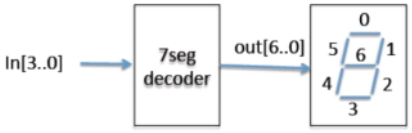
# Experiment 1

17 November 2017 09:38

FPGA (Field Programmable Gate Array) is a programmable logic device that allows one to implement a wide range of digital circuits. An FPGA consists of:

- Logic Blocks
- Routing Resources
- I/O Pad



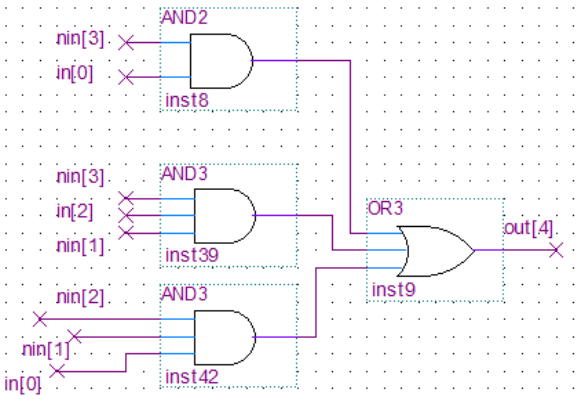

Finding the simplified Boolean for Out4

O4= /I3*/i2*/I1/I0+/I3*/I2**I1*I0+/I3*I2*/I1*/I1+/I3*I2*/I1*I0+/I3*I2*I1*I0+I3*/I2*/I1*I0

K-Map

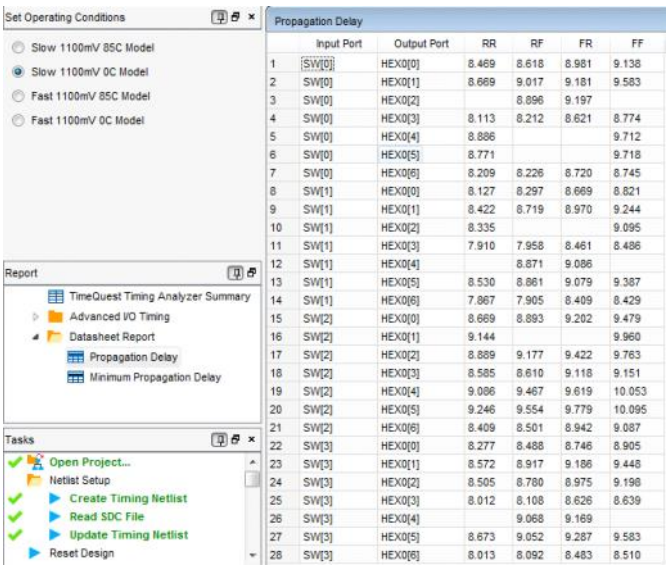
| I3I2\I1I0 | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | 1 | 1 | |
| 01 | 1 | 1 | 1 | |
| 11 | | | | |
| 10 | | 1 | | |

This gives a simplified version O4=/I3*I0+/I3*I2*/I1+/I2*/I1*I0




This is a schematic for Out4 for the seven seg display

0°C




| | 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 |

Some values are missing in the table because those combinations never occur in our design.

RR- rise rise
RF -rise fall
FR - fall rise
FF - fall fall

80°C

Set Operating Conditions

- Slow 1100mV 85C Model
- Slow 1100mV 0C Model
- Fast 1100mV 85C Model
- Fast 1100mV 0C Model

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
  - Update Timing Netlist
- Reset Design

Propagation Delay

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

This table shows input to output propagation delay. For example RF corresponds to a rise in the input causing a fall in the output. The higher the temperature the longer the propagation delay. This is because at higher temperatures there is more resistance.

Some combinations are missing because they never occur in our circuit design. For example the rise of sw[0] (the switch going high) never causes the 2nd bar to go high on the 7seg display.

Flow Summary

| | |
|---|---|
| Flow Status | Successful - Fri Nov 17 10:15:50 2017 |
| Quartus Prime Version | 16.0.0 Build 211 04/27/2016 SJ Standard Edition |
| Revision Name | ex1_top |
| Top-level Entity Name | ex1_top |
| Family | Cyclone V |
| Device | 5CSEMA5F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 4 / 32,070 ( < 1 % ) |
| Total registers | 0 |
| Total pins | 11 / 457 ( 2 % ) |
| Total virtual pins | 0 |
| Total block memory bits | 0 / 4,065,280 ( 0 % ) |
| Total 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 % ) |

Compilation Report for 7 seg display. This shows that in our design we have only used 4/32,070 ALMs.

# Exppeiment 2

Friday, November 17, 2017 10:54 AM

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

The pin assignments can be done by editing the qsf file in the following format. The top line specifies the voltage standard and the second the physical pin location

```
1   module hex_to_7seg (out,in);
2
3      output [6:0] out; //low-active ouput
4      input[3:0] in; //4-bit binary input
5
6      reg [6:0] out; // make out a variable
7
8      always @ (*)
9         case (in)
10           4'h0: out = 7'b1000000;
11           4'h1: out = 7'b1111001;
12           4'h2: out = 7'b0100100;
13           4'h3: out = 7'b0110000;
14           4'h4: out = 7'b0011001;
15           4'h5: out = 7'b0010010;
16           4'h6: out = 7'b0000010;
17           4'h7: out = 7'b1111000;
18           4'h8: out = 7'b0000000;
19           4'h9: out = 7'b0011000;
20           4'ha: out = 7'b0001000;
21           4'hb: out = 7'b0000011;
22           4'hc: out = 7'b1000110;
23           4'hd: out = 7'b0100001;
24           4'he: out = 7'b0000110;
25           4'hf: out = 7'b0001110;
26        endcase
27  endmodule
```

The code for the 7 seg. Originally got two errors because we forgot semicolon after reg[6:0] out and input[3;0]in

Originally had errors because we forgot to put semicolons at the end of line 6 and 4.

```
1   module ex2_top (
2      SW,   //input switches
3      HEX0  // Hex output on 7 segment display
4      );
5      input [3:0] SW;   // declare input/output ports
6      output[6:0] HEX0;
7
8      hex_to_7seg SEG0 (HEX0, SW);
9
10  endmodule
```

This is the top-level specification for the 7 seg. Note: SEG0 just specifies the instance of the module. If it wasn't there Quartus would randomly assign it one.

# Experiment 3

Friday, November 17, 2017 11:30 AM

```
module ex2_top (
    SW,   //input switches
    HEX0, HEX1, HEX2  // Hex output on 7 segment display
    );
    input [9:0] SW;   // declare input/output ports
    output[6:0] HEX0;
    output[6:0] HEX1;
    output[6:0] HEX2;

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

endmodule
```

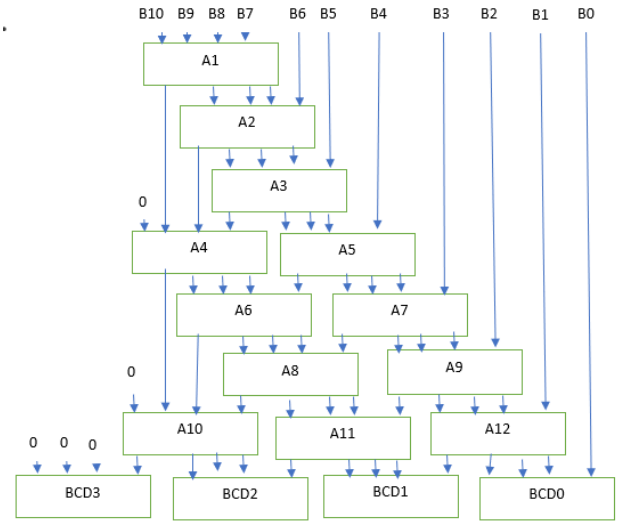
To make the 10 bit to 7 seg display we had to add two more hex outputs. Since the first 4 bits correspond to the first digit of hex and bits 4 to 7 bit correspond to the second digit of hex. The last two coresspond to the last digit of hex. The two MSBs will automatically be set to zero.

Had to change the names of the instances.

In the file ex3_top.qsf it has the pin assignment for HEX1 and HEX2 already defined .

# Experiment 4

Friday, November 17, 2017 11:45 AM




The diagram for the 10 bit binary to binary coded decimal. The blocks show where the add 3 happens. And the arrows show the sifting

```
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]};       // w_n is the input port to module a_n
   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
```

```
module ex4_top (
   SW,   //input switches
   HEX0, HEX1, HEX2  // Hex output on 7 segment display
   );
   input [9:0] SW;   // declare input/output ports
   wire [3:0] BCD0;
   wire [3:0] BCD1;
   wire [3:0] BCD2;
   wire [3:0] BCD3;
   output[6:0] HEX0;
   output[6:0] HEX1;
   output[6:0] HEX2;

   bin2bcd_10 (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
```

# Experiment 5

17 November 2017 10:43

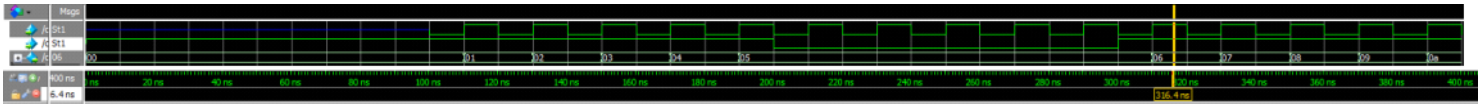
Any circuit in Verilog can be tested using the built in ModelSim program. We created an 8 bit counter using the code below

```
`timescale 1ns/100ps // unit time is 1ns, resolutoin 100ps
//Design Name: Counter_8
//Function : an 8 bit counter synchronous counterwith 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 need 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
```

We tested the counter with model sim by inputting the following commands into the transcript pane

```
VSIM 4> add wave clock enable
VSIM 5> add wave -hexadecimal
# Missing signal name or pattern.
VSIM 6> add wave -hexadecimal count
VSIM 7> force enable 1
VSIM 8> run 100ns
VSIM 9> force clock 0 0, 1 10ns -repeat 20ns
VSIM 10> force enable 1
VSIM 11> run 100ns
VSIM 12> force enable 0
VSIM 13> run 100ns
VSIM 14> force enable 1
VSIM 15> run 100ns

VSIM 16> force enable 0
```

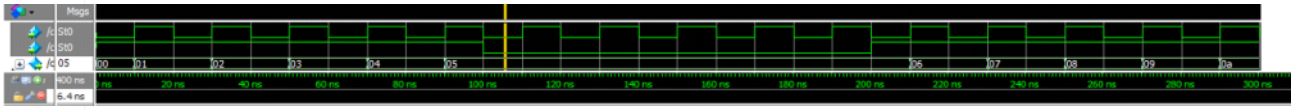
Made error in command (run for 100ns without specifying the clock) that is why it is constantly low for first 100ns



Instead of typing out the commands into the transcript pane every time the simulation needs to be run a testbench file can be created which can be run with a single command. The contents are as follow:

C:/part_2/simulation/modelsim/tb_counter.do - Default

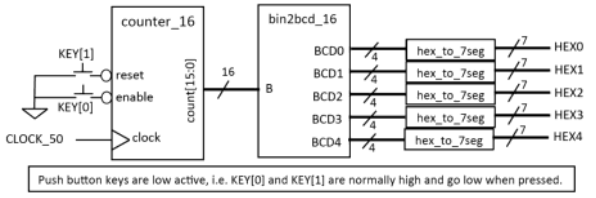
```
add wave clock enable
add wave -hexadecimal count
force clock 0 0, 1 10ns -repeat 20ns
force enable 1
run 100ns
force enable 0
run 100ns
force enable 1
run 100ns
```



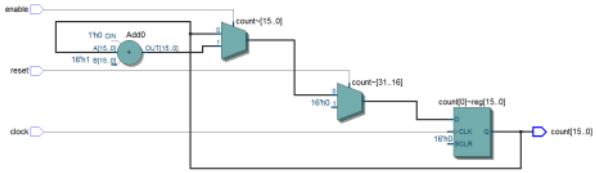
# Experiment 6

Friday, November 24, 2017 11:35 AM

| | |
|---|---|
| The counter is now changed to 6 bits, has a reset and an enable. To testit we connected it to the binary to bcd module, which is thenx`x | connected to 5 hex digit displays. |




The enable and reset can be implemented using multiplexers




```
`timescale 1ns/100ps // unit time is 1ns, resolutoin 100ps
//Design Name: Counter_8
//Function : an 8 bit counter synchronous counterwith enable input


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

 // Declare ports

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

 //count need 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) begin
    if(enable == 1'b1)
    count <= count +1'b1;
    if(reset == 1)
    count <= 0;
end
 endmodule
```

To make Quartus aware of the clock frequency the clock is mapped to we have to specify the clock in the file ex6_top.sdc using the following line:

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

.sdc files are Synopsis Delay Constraint files which are used by CAD tools such as the TimeQuest Timing Analyzer

**Slow 1100mV 85C Model Fmax Summary**

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

The maximum predicted frequency for our circuit is 444 MHz at 85 degrees Celsius and 422 MHz at 0 degrees Celsius. Although we would expect an increase in temperature to decrease the maximum frequency – it is the opposite. Because the circuit is relatively small the increase in resistance that causes routing delays is compensated by the decrease in the delay of the transistors.

**Slow 1100mV 0C Model Fmax Summary**

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

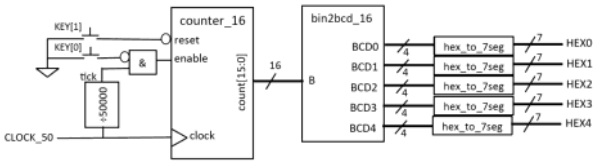
**Input Transition Times**

| | Pin | I/O Standard | 10-90 Rise Time | 90-10 Fall Time |
|---|---|---|---|---|
| 1 | CLOCK_50 | 3.3-V LVTTL | 2640 ps | 2640 ps |
| 2 | KEY[1] | 3.3-V LVTTL | 2640 ps | 2640 ps |
| 3 | KEY[0] | 3.3-V LVTTL | 2640 ps | 2640 ps |

Other timing data such as setup, hold times or minimum pulse width summary are included in the timing analyzer

| Analysis & Synthesis Resource Usage Summary | | |
|---|---|---|
| | Resource | Usage |
| 1 | Estimate of Logic utilization (ALMs needed) | 98 |
| 2 | | |
| 3 | Combinational ALUT usage for logic | 182 |
| 1 | -- 7 input functions | 0 |
| 2 | -- 6 input functions | 14 |
| 3 | -- 5 input functions | 1 |
| 4 | -- 4 input functions | 145 |
| 5 | -- <=3 input functions | 22 |
| 4 | | |
| 5 | Dedicated logic registers | 16 |
| 6 | | |
| 7 | I/O pins | 38 |
| 8 | | |
| 9 | Total DSP Blocks | 0 |
| 10 | | |
| 11 | Maximum fan-out node | KEY[1]~input |
| 12 | Maximum fan-out | 17 |
| 13 | Total fan-out | 853 |
| 14 | Average fan-out | 3.11 |

Test Yourself:
The counter is now enabled by a tick pulse provided by the clktick module




Initially instead of passing a tick every 50,000 cycles of the clock into the enable, we passed a clock with a frequency divided by 50,000. That didn't work because the counter still had the same clock and enable was on 50% of the time. By doing so we have essentially dividedthe frequency of the count by 2 and not by 50 000 as intended.

We overcame the problem by writing a new tick module, which generate a tick for 1 clock cycle every 50 000 cycles of the initial clock. As this tick was ANDed with key[0] and passed into the counter it allowed the count to be increased only once every 50 000 cycles.

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

 parameter N_BIT = 16;
 //input ports
 input clkin;
 input enable;
 input [N_BIT-1:0] N;

 //output ports
 output tick;

 //output prts data types
 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
```

```
module ex6_top (
    CLOCK_50,
    KEY,
    HEX0, HEX1, HEX2, HEX3, HEX4
);
    input [1:0]KEY;
    input CLOCK_50;
    output [6:0]HEX0;
    output [6:0]HEX1;
    output [6:0]HEX2;
    output [6:0]HEX3;
    output [6:0]HEX4;
    wire [15:0] WIRE1;
    wire [3:0] WIRE2;
    wire [3:0] WIRE3;
    wire [3:0] WIRE4;
    wire [3:0] WIRE5;
    wire [3:0] WIRE6;
    wire TICK;

    reg delay;
    reg clocker;
    reg enable;


    clktick CLKTICK0(CLOCK_50, 1, 16'd50000, TICK);
    counter_16 COUNT0(CLOCK_50, TICK & ~KEY[0], WIRE1, ~KEY[1]);
    bin2bcd_16 BIN(WIRE1, WIRE2, WIRE3, WIRE4, WIRE5, WIRE6);
    hex_to_7seg SEG0(HEX0, WIRE2);
    hex_to_7seg SEG1(HEX1, WIRE3);
    hex_to_7seg SEG2(HEX2, WIRE4);
    hex_to_7seg SEG3(HEX3, WIRE5);
    hex_to_7seg SEG4(HEX4, WIRE6);
endmodule
```

# Experiment 7

Wednesday, November 29, 2017 2:56 PM

We used linear feeback shift registers to implement a PRBS of the polynomial 1+X+X^7. This polynomial implies that the input into the first shift register is the 6th and 0th bit XORed.

```
module ex7_top(
    data_out,
    KEY,
    HEX0, HEX1
);

    output [6:0] data_out;
    input [3:0] KEY;
    output[6:0] HEX0;
    output[6:0] HEX1;

    reg[6:0] sreg;

    initial sreg = 7'b1;

    always @ (posedge KEY[3])
       sreg <= {sreg[5:0], sreg[6] ^ sreg[2]};

    assign data_out = sreg;

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


endmodule
```

Output:

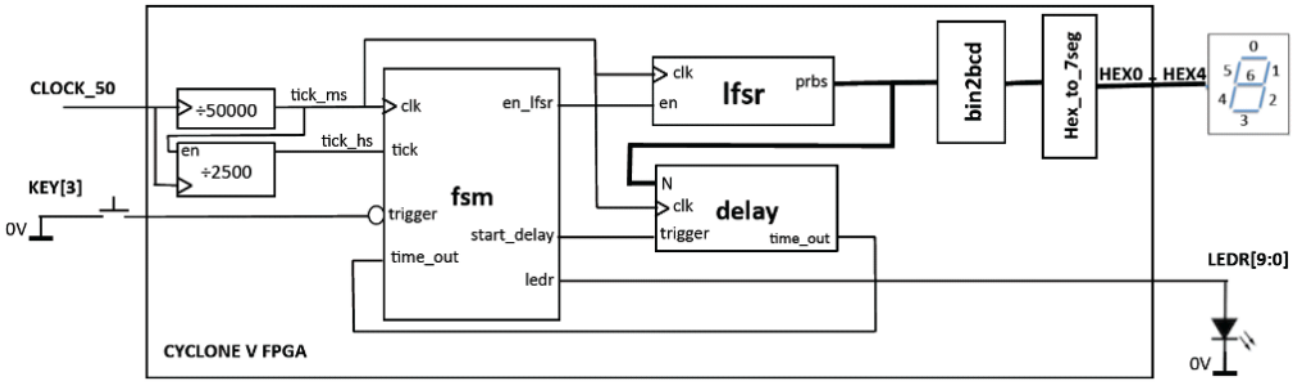
| Expected binary | Expected hex | Result hex |
|---|---|---|
| 1 | 1 | 1 |
| 11 | 3 | 3 |
| 111 | 7 | 7 |
| 1111 | 0F | 0F |
| 11111 | 1F | 1F |
| 111111 | 3F | 3F |
| 1111111 | 7F | 7F |
| 1111110 | 7E | 7E |
| 1111101 | 7D | 7D |
| 1111010 | 7A | 7A |
| 1110101 | 75 | 75 |
| 1101010 | 6A | 6A |
| 1010101 | 55 | 55 |

# Experiment 8&9

Thursday, December 14, 2017 8:55 PM

SPEC:

1. The circuit is triggered by pressing KEY[3]
2. The 10 LEDs will start turning on from left to right at 0.5 second interval until they are all ON
3. The circuit then waits for a random period of time between 0.25 and 16 seconds before turning all LEDs OFF
4. The random value generated by lfsr is to be shown on the 7-segment displays




The tick_ms and tick_hs were generating using the previously defined clktick module. There is a mistake in the specification, because tick_hs is meant to tick every 0.5s, to achieve that the 2Hz is required, which should be the division of the clock by 50,000 followed by 500 and not 2500 as specified.

```
module fsm(
clock,
tick,
trigger,
time_out,
en_lfsr,
start_delay,
ledr
);

input clock;
input tick;
input trigger;
input time_out;
output en_lfsr;
output start_delay;
output [9:0]ledr;

reg [3:0]state;
reg  start_delay;
reg  [9:0]ledr;
reg  en_lfsr;

parameter IDLE = 4'b1010;
parameter LED9 = 4'b1001;
parameter LED8 = 4'b1000;
parameter LED7 = 4'b0111;
parameter LED6 = 4'b0110;
parameter LED5 = 4'b0101;
parameter LED4 = 4'b0100;
parameter LED3 = 4'b0011;
parameter LED2 = 4'b0010;
parameter LED1 = 4'b0001;
parameter LED0 = 4'b0000;
parameter DELAY = 4'b1011;

initial state = IDLE;
initial en_lfsr = 1'b0;
initial start_delay = 0;
//initial ledr = 10'b0;
```

```
//states
always @ (posedge clock) begin
        case (state)
            IDLE : if(trigger==1'b1)
                   state <= LED9;

            LED9 : if(tick==1'b1)
                   state <= LED8;

            LED8 : if(tick==1'b1)
                   state <= LED7;

            LED7 : if(tick==1'b1)
                   state <= LED6;

            LED6 : if(tick==1'b1)
                   state <= LED5;

            LED5 : if(tick==1'b1)
                   state <= LED4;

            LED4 : if(tick==1'b1)
                   state <= LED3;

            LED3 : if(tick==1'b1)
                   state <= LED2;

            LED2 : if(tick==1'b1)
                   state <= LED1;

            LED1 : if(tick==1'b1)
                   state <= LED0;

            LED0 : if(tick==1'b1)
                   state <= DELAY;

            DELAY : if(time_out==1'b
                    state<=IDLE;
            default:  ; //do nothing


        endcase
end
```

```
//output
always @(*) begin
    case(state)
    IDLE: begin
          en_lfsr<=1'b1;
          start_delay<=1'b0;
          ledr<=10'b0;
          end
    LED9: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1000000000;
          end
    LED8 : begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1100000000;
          end
    LED7: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1110000000;
          end
    LED6: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1111000000;
          end
    LED5: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1111100000;
          end
    LED4: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1111110000;
          end
    LED3: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1111111000;
          end
    LED2: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1111111100;
          end
    LED1: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1111111110;
          end
    LED0: begin
          en_lfsr<=1'b0;
          start_delay<=1'b0;
          ledr<=10'b1111111111;
          end
```

```
        LED0: begin
              en_lfsr<=1'b0;
              start_delay<=1'b0;
              ledr<=10'b1111111111;
              end
        DELAY: begin
              en_lfsr<=1'b0;
              start_delay<=1'b1;
              ledr<=10'b1111111111;
              end
        default: ;
      endcase
    end

  endmodule
```

The FSM has a few signals to control. First of all to comply with the first part of the spec it needs to take in a trigger signal and if that is high chage from the first IDLE state to counting the LEDs. Once that is done, it goes in order through states LED9-1 ensuring a transition happens only at tick (0.5s apart). Once it goes to the LED state it goes into a DELAY state. Once the delay is finished it goes back to the IDLE state.

The 3rd block of code defines the changes that need to happen when a change of state occurs. Firsly, en_fslr should only go high if the FSM is in the IDLE state, secondly, the start_delay should be high only when the FSM is in the delay state. Lastly, each LED state should make the correct LEDs light up.

The lfsr14 (Linear Feedback Shift Register) module generates a pseudo random 14 bit number between 250 and 16000 corresponding to 0.25s and 16s. To do this we force any number less than 250 to 250 and larger than 16000 increasing the probability of these numbers to occur but comlplying with the spec.

```
module lfsr14(
    enable, clock, data_out
);

    input clock;
    input enable;
    output [13:0] data_out;

    reg[13:0] sreg;

    initial sreg = 14'b1;

    always @ (posedge clock)
       begin
          if(enable==1)
             sreg <= {sreg[13:0],sreg[0]^sreg[7]^sreg[9]^sreg[13]};
          if(sreg<16'd250)
             sreg <= 16'd250;
          if(sreg>16'd16000)
             sreg <=16'd16000;
    end
    assign data_out = sreg;


endmodule
```

The delay module is resposible for carrying out the delay only when the trigger (start_delay) signal is sent from the FSM. When the trigger pulse comes in the delay goes into a 'lock' mode, which sets the count to the number from the lfsr and starts counting down. It sets time_out to 1 when it is done and unlocks itself. When the FSM recieves the time_out signal it turns all LEDs off.

```
module delay(
N,
clock,
trigger,
time_out
);

input [13:0]N;
input clock;
input trigger;
output time_out;

reg time_out;
reg lock;
reg [13:0]count;

initial time_out = 0;
initial lock = 0;
initial count = 14'b0;

always @ (posedge clock) begin
   time_out<=0;
   if(trigger == 1'b1 && lock == 0)begin
      lock<=1;
      count<=N;
      end
   if (lock == 1)
      count<=count-1;
   if (count == 0 && lock == 1)begin
      lock<=0;
      time_out<=1;
      end
end

endmodule
```

Experiment 9 requires us to measure the reactions time once all of the light have turned off. The following fsm is implemented

```
module reactions(clock, time_out, react, key);
   input clock, time_out, key;
   output[13:0] react;
   reg[1:0] state;
   initial state <= 0;
   reg[15:0] count;
   initial count = 0;

   always @ (posedge clock)
      case (state)
      2'b00: if (time_out == 1) begin state <= 2'b01;
      count <=0; end
      2'b01: if (key == 1) state <= 2'b10;
      else count <= count + 1;
      2'b10: if (time_out == 0) state <= 2'b00;
      endcase

      assign react = count;
endmodule
```

The whole project is linked together in the following manner:

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

    input CLOCK_50;
    input [3:0]KEY;

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

    wire tick_ms;
    wire tick_hs;
    wire en_lfsr;
    wire time_out;
    wire start_delay;
    wire [3:0] BCD_0, BCD_1, BCD_2, BCD_3, BCD_4;
    wire [15:0]react;
    wire [13:0]prbs;

    clktick div2500(CLOCK_50,tick_ms,12'd500, tick_hs);
    clktick  div50000(CLOCK_50,1,16'd50000,tick_ms);
    fsm FSM0(tick_ms,tick_hs,~KEY[3],time_out,en_lfsr,start_delay,LEDR[9:0]);
    lfsr14 LFSR14(en_lfsr, tick_ms, prbs[13:0],);
    delay DELAY0(prbs[13:0],tick_ms,start_delay,time_out);
    reactions REACTION0(CLOCK_50, time_out, react, ~KEY[0]);
    bin2bcd_16 B2BCD0(react,BCD_0, BCD_1, BCD_2, BCD_3, BCD_4);

    hex_to_7seg HEX_0(HEX0,BCD_0);
    hex_to_7seg HEX_1(HEX1,BCD_1);
    hex_to_7seg HEX_2(HEX2,BCD_2);
    hex_to_7seg HEX_3(HEX3,BCD_3);
    hex_to_7seg HEX_4(HEX4,BCD_4);

endmodule
```

# Experiment 10

Friday, December 1, 2017 9:03 AM

**TABLE 3-1: PIN FUNCTION TABLE**

| PDIP, MSOP, SOIC | DFN | Symbol | Description |
|---|---|---|---|
| 1 | 1 | $V_{DD}$ | Supply Voltage Input (2.7V to 5.5V) |
| 2 | 2 | $\overline{CS}$ | Chip Select Input |
| 3 | 3 | SCK | Serial Clock Input |
| 4 | 4 | SDI | Serial Data Input |
| 5 | 5 | $\overline{LDAC}$ | DAC Output Synchronization Input. This pin is used to transfer the input register (DAC settings) to the output register ($V_{OUT}$) |
| 6 | 6 | $V_{REF}$ | Voltage Reference Input |
| 7 | 7 | $V_{SS}$ | Ground reference point for all circuitry on the device |
| 8 | 8 | $V_{OUT}$ | DAC Analog Output |
| — | 9 | EP | Exposed Thermal Pad. This pad must be connected to $V_{SS}$ in application |

The communication with the DAC is unidirectional, so only write commands are allowed. The CS pin has to be low for the duration of the write. The first 4 bits loaded are configuration bits and the next 12 are data. Any bits after that are ignored.

**REGISTER 5-2: WRITE COMMAND REGISTER FOR MCP4911 (10-BIT DAC)**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| W-x | W-x | W-x | W-0 | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x |
| 0 | BUF | $\overline{GA}$ | $\overline{SHDN}$ | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | x | x |
| bit 15 | | | | | | | | | | | | | | | bit 0 |

**REGISTER 5-3: WRITE COMMAND REGISTER FOR MCP4901 (8-BIT DAC)**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| W-x | W-x | W-x | W-0 | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x |
| 0 | BUF | $\overline{GA}$ | $\overline{SHDN}$ | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | x | x | x | x |
| bit 15 | | | | | | | | | | | | | | | bit 0 |

Where:

bit 15 0 = Write to DAC register
1 = Ignore this command

bit 14 **BUF:** $V_{REF}$ Input Buffer Control bit
1 = Buffered
0 = Unbuffered

bit 13 **$\overline{GA}$:** Output Gain Selection bit
1 = 1x ($V_{OUT} = V_{REF} * D/4096$)
0 = 2x ($V_{OUT} = 2 * V_{REF} * D/4096$)

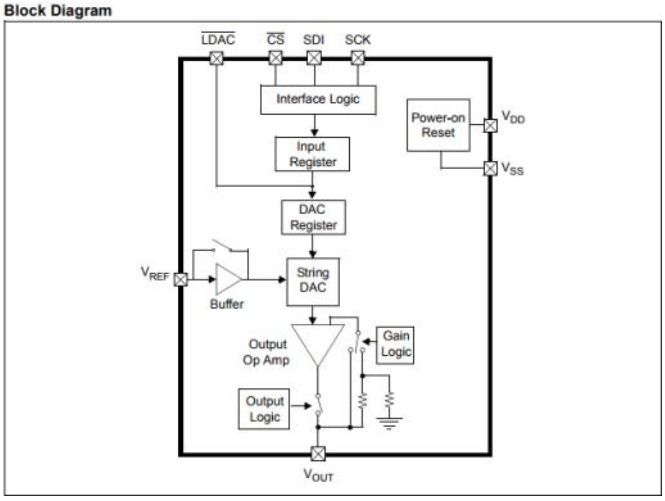
bit 12 **$\overline{SHDN}$:** Output Shutdown Control bit
1 = Active mode operation. VOUT is available.
0 = Shutdown the device. Analog output is not available. VOUT pin is connected to 500 kΩ (typical).

bit 11-0 **D11:D0:** DAC Input Data bits. Bit x is ignored.

| Legend | | | |
|---|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as '0' | |
| -n = Value at POR | 1 = bit is set | 0 = bit is cleared | x = bit is unknown |

When LDAC is low the contents of the input registers are tranferred into the DAC register. The two are seperated, because the contents of the DAC register are converted asynchrounsly and the result would glitch when the digital value is loaded

**Block Diagram**




Since our DAC is 10 bit the last 2 bits in the sequence aren't loaded

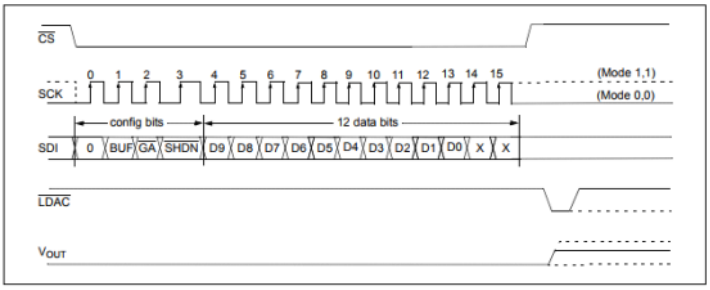



*FIGURE 5-2: Write Command for MCP4911 (10-bit DAC). Note: X are don't care bits.*

**AC CHARACTERISTICS (SPI TIMING SPECIFICATIONS)**

**Electrical Specifications:** Unless otherwise indicated, $V_{DD}$= 2.7V – 5.5V, $T_A$= -40 to +125°C. Typical values are at +25°C.

| Parameters | Sym | Min | Typ | Max | Units | Conditions |
|---|---|---|---|---|---|---|
| Schmitt Trigger High Level Input Voltage (All digital input pins) | $V_{IH}$ | 0.7 $V_{DD}$ | — | — | V | |
| Schmitt Trigger Low Level Input Voltage (All digital input pins) | $V_{IL}$ | — | — | 0.2 $V_{DD}$ | V | |
| Hysteresis of Schmitt Trigger Inputs | $V_{HYS}$ | — | 0.05 $V_{DD}$ | — | | |
| Input Leakage Current | $I_{LEAKAGE}$ | -1 | — | 1 | µA | $\overline{LDAC}$ = $\overline{CS}$ = SDI = SCK = $V_{REF}$ = $V_{DD}$ or $V_{SS}$ |
| Digital Pin Capacitance (All inputs/outputs) | $C_{IN}$, $C_{OUT}$ | — | 10 | — | pF | $V_{DD}$ = 5.0V, $T_A$ = +25°C, $f_{CLK}$ = 1 MHz **(Note 1)** |
| Clock Frequency | $F_{CLK}$ | — | — | 20 | MHz | $T_A$ = +25°C **(Note 1)** |
| Clock High Time | $t_{HI}$ | 15 | — | — | ns | **Note 1** |
| Clock Low Time | $t_{LO}$ | 15 | — | — | ns | **Note 1** |
| $\overline{CS}$ Fall to First Rising CLK Edge | $t_{CSSR}$ | 40 | — | — | ns | Applies only when $\overline{CS}$ falls with CLK high **(Note 1)** |
| Data Input Setup Time | $t_{SU}$ | 15 | — | — | ns | **Note 1** |
| Data Input Hold Time | $t_{HD}$ | 10 | — | — | ns | **Note 1** |
| SCK Rise to $\overline{CS}$ Rise Hold Time | $t_{CHS}$ | 15 | — | — | ns | **Note 1** |
| $\overline{CS}$ High Time | $t_{CSH}$ | 15 | — | — | ns | **Note 1** |
| $\overline{LDAC}$ Pulse Width | $t_{LD}$ | 100 | — | — | ns | **Note 1** |
| $\overline{LDAC}$ Setup Time | $t_{LS}$ | 40 | — | — | ns | **Note 1** |
| SCK Idle Time before $\overline{CS}$ Fall | $t_{IDLE}$ | 40 | — | — | ns | **Note 1** |

**Note 1:** This parameter is ensured by design and not 100% tested.

0x23b = 10 0011 1011

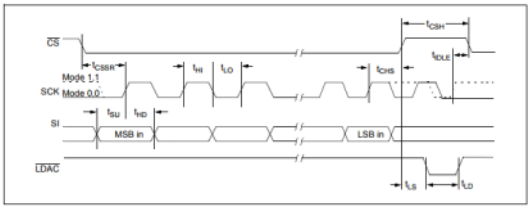
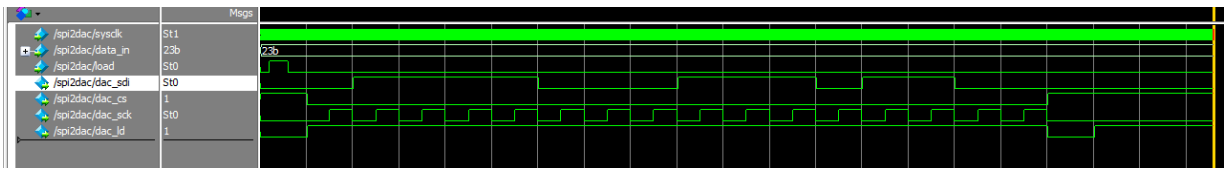


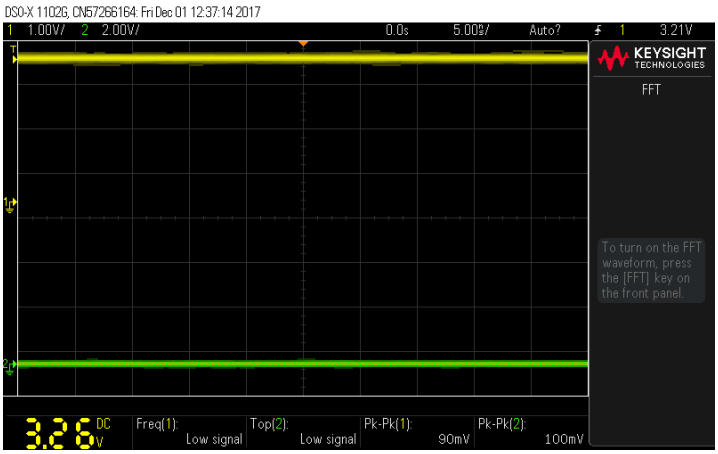

*FIGURE 1-1: SPI Input Timing Data.*

```
module ex10_top (
    CLOCK_50,
    SW,
    DAC_SDI, DAC_CS, DAC_SCK, DAC_LD
);
    input [9:0]SW;
    input CLOCK_50;
    wire TICK;
    output DAC_SDI;        // SPI serial data out
    output DAC_CS;         // chip select - low when sending data to dac
    output DAC_SCK;        // SPI clock, 16 cycles at half sysclk freq
    output DAC_LD;

    clktick CLKTICK0(CLOCK_50, 1, 16'd5000, TICK);
    spi2dac SPI2DAC0(CLOCK_50, SW[9:0], TICK, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);
endmodule
```



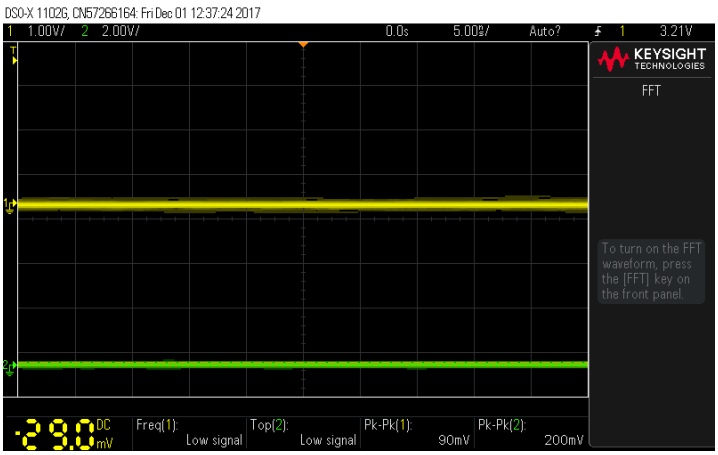

The spi2dac module produces exactly the same waveform as predicted above

```
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


```



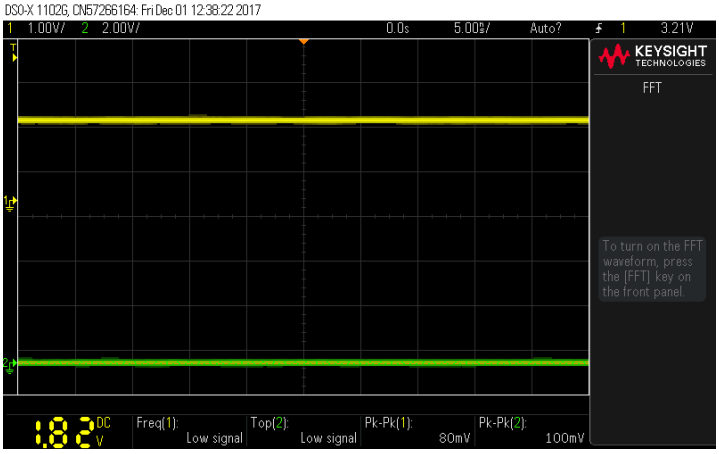

The maximum voltage measured is 3.26V

- The expected one is 3.3V

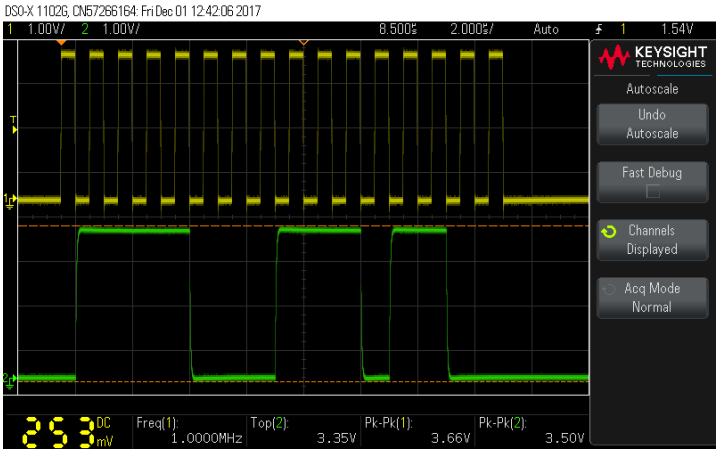



The minimum voltage measured is -29 mV

- The expected one 0V




When the value 0x23b is given in digital one would expect the output to be:
0x23b/0x3FF * 3.3 = 1.84V




The wave above shows the DACK_SCK (TP3) waveform, which is the clock driving the desing. The waveform below is DAC_SDI (TP1), which is theserial input into the DAC. Here the value 0x23b is loaded and the result is the same as predicted!

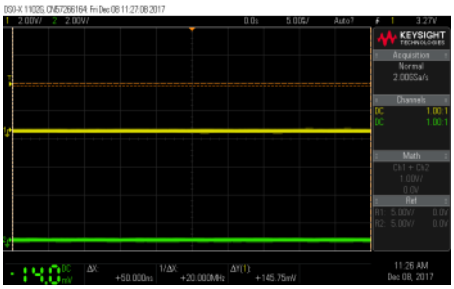
TP1 - DAC_SDI
TP2 - DAC_CS
TP3 - DAC_SCK
TP4 - DAC_LD
TP5 - pwm output

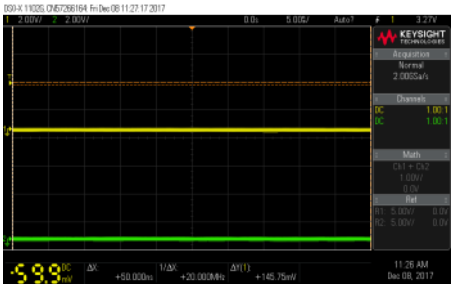
TP6
TP7
TP8 - DAC output (Analogue right)
TP9 - low pass filtered pwm output (Analogue left)

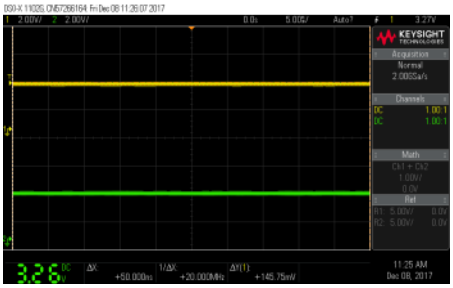
# Experiment 11

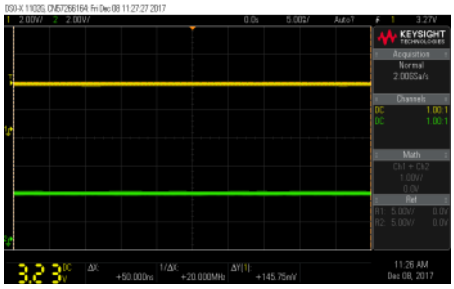
Friday, December 1, 2017 10:28 AM

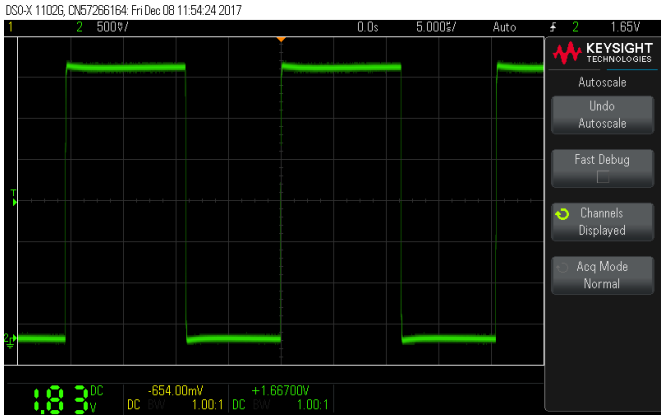
The green measurment corresponds to the pwm output after a lowpass filter is applied. The yellow is the dac output. As can be seen the range of both ways to convert analogue to digital is very simmilar.



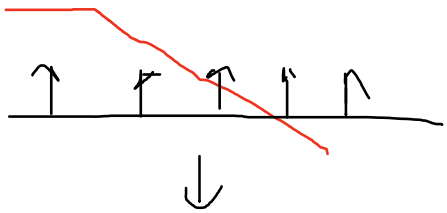




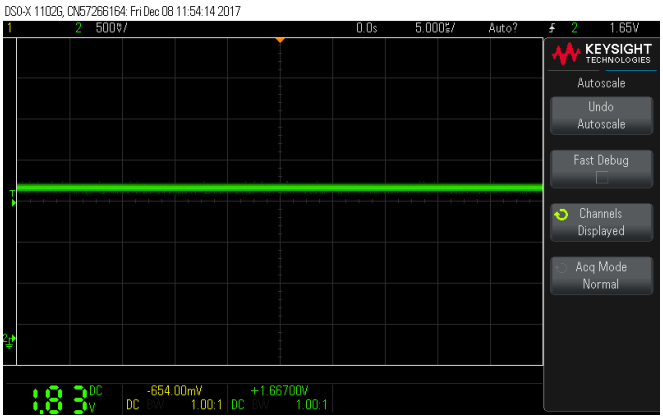





The output of the pwm is a square waveform. By increasing the digital input the duty cycle of the waveform also increases.



The filtered periodic delta function in the frequency domain results in a single delta function (dc component)




A lowpass filter is also an averaging circuit - therefore by varying the duty cycle of the wave the average changes. A higher digital value means a higher analouge one.

A low pass filter is an avergin circuit, because taking out the high frequency compenent leaves only dc.

```
module ex11_top (
    CLOCK_50,
    SW,
    DAC_SDI, DAC_CS, DAC_SCK, DAC_LD, PWM_OUT
);
    input [9:0]SW;
    input CLOCK_50;
    wire TICK;
    output DAC_SDI;       // SPI serial data out
    output DAC_CS;        // chip select - low when sending data to dac
    output DAC_SCK;       // SPI clock, 16 cycles at half sysclk freq
    output DAC_LD;
    output PWM_OUT;


    clktick CLKTICK0(CLOCK_50, 1, 16'd5000, TICK);
    pwm PWM0(CLOCK_50, SW, TICK, PWM_OUT);
    spi2dac SPI2DAC0(CLOCK_50, SW[9:0], TICK, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);

endmodule
```

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

input clk;
input [9:0] data_in;
input load;
output pwm_out;

reg [9:0] d;
reg [9:0] count;
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

endmodule
```

# Experiment 12 & 13

Friday, December 1, 2017 1:14 PM

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

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

    ROM rom0 (SW[9:0], CLOCK_50, data);

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

endmodule
```

The ROM holds exactly one peroid of a sine wave. The values were max and min at 1/4 and 3/4 way mark respectively. Halfway the value returned to the original 512. By setting the offset to 512 both the negative and positive sides of the sinewave could be represented with 3 hex digits (10 bits).

# Experiment 13

Friday, December 1, 2017 2:05 PM

In this exersice we have connected a counter to the ROM instead of controlling it with switches. This caused a sine wave to be created. The frequency of the wave we have measured is 9.77Hz. Unfortunetaly the screenshot of the oscoliscope was deleted by mistake.

```
module ex13_top(
    SW, CLOCK_50,
    HEX0, HEX1, HEX2,
    DAC_SDI, DAC_CS, DAC_SCK, DAC_LD, PWM_OUT
);

    input [9:0]SW;
    input CLOCK_50;
    wire clock;
    wire [9:0]data;
    wire [9:0] count;
    output [6:0] HEX0;
    output [6:0] HEX1;
    output [6:0] HEX2;
    output DAC_SDI;        // SPI serial data out
    output DAC_CS;         // chip select - low when sending data to dac
    output DAC_SCK;        // SPI clock, 16 cycles at half sysclk freq
    output DAC_LD;
    output PWM_OUT;

    clkdiv clkdiv (CLOCK_50, 1, 12'd2499, clock);
    counter_10 counter0 (CLOCK_50, clock, count, 0);
    ROM rom0 (count, clock, data);
    pwm PWM0(CLOCK_50, data, clock, PWM_OUT);
    spi2dac SPI2DAC0(CLOCK_50, data, clock, DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);


endmodule
```

# Experiment 16

Friday, November 17, 2017 11:14 AM

In order to multiply the signal by two it can be shifted twice to the left. As can be seen below we have used a << operator. In fact it would be more efficient to just use the {} concatination operator

```
//-----------------------------
// Module name: allpass processor
// Function: Simply to pass input to output
// Creator:  Peter Cheung
// Version:  1.1
// Date:     24 Jan 2014
//-----------------------------

module processor (sysclk, data_in, data_out);

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

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

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

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

   // This part should include your own processing hardware
   // ... that takes x to produce y
   // ... In this case, it is ALL PASS.
   assign y = x << 2 ;

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

endmodule
```

# Experiment 17

Wednesday, December 13, 2017 12:12 PM

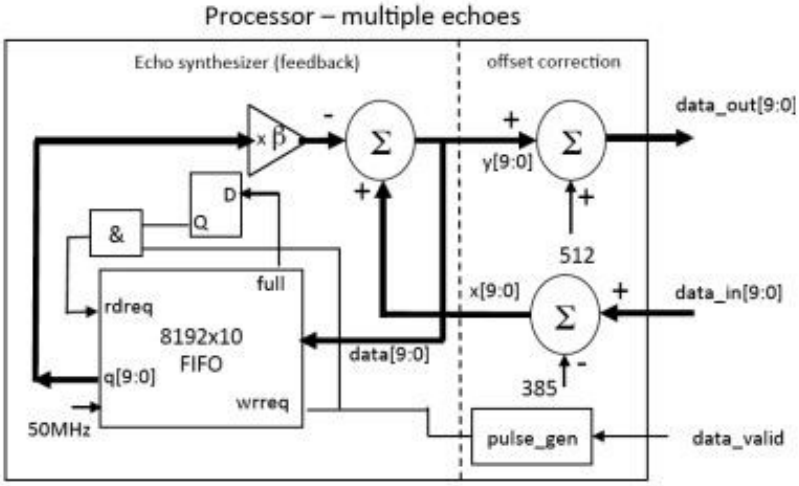
In order to implement the echo a FIFO (first in first out buffer, which is just a lot of shift registers) and a pulse generation module are required. The data_valid signal is inputed into the pulse_gen in order to generate one tick at a clock edge. This is fed into the FIFO to signal a write. The output of the full output of the FIFO has to be stored in a register and ANDED with the data_valid to allow a read. Data should not be read until the FIFO is full.

```
//-----------------------------
// Module name: allpass processor
// Function: Simply to pass input to output
// Creator:  Peter Cheung
// Version:  1.1
// Date:     24 Jan 2014
//-----------------------------

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

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

   reg q;

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

   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.
   pulse_gen   PULSE(pulse,data_valid,sysclk);
   fifo  FIFO0 (sysclk, x, rdreq, pulse,full, echo);

   always @(posedge sysclk)
      q <= full;

   assign rdreq = q & pulse;

   assign y = x + {echo[9], echo[9:1]};

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

endmodule
```

# Experiment 18

Wednesday, December 13, 2017 2:30 PM

Multiple echos are achieved by feeding the output back into the input. The echo is substracted from the input singnal in oder to have negative feedback. The phase shift is not percieved by the human ear.




```
// Version:  1.1
// Date:     24 Jan 2014
//-----------------------------

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

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

   reg q;

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

   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.
   pulse_gen   PULSE(pulse,data_valid,sysclk);
   fifo  FIFO0 (sysclk, y, rdreq, pulse,full, echo);

   always @(posedge sysclk)
      q <= full;

   assign rdreq = q & pulse;

   assign y = x + {echo[9], echo[9:1]};

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

endmodule
```