Overview

30 September 2019 11:32

Course Homepage
https://intranet.ee.ic.ac.uk/t.clarke/ee2lab/
https://intranet.ee.ic.ac.uk/intranet/labweb/
http://www.ee.ic.ac.uk/pcheung/teaching/E2_Experiment/

Timetable

Screen clipping taken: 30/09/2019 11:35

Aims:
The laboratory course aims to introduce you to best practice in experimental work, while supporting the lecture courses and giving practical exposure to real systems.

Learning Outcomes:
You will learn to use test equipment with accuracy and confidence, while deepening your understanding of the taught material by gaining practical insight and skills.

Syllabus:
Hardware experiments in Communications, Control, uP A/D & D/A. Software experiments in Signal processing & Communications (MATLAB).

From <http://intranet.ee.ic.ac.uk/electricalengineering/eecourses_t4/course_content.asp?c=EE2-ILABE&s=I2#start>

Veri lab

19 November 2019 11:30

Declare all inputs/outputs in module, name the module the same as file name

Declare all the things in module as input/output, any variables declare as reg, and things you want constant declare as parameters. Parameters allow you to generalise your module blocks to be reused with a different specification, defparam can be used to override the parameter initialisation.

If enable is set, then count increments on the rising edge of clock. You do not need to reset count, as it will roll over to 0 when it increments past its bit width.

Changed BIT_SZ to 16 so that the counter now counts up to 16 not 8.

In the always block, added 'if reset == 0' to provide resetting functionality. This is synchronous, if it was in the sensitivity list as well as the clock it would be asynchronous.

In the top level function we edit the previous code by making the binary BCD by calling the new module bin2bcd_10

The 7 seg modules are called like before

We change the hex to 7seg file so that it only goes 0-9

Only the very first inputs, and final outputs are in the module list. All intermediary inputs/outputs of modules are labelled as wire.

Each module is called and the wires connect them together

KEY[0] is reset, and is inverted (using ~) before entering the counter module to meet the specifications required.

This is 4bit LFSR so repeats every 2^n-1 = 15 cycles

This is the code for the X^7 LFSR

XOR gate makes a random sequence (repeating every 2^7 - 1 cycles)

{} mean concatenation (what does this mean?)

To make the LFSR from X^4 to 1+ X+ X^7 we change what is XOR to the first and last registers, and then change the number of registers to 7

Output = high, Count until data in, at which point set output = low this produces a signal with high's based on b

Declares the I/O

Defines the I/O

Calls the pwm module

DAC_SCK

DAC_SDI

TP8

TP5

The signals are the same, so that shows that it doesn't matter whether we use pwm or the DAC, it gives the same result.

TP9

TP5

To attenuate by 0.5, it is dividing by 2, so shift right by 1.

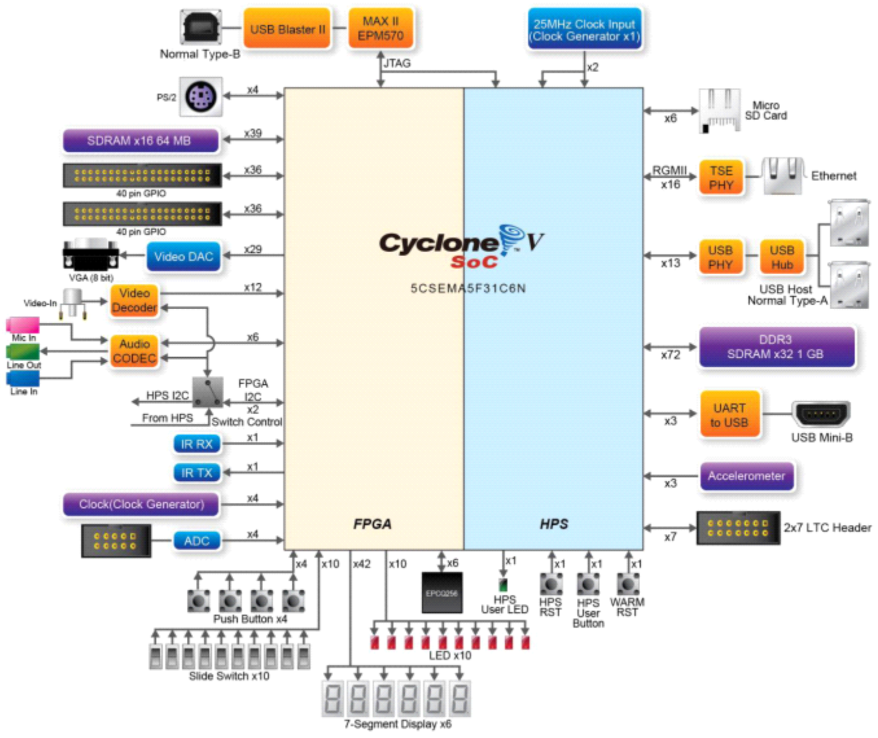
FIFO can be used to implement constant delay

Part 1

06 December 2019 20:09

Part 1

Block diagram of the DE1-SOC board:



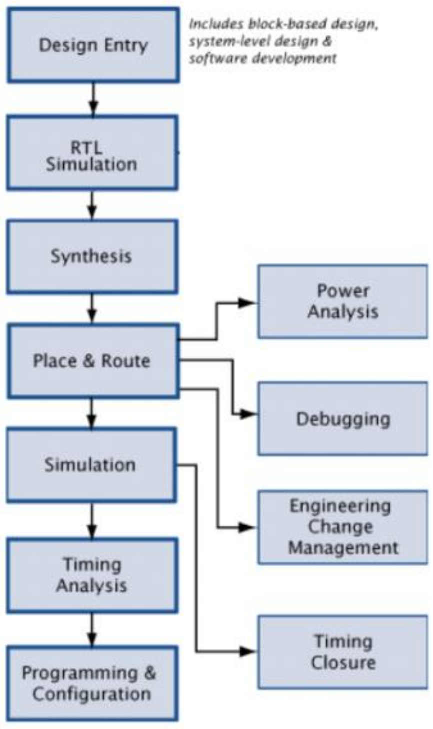
Ex1: Boolean equation for out[4] = /in3*in0 + /in3*in2*/in1 + /in2*/in1*in0

Using the graphic editor:

Clicking on the gate icon or workspace allows you to place gates, click on the gates legs and move the cursor to add wires.
Once schematic for a block section is finished create a symbol for it in create/update.
Create a top level file to indicate how the I/O of the chip should interact with your design.

To program FPGA:

1. Analysis and Elaboration
2. Compilation
3. Program the FPGA
    a. tools -> programmer
    b. Hardware setup -> DE-SOC
    c. Auto detect -> 5SC… -> delete SOC -> add files -> output files -> select file.


Design Entry
Includes block-based design, system-level design & software development
RTL Simulation
Synthesis
Place & Route
Simulation
Timing Analysis
Programming & Configuration
Power Analysis
Debugging
Engineering Change Management
Timing Closure

Screen clipping taken: 04/12/2019 15:16

Analysis & Synthesis optimises the design considering the specific logic. It checks for logical completeness and checks for syntax errors (spelling, incorrect port names etc). It optimises the design for the specific device used. It uses algorithms to minimise gate count and redundant logic, as well as use the device architecture as efficiently as possible. It creates state assignments for state machines and reduces resources used. It then maps combinational resources to individual cell-sized logic units. Builds a database that integrates all the design files in the design. Creates a register level model of design for RTL simulation.

Analyse current file: To check current file for syntax and semantics errors.

Analysis and elaboration: Partially compiles the current design.

Pin assignments can be done inside the pin planner, or in the .qsf file.

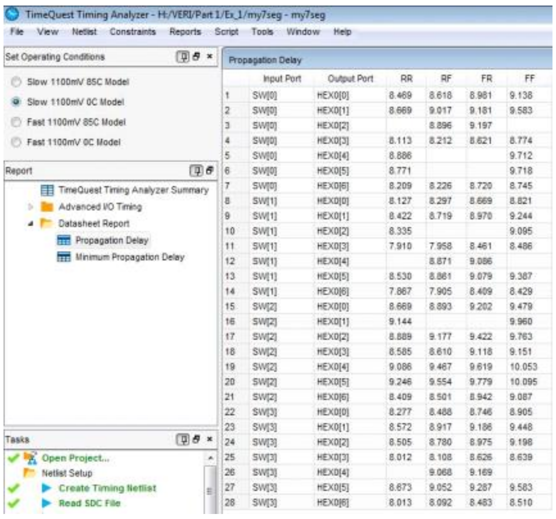
Set_instance_assignment -name //SomeName// "3.3-V LVTTL" -to HEX0[4]
Defines the voltage
Set_location_assignment //PIN NAME// -to HEX0[4]
Defines the pin location of HEX0[4] to pin name.
In order to compile we need to assign pins so that Quartus knows which signal to connect to which pin on FPGA.

Top level specification:

Can be the .bdf file or .v Verilog file.

Compilation generates a .sof file. It uses the Analysis & synthesis, Fitter, Assembler, Design Assistant and Timing Analyser to create a programming file (a file that programs the board).

The first experiment involved programming a 7 segment display decoder using schematic capture on Quartus. The following are timing analyser reports at different temperatures:

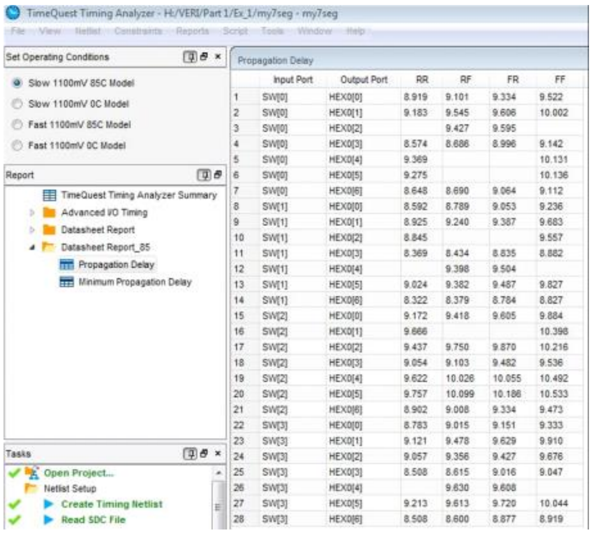

TimeQuest Timing Analyzer - H:/VERI/Part 1/Ex_1/my7seg - my7seg
File View Netlist Constraints Reports Script Tools Window Help
Report
TimeQuest Timing Analyzer Summary
Advanced I/O Timing
Datasheet Report
Propagation Delay
Minimum Propagation Delay
Tasks
Open Project...
Netlist Setup
Create Timing Netlist
Read SDC File

Set Operating Conditions

Slow 1100mV 85C Model
[✓ Slow 1100mV 0C Model]
Fast 1100mV 85C Model
Fast 1100mV 0C Model

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


TimeQuest Timing Analyzer - H:/VERI/Part 1/Ex_1/my7seg - my7seg
File View Netlist Constraints Reports Script Tools Window Help
Report
TimeQuest Timing Analyzer Summary
Advanced I/O Timing
Datasheet Report
Datasheet Report_85
Propagation Delay
Minimum Propagation Delay
Tasks
Open Project...
Netlist Setup
Create Timing Netlist
Read SDC File

Set Operating Conditions

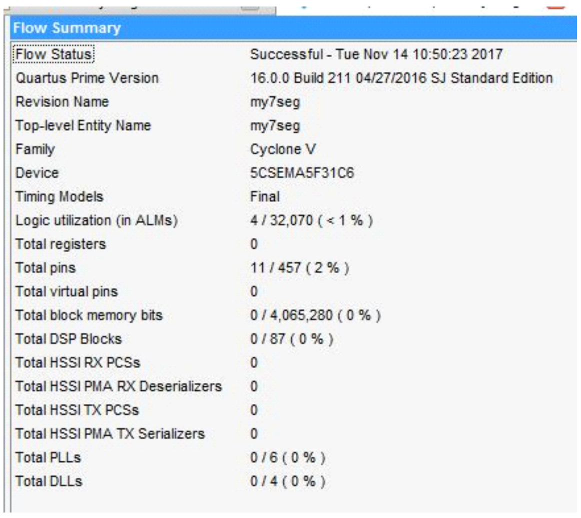
[✓ Slow 1100mV 85C Model]
Slow 1100mV 0C Model
Fast 1100mV 85C Model
Fast 1100mV 0C Model

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

Propagation Delay: Reports longest delay in nanoseconds between the edges of a signal propagating from an input port to an output port. a. RR shows the longest delay measured from rising edge to rising edge b. RF shows the longest delay measured from rising edge to falling edge c. FR shows the longest delay measured from falling edge to rising edge d. FF shows the longest delay measured from falling edge to falling edge Note that, in the propagation delay the highest delay (Worst-Case) is important

The Rise and fall times seem to be generally longer at 85 degrees than 0 degrees because of more power wastage at higher temperatures.



Flow Summary

Flow Status	Successful - Tue Nov 14 10:50:23 2017
Quartus Prime Version	16.0.0 Build 211 04/27/2016 SJ Standard Edition
Revision Name	my7seg
Top-level Entity Name	my7seg
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 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 %)

Here we can see only 11 pins and 4 ALUs are used due to optimisation that the software provides.

Exercise 2

In this experiment, we Verilog hardware description language to implement the 7 segment decoder.
Hex to 7 seg module

```
module hex_to_7seg (out,in);

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

        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
```

Always @ (*) means any input variable used inside this block will be in the sensitivity list (i.e. this block will execute if any of them change).
Reg is used to declare a variable that holds a value, and can be changed anytime in a simulation.

Top level module

```
module ex2_top (
                SW,
                HEX0
);

        input[3:0] SW;
        output[6:0] HEX0;

        hex_to_7seg     SEG0 (HEX0,SW);

endmodule
```

Using Verilog HDL was much less tedious and less prone to errors than using schematic capture.

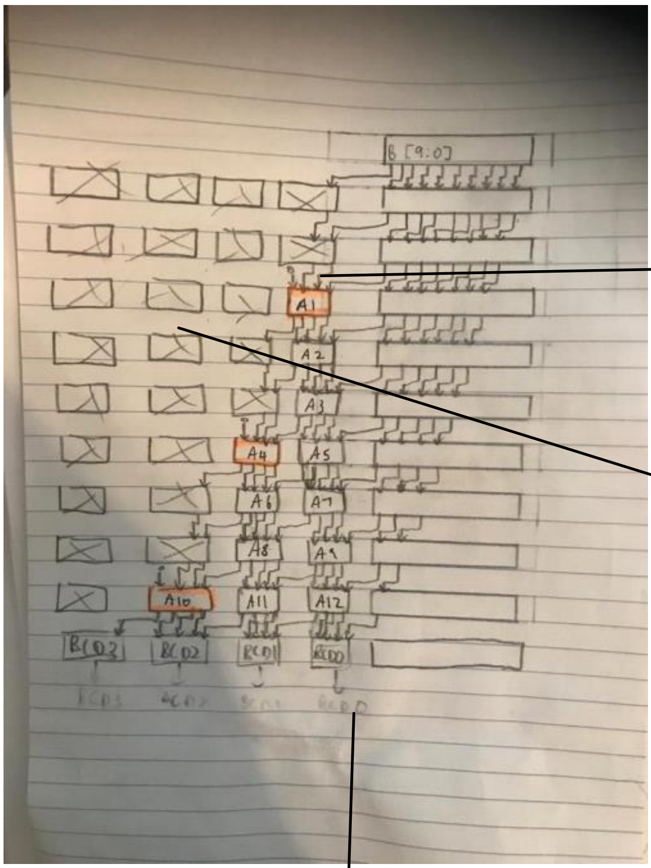
Exercise 3

Top level module

```
module ex3_top(SW,HEX0,HEX1,HEX2);

        input [9:0] SW; //INPUT SWITCHES
        output [6:0] HEX0,HEX1,HEX2;

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

endmodule
```

Here the {2'b0, SW[9:8]} means concatenation, so the result of this is number: 00XY
Where X = SW[9] and Y = SW[8]

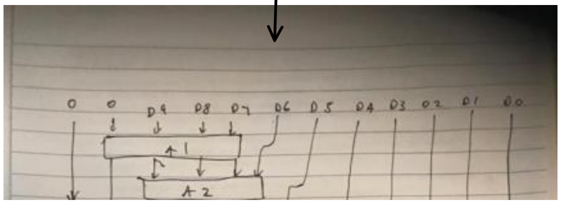
Optional - Exercise 4

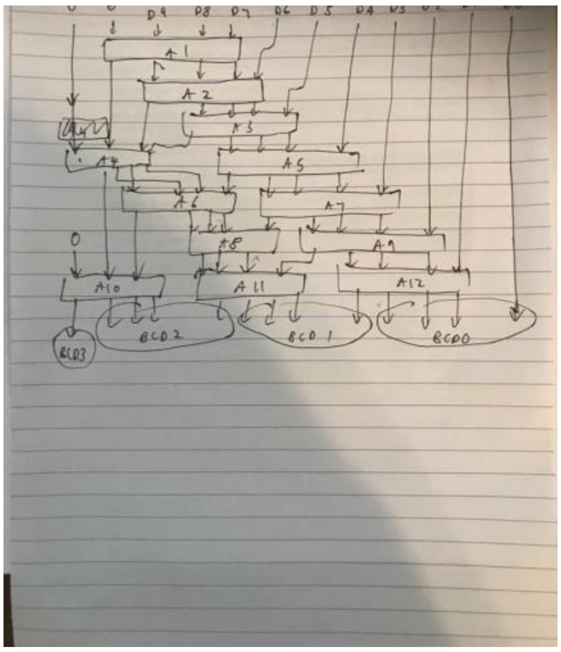


This exercise involved extending the theory for a 8 bit binary to BCD convertor into 10 bit. We followed the shift and add algorithm to come up the following mappings which we then coded as shown below.

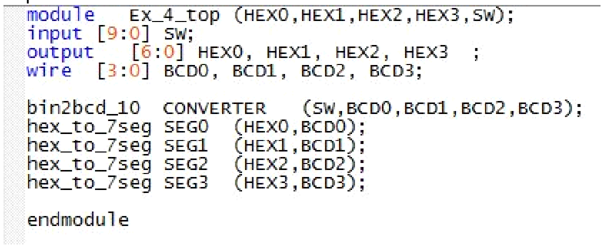
At each orange box, a new 'wire' is created as a new bit would be required if the value inside adjust box is greater than 4.

We only need to code adjust boxes where the value could be greater than 4- ignore the rest.



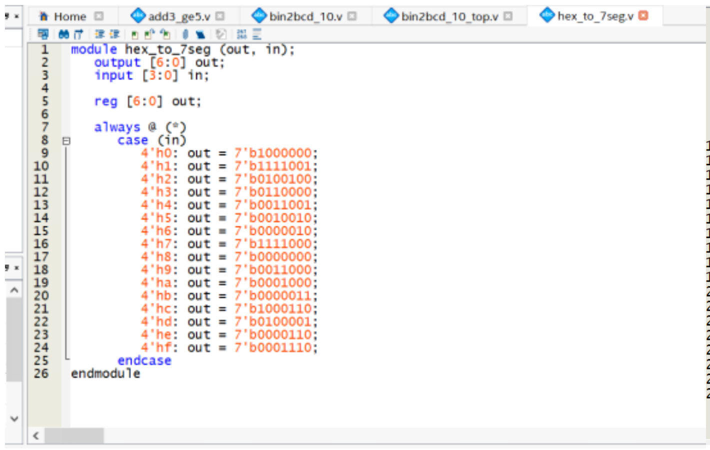


Top level module

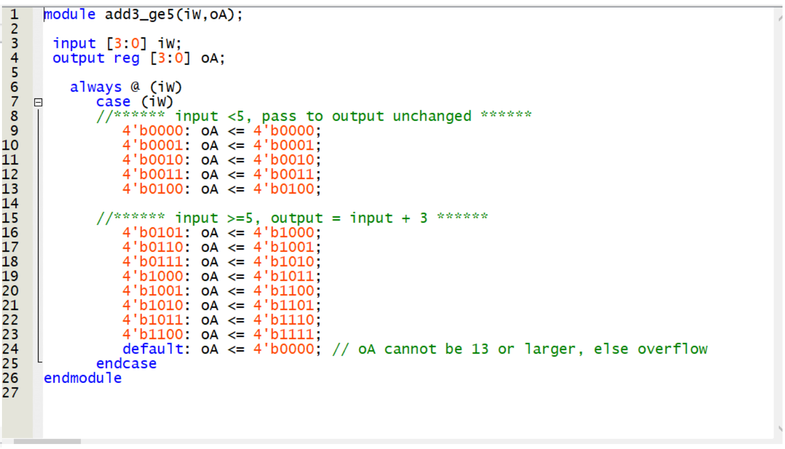


```
module   Ex_4_top (HEX0,HEX1,HEX2,HEX3,SW);
input [9:0] SW;
output   [6:0] HEX0, HEX1, HEX2, HEX3  ;
wire  [3:0] BCD0, BCD1, BCD2, BCD3;

bin2bcd_10  CONVERTER   (SW,BCD0,BCD1,BCD2,BCD3);
hex_to_7seg SEG0  (HEX0,BCD0);
hex_to_7seg SEG1  (HEX1,BCD1);
hex_to_7seg SEG2  (HEX2,BCD2);
hex_to_7seg SEG3  (HEX3,BCD3);

endmodule
```

```
module bin2bcd_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;
   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 = a10[3];

endmodule
```



```
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
```



```
module add3_ge5(iW,oA);

 input [3:0] iW;
 output reg [3:0] oA;

   always @ (iW)
      case (iW)
      //****** input <5, pass to output unchanged ******
         4'b0000: oA <= 4'b0000;
         4'b0001: oA <= 4'b0001;
         4'b0010: oA <= 4'b0010;
         4'b0011: oA <= 4'b0011;
         4'b0100: oA <= 4'b0100;

      //****** input >=5, output = input + 3 ******
         4'b0101: oA <= 4'b1000;
         4'b0110: oA <= 4'b1001;
         4'b0111: oA <= 4'b1010;
         4'b1000: oA <= 4'b1011;
         4'b1001: oA <= 4'b1100;
         4'b1010: oA <= 4'b1101;
         4'b1011: oA <= 4'b1110;
         4'b1100: oA <= 4'b1111;
         default: oA <= 4'b0000; // oA cannot be 13 or larger, else overflow
      endcase
endmodule
```

This module implements the 10 bit binary to bcd conversion.

It assigns the values to in, and calls the add3shift module.

Then it assigns the BCD outputs.

Part 2

06 December 2019 20:09

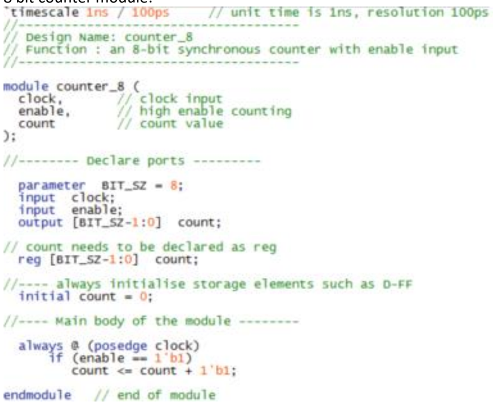
Outcomes:
- Using Modelsim to verify correct function of your design and testbenches
- How to design different types of counters and timers
- Predict max operating clock frequency of circuit

Experiment 5

`timescale time_unit / time_precision : time_unit and precision take in arguments magnitude(1, 10 or 100) and unit time (s,ms, us, ns, ps, fs). Time precision must be <= time unit value. Time unit is the measurement of delays and simulation time, time precision is how delay values are rounded before being used. When a delay isgiven (e.g. #1) then it is multiplied with time_unit, and rounded by time_precision.

To set as top level entity = Project -> add as top level entity

8 bit counter module:

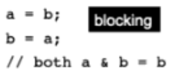


```
`timescale 1ns / 100ps     // unit time is 1ns, resolution 100ps
//-------------------------------------
// Design Name: counter_8
// Function : an 8-bit synchronous counter with enable input
//-------------------------------------

module counter_8 (
  clock,       // clock input
  enable,      // high enable counting
  count        // count value
);

//-------- Declare ports ---------

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

// count needs to be declared as reg
  reg [BIT_SZ-1:0]  count;

//---- always initialise storage elements such as D-FF
  initial count = 0;

//---- Main body of the module --------

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

endmodule   // end of module
```

Screen clipping taken: 04/12/2019 16:23

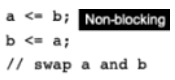
This counts up to the parameter BIT_SZ which can be adjusted for other sizes.

<u>Blocking vs Non-blocking:</u>



```
a = b;  blocking
b = a;
// both a & b = b
```

Screen clipping taken: 09/12/2019 16:11



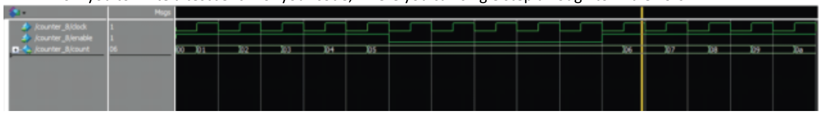
```
a <= b;  Non-blocking
b <= a;
// swap a and b
```

Screen clipping taken: 09/12/2019 16:11

For non-blocking the statements are executed in parallel, for blocking they are executed sequentially.

Modelsim
You must tell Modelsim what to simulate by doing Simulate -> Start simulation, then work -> myFile.

Do files:
Allow you to write a testbench for your code, where you can single step through to find errors.



Output from Modelsim

Question answers:
- Predicted max frequencies for this circuit: 85C - 445.04MHz, 0C - 422.48MHz
- Input / Output ports and paths are unconstrained - the outputs/inputs to the circuit are not timing constrained - the inputs may change without any clock as they are not held (it cannot make a full time analysis as it doesn't know if the inputs may change).

Experiment 6:

Module counter_16

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

//Design Name: counter_16
//Function: a 16-bit synchronous counter with enable input

module counter_16 (
        clock,  //clock input
        enable, //high enable counting
        reset,
        count
);

//Declaring the ports:

        parameter BIT_SZ = 16;
        input clock;
        input enable;
        input reset;
        output [BIT_SZ-1:0] count;

// count needs to be declared as reg which can save values assigned to it

        reg [BIT_SZ-1:0] count;

//always initialise storage elements such as D-FF
        initial count = 0;

//MAIN BODY OF MODULE:

        always @ (posedge clock)
        begin
                if(enable == 1'b1)
                        count <= count + 1'b1;
                if(reset == 1'b1)
                        count <= 1'b0;
        end
endmodule
```

To convert the 8 bit counter to 16 bits, simply change the parameter BIT_SZ. To reset synchronously to the clock we add if(reset) inside the always block. If we wanted it to reset asynchronously we could put it in the sensitivity list.

Top level module:

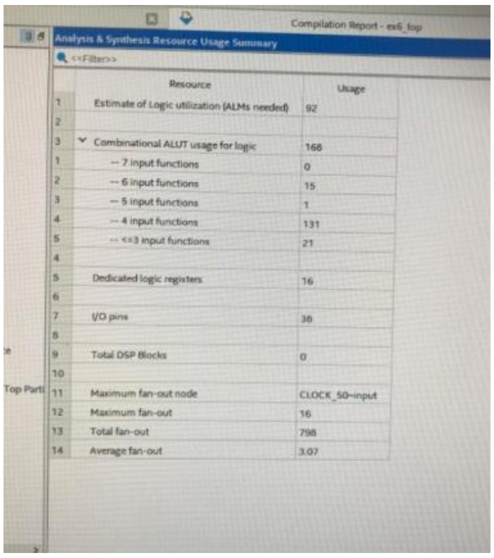
```
module ex6_top(KEY,CLOCK_50,HEX0,HEX1,HEX2,HEX3,HEX4);

        input [1:0] KEY;
        wire [3:0] BCD0,BCD1,BCD2,BCD3,BCD4;
        input CLOCK_50;
        output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4;
        wire [15:0] counter;


                counter_16 temp0(CLOCK_50,~KEY[0],~KEY[1],counter);

                bin2bcd_16 temp1(counter,BCD0,BCD1,BCD2,BCD3,BCD4);

                hex_to_7seg temp2(HEX0,BCD0);
                hex_to_7seg temp3(HEX1,BCD1);
                hex_to_7seg temp4(HEX2,BCD2);
                hex_to_7seg temp5(HEX3,BCD3);
                hex_to_7seg temp6(HEX4,BCD4);

endmodule
```

For the top level we just needed to call the counter, call the bin2bcd and call the hex to 7 seg modules. The keys were inverted to meet the specification using the bitwise inverter (~). It was more generic to invert them here than change the module.

To create a clock frequency:

- Make myName_top.sdc file
- Create_clock -name "myName" -period xs [get_ports {myName}]

Timing analysis results:



Analysis & Synthesis Resource Usage Summary

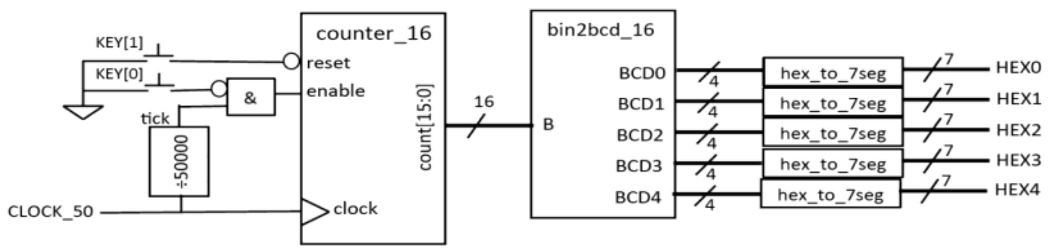
	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	92
2		
3	Combinational ALUT usage for logic	168
1	-- 7 input functions	0
2	-- 6 input functions	15
3	-- 5 input functions	1
4	-- 4 input functions	131
5	-- <=3 input functions	21
4		
5	Dedicated logic registers	16
6		
7	I/O pins	38
8		
9	Total DSP Blocks	0
10		
11	Maximum fan-out node	CLOCK_50~input
12	Maximum fan-out	16
13	Total fan-out	798
14	Average fan-out	3.07

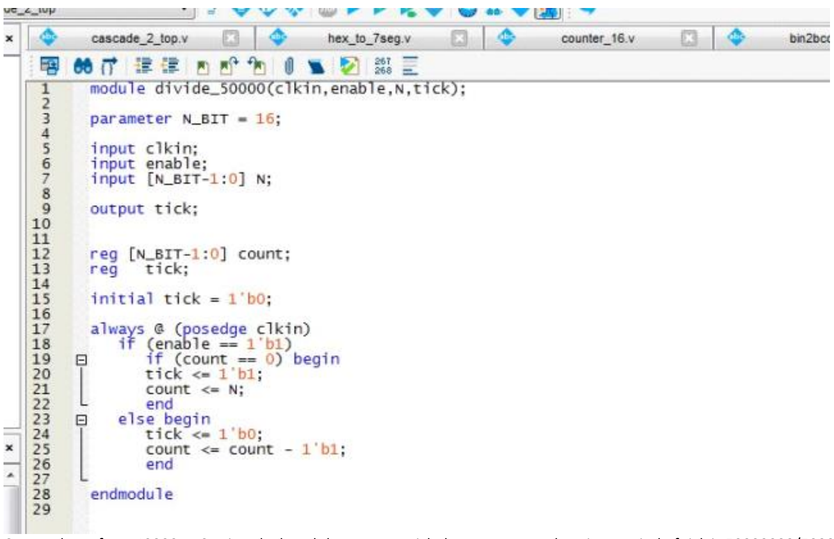
Screen clipping taken: 29/11/2019 10:22

Logic registers - 16 cause its 16 bit counter

Cascade Counter

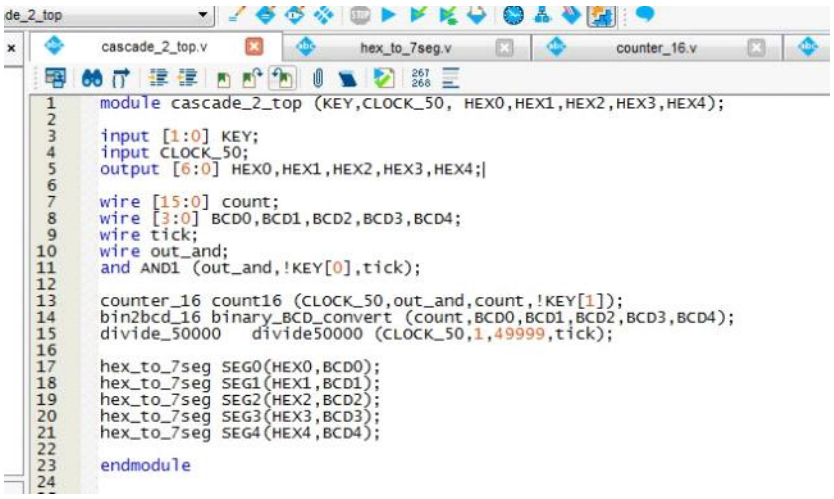
CASCADE COUNTER





```
module divide_50000(clkin,enable,N,tick);

parameter N_BIT = 16;

input clkin;
input enable;
input [N_BIT-1:0] N;

output tick;


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

initial tick = 1'b0;

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

endmodule
```

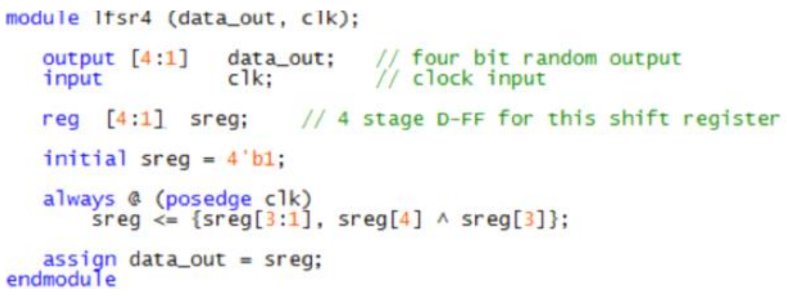
Counts down from 49999 to 0 using clock and then creates tick then repeats so that time period of tick is 50000000/50000=1ms.



```
module cascade_2_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 [3:0] BCD0,BCD1,BCD2,BCD3,BCD4;
wire tick;
wire out_and;
and AND1 (out_and,!KEY[0],tick);

counter_16 count16 (CLOCK_50,out_and,count,!KEY[1]);
bin2bcd_16 binary_BCD_convert (count,BCD0,BCD1,BCD2,BCD3,BCD4);
divide_50000   divide50000 (CLOCK_50,1,49999,tick);

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
```

Experiment 7:

Original code from lecture



```
module lfsr4 (data_out, clk);

   output [4:1]   data_out;   // four bit random output
   input          clk;        // clock input

   reg  [4:1]  sreg;    // 4 stage D-FF for this shift register

   initial sreg = 4'b1;

   always @ (posedge clk)
       sreg <= {sreg[3:1], sreg[4] ^ sreg[3]};

   assign data_out = sreg;
endmodule
```

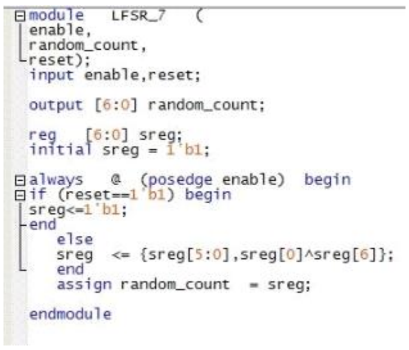
Screen clipping taken: 04/12/2019 17:11

Manually working out the first 10 sequence values

Q6	Q5	Q4	Q3	Q2	Q1	Q0	Count
0	0	0	0	0	0	1	1
0	0	0	0	0	1	1	3
0	0	0	0	1	1	1	7
0	0	0	1	1	1	1	15
0	0	1	1	1	1	1	31
0	1	1	1	1	1	1	63
1	1	1	1	1	1	1	127
1	1	1	1	1	1	0	126
1	1	1	1	1	0	1	125
1	1	1	1	0	1	0	122

XOR Q0 and Q6

LFSR modified to work for x^7

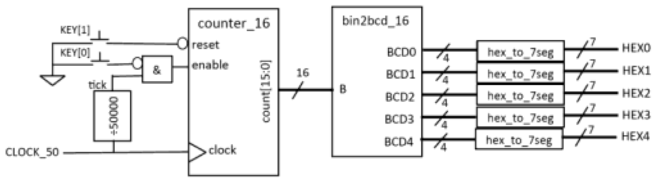


```
module   LFSR_7   (
enable,
random_count,
reset);
input enable,reset;

output [6:0] random_count;

reg    [6:0] sreg;
initial sreg = 1'b1;

always   @  (posedge enable)  begin
if (reset==1'b1) begin
sreg<=1'b1;
end
    else
    sreg   <= {sreg[5:0],sreg[0]^sreg[6]};
    end
    assign random_count  = sreg;

endmodule
```

Here we simply changed which registers to concatenate (we want 1+X+X^7 so we need 0, and 6 in this system as we are starting counting from 0 not 1). To add reset functionality simply add if(reset== 1) inside the always block. This resetting is synchronous. Enable is set to ~KEY[3] in the top level function, as per specification.

Test yourself task - Cascade counter

Specification diagram:



From this we create the top level module:
```
module casc_count_top(KEY,CLOCK_50,HEX0,HEX1,HEX2,HEX3,HEX4);


        input [1:0] KEY;
        wire [3:0] BCD0,BCD1,BCD2,BCD3,BCD4;
        input CLOCK_50;
        output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4;
        wire [15:0] counter;
        wire clock_out;
        reg result;

        clkdiv temp7(CLOCK_50,clock_out);

        always @ (*)
           begin
               result <= ~KEY[0] & clock_out;
      end

        counter_16 temp0(CLOCK_50,result,~KEY[1],counter);
        bin2bcd_16 temp1(counter,BCD0,BCD1,BCD2,BCD3,BCD4);

        hex_to_7seg temp2(HEX0,BCD0);
        hex_to_7seg temp3(HEX1,BCD1);
        hex_to_7seg temp4(HEX2,BCD2);
        hex_to_7seg temp5(HEX3,BCD3);
        hex_to_7seg temp6(HEX4,BCD4);
endmodule
```

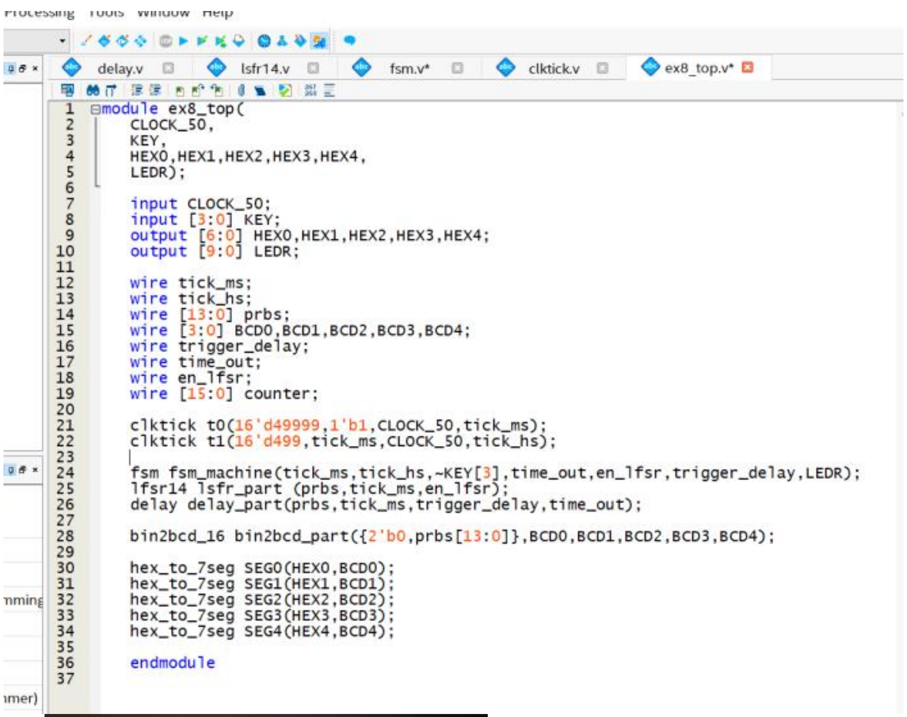
To make the divide by 50,000 tick we make a clkdiv module:

```
module clkdiv(clock_in,clock_out);

        input clock_in; // input clock on FPGA
        output reg clock_out; // output clock after dividing the input clock by divisor
        reg[15:0] counter=16'd0;
        parameter HIGH = 16'd49999;

        always @(posedge clock_in)
                begin
                        if(counter == HIGH)
                                begin
                                        counter <= 0;
                                        clock_out <= 1'b1;
                                end
                        else
                                begin
                                        counter <= counter + 1;
                                        clock_out <= 1'b0;
                                end
                end
endmodule
```

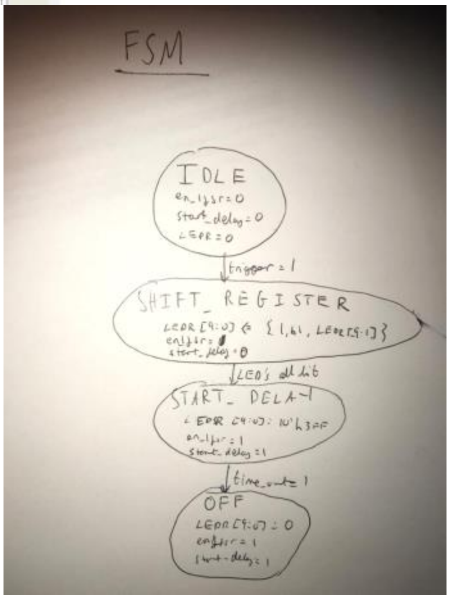
This is developed from the one given in lecture, except with the parameter changed so that it counts to 49,999 before givinga short pulse of one, and then repeating the process.

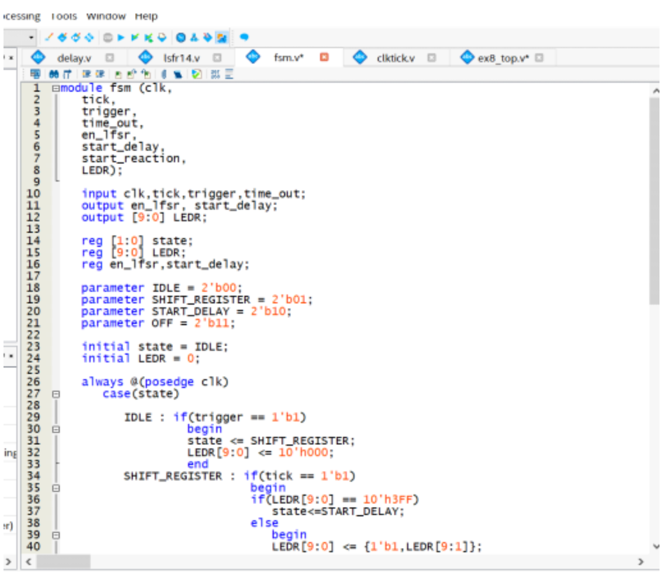
Optional - Experiment 8

1. The circuit is triggered (or started) by pressing KEY[3] (don't forget KEY[3] is low active);
2. The 10 LEDs (below the 7-segment displays) will then start lighting up from left to right at 0.5 second interval, until all LEDs are ON;
3. The circuit then waits for a random period of time between 0.25 and 16 seconds before all LEDs turn OFF;
4. You should also display the random delay period in milliseconds on five 7-segment displays.

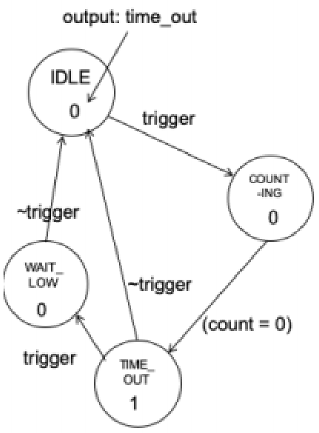


```
module ex8_top(
    CLOCK_50,
    KEY,
    HEX0,HEX1,HEX2,HEX3,HEX4,
    LEDR);

    input CLOCK_50;
    input [3:0] KEY;
    output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4;
    output [9:0] LEDR;

    wire tick_ms;
    wire tick_hs;
    wire [13:0] prbs;
    wire [3:0] BCD0,BCD1,BCD2,BCD3,BCD4;
    wire trigger_delay;
    wire time_out;
    wire en_lfsr;
    wire [15:0] counter;

    clktick t0(16'd49999,1'b1,CLOCK_50,tick_ms);
    clktick t1(16'd499,tick_ms,CLOCK_50,tick_hs);

    fsm fsm_machine(tick_ms,tick_hs,~KEY[3],time_out,en_lfsr,trigger_delay,LEDR);
    lfsr14 lsfr_part (prbs,tick_ms,en_lfsr);
    delay delay_part(prbs,tick_ms,trigger_delay,time_out);

    bin2bcd_16 bin2bcd_part({2'b0,prbs[13:0]},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
```

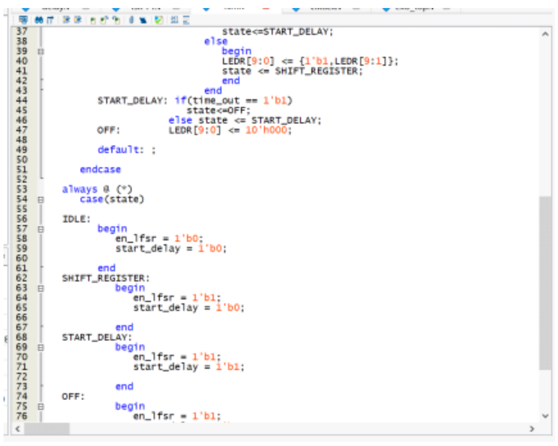




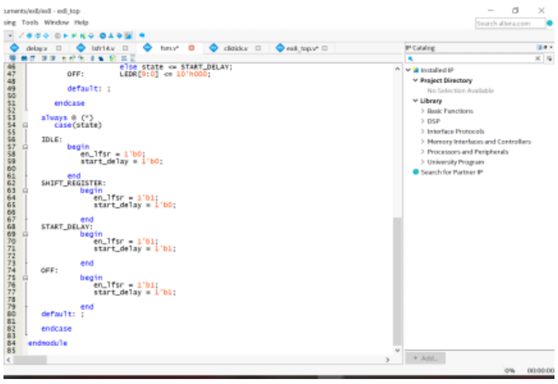
```
module fsm (clk,
    tick,
    trigger,
    time_out,
    en_lfsr,
    start_delay,
    start_reaction,
    LEDR);

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

    reg [1:0] state;
    reg [9:0] LEDR;
    reg en_lfsr,start_delay;

    parameter IDLE = 2'b00;
    parameter SHIFT_REGISTER = 2'b01;
    parameter START_DELAY = 2'b10;
    parameter OFF = 2'b11;

    initial state = IDLE;
    initial LEDR = 0;

    always @(posedge clk)
       case(state)

          IDLE : if(trigger == 1'b1)
                    begin
                    state <= SHIFT_REGISTER;
                    LEDR[9:0] <= 10'h000;
                    end
          SHIFT_REGISTER : if(tick == 1'b1)
                    begin
                    if(LEDR[9:0] == 10'h3FF)
                       state<=START_DELAY;
                    else
                       begin
                       LEDR[9:0] <= {1'b1,LEDR[9:1]};
```



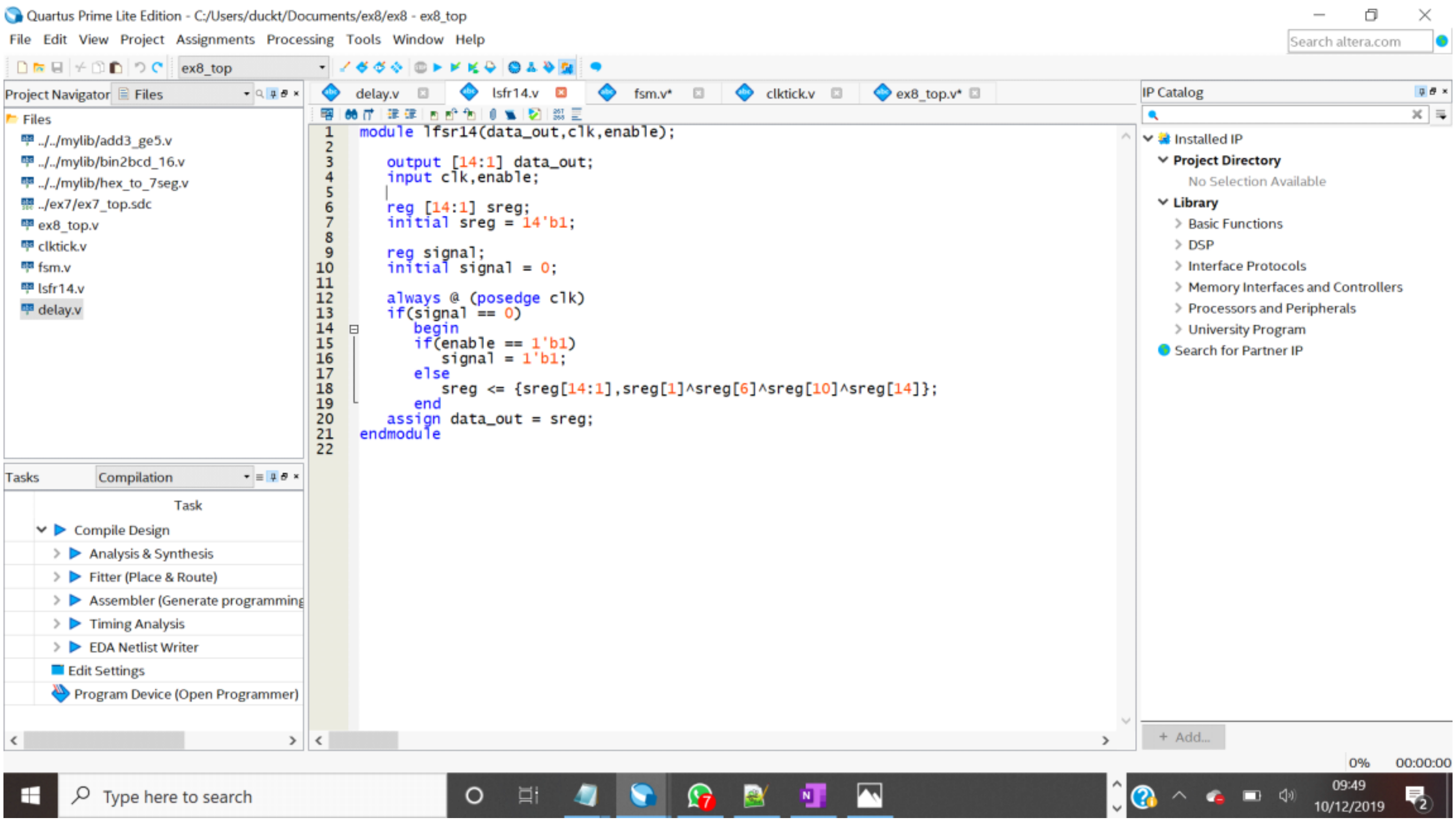
The diagram for delay module:



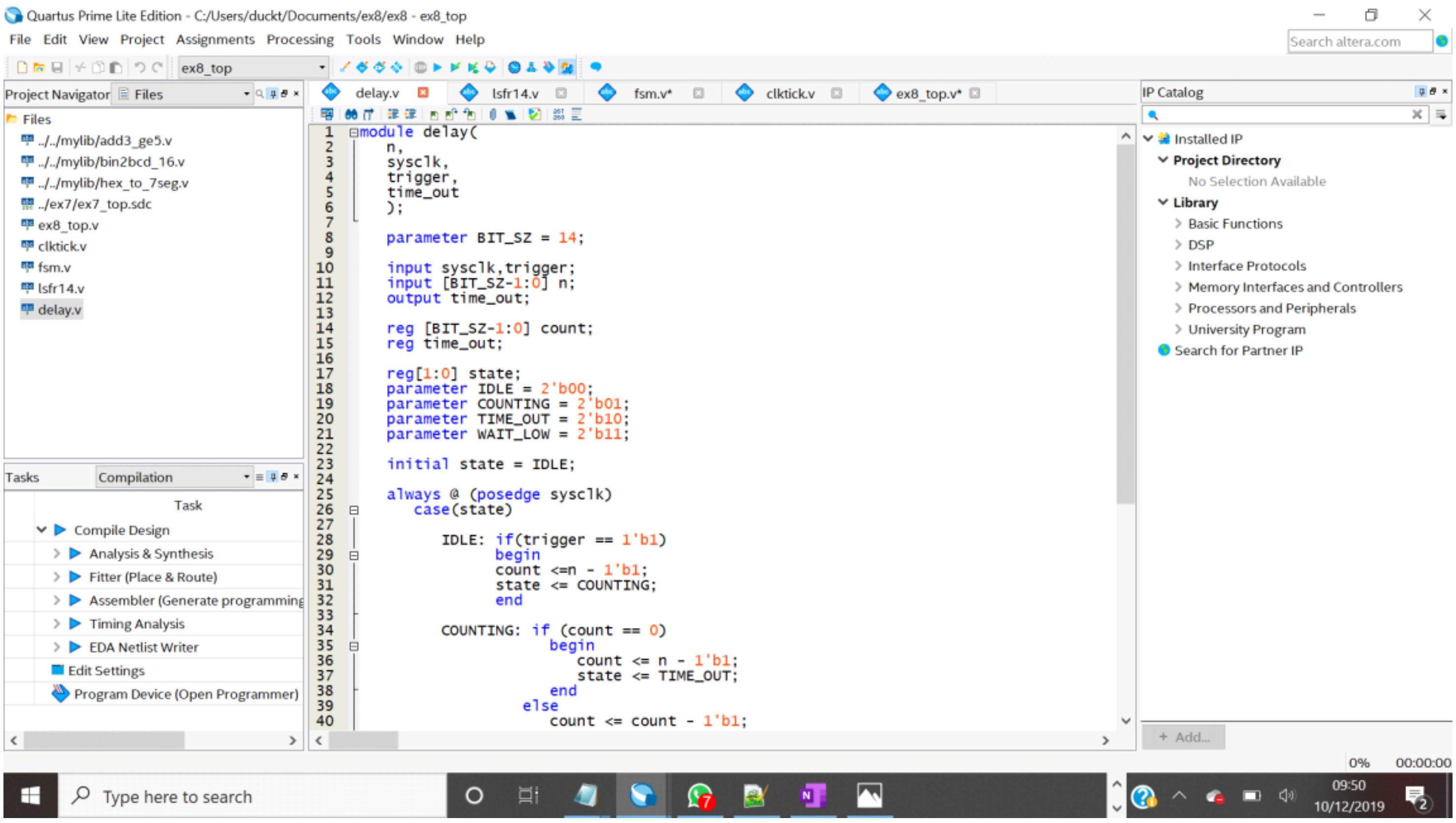
```
                              state<=START_DELAY;
                           else
                              begin
                              LEDR[9:0] <= {1'b1,LEDR[9:1]};
                              state <= SHIFT_REGISTER;
                              end
                           end
          START_DELAY: if(time_out == 1'b1)
                          state<=OFF;
                       else state <= START_DELAY;
          OFF:         LEDR[9:0] <= 10'h000;

          default: ;

       endcase

    always @ (*)
       case(state)

    IDLE:
          begin
             en_lfsr = 1'b0;
             start_delay = 1'b0;

          end
    SHIFT_REGISTER:
             begin
                en_lfsr = 1'b1;
                start_delay = 1'b0;

             end
    START_DELAY:
             begin
                en_lfsr = 1'b1;
                start_delay = 1'b1;

             end
    OFF:
             begin
                en_lfsr = 1'b1;
```



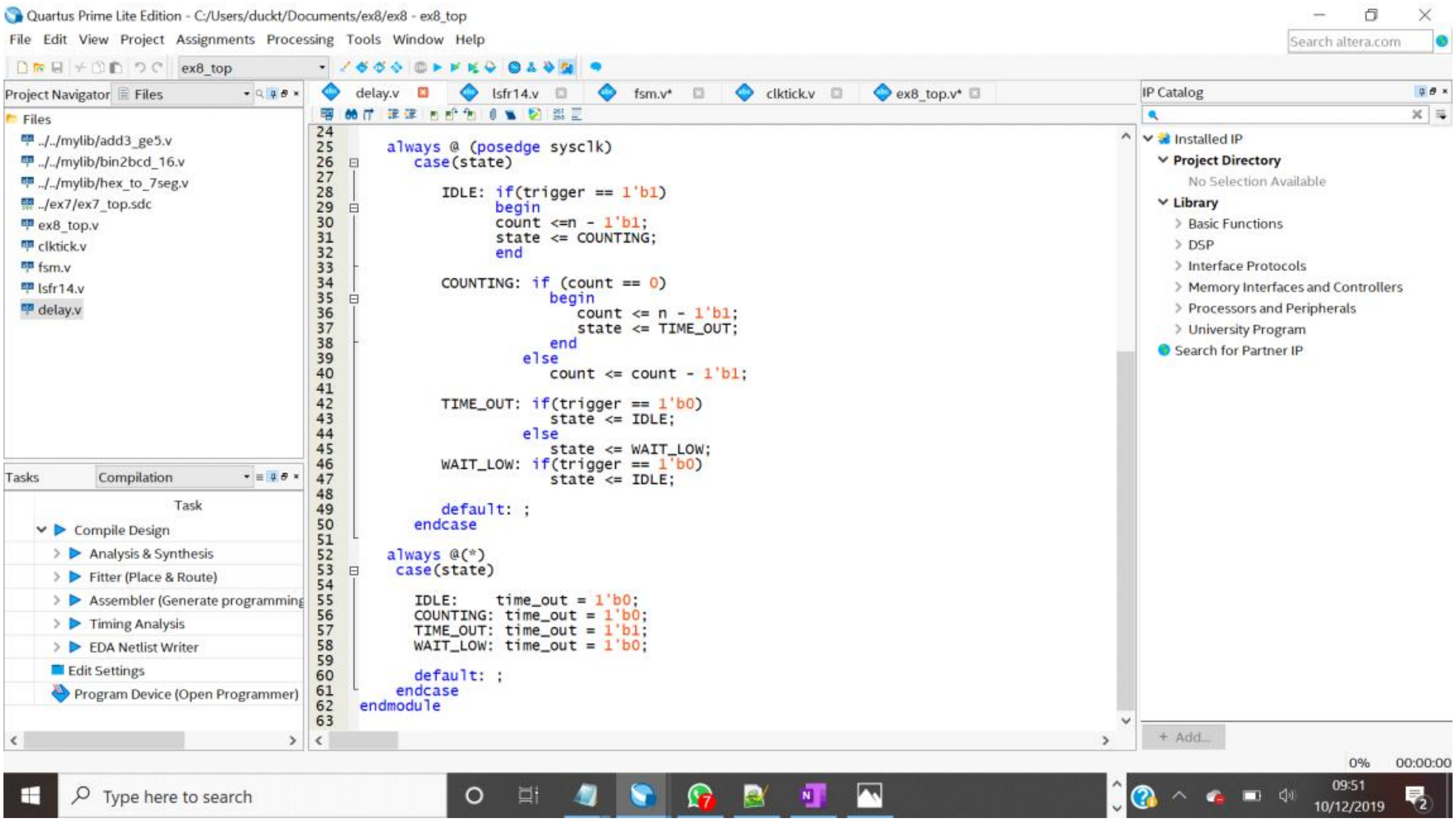
```
                       else state <= START_DELAY;
          OFF:         LEDR[9:0] <= 10'h000;

          default: ;

       endcase

    always @ (*)
       case(state)

    IDLE:
          begin
             en_lfsr = 1'b0;
             start_delay = 1'b0;

          end
    SHIFT_REGISTER:
             begin
                en_lfsr = 1'b1;
                start_delay = 1'b0;

             end
    START_DELAY:
             begin
                en_lfsr = 1'b1;
                start_delay = 1'b1;

             end
    OFF:
             begin
                en_lfsr = 1'b1;
                start_delay = 1'b1;

             end
    default: ;

    endcase

endmodule
```



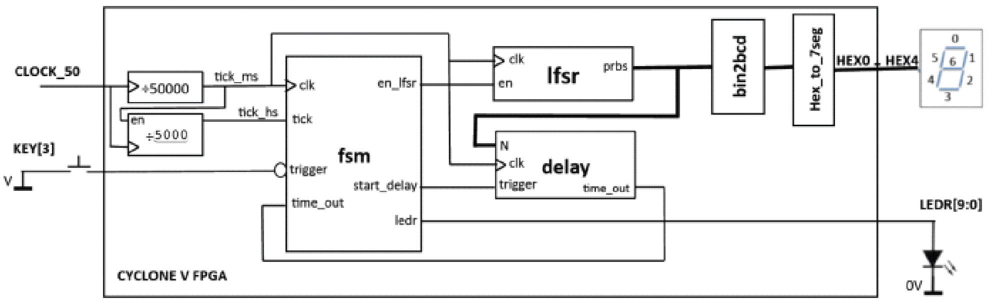
```verilog
module lfsr14(data_out,clk,enable);

   output [14:1] data_out;
   input clk,enable;

   reg [14:1] sreg;
   initial sreg = 14'b1;

   reg signal;
   initial signal = 0;

   always @ (posedge clk)
   if(signal == 0)
      begin
      if(enable == 1'b1)
         signal = 1'b1;
      else
         sreg <= {sreg[14:1],sreg[1]^sreg[6]^sreg[10]^sreg[14]};
      end
   assign data_out = sreg;
endmodule
```



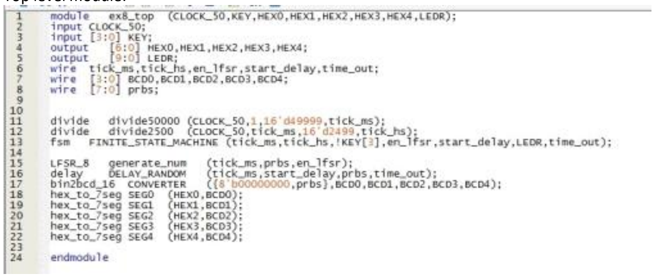
```verilog
module delay(
   n,
   sysclk,
   trigger,
   time_out
   );

   parameter BIT_SZ = 14;

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

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

   reg[1:0] state;
   parameter IDLE = 2'b00;
   parameter COUNTING = 2'b01;
   parameter TIME_OUT = 2'b10;
   parameter WAIT_LOW = 2'b11;

   initial state = IDLE;

   always @ (posedge sysclk)
      case(state)

         IDLE: if(trigger == 1'b1)
               begin
               count <=n - 1'b1;
               state <= COUNTING;
               end

         COUNTING: if (count == 0)
                     begin
                        count <= n - 1'b1;
                        state <= TIME_OUT;
                     end
                   else
                     count <= count - 1'b1;
```



```verilog

	always @ (posedge sysclk)
	  case(state)

		IDLE: if(trigger == 1'b1)
				begin
				count <=n - 1'b1;
				state <= COUNTING;
				end

		COUNTING: if (count == 0)
					begin
						count <= n - 1'b1;
						state <= TIME_OUT;
					end
				else
					count <= count - 1'b1;

		TIME_OUT: if(trigger == 1'b0)
					state <= IDLE;
				else
					state <= WAIT_LOW;
		WAIT_LOW: if(trigger == 1'b0)
					state <= IDLE;

		default: ;
	  endcase

	always @(*)
	 case(state)

	  IDLE:     time_out = 1'b0;
	  COUNTING: time_out = 1'b0;
	  TIME_OUT: time_out = 1'b1;
	  WAIT_LOW: time_out = 1'b0;

	  default: ;
	 endcase
endmodule

```



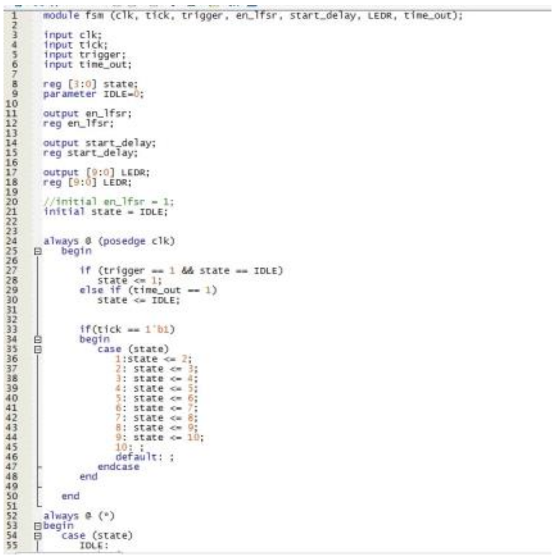
Top level module:



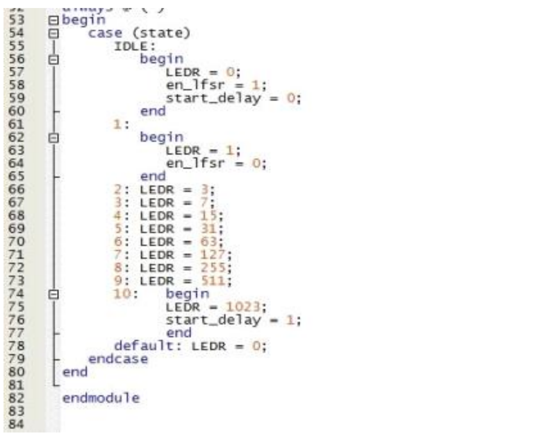
```verilog
module   ex8_top  (CLOCK_50,KEY,HEX0,HEX1,HEX2,HEX3,HEX4,LEDR);
input CLOCK_50;
input [3:0] KEY;
output   [6:0] HEX0,HEX1,HEX2,HEX3,HEX4;
output   [9:0] LEDR;
wire  tick_ms,tick_hs,en_lfsr,start_delay,time_out;
wire  [3:0] BCD0,BCD1,BCD2,BCD3,BCD4;
wire  [7:0] prbs;


divide   divide50000 (CLOCK_50,1,16'd49999,tick_ms);
divide   divide2500  (CLOCK_50,tick_ms,16'd2499,tick_hs);
fsm   FINITE_STATE_MACHINE (tick_ms,tick_hs,!KEY[3],en_lfsr,start_delay,LEDR,time_out);

LFSR_8   generate_num   (tick_ms,prbs,en_lfsr);
delay    DELAY_RANDOM   (tick_ms,start_delay,prbs,time_out);
bin2bcd_16  CONVERTER   ({8'b00000000,prbs},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 create one divide clock module, that takes in a number as parameter to count up to. This way we could reuse the module rather than make two for the different clocks.
The top level is wired up to follow the following schematic:
The converter takes in a 16 bit number, but prbs is only 8 bits so had to be concatenated to make the correct size.

FSM module:



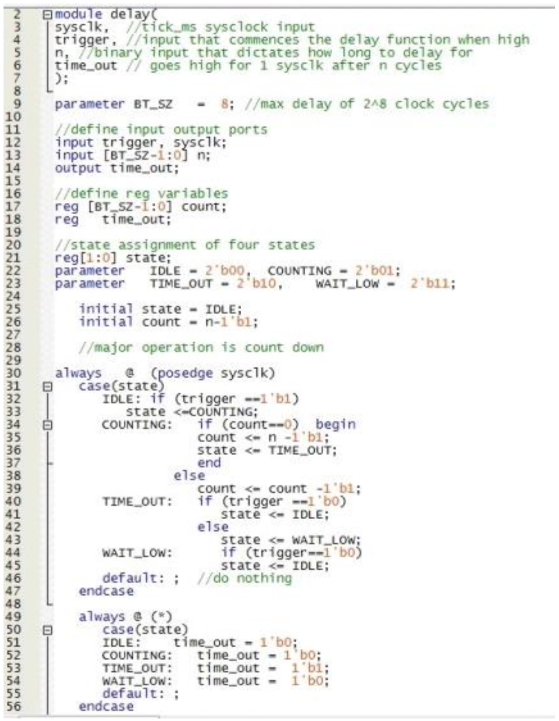
```
module fsm (clk, tick, trigger, en_lfsr, start_delay, LEDR, time_out);

input clk;
input tick;
input trigger;
input time_out;

reg [3:0] state;
parameter IDLE=0;

output en_lfsr;
reg en_lfsr;

output start_delay;
reg start_delay;

output [9:0] LEDR;
reg [9:0] LEDR;

//initial en_lfsr = 1;
initial state = IDLE;


always @ (posedge clk)
   begin

      if (trigger == 1 && state == IDLE)
         state <= 1;
      else if (time_out == 1)
         state <= IDLE;


      if(tick == 1'b1)
      begin
         case (state)
            1:state <= 2;
            2: state <= 3;
            3: state <= 4;
            4: state <= 5;
            5: state <= 6;
            6: state <= 7;
            7: state <= 8;
            8: state <= 9;
            9: state <= 10;
            10: ;
            default: ;
         endcase
      end

   end

always @ (*)
begin
   case (state)
      IDLE:
```



```
begin
   case (state)
      IDLE:
         begin
            LEDR = 0;
            en_lfsr = 1;
            start_delay = 0;
         end
      1:
         begin
            LEDR = 1;
            en_lfsr = 0;
         end
      2: LEDR = 3;
      3: LEDR = 7;
      4: LEDR = 15;
      5: LEDR = 31;
      6: LEDR = 63;
      7: LEDR = 127;
      8: LEDR = 255;
      9: LEDR = 511;
      10:    begin
            LEDR = 1023;
            start_delay = 1;
            end
      default: LEDR = 0;
   endcase
end

endmodule
```

As shown in lectures we use binary encoded states. A state 'wait' must be included when waiting for the input from KEY3. Thismeans nothing happens as state IDLE is not in the case(state) statement, and so state is not incremented until it is set to 1 first, which only happens when trigger == 1 (the key is pressed). The LFSR is initiallyenabled. When the key has been pressed the counting starts, the LSFR enable is set to 0, and the LEDR set to the numbers worked out for the characteristic polynomial in experiment 7. The delay module is activated and waits for the specified time before turning all LEDs off.

Delay module:

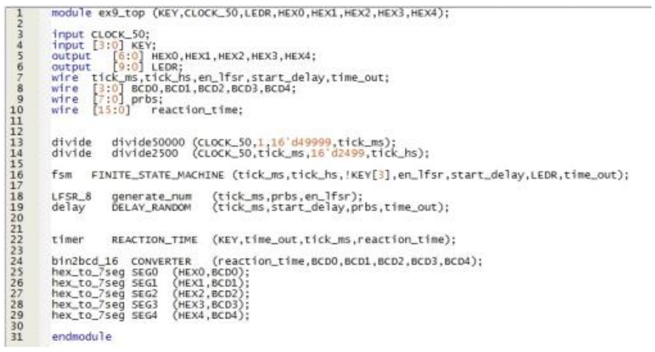


```
module delay(
sysclk,  //tick_ms sysclock input
trigger, //input that commences the delay function when high
n, //binary input that dictates how long to delay for
time_out // goes high for 1 sysclk after n cycles
);

parameter BT_SZ   =  8; //max delay of 2^8 clock cycles

//define input output ports
input trigger, sysclk;
input [BT_SZ-1:0] n;
output time_out;

//define reg variables
reg [BT_SZ-1:0] count;
reg   time_out;

//state assignment of four states
reg[1:0] state;
parameter   IDLE = 2'b00,  COUNTING = 2'b01;
parameter   TIME_OUT = 2'b10,    WAIT_LOW =  2'b11;

   initial state = IDLE;
   initial count = n-1'b1;

   //major operation is count down

always   @  (posedge sysclk)
   case(state)
      IDLE: if (trigger ==1'b1)
         state <=COUNTING;
      COUNTING:   if (count==0)  begin
                  count <= n -1'b1;
                  state <= TIME_OUT;
                  end
               else
                  count <= count -1'b1;
      TIME_OUT:   if (trigger ==1'b0)
                     state <= IDLE;
                  else
                     state <= WAIT_LOW;
      WAIT_LOW:      if (trigger==1'b0)
                     state <= IDLE;
      default: ;  //do nothing
   endcase

   always @ (*)
      case(state)
      IDLE:     time_out = 1'b0;
      COUNTING:   time_out = 1'b0;
      TIME_OUT:   time_out =  1'b1;
      WAIT_LOW:   time_out =  1'b0;
      default: ;
   endcase
```

Delay module was given in the lectures. Here we use it and change the size to 8.
The module detects a rising edge on the trigger, and counts n clock cycles, then outputs a pulse on time_out.

Optional - Experiment 9
Following Experiment 8, we add one extra module 'timer', which counts the time it takes for the user to press KEY0.

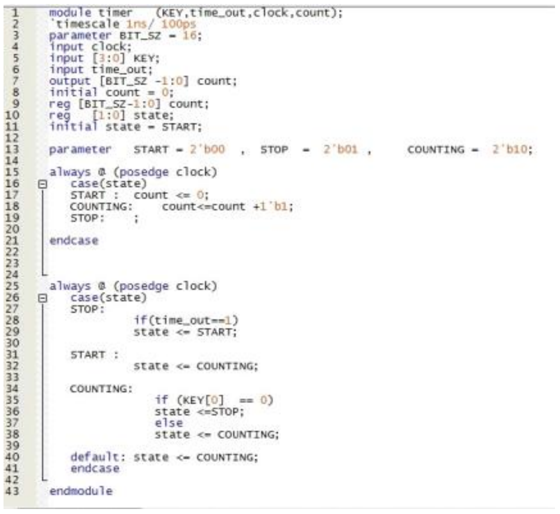
Top level module:



```verilog
module ex9_top (KEY,CLOCK_50,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4);

input CLOCK_50;
input [3:0] KEY;
output    [6:0] HEX0,HEX1,HEX2,HEX3,HEX4;
output    [9:0] LEDR;
wire  tick_ms,tick_hs,en_lfsr,start_delay,time_out;
wire  [3:0] BCD0,BCD1,BCD2,BCD3,BCD4;
wire  [7:0] prbs;
wire  [15:0]   reaction_time;


divide   divide50000 (CLOCK_50,1,16'd49999,tick_ms);
divide   divide2500  (CLOCK_50,tick_ms,16'd2499,tick_hs);

fsm   FINITE_STATE_MACHINE (tick_ms,tick_hs,!KEY[3],en_lfsr,start_delay,LEDR,time_out);

LFSR_8   generate_num   (tick_ms,prbs,en_lfsr);
delay    DELAY_RANDOM   (tick_ms,start_delay,prbs,time_out);


timer    REACTION_TIME  (KEY,time_out,tick_ms,reaction_time);

bin2bcd_16  CONVERTER   (reaction_time,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
```

This is the same as experiment 8 but with timer module included, and that the output from this is input into the BCD converter, and onto the LEDs.

Timer module:



```verilog
module timer   (KEY,time_out,clock,count);
`timescale 1ns/ 100ps
parameter BIT_SZ = 16;
input clock;
input [3:0] KEY;
input time_out;
output [BIT_SZ -1:0] count;
initial count = 0;
reg [BIT_SZ-1:0] count;
reg   [1:0] state;
initial state = START;

parameter   START = 2'b00  ,  STOP  =  2'b01 ,     COUNTING =  2'b10;

always @ (posedge clock)
   case(state)
   START :  count <= 0;
   COUNTING:    count<=count +1'b1;
   STOP:    ;

endcase



always @ (posedge clock)
   case(state)
   STOP:
            if(time_out==1)
            state <= START;

   START :
            state <= COUNTING;

   COUNTING:
               if (KEY[0]  == 0)
               state <=STOP;
               else
               state <= COUNTING;

   default: state <= COUNTING;
   endcase

endmodule
```

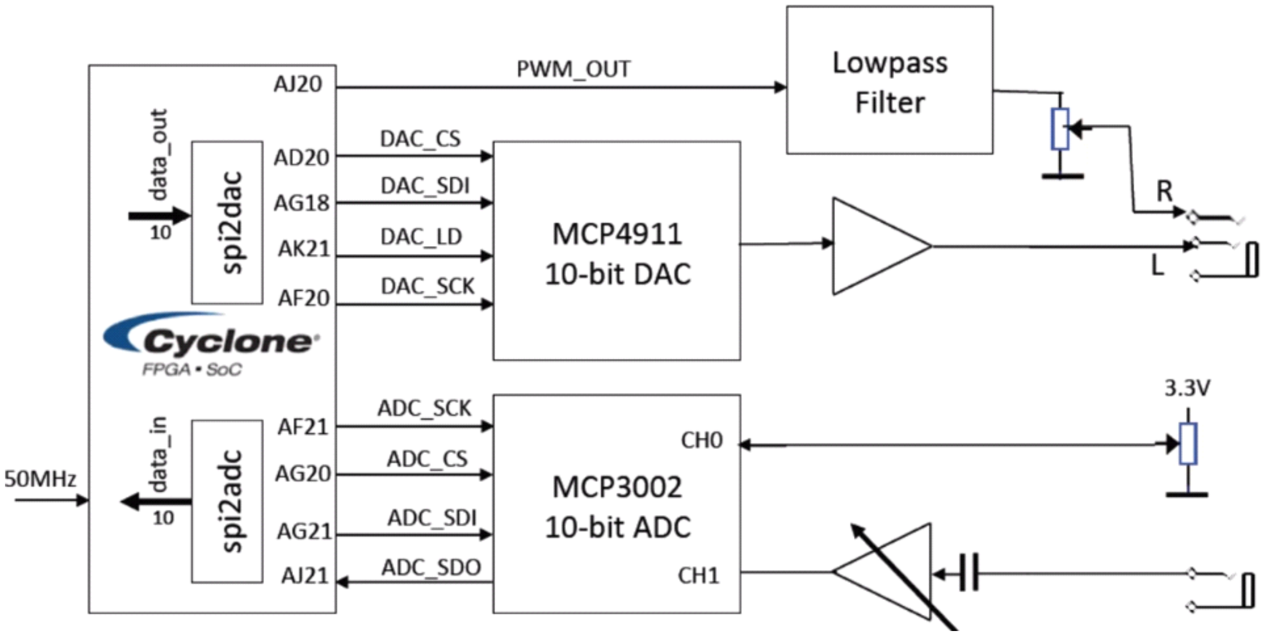
We use one hot encoding to define the states. START sets count to 0, COUNTING increments count, and STOP does no operation. If the state is in START, then it is updated to counting, if the state is in COUNTING and the key pressed, then state is updated to STOP, otherwise it continues counting. If the state is in STOP then the state only changes to START if time_out pulse has been sent.

Operation order: time_out pulse -> state = start, LEDs display count = 0 -> state=count, LEDs display count -> key pressed-> state = STOP -> LEDs display (final) count.

Part 3

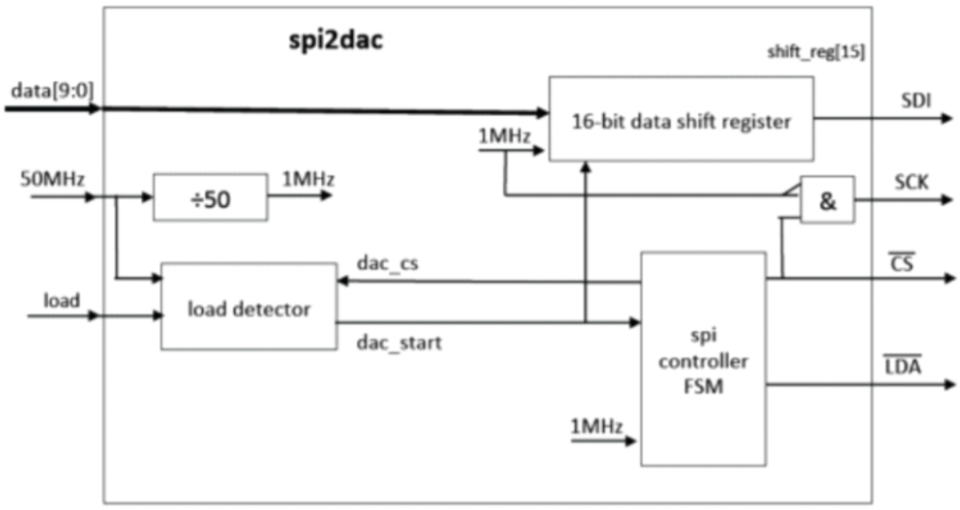
06 December 2019 20:09

Experiment 10
Schematic of Analogue I/O card:



Screen clipping taken: 09/12/2019 13:24

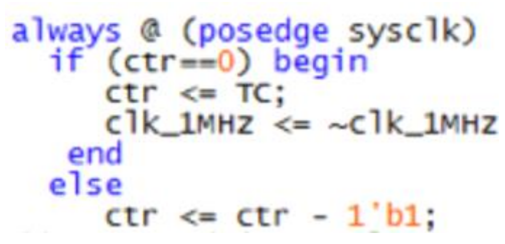
Spi2dac code:



Screen clipping taken: 04/12/2019 17:28

Spi2dac takes in 2 inputs, the 10-bit data to be converted by DAC, and a load signal which triggers the data transfer and conversion to begin.

The ÷50 module produces a 1MHz clock for the FSM.

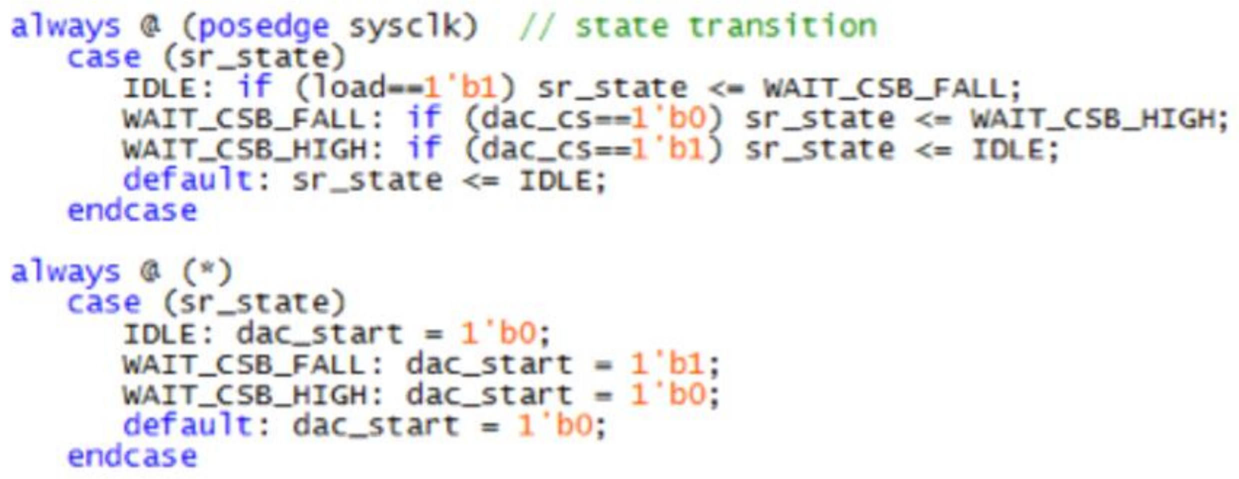


```
always @ (posedge sysclk)
  if (ctr==0) begin
     ctr <= TC;
     clk_1MHz <= ~clk_1MHz
   end
  else
     ctr <= ctr - 1'b1;
```

Screen clipping taken: 04/12/2019 17:49

It does this by toggling the output after 25 cycles (TC is set to 5'd24)

Load detector produces control signals to the SPI state machine and shift register.



```
always @ (posedge sysclk)  // state transition
   case (sr_state)
      IDLE: if (load==1'b1) sr_state <= WAIT_CSB_FALL;
      WAIT_CSB_FALL: if (dac_cs==1'b0) sr_state <= WAIT_CSB_HIGH;
      WAIT_CSB_HIGH: if (dac_cs==1'b1) sr_state <= IDLE;
      default: sr_state <= IDLE;
   endcase

always @ (*)
   case (sr_state)
      IDLE: dac_start = 1'b0;
      WAIT_CSB_FALL: dac_start = 1'b1;
      WAIT_CSB_HIGH: dac_start = 1'b0;
      default: dac_start = 1'b0;
   endcase
```

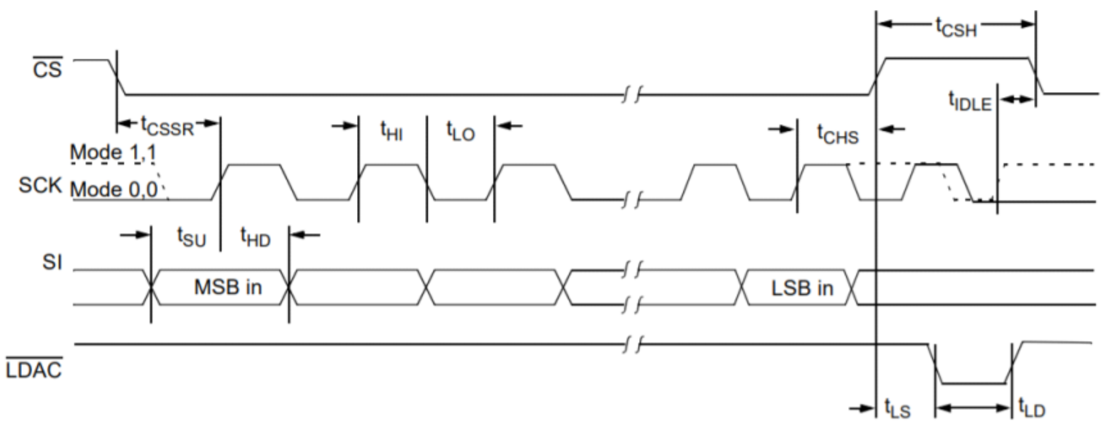
Screen clipping taken: 04/12/2019 18:20

At each rising edge the state of FSM is checked, and updated according to the load/DAC_CS signals. The second always block sets DAC_START to high only if FSM is waiting for falling edge

Shift register sends the control bits and 10 bit data to SDI output

Predicted waveform:

Datasheet timing diagram:



Screen clipping taken: 04/12/2019 20:51

So according to this we predict the following output:

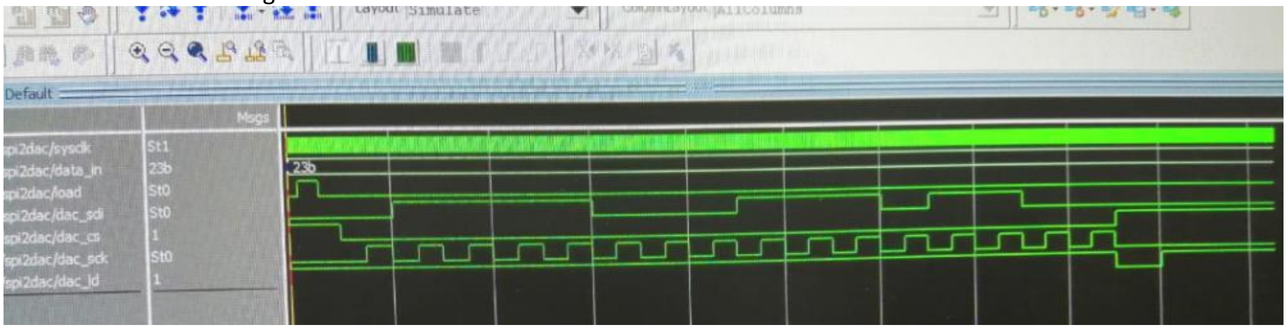
| Insert drawn output here|

We write the following top level code for the diagram supplied

```
module ex10_top(SW,CLOCK_50,DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);

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

        clktick_16 newclock(CLOCK_50,clock_out);
        spi2dac temp(CLOCK_50, SW, clock_out, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);

endmodule
```

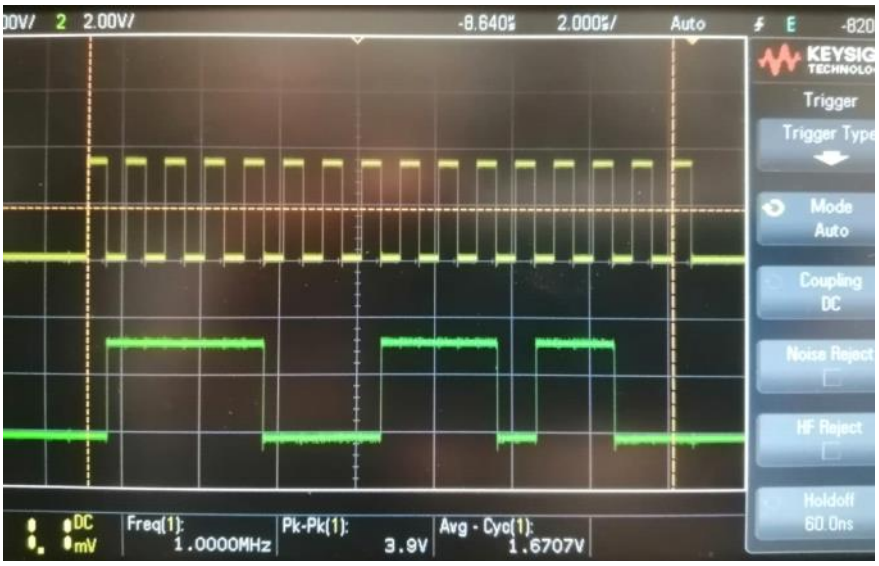
This is just connecting the relevant inputs/outputs to the spi2dac.

Waveform in Modelsim after running with do file:



```
Msgs
spi2dac/sysclk      St1
spi2dac/data_in     23b
spi2dac/load        St0
spi2dac/dac_sdi     St0
spi2dac/dac_cs      1
spi2dac/dac_sck     St0
spi2dac/dac_ld      1
```

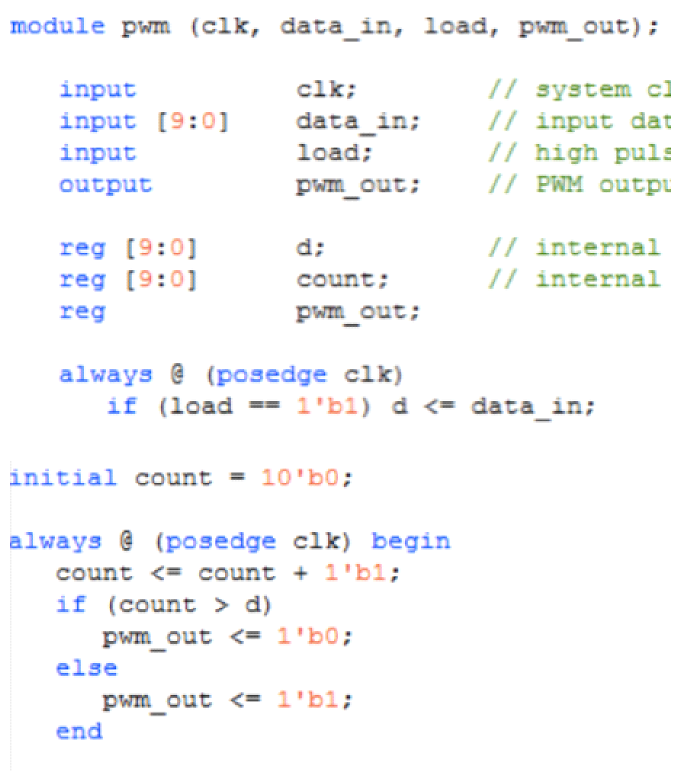
Actual waveform:



Comparing the signals we see that the predicted signal by Modelsim and the signal seen on the oscilloscope are the same when inputting the number 10'h23b.

Experiment 11

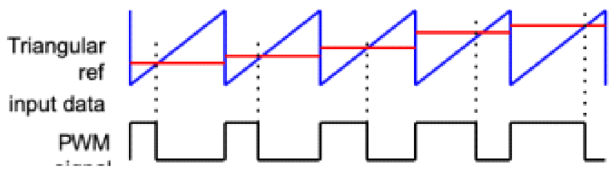
Pwm.v module:



```
module pwm (clk, data_in, load, pwm_out);

   input             clk;        // system cl
   input [9:0]       data_in;    // input dat
   input             load;       // high puls
   output            pwm_out;    // PWM outpu

   reg [9:0]         d;          // internal
   reg [9:0]         count;      // internal
   reg               pwm_out;

   always @ (posedge clk)
      if (load == 1'b1) d <= data_in;

initial count = 10'b0;

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

Screen clipping taken: 04/12/2019 21:46

Diagram of how PWM works:



Screen clipping taken: 09/12/2019 16:24

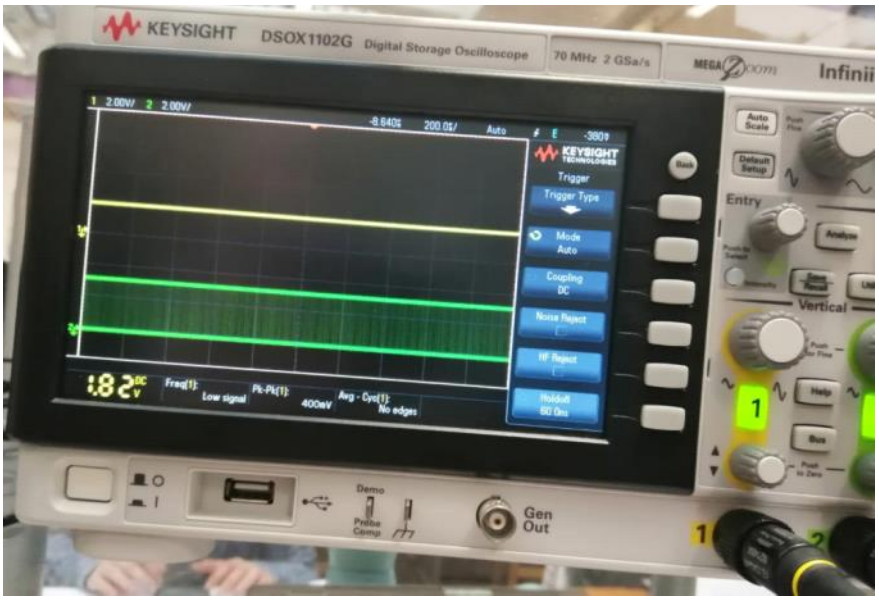
PWM essentially compares: is the counter value >= the data in value, if so go low, if not go high. Passing PWM through a lowpass filter gives an analogue output which is linearly related to data in.

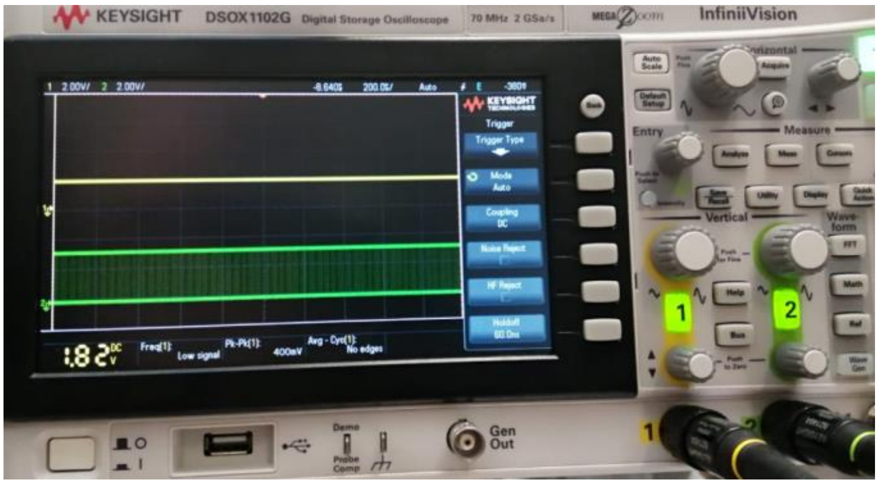
Top level function:

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

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

        clktick_16 temp1(CLOCK_50,clock_out);
        spi2dac temp2(CLOCK_50,SW,clock_out,DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);
        pwm temp3(CLOCK_50,SW,clock_out,PWM_OUT);

endmodule
```

Again, the top level function is just connecting input/outputs to the pwm and spi2dac.

Comparing the TP8 and TP9 values - the value is the same so we see there is no difference between using PWM or DAC.





For Pulse width modulation, the larger the input value the greater the it will be at logic high. Theoretically when all the Switches are 1, the pwm_out signal should be high all of the time, and depending upon the input, the RMS voltage will vary in proportion to the value of data_in applied. The other four outputs coming from the spi2dac module won't change.

There is a low pass filter on the add-on card, and the output seen is a flat line, retaining all the DC components but removing the high frequency components (hence why no variation in time at TP9).

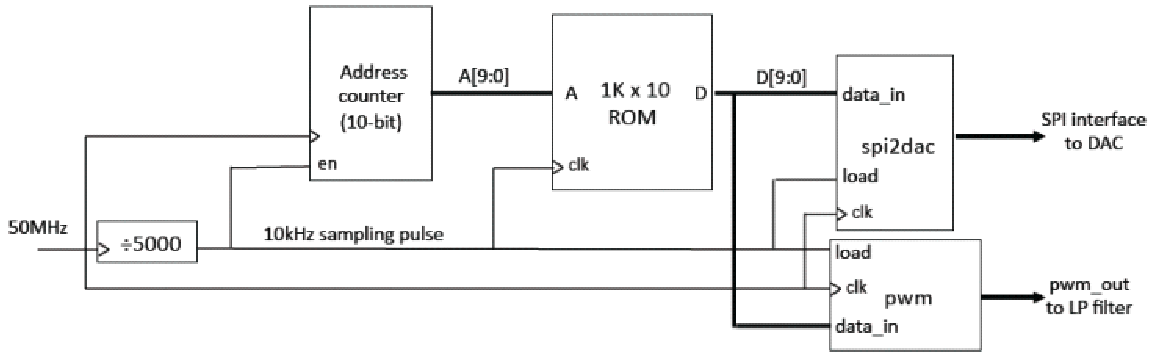
Experiment 12
Used the interface to create the ROM.
Top level module:

```
module ex12_top(SW,HEX0,HEX1,HEX2,CLOCK_50);

        input [9:0] SW;
        input CLOCK_50;
        output [6:0] HEX0,HEX1,HEX2;
        wire [9:0] q;

        ROM ROMtemp(SW[9:0],CLOCK_50,q[9:0]);

        hex_to_7seg SEG0(HEX0, q[3:0]);
        hex_to_7seg SEG1(HEX1, q[7:4]);
        hex_to_7seg SEG2(HEX2, q[9:8]);

endmodule
```

The ROM stores sine values, it must be offset because the DAC only accepts positive (0-1023) so we add 512.

Experiment 13

Schematic diagram for the top level



The ROM stores all the values of the sine wave as before, the counter is used to advance the phase of the sine wave (specified as the address X of ROM).

Top level module:

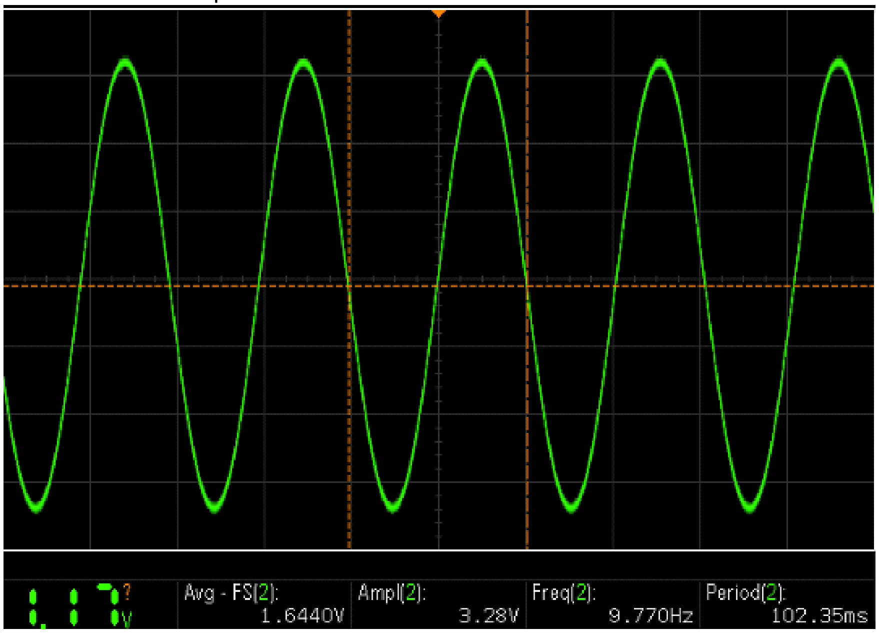
```
module ex13_top(CLOCK_50,DAC_CS,DAC_SDI,DAC_LD,DAC_SCK,PWM_OUT);

        input CLOCK_50;
        output DAC_CS,DAC_SDI,DAC_LD,DAC_SCK,PWM_OUT;
        wire tick;
        wire [9:0] count, out;

        clktick temp0(CLOCK_50,tick);

        counter_10 temp1(CLOCK_50,tick,count);

        ROM temp2(count,tick,out);

        spi2dac temp3(CLOCK_50,out,tick,DAC_SDI,DAC_CS,DAC_SCK,DAC_LD);

        pwm temp4(CLOCK_50,out,tick,PWM_OUT);


endmodule
```

For this we are just connecting together modules we have already made. We also make the counter 10 bit by adjusting the BIT_SZ as before in the counter module.

The frequency of the sine wave should be:

$$f = \frac{Clock\_f}{2^n \times Clock\_div\_f} = \frac{50 \times 10^6}{1024 \times 5000} = 9.76\ Hz$$

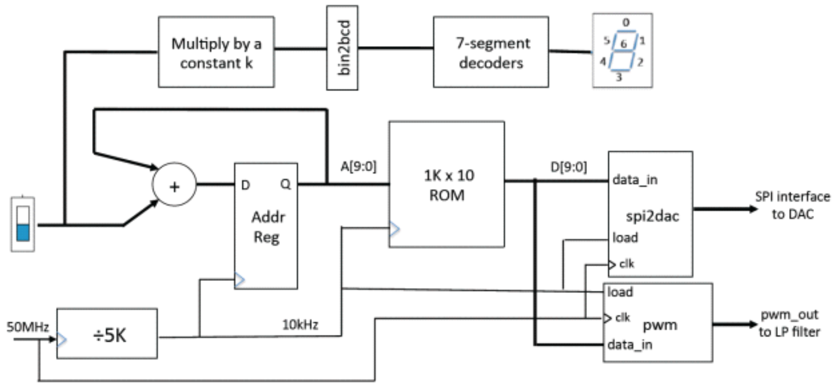
From the oscilloscope we measure:



Freq = 9.77 Hz, and amplitude 0-3.3V.

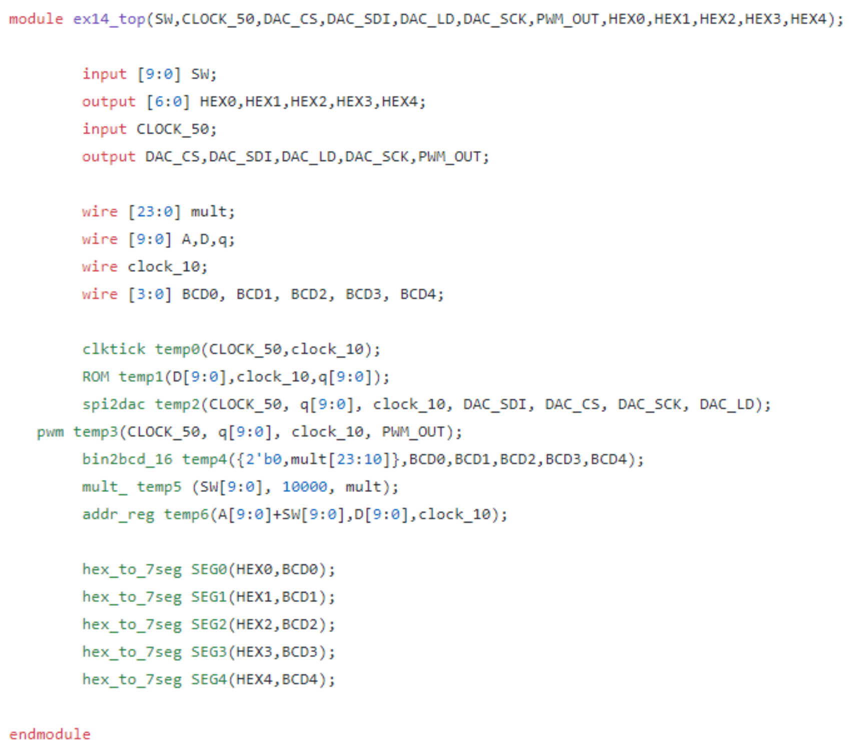
Optional - Experiment 14

Schematic diagram for the top level:



In order to get the frequency you take the top 14 bits as this is the same as dividing by 1024

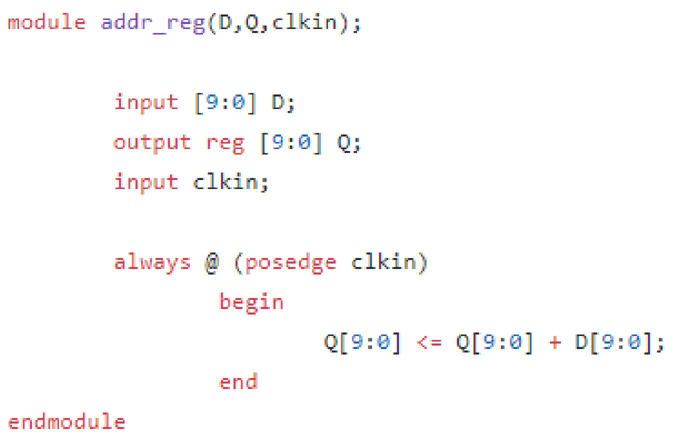
Top level module:



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

        input [9:0] SW;
        output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4;
        input CLOCK_50;
        output DAC_CS,DAC_SDI,DAC_LD,DAC_SCK,PWM_OUT;

        wire [23:0] mult;
        wire [9:0] A,D,q;
        wire clock_10;
        wire [3:0] BCD0, BCD1, BCD2, BCD3, BCD4;

        clktick temp0(CLOCK_50,clock_10);
        ROM temp1(D[9:0],clock_10,q[9:0]);
        spi2dac temp2(CLOCK_50, q[9:0], clock_10, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);
   pwm temp3(CLOCK_50, q[9:0], clock_10, PWM_OUT);
        bin2bcd_16 temp4({2'b0,mult[23:10]},BCD0,BCD1,BCD2,BCD3,BCD4);
        mult_ temp5 (SW[9:0], 10000, mult);
        addr_reg temp6(A[9:0]+SW[9:0],D[9:0],clock_10);

        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
```

Where clktick has been changed to count up to 5000 before giving a high pulse. Mult was made using the IP catalog.

Addr_reg:



```
module addr_reg(D,Q,clkin);

        input [9:0] D;
        output reg [9:0] Q;
        input clkin;

        always @ (posedge clkin)
                begin
                        Q[9:0] <= Q[9:0] + D[9:0];
                end
endmodule
```

This module simply adds the input and output of a flip flop, on a clock high.

This circuit works as following:

1. For a combination of input switches, the number is multiplied and then converted into BCD and displayed on 7-segment displays
2. This number is input to a D flip flop, and added to the current Q.
3. A 50MHz clock is divided with clktick which clocks the flip flop and ROM, and serves as the load to spi2dac and pwm.
4. The output Q is used as the address to obtain the value in ROM, which is used as the data in to the spi2dac and pwm.

Part 4

06 December 2019 20:09

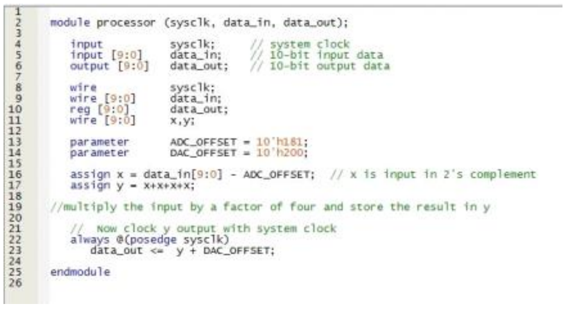
Part 4

Experiment 16

If you associate the signal names inside the module to outside it allows connections to be defined independent of order.

```
spi2adc SPI_ADC (
     .sysclk (CLOCK_50),
     .channel (1'b1),
     .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));
```

All pass module corrects ADC converter data by subtracting offset from data_out to get a 2's complement value x. It connects x to y, and converts y from 2's compliment to offset binary for DAC.

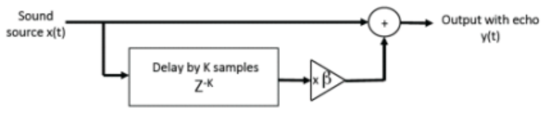


```

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
   assign y = x+x+x+x;

//multiply the input by a factor of four and store the result in y

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

endmodule

```

Mult.v can be done by shifting by 2 (assign y = x << 2), adding x 4 times to itself (as done here), or making a multiply module.

Experiment 17

Schematic:



Top level file is the same as ex 16 with the following additions:

```
pulse_gen             temp(pulse, data_valid, CLOCK_50);
processor             simple_echo (CLOCK_50, data_in, data_out, pulse);      // do some processing on the data
```

We create a FIFO component of size 8192 x 10 bit using the IP catalog. In FIFO received data is stored in a sequence so it can be retrieved in the order it arrived in. If data is retrieved it is removed which can cause the rates to be different. If the send is greater than retrieve rate then the buffer will get full, and should not receive more data. Empty buffers cannot provide data for retrieval. This FIFO is used to implement the delay.

Pulse generator module:

```
module pulse_gen(pulse, in, clk);

        output                  pulse;  // output pulse lasting one clk cycle
        input                   in;             // input, +ve edge to be detected
        input                   clk;            // clock signal

        reg [1:0]       state;
        reg     pulse;

        parameter       IDLE = 2'b0;    // state coding for IDLE state
        parameter       IN_HIGH = 2'b01;
        parameter       WAIT_LOW = 2'b10;

        initial state = IDLE;

        always @ (posedge clk)
                begin
                        pulse <= 0;             // default output
                        case (state)
                        IDLE:   if (in == 1'b1) begin
                                                        state <= IN_HIGH;       pulse <= 1'b1; end
                                                else
                                                        state <= IDLE;
                        IN_HIGH: if (in == 1'b1)
                                                        state <= WAIT_LOW;
                                                else
                                                        state <= IDLE;
                        WAIT_LOW: if (in == 1'b1)
                                                        state <= WAIT_LOW;
                                                 else
                                                        state <= IDLE;
                        default: ;
                        endcase
                end                     //... always
endmodule
```

This sets the state to IDLE, unless in == 1, in which case it only sets pulse to 1 if the current state is IDLE, in all other states they are set to WAIT_LOW.

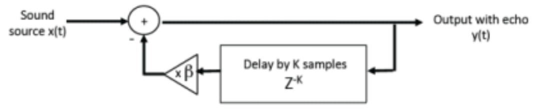
Processor module changes:

```
wire                            full;
wire    [9:0]           q;
reg                             DFF;
wire                            data_valid;

initial                         DFF = 0;

fifo                            delay(sysclk, x , (DFF&wrreq), wrreq, full, q[9:0]);

assign                  y = {q[9],q[9:1]} + x[9:0];
```

The FIFO is used to create a delay. The concatenation is effectively divide by 2 (it removes LSB, and extends by MSB) which implements the attenuation required. We also set the DFF <= full in the always block after this.

Experiment 18

The only change to this needed is to take away the output of the FIFO rather than add in the echo module. No other changes were made anywhere.
Schematic diagram:



Changes in echo module:

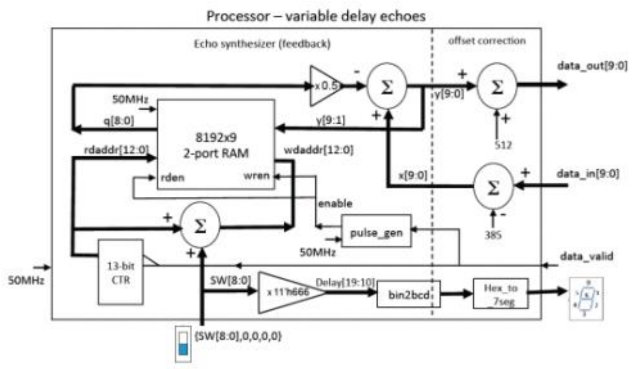
```
assign                  y = x[9:0] - {q[9],q[9:1]};
```

Experiment 19

Top level module changes:

```
processor variable_delay(CLOCK_50, data_in, data_out, data_valid,SW, BCD_0, BCD_1, BCD_2);
```

Variable delay:
Schematic:



Changes to processor:

```
assign                  wdaadr = rdaadr + {SW[8:0],4'b0};
assign                  Delay = SW[8:0] * 11'h666;
bin2bcd_16              temp({5'b0,Delay[19:10]},BCD_0, BCD_1, BCD_2, BCD_3, BCD_4);
pulse_gen               temp2(pulse,data_in,sysclk);
wire                            data_valid;

counter_13              mycount(sysclk,data_valid,rdaadr);
delay_ram               myRam(sysclk,y[9:1],rdaadr,pulse,wdaadr,pulse,q);


assign                  y = x[9:0] - {q[8],q[8],q[8:2]};
```

Write address = read address + switches (concatenated with 0's to make correct size)

Delay = switches multiplied by 1638
Take the most significant bits of Delay, concatenate to make correct size for BCD converter

Counter to get read address

Ram delay

As before to attenuate

We use the catalog IP to create the RAM, with one write port (to store ADC samples) and one read port. A 13 bit counter is used to generate the read address to the RAM - we reuse the module counter, and edit BIT_SZ to 13 (this is because the RAM is 8192 x 9 bit, and 2^13 = 8192). The address generator computes the address used on the next read and write cycle as it is incremented on the negative edge of the data valid signal. The write address = read address + SW[8:0]. Embedded memory in FPGA is 9 bit data width not 10 bit, so read value and write value are truncated to 9 bits. To display the delay value SW[8:0] is multiplied by 1638 and a constant. The most significant 10 bits is the delay in ms. Then it is converted from binary to BCD to display on 7 segs.

Part 3me

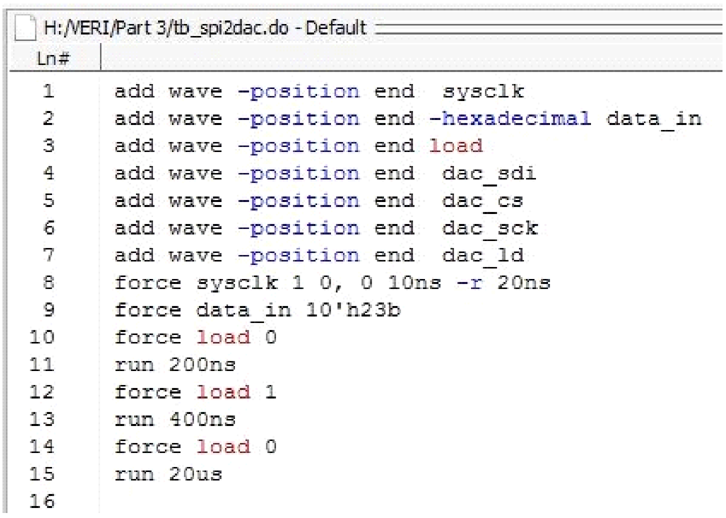
10 December 2019 09:18

EXPERIMENT 10 - Interface with the MCP4911 Digital-to-Analogue Converter

We have a DAC_CS signal that enables the DAC when it is low…

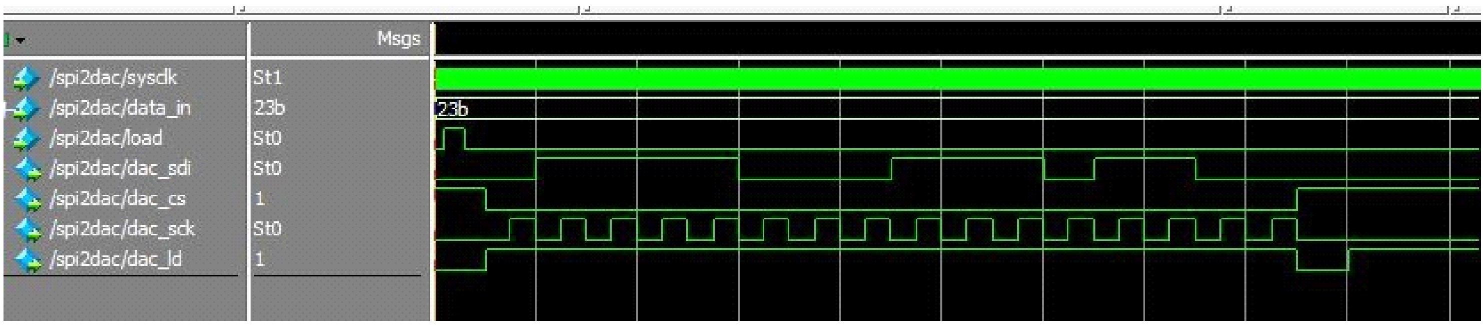
The following test bench do file enabled the SPI2DAC module to be loaded with a data input *data_in*
In addition to a system clock that will alow the data to be read upon the rising edge of the clock and read out as analogue voltages. Once the load pulse has gone high for a period for one clock cycle, then we commence the conversion from digital to analogue. All of the pins *dac_sdi, dac_cs, dac_sck, dac_ld* can be recorded off the various pins on the add on card circuit.

Configuration bit settings
Bit 15 = 0 : write to DAC register
Bit 14 = 0: input buffer control bit is 0
Bit 13 = 1: 1*vout
Bit 12 = 1: shutdownbar = 1. enable operation of DAC.

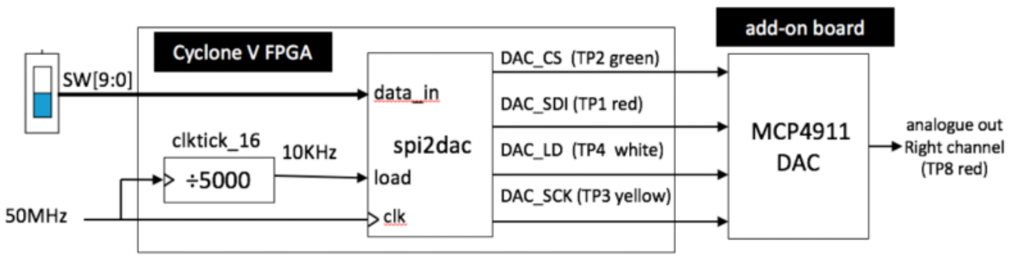

H:/VERI/Part 3/tb_spi2dac.do - Default

```
add wave -position end  sysclk
add wave -position end -hexadecimal data_in
add wave -position end load
add wave -position end  dac_sdi
add wave -position end  dac_cs
add wave -position end  dac_sck
add wave -position end  dac_ld
force sysclk 1 0, 0 10ns -r 20ns
force data_in 10'h23b
force load 0
run 200ns
force load 1
run 400ns
force load 0
run 20us
```

The results we obtained from this *.do* file being called upon in the command line I the following:

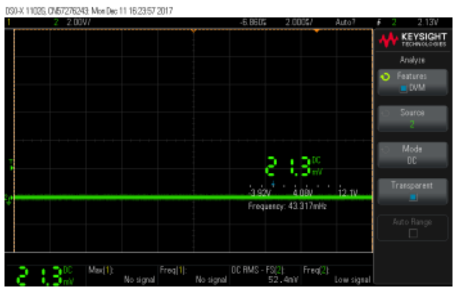


It is important to set up the waveforms to be displayed by using various *add wave* commands for the multiple inputs (including *load and data_in)* and outputs (*like dac_SCK,data_SDI)* that we have
Once the DAC has been loaded the *dac_ld* signal will remain high whilst all of the data is being read in from the *dac_sdi* input. As specified from the code above the system clock *sysclock* is specified as a 20ns square wave clock, and simply specify the load signal to give a short lasting pulse. Whenever the dac_ld signal is high, we have that the system clock behaves as it is expected to.



The **data_in** value determined by the 10 switches (**SW[9:0]**) is loaded to the **spi2dac** module at a rate of 10k samples per second as governed by the **load** signal. The steps for this part are:

Measurements on the Oscilloscope

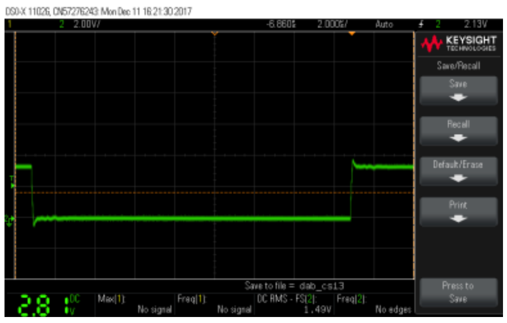


DAC output(TP8) with DVM when SW{9:0] =0

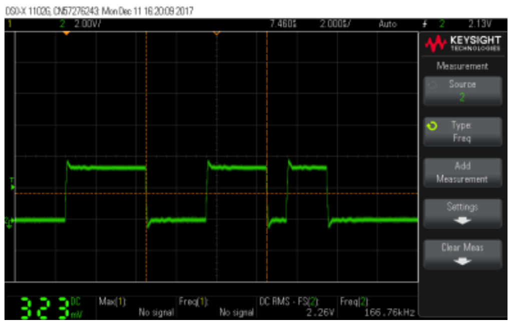
The main principle behind digital to analogue converter is that if we supply a larger binary input, then we will get a larger analogue voltage out and vice verca. We noticed from our readings that we when changed the switch positions, the DC voltage read would change in height for different values.

Theoretically, we require that the input divided by the maximum input represent the proportion of the maximum output voltage that we can obtain. More specifically:

$$\frac{IN[9:0]}{1024}V_{out_{max}} = V_{out} \quad where\ V_{outmax} = 3.3V$$



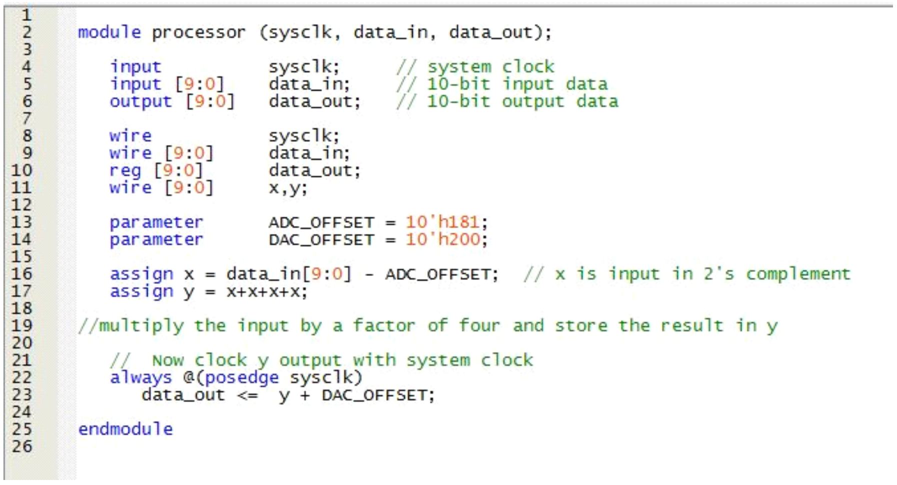
DAC_CS



DAC_SDI

Part 4 me

EXPERIMENT 16

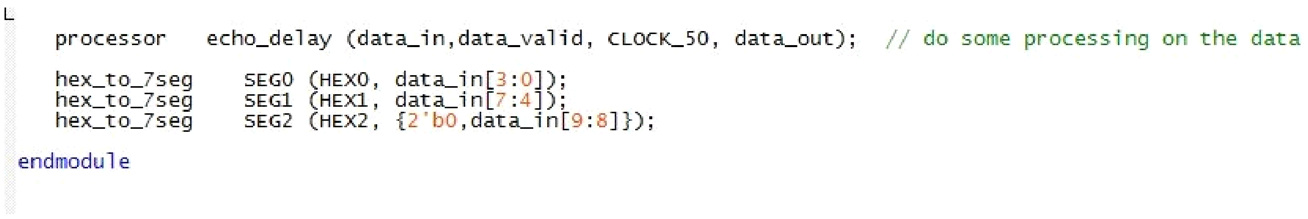


```verilog

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
   assign y = x+x+x+x;

//multiply the input by a factor of four and store the result in y

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

endmodule

```

The multiply block can be done by adding the same entity four times to itself. It can also be done by simply left shifting the binary number twice. However it can also be done by creating a multiply module.

The all pass processor does not do any signal processing other than just shift by two different quantities.

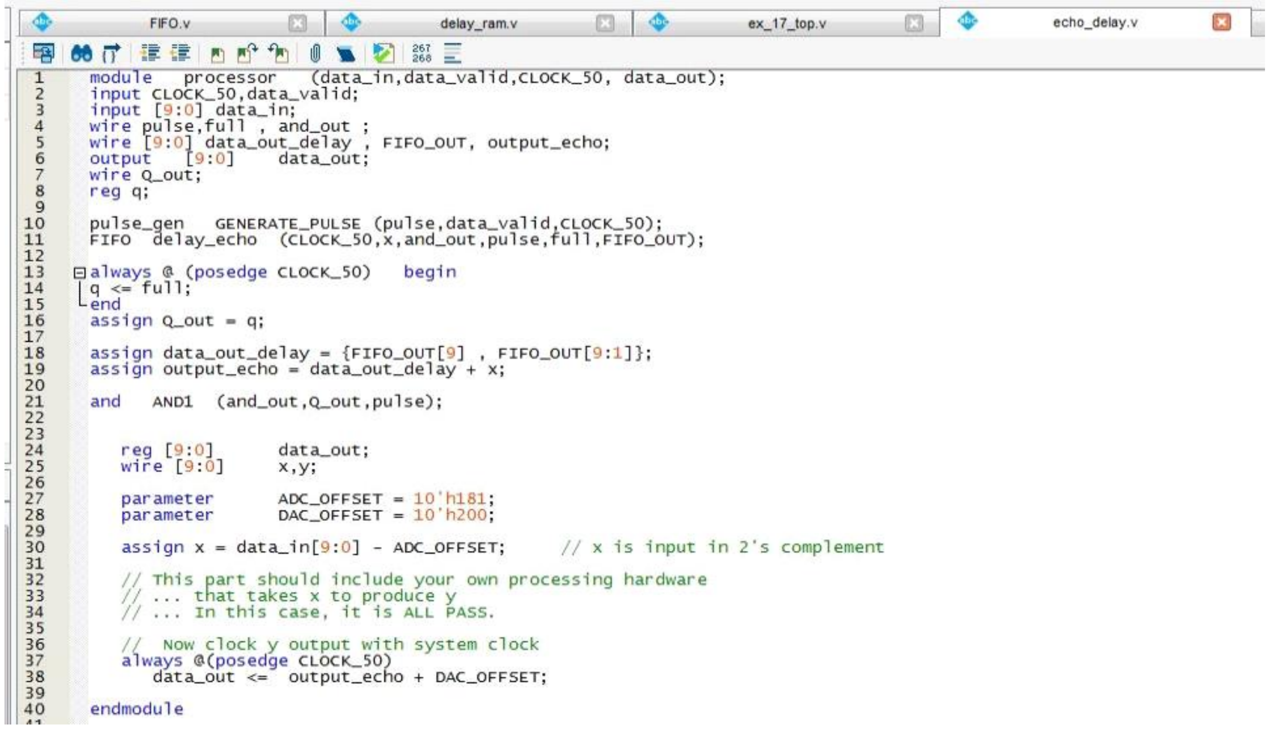
EXPERIMENT 17



```verilog
    processor   echo_delay (data_in,data_valid, CLOCK_50, 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
```

The above picture represents the only part that changed from the top file in experiment 16.
The other parts included instantiating the DAC and ADC modules, which need not be changed for this experiment.

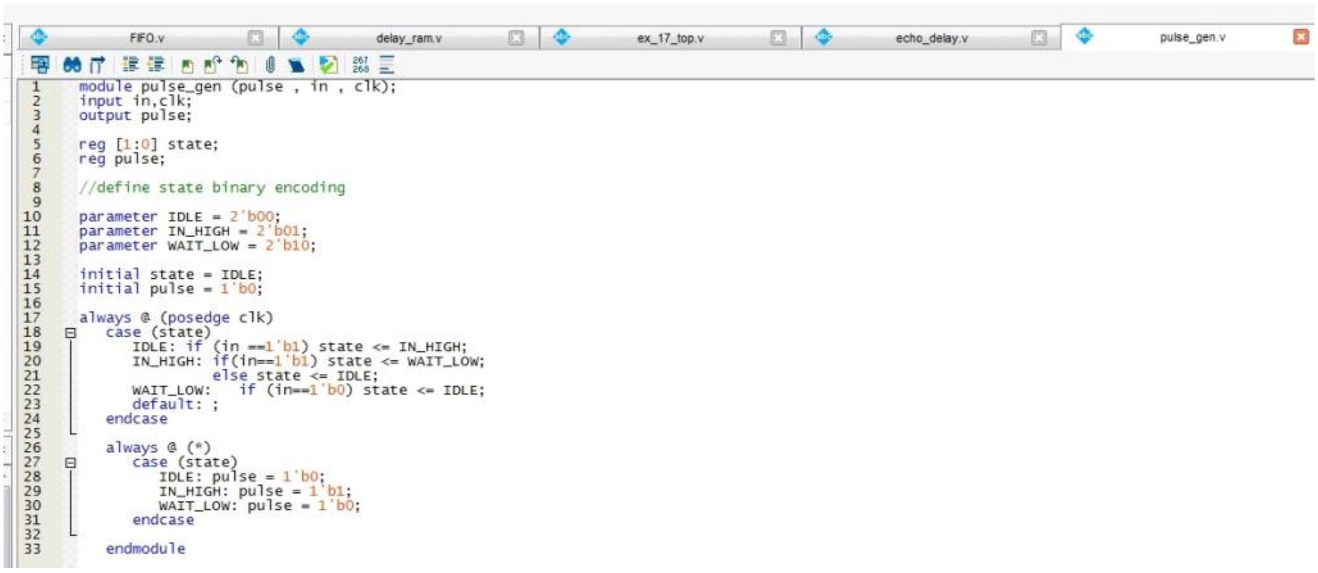
In order to implement the attenuation factor by 0.5, we can shift the bits to the right by one.



FIFO.v | delay_ram.v | ex_17_top.v | echo_delay.v

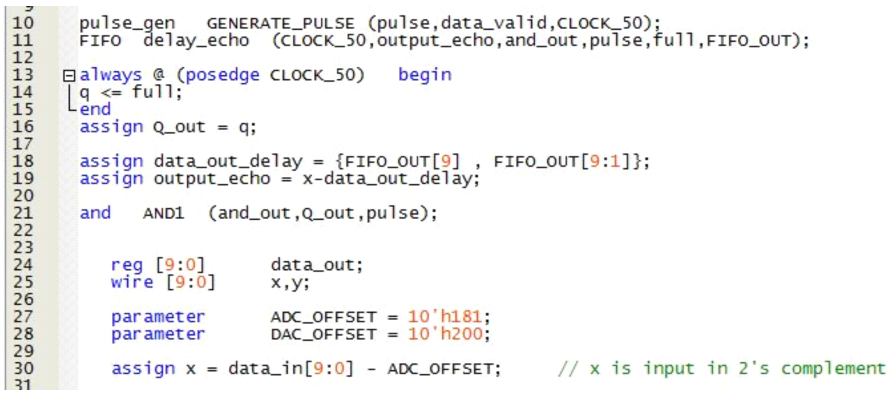
```verilog
module   processor   (data_in,data_valid,CLOCK_50, data_out);
input CLOCK_50,data_valid;
input [9:0] data_in;
wire pulse,full , and_out ;
wire [9:0] data_out_delay , FIFO_OUT, output_echo;
output   [9:0]    data_out;
wire Q_out;
reg q;

pulse_gen   GENERATE_PULSE (pulse,data_valid,CLOCK_50);
FIFO  delay_echo  (CLOCK_50,x,and_out,pulse,full,FIFO_OUT);

always @ (posedge CLOCK_50)   begin
q <= full;
end
assign Q_out = q;

assign data_out_delay = {FIFO_OUT[9] , FIFO_OUT[9:1]};
assign output_echo = data_out_delay + x;

and   AND1  (and_out,Q_out,pulse);


   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.

   //  Now clock y output with system clock
   always @(posedge CLOCK_50)
      data_out <=  output_echo + DAC_OFFSET;

endmodule
```

We use the FIFO to implement the constant delay, which we find from the IP Catalogue. We fill up the FIFO first using the write request and when it is full, we starting reading the elements from the FIFO as a stream of binary data upon every active edge of the clock.



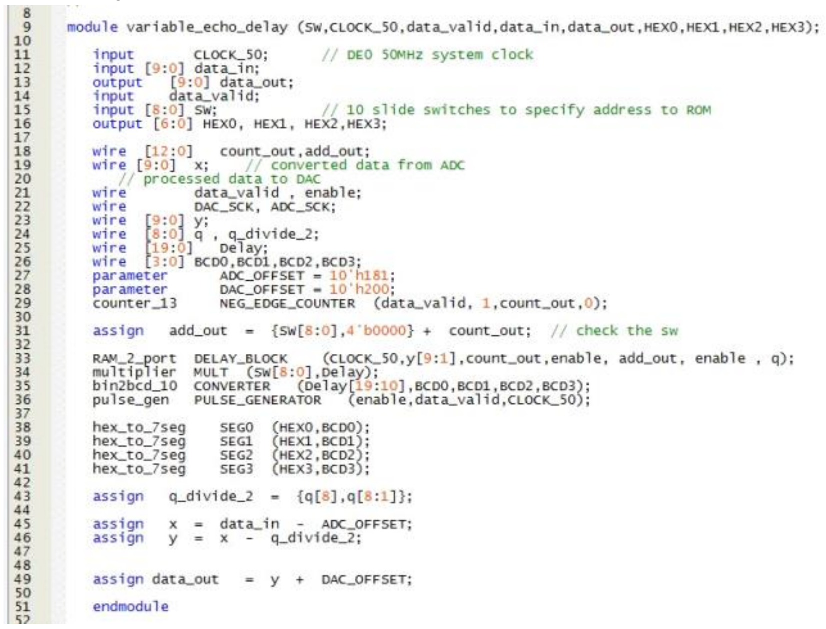
```verilog
module pulse_gen (pulse , in , clk);
input in,clk;
output pulse;

reg [1:0] state;
reg pulse;

//define state binary encoding

parameter IDLE = 2'b00;
parameter IN_HIGH = 2'b01;
parameter WAIT_LOW = 2'b10;

initial state = IDLE;
initial pulse = 1'b0;

always @ (posedge clk)
   case (state)
      IDLE: if (in ==1'b1) state <= IN_HIGH;
      IN_HIGH: if(in==1'b1) state <= WAIT_LOW;
               else state <= IDLE;
      WAIT_LOW:   if (in==1'b0) state <= IDLE;
      default: ;
   endcase

   always @ (*)
      case (state)
         IDLE: pulse = 1'b0;
         IN_HIGH: pulse = 1'b1;
         WAIT_LOW: pulse = 1'b0;
      endcase

   endmodule
```

The data valid signal is applied as input to the pulse gen module, which gives a pulse lasting for one cycle once it has counted up.
In order to implement the attenuation factor by 0.5, we can shift the bits to the right by one. No adder block module is needed to implement the addition.



```verilog
pulse_gen   GENERATE_PULSE (pulse,data_valid,CLOCK_50);
FIFO  delay_echo   (CLOCK_50,output_echo,and_out,pulse,full,FIFO_OUT);

always @ (posedge CLOCK_50)    begin
q <= full;
end
assign Q_out = q;

assign data_out_delay = {FIFO_OUT[9] , FIFO_OUT[9:1]};
assign output_echo = x-data_out_delay;

and   AND1  (and_out,Q_out,pulse);


   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
```

EXPERIMENT 19



```verilog
module variable_echo_delay (SW,CLOCK_50,data_valid,data_in,data_out,HEX0,HEX1,HEX2,HEX3);

   input       CLOCK_50;      // DEO 50MHz system clock
   input [9:0] data_in;
   output   [9:0] data_out;
   input    data_valid;
   input [8:0] SW;            // 10 slide switches to specify address to ROM
   output [6:0] HEX0, HEX1, HEX2,HEX3;

   wire  [12:0]   count_out,add_out;
   wire [9:0]  x;     // converted data from ADC
      // processed data to DAC
   wire        data_valid , enable;
   wire        DAC_SCK, ADC_SCK;
   wire  [9:0] y;
   wire  [8:0] q , q_divide_2;
   wire  [19:0]   Delay;
   wire  [3:0] BCD0,BCD1,BCD2,BCD3;
   parameter      ADC_OFFSET = 10'h181;
   parameter      DAC_OFFSET = 10'h200;
   counter_13     NEG_EDGE_COUNTER  (data_valid, 1,count_out,0);

   assign   add_out  =  {SW[8:0],4'b0000} +  count_out;  // check the sw

   RAM_2_port  DELAY_BLOCK    (CLOCK_50,y[9:1],count_out,enable, add_out, enable , q);
   multiplier  MULT  (SW[8:0],Delay);
   bin2bcd_10  CONVERTER   (Delay[19:10],BCD0,BCD1,BCD2,BCD3);
   pulse_gen   PULSE_GENERATOR   (enable,data_valid,CLOCK_50);

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

   assign   q_divide_2  =  {q[8],q[8:1]};

   assign   x  =  data_in  -  ADC_OFFSET;
   assign   y  =  x  -  q_divide_2;


   assign data_out   =  y  +  DAC_OFFSET;

   endmodule
```

```verilog
  variable_echo_delay  PROCESSOR (SW,CLOCK_50,data_valid,data_in,data_out,HEX0,HEX1,HEX2,HEX3);   // do some processing on the data
```

All of the other code stayed the same. We used channel one for the analogue input. Only the signal processing is changed. ex