40 Gbps SiGe PATTERN GENERATOR IC WITH VARIABLE CLOCK SKEW AND OUTPUT LEVELS

By

MATTHEW JOHN ZAHLLER

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINERING

WASHINGTON STATE UNIVERSITY
Department of Electrical and Computer Engineering

DECEMBER 2006

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of MATTHEW JOHN ZAHLLER find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGMENT

I would like to first acknowledge the help of my advisor Dr. George La Rue for his help and guidance with formulating the idea for the pattern generator and its unique components as well as for the answering of numerous questions.

Also I would like to acknowledge Dirk Robinson for his guidance with design issues, operating system problems, and most importantly test procedure and equipment lessons.

40 Gbps SiGe PATTERN GENERATOR IC WITH

VARIABLE CLOCK SKEW AND OUTPUT LEVELS

Abstract

by Matthew John Zahller, MS
Washington State University
December 2006

Chair: George S. La Rue

A single-chip 40 Gbps pattern generator design in 0.18 μm SiGe BiCMOS technology is described. An on-chip 128x128 bit RAM with an access time of 3 ns stores the data pattern. A hybrid 128:1 CMOS/ECL multiplexer increases the output data rate from the RAM to 40 Gbps. The output driver is back terminated with 50 ohms and provides programmable levels in the range -2 V to 2 V into a 50 ohm load. The simulated pattern dependent jitter is under 1 ps at all output levels. The clock can be delayed by a programmable number of clock cycles plus a vernier delay of up to 50 ps in 0.2 ps steps in simulation. Power dissipation is up to 1.5 W depending on the output amplitude and termination voltage.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Dedication

I would like to dedicate this to my wife Kimber for her patience, encouragement and love. Without her I would never have completed my goals.

CHAPTER ONE

INTRODUCTION

The continued advances in CMOS and bipolar integrated circuit technologies has given rise to higher speed circuits and the need for low-cost high-speed pattern generators to test these circuits. High speed SiGe BiCMOS technology, which combines very high speed heterojunction bipolar transistors (HBTs) and high density CMOS, is a natural choice for integrating pattern memory with high-speed multiplexers and output drivers. InP and GaAs components at 40 Gbps [1] cannot provide a one-chip solution and the cost is much higher. Available pattern generators include the DG2020, DG2030, DG2040 series from Tektronics; M2i.7021-Digital I/O from Spectrum; and Stressed Pattern Generator from BERTScope.

The requirements for our high-speed pattern generator are 1) differential outputs with 50 ohm output impedance; 2) programmable output high and low levels; 3) a maximum swing of 1.5 V into 50 ohm loads; 4) programmable delay with resolution of 0.2 ps; 5) a memory depth of at least 64K bits; and 6) pattern dependent jitter less than 1 ps.

Vernier Generation Technique

The creation of high speed data pattern generation has increased the need for high accuracy delay techniques. High accuracy delays are utilized for use in multiple channel pattern generation where individual channel delay paths may need to be aligned or purposely delayed with respect to each other. Data pattern speed of 40Gps therefore requires very fine timing resolution. The use of bipolar technology instead of CMOS technology not only allows for much higher speed data pattern generation, but also allows

the realization of data dependent jitter in the hundreds of femtoseconds region leading to very high resolution.

Multiple methods have been developed to create a variable delay. The vernier delay line method is extremely popular with multiple variations such as: two level, oscillation mismatch, folding, self-sampled, component invariant, and built in self test pulse width modulation calibration[2]-[7]. VDL techniques require very accurate matching, PLL or DLL technology or additional manipulations to reduce clock jitter and mismatch specifications. Using a single absolute delay at higher speeds will not give the timing resolution required[8]. The two inverter current switching technique allows for a differential delay approach rather than a single ended approach, and increases linearity and common mode noise by having an inverter structure which utilizes differential input pairs. Bipolar differential pairs also contribute to a more accurate delay value as long as the current from the current sources are accurately controlled.

The structure of the delay is to partially switch binary weighted currents between a one inverter and two inverter path. This structure's accuracy and reliability does not rely on a clocking system, or flip flop timing.

In addition, high speed data generation can be especially sensitive to noise. This noise will be seen in the output data pattern as timing jitter. Again, by using differential logic, we greatly reduce common mode noise, as well as increase power supply rejection. Also all CMOS current sources were isolated with guard rings to protect them from switching noise.

PRBS Generators in SiGe Bipolar Technology

PRBS generators can provide high bits rates without requiring the multiplexing of lower speed data with multiple pipelined stages of multiplexers[9]. A significant achievement for this pattern generator is the ability for it to have on chip RAM and the ability to have clock delay control as well as control over the output levels.

In the past PRBS generators have reached speeds of up to 100Gbps with a data dependent jitter of 2ps[9]. At even lower rates of 80Gbps and 40Gbps output jitter of 700fs and less than 500fs were achieved[9]. There have also been variable current sources which allow for variable output amplitude. But the actual output swings have been limited to 300mVpp and 1Vpp[9]. The current output driver design can vary the output levels from 2 volts to -2 volts, with a swing of 0 volts to 1.5 volts. It also has the ability to have its outputs terminated to multiple voltages. An input buffer circuit controls clock jitter. This more than doubles the available output amplitude swing and allows for the pattern generator to drive multiple logic families without changing power supply levels.

Silicon Bipolar IC

Finally, the development of high speed components in the area of optical fibre communications systems require high speed test equipment in the 20Gbps range and higher[10]. The current design utilizes a clock input range of 10GHz to 20GHz, and since it is a half rate clock system, our range is therefore up to 40Gbps. Characteristics such as the ability to have on chip RAM and a wide range of variability in the output levels and output swing would allow a low cost, versatile, high-performance solution for testing high-speed communication, mixed-signal and other circuits.

Outline

This paper describes a one-chip solution for high speed and high accuracy data pattern generation. Specifically, Chapter 2 contains the methodology used to design the pattern generator as well as future components not yet included in the current fabrication. Chapter 3 outlines multiple simulations done to test each system component and verify specifications. Chapter 4 shows measured data of the fabricated prototype chip. Chapter 5 utilizes both measurements and additional post fabrication simulations to explain future modifications needed to improve performance.

CHAPTER TWO

DESIGN AND METHODOLOGY

Circuit Architecture

Fig. 1 shows the block diagram of the pattern generator IC. The pattern data is stored in four 1024 x 128 bit SRAM, with an access time of about 3 ns. The 128 bit wide output words are first multiplexed by sixteen 8:1 CMOS multiplexers to 2.5 Gbps and then by a 16:1 HBT ECL multiplexer to 40 Gbps. The final multiplexer uses the clock to directly select the data thus reducing the maximum clock rate to 20 GHz. This puts a requirement on the duty cycle to be 50% however.

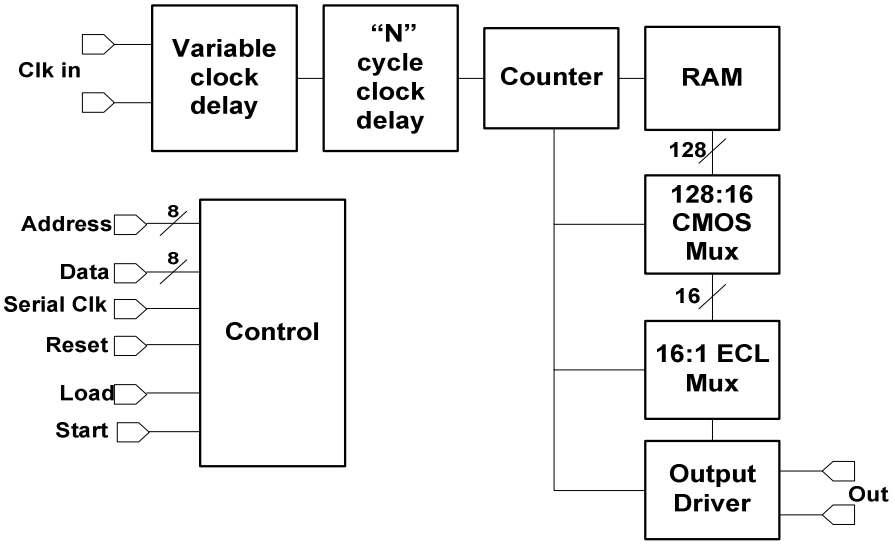

Fig. 1 Block diagram of the pattern generator IC

Fig. 2 shows the block diagram for the hybrid 128:1 multiplexer. A high speed clock is fed into a clock driver which drives the high speed bipolar components in the final 16:1 mux. This clock signal is also divided down and converted to the CMOS levels to drive the slower 128:16 mux logic.

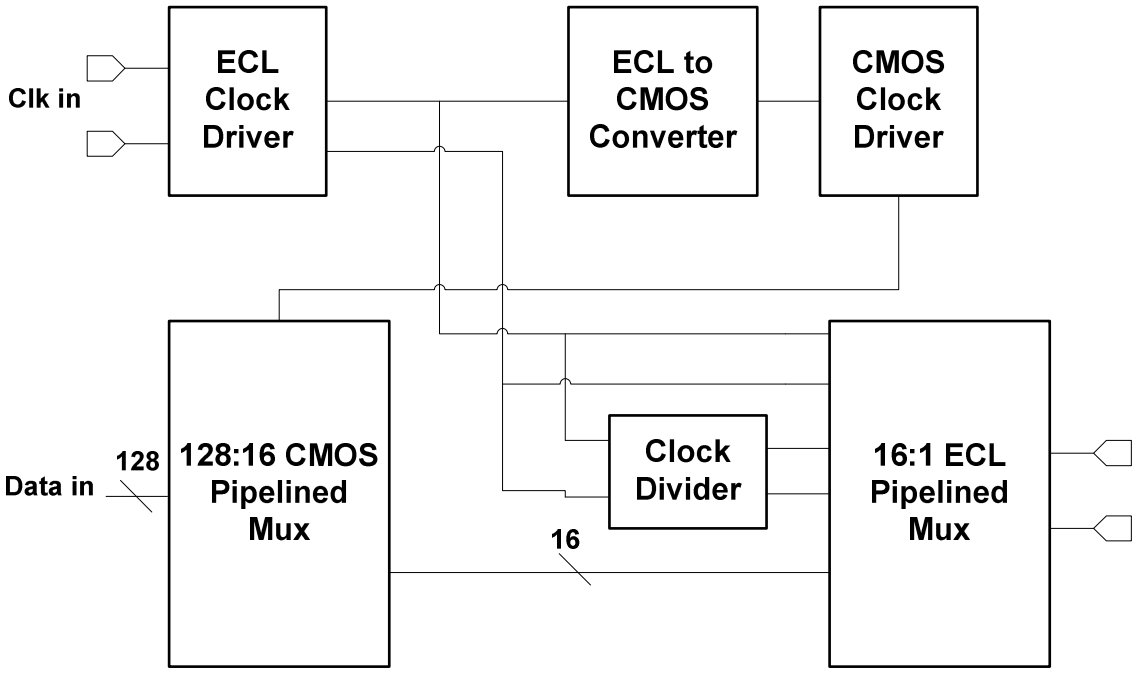

Fig. 2 Block diagram of the 128:1 Mux

The high rate data is then fed into the output driver shown in Fig. 3 which contains a variable level shift circuit, three variable high and low level current DACs, and an output differential pair. The variable level shift circuit is utilized to change the amplitude of the incoming signal driving to the output differential pair. The data dependent output jitter can be reduced with this approach. The variable level shift circuit has a current source that can vary from 0-6mA. An emitter follower stage provides adequate base current at full output voltage swing. The output voltage swing is varied by controlling a 0-60mA current source.

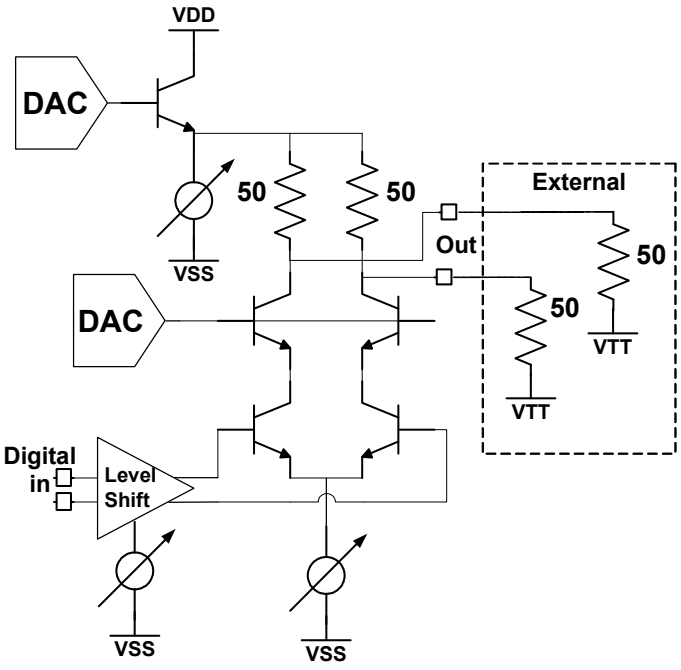

Fig. 3 Block diagram of the output driver

Fig. 4-a shows the vernier differential delay circuit which delays the clock by one to two inverter delays. Bipolar differential pairs steer the variable tail currents to either the single inverter at the top of the diagram or the load inverter at the very right of the diagram. If the top DAC was completely shut off, then the clock signal travels through two inverter paths. However, if the top DAC is completely on and the DAC on the right of the figure is completely off, the clock path consists only of a single inverter delay. By choosing intermediate current values between the two paths a differential delay can be achieved consisting of value greater than one inverter delay, but less than or equal to a two inverter delay path. The sum of the two DAC controlled tail currents must remain constant to obtain a constant signal amplitude. The vernier delay circuit provides a programmable delay of up to 8 ps in 0.2 ps steps.

To achieve the total delay goal of 50 ps two vernier circuits were placed in series and four additional paths were introduced which consisted of 0, 2, 4, or 6 series inverters. The two vernier circuits in series allow for larger variability, while the inverter delay paths allow for longer delay values without having to control more than two vernier DAC registers. This method gives much longer delay values with far less power dissipation. The specific circuit configuration is detailed in Fig. 4-b.

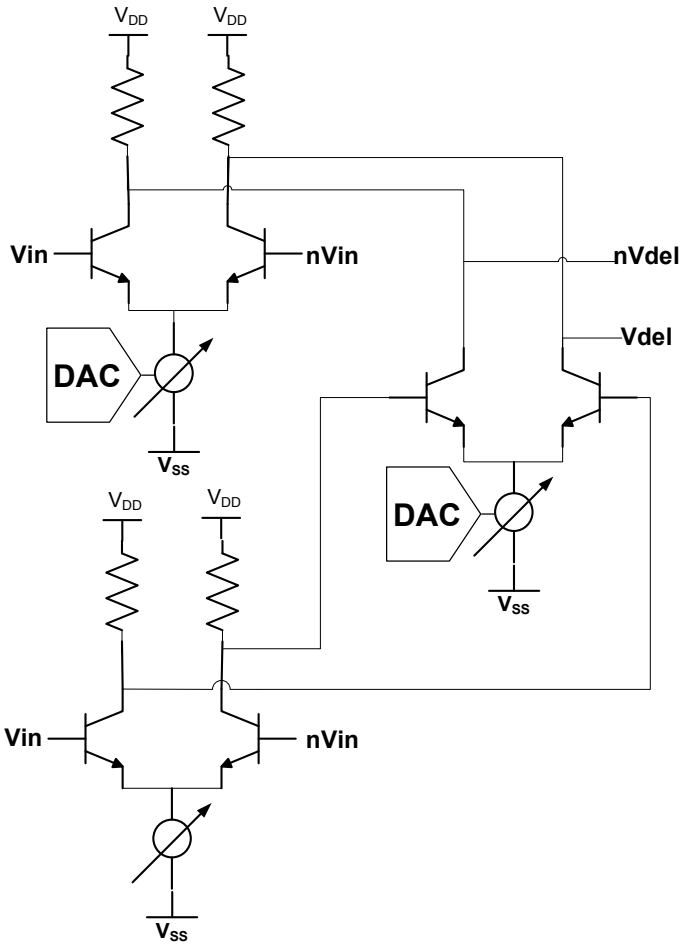

Fig. 4-a Block diagram of the Vernier Delay Circuit

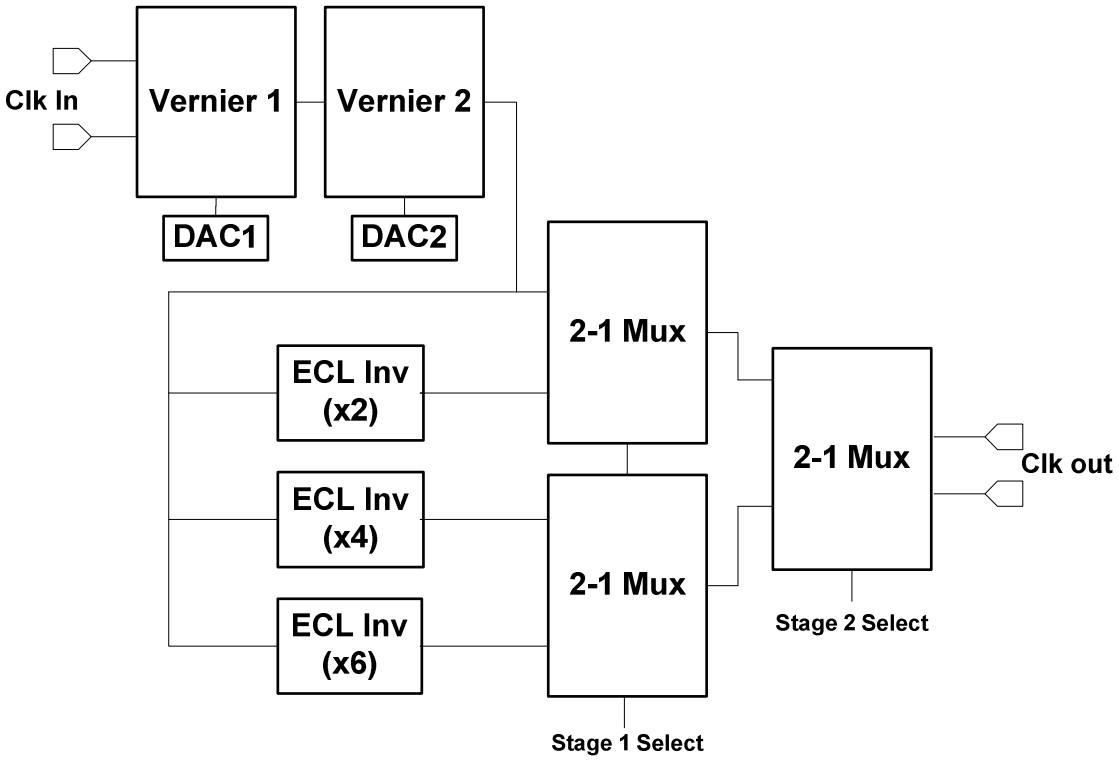

Fig. 4-b Block diagram of the Inverter Delay Circuit combined with Vernier Delay

Fig. 5 shows an additional delay circuit which extends the clock delay by a programmable number of up to 255 clock cycles. An 8-bit counter is loaded with the desired number of clock cycles to be delayed and then counts down to zero. The detection logic block enables the clock feed-through logic to allow the half rate clock to pass on to other circuits. This circuit has been designed, simulated, and laid out, but was not included on the prototype due to lack of area.

Therefore, only the circuit structure in Fig. 4-b was implemented on the prototype chip.

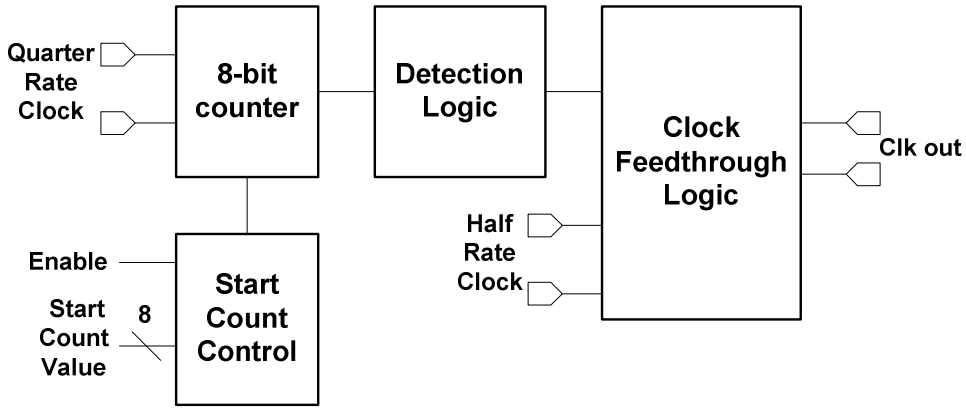

Fig. 5 Block diagram of the "N" cycle clock delay circuit

RAM Speed

The SRAM is organized as 1024 columns of 128-bit words. Since the output word width of the SRAM is large, the 8-bit counter that addresses the columns is only required to operate at 300 MHz. The counter can perform looping to effectively extend pattern length. The counter is initially reset and after the start signal is enabled, it will continue until the counter output is equal to the variable end register. The counter will then jump to the variable start register and repeat until reset is enabled.

The counter that addresses the RAM and multiplexers uses a combination of SiGe bipolar and CMOS logic. The high speed clock is first divided using bipolar flip-flops and then converted to CMOS levels when speeds are low enough. In order to provide lower data rates without duplicating data in the memory, the CMOS clock that drives the SRAM column counter can be divided by 1, 2, 4, or 8.

The prototype chip does not contain the SRAM or its control circuitry. A 128 bit shift register line was constructed to load data and control values. This shift register implementation shows the feasibility of the pattern generator due to the fact that it is close enough to the RAM implementation from a speed point of view.

RAM Controller

The purpose was to design a flexible RAM controller that would control a minimum of 512MB of RAM with functions such as looping and partial column jumps. This methodology was designed in verilog to verify functionality. Fig. 6 shows the data flow control of four different SRAM banks into the 128 bit wide input of a single channel data pattern generator.

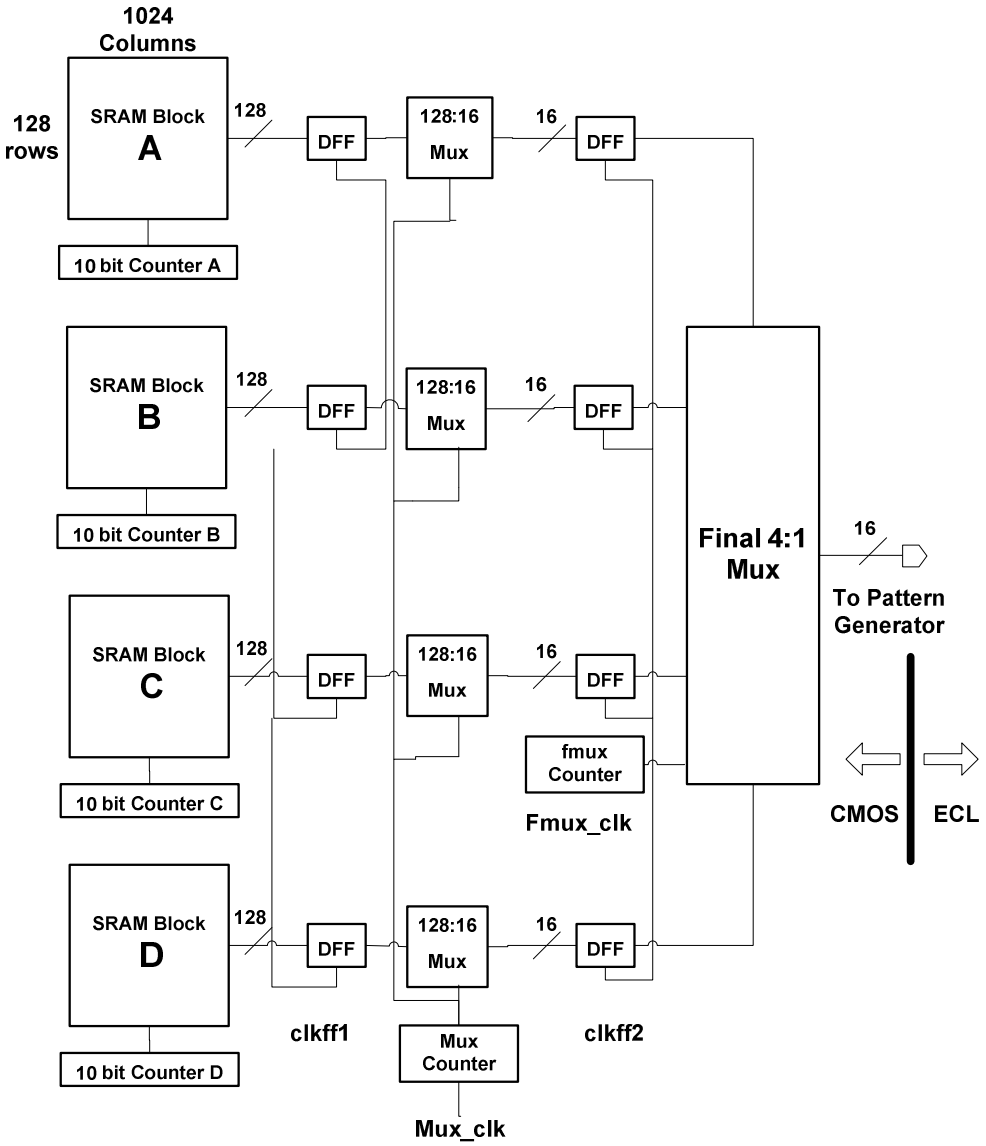

Fig. 6 Block Diagram of Data Path

The components of the system include 4 RAM blocks which have 128 rows and 1024 columns. Blocks A, B, and C are used to store the pattern data, while block D will is used as a copy register in the event that a partial column jump is desired. Multiple 10-bit counters control which column is allowed to write or read while each RAM block's 128 outputs go to a 128:16 multiplexer. This 128:16 multiplexer is controlled by a 4-bit mux counter. These multiplexers select 8-bits at a time and pass the data to the final 8-bit wide 4:1 multiplexer, which determine which RAM block data should be allowed to proceed to the bipolar multiplexer.

Data pattern generators to test communication circuits usually use linear feedback shift registers to generate pseudo-random sequences. Therefore, the bit sequences are over a million bits long. 500kb of on-chip SRAM may not be adequate for these applications. To achieve a longer pattern techniques such as looping and jumping can be employed. Initially, a user may input into table entries the start address value, stop address value, and number of cycles to complete before jumping to the next table entry start value. Jump values need to be considered depending on which RAM bank they are allowed to come from or go to. In addition control logic is needed to properly send the data out of the correct RAM block. Two methods of control can be used. Either the addressing scheme can be controlled to switch between the wanted data values, or the addressing scheme can be made less complicated and the output data can be controlled. This scheme uses a data control concept in which multiple data blocks can be sending out data at the same time and multiplexers control which data is allowed to pass until all of the data choices are reduced to one final correct 8-bit word.

The basic operation of the system is that the user will enter information in 4 table entries. The number of table entries can be easily increased if needed. The user will enter for each cycle the start address, stop address, and number of cycles. It is assumed that software will then place the data in memory depending on whether partial or full column jumps are required. The responsibility of this code is to multiplex the data out correctly. The start register is 12 bits. Two bits are dedicated to which RAM block the start value is in, and the other 10 bits will address which column in that block should be addressed. The stop value register is 16 bits. Four of the bits determine which row contains the last of the data, 2 bits determine the stop block and the other 10 bits determine what column. The stop register would only need to be 14 bits if there is a restriction that the start and stop register values must be in the same block.

The basic operation of the code first initializes all counters and clock signals until an enable signal is present. This code also determines when to switch from the previous table entry to the next. In addition, for each table entry partial or full column stop values are calculated to allow for the correct section of code to be activated. These separate sections of code are then executed until all of the table entry values have been completed. More detailed information of the specific operation of the initialization code, partial column jumps and full column jumps is located in the beginning of Appendix A titled "Detailed Operation."

Test Architecture

Shift registers were implemented to load all of the needed DAC values for testing. The shift register path is shown in Fig. 7. SR1 consists of only two bits which choose a particular delay path. As mentioned above, there are four delay paths which can be chosen to achieve up to 50ps of clock delay. These two bits control an equivalent 4:1 multiplexer which consists of three 2:1 multiplexers seen in Fig. 4-b. SR2 and SR3 load the 8-bit DAC registers for each of the two vernier delay circuits. SR4 controls the bias voltage for a cascade pair of bipolar transistors on the output differential pair of the output driver. When the high level is lower than -1V these values must be changed in order to not violate any breakdown voltages and also leave enough headroom for the differential input pair. SR5-SR8 load the four variable DAC registers for the output driver to set the high level, low level, output swing, and jitter reduction. SR_D consists of the 16 8-bit registers which load the data inputs of the pattern generator.

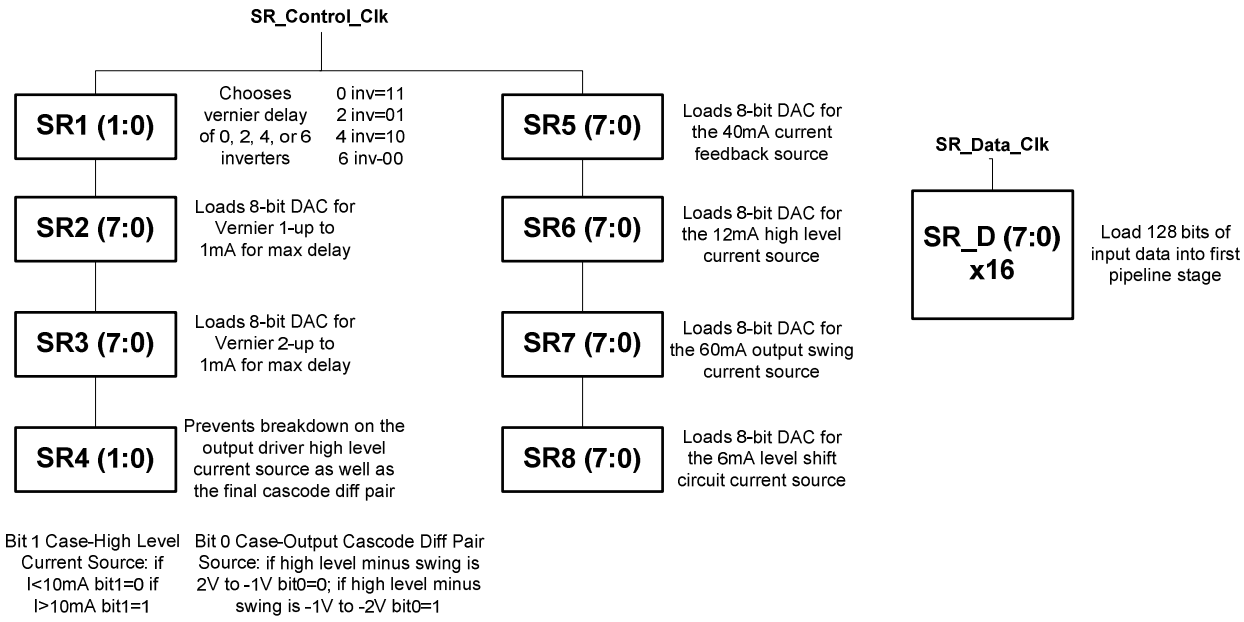

Fig. 7 Block diagram of the Shift Register Path

CHAPTER THREE

SIMULATIONS

Output Driver

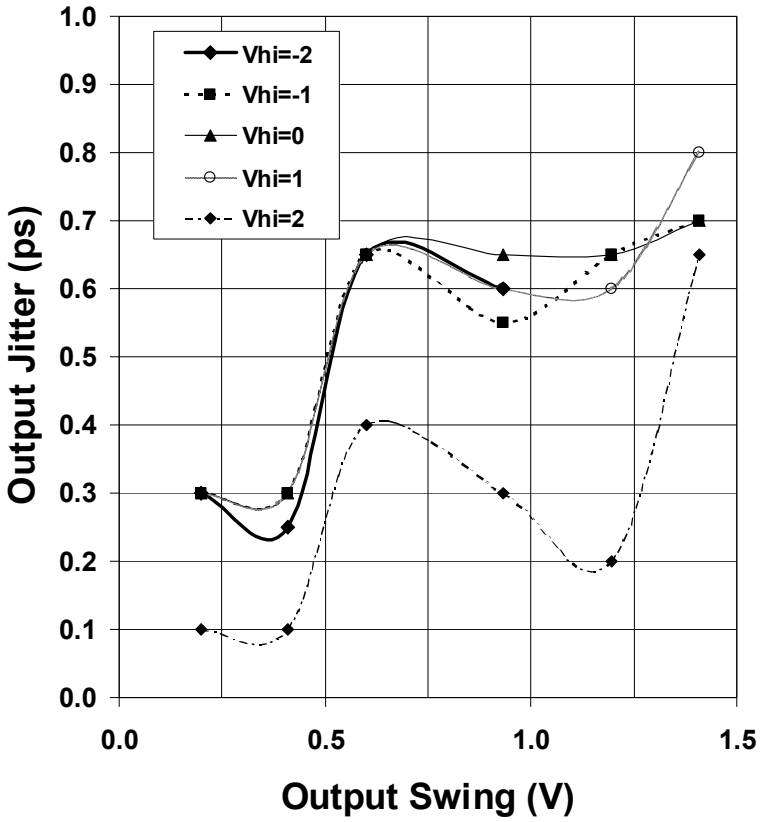

Fig. 8 Output Jitter vs. Voltage Swing

Fig. 8 shows the data dependent jitter simulations of the output driver at different high level values and output swings. At each high level and output swing value, the level shift circuit amplitude was modified to achieve the lowest possible jitter. It was seen that the data dependent jitter was well under the 1ps specification.

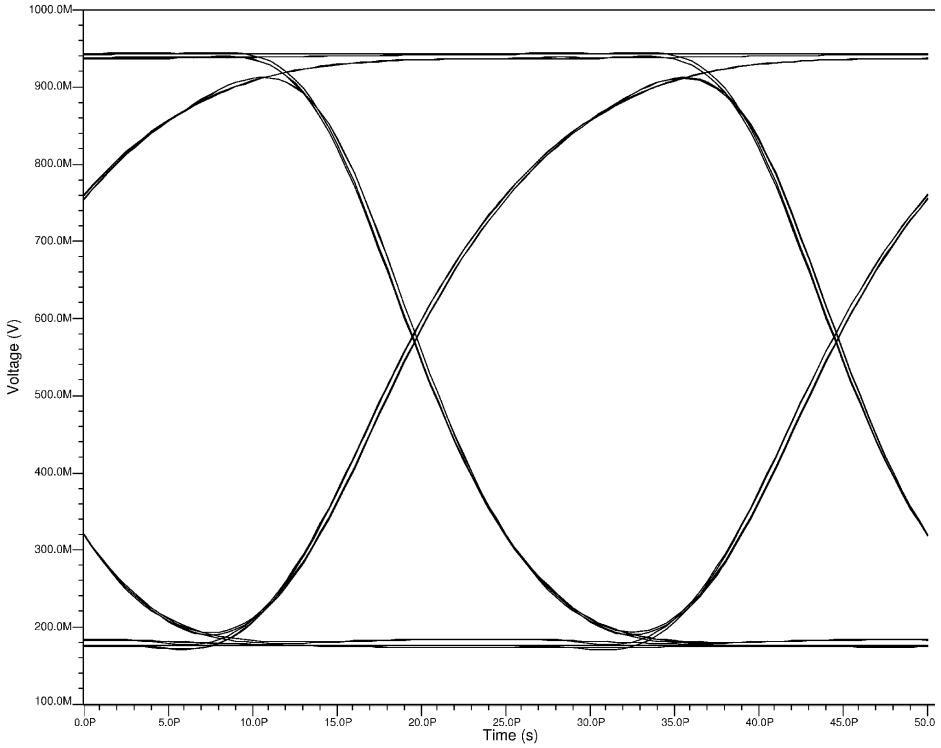

Fig. 9 Simulated eye diagram at 40 Gbps and an output amplitude of 0.75V.

Fig. 9 shows an example eye diagram. The eye diagrams were used to measure the data dependent jitter at the threshold point with an input data pattern of 1.5ns length at a rate of 40Gbps.

Vernier Delay

Table. 1-Inverter Chain Delay Values

	del_0or4	ndel_0or4	del_low	ndel_low	Total Delay	Relative Delay
Zero Inverters	1	0	1	0	5.25152E-11	0
Two Inverters	0	1	1	0	6.65217E-11	1.40065E-11
Four Inverters	1	0	0	1	8.28986E-11	3.03834E-11
Six Inverters	0	1	0	1	9.84058E-11	4.58906E-11

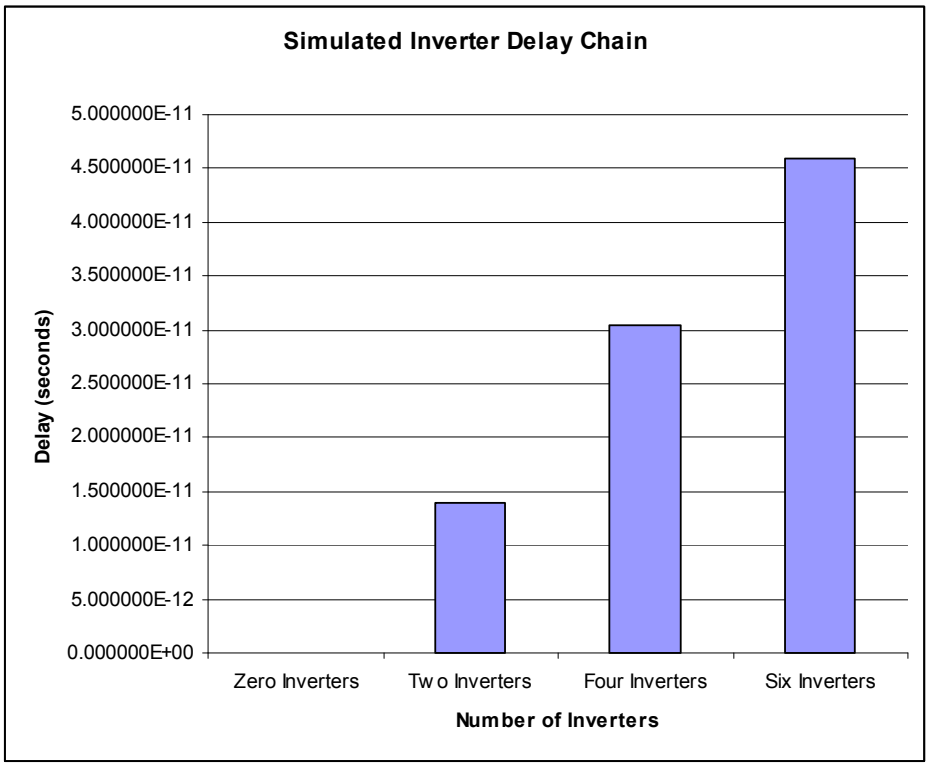

Fig. 10 Inverter Chain Delay Plot

Table. 2-Vernier Delay Values

Input Code	Vernier 1	Vernier 2	Vernier 1 & 2	Vernier 1 Rel	Vernier 2 Rel	Vernier 1 & 2 Rel
11111111	5.725E-11	5.70E-11	6.370E-11	4.742E-12	4.527E-12	1.119E-11
10000000	5.338E-11	5.327E-11	5.532E-11	8.719E-13	7.647E-13	2.807E-12
00000001	5.278E-11	5.278E-11	5.295E-11	2.653E-13	2.653E-13	4.362E-13

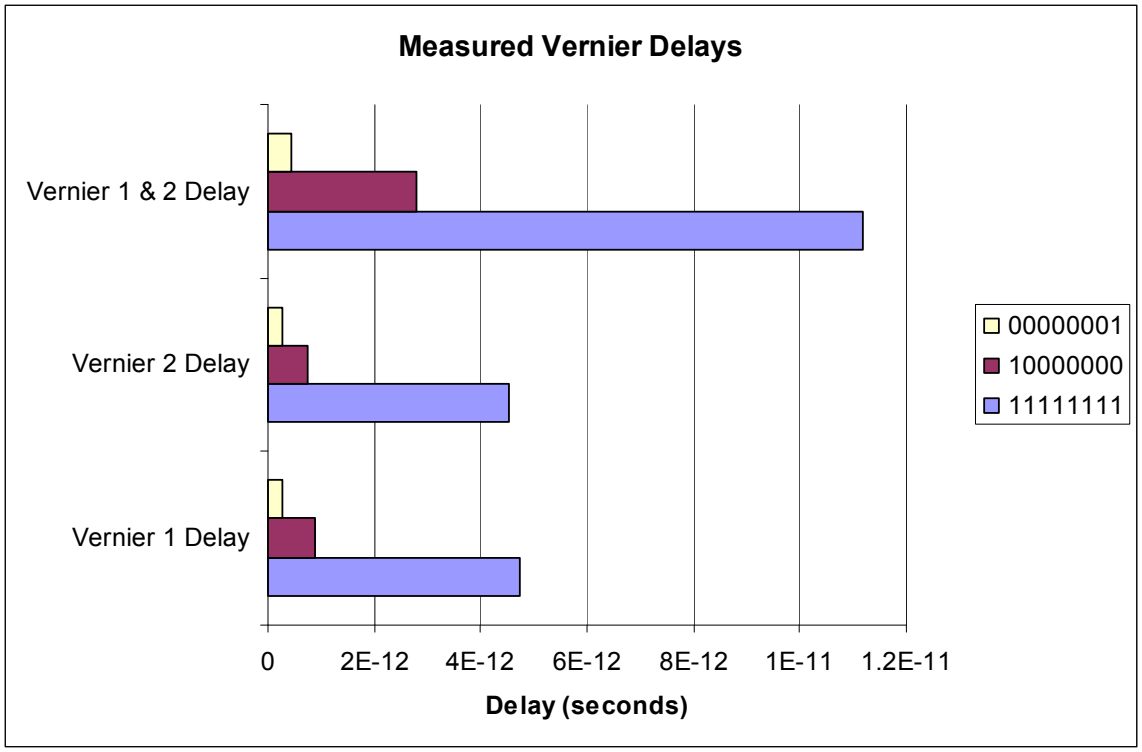

Fig. 11 Vernier Chain Delay Plot

Simulations of the individual vernier circuitry as well as the inverter chain paths in Fig. 4-b were completed. The intrinsic delay path was calculated as the path through Fig. 4-b with both vernier circuits at their minimum delay value as well as choosing the additional zero inverter path. This delay is recorded in Table 1 as approximately 52.5 ps. Each subsequent Total Delay value was then simulated. To find the relative delay of each path approximately 52.5 ps was subtracted from it. In addition, the combination of the vernier 1 and vernier 2 were measured. This information gave linearity as well as expected delay values to compare to measurements taken from the prototype chip. The total delay range when combining the inverter chain and the vernier delay circuits was simulated in the range of 0ps to 57.08508ps.

RAM Controller

The following simulation is a functional simulation for a full column stop values.

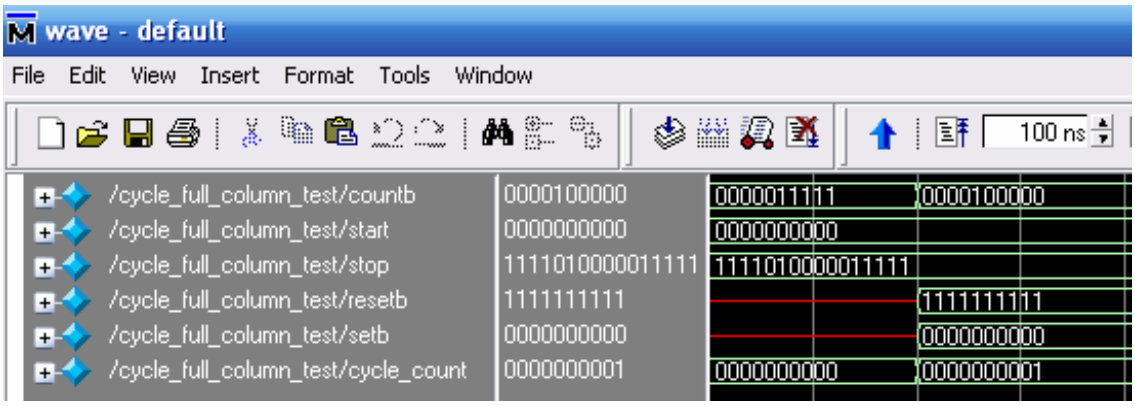

Fig. 12 Verilog Simulation of Full Column Code

This simulation shows the basic function of full column operation. The four most significant bits(1111) show that the stop value is at the end of the column and therefore a full column stop. The next two bits(01) determine which counter values to look at; in this case RAM block B seen in Fig. 6. The final ten bits of the stop value(0000011111) is the stop column at which the specific table entry sequence should end. The counter output of the counter which controls the addressing of RAM block B(countb) counts until it reaches the final ten bits of the stop value. Upon the completion of that bit code setb and resetb are triggered to start counter B back at the start value. Due to the fact that the start value is 0000000000 all ten bits of resetb are triggered and no bits of setb are to reset the counter back to the start value. The signal cycle_count is then incremented to show that one completion of the table entry cycle has been completed. This value would then be compared to the wanted cycle value and if equal, the transition to the next table entry would activate. During this sequence the control of the multiplexers becomes very simple and would be based on the stop value code for which RAM block to allow the correct data to pass.

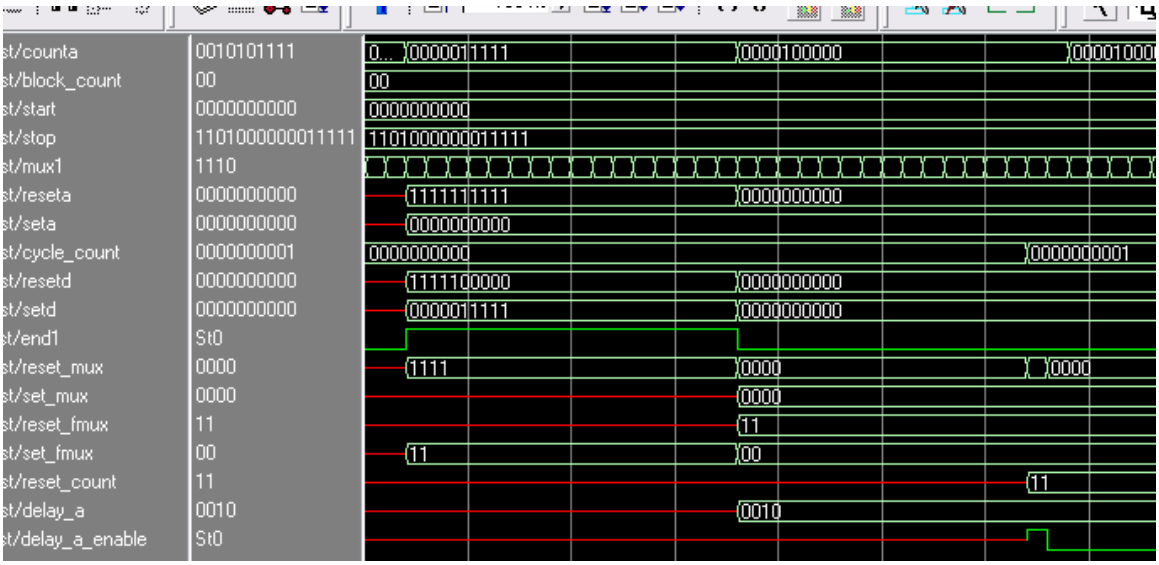

Fig. 13 Verilog Simulation of Partial Column Code

This simulation shows the operation of a partial column code. The signal block_count is used to determine which block section the code should be looking at. In this case 00 means block A. When the column counter for block A has read the last full column in the sequence, the block counter for block A is reset to the start value. This event can be seen in Fig. 13 when signals reseta and seta are triggered. This will allow the start value and SRAM block D value to be read at the same time. The code then detects the stop row of the data in SRAM block D and then resets the mux values to go back to the start value in SRAM block A. The control will then trigger the signal delay_a_enable to determine the counter delay so that the entire column can be multiplexed completely. The control signals for the multiplexers follow the same pattern as the full column stop logic. In Fig. 13 the mux control can be seen switching to SRAM block D(set_fmux=11) and then back to SRAM block A(reset_fmux=11) when the extra data stored in SRAM block D has been read. More detailed information on the partial column code operation is located in Appendix A.

CHAPTER FOUR

MEASUREMENT DATA

Measurement data was taken using the TDS8000 sampling scope. Shift register read and write functions were executed by a Velleman test board. Level shift circuitry was designed to shift the Velleman board values into the shift register data and clock lines. DC probes provided the additional 4V supply for the output driver as well as for the low speed shift register lines. The vernier circuit delay output was measured separately from the data output using the crossing point measurement. The output driver variations were measured using the maximum and minimum scope values. GPIB recorded the measurements after adequate settling time had passed. In addition, the shift register control data was verified during simulation time. Example GPIB programs are contained in Appendix B. All high frequency outputs were measured using RF probes. SMA cables connecting the RF probes to the TDS8000 are rated for frequencies up to 3GHz.

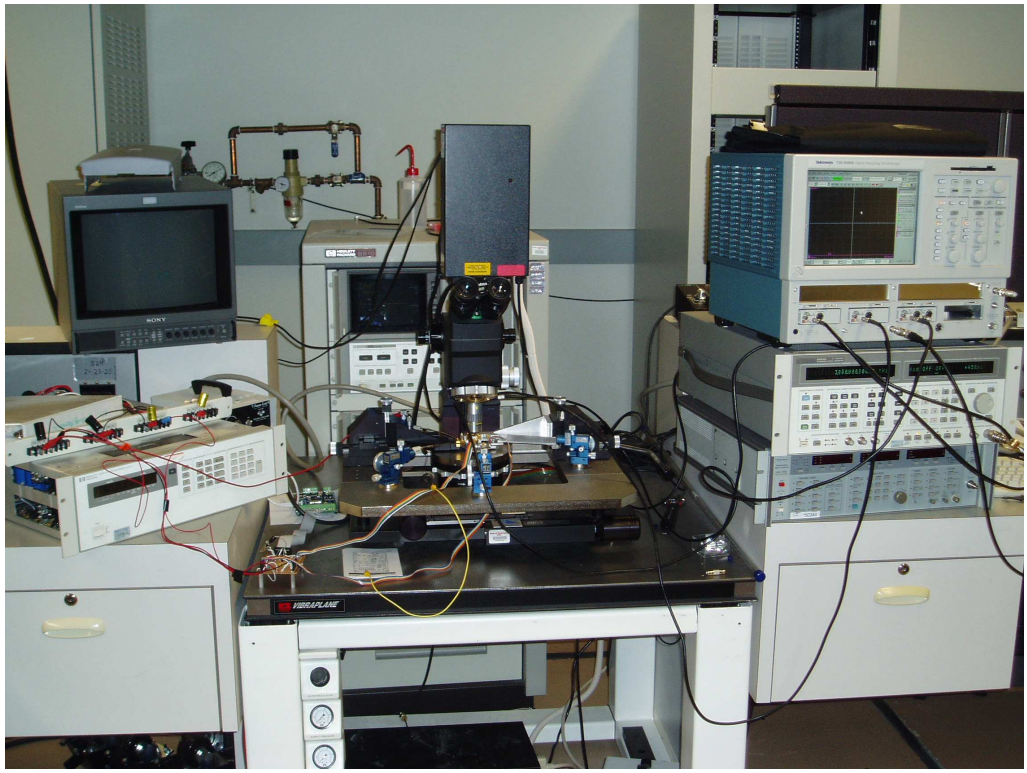

Fig. 14 Picture of Test Setup

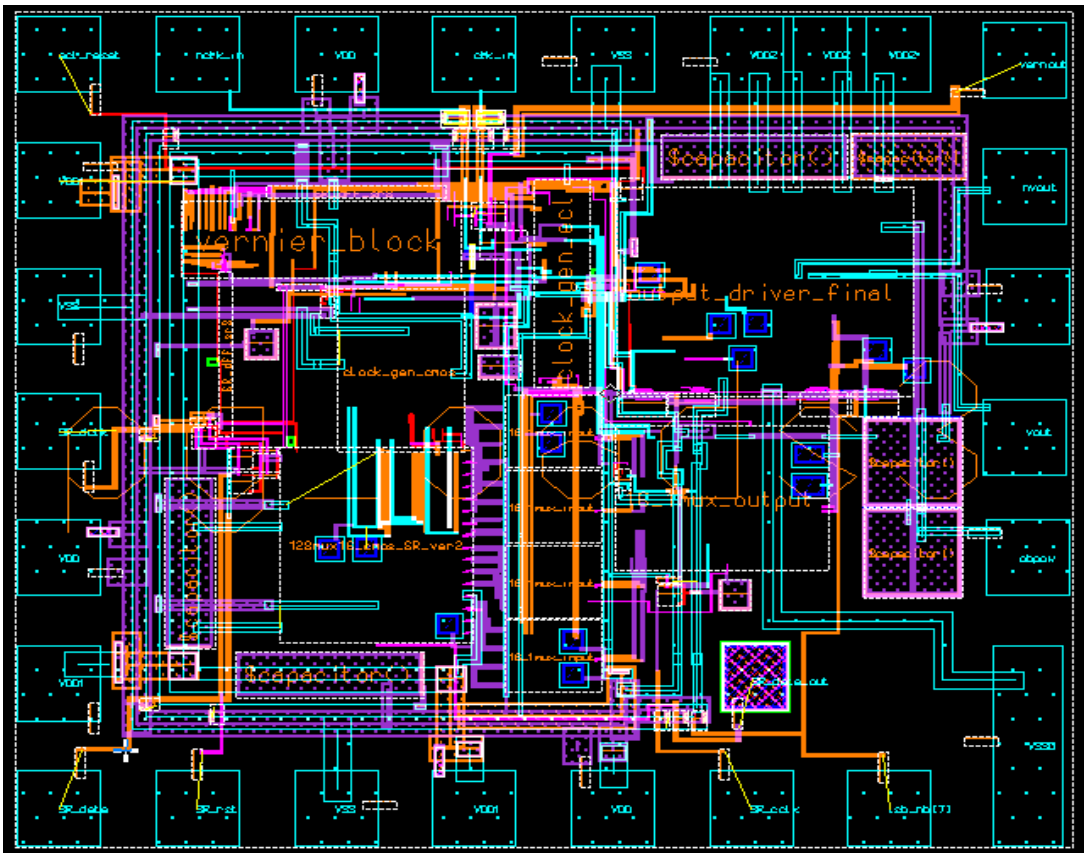

Fig. 15 Prototype Layout 1.131mm x 1mm

Output Swing Measurements

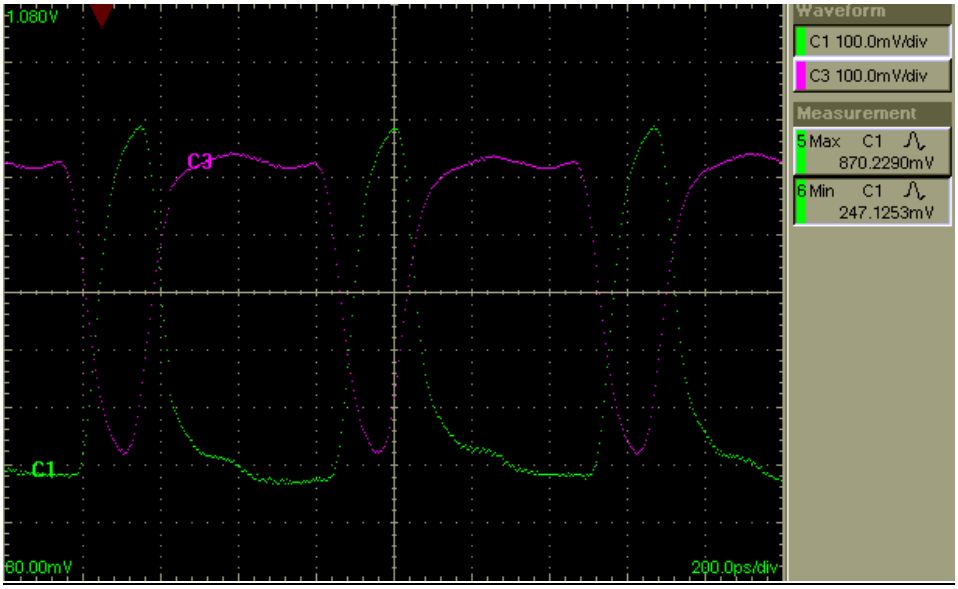

Fig. 16 Pattern Generator Differential Outputs

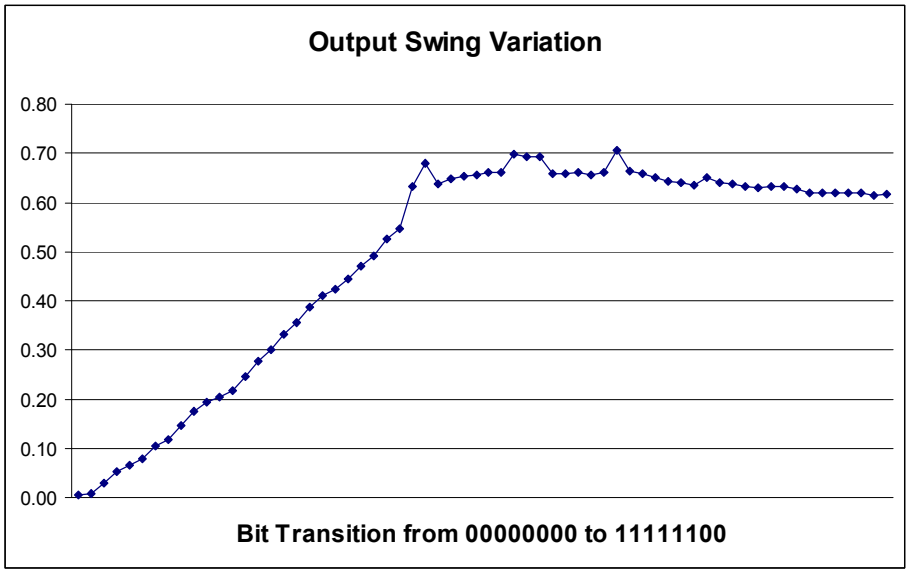

Fig. 17-a Output Driver Output Swing Chip 1

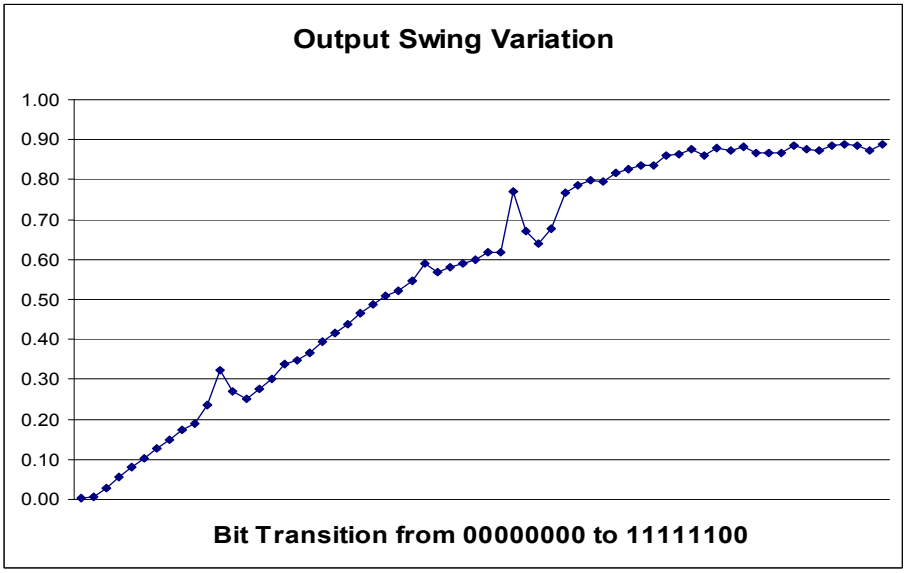

Fig. 17-b Output Driver Output Swing Chip 2

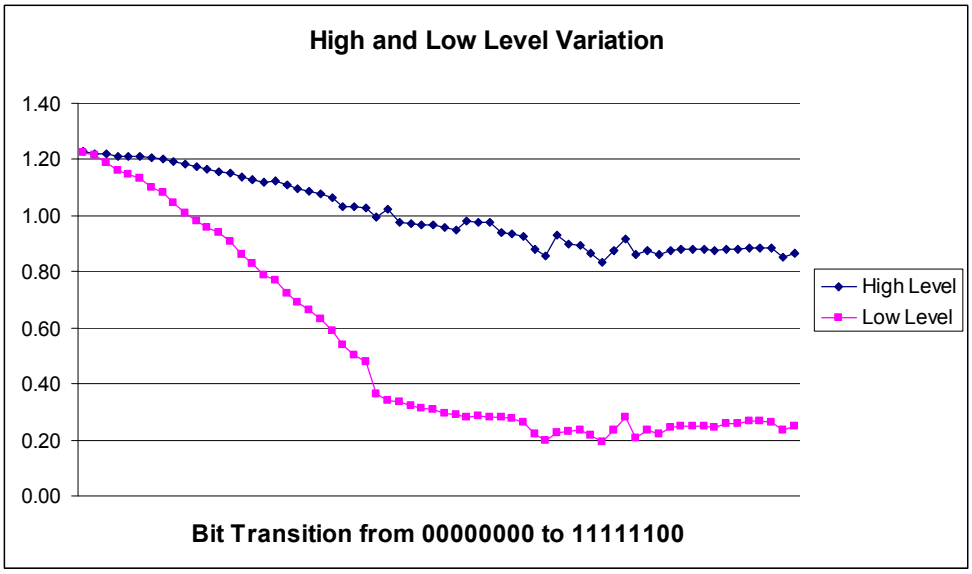

Fig. 18-a Output Driver High and Low Level Chip 1

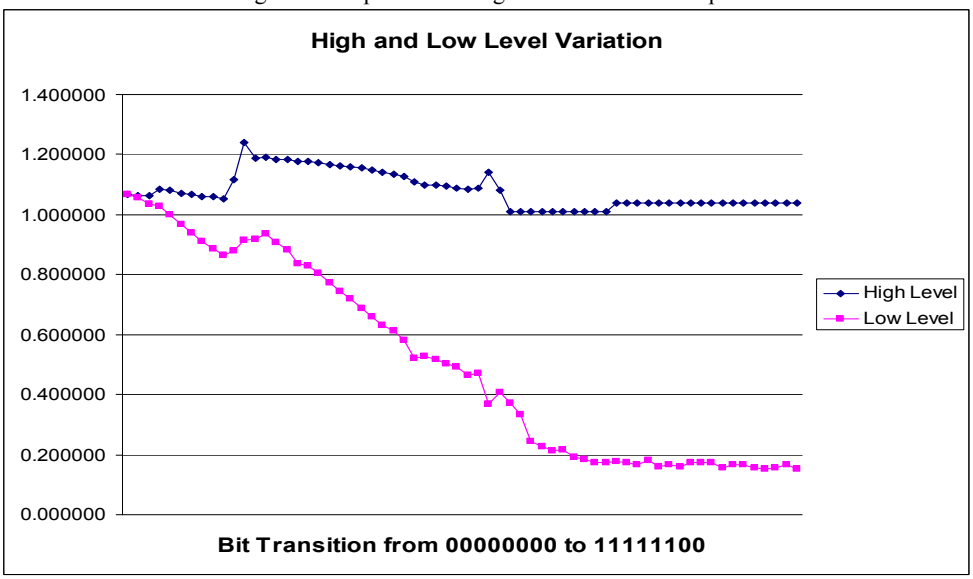

Fig. 18-b Output Driver High and Low Level Chip 2

Measurements were taken at 3GHz to measure the variation of the output driver variables. The control of the output driver voltage swing obtained values that were half of the expected values with both of the outputs terminated to 50 ohms and with ground as the termination voltage. A max voltage swing of .75 volts and .88 volts was reached approximately halfway and three quarters through the binary sequence respectively. The swing should have continued to the simulation expectation of 60mA into an equivalent 25 ohms or 1.5V. This effect could be due to malfunctioning of the current mirror,

temperature, parasitic inductance associated with the test setup, as well as the output swing bias current not being switched positive to negative as fully as expected due to a clock error in the data path which is explained in Chapter 5. In addition, further temperature and power supply variations are looked at in more detail in Chapter 5. The layout of the output driver also could have a contribution. Specifically, the distance of the bias current to the differential pair, and the distance between the differential pair outputs and the measurement pads. With further testing it was seen that the current level change in the power supplies did not reflect the expected current change. Since all of the bias current change would be seen in the -4V supply, the current levels at each end of the binary sequence were measured.

Table. 3-Measured Current Levels during Output Swing Test

	Bit Pattern=00000000	Bit Pattern=11111100
Voltage Supply	Current Measured	Current Measured
4V	28.3mA	51.5mA
-2.2V	11mA	11.55mA
-4V	230mA	260mA

The -4V currently supply changed by only 30mA, or half of the wanted current. This deviation is consistent with half of the expected voltage swing being obtained. This information then points to the DAC circuit malfunctioning or bias voltage variation. This could be due to a parasitic IR drop of the bias voltage which feeds the binary weighted transistors, resistor variation in the bipolar current mirror stage due to temperature or power supply variation due to measurement cable inductance or IR drops.

In addition, the high level did not remain at its peak value. The high level voltage is created by using the 4V supply and two corresponding diode drops. The diodes are simple bipolar transistors which due to the increase in output current have an increased base to emitter voltage drop.

High Level Measurements

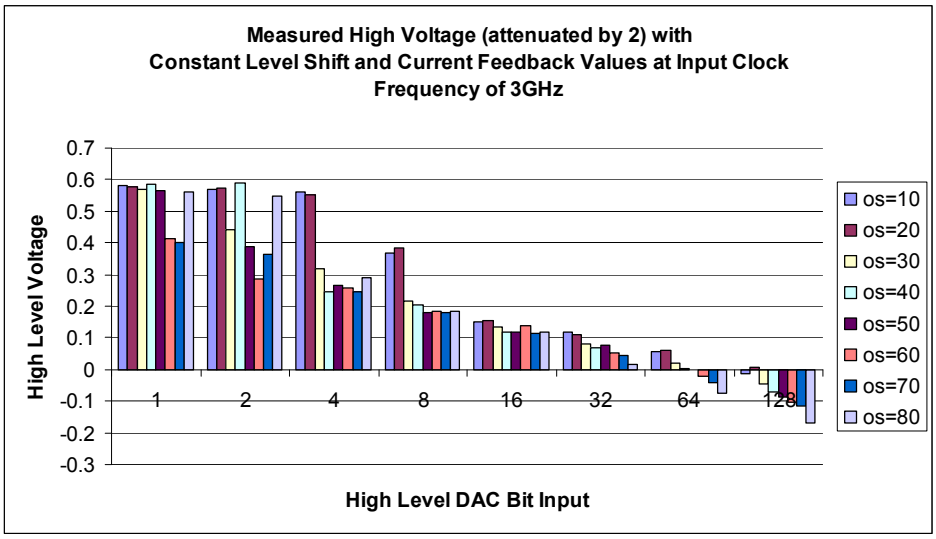

Fig. 19 High Level Output Voltage Measurement at Different Output Swing Values(os given in hex format)

The high level control of the output drive seemed fairly constant when using multiple output swing values. But, the linearity of the different DAC values did seem too dependent on the level of the output swing. This problem points again to the current mirror topology, which in the case of the high level set circuitry is equivalent to the output swing circuit. Minor oscillations were also noticed depending upon the value of the high level DAC. The most common bits where this occurred were the lower three. By changing the output swing the bias point of the output also changes and could therefore affect the output pattern as well as increase the chances of some instability.

In some cases the high level voltage would decrease dramatically when the input to the high level DAC reached hex 08. Looking at the current sourced by the power supplies when this crashed occurred, it was observed that the current through the 4V supply dropped dramatically (38mA to 25mA), even though the high level lowered. Current levels measured in the other two supplies remained constant. This indicates a non ideal current path. The circuit diagram shows the same basic current mirror that is used in the output swing current source. There is also a high voltage bipolar to help with

breakdown issues due to the fact that the voltage swing at the negative end of the resistor can be from 4V to -1V.

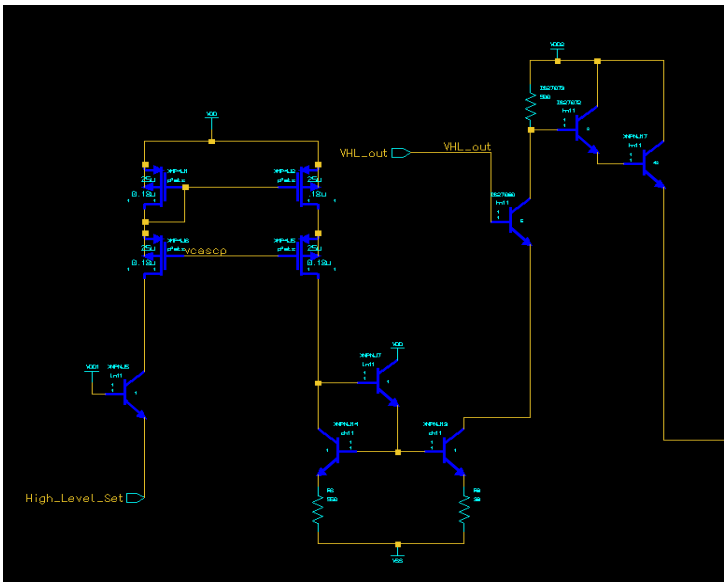

Fig. 20 High Level Set Circuitry

Current Feedback Measurement

The feedback current source, which is used for the purpose of helping to set the high level when the desired high level is less than the termination voltage, was tested to determine if the correct amount of current was being created. This source simulated to have a range of 0-40mA. By measuring the power supply current a change of only 26mA was seen after counting though the corresponding DAC values.

Level Shift Measurement

Due to the clock routing error, the level shift circuit could not be fully tested. However, its effect on the output swing did give insight to the idea that perhaps the output voltage swing was not reaching its full value because the current was not being fully switched in the final differential pair.

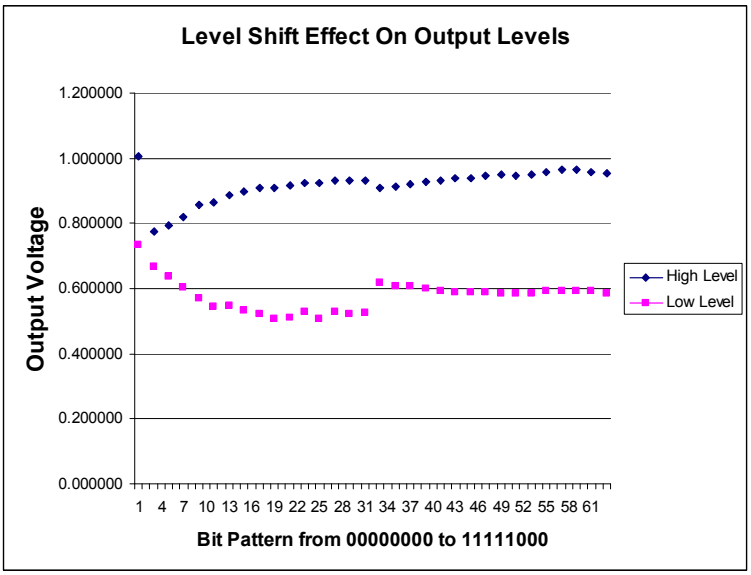

Fig. 21 Level Shift Effect on Output Levels

Vernier Delay Circuit Measurements

The direct output of the vernier delay circuit was measured at 3GHz. Both the response of the four inverter chain paths as well as the response of the individual vernier blocks was measured. A bias T was used to set the termination voltage to approximately -1.6 V. When terminating the output through 50 ohms to ground the output of the vernier circuit was connected equivalently to Fig. 22. With a bias current in the emitter follower of 1mA, Vout was only allowed to swing as low as -.1V. This configuration did not allow Vout to swing beyond a base-emitter drop of the input signal. Setting the termination voltage to a -1.6V, the emitter follower was allowed to drop far enough to see a significant voltage swing at Vout.

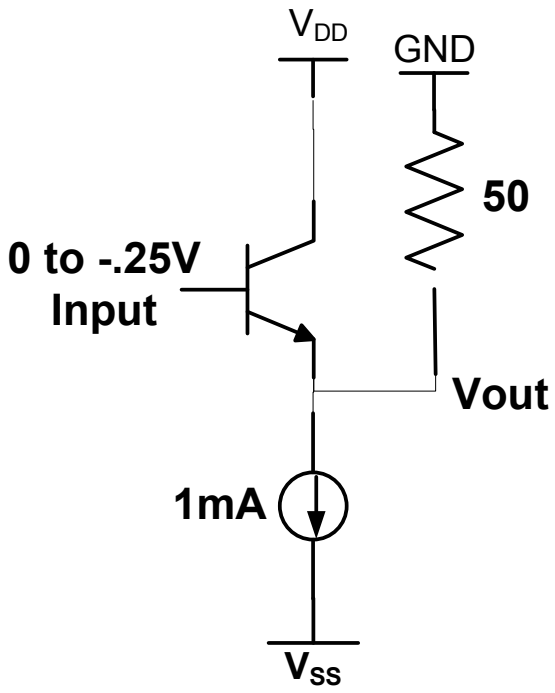

Fig. 22 Schematic of Vernier output before Bias T

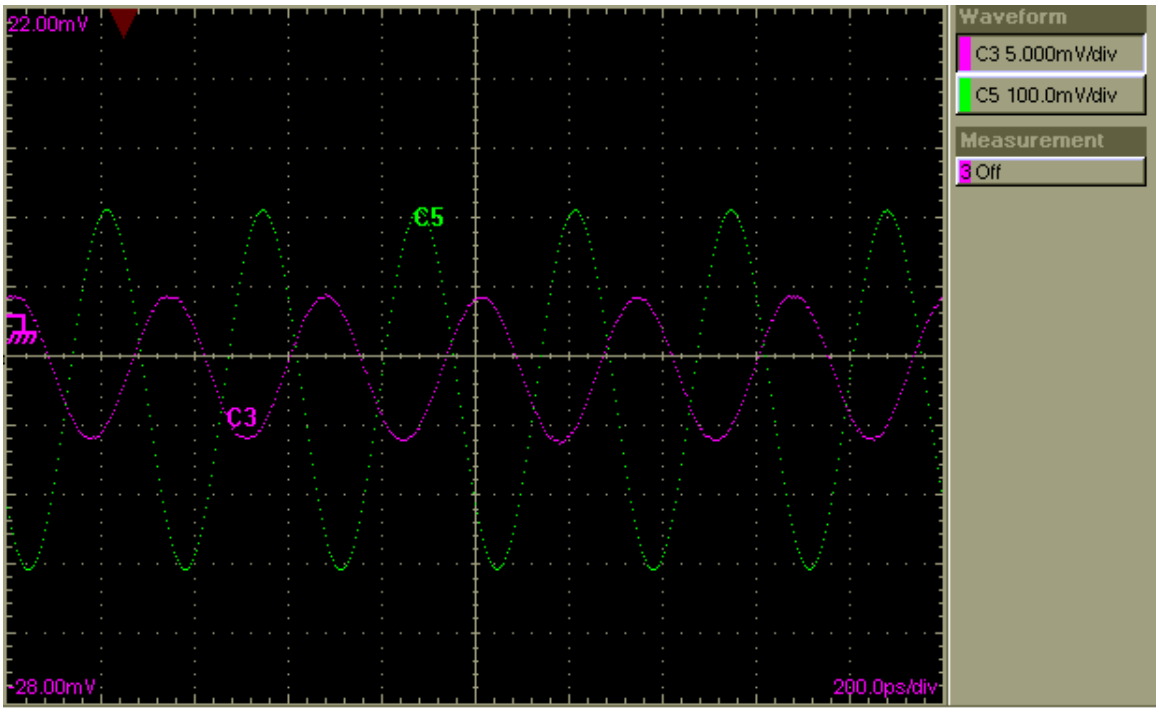

Fig. 23 3GHz clock Input(Green) and Vernier Circuit Output(Purple)

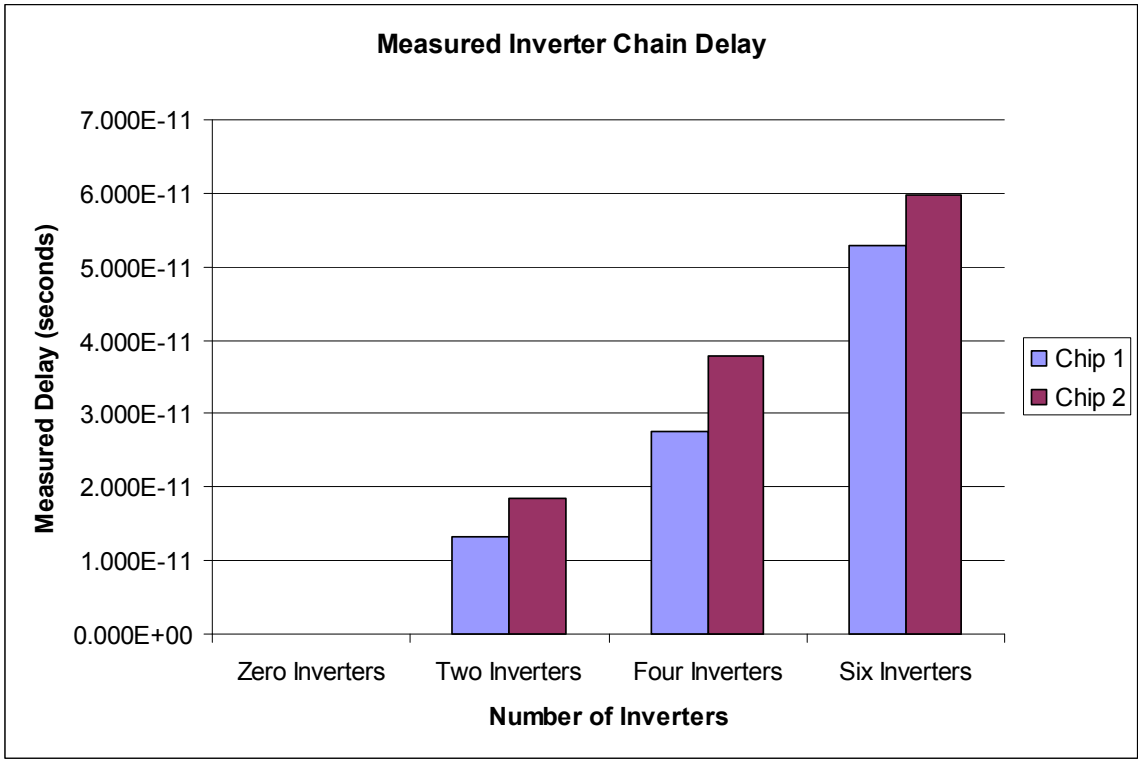

Fig. 24 Measured Inverter Chain Delays

The measurements were made using the positive crossing point measurement of the sampling scope. Adequate settling time was given, and relative accuracy was seen to be within 1ps or less. The inverter chain measurements matched up very closely to the simulated values.

Table. 4-Simulated vs. Measured Inverter Chain Delay Chip 1

	Measured	Simulated	% Error
Zero Inverters	0.000E+00	0.000E+00	0.00
Two Inverters	1.328E-11	1.401E-11	5.19
Four Inverters	2.768E-11	3.038E-11	8.90
Six Inverters	5.289E-11	4.589E-11	-15.25

Measurement of the individual vernier block showed expected behavior, but with multiple linearity problems and poor reproducibility. The linearity could be an issue with the timing of the scope due to the fact that the total inverter delay is only expected to be around 6-8ps in simulation. Other contributing factors to the response of the vernier are jitter from the clock source, slew rate limiting, and the 10mV amplitude of the signal itself.

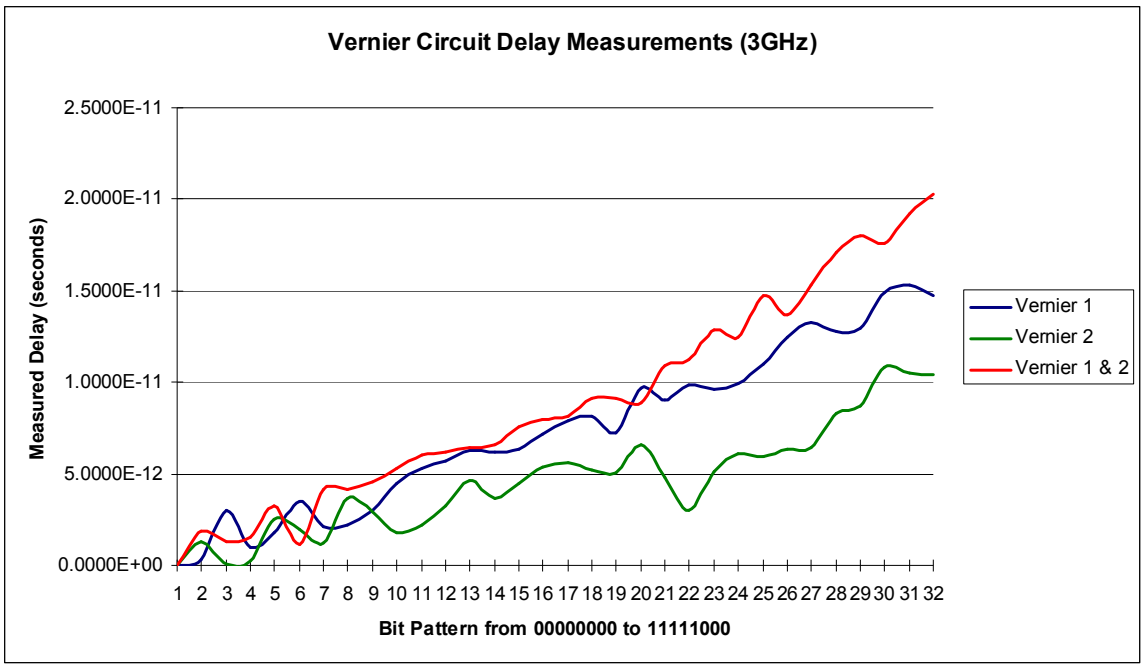

Fig. 25 Measured Vernier block delays

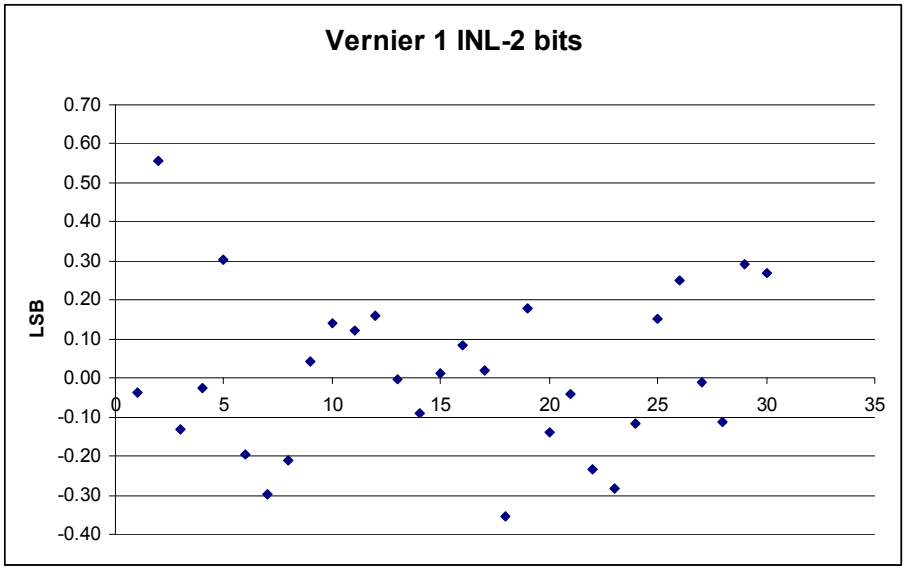

Fig. 26 Vernier 1 INL

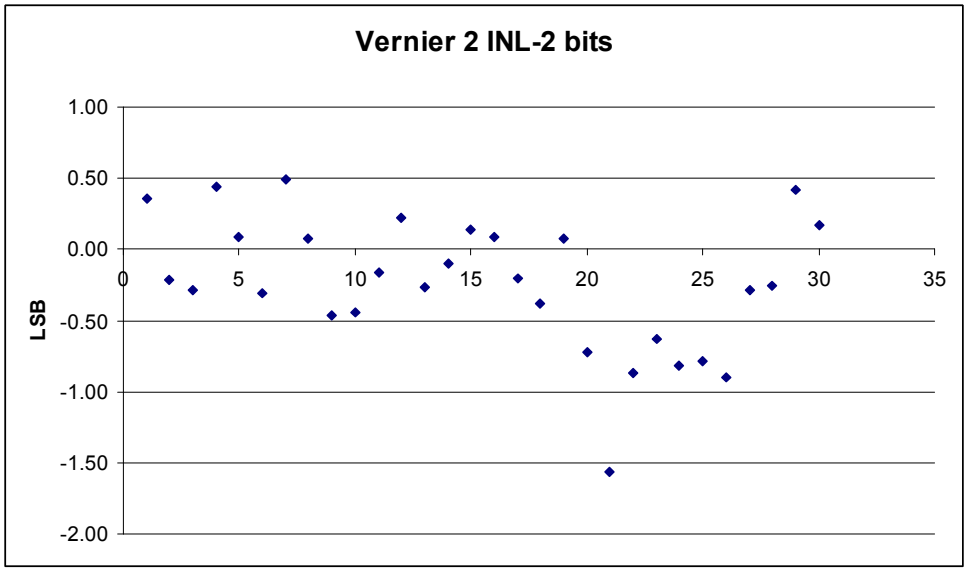

Fig. 27 Vernier 2 INL

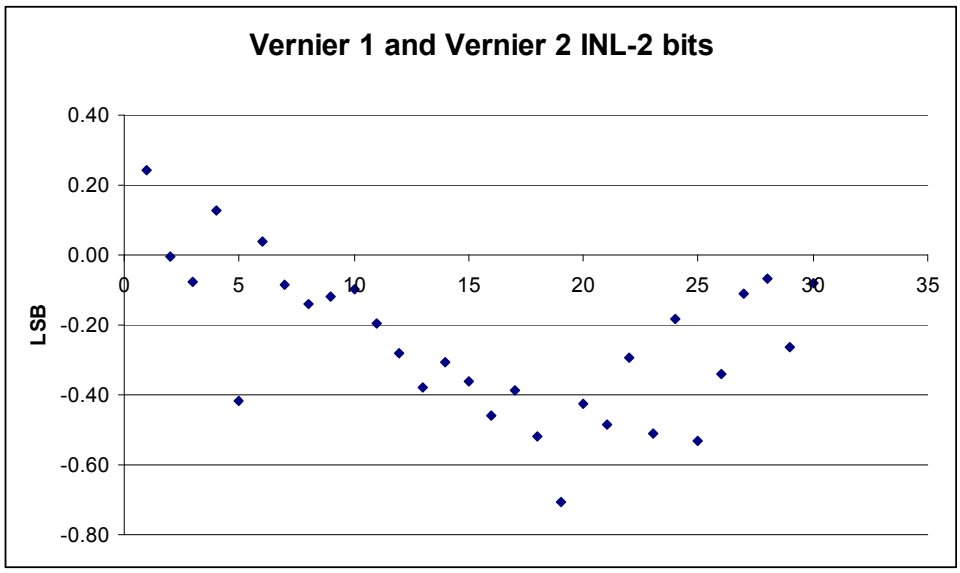

Fig. 28 Vernier 1 and Vernier 2 INL

The INL measurements show that either of the vernier circuits alone, or combined may have 8-bit resolution, but only 2-bit accuracy to approximately .5 LSB. Simulations also indicated that the circuit topology itself does not have high bit accuracy. This fact can be overcome and modified due to the higher resolution to be able to initiate the wanted delay.

CHAPTER FIVE

NEEDED MODIFICATIONS

Shift Register Input Buffer

The data shift register input circuit worked properly using the ideal power supply voltages. The control shift register was seen to give data patterns that suggested multiple unwanted clock pulses as well as random behavior. The control register path did work when the CMOS power supplies were first lowered from -2.2 to -4 to -2.8 to -3.7. This may have worked due to the fact that it gave the input inverter more headroom, while giving the final CMOS inverters a smaller threshold value. The smaller threshold value could have helped because the intermediate node tied to the resistor was not swinging high enough to change the output. Reasons for this are the value of the resistor could be significantly off, or that the drive strength of the second stage PFET connected before the bipolar diode drops was too small. The location of the control register input buffer circuit is also located near the high speed bipolar multiplexers as well as the current path of the output driver. This area has a much higher power dissipation than that of the data path buffer circuit and could therefore be at a much higher temperature. This temperature factor could also be causing variations in the threshold values of the transistors as well as the resistor value. During simulation and using worse case models it was seen that the intermediate node voltage swing occurred from -4 to -2.8 volts. In simulation this was still enough swing, but to give headroom for actual measurements the second stage PFET could have its width increased to 2u from 1u. In simulation this increase in width showed full power supply to power supply swings at that node using worst case simulation conditions.

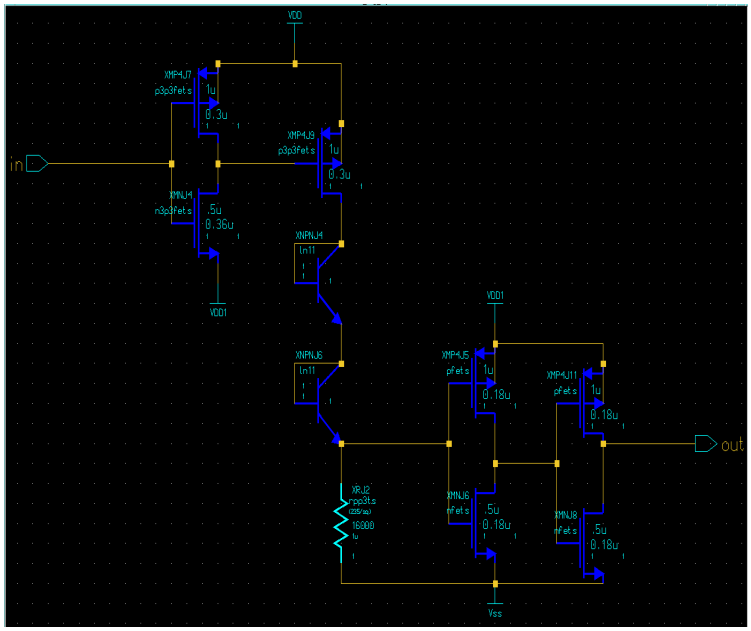

Fig. 29 Shift Register Input Buffer

Output Driver

The output driver's output swing characteristics are consistent with the simulation behavior at half of the bias current. Measurements of the prototype chip showed the output swing reaching a maximum value halfway to three quarters through the entire bit sequence rather than at the end of the bit sequence. The linearity problem is due to a temperature dependence in the CMOS reference current mirrors and 8-bit current DAC's; which at a higher temperature reach the maximum reference current level earlier in the bit sequence. The high level circuitry was simulated to be very sensitive to voltage drops in the -4V supply. Inconsistent measured and simulated voltage swing values are due to the source degenerated resistor pair in the bipolar current mirror stages. This resistor pair was constructed using two different material classes which had extremely different and opposite linear and quadratic temperature coefficients. Temperature and power supply

variation simulations for both the high level and output swing circuitry are recorded in Fig. 30-36.

Output Driver-High Level Simulations

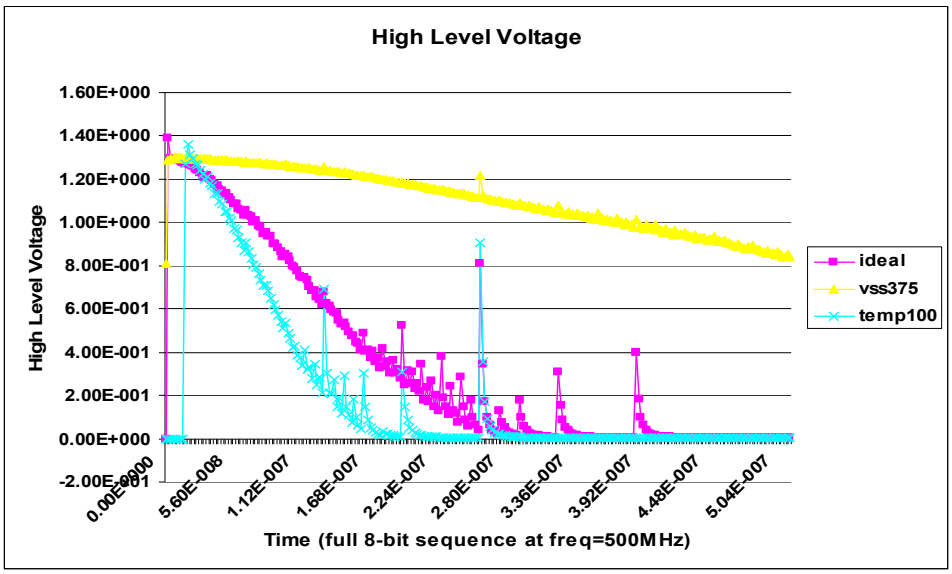

Fig. 30 Simulated High Level Voltage

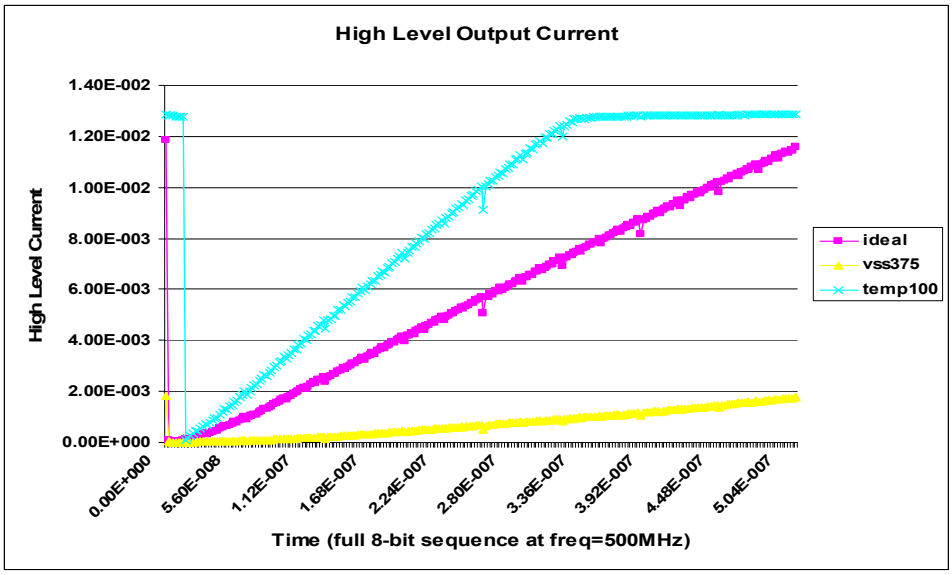

Fig. 31 Simulated High Level Output Current

The data contained in Fig. 28 and Fig. 29 was simulated from the circuit in Fig. 27. From this data it can be seen that a temperature rise caused the high level output

current to increase and for the high level voltage to be pulled down at an earlier time in the bit sequence. By raising the power supply from -4V to -3.75V the current mirror output dropped by approximately 83%.

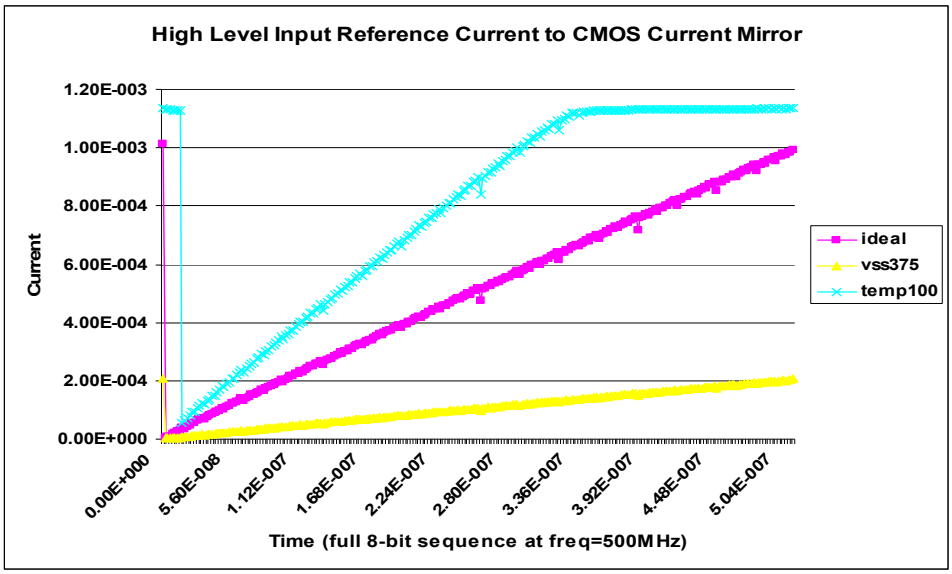

Fig. 32 Simulated High Level Input Reference Current to CMOS Current Mirror

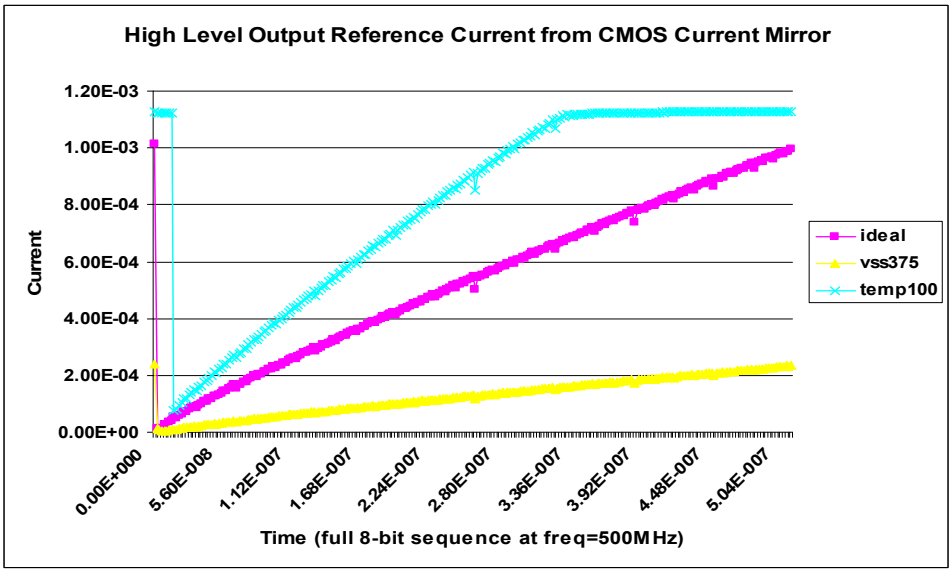

Fig. 33 Simulated High Level Output Reference Current from CMOS Current Mirror

The CMOS portion of the current mirror topology was also simulated to see the effects of power supply voltage drop and temperature. Again, a rise in temperature did

not affect the end value of the current levels, but increased the rate at which the current increased until its maximum value of approximately 1.0mA-1.2mA was met. A power supply voltage drop of 250mV caused a dramatic decrease in the reference current to the bipolar current mirror.

Output Driver-Output Swing Simulations

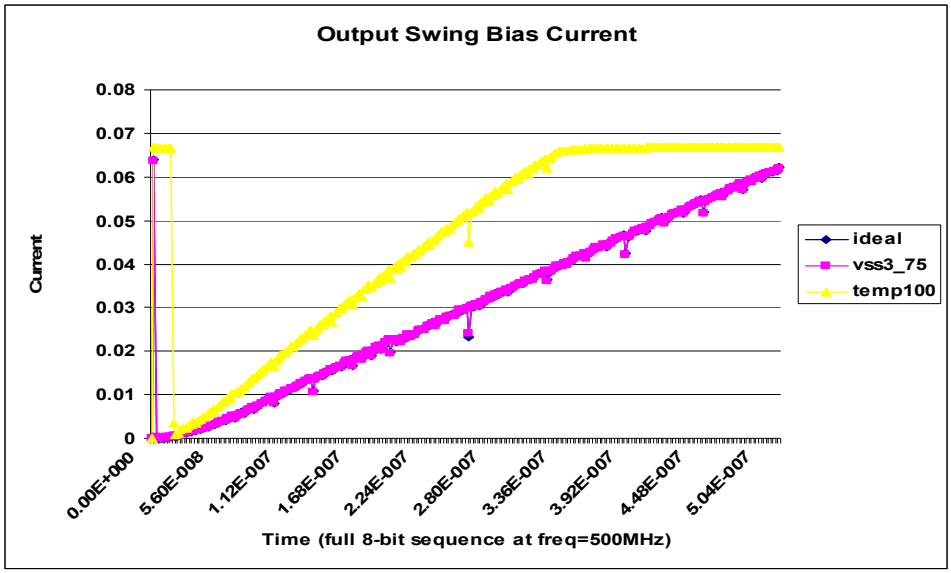

Fig. 34 Simulated Output Swing Bias Current

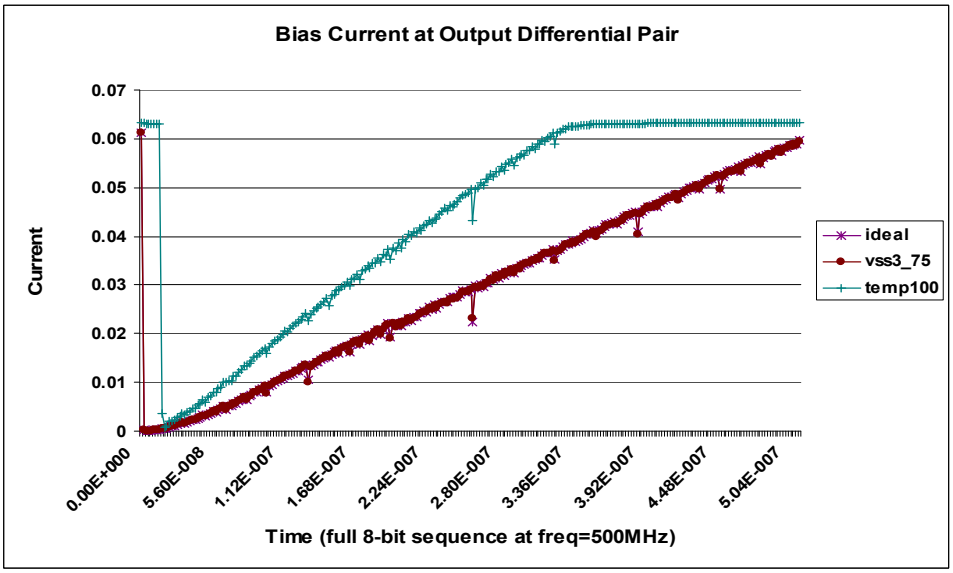

Fig. 35 Simulated Bias Current at Output Differential Pair

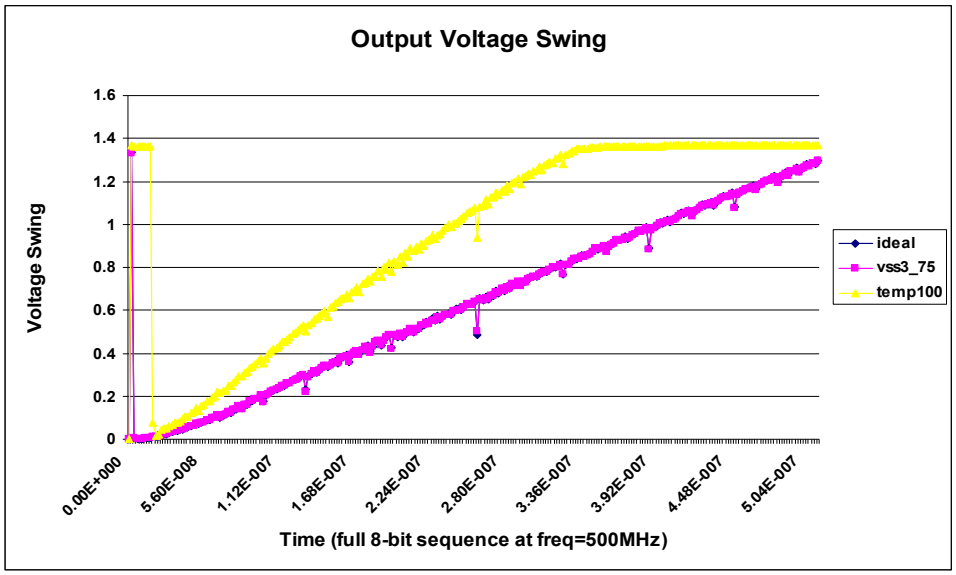

Fig. 36 Simulated Output Voltage Swing

Simulations completed on the output bias current which ideally should swing from 0-60mA into an equivalent 25 ohms when 50 ohm termination is used. From Fig. 32-34 the increase in temperature had the equivalent effect as was seen on the high level circuitry. Equivalent final current values were recorded at both 27 degrees C and 100 degrees C, although at high temperature, the maximum current value was reached much earlier in the bit sequence. Equivalent CMOS current mirrors were used in both the high level circuitry as well as the output swing circuitry, giving both circuits the same temperature dependence. However, the output swing circuitry was not found to be susceptible to a 250mV power supply drop.

Output Driver-Modifications

The current method of using equivalent 8-bit current DAC's and then using current mirrors composed of PFETs can be altered to use specific current level DAC's that do not require current mirrors. This should increase the linearity of the current

changes as well and remove the dependence upon the current mirror technology, which may be causing multiple linearity problems due a possible temperature dependence of the PFETs, parasitics, or even differences in the ideal and actual bias voltage value which feeds each of the binary weighted current sources. Power supply voltage drops due to either measurement line inductance or layout resistance can also be compensated for by widening power lines and either using lower inductance cable or raising power supply values to see any effects.

The high level circuitry needs to be simulated further to determine what is causing the sudden current drops in addition to further simulation of the output differential pair to look at possible current switching issues and breakdown voltage problems.

Resistor modifications need to be made so that the source degenerated resistor pairs in the bipolar current mirror circuitry are the same class of resistor. In addition, simulations were done using the worst case value of the resistors. Both the high level and output swing circuitry's response was greatly affected. More ideal current values form the current mirrors can be improved by using common centroid layout techniques to receive better resistor matching. This can help to improve the expected current multiplication.

Vernier Delay

Improvement of the vernier delay circuit need to be completed so that the overall behavior of the circuitry at least becomes monotonic. Other issues that need to be addressed are problems with clock drift and measurement accuracy.

Layout Issues

A clock routing error occurred during the layout that prevented the passage of the random data sequence through to the final output. Specifically, the clock signals that drive the first bipolar stage 4-1 Mux's which should be differential and the same voltage levels are non-differential and different voltage levels. This prevented measurement of data dependent output jitter and correct operation of the multiplexer. This error can be fixed with minor modifications. Below is a figure of the 4-1 mux along with a simulation showing the result of the routing error and the result after the error had been corrected.

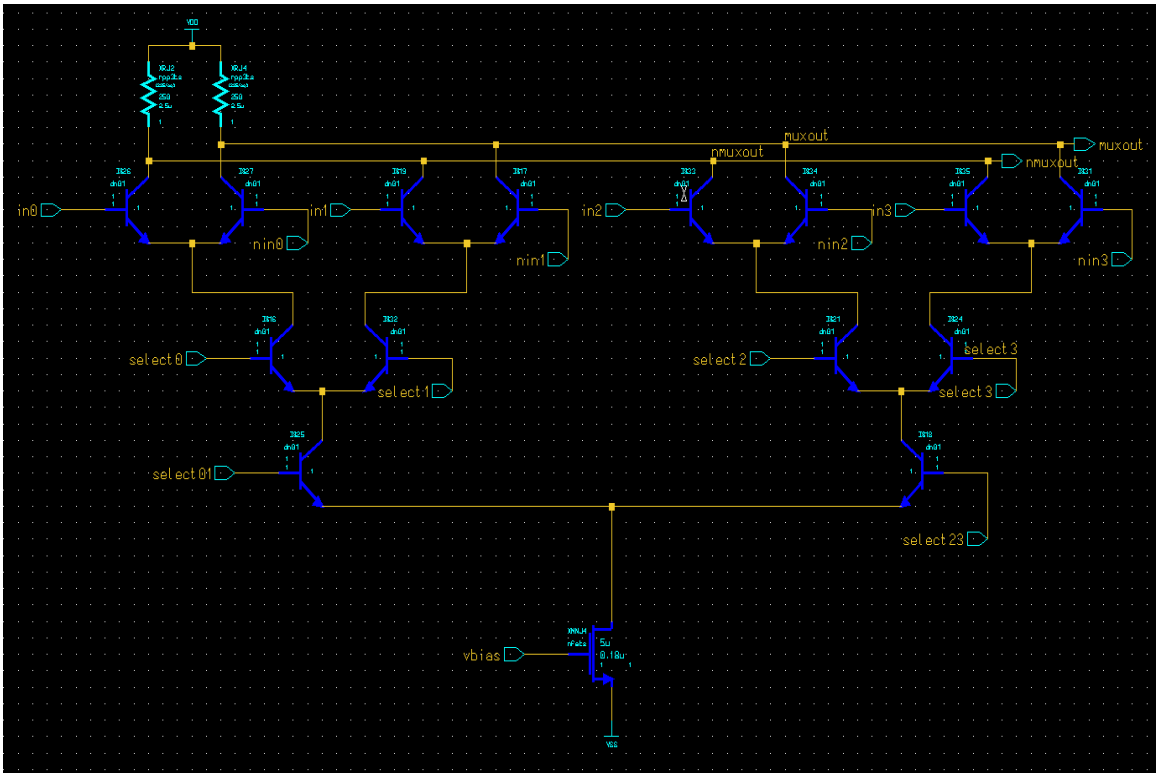

Fig. 37 Bipolar 4-1 Mux

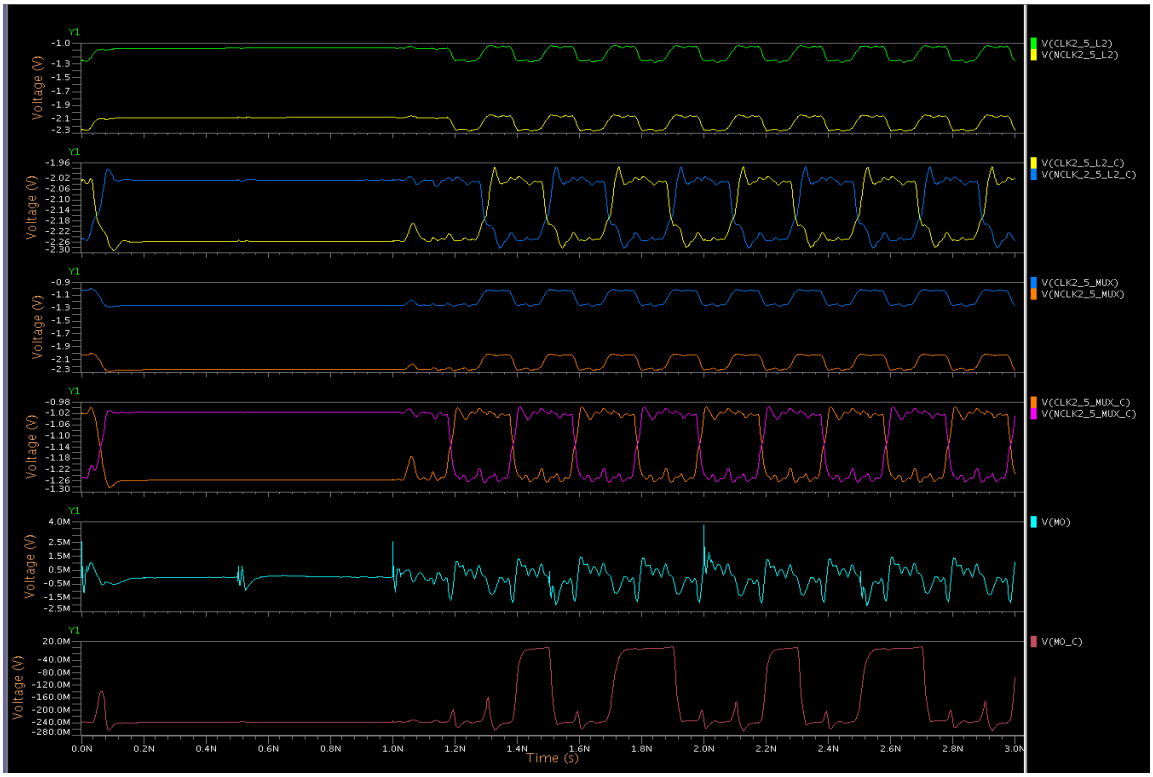

Fig. 38 4-1 Mux Error Simulation

From the above simulation the first and third sets of signals are what the error in layout is currently causing. By correcting the error, the second and fourth sets of signals are created. The final two signals are firstly, the output of the mux with the error, and the output of the mux after correction. This change again can be easily done by correcting the clock generator layout.

High Frequency Measurements

With the above mentioned corrections high frequency measurement will be able to be taken. This will allow for the further characterization of the vernier circuit, output driver, and jitter response of the overall pattern generator. Achieving measurements in the 20GHz range will require a better setup to reduce power supply lead inductance and better thermal capacity for temperature effects.

REFERENCES

[1] M. Siddiqui, et al., "InP and GaAs components for 40 Gbps applications," GaAs MANTECH Conference, 2002.

[2] C. Hwang, et al., "A High-Precision Time-To-Digital Converter Using A Two-Level Conversion Scheme," IEEE 2004 0-7803-8257-9/04:174-176.

[3] T. Xia, et al., "Self-Referred on-chip Jitter Measurement Circuit Using Vernier Oscillators," IEEE 2005 0-7695-2365-X/05.

[4] V. Ramakrishnan, et al., "A Wide-Range, High-Resolution, Compact, CMOS Time to Digital Converter," IEEE 2006 1063-9667/06.

[5] K. Cheng, et al., "Self-sampled vernier delay line for built-in clock jitter measurement," IEEE 2006 0-7803-9390-2/06:1591-1594.

[6] A. Chan, et al., "A SYNTHESIZABLE, FAST AND HIGH-RESOLUTION TIMING MEASUREMENT DEVICE USING A COMPONENT-INVARIANT VERNIER DELAY LINE," IEEE 2001 0-7803-7169-0/01:858-867.

[7] B. Nelson, et al., "ON-CHIP CALIBRATION TECHNIQUE FOR DELAY LINE BASED BIST JITTER MEASUREMENT," IEEE 2004 0-7803-8251-X/04:944-947.

[8] G.C. Moyer, et al., "The Delay Vernier Pattern Generation Technique," IEEE 1997 Apr, 32(4):551-562.

[9] H. Knapp, et al., "100-Gb/s 2^7-1 and 54-Gb/s 2^11-1 PRBS Generators in SiGe Bipolar Technology," IEEE 2005 Oct, 40(10):2118-2125.

[10] F. Schumann, et al., "Silicon bipolar IC for PRBS testing generates adjustable bit rates up to 25Gbit/s," ELECTRONIC LETTERS 1997 Nov, 33(24):2022-2023.

APPENDIX A

Detailed Operation

The initialization state checks for certain global situations to make sure that the correct section of code is working properly at all times. The initial purpose of the code is to either reset and initialize all register values and counters when an enable signal is low, and allow basic operation whenever the enable signal is high. When enable goes high the operation of the circuit starts in the first cycle. When the first cycle has completed, a variable c1 is set high. This tells the global code to switch to the second cycle code. As the second cycle finished, another variable c2 is set high and correspondingly c1 is set low. This behavior continues through each of the cycles until cycle four is finished. Additional jump variables are also set. Detection of a partial jump is easily done by checking bits 15 to 12 of the stop register value. If these four bits are not all logic 1, then a partial jump is occurring. During initialization variables j1, j2, j3, and j4 are set either high or low depending upon there being a partial jump detection or not. This will direct the initiation of the correct function block to commence.

After each cycle has ended the operation of the full chip should continue to step through the code until the next stop value is detected. This transition is over different RAM blocks and therefore there needs to be additional logic to make sure that when the final column in a RAM block has been loaded, the next column that will be read is the first column of the next corresponding block. This code simply checks for the location of the master signal and which RAM block counter it is currently clocking. It then checks for the situation where the final column in that block has been loaded and subsequently switches the master clock signal to the next RAM block.

A jump occurs when the stop value of one table entry has been reached and therefore the data multiplexers must switch back to the start value of that table entry or if the wanted number of cycles has been completed the circuit should operate normally until next table entry stop value. For this code a restriction is placed on a single table entry to have the start and stop register address be in the same block. The data through can only be multiplexed correctly if the data is ready at the outputs of the RAM. The simple way to do this is to have the control circuitry of the SRAM on the opposite clock phase as the actual multiplexers. During phase one of the clock signal the SRAM is allowed to have the correct data read and ready at its outputs. During phase 2 of the clock the data is sent through the multiplexer chain. Using basic flip flops will allow for the previous SRAM data to be held constant while the input values are changed to the next data set. This scheme works as long as the stop value of the table entry goes through the entire selected column. This allows for a set clock period to allow for the entire 128 bits in that column to be ready at the output and then multiplexed out while the entire next column value is being read. The code operation looks for the stop_reg[9:0]+1. This is the time at which the stop column for that table entry has had enough time to be read from the memory. At this time the counter for the specific block that the table entry is associated with is reset to the start value. In addition the cycle counter is clocked to show that cycle has been completed. This value will be continually checked to determine if the next table entry value should begin.

Operation with partial column stop values are a problem due to the fact that during phase 1 of the clock cycle the block counters will be reading the entire stop column. During phase 2 of the clock cycle the mux's will be counting through the stop

column partially, and then expect to go back to the start value of that table entry. This will be happening at the same time as the read of the start value and could cause data error. To combat this, an additional RAM block D is created to store the final stop register value if it is determined that it is a partial column value. Then during phase 1 of the clock cycle when the code has determined that the correct block counter has reached the end of stop reg [9:0]-1, or it has been read, the next values to be read are the start value of the table entry as well as the copied column of the stop value in RAM D. By doing this, both the stop value and start value will be available to be read during clock phase 2. Therefore, the mux detection will have to detect the final stop row and then switch to the start value. In addition, the multiplexers will be to deal with the partial stop column and the entire start column. This will require phase 2 of the clock to be longer by the length of the partial stop column value. Therefore a clock delay will also have to be inserted to ensure that the proper data values are multiplexed. This delay can be based on a lookup table and a high speed clock.

```
//*****************************************************************************
*******************************
//Matthew Zahller
//Washington State University
//40 Gbps SiGe Data Pattern Generator Code for Full Column Operation
//This code is a simplified version for
//functionality testing and code checks.

//Code for a single full column stop cycle.  This code is used for a simplified version for
//functionality testing and code checks.

//Important to operation that the counters controlled by the clock signals are only allowed
//to change values when a pos edge or neg edge is detected.  Even in the case of a set or
reset

//*****************************************************************************
************************
```

```
//Reference time is 1ns with a precision of 1ns
`timescale 1ns / 1ns

////////////////////////////////Input and Ouput Ports and Variables/////////////////////////////////////////

//Input and Ouput Ports
module cycle_full_column (
//Input Signals
   input enable,       //Circuit Enable
   input partial_enable, //if 1 partial code executed, if 0 full column code executed (j1-j4)
   input cycle_enable, //high to enable code only during one of the four cycles
   input clk_fast,     //High Speed Synchronization Clock
   input [9:0] counta, countb, countc,   //Outputs from the four block counters
   input [9:0] start,
   input [15:0] stop,  //Bits 15:12 tell stop row, 11:10 tell stop block, 9:0 tell stop column
   input [9:0] cycle,  //Cycle registers hold the number of iterations for each cycle

 //Output Signals
   output reg [9:0] reseta, resetb, resetc, seta, setb, setc, cycle_count
   );

////////////////////////////////////////////////////////////////////////////////////////////////////////////



always @(posedge clk_fast) //This section should be in state0
if(!enable)
cycle_count=10'b0000000000;


////////////////////////////////////////////////////////////////////////////////////////////////////////////

//*************************************************************************
************************
//Note:
//All of these code sections need an external clock signal that is the same frequency
//as the counta or countb or countc outputs to allow for proper operation this is actually
//already done by using a counter block because the block itself will not allow the counter
//output to change unless it detects a rising or falling edge of the control clock along
//with the set and reset values
//*************************************************************************
************************
```

```
 ////////////////////////////////////SRAM Full Column Stop
Code////////////////////////////////////////////////
always @(counta or countb or countc)

if(!partial_enable & cycle_enable & !(&(cycle_count ~^ (cycle+1))) & enable)
begin

case({stop[11], stop[10]})

2'b00 :
begin
if(&(counta ~^ stop[9:0]+1))
begin
seta=start;
reseta=~start;
cycle_count=cycle_count+1;
end
end

2'b01 :
begin
if(&(countb ~^ stop[9:0]+1))
begin
setb=start;
resetb=~start;
cycle_count=cycle_count+1;
end
end

2'b10 :
begin
if(&(countc ~^ stop[9:0]+1))
begin
setc=start;
resetc=~start;
cycle_count=cycle_count+1;
end
end

default :
begin
seta=10'b0000000000;
reseta=10'b0000000000;
setb=10'b0000000000;
resetb=10'b0000000000;
setc=10'b0000000000;
```

```
resetc=10'b0000000000;
end
endcase
end

endmodule
////////////////////////////////////////////////////////////////////////////////
```

```
//*****************************************************************************
*******************************
//Matthew Zahller
//Washington State University
//40 Gbps SiGe Data Pattern Generator Code for Partial Column Stop Values
//This code is a simplified version for
//functionality testing and code checks.

//Notes:
//Important to operation that the counters controlled by the clock signals are only allowed
//to change values when a pos edge or neg edge is detected.  Even in the case of a set or
reset

//*****************************************************************************
*******************************
```

```
//Reference time is 1ns with a precision of 1ns
`timescale 1ns / 1ns

//////////////////////////////Input and Ouput Ports and Variables/////////////////////////////////////////

module cycle_partial_column_single (
//Input Signals
   input enable,       //Circuit Enable, when code is consolidated anything that depends on
enable=0
                  //will be transferred to state0 code
   input partial_enable, //if 1 partial code executed, if 0 full column code executed (j1-j4)
   input cycle_enable,//high to enable code only during one of the four cycles
   input clk_fast,     //High Speed Synchronization Clock
   input [9:0] counta, countb, countc,  //Outputs from the three block counters
   input [1:0] block_count, //Logic to set which SRAM block code to use
   input [9:0] start,  //10 bits which hold the starting column value
   input [15:0] stop,  //Bits 15:12 tell stop row, 11:10 tell stop block, 9:0 tell stop column
   input [9:0] cycle,  //Cycle registers hold the number of iterations for each cycle
   input [3:0] mux1,   //128:16 Mux output
   input [9:0] Block_D_Storage, //User defined storage location for copied partial column

 //Output Signals
   output reg [9:0] reseta, resetb, resetc, seta, setb, setc, cycle_count,
   output reg [9:0] resetd, setd,
   output reg end1, end2, end3, delay1_dec,
   output reg [3:0] reset_mux, set_mux,
   output reg [1:0] reset_fmux, set_fmux, reset_count, set_count,
   output reg [3:0] delay_a, delay_b, delay_c,
   output reg delay_a_enable, delay_b_enable, delay_c_enable
   );

//Variables
   reg x1, x2, x3;


////////////////////////////////////////////////////////////////////////////////////////////////////////

always @(posedge clk_fast) //This section should be in state0 when consolidated
if(!enable)
cycle_count=10'b0000000000;


//////////////Code for Detection of the Completion of Read Time for Final Full Column
Value////////////////

always @(counta or countb or countc)
```

```
case({stop[11], stop[10]})

2'b00 :
begin
end1=&(counta ~^stop[9:0]);
end2=1'b0;
end3=1'b0;
end
2'b01 :
begin
end1=1'b0;
end2=&(countb ~^ stop[9:0]);
end3=1'b0;
end
2'b10 :
begin
end1=1'b0;
end2=1'b0;
end3=&(countc ~^ stop[9:0]);
end

default :
begin
end1=0;
end2=0;
end3=0;
end
endcase

////////////////////////////////////////////////////////////////////////////////////////////////

//*****************************************************************************
************************
//Note:
//All of these code sections need an external clock signal that is the same frequency
//as the counta or countb or countc outputs to allow for proper operation this is actually
//already done by using a counter block because the block itself will not allow the counter
//output to change unless it detects a rising or falling edge of the control clock along
//with the set and reset values
//*****************************************************************************
************************

 ////////////////////////////////////SRAM Block A Code////////////////////////////////////////////////

always @(posedge counta[0] or negedge counta[0] or posedge mux1[0] or negedge
mux1[0])
```

```
   if(!block_count[1] & !block_count[0] & partial_enable & cycle_enable &
!(&(cycle_count ~^ (cycle+1))) & enable)
      begin


   if(end1)
   begin
      x1=1'b1;
      seta=start;
      reseta=~start;
      set_fmux=2'b11;//Sets the final mux to block D
      reset_mux=4'b1111;
      setd=Block_D_Storage; //Sets column counter in block D to designated storage area
      resetd=~Block_D_Storage;
   end
   else if(!end1 & x1)
      begin
      //Allows normal operation until end1 is activated
      seta=10'b0000000000;
      reseta=10'b0000000000;
      set_fmux=2'b00;
      reset_fmux=2'b11; //Corresponds to block A
      reset_mux=4'b0000;
      set_mux=4'b0000;
      setd=10'b0000000000;
      resetd=10'b0000000000;


      //This code detects the completion of the partial column data and initiates the proper
      //resets to continue with normal operation
      if(&(stop[15:12]+1 ~^ mux1))
      begin
      reset_mux=4'b1111; //128 to 16 Block Mux Reset
      reset_fmux=2'b11; //end1 corresponds to block A
      cycle_count=cycle_count+1;//Corresponds to first cycle
      reset_count=2'b11; //Reset block counter to block A
      delay_a_enable=1'b1;//Signal finish of block D values
      end

      //Clock Delay Code which only initiates after block D values have been used
      delay_a=4'b1111-stop[15:12]; //Leftover delay for counter to stay in the current
column
      if(delay_a_enable & &(mux1 ~^ delay_a))
      begin
```

```
        seta=counta; //If fed to a counter it should retain its current value
        reseta=~counta;
        end

        else if(&(mux1))
        begin
        seta=10'b0000000000;
        reseta=10'b0000000000;
        delay_a_enable=1'b0;
        x1=1'b0;
        end
    end
    end

 ////////////////////////////////////SRAM Block B Code////////////////////////////////////////////////

always @(posedge countb[0] or negedge countb[0] or posedge mux1[0] or negedge
mux1[0])


    if(!block_count[1] & block_count[0] & partial_enable & cycle_enable &
!(&(cycle_count ~^ (cycle+1))) & enable)
        begin


    if(end2)
    begin
        x2=1'b1;
        setb=start;
        resetb=~start;
        set_fmux=2'b11;//Sets the final mux to block D
        reset_mux=4'b1111;
        setd=Block_D_Storage; //Sets column counter in block D to designated storage area
        resetd=~Block_D_Storage;
    end
    else if(!end2 & x2)
        begin
        //Allows normal operation until end2 is activated
        setb=10'b0000000000;
        resetb=10'b0000000000;
        reset_fmux=2'b10; //Corresponds to block B
        set_fmux=2'b01;
        set_mux=4'b0000;
        reset_mux=4'b0000;
        setd=10'b0000000000;
        resetd=10'b0000000000;
```

```
        delay_b_enable=1'b1;


        //This code detects the completion of the partial column data and initiates the proper
        //resets to continue with normal operation
        if(&(stop[15:12]+1 ~^ mux1))
        begin
        reset_mux=4'b1111; //128 to 16 Block Mux Reset
        reset_fmux=2'b10; //end2 corresponds to block B
        set_fmux=2'b01;
        cycle_count=cycle_count+1;//Corresponds to first cycle
        reset_count=2'b10; //Reset block counter to block B
        set_count=2'b01;
        delay_b_enable=1'b1;//Signal finish of block D values
        end

        //Clock Delay Code which only initiates after block D values have been used
        delay_b=4'b1111-stop[15:12]; //Leftover delay for counter to stay in the current
column
        if(delay_b_enable & &(mux1 ~^ delay_b))
        begin
        setb=countb; //If fed to a counter it should retain its current value
        resetb=~countb;
        end

        else if(&(mux1))
        begin
        setb=10'b0000000000;
        resetb=10'b0000000000;
        delay_b_enable=1'b0;
        x2=1'b0;
        end
    end
    end

 ////////////////////////////////////SRAM Block C Code////////////////////////////////////////////////

always @(posedge countc[0] or negedge countc[0] or posedge mux1[0] or negedge
mux1[0])


    if(block_count[1] & !block_count[0] & partial_enable & cycle_enable &
!(&(cycle_count ~^ (cycle+1))) & enable)
        begin
```

```
    if(end3)
    begin
       x3=1'b1;
       setc=start;
       resetc=~start;
       set_fmux=2'b11;//Sets the final mux to block D
       reset_mux=4'b1111;
       setd=Block_D_Storage; //Sets column counter in block D to designated storage area
       resetd=~Block_D_Storage;
    end
    else if(!end3 & x3)
       begin
       //Allows normal operation until end3 is activated
       setc=10'b0000000000;
       resetc=10'b0000000000;
       reset_fmux=2'b01; //Corresponds to block C
       set_fmux=2'b10;
       set_mux=4'b0000;
       reset_mux=4'b0000;
       setd=10'b0000000000;
       resetd=10'b0000000000;
       delay_c_enable=1'b1;


       //This code detects the completion of the partial column data and initiates the proper
       //resets to continue with normal operation
       if(&(stop[15:12]+1 ~^ mux1))
       begin
       reset_mux=4'b1111; //128 to 16 Block Mux Reset
       reset_fmux=2'b01; //end2 corresponds to block C
       set_fmux=2'b10;
       cycle_count=cycle_count+1;//Corresponds to first cycle
       reset_count=2'b01; //Reset block counter to block C
       set_count=2'b10;
       delay_c_enable=1'b1;//Signal finish of block D values
       end

       //Clock Delay Code which only initiates after block D values have been used
       delay_c=4'b1111-stop[15:12]; //Leftover delay for counter to stay in the current
column
       if(delay_c_enable & &(mux1 ~^ delay_c))
       begin
       setc=countc; //If fed to a counter it should retain its current value
       resetc=~countc;
       end
```

```
        else if(&(mux1))
        begin
        setc=10'b0000000000;
        resetc=10'b0000000000;
        delay_c_enable=1'b0;
        x3=1'b0;
        end
    end
    end


endmodule
//////////////////////////////////////////////////////////////////////////////////
```

APPENDIX B

```
//This program steps through the feedback current source and
//measures its effects


#include <ni488.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "Pattern_Gen.h"
void wrt(int ud, char *x)
{ ibwrt(ud,x,strlen(x)); }

int main()
{
FILE *fptr;
fptr=fopen("Current Feedback.txt", "w");
char temp[80];
char temp2[80];
        int tds8000;


   // Initialize a GPIB device (voltage source)
   tds8000 = ibdev(0, // Board ID
            1,   // Primary ID
            0,   // Secondary ID
            12,  // Timeout (1 second)
            1,   // 'eoi on last byte' flag
            0);  // end-of-string mode

        int hp6629;


   // Initialize a GPIB device (voltage source)
   hp6629 = ibdev(0, // Board ID
            5,   // Primary ID
            0,   // Secondary ID
            12,  // Timeout (1 second)
            1,   // 'eoi on last byte' flag
            0);  // end-of-string mode


       unsigned char delay={0x00};
       unsigned char vern1={0x00};
       unsigned char vern2={0x00};
       unsigned char breakdown={0xff};
       unsigned char cfs;
```

```
unsigned char hls={0x00};
unsigned char os={0xff};
unsigned char ls={0xaa};


OpenDevice(0);
c
int i;

fprintf(fptr, "Current Feedback\n");
//Vern1 Count
for(i=0; i<256; i=i+8)
{
cfs=i;

SetDigitalChannel(VDD);
ClearDigitalChannel(SR_RESET);

//Set Power Supplies to Needed Threshold Values
wrt(hp6629, "VSET 3, 3.7");
usleep( 400000); // sleep for .4 second*/
wrt(hp6629, "VSET 2, .9");
usleep( 400000); // sleep for .4 second*/

control_serial(delay, vern1, vern2, breakdown, cfs, hls, os, ls);

//Reset Power Supply to Full Swing and wait for settling
wrt(hp6629, "VSET 3, 4");
usleep( 1000000); // sleep for 1 second
wrt(hp6629, "VSET 2, 1.8");
usleep( 1000000); // sleep for 1 second
usleep(10000000);
wrt(tds8000, "Measu:Meas5:Val?");
ibrd(tds8000, temp, 80);
wrt(tds8000, "Measu:Meas6:Val?");
ibrd(tds8000, temp2, 80);
fprintf(fptr, "%s, %s\n ", temp, temp2);
fflush(fptr);

}


}
```

```
//This file includes all control functions
//for the Velleman board interface


#include <k8055.h>

#define SR_CC                   1
#define SR_RESET    2
#define VDD             3
#define SR_DATA                 4
#define SR_DC                   5
#define CHIP_CLK_RESET  6
#define SR_OUT                  1

void control_reset()
{
        ClearDigitalChannel(SR_CC);
        SetDigitalChannel(SR_RESET);
        SetDigitalChannel(SR_CC);
        ClearDigitalChannel(SR_CC);
        ClearDigitalChannel(SR_RESET);
}

void data_reset()
{
        ClearDigitalChannel(SR_DC);
        SetDigitalChannel(SR_RESET);
        SetDigitalChannel(SR_DC);
        ClearDigitalChannel(SR_DC);
        ClearDigitalChannel(SR_RESET);
}

void gen_sendbit(int b, int clk)
{
        if(b)   {
                SetDigitalChannel(SR_DATA);
                printf("1");
        }else   {
                ClearDigitalChannel(SR_DATA);
                printf("0");
        }

                SetDigitalChannel(clk);
                ClearDigitalChannel(clk);

}
```

```
void gen_sendbyte(signed char b, int clk)
{
	gen_sendbit(b&0x80,clk);
	gen_sendbit(b&0x40,clk);
	gen_sendbit(b&0x20,clk);
	gen_sendbit(b&0x10,clk);
	gen_sendbit(b&0x08,clk);
	gen_sendbit(b&0x04,clk);
	gen_sendbit(b&0x02,clk);
	gen_sendbit(b&0x01,clk);
}

void gen_sendbit2(signed char b, int clk)
{
	gen_sendbit(b&0x02,clk);
	gen_sendbit(b&0x01,clk);

}

void data_serial(signed char data[16])
{

	int i;

	for(i=0; i<16; i++)
	{
		gen_sendbyte(data[i], SR_DC);
		printf("\n");
	}

}

unsigned char send_read_byte(unsigned char b, int clk, int bits)
{


	int i, mask;
	unsigned char x=0;
	for(i=bits-1; i>=0; i--)
	{
		mask=1<<i;
		ReadDigitalChannel(SR_OUT);
		if(!ReadDigitalChannel(SR_OUT))
```

```
                {
                        x |=mask;
                }
                if(b & mask)
                {
                        SetDigitalChannel(SR_DATA);
                }
                else
                {
                        ClearDigitalChannel(SR_DATA);
                }

                SetDigitalChannel(clk);
                ClearDigitalChannel(clk);
        }
        printf("Wrote:%02x  Read:%02x\n",b,x);
        return x;

}

void control_serial(signed char delay, signed char vern1, signed char vern2,
signed char breakdown, signed char cfs, signed char hls, signed char os, signed char ls)
{

        send_read_byte(ls, SR_CC, 8);
        send_read_byte(os, SR_CC, 8);
        send_read_byte(hls, SR_CC, 8);
        send_read_byte(cfs, SR_CC, 8);
        send_read_byte(breakdown, SR_CC, 2);
        send_read_byte(vern2, SR_CC, 8);
        send_read_byte(vern1, SR_CC, 8);
        send_read_byte(delay, SR_CC, 2);
//      gen_sendbyte(ls, SR_CC);
//      printf("\n");
//      gen_sendbyte(os, SR_CC);
//      printf("\n");
//      gen_sendbyte(hls, SR_CC);
//      printf("\n");
//      gen_sendbyte(cfs, SR_CC);
//      printf("\n");
//      gen_sendbit2(breakdown, SR_CC);
//      printf("\n");
//      gen_sendbyte(vern2, SR_CC);
//      printf("\n");
//      gen_sendbyte(vern1, SR_CC);
//      printf("\n");
```

```
//        gen_sendbit2(delay, SR_CC);
//        printf("\n");

}


unsigned char readbyte(int clk)
{

        int i;
        unsigned char x=0;
        for(i=0; i<8; i++)
        {
                ReadDigitalChannel(SR_OUT);
                x=ReadDigitalChannel(SR_OUT) | (x<<1);
                SetDigitalChannel(clk);
                ClearDigitalChannel(clk);
        }

        return x;

}


unsigned char readbit2(int clk)
{

        int i;
        unsigned char x=0;
        for(i=0; i<2; i++)
        {
                ReadDigitalChannel(SR_OUT);
                x=ReadDigitalChannel(SR_OUT) | (x<<1);
                SetDigitalChannel(clk);
                ClearDigitalChannel(clk);
        }

        return x;

}
```