Smooth Ternary Signaling For Deep-Submicron(DSM) Buses

By
Robert Endicott Hanson and Ryan Ian Fullerton

Advisor:
Vladimir Prodanov

Senior Project
Bachelor of Science
Electrical Engineering Program
California Polytechnic State University
San Luis Obispo
June 4th 2011

Contents

Figures

Acknowledgements

We would like to thank Vladimir Prodanov for his help in advising us for the duration of the project design and construction, and for his excellent instruction during EE 308 and EE 409. We would also like to thank our families for their emotional and financial support throughout our time at Cal Poly.

Abstract

This project demonstrates the error mitigating effects of using a three-level “smooth ternary” signal on a Deep-Submicron(DSM) bus with large inter-wire capacitance. An RC circuit used to simulate and exaggerate (for easier testing) the charging and discharging of the DSM bus. An encoder stage translates binary input data into a three-level signal before sending it to the bus. A decoder stage then translates the three-level signal back to binary that is ideally identical to the original signal. Our results show that the three-level signal has much cleaner transitions than a traditional binary signal. A few glitches that appeared when decoding the ternary signal back to binary can be addressed and fixed in future iterations of the design.

I. Introduction

This Project simulates the effects of coupling capacitance when a smooth ternary signal is sent over a designed DSM bus. A 3-level (0, .5, 1) ternary signal is tested against a binary signal, passed through the same bus, to observe changes in timing and signal integrity. Crosstalk-induced performance degradation is one of the main determinants in overall system performance of a high-speed chip. It occurs due to pulling and pushing of neighboring voltage waveforms due to coupling capacitance between interconnects. A scheme to reduce crosstalk degradation and maintain signal integrity is devised and tested in this report. The main portion of the design utilizes LTSpice for quick changes and early simulation.

II. Background

DSM Bus

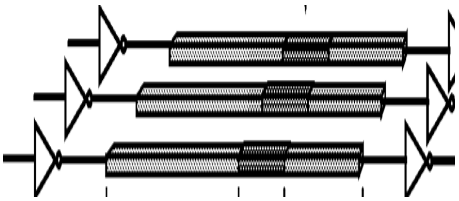

Figure 1: DSM Bus

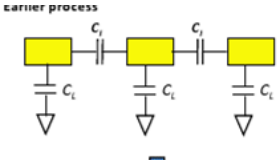

Figure 2: DSM Parasitic Capacitances

Charging and discharging of both inter-wire (C_I) and substrate (C_L) capacitances in a CMOS on-chip parallel bus make up a DSM bus. *Figure 3* shows the ratio between these two capacitances in regard to technology generation. Since inter-wire capacitance dominates substrate capacitance in smaller technologies (DSM), the substrate capacitance is disregarded for simulation.

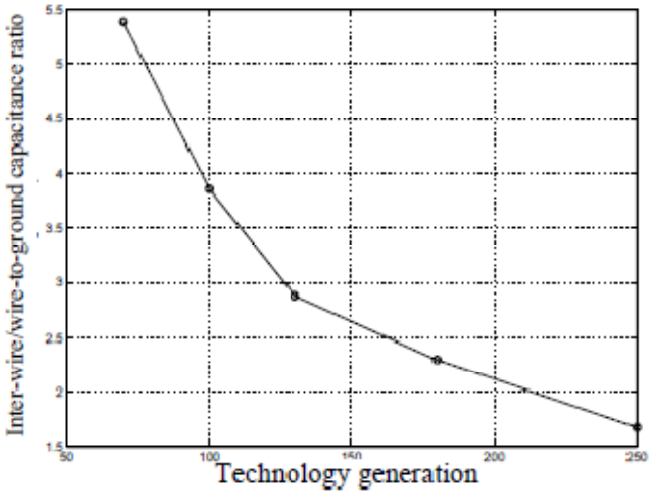

Figure 3: Capacitance Ratio in CMOS Technology

Crosstalk

Many schemes have been proposed to counter the effects of crosstalk. Some of these are passive and active shielding, data-dependent bus inversion, cross-talk avoidance coding, low voltage and multi-level signaling(Prodanov). Crosstalk via inter-wire capacitance dominates signal disruption and its modification depends on the transitions between three parallel data lines. We classify these transitions into three different categories. “Good” transitions occur when all three datelines mimic each other and either rise or fall. “Bad” transitions occur when then middle line rises and its neighbors hold their values and “Nasty” transitions occur when both the middle and outer lines transition but in opposite directions.

Equation 1:

$$\Delta Q = C_I V_{DD} \times \left(2\frac{\Delta V_{mid}}{V_{DD}} - \frac{\Delta V_{left}}{V_{DD}} - \frac{\Delta V_{right}}{V_{DD}}\right)$$

When looking at three parallel data lines, the badness of a transition can be quantified using the equation above representing the change in charge supported by the middle line, where C_I is the inter-wire capacitance and ΔV is the change in voltage on a particular line. When dealing with binary signals ΔV can take the value of *0V or* $\pm V_{DD}$. This means that the equation formula in the parentheses can take one of these values:

0, ±1, ±2, ±3, ±4

4 is the worst type of transition, and 0 is the best.

The worst transitions can be avoided by adding another voltage level to the system: $0.5V_{DD}$.

Now we have these possible values for ΔV: *0V, ±0.5V_{DD}, and ±V_{DD}.*

If we remove transitions of ΔV = ±V_{DD}, we now have only these transitions: *0V and ±0.5V_{DD}*. Plugging these values into the parentheses of the *equation 1* gives us these types of transitions:

0, ±0.5, ±1.0, ±1.5, ±2.0

This eliminates the "nasty" transitions of type 3 and 4.

III. Requirements

1. Translations from binary to ternary occur without flaws.
2. Minimal response time between input and output signals
3. Power supply provided exclusively from FPGA (Nexys 2).
4. Plug in Play connections for easy assembly and remodeling.
5. Bus designed with emphasis on crosstalk degradation on a binary signal for comparison between encoded and non-encoded signals.
6. Easy comparison between input and output signals for quick testing.

IV. Design

The complete design of this project requires four conceptual stages that comprise a system that effectively displays the difference between the binary and ternary signals.

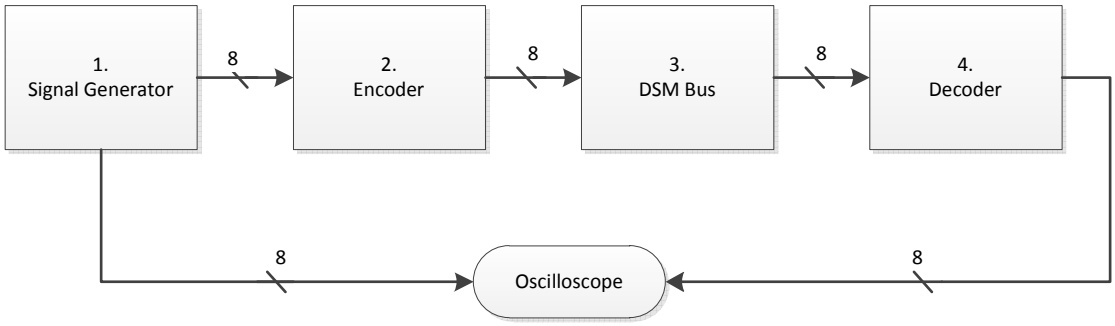

Figure 4: System Block Diagram

1. The signal generator creates the binary signal that will be used as an input to the encoder and will be compared to the final signal.
2. The encoder will take a binary input and encode it into the "smooth ternary" described in Figure 4: System Block Diagram
3. The DSM Bus simulates the inter-wire capacitance seen in a DSM bus.
4. The decoder takes the ternary signal and translates it back to the original binary signal.

The original signal and the final decoded signal can then be viewed and compared using an oscilloscope.

Portions of this design are purely logic based; therefore, for maximized customizability and minimal parts costs, much of the design was implemented on a Nexys 2 board equipped with a Spartan 3E FPGA and many IO ports.

Signal Generator Design

The signal generator is easily implemented on an FPGA. A signal model was created using VHDL to cover all of the possible data transitions on the bus. The sequences of bits were selected with the interaction of two neighboring data lines in mind. The signal sequence can be found in Appendix A.

Encoder Design

The encoder, in order to effectively prevent bad transitions on the signal sent through the bus, must take neighboring input data lines into account. The worst transitions occur when neighboring lines are charging opposite to each other, so in cases when neighboring data lines have potentials opposite to each other, the encoder outputs a value of 0.5Vdd to create a minimal transition.

b_{k-1}	b_k	S_1	S_0	t_k
0	0	0	0	0
0	1	0	1	0.5
1	0	0	1	0.5
1	1	1	0	1

Figure 5: Encoder Truth Table

In this truth table b_k signifies the value of the input data line, b_{k-1} signifies the previous neighboring line, and t_k signifies the output "smooth" ternary signal value.

This design is implemented using a 4:1 MAX4518 analog MUX with 3 of the inputs connected to reference voltages signifying the logical states 0, 1, and 0.5. The fourth MUX input is tied to ground and not used. There is one MUX per data line on the bus, making eight MUXs in the Encoder. The Encoder select bits are driven by the FPGA logic which follows this structure:

sel_out(1) <= b_k AND b_{k-1};

sel_out(0) <= b_k XOR b_{k-1};

DSM Bus Design

The DSM bus is designed as a simple RC network to simulate inter-wire capacitance on the bus.

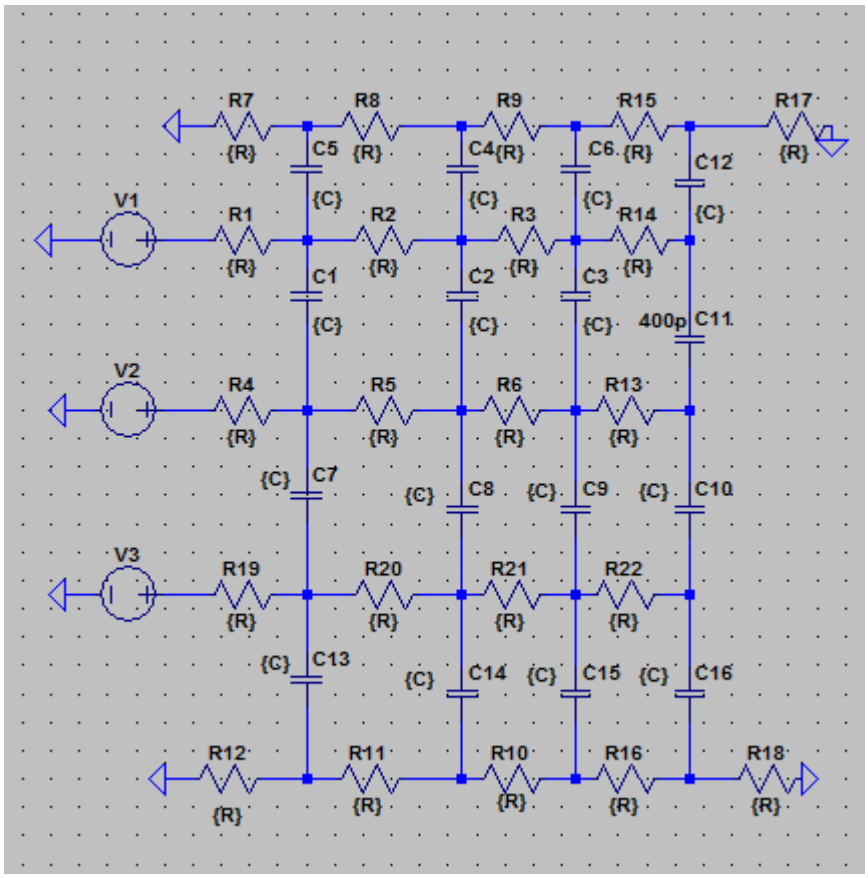

Figure 6: DSM Bus Simulation Design

Figure 6 shows the preliminary design of the DSM bus in LTspice. The figure shows three data lines with shielding lines on either side of the bus. Each of the lines has inter-wire capacitance between them. The R and C values were selected to effectively show crosstalk on the center line when 10kHz baud rate signals were applied at V1, V2 and V3.

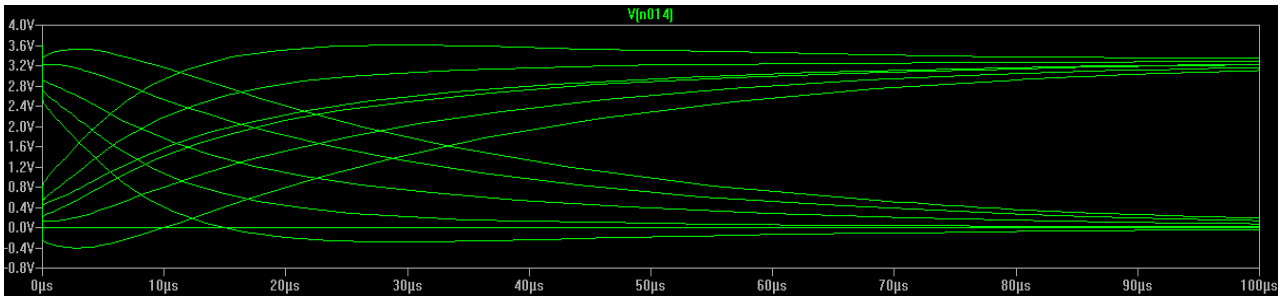

Figure 7: Simulated Eye Diagram for DSM Bus

Figure 7 shows the eye diagram produced by LTspice with R=2.4kΩ, C=470pF, and 10 kHz baud rate signals through the bus. These values were chosen because you can clearly see each type of transition on the simulated eye diagram. The worse transition takes 35µs to reach 50% of its final value. The worst transitions show the charging of the center line while both of its neighbors are charging in the opposite direction.

With the final values chosen we modeled our final bus design after *figure 6* but with eight data lines instead of three.

Decoder Design

The decoder has the task of detecting three levels and translating back to binary. In order to detect the 0.5Vdd signal we need a window detector.

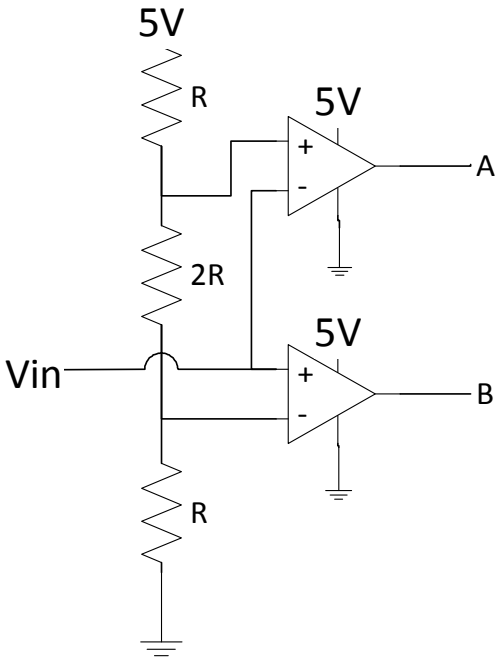

Figure 8: Design for Window Detector

The window detector tells the FPGA whether the signal coming from the bus is in the range of 0, 1, or 0.5. The center resistor from the voltage divider is twice the value of the other resistors because a signal at 0.5Vdd is susceptible to being pulled in either direction by crosstalk, necessitating a larger window for increased noise margin.

t_k	A	B	b_k
1	0	1	1
0	1	0	0
0.5	1	1	NOT b_{k-1}

Figure 9: Truth Table for Decoder

The decoder logic in the FPGA folows this structure:

b_k <= (NOT b_{k-1}) when (A AND B) = '1' else B;

This code implements the truth table seen in *figure 9*.

With the final signal decoded back into its original binary form, the Nexys 2 board outputs the decoded signals for connection to an oscilloscope for comparison with the original generated signals.

V. Test Plans

The purpose of this project is to observe the error mitigating effects of a three-level signal over a binary signal; therefore, we set up our testing to do this. The tests are mainly comparisons between two signals to view their distortions.

This is a list of the test plans created for the project

1. Observe neighboring binary signals on the bus and their effects on each other.

   Input – binary signals generated by the FPGA, directly connected to the bus without encoder.

   Test Points – output of bus at multiple neighboring data lines

2. Observe neighboring ternary signals on the bus and their effects on each other.

   Input – ternary signal produced by encoder

   Test Points - output of bus at multiple neighboring data lines

3. Compare original and final binary signals

   Input – complete connected system:

   generator->encoder->bus->decoder

   Test Points – one probe connected to original signal produced by the generator, another probe connected to the decoder output

These test results, when viewed in aggregate, create a good model of the systems functionality.

VI. Development and Construction

Our final designs for the encoder, decoder, and bus uses a minimum of four breadboards to complete and it is for this reason that proto-boards were used instead. Using proto-boards, laying out integrated circuits in a compact design was possible. The encoder and decoder were designed to keep all the inputs and outputs at convenient points to attach connectors that interface between the components.

Encoder

To reduce the number of IC components required for our design, the Nexys 2 board was used to carry out logic processes.

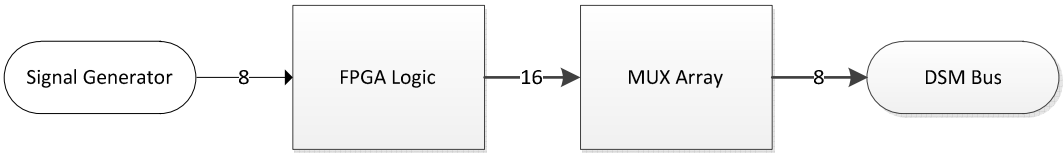

Figure 10: Encoder Block Diagram

The FPGA on the Nexys 2 produces the select bits that drive an array of 8 MUXs (one for each data line on the bus). Because each MUX requires two select bits, there are 16 lines from the Nexys 2 to the MUX array. The MUX array uses the 16 select lines to produce eight ternary signals that drive the bus.

b_k	b_{k-1}	S_1	S_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 11: Nexys 2 MUX Select Bit Truth Table

The logic on the FPGA follows the truth table in *figure 11*, where b_k is the binary input line being encoded, b_{k-1} is the previous input line, and S1 and S1 are the select bits that control the MUX. Because the very first binary line does not have a previous neighbor, its value is passed directly through the encoder into the bus.

Initial testing of the MUX array design was carried out on a breadboard. Using the MAX4518 data-sheet as a reference, we verified the functionality of the MUX truth table when our reference voltages (0V, 2.5V, and 5V) were connected to the inputs. The 2.5V reference is created using a voltage divider consisting of two 2.4kΩ resistors and was buffered using a MCP6002 op-amp in unity gain configuration. *Figure 12* displays the results of testing:

S1	S0	Expected Output (V)	Experimental Output (V)
0	0	0	0
0	1	2.5 (0.5Vdd)	2.5
1	0	5	5
1	1	Not Used (grounded)	0

Figure 12: Experimental MUX Truth Table

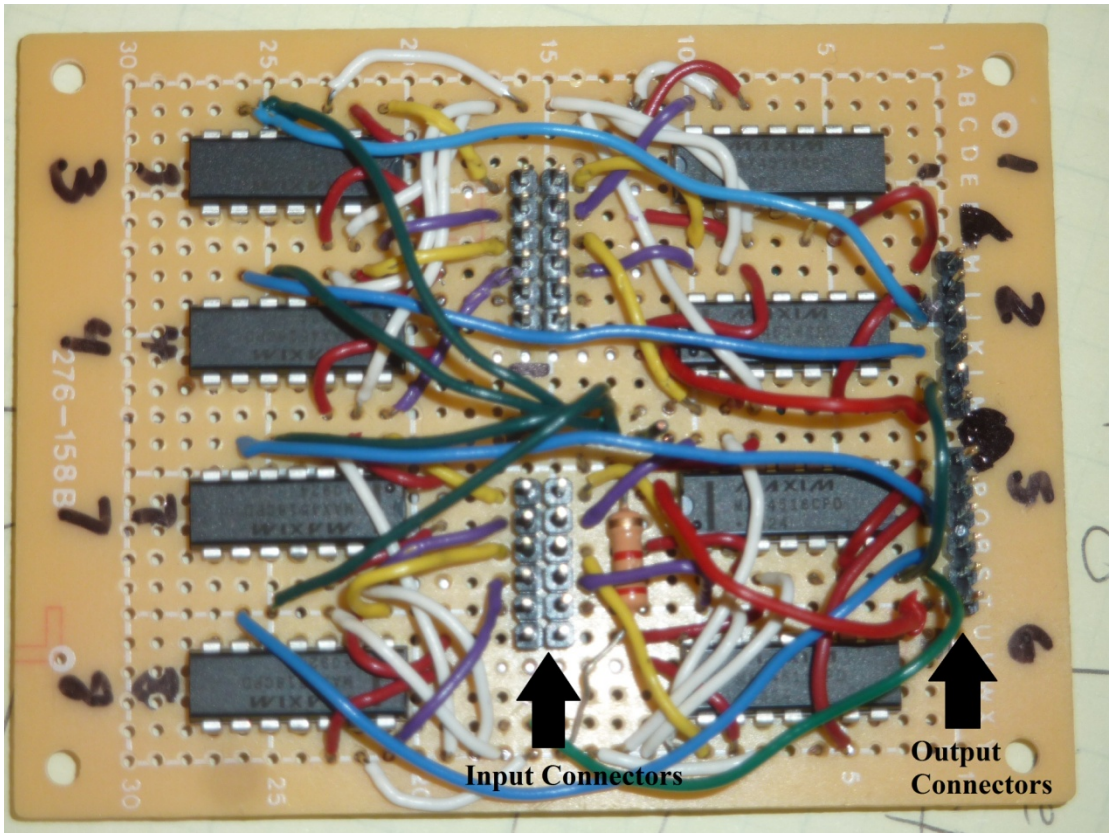

Figure 13: Constructed Encoder MUX Array

Featuring eight multiplexers, the encoder needed to be laid out for easy access to the input and output connections. As seen in *figure 13*, the input to the MUX array is located in the center of the proto-board, allowing easy access to data and power pins for each of the MUX ICs.

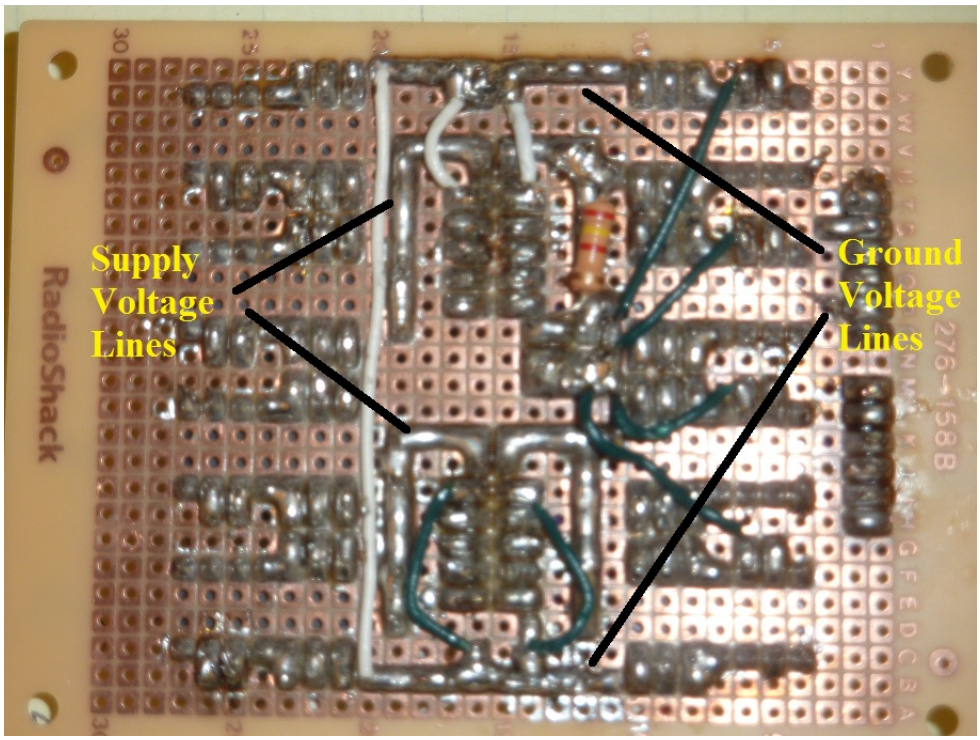

Figure 14: Bottom of Encoder MUX Array

Figure 5 shows the layout of the bottom of the board. Here you can see how the power reference voltage lines were dispersed across the board for efficient connections.

DSM Bus

The DSM bus has a simple RC network design that can be reviewed in the design section of this paper. The values of the network were selected based on LTspice simulations where we exaggerated the effects of the inter-wire capacitance until the bad transitions were clearly visible. We selected the values to be R1 = 2.4k

and C1 = 470pF.

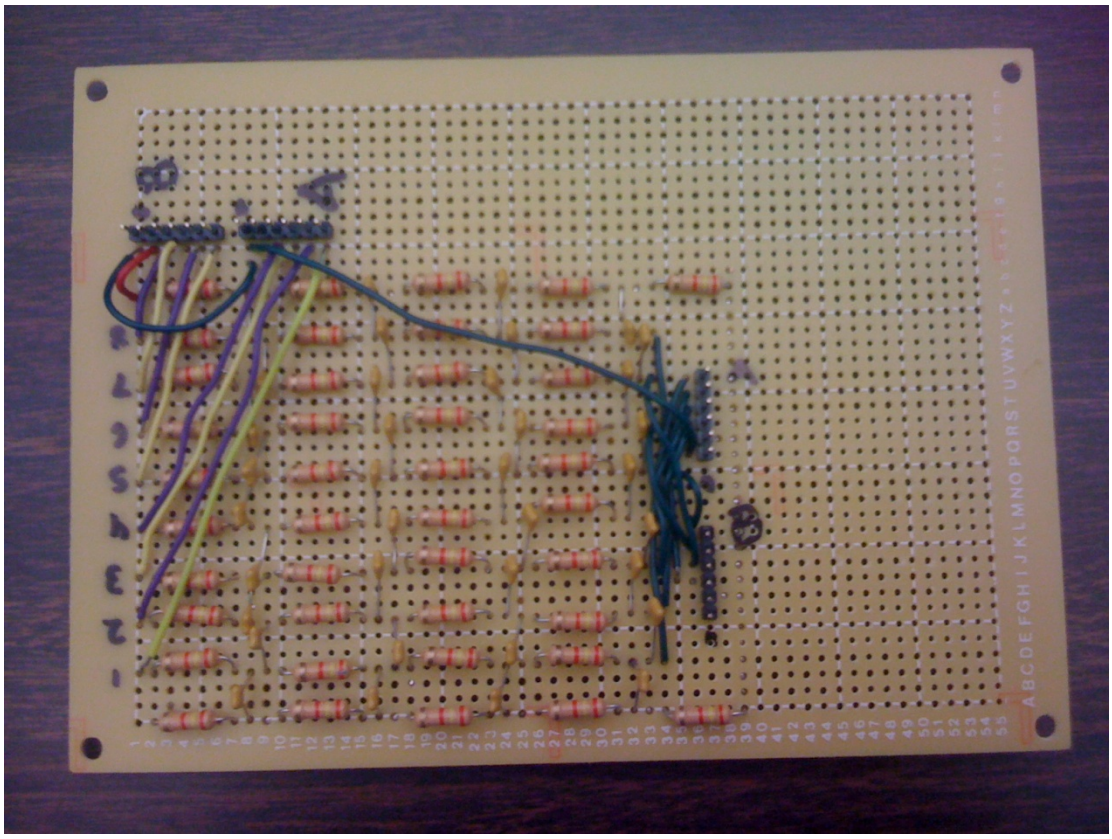

Figure 15: Constructed DSM Bus

The completed design is shown in *figure 15*. The inputs and outputs were combined to create simple plugs for easy assembly.

Decoder

In the decoder design, the FPGA takes care of most of the logic.

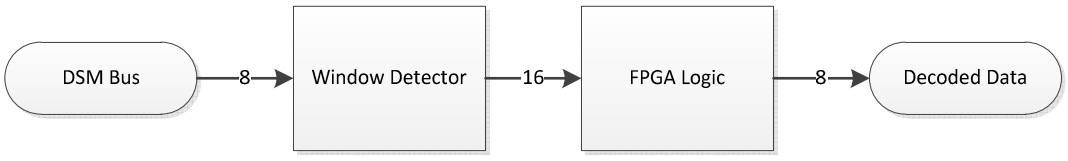

Figure 16: Decoder Block Diagram

The window detector outputs two values that tell the FPGA what range the detected voltage is currently in. The truth table for the window detector can be seen in the Decoder Design section of this paper.

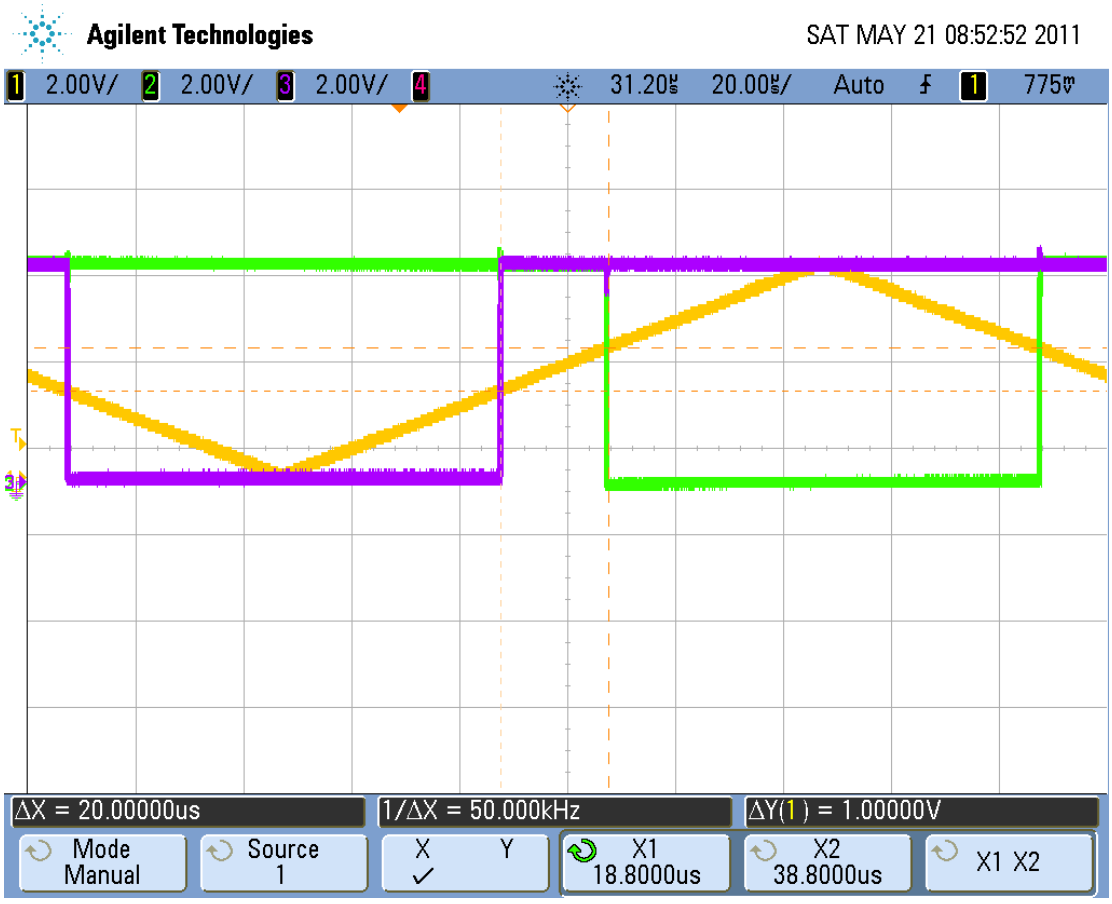

Figure 17: Window Detector Performance

Figure 17 shows the response of the window detector while being driven by a triangle wave. Observe that when the driving signal is between the high and low ranges, both output signals are pulled high. And when the input signal is either high or low, the output signals are latched opposite from each other.

The window detector stage is constructed in an array, much like the MUXs of the encoder stage. There is one window with detector per data line on the bus, making 16 lines to the FPGA.

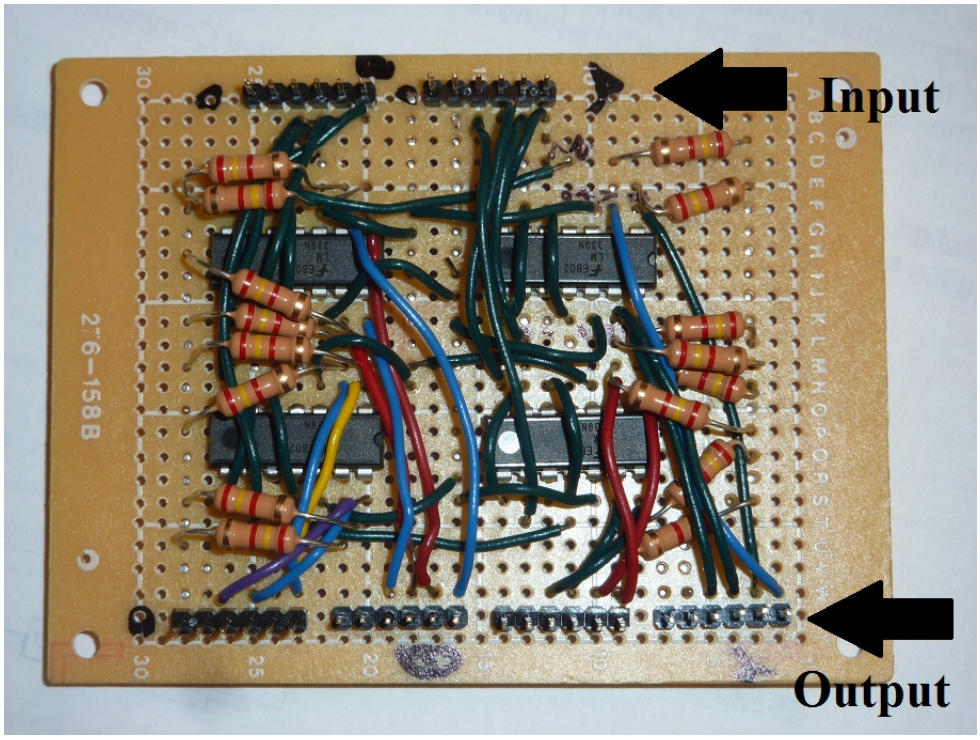

Figure 18: Constructed Window Detector Array

The Window detector is constructed from LM339 comparators. There are four comparators per IC so the construction only requires four ICs to make eight window detectors. The resistors on the board are pull-up resistors to the 3.3V reference coming from the Nexys 2. They are pulled to 3.3V, instead of 5V, to make them compatible with the Nexys 2 logic inputs.

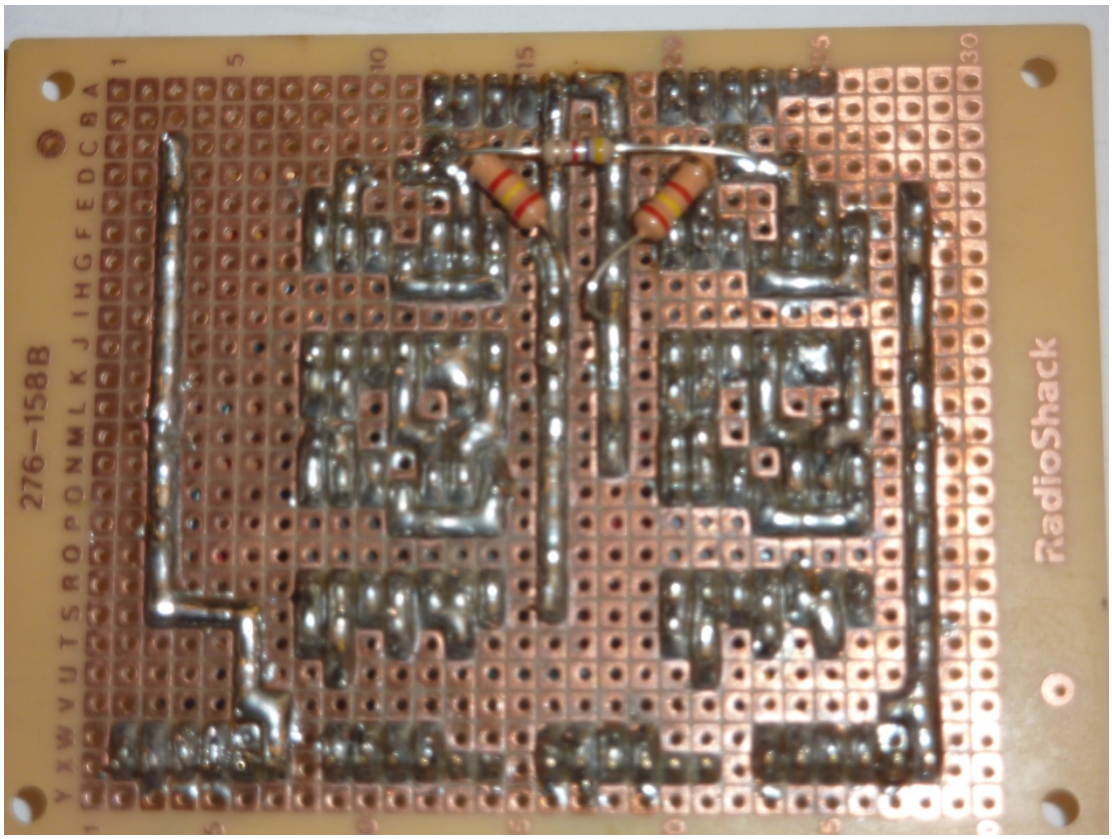

Figure 19: Bottom of Window Detector Array

The resistive divider creating the detector reference voltages is visible in *figure 19*.

VII. Integration and Test Results

The integration of the separate system components is simple due to our inclusion of Pmod connectors on each component during the construction phase. In order to easily understand how to connect each component we labeled the connectors with the letters A and B and placed dots on the proto-board nearest to the power pins of each connector.

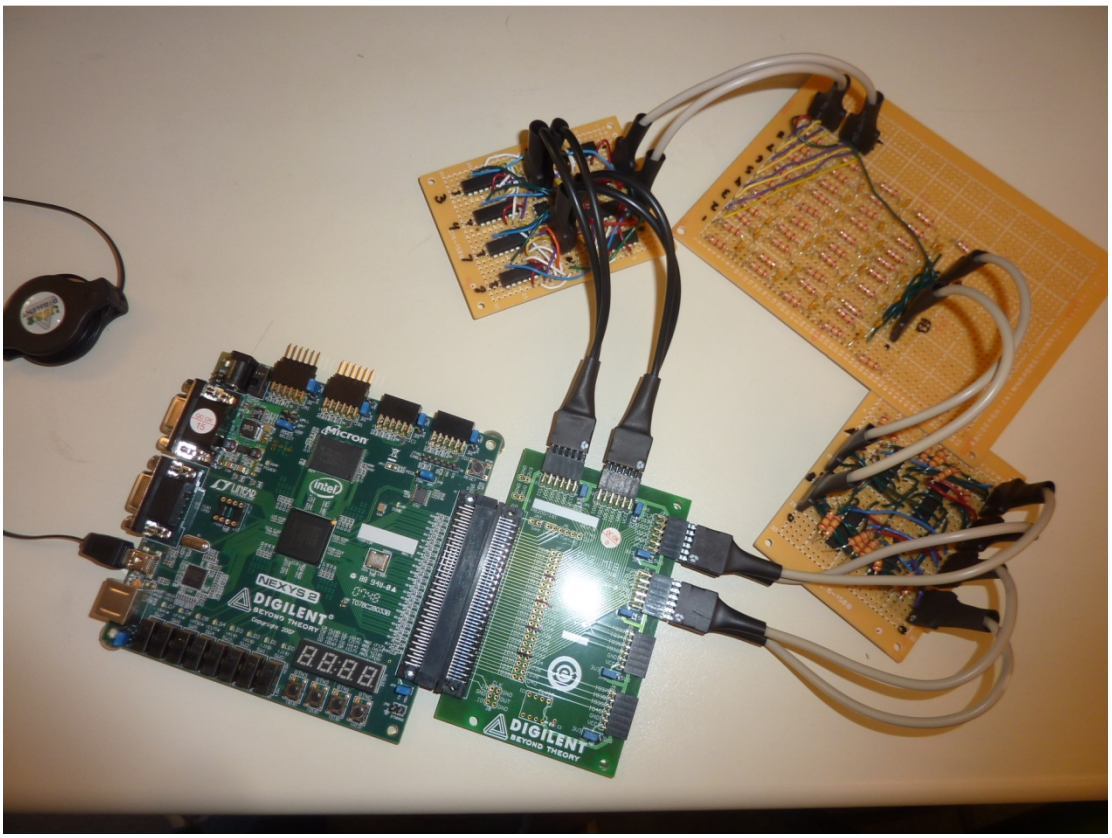

Figure 20: Constructed System

Figure 20 shows the completed system with all connections made.

It is evident from viewing the system that further iterations of the design can benefit from some sort of stabilizing structure to keep the components in place.

Test Results

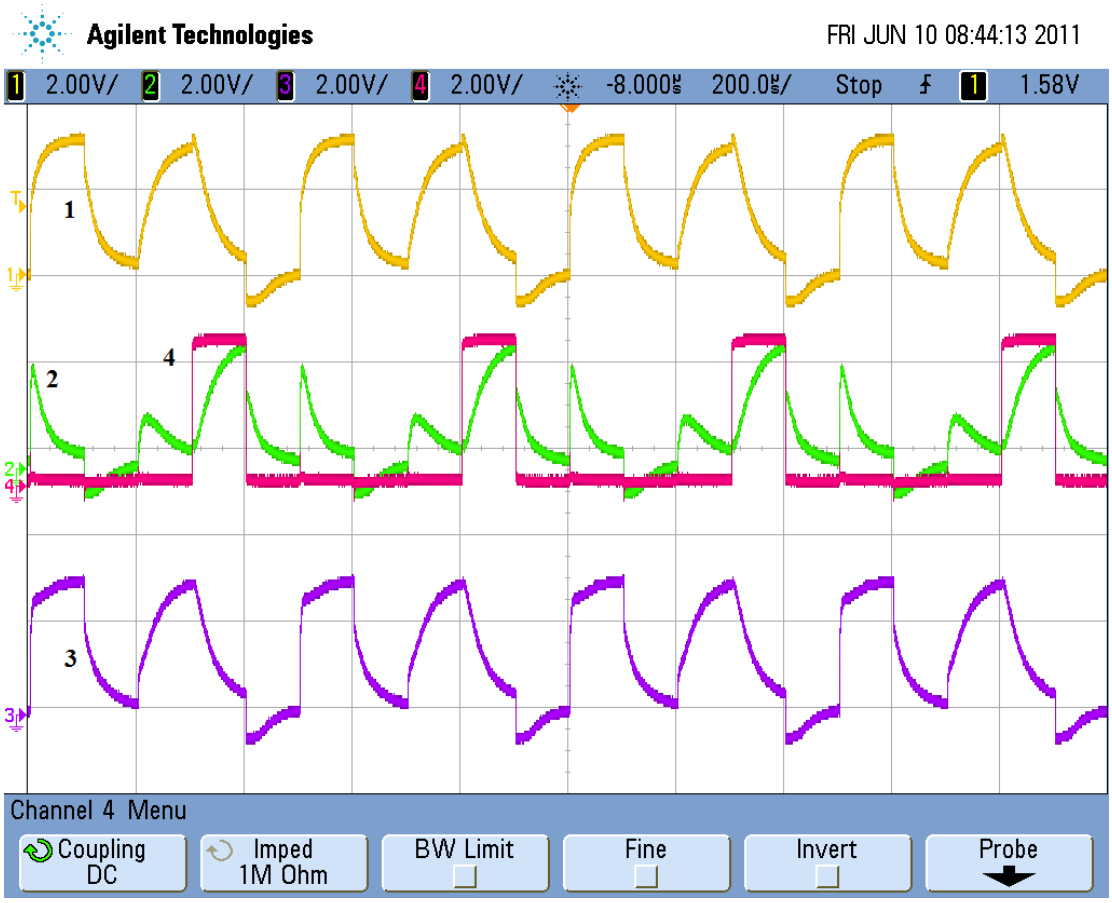

Figure 21: Binary Signals on a Simulated DSM Bus

Traces 1, 2, and 3 on the oscilloscope plot in figure 6 show three neighboring lines on the bus being driven by *binary* data. The data line signified by trace 2 is located between the other two lines and is subject to their crosstalk. Trace 4, overlaying trace 2, shows the undistorted binary signal from which the green lin e is being driven.

Clearly, on a bus with inter-wire capacitance of this magnitude, the signal output from the bus does not resemble the input signal driving it.

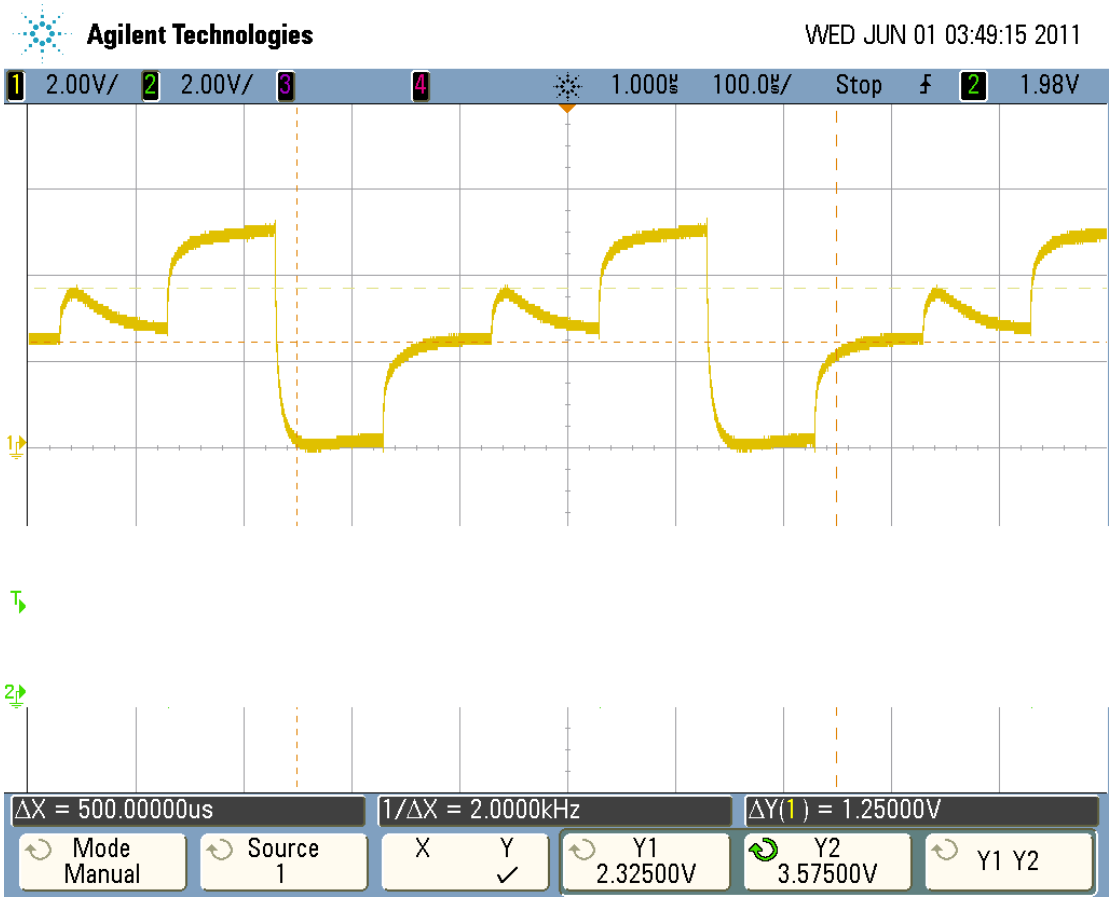

Figure 22: Ternary Signal on DSM Bus

Figure 22 shows a ternary signal taken from the output of the DSM Bus, it is apparent that there is some distortion from crosstalk; however, it is of a much lesser magnitude than the distortion seen in *figure 21*.

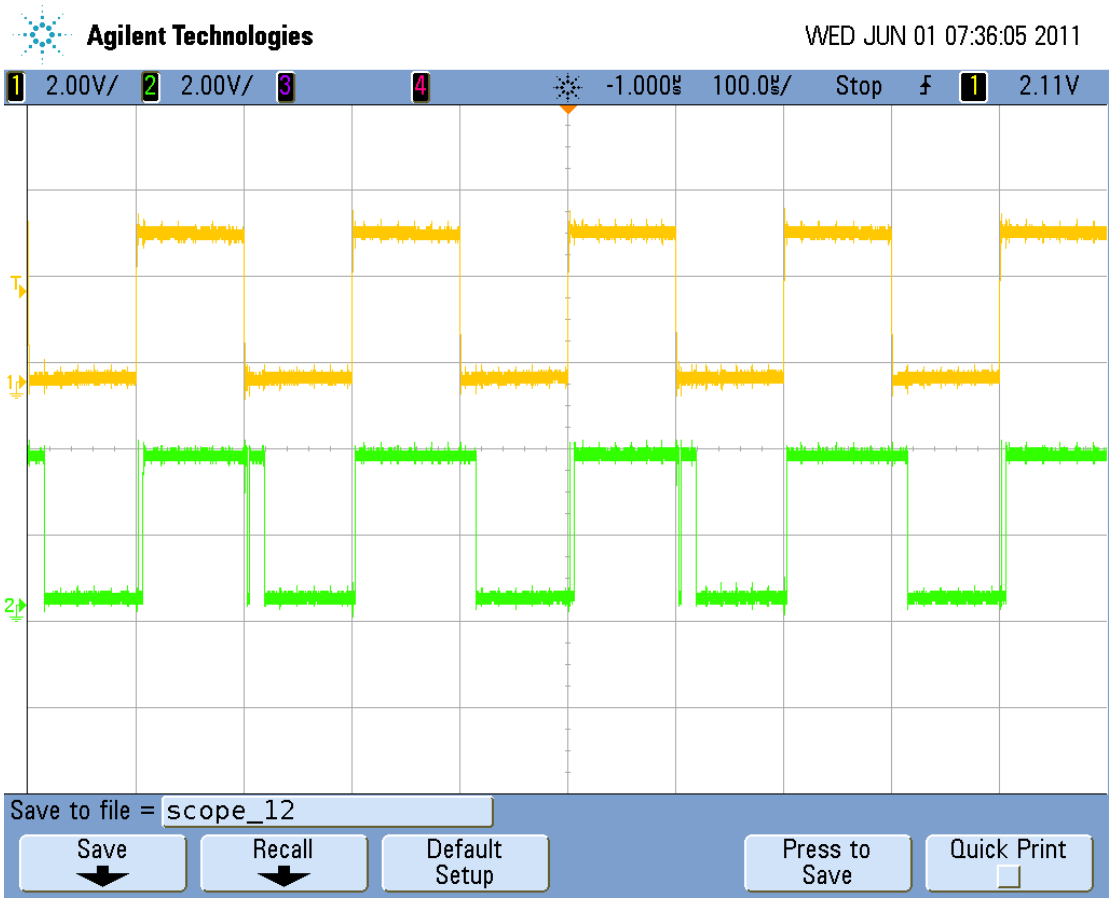

Figure 23: Original and Final Signals

Figure 8 shows a test signal that was sent through the system. The top signal is the original data that was input to the encoder. The bottom signal is the final decoded signal after being sent through the entire system. Careful inspection shows that the final signal is not perfect. There are some glitches that occur in the logic sometimes when the signal on the bus charges or discharges from rail to rail. This is due to the fact that the window detector reads the value as 0.5Vdd for a portion of the transition. This issue might be resolved with some modification of the decoder logic, a filtering

capacitor on the output of the system, or some logic that ignores changes in logic that do not persist for a minimum amount of time.

VIII. Conclusion

Our system shows the potential for effective crosstalk mitigating systems on a bus with considerable inter-wire capacitance. The addition of a logic level on the bus reduces the magnitude of charging and discharging between changes in state. The output of the system was not a perfectly identical to the input signal; however, the complete data can be interpreted from the result. From the test results it is clear that the ternary signals do not experience as much distortion due to crosstalk that the binary signals do.

The RC values that we chose for DSM Bus were greatly exaggerated; in future iterations of the system design, more realistic values might be chosen. However, for the purposes of our research the large RC time-constant clearly accentuated the distortion that can be encountered on high inter-wire capacitance buses.

IX. Bibliography

Duan, Chunjie, and Sunil P. Khatri. *Energy Efficient and High Speed On-Chip Ternary Bus*. Tech. no. 978-3-9810801-3-1. Mitsubishi Electric Research Laboratories, 2008. Print.

Najeeb, K., Vishal Gupta, and V. Kamakoti. "Delay and Peak Power Minimization for On-Chip Buses Using Temporal Redundancy." Reading. Department of Computer Science & Engineering, India. 2010. Dr. V. Kamakoti. Web. <http://vlsi.cs.iitm.ernet.in>.

Prodanov, Vladimir. "3-level Signaling on a DSM Bus." California Polytechnic, San Luis Obispo. Jan. 2011. Lecture.

"MAX4518/MAX4519 datasheet" MAXIM. May 1998

"MCP6001/1R/1U/2/4 datasheet" Microchip Technology. 26 Mar 2009

"LM139/LM239/LM339/LM2901/LM3302 datasheet" National Semiconductor. Mar 2004

"Digilent Nexys2 Board Reference Manual" Digilent. 21 June 2008

"Nexys II Schematic" Digilent. 19 July 2007

Appendix A: VHDL Code

Top Level

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DSMternary is
   Port ( clk : in  STD_LOGIC;
                              reset : in STD_LOGIC;
                              clk_5k_out : out STD_LOGIC;
                  sel_out : out  STD_LOGIC_VECTOR (15 downto 0);
                              detectors : in STD_LOGIC_VECTOR (15 downto 0);
                              comparator_in : in STD_LOGIC_VECTOR (15 downto 0);
          bk : out  STD_LOGIC_VECTOR (7 downto 0);
                              bk_original : out  STD_LOGIC_VECTOR (7 downto 0)
                              );

end DSMternary;

architecture Behavioral of DSMternary is

component Encoder_Bus is
   Port ( bk_enc_in : in  STD_LOGIC_VECTOR (7 downto 0);
          sel_out : out  STD_LOGIC_VECTOR (15 downto 0));
end component;

component Decoder_Bus is
   Port ( detectors : in  STD_LOGIC_VECTOR (15 downto 0);
          bk : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component signal_generator is
   Port ( clk : in  STD_LOGIC;
                              reset : in STD_LOGIC;
                              clk_5k_out : out STD_LOGIC;
          data_out : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

signal bk_enc_in : std_logic_vector(7 downto 0);


begin
```

```
--Encoder Bus Instantiation
inst_encoder_bus : Encoder_Bus
        port map (
                bk_enc_in => bk_enc_in,
                sel_out => sel_out
                );

--Decoder Bus Instantiation
inst_decoder_bus : Decoder_Bus
port map (
                detectors => detectors,
                bk => bk
                );

--Signal Generator Instantiation
inst_signal_generator : signal_generator
port map (
                clk => clk,
                reset => reset,
                clk_5k_out => clk_5k_out,
                data_out => bk_enc_in
                );

bk_original <= bk_enc_in;

end Behavioral;
```

Signal Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity signal_generator is
    Port ( clk : in  STD_LOGIC;
                                reset : in STD_LOGIC;
                                clk_5k_out : out STD_LOGIC;
           data_out : out  STD_LOGIC_VECTOR (7 downto 0));
end signal_generator;

architecture Behavioral of signal_generator is

type generator_state is (s1, s2, s3, s4, s5, s6);

signal state_cur: generator_state;
signal state_next: generator_state;
signal clk_5k : std_logic;
signal data_cur : std_logic_vector (7 downto 0) := "11111111";
signal data_next : std_logic_vector (7 downto 0);

begin
--Generate signal clock
process(reset, clk)
          variable counter_50M : integer range 0 to 5000; --Divide clock downto 5kHz
          begin
                    if reset = '1' then
                              counter_50M := 0;
                    elsif rising_edge(clk) then
                                        counter_50M := counter_50M + 1;
                              if counter_50M = 2500 then
                                        clk_5k <= not clk_5k;
                                        counter_50M := 0;
                              end if;
                    end if;
end process;

clk_5k_out <= clk_5k;


--GENERATE SIGNAL

--Update data
```

```
process(reset, clk_5k)
	begin
		if reset = '1' then
			data_cur <= "00110011";
			state_cur <= s1;
		elsif rising_edge(clk_5k) then
			data_cur <= data_next;
			state_cur <= state_next;
		end if;
end process;

--Define data sequence
process(reset, clk_5k)
	begin
		case state_cur is
			when s1 =>
				data_next <= "00110011";
				state_next <= s2;
			when s2 =>
				data_next <= "01100110";
				state_next <= s3;
			when s3 =>
				data_next <= "11001100";
				state_next <= s4;
			when s4 =>
				data_next <= "00110011";
				state_next <= s5;
			when s5 =>
				data_next <= "11001100";
				state_next <= s6;
			when s6 =>
				data_next <= "10011001";
				state_next <= s1;
			when others =>
				data_next <= "00110011";
				state_next <= s1;
		end case;
end process;

--output data
data_out <= data_cur;

end Behavioral;
```

Encoder Bus

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Encoder_Bus is
    Port ( bk_enc_in : in  STD_LOGIC_VECTOR (7 downto 0);
           sel_out : out  STD_LOGIC_VECTOR (15 downto 0));
end Encoder_Bus;

architecture Behavioral of Encoder_Bus is

component Encoder
    Port ( bk_in : in  STD_LOGIC_VECTOR (1 downto 0);
           sel_out : out  STD_LOGIC_VECTOR (1 downto 0));
end component;

signal encoder1_in : std_logic_vector(1 downto 0);

begin

-- encoder 1
encoder1_in <= bk_enc_in(0) & '0'; -- Tie bk_prev of channel 1 to '0'

encoder1 : Encoder
  port map (
                    bk_in => encoder1_in(1 downto 0),
                    sel_out => sel_out(1 downto 0)
                    );

-- encoder 2
encoder2 : Encoder
  port map (
                    bk_in => bk_enc_in(1 downto 0),
                    sel_out => sel_out(3 downto 2)
                    );

-- encoder 3
encoder3 : Encoder
  port map (
                    bk_in => bk_enc_in(2 downto 1),
                    sel_out => sel_out(5 downto 4)
                    );
```

```
-- encoder 4
encoder4 : Encoder
  port map (
                bk_in => bk_enc_in(3 downto 2),
                sel_out => sel_out(7 downto 6)
                );

-- encoder 5
encoder5 : Encoder
  port map (
                bk_in => bk_enc_in(4 downto 3),
                sel_out => sel_out(9 downto 8)
                );

-- encoder 6
encoder6 : Encoder
  port map (
                bk_in => bk_enc_in(5 downto 4),
                sel_out => sel_out(11 downto 10)
                );

-- encoder 7
encoder7 : Encoder
  port map (
                bk_in => bk_enc_in(6 downto 5),
                sel_out => sel_out(13 downto 12)
                );

-- encoder 8
encoder8 : Encoder
  port map (
                bk_in => bk_enc_in(7 downto 6),
                sel_out => sel_out(15 downto 14)
                );

end Behavioral;
```

Encoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Encoder is
    Port ( bk_in : in  STD_LOGIC_VECTOR (1 downto 0);
           sel_out : out  STD_LOGIC_VECTOR (1 downto 0));
end Encoder;

architecture Behavioral of Encoder is

signal xor_out : std_logic;

begin
sel_out(1) <= bk_in(1) AND bk_in(0);
sel_out(0) <= bk_in(1) XOR bk_in(0);

end Behavioral;
```

Decoder Bus

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Decoder_Bus is
    Port ( detectors : in  STD_LOGIC_VECTOR (15 downto 0);
           bk : out  STD_LOGIC_VECTOR (7 downto 0));
end Decoder_Bus;

architecture Behavioral of Decoder_Bus is

component Decoder is
    Port ( detector_A : in STD_LOGIC;
                                detector_B : in STD_LOGIC;
           bk_prev : in  STD_LOGIC;
           bk : out  STD_LOGIC);
end component;

signal bk_sig : std_logic_vector(7 downto 0);

begin

-- decoder 1
decoder1 : Decoder
   port map (
             detector_A => (not detectors(1)), --channel one has unique behavior because
it is the first channel on the bus
                     detector_B => detectors(1),
      bk_prev => bk_sig(0),
      bk => bk_sig(0)
                     );

-- decoder 2
decoder2 : Decoder
   port map (
             detector_A => detectors(2),
                     detector_B => detectors(3),
      bk_prev => bk_sig(0),
      bk => bk_sig(1)
                     );

-- decoder 3
decoder3 : Decoder
```

```
  port map (
          detector_A => detectors(4),
                  detector_B => detectors(5),
    bk_prev => bk_sig(1),
    bk => bk_sig(2)
                  );

-- decoder 4
decoder4 : Decoder
  port map (
          detector_A => detectors(6),
                  detector_B => detectors(7),
    bk_prev =>bk_sig(2),
    bk => bk_sig(3)
                  );

-- decoder 5
decoder5 : Decoder
  port map (
          detector_A => detectors(8),
                  detector_B => detectors(9),
    bk_prev => bk_sig(3),
    bk => bk_sig(4)
                  );

-- decoder 6
decoder6 : Decoder
  port map (
          detector_A => detectors(10),
                  detector_B => detectors(11),
    bk_prev => bk_sig(4),
    bk => bk_sig(5)
                  );

-- decoder 7
decoder7 : Decoder
  port map (
          detector_A => detectors(12),
                  detector_B => detectors(13),
    bk_prev => bk_sig(5),
    bk => bk_sig(6)
                  );

-- decoder 8
```

```
decoder8 : Decoder
  port map (
            detector_A => detectors(14),
                  detector_B => detectors(15),
     bk_prev => bk_sig(6),
     bk => bk_sig(7)
                  );

bk <= bk_sig; --Output decoded bk values

end Behavioral;
```

Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Decoder is
    Port ( detector_A : in  STD_LOGIC;
                detector_B : in STD_LOGIC;
           bk_prev : in  STD_LOGIC;
           bk : out  STD_LOGIC);
end Decoder;

architecture Behavioral of Decoder is

begin

bk <= (NOT bk_prev) when (detector_A AND detector_B) = '1' else detector_B;

end Behavioral;
```