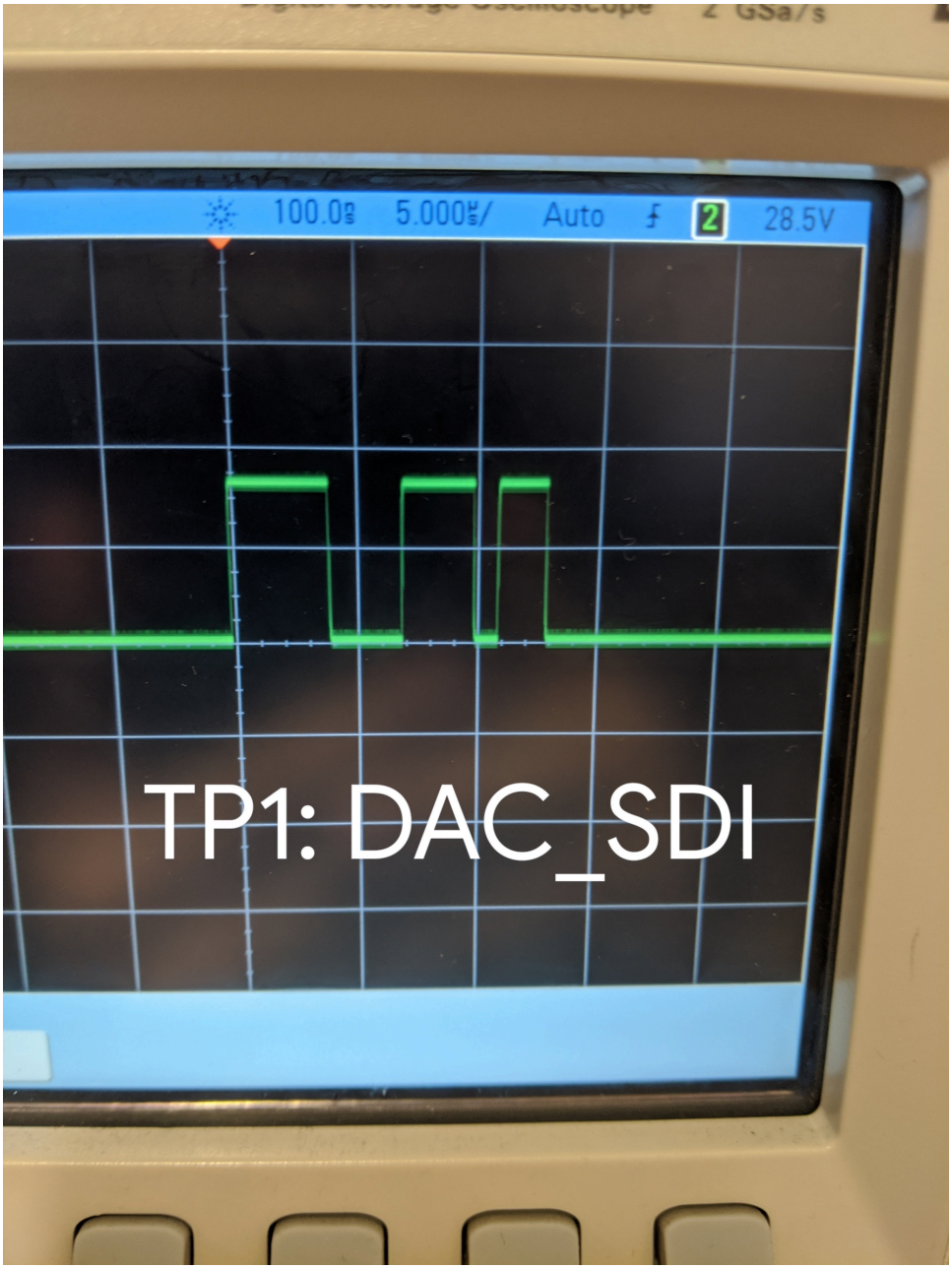


Everything worked fine... on the surface! It would take an analysis of our analogue out signal to truly know whether it was working or not.

5. **Verifying signals on oscilloscope**

After triggering our signal on our scope, we were very happy to see our word 10'h23b was sent to our DAC! It was shown here on our DAC output.

DAC OUTPUT FOR SW SET TO 23B



Experiment 11: D-to-A conversion u

Sunday, December 8, 2019 1:54 PM

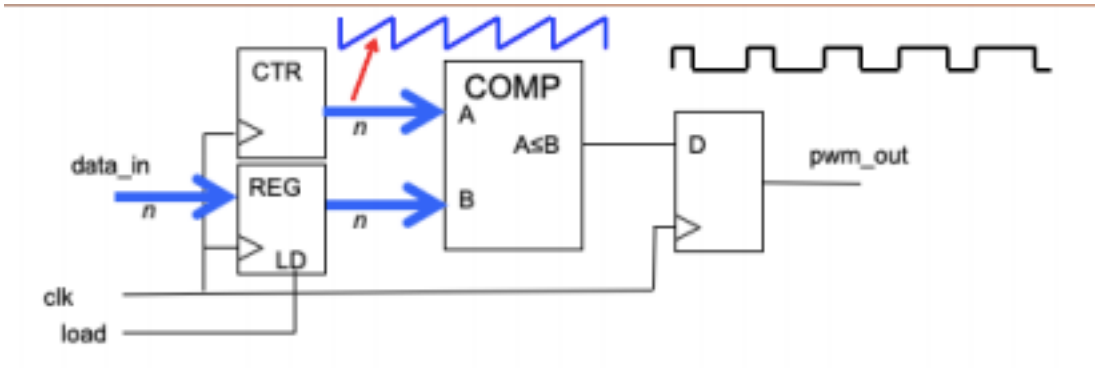
Key outcome from this experiment:

- Test the DAC and **measure its output voltage range**

An alternative to using a DAC chip (SPI serial interface to communicate with chip), we cou
analogue output from a digital number using pulse-width modulation (PWM).

Whilst the concept of a varying width displaying a real signal seemed difficult at first, the i
in fact quite straightforward, making it a clearly useful tool if we had no resources to set u
DAC chip.

The block diagram of a PWM is as follows:



And the Verilog code is even more simple:

```
module pwm(CLOCK_50, SW, load, pwm_out);

    input CLOCK_50; //system clock
    input [9:0] SW; //input data for conversion
    input load;//high pulse to load
    output pwm_out; //PWM output

    reg [9:0] d; //internal register
    reg [9:0] count; //internal 10-bit counter
    reg pwm_out;

    always @ (posedge CLOCK_50)
       if (load == 1'b1) d <= SW;
```

sing pulse modulation

ld just directly produce an

implementation in Verilog was
up the SPI interface with the

```
    initial count = 10'b0;

    always @ (posedge CLOCK_50) begin
        count <= count + 1'b1;
        if(count > d)
            pwm_out <= 1'b0;
        else
            pwm_out <= 1'b1;
        end
endmodule
```

We had two questions

1. What was the effect of the low-pass filter in this circuit?
2. How would the voltage range change from the DAC vs. PWM?

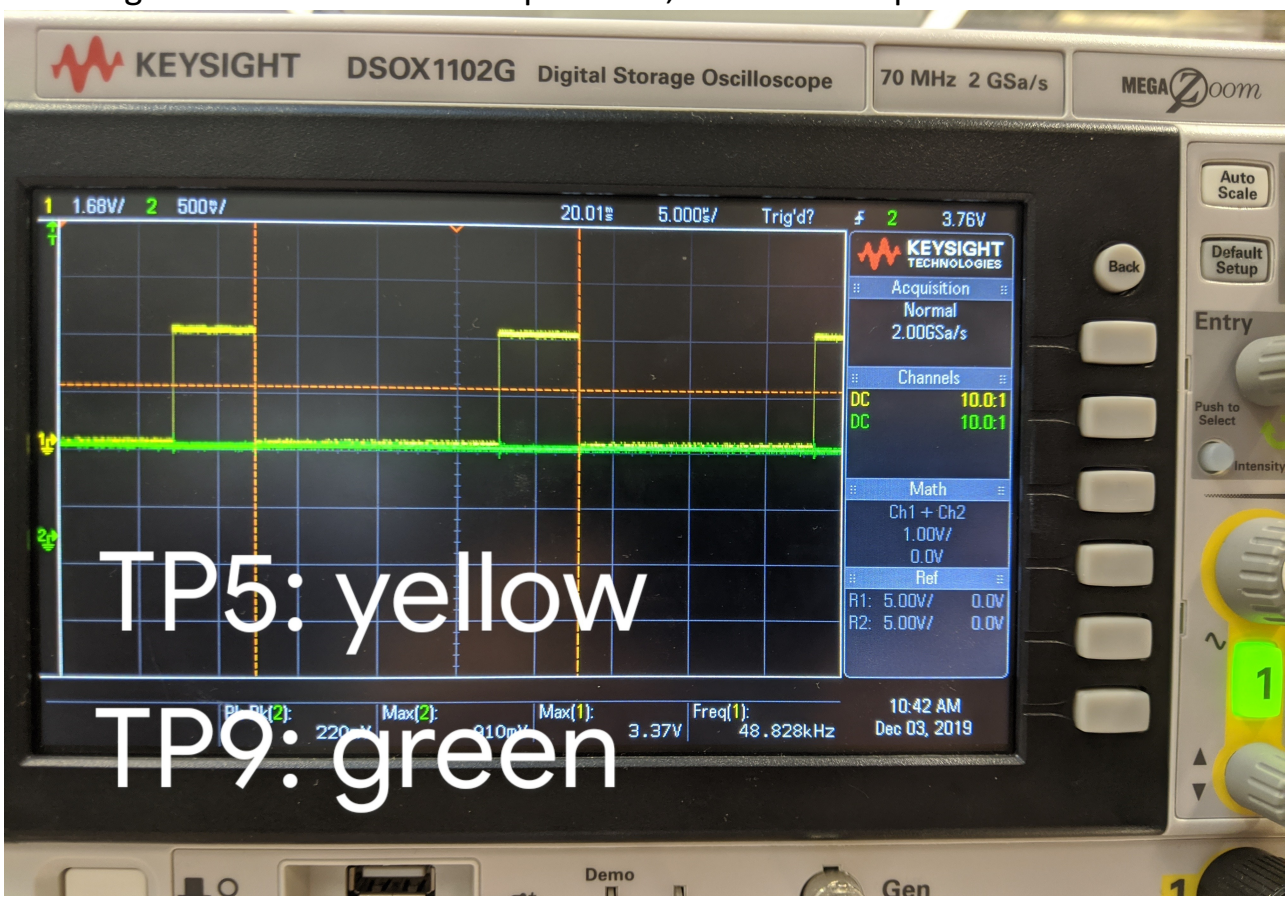
Examine signals at TP5 & TP8

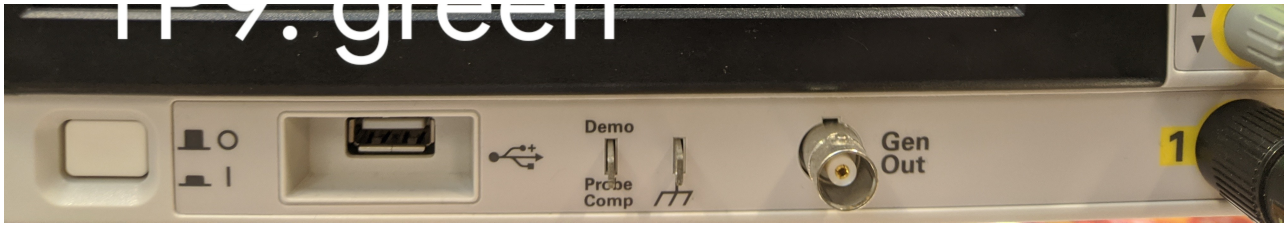
- TP5 – gives us our pw5_out before the low pass filter
- TP5 - analogue out right channel (after passing though the add-on board)

Compare output voltage ranges at TP8 & TP9:

- TP8 ranges from -0.17V to 3.10V
- TP9 ranges from 40mv to 3.38V (TP5 ranges from 3.30V to 3.47V

We sought out to answer these questions, and first compared TP5 to TP9 to see the effect

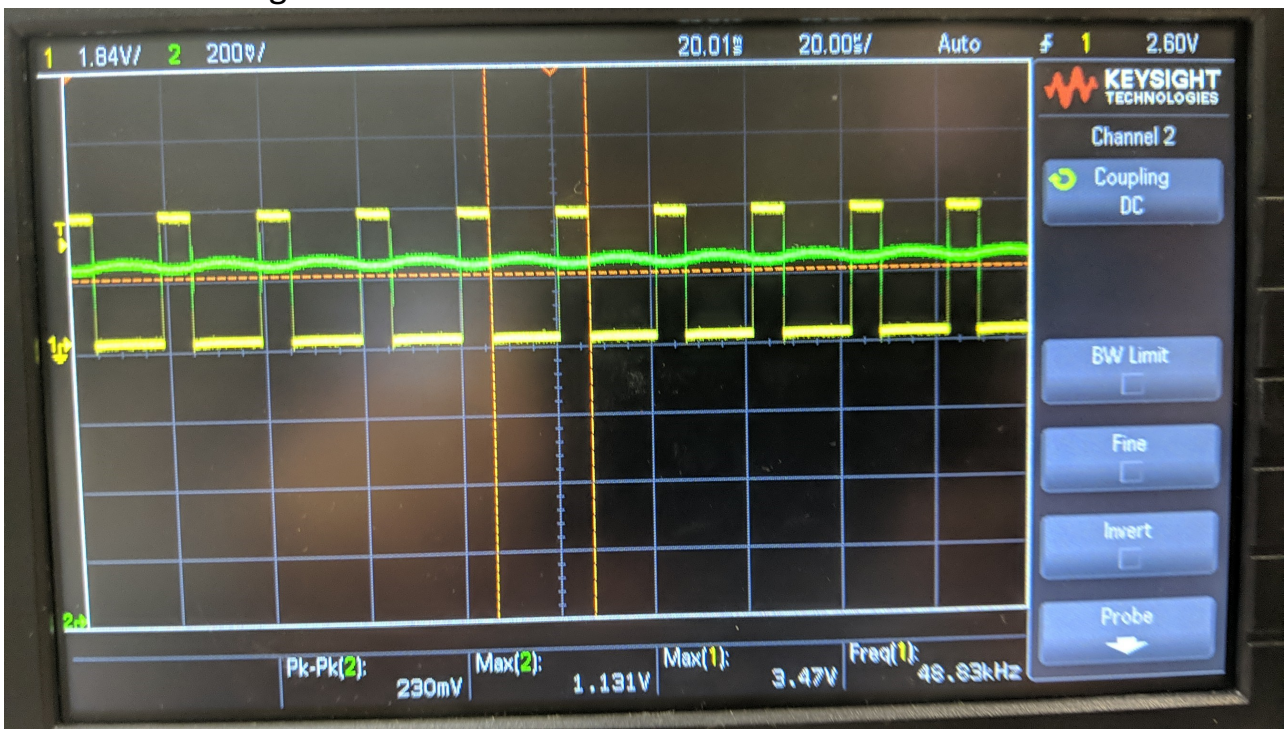




We looked into TP5 further to understand how PWM affects the pulse width.

The higher the bit, the wider the pulse.

This leads to a higher value at TP8



Displaying to us the effect of PWM on DAC.

Experiment 12: Designing and testing a sinewave table in ROM

Sunday, December 8, 2019 1:55 PM

Key outcomes of Experiment 12:

- Work with using a **ROM** in Verilog

This experiment provides us with a new tool: the Read-Only Memory (ROM).

We will achieve our initial aim of learning how to use a **ROM** generating a table of sine values hard-coded into memory.

The relationship between the ROM content and address was given, as follows:

$$D[9:0] = \text{int}(511*\sin(A[9:0]*2*\pi/1024)+512) \qquad \text{for } 1023 \geq A[9:0] \geq 0$$

1. **Understanding Memory Initialization**

We are designing a 1K x 10 bit ROM (I.e. 1000 memory addresses, each holding 10bits). If we wanted to hard-code each value into this ROM ourselves, it could take a ~Veri~ long time! Besides the time factor, we are also much more prone to human errors if we hardcoded each address ourselves. Its clear some form of automation is necessary.

For that reason, our attention is drawn to the idea of Scripting. We were kindly provided with two scripts which essentially generate the same memory we are looking for:

Python Script

```
# sinegen.py - Generate sinewave table
#  ... for use with Altera ROMs
#
from math import sin, cos, radians

DEPTH = 1024  # Size of ROM
WIDTH = 10    # Size of data in bits
OUTMAX = 2**WIDTH - 1   # Amplitude of sinewave
```

```
10  filename = "rom_data.mif"
11  f = open(filename,'w')
12
13  #  Header for the .mif file
14  print >> f, "-- ROM Initialization file\n"
15  print >> f, "DEPTH = %d;" % DEPTH
16  print >> f, "WIDTH = %d;" % WIDTH
17  print >> f, "ADDRESS_RADIX = HEX;"
18  print >> f, "DATA_RADIX = HEX;\n"
19  print >> f, "CONTENT\nBEGIN\n"
20
21  for address in range(DEPTH):
22      angle = (address*2*pi)/DEPTH
23      sine_value = sin(angle)
24      data = int((sine_value)*0.5*OUTMAX)+OUTMAX/2
25
26      print "%4x : %4x;" % (address, data)
27      print >> f, "%4x : %4x;" % (address, data)
28
29  print >> f, "END;\n"
30
31  f.close()
```

The function of this script is clear: it iteratively generates sine values and inputs them into our file "rom_data.mif"

Here's a general breakdown:

- Line 4 : necessary libraries for script operations
- Line 6-8: parameter values
- Line 14-19: .mif file header
- Line 21-2: calculation and inser
    - Calculation: for each address in ROM, calculate:

$$D[9:0] = \text{int}(511*\sin(A[9:0]*2*pi/1024)+512) \qquad \text{for } 1023 \geq A[9:0] \geq 0$$

Having understood how the .mif file could be made automatically, we proceeded to look into whether the .mif file took

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0c0	3D8	3D9	3DA	3DC	3DD	3DE	3DF	3E0

0c8	3E1	3E2	3E3	3E4	3E5	3E6	3E7	3E8
0d0	3E9	3EA	3EB	3EC	3EC	3ED	3EE	3EF
0d8	3F0	3F0	3F1	3F2	3F3	3F3	3F4	3F5
0e0	3F5	3F6	3F6	3F7	3F7	3F8	3F9	3F9
0e8	3F9	3FA	3FA	3FB	3FB	3FC	3FC	3FC
0f0	3FD	3FD	3FD	3FD	3FE	3FE	3FE	3FE
0f8	3FE	3FF	3FF	3FF	3FF	3FF	3FF	3FF
100	3FF	3FF	3FF	3FF	3FF	3FF	3FF	3FF
108	3FE	3FE	3FE	3FE	3FE	3FD	3FD	3FD
110	3FD	3FC	3FC	3FC	3FB	3FB	3FA	3FA
118	3F9	3F9	3F9	3F8	3F7	3F7	3F6	3F6
120	3F5	3F5	3F4	3F3	3F3	3F2	3F1	3F0
128	3F0	3EF	3EE	3ED	3EC	3EC	3EB	3EA
130	3E9	3E8	3E7	3E6	3E5	3E4	3E3	3E2

A quick glance at this data showed its periodic nature as the values tended upwards and downwards, suggesting a successful sinewave. Looking at numbers was nice, but as always- we can only be sure once we put it to test!

2. **Practical testing of ROM functionality in Verilog**

Having initialized our memory with the mif file, we were still interested in integrating it with our FPGA.

This started by creating our 1-port ROM in Verilog:

I/O sizes

```
module ROM (
    address,
    clock,
    q);
```

```
   input [9:0]  address;
   input    clock;
   output    [9:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
   tri1     clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

   wire [9:0] sub_wire0;
   wire [9:0] q = sub_wire0[9:0];
```

1000 addresses (1024):

```
altsyncram_component.numwords_a = 1024,
```

Using our .mif file

```
altsyncram_component.init_file = "./rom_data.mif/rom_data.mif",
```

Having created our ROM 'block', we now had to integrate it into our program, where we could treat it as a regular module

```
1   module ex12_top(
2   CLOCK_50,
3   SW, HEX0, HEX1, HEX2, HEX3, HEX4
4   );
5       input CLOCK_50;
6       input [9:0] SW;
7       output [6:0] HEX0;
8       output [6:0] HEX1;
9       output [6:0] HEX2;
10      output [6:0] HEX3;
11      output [6:0] HEX4;
12
13      wire [9:0] ROM_OUT;
14      wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;
15
16      ROM rom1(SW, CLOCK_50, ROM_OUT);
17
18      bin2bcd_16 display({6'b0,ROM_OUT},BCD0,BCD1, BCD2, BCD3, BCD4);
19
20      hex_to_7seg seg0(HEX0, BCD0);
21      hex_to_7seg seg1(HEX1,BCD1);
22      hex_to_7seg seg2(HEX2, BCD2);
23      hex_to_7seg seg3(HEX3, BCD3);
24      hex_to_7seg seg4(HEX4, BCD4);
25
26  endmodule
```

This top function is quite primitive. The steps can be summarized as follows:

1. Switches SW[9:0] correspond to Address A[9:0], which is used to access ROM
    a. 50MHz clock allows us to access ROM value at clock edge (every 20ns)
2. Output of ROM, D[9:0], turned to BCD, then sent to 7-seg display

Experiment 13: A fixed frequency sinewave generator

Sunday, December 8, 2019 1:55 PM

Key outcomes of Experiment 13:

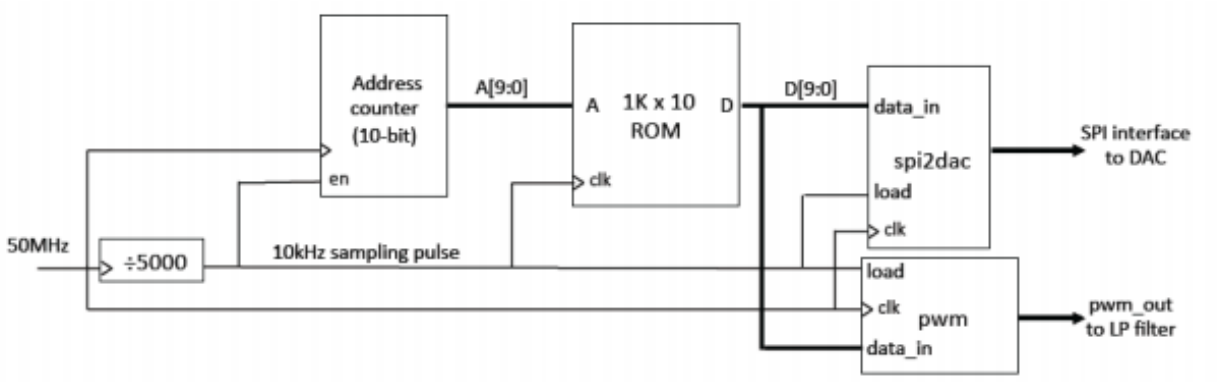
- Test the DAC and **measure its output voltage range**
- Work with using a **ROM** in Verilog

Instead of specifying and accessing a single ROM address, as in Exercise 12, what if we could access multiple values?

That is the focus of Experiment 13.

Here we replace our slide switches from Ex12 with a 10-bit binary counter, and connect our ROM data output to **spi2dac** and **pwm** modules.

The block diagram for this is as follows:



We implemented the circuit design specified in the schematic above as follows:

The code works as follows:

TOP FUNCTION:

```
4       output DAC_CS;
5       output DAC_SDI;
6       output DAC_LD;
7       output DAC_SCK;
8       output PWM_OUT;
```

```
10      wire tick,load;
11      wire [9:0] A;
12      wire [9:0] D;
13
14
15      clktick_16 TICK(CLOCK_50, tick);
16      assign load = tick;
17
18      reg reset_count;
19      initial reset_count = 1'b0;
20      counter_16 COUNT(CLOCK_50, load, reset_count, A);
21
22      ROM rom(A, load, D);
23
24      spi2dac SPI2DAC(CLOCK_50, D, load, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);
25
26      pwm PWM(CLOCK_50, D, load, PWM_OUT);
27
28      always @ (posedge CLOCK_50)
29      begin
30            if (reset_count == 16'd1023)
31               reset_count <= 1'b1;
32            else
33               reset_count <= 1'b0;
34      end
35
36  endmodule
37
```

Two "utility" modules were added to the function.

- Address counter
- 10kHz sampling pulse

We added this 10-bit counter as follows:

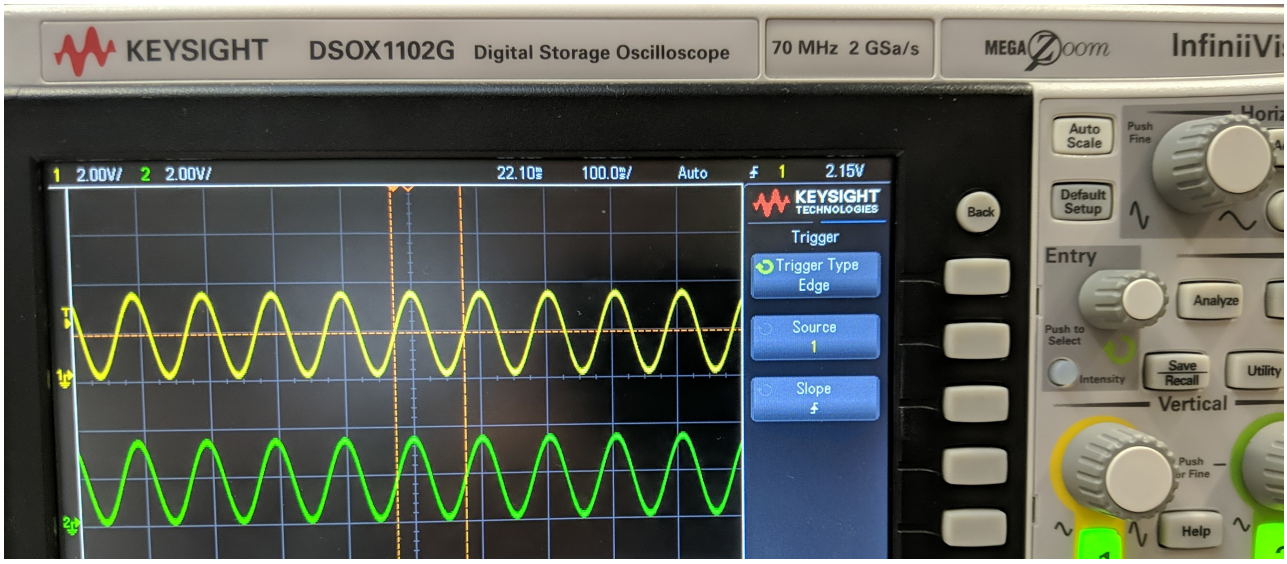
```
 1  `timescale 1ns / 100ps
 2
 3  module counter_16(
 4      clock,
 5      enable,
 6      reset,
 7      count
 8  );
 9
10  parameter BIT_SZ = 16;
11  input clock;
12  input enable;
13  input reset;
14  output [BIT_SZ-1:0] count;
15
16  reg [BIT_SZ-1:0] count;
17
18  initial count = 0;
19
20 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

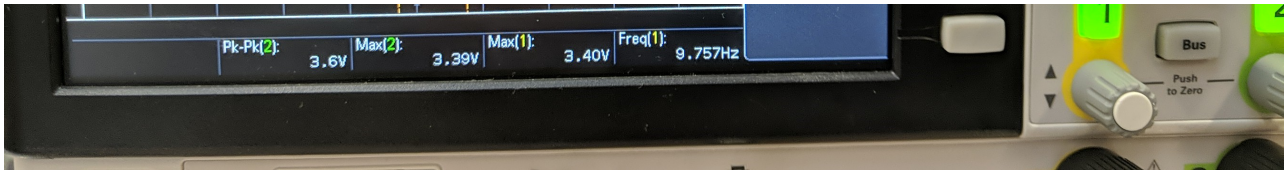
```
25        end
26    endmodule
27
```

```
 1  module clktick_16( //DIVIDE BY 500
 2  clkin,
 3  tick);
 4
 5      input clkin;
 6      output tick;
 7
 8      reg [13:0] count;
 9      reg tick;
10
11
12      initial tick = 1'b0;
13      initial count = 0;
14
15      always @ (posedge clkin)
16      begin
17            if (count == 3'd500)
18            begin
19                tick <= 1'b1;
20                count <= 0;
21            end
22            else
23            begin
24                count <= count + 1'b1;
25                tick <= 1'b0;
26            end
27      end
28
29  endmodule
30
```

Having implemented these modules, we had to test whether it was indeed working.

Hence, we connected to TP8 and TP9, checking if it was giving the elusive sine wave we were looking for





The sine waves we received at TP8 and TP9 were the same, both holding peak to peak of 3.6V, and the following frequencies:

DAC signal – Frequency: 9.76Hz
PWM signal – Frequency: 9.76Hz

We received two key takeaways from this experiment:

1. Our DAC and PWM modules didn't work "by luck"- they were both genuinely capable of taking any digital value and converting it into analog, with good accuracy. This was shown with how they both interpreted at the same 9.76Hz, similar to peers in the lab.
2. **The frequency of the sinewave in our 1K x 10 ROM is shown to be 9.76Hz**

Experiment 14 (Optional): A variable sinewave generator

Sunday, December 8, 2019 1:55 PM

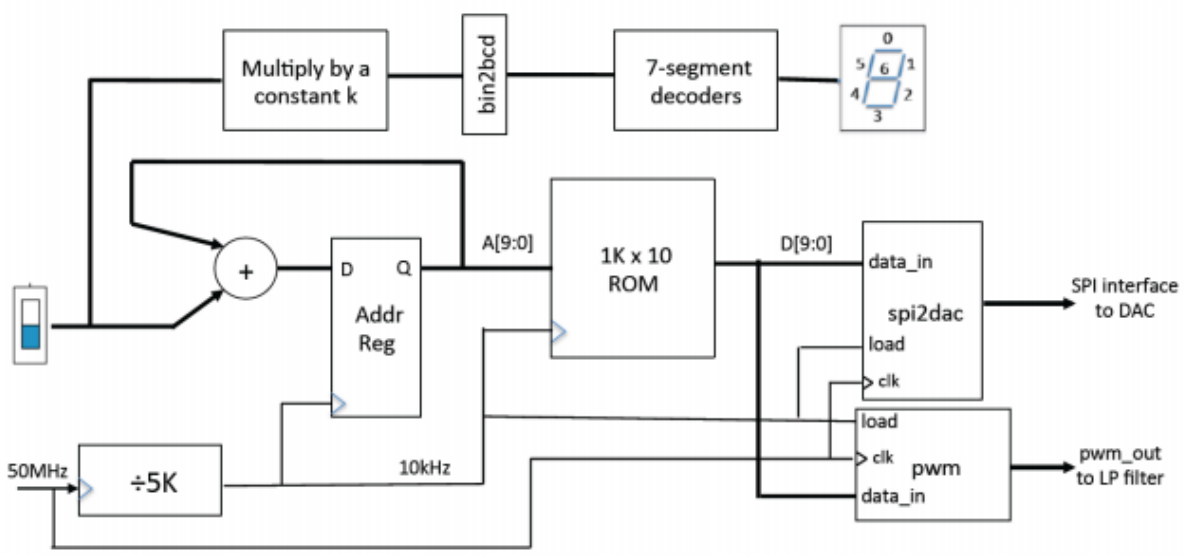
- Design a sinewave tone generator, with variable frequency, controlled by slide switches, and shown on 7-seg displays in decimal format

  In Experiment 12, we learnt how to use switches to read from a ROM and output it on a 7-seg display.

  In Experiment 13, we learnt how to count through a ROM using a counter, and output it onto two analogue outputs.

  Here, we are tasked to combine these 2 lessons to create a variable sinewave generator.

  The block diagram is as follows:



Lower part: understanding the frequency maneuver and modifying experiment 13
We immediately notice that we can reuse our bottom half (div5k, address counter, ROM, spi2dac, pwm) modules from experiment 14.

All we have to amend from it, is to add an adder block before the counter, adding our current address with the value in our switches SW[9:0].

our current address with the value in our switches SW[9:0].

Why? This allows us to "skip" values in our ROM at each iteration. For example, if our switches were set to 2, we would be reading **every 2** values in our ROM – while still at 10kHz. Hence, in this case we are doubling our frequency.
We can generalize this to any "skip" ***x*** is equivalent to a multiple of our function frequency.

Upper part: understanding the decimal multiplication and displaying necessary value
Whereas the bottom part is concerned with changing our output frequency, the top is simply in charge of displaying this change on a screen.

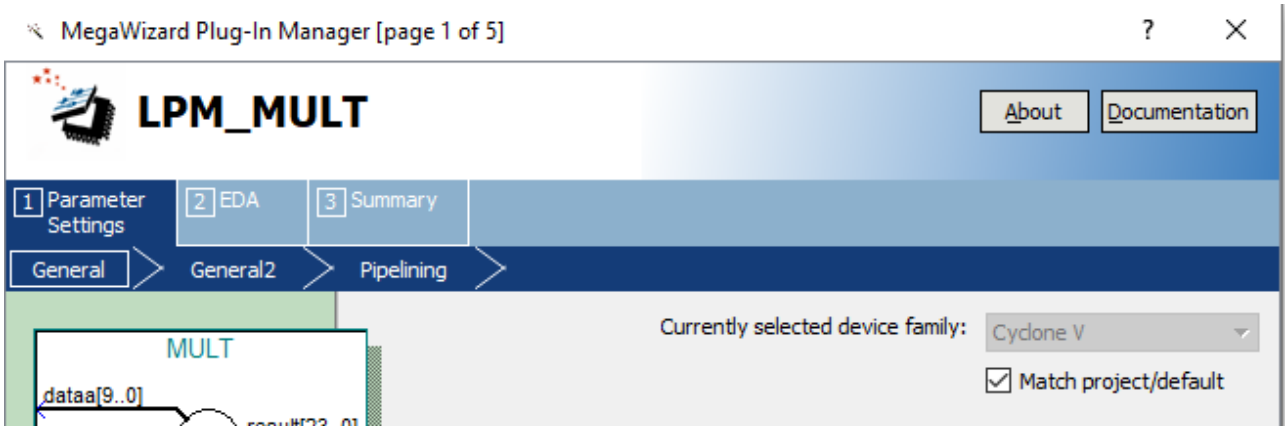
The first block of interest is ***Multiply by a constant k*** block. We can create this constant coefficient multiplier using the IP-catalog tool- but before doing so, we first need to decide on what exactly is our constant ***k*** !

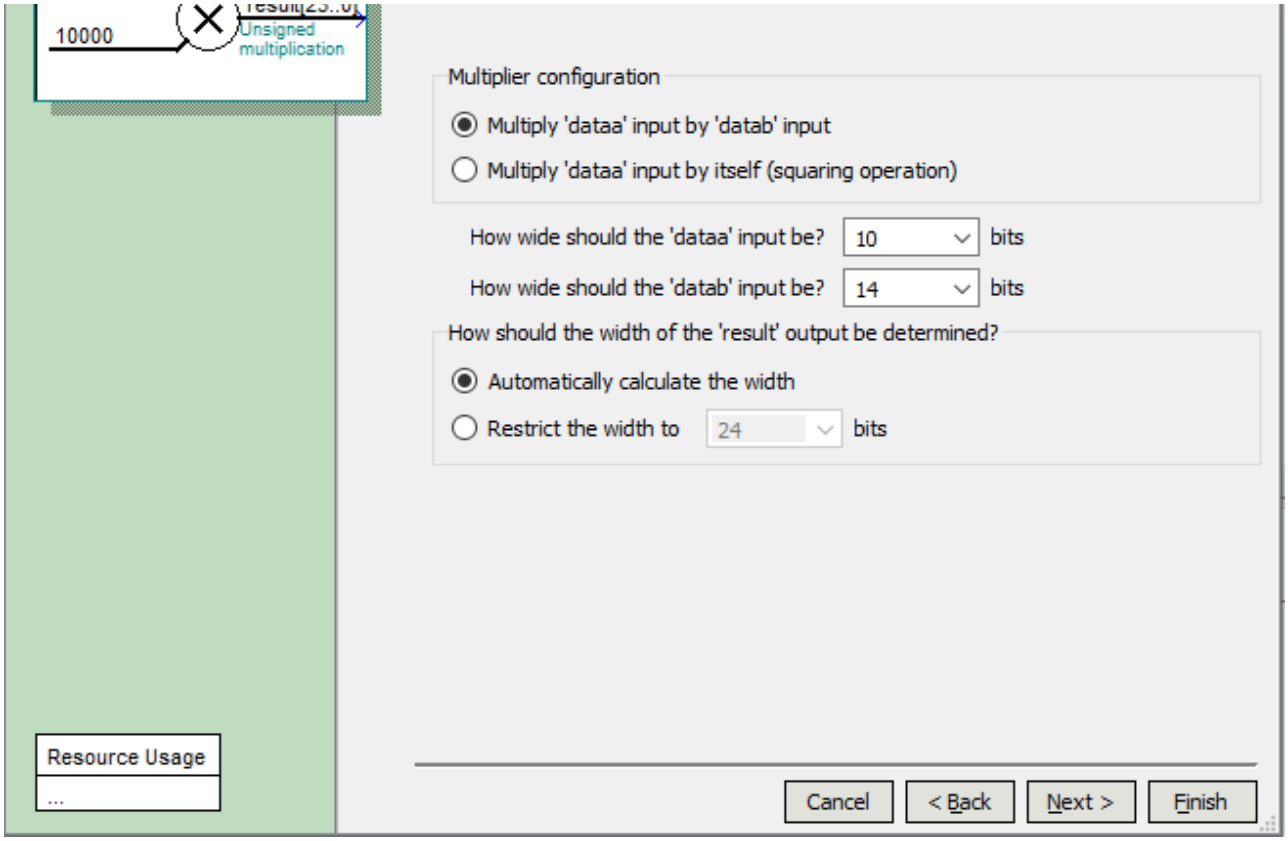
If the bottom part is focused on multiplying the frequency of the sinewave in our ROM, and last experiment we established that this frequency was 9.76Hz, we know this is what we need to multiply by!

We tried doing this in our IP catalog, but couldn't input a decimal value as datab.

We realized we could only do integer operations, so had to be a little more inventive.
It was here that we realized the importance of the top-14 bits and exactly why we were encouraged to multiply by the 14-bit constant 14'h2710. Because by multiplying by 14'h2710 (which is 14'd1000) and then keeping the top 14 bits, we were essentially bit shifting our product 10 bits to the right- dividing it by 2^10.

10000/(2^10) = 9.7656 -> the constant we were looking for!





Selection of top 14 bits:

```
33      bin2bcd_16 top14_to_bcd({2'b0, MULT_RES[23:10]}, BCD0, BCD1, BCD2, BCD3, BCD4);
```

Top function:

```
 1   module ex14_top(
 2   CLOCK_50,
 3   SW,
 4   HEX0, HEX1, HEX2, HEX3, HEX4,
 5   DAC_SDI, DAC_CS, DAC_SCK, DAC_LD,
 6   PWM_OUT);
 7
 8       input CLOCK_50;
 9       input [9:0] SW;
10
11       output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4;
12       output DAC_SDI, DAC_CS, DAC_SCK, DAC_LD, PWM_OUT;
13
14       wire TICK_10KHz;
15       wire [9:0] DATA;
16       wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;
17       wire [23:0] MULT_RES;
18
19       reg [9:0] COUNT;
20
21       initial COUNT = 10'b0;
22
23       clktick_16 div_5k(CLOCK_50, TICK_10KHz, 16'd5000, 1'b1);
24
25       ROM rom_data(COUNT, TICK_10KHz, DATA);
26
27       spi2dac analogue_out0(CLOCK_50, DATA, TICK_10KHz, DAC_SDI, DAC_CS, DA
28
29       pwm analogue_out1(CLOCK_50, DATA, TICK_10KHz, PWM_OUT);
```

```
30
31      MULT times_10k(SW,MULT_RES);
32
33      bin2bcd_16 top14_to_bcd({2'b0, MULT_RES[23:10]}, BCD0, BCD1, BCD2, BC
34
35      hex_to_7seg SEG0(HEX0, BCD0);
36      hex_to_7seg SEG1(HEX1, BCD1);
37      hex_to_7seg SEG2(HEX2, BCD2);
38      hex_to_7seg SEG3(HEX3, BCD3);
39      hex_to_7seg SEG4(HEX4, BCD4);
40
41      always @ (posedge TICK_10KHz)
42         COUNT <= COUNT + SW;
43
44      endmodule
45
```

Experiment 15 (Optional Challenge): Using the A-t

Sunday, December 8, 2019 1:56 PM

Key outcome of experiment 15:

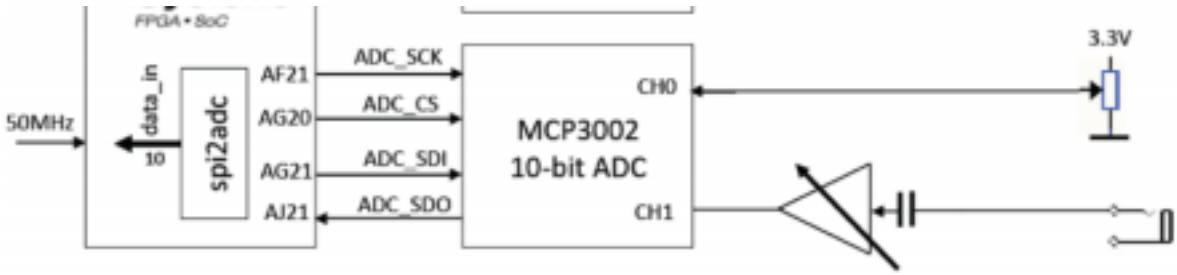
- Use the Analogue-to-Digital converter MCP3002 to convert dc voltages

At the beginning of part 3, we set out to do many things, such as understanding the SPI, u processes, using a ROM, and using an ADC to convert DC voltages.

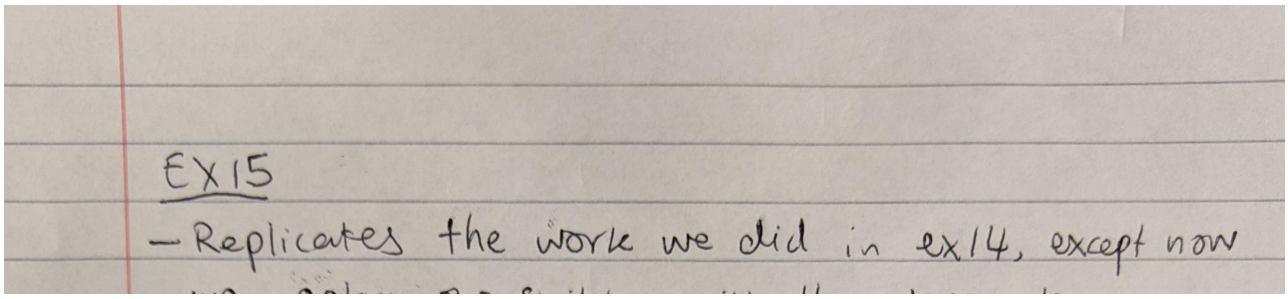
We achieved all these goals except for the last. We felt this was a necessary challenge for understood the relation between both directional flows of digital and analogue data.

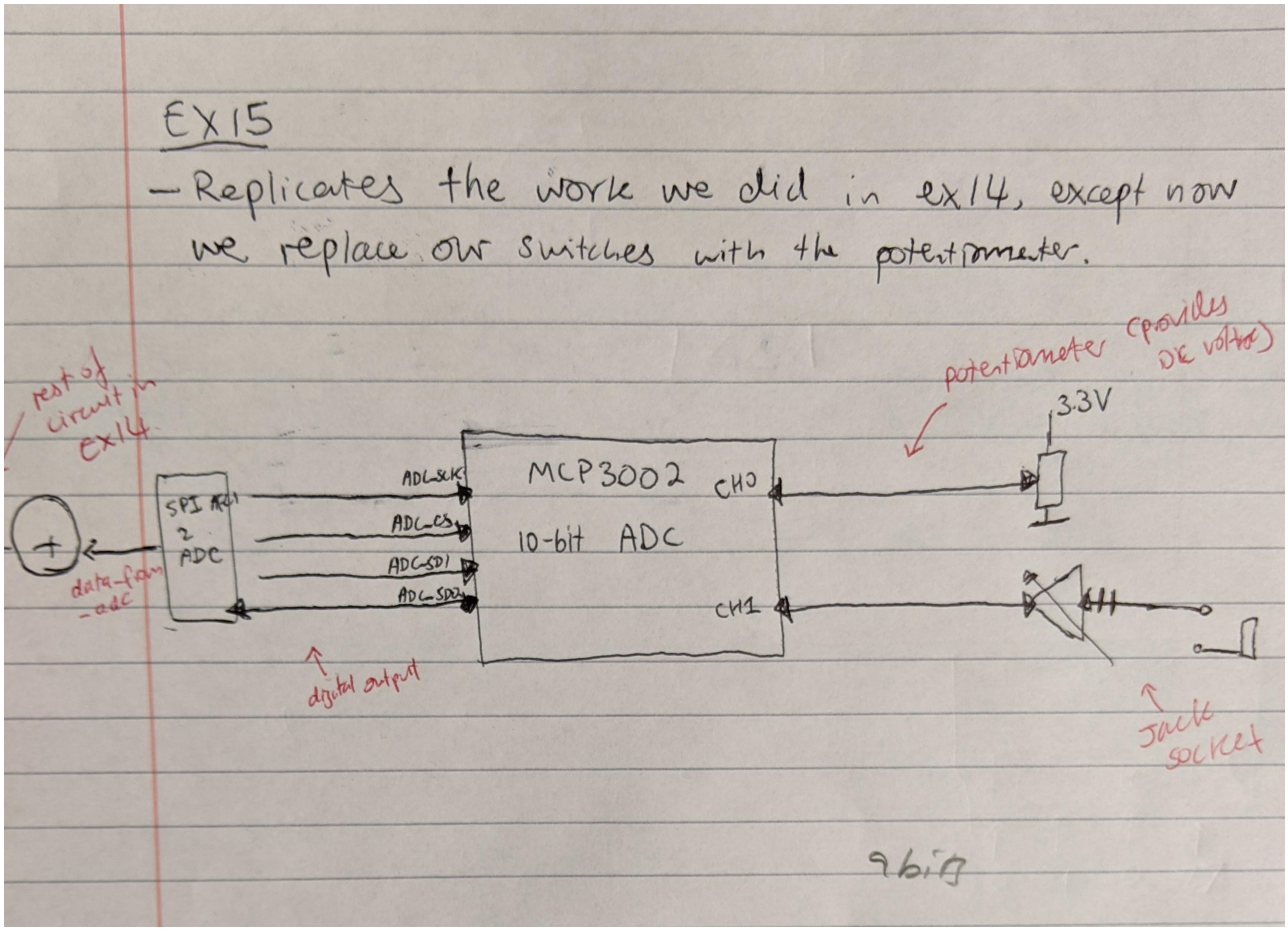
The task was clearcut- "instead of using the slide switches to control the frequency, use th the DC voltage of the potentiometer (which is between 0v and 3.3v) and use this converte

We weren't sure about the potentiometer component, so referred to our board schemati



Once we familiarized ourselves with the voltage source/potentiometer as shown in the di realized that this experiment was a clone copy of experiment 14, except with our analogu potentiometer instead of the switches we were familiar with.





To work with this analogue source, we would have to include the spi2adc to turn these into bits we can work with in our circuit. The adder is the "plug" to what we completed in experiment 14.

We implemented this as follows:

```
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 div_5k(CLOCK_50, TICK_10KHz, 16'd5000, 1'b1);

    ROM rom_data(COUNT, TICK_10KHz, DATA);

    spi2dac analogue_out0(CLOCK_50, DATA, TICK_10KHz, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);
```

```
33
34      spi2adc analogue_in(
35         .sysclk (CLOCK_50),
36         .channel (1'b0),                                  //set Channel 0 for the potentiomete
37         .start (TICK_10KHz),
38         .data_from_adc (a2d_data),
39         .data_valid (data_valid),                         //not used
40         .sdata_to_adc (ADC_SDI),
41         .adc_cs (ADC_CS),
42         .adc_sck (ADC_SCK),
43         .sdata_from_adc (ADC_SDO));
44
45      pwm analogue_out1(CLOCK_50, DATA, TICK_10KHz, PWM_OUT);
46
47      mult_24 times_10k(DATA,MULT_RES);
48
49      bin2bcd_16 top14_to_bcd({2'b0, MULT_RES[23:10]}, BCD0, BCD1, BCD2, BCD3, BCD4);
50
51      hex_to_7seg SEG0(HEX0, BCD0);
```

From the code, we note the implementation of spi2adc. We are passing signals tick_10k as the load input, and have selected channel 0 for the SPI-ADC input (so that we may read from our potentiometer.

A2data is our address offset to be added to our counter. This is done at the very end of our top function, as follows:

```
57      always @ (posedge TICK_10KHz)
58         COUNT <= COUNT + a2d_data;
59
```

Succesfully extending Exp.14 for analogue input.

SPI2ADC

```
 7   //------------------------------
 8
 9   module spi2adc (sysclk, start, channel, data_from_adc, data_valid,
10                   sdata_to_adc, adc_cs, adc_sck, sdata_from_adc);
11
12      input sysclk;         // 50MHz system clock of DE0
13      input start;          // Pulse to start ADC, minimum wide = clock period
14      input channel;        // channel 0 or 1 to be converted
15      output [9:0] data_from_adc;   // 10-bit ADC result
16      output data_valid;    // High indicates that converted data valid
17      output sdata_to_adc; // Serial commands send to adc chip
18      output adc_cs;        // chip select - low when converting
19      output adc_sck;       // SPI clock - active during conversion
20      input sdata_from_adc;   // Converted serial data from ADC, MSB first
21
22   //-------------Input Ports-----------------------------
23   // All the input ports should be wires
24      wire  sysclk, start, sdata_from_adc;
25
26   //-------------Output Ports----------------------------
27   // Output port can be a storage element (reg) or a wire
28      reg [9:0]   data_from_adc;
29      reg         adc_cs;
30      wire        sdata_to_adc, adc_sck, data_valid;
31
32   //-------------Configuration parameters for ADC --------
33      parameter   SGL=1'b1;   // 0:diff i/p, 1:single-ended
34      parameter   MSBF=1'b1;  // 0:LSB first, 1:MSB first
35
36   // --- Submodule: Generate internal clock at 1 MHz -----
37      reg         clk_1MHz;   // 1Mhz clock derived from 50MHz
38      reg [4:0]   ctr;        // internal counter
39      reg         tick;       // 1MHz clock tick lasting 20ns, i.e. one 50MHz cycle
40      parameter   TIME_CONSTANT = 5'd24;  // change this for diff clk freq
41      initial begin
42         clk_1MHz = 0;        // don't need to reset - don't care if it is 1 or 0 to start
43         ctr = 5'b0;          //  ... to start.  Initialise to make simulation easier
44         tick = 1'b0;
45      end
46
47      always @ (posedge sysclk)   //
48        if (ctr==0) begin
49           ctr <= TIME_CONSTANT;
50           if (clk_1MHz==1'b0)
51              tick <= 1'b1;
52           clk_1MHz <= ~clk_1MHz; // toggle the output clock for squarewave
53        end
54      else  begin
55         ctr <= ctr - 1'b1:
```

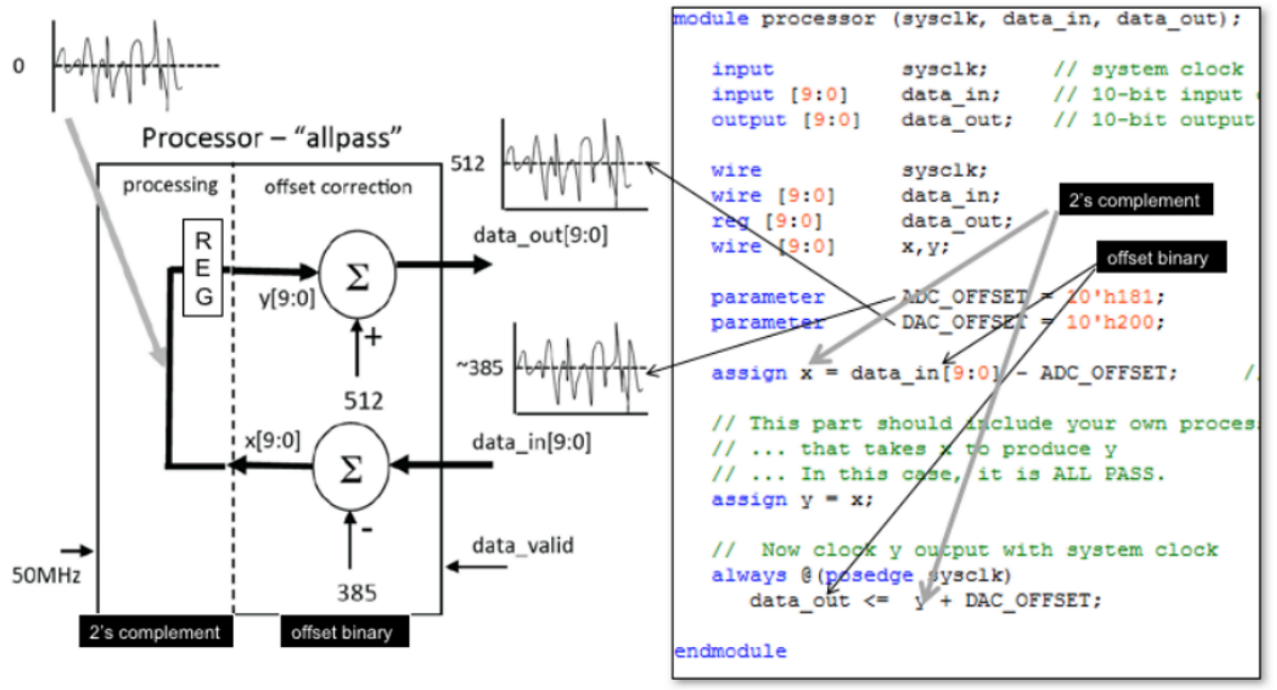
```
56          tick <= 1'b0;
57        end
58    // ---- end internal clock generator ----------
59
60    // ---- Detect start is asserted with a small state machine
```

```
58    // ---- end internal clock generator ----------
59
60    // ---- Detect start is asserted with a small state machine
61       // .... FF set on positive edge of start
62       // .... reset when adc_cs goes high again
63       reg [1:0]    sr_state;
64       parameter    IDLE  = 2'b00,WAIT_CSB_FALL = 2'b01, WAIT_CSB_HIGH = 2'b10;
65       reg          adc_start;
66
67       initial begin
68          sr_state = IDLE;
69          adc_start = 1'b0; // set while sending data to ADC
70          end
71
72       always @ (posedge sysclk)
73          case (sr_state)
74             IDLE: if (start==1'b0) sr_state <= IDLE;
75                   else  begin
76                      sr_state <= WAIT_CSB_FALL;
77                      adc_start <= 1'b1;
78                      end
79             WAIT_CSB_FALL: if (adc_cs==1'b1) sr_state <= WAIT_CSB_FALL;
80                   else sr_state <= WAIT_CSB_HIGH;
81
82             WAIT_CSB_HIGH: if (adc_cs==1'b0) sr_state <= WAIT_CSB_HIGH;
83                   else begin
84                      sr_state <= IDLE;
85                      adc_start <= 1'b0;
86                      end
87             default: sr_state <= IDLE;
88          endcase
89    //------- End circuit to detect start and end of conversion
90
91
92    // spi controller designed as a state machine
93    // .... with 16 states (idle, and S1-S15 indicated by state value
94
95       reg [4:0]   state;
96       reg   adc_done, adc_din, shift_ena;
97
98       initial  begin
99          state = 5'b0; adc_cs = 1'b1; adc_done = 1'b0;
100         adc_din = 1'b0; shift_ena <= 1'b0;
101         end
102
103      always @(posedge sysclk)
104         if (tick==1'b1)  begin
105
106      // default outputs and state transition
107         adc_cs <= 1'b0;  adc_done <= 1'b0;  adc_din <= 1'b0; shift_ena <= 1'b0;
108         state <= state + 1'b1;
109         case (state)
110            5'd0: begin
111                     if (adc_start==1'b0) begin
104      if (tick==1'b1)  begin
105
106      // default outputs and state transition
107         adc_cs <= 1'b0;  adc_done <= 1'b0;  adc_din <= 1'b0; shift_ena <= 1'b0;
108         state <= state + 1'b1;
109         case (state)
110            5'd0: begin
111                     if (adc_start==1'b0) begin
112                        state <= 5'd0;       // still idle
113                        adc_cs <= 1'b1;      // chip select not active
114                        end
115                     else begin
116                        state <= 5'd1;       // start converting
117                        adc_din <= 1'b1;     // start bit is 1
118                        end
119                  end
120            5'd1: adc_din <= SGL;            // SGL bit
121            5'd2: adc_din <= channel;        // CH bit
122            5'd3: adc_din <= MSBF;           // MSB first bit
123            5'd4: shift_ena <= 1'b1;         // start shifting data from adc
124            5'd15: begin
125                     shift_ena <= 1'b0;
126                     adc_done <= 1'b1;
127                    end
128            5'd16: begin
129                     adc_cs <= 1'b1;      // last state - disable chip select
130                     state <= 5'd0;       // go back to idle state
131                   end
132            default:
133                     shift_ena <= 1'b1;   // unspecified states are covered by default above
134            endcase
135         end   // ... always
136
137      // shift register for output data
138      reg [9:0] shift_reg;
139      initial  begin
140            shift_reg = 10'b0;
141            data_from_adc = 10'b0;
142            end
143
144      always @(negedge sysclk)
145         if((adc_cs==1'b0)&&(shift_ena==1'b1)&&(tick==1'b1))      // start shifting data_in
146               shift_reg <= {shift_reg[8:0],sdata_from_adc};
147
148      // Latch converted output data
149      always @(posedge sysclk)
150         if(adc_done)
151               data_from_adc = shift_reg;
```

Experiment 16: An audio in-and-out (all pass) loop

Sunday, December 8, 2019 1:56 PM

In Experiment 16, we were tasked with understanding the ALLPASS processing module. The code for which is analyzed below:



The input data_in[9:0] is used to represent the analogue signal input (which is bipolar) as offset binary. There is an offset of ~385 if the input is connected to 0 (no signal) - the output data_out[9:0] also has an offset. To get Vout = 0V, you need to send in the binary number 512.

To process the signal using normal arithmetic operations such as + or * we need to use the 2s compliment number system so that the ADC data is offset correctly by subtracting the 385 from the converted data to yield x[9:0]. Then the output y[9:0] is again converted back to offset binary for the DAC to output it. This is done by adding 512 to y[9:0].

We were also tasked with creating a mult4 module which amplifies the input signal by 4 before outputting it. This was achievable by performing an arithmetic shift left by 2 bits.

```
module mult4(
   data_in,
```

```
   data_out)

   input [9:0] in;
   output [9:0] out;

   assign data_out[9:0] = {data_in [7:0], 2'b0}; //logical shift by 2 (i.e. multiply by 4)

endmodule


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
   spi2dac SPI_DAC (CLOCK_50, data_out, tick_10k,      // send processed sample t
                    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-c

   spi2adc SPI_ADC (                                       // perform a A-to-D conver
      .sysclk (CLOCK_50),                                  // order of parameters do
      .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 t

   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 compleme

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

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

endmodule
```