NTNU
Kunnskap for en bedre verden

Department of Electronic Systems

TFE4152 - Design of Integrated Circuits

Term Project

Authors:
Andrew Glover Martey

Henrik Nøsen Dahle

November, 2021

Table of Contents

List of Figures

1 Introduction

Digital cameras have become a major part of a modern life where almost everyone carries a camera in their pockets in form of a mobile phone. To achieve high quality photos in a small camera technologies such as CMOS sensors can be used. A modern camera consist of a analog circuit and a digital one, which makes it a mixed circuit design. The analog circuit is used to capture an analog value based on the light intensity hitting the sensor and convert it into a digital signal. The conversion of the analog signal into a digital is done by an Analog-to-Digital Converter (ADC). The digital circuit is responsible for managing the analog circuit and treating the digital signal.

This project will omit certain design elements due to its scale. In the analog design noise and variance will be ignored during the design, while in the digital circuit testability is drastically reduced. Since the tools given for the project does not support mixed-signal design, where the analog and digital circuit is simulated together, certain shortcuts will be taken in the design to ensure that the modules work as expected. These design elements will be discussed later in the paper.

This project report is five (5) chapters or sections. Chapter one (1) is the introduction to the project. It entails a brief background of this project, problem statement, outline of the project report, scope, and limitations. Chapter (2) reviews literature from the project paper and similar papers relevant and related to this project. Chapter (3) is the implementation of the project. Here the methodology is discussed. Chapter (4) presents the results obtained; chapter (5) discusses the results obtained. Chapter (6) is the conclusion(s) made based on results and discussions made from the previous chapters. Recommendations are also given in this chapter.

2 Theory

2.1 Analog circuit design

Analog circuits can be designed with the help of spice code. Spice allows the user to describe how the circuit will be connected and how the transistors sized. Analog design are usually verified with transient simulation, this will simulate each time step to find how the voltages and currents changes. Analog design requires careful sizing of the transistors since it operates in a continues voltage space, where noise can drastically affect the circuit behavior.

2.2 Digital circuit design

Digital circuits are also designed with a type of code, which is called hardware description language (HDL). This makes the process of creating large and complex circuits faster to write and easier to understand by the user. In this project the HDL used is system verilog. Digital circuits are usually designed with minimal transistor size since it only operates in two voltage modes, high or low. This makes the circuit more resistant towards noise and other variances.

2.3 Counters

Counters are a useful tool to count up or down towards a goal. The simplest way to implement a counter is with a flip flops that changes state whenever the clock signal is triggered. The output of this flip flop then drives the next flip flop and so on. With this configuration it is possible to read the output of the flip flops to get the current amount it has counted.

Another way to count is with instead a gray encoded counter. Gray encoding is created with the principle that only one bit flips on each cycle, while a binary encoding may flip all the bits at once. This allows for a more stable analog circuit since the noise generated from the counter is reduced.

3 Implementation

This section will explain the implementation of the digital and analog circuit. how these circuits work are also explained along with the basic testbenches. The analog circuit is implemented with spice and can be found in appendix A. The digital circuit is implemented with system verilog and can be located in appendix B and C.

3.1 Analog Circuit

The pixel sensor, as seen in figure 1, consist of 3 discrete components, a light sensor, comparator and memory latch [1]. The light sensor is responsible for converting the light into a analog signal which drives the comparator. The analog signal is then compared against a ramp signal. While the ramp signal is lower than the analog signal the comparator will drive a logical 1, but otherwise it will be a logical 0. The digital signal will then drive the memory latch. The latch is used to record a digital version of the ramp signal. While the latch is driven by a logical 1 the latch will track the digital ramp and when it switches to 0 the latch will stop tracking and store the last value. This value can be used to decide the light intensity of the pixel sensor.

Since each pixel contains all of the required analog logic required to convert the data from analog to digital the architecture is highly scalable. The pixels are connected in a matrix configuration. Since all of the pixels can convert the pixel data independent of each other all of the pixels can work in parallel.

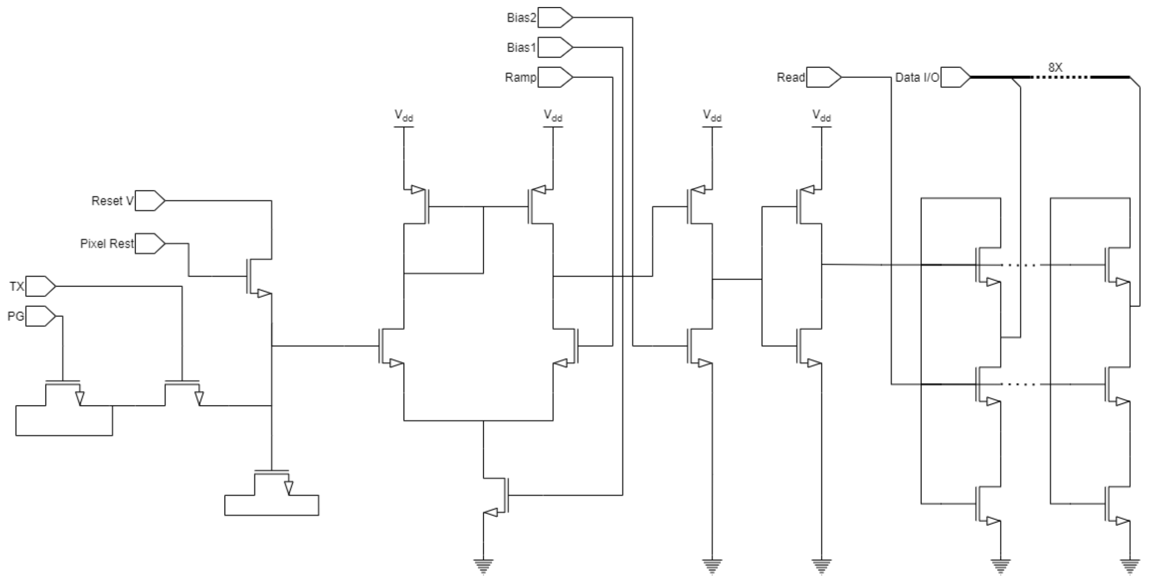


Figure 1: Pixel Schematic [1]

3.2 Photo Diode

The photo diode schematic can be found in figure 2. The photo diode circuit can be operated in three different modes, transfer, reset and sampling. The reset mode is used to remove any charge that is left over in the circuit so the next sampling becomes more accurate. A sampling signal is used to samples the current light intensity that is hitting the mosfet. Finally the transfer signal is used to the data over to the store capacitor to drive the comparator. The capacitor is implemented with a mosfet where the source and drain ports are connected together. This configuration can be called a moscap.

Since this circuit is not able to simulated, due to the lack of a physical product, a virtual sensor based on voltage sources and resistors was given at the beginning of the project. This allows for the verification of the rest of the analog circuit.

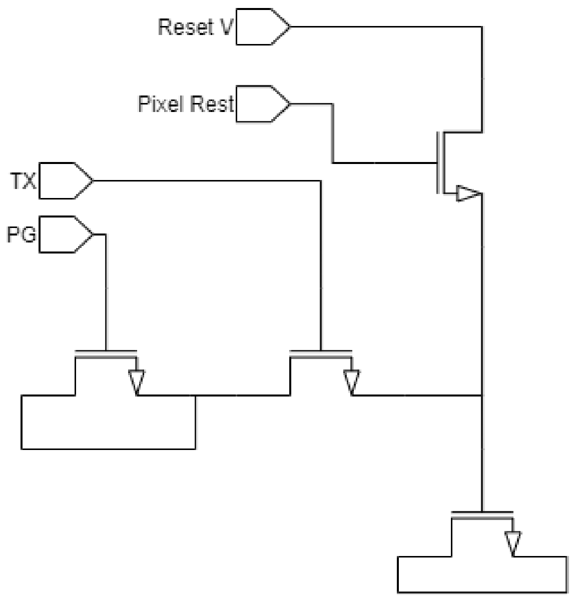


Figure 2: Schematic of the photo diode

3.3 Comparator

The comparator circuit, which can be seen in figure 6, is used as an analog to digital converter (ADC) to find the light intensity which hit the photo diode [1]. To be able to convert the analog signal into a digital one a analog ramp is utilized. The analog ramp is converted from a digital signal which is also ramping. The comparator is connected such that the input from the photo diode is connected to the positive input while the ramp is connected to the negative. As long as the ramp is lower than the input the output of the comparator will be high, but when the ramp passes the input the output will be low.

The comparator is created with a single-ended differential gain stage that will give the difference between the inputs. This can be described as $V_{out} = A * (V_{in} - V_{ramp})$ where A is the gain factor of the stage. Two buffers are placed at the output of the comparator to help drive the memory latch. The first buffer such that it amplifies the output of the comparator for a better output, but this inverts the signal. The gain stage can be described as $V_{out} = -A * V_{in}$. To solve the inversion a final buffer is used that is connected as an inverter. In this project the *bias1* and *bias2* voltages are equal.

3.4 Memory Latch

To store the digital result of the ADC conversion a simple memory latch was used [1], as seen in figure 4. The memory latch is created so when the output of the comparator is high the latch is tracking the digital version of the ramp signal. When the comparator switches from high to low the latch will stop tracking and save the signal. The memory used in this project consist of 8 bits giving the resolution of 255. Since the pixel cell is designed for high speed operation the memory latch is only designed for holding the value for a short time.

3.5 Control Signals

Figure 5 shows when the control signals *convert*, *erase*, *expose* and *read* are applied. *Erase* is at a high state when the program starts and after it is done reading, otherwise it is low. After the pixel

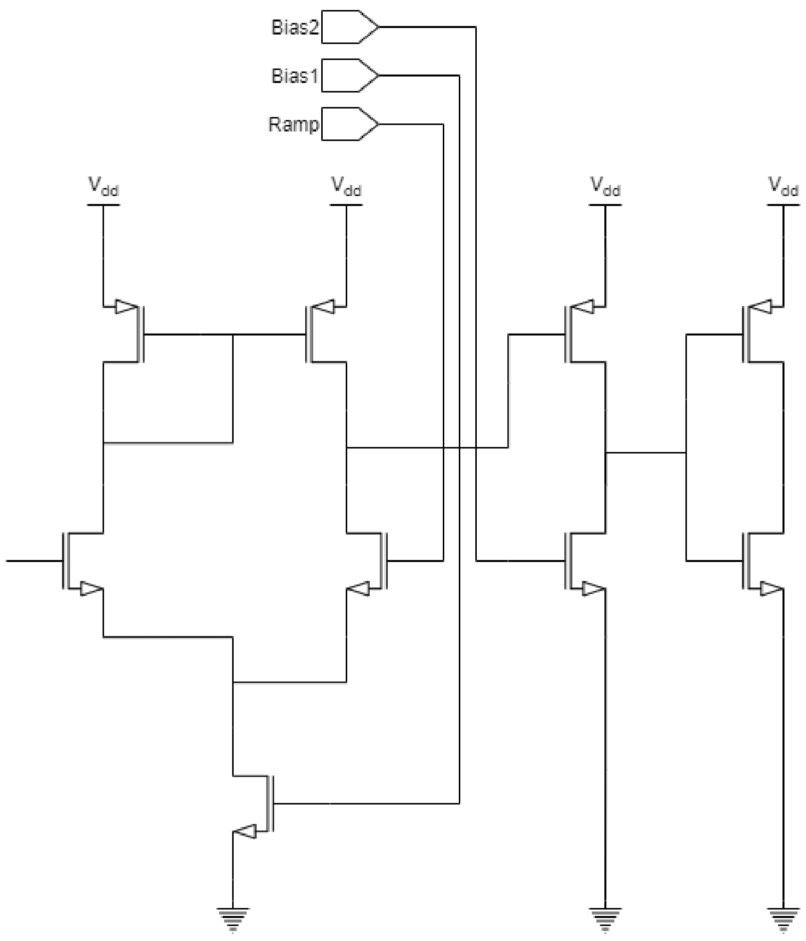


Figure 3: Schematic of the comparator

is reset the *expose* signal goes high so the pixel can measure the light intensity, and after that the signal is converted to a digital signal.

The analog to digital conversion is done with a comparator and a voltage ramp. The voltage ramp is driven by a digital counter which will count from 0 to 255. The comparator configuration is shown in figure 6. Figure 7 shows how the voltage ramp is applied to comparator.

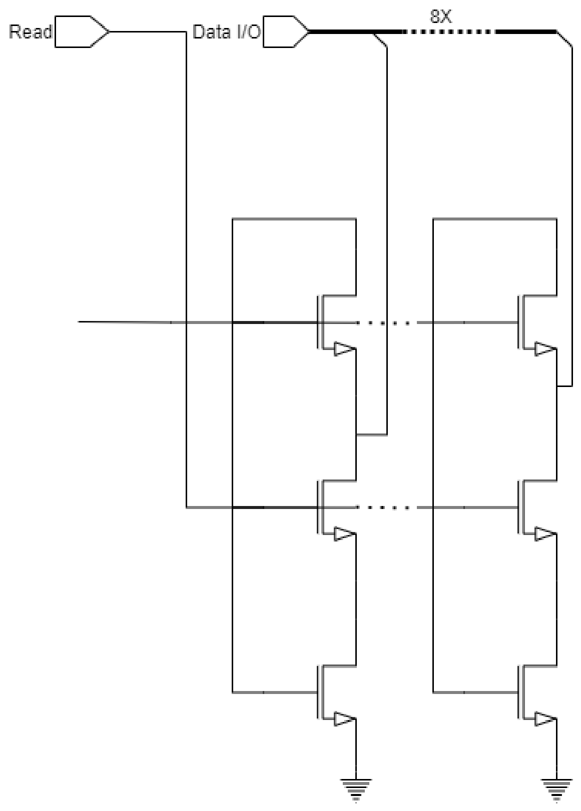


Figure 4: Schematic of the memory latch

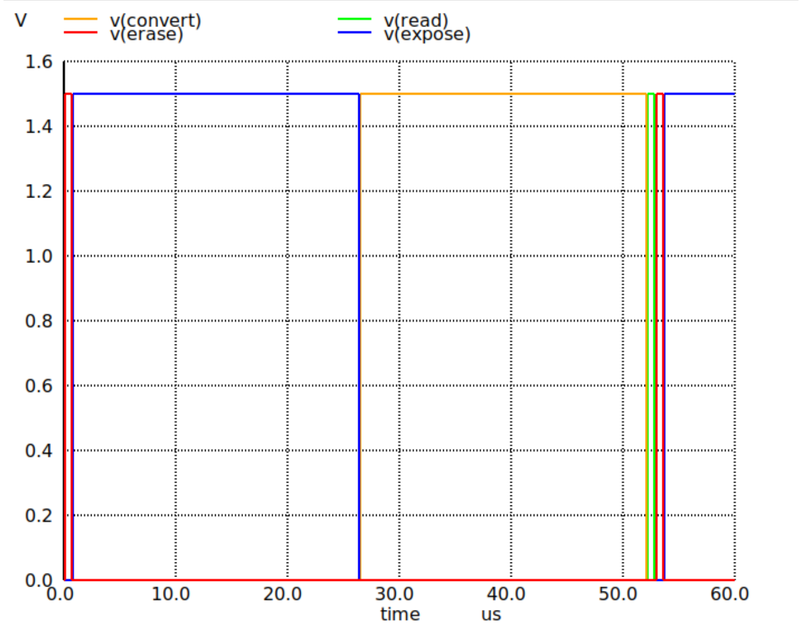


Figure 5: Control signals of the analog testbench

3.6 Digital circuit

A single finite state machine was implemented which controlled all four pixel sensors. The system returns to the idle state temporarily to reset the counter.

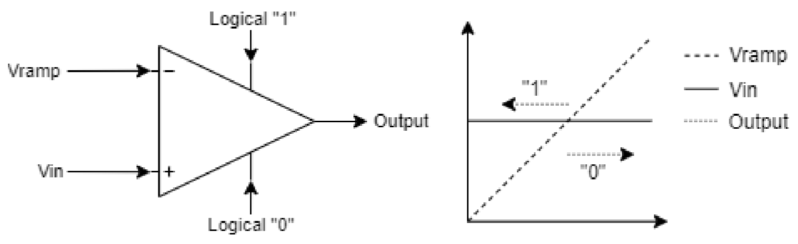


Figure 6: Analog to digital converter based on a single comparator

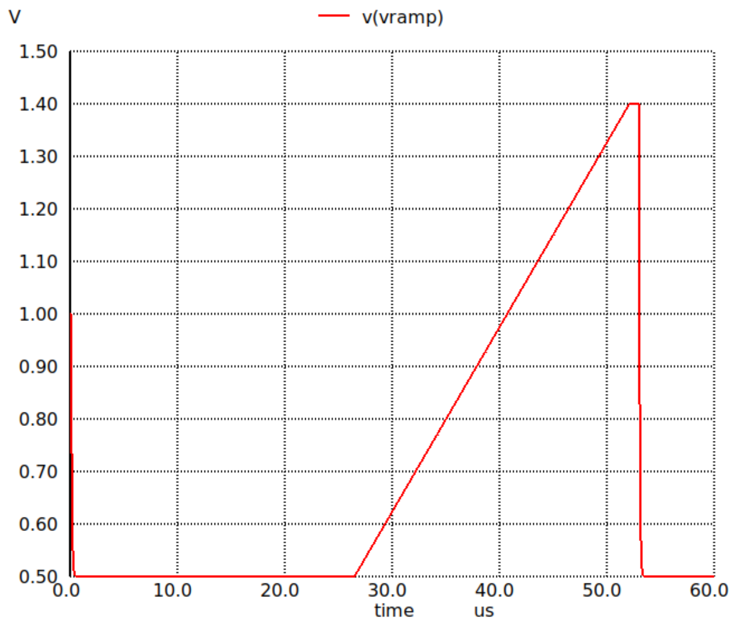


Figure 7: the voltage ramp for the comparator

4 Result

4.1 Analog circuit

Figure 10 shows how the date of the a single pixel behaves. The start of the signal is stable at around 45 where the system is in a steady state. Then the ramp signal can be seen, this is the signal used to calculate the digital value of the analog signal. When the conversion is complete the reading of the data begins. This stage is very short and occurs around 52 μS. In this test the value read out was approximate 125.

Figure 11 and 12 shows an example of result that was low and high, respectfully. The red line shows the digital signal that is applied to the memory cell. When the comparator changes state and locks the latch is the memory the digital signal of that time is latched and saved for the read state. The blue line is the what the memory cell reads to the digital circuit.

Figure 11 shows the fifth bit position for the memory cell, while Figure 12 shows the sixth bit position.

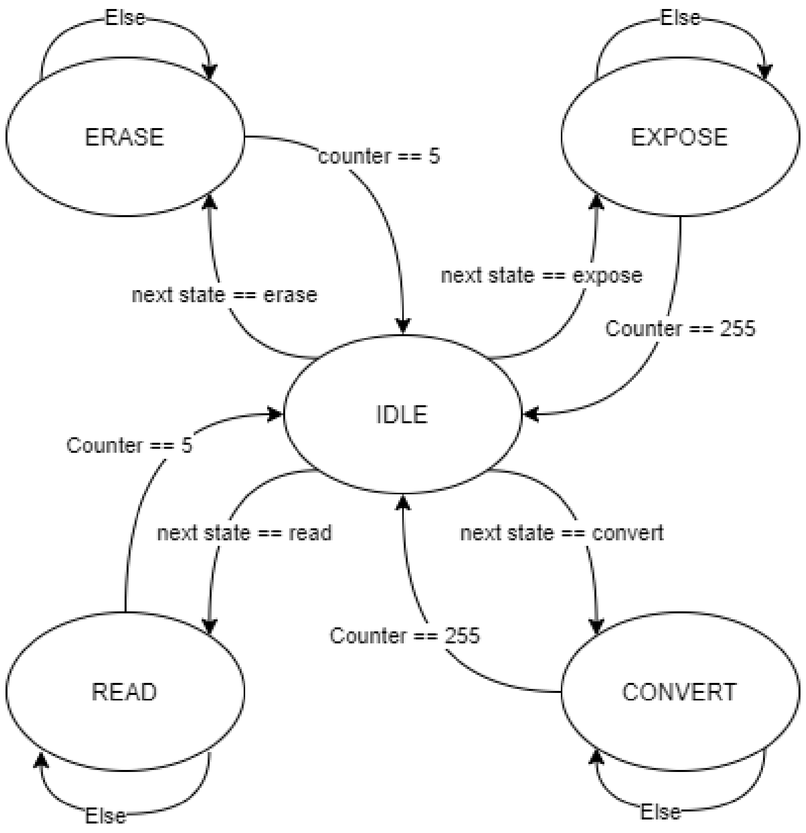


Figure 8: Finite state machine of the digital control circuit

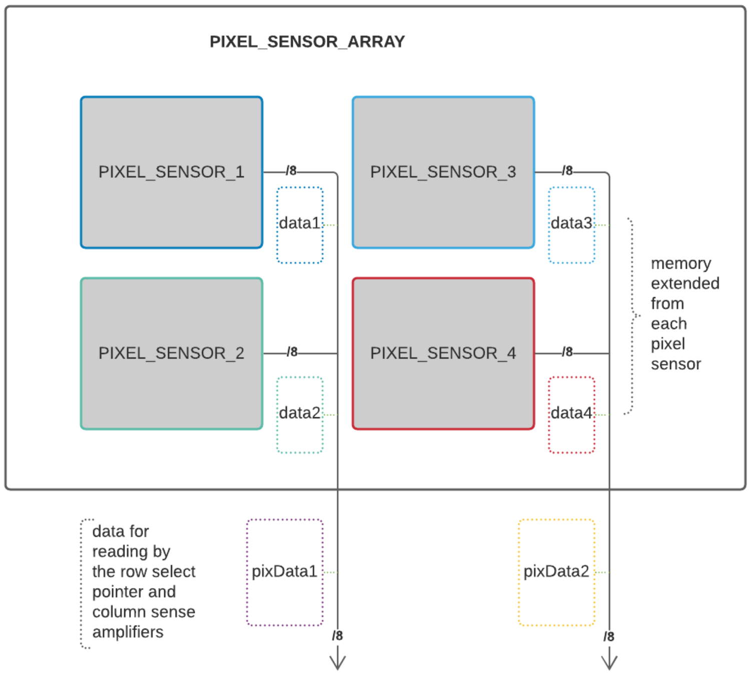


Figure 9: Overview of the digital circuit

4.2 Digital circuit

Figure 13 displays how the system is initialized after startup.

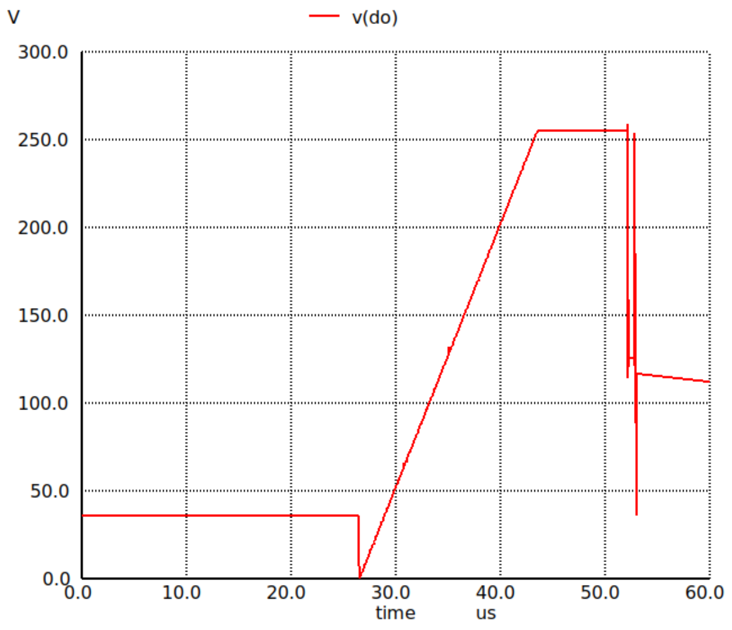


Figure 10: Graph showing the data bus of the pixel

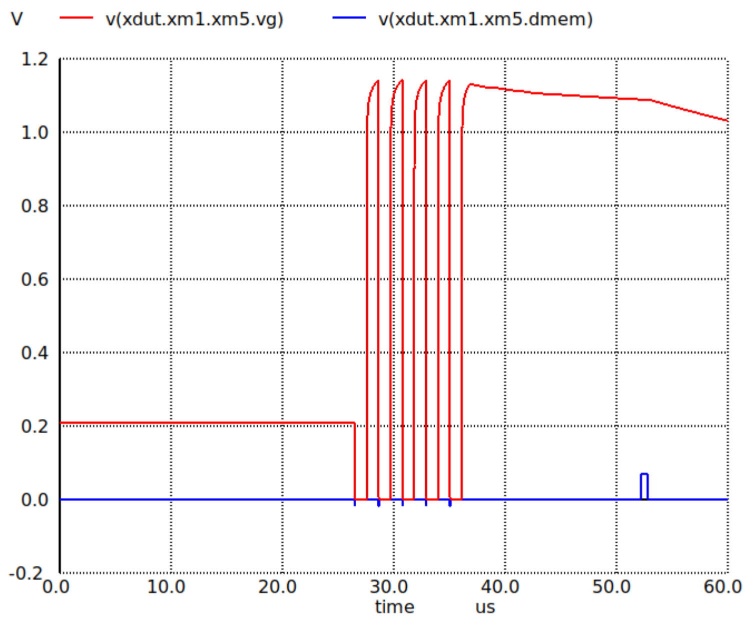


Figure 11: Behavior of the memory cell of the 5th bit

Figure 15 shows clearly that from here that the idle, read and erase states are very short as expected.

Figure 16 shows that the resetting and being incremented as expected.

Figure 17 clearly shows that the data bus is at a high impedance state allowing for reading of data stored in the buffers of the pixel sensors by the row select pointer and column sense amps.

Figure 18 shows convert signals to indicate the start and stop of conversion.

Figure 19 further shows the changes in the states clearly. When the convert signal is high, the analog ramp generates the signals needed for converting the signals from Analog to digital of the ADC. The data bus is driven with these signals into the buffers. As expected, there is a slight

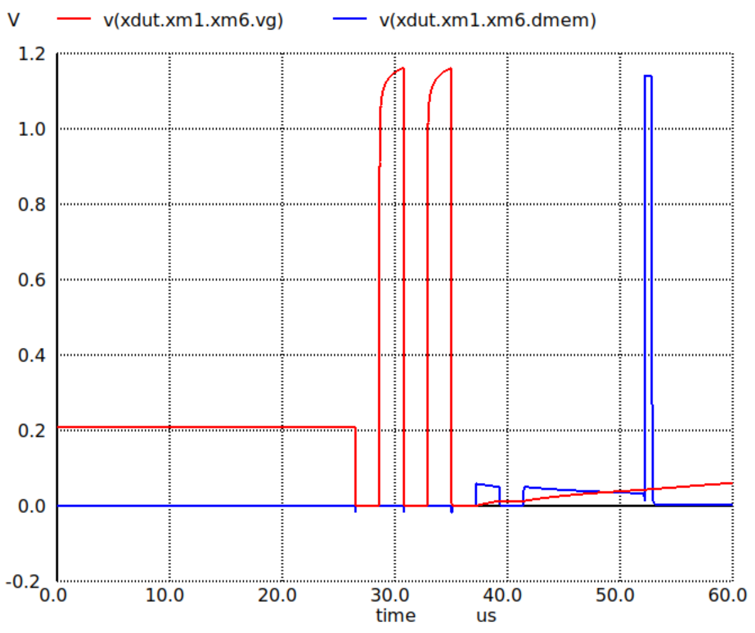
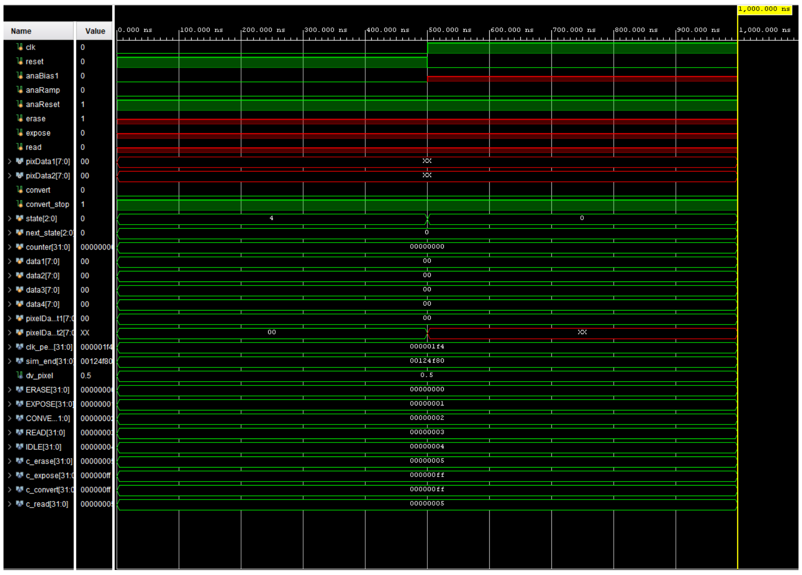


Figure 12: Behavior of the memory cell of the 6th bit



Name	Value
clk	0
reset	0
anaBias1	0
anaRamp	0
anaReset	1
erase	1
expose	0
read	0
pixData1[7:0]	00
pixData2[7:0]	00
convert	0
convert_stop	1
state[2:0]	0
next_state[2:0]	0
counter[31:0]	0000000
data1[7:0]	00
data2[7:0]	00
data3[7:0]	00
data4[7:0]	00
pixelDa...t1[7:0	00
pixelDa...t2[7:0	XX
clk_pe...[31:0]	000001f4
sim_end[31:0]	00124f80
dv_pixel	0.5
ERASE[31:0]	0000000
EXPOSE[31:0]	0000000
CONVE...1:0]	0000000
READ[31:0]	0000000
IDLE[31:0]	0000000
c_erase[31:0]	0000000
c_expose[31:0	000000ff
c_convert[31:0	000000ff
c_read[31:0]	0000000

Figure 13: Plot of when the system is initialized

delay for when data is stored in the buffer.

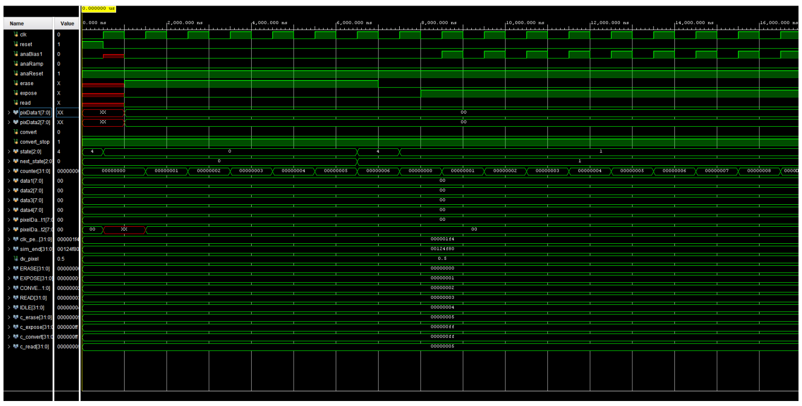


Figure 14: Changes in state upon start up

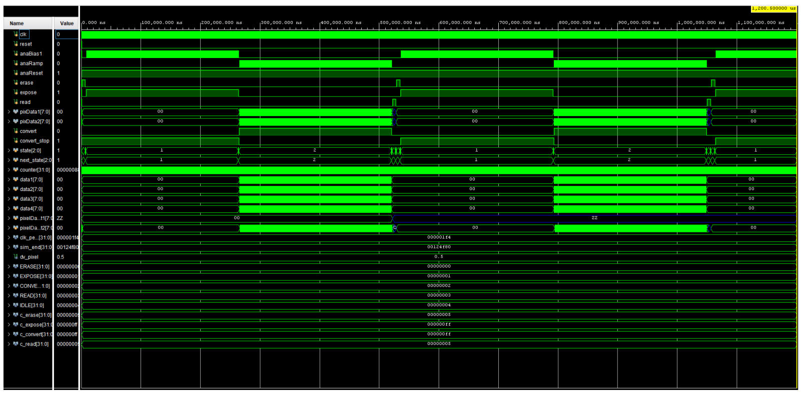


Figure 15: Simulation observed from startup to 1,200,00 ns

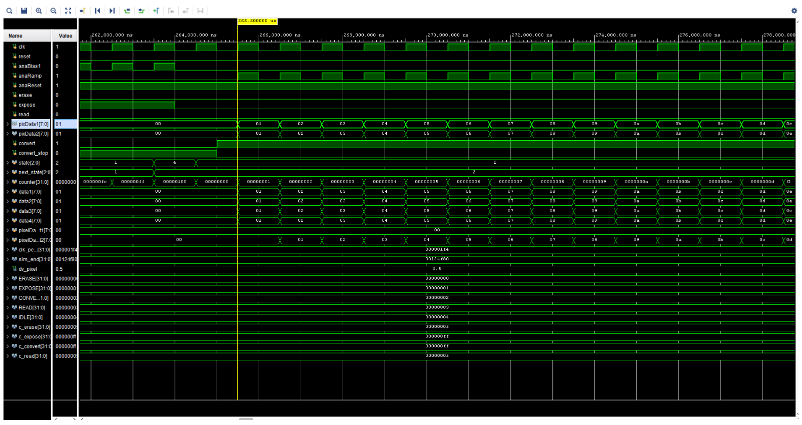


Figure 16: Plots showing some changes in states, counters and data manipulation

5 Discussion

5.1 Analog circuit

The analog circuit was convert the analog value of a simulated photo diode into a digital one. The conversion was expected to give 127 as the result, but instead gave 125. This could be due to a memory effect in the comparator or some other noise source. However for this project this result is acceptable since noise was not a design factor. The memory showed the correct behavior where the latch was able to hold the value properly for a long time. The testbench was able to read the values from the memory without issue.

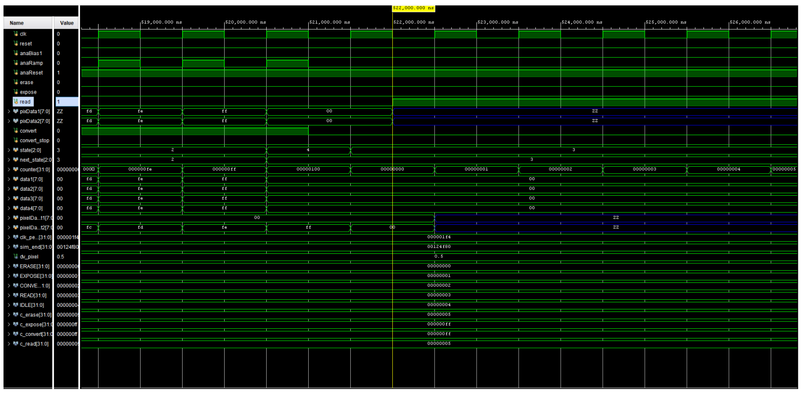


Figure 17: High impedance data bus

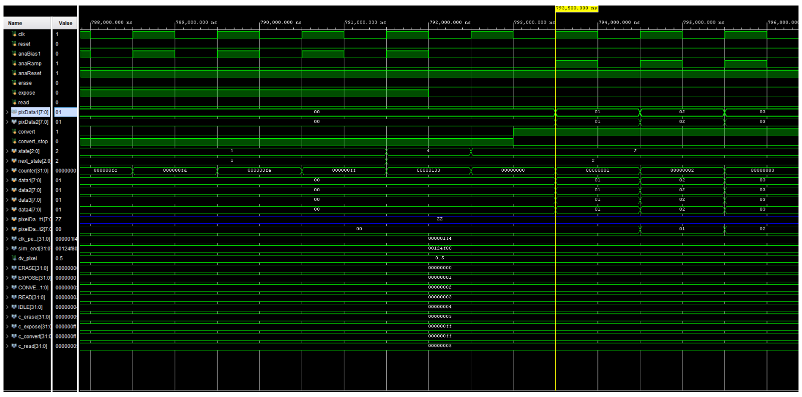


Figure 18: Start and stop of the conversion

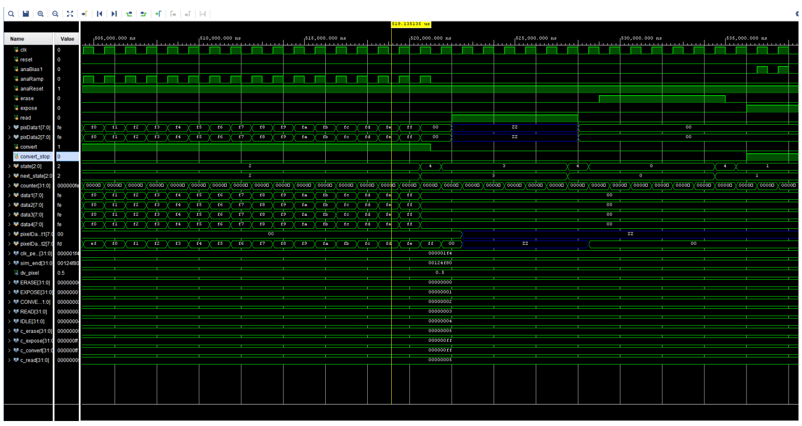


Figure 19: State change example

5.2 Digital circuit

During read operations the data bus is driven with data from the memory of pixel sensors on the same row. All pixel sensors read and convert at the same time. Our design considers the possibility of one of the two pixel sensors on the same column not being functional. This is handled by performing an or operation on the data stored in the memory of the adjacent sensor before been driven unto the bus at any time. Data from each sensor is read out in rows. Sensors sharing the same bus do not read out data at any given time as this is controlled by the memory row read pointer and sense amps. Using the same technique from above, or operation on the data stored in the buffers before being driven on the bus handles this scenario as well. This is valid because all sensors have the same timings for all the five states and are controlled using the same controllers and signals.

The testbenches showed the data conversion we expected. Data was incremented and stored in memory.

We observed an inaccuracy on one of the output buffers. It was always in a high impedance state. We could not really figure why this behaviour was exhibited.

6 Conclusion

A 2x2 pixel array digital sensor was designed and simulated using Spice and System Verilog. Data read out and control is handled by the row select pointer and column sense amps. The analog circuit which was simulated in spice gave satisfactory result for this project. The result showed the correct behavior, but with an inaccuracy in the conversion of the data.

A recommendation for future work on this project will be to use gray code counters to improve power efficiency.

Bibliography

[1] S. Kleinfelder, S. Lim, X. Liu and A. El Gamal, ‘A 10000 frames/s cmos digital pixel sensor’, *IEEE Journal of Solid-State Circuits*, vol. 36, no. 12, pp. 2049–2059, 2001. DOI: 10.1109/4.972156.

Appendix

A Spice Netlist

```
.SUBCKT PIXEL_SENSOR VBN1 VRAMP VRESET ERASE EXPOSE READ
+ DATA_7 DATA_6 DATA_5 DATA_4 DATA_3 DATA_2 DATA_1 DATA_0 VDD VSS

    XS1 VRESET VSTORE ERASE EXPOSE VDD VSS SENSOR

    XC1 VCMP_OUT VSTORE VRAMP VDD VSS VBN1 VBN1 COMP

    XM1 READ VCMP_OUT DATA_7 DATA_6 DATA_5 DATA_4 DATA_3
    + DATA_2 DATA_1 DATA_0 VDD VSS MEMORY

.ENDS

.SUBCKT MEMORY READ VCMP_OUT
+ DATA_7 DATA_6 DATA_5 DATA_4 DATA_3 DATA_2 DATA_1 DATA_0 VDD VSS

    XM1 VCMP_OUT DATA_0 READ VSS MEMCELL
    XM2 VCMP_OUT DATA_1 READ VSS MEMCELL
    XM3 VCMP_OUT DATA_2 READ VSS MEMCELL
    XM4 VCMP_OUT DATA_3 READ VSS MEMCELL
    XM5 VCMP_OUT DATA_4 READ VSS MEMCELL
    XM6 VCMP_OUT DATA_5 READ VSS MEMCELL
    XM7 VCMP_OUT DATA_6 READ VSS MEMCELL
    XM8 VCMP_OUT DATA_7 READ VSS MEMCELL

.ENDS

.SUBCKT MEMCELL CMP DATA READ VSS
    M1 VG CMP DATA VSS nmos  w=0.2u  l=0.13u
    M2 DATA READ DMEM VSS nmos  w=0.4u  l=0.13u
    M3 DMEM VG VSS VSS nmos  w=1u  l=0.13u
    C1 VG VSS 1p
.ENDS
```

```
.SUBCKT SENSOR VRESET VSTORE ERASE EXPOSE VDD VSS

    * Capacitor to model gate-source capacitance
    C1 VSTORE VSS 100f
    Rleak VSTORE VSS 100T

    * Switch to reset voltage on capacitor
    BR1 VRESET VSTORE I=V(ERASE)*V(VRESET,VSTORE)/1k

    * Switch to expose pixel
    BR2 VPG VSTORE I=V(EXPOSE)*V(VSTORE,VPG)/1k

    * Model photocurrent
    Rphoto VPG VSS 1G
.ENDS

.SUBCKT COMP VCMP_OUT VSTORE VRAMP VDD VSS
+VBIAS1 VBIAS2

    *******
    * OTA
    *******
    * current mirror pair
    M1 VMIRROR VMIRROR VDD VDD pmos w=6u l=0.26u
    M2 VOTA_OUT VMIRROR VDD VDD pmos w=6u l=0.26u

    * input pair
    M3 VMIRROR VSTORE VOTABIAS VOTABIAS nmos w=6u l=0.26u
    M4 VOTA VRAMP VOTABIAS VOTABIAS nmos w=6u l=0.26u

    * bias fet
    M5 VOTABIAS VBIAS1 VSS VSS nmos w=6u l=0.26u

    *******
    * Output buffers
    *******
    * bias inverted amplifier
    M6 VOTAINV_OUT VOTA_OUT VDD VDD pmos w=6u l=0.26u
    M7 VOTAINV_OUT VBIAS2 VSS VSS nmos w=6u l=0.26u

    * inverter - buffer
    M8 VCMP_OUT VOTAINV_OUT VDD VDD pmos w=2u l=0.13u
    M9 VCMP_OUT VOTAINV_OUT VSS VSS nmos w=2u l=0.13u

.ENDS
```

B System Verilog Netlist

```
//----------------------------------------------------------------
// Model of pixel sensor, including
//  - Reset
//  - The sensor
//  - Comparator
//  - Memory latch
//  - Readout of latched value
//----------------------------------------------------------------

//EDITED BY ANDREW GLOVER MARTEY AND HENRIK N. DAHLE
//EACH PIXEL SENSOR IS CREATED AS A MODULE
//THE 2X2 ARRAY IS IMPLEMENTED AS A SINGLE MODULE USING THE INDIVIDUAL
//    SENSORS
`timescale 1 ns / 1ps

module PIXEL_SENSOR_1
  (
   input logic      VBN1,
   input logic      RAMP,
   input logic      RESET,
   input logic      ERASE,
   input logic      EXPOSE,
   input logic      READ,
   inout [7:0]      DATA1

   );

   real             v_erase = 1.2;
   real             lsb = v_erase/255;
   parameter real   dv_pixel = 0.5;

   real             tmp;
   logic            cmp;
   real             adc;

   logic [7:0]      p_data;

   //----------------------------------------------------------------
   // ERASE
   //----------------------------------------------------------------
   // Reset the pixel value on pixRst
   always @(ERASE) begin
      tmp = v_erase;
      p_data = 8'b0;
      cmp  = 0;
      adc = 0;
   end

   //----------------------------------------------------------------
   // SENSOR
   //----------------------------------------------------------------
   // Use bias to provide a clock for integration when exposing
   always @(posedge VBN1) begin
      if(EXPOSE)
        tmp = tmp - dv_pixel*lsb;
   end
   //----------------------------------------------------------------
   // Comparator
   //----------------------------------------------------------------
   // Use ramp to provide a clock for ADC conversion, assume that ramp
   // and DATA are synchronous
   always @(posedge RAMP) begin
      adc = adc + lsb;
      if(adc > tmp)
        cmp <= 1;
   end

   //----------------------------------------------------------------
   // Memory latch
   //----------------------------------------------------------------
   always_comb  begin
      if(!cmp) begin
         p_data = DATA1;
      end

   end

   //----------------------------------------------------------------
   // Readout
   //----------------------------------------------------------------
   // Assign data to bus when pixRead = 0
   assign DATA1 = READ ? p_data : 8'bZ;

endmodule // re_control

module PIXEL_SENSOR_2
  (
   input logic      VBN1,
   input logic      RAMP,
   input logic      RESET,
   input logic      ERASE,
   input logic      EXPOSE,
   input logic      READ,
   inout [7:0]      DATA2

   );

   real             v_erase = 1.2;
   real             lsb = v_erase/255;
   parameter real   dv_pixel = 0.5;

   real             tmp;
   logic            cmp;
   real             adc;

   logic [7:0]      p_data;
   //----------------------------------------------------------------
   // ERASE
   //----------------------------------------------------------------
   // Reset the pixel value on pixRst
   always @(ERASE) begin
      tmp = v_erase;
      p_data = 8'b0;
      cmp  = 0;
      adc = 0;
   end

   //----------------------------------------------------------------
   // SENSOR
   //----------------------------------------------------------------
   // Use bias to provide a clock for integration when exposing
   always @(posedge VBN1) begin
      if(EXPOSE)
        tmp = tmp - dv_pixel*lsb;
   end

   //----------------------------------------------------------------
   // Comparator
   //----------------------------------------------------------------
   // Use ramp to provide a clock for ADC conversion, assume that ramp
   // and DATA are synchronous
   always @(posedge RAMP) begin
      adc = adc + lsb;
      if(adc > tmp)
        cmp <= 1;
   end

   //----------------------------------------------------------------
   // Memory latch
   //----------------------------------------------------------------
   always_comb  begin
      if(!cmp) begin
         p_data = DATA2;
      end

   end

   //----------------------------------------------------------------
   // Readout
   //----------------------------------------------------------------
   // Assign data to bus when pixRead = 0
   assign DATA2 = READ ? p_data : 8'bZ;

endmodule // re_control
```

```
module PIXEL_SENSOR_3
  (
   input logic      VBN1,
   input logic      RAMP,
   input logic      RESET,
   input logic      ERASE,
   input logic      EXPOSE,
   input logic      READ,
   inout [7:0]      DATA3

   );

   real             v_erase = 1.2;
   real             lsb = v_erase/255;
   parameter real   dv_pixel = 0.5;

   real             tmp;
   logic            cmp;
   real             adc;

   logic [7:0]      p_data;

   //----------------------------------------------------------------
   // ERASE
   //----------------------------------------------------------------
   // Reset the pixel value on pixRst
   always @(ERASE) begin
      tmp = v_erase;
      p_data = 0;
      cmp  = 0;
      adc = 0;
   end

   //----------------------------------------------------------------
   // SENSOR
   //----------------------------------------------------------------
   // Use bias to provide a clock for integration when exposing
   always @(posedge VBN1) begin
      if(EXPOSE)
        tmp = tmp - dv_pixel*lsb;
   end

   //----------------------------------------------------------------
   // Comparator
   //----------------------------------------------------------------
   // Use ramp to provide a clock for ADC conversion, assume that ramp
   // and DATA are synchronous
   always @(posedge RAMP) begin
      adc = adc + lsb;
      if(adc > tmp)
        cmp <= 1;
   end
   //----------------------------------------------------------------
   // Memory latch
   //----------------------------------------------------------------
   always_comb  begin
      if(!cmp) begin
         p_data = DATA3;
      end

   end

   //----------------------------------------------------------------
   // Readout
   //----------------------------------------------------------------
   // Assign data to bus when pixRead = 0
   assign DATA3 = READ ? p_data : 8'bZ;

endmodule // re_control

module PIXEL_SENSOR_4
  (
   input logic      VBN1,
   input logic      RAMP,
   input logic      RESET,
   input logic      ERASE,
   input logic      EXPOSE,
   input logic      READ,
   inout [7:0]      DATA4

   );

   real             v_erase = 1.2;
   real             lsb = v_erase/255;
   parameter real   dv_pixel = 0.5;

   real             tmp;
   logic            cmp;
   real             adc;

   logic [7:0]      p_data;

   //----------------------------------------------------------------
   // ERASE
   //----------------------------------------------------------------
   // Reset the pixel value on pixRst
   always @(ERASE) begin
      tmp = v_erase;
      p_data = 0;
      cmp  = 0;
      adc = 0;
   end
   //----------------------------------------------------------------
   // SENSOR
   //----------------------------------------------------------------
   // Use bias to provide a clock for integration when exposing
   always @(posedge VBN1) begin
      if(EXPOSE)
        tmp = tmp - dv_pixel*lsb;
   end

   //----------------------------------------------------------------
   // Comparator
   //----------------------------------------------------------------
   // Use ramp to provide a clock for ADC conversion, assume that ramp
   // and DATA are synchronous
   always @(posedge RAMP) begin
      adc = adc + lsb;
      if(adc > tmp)
        cmp <= 1;
   end

   //----------------------------------------------------------------
   // Memory latch
   //----------------------------------------------------------------
   always_comb  begin
      if(!cmp) begin
         p_data = DATA4;
      end

   end

   //----------------------------------------------------------------
   // Readout
   //----------------------------------------------------------------
   // Assign data to bus when pixRead = 0
   assign DATA4 = READ ? p_data : 8'bZ;

endmodule // re_control

module PIXEL_SENSOR_ARRAY (VBN1, RAMP, RESET, ERASE, EXPOSE, READ,DATA1,DATA2,
        DATA3,DATA4,DATAX,DATAY); // A 2X2 PIXEL ARRAY IS IMPLEMENTED HERE

     input VBN1, RAMP, RESET, ERASE, EXPOSE, READ;
     inout DATA1,DATA2,DATA3,DATA4;
     output DATAX,DATAY;
     assign DATAX = DATA1 | DATA2;
     assign DATAY = DATA3 | DATA4;
     PIXEL_SENSOR_1  PS1 (VBN1, RAMP, RESET, ERASE, EXPOSE, READ, DATA1);
     PIXEL_SENSOR_2  PS2 (VBN1, RAMP, RESET, ERASE, EXPOSE, READ, DATA2);
     PIXEL_SENSOR_3  PS3 (VBN1, RAMP, RESET, ERASE, EXPOSE, READ, DATA3);
     PIXEL_SENSOR_4  PS4 (VBN1, RAMP, RESET, ERASE, EXPOSE, READ, DATA4);

endmodule
```

```
//EDITED BY ANDREW GLOVER MARTEY AND HENRIK N. DAHLE
//EACH PIXEL SENSOR IS CREATED AS A MODULE
//THE 2X2 ARRAY IS IMPLEMENTED AS A SINGLE MODULE USING THE INDIVIDUAL SENSORS

`timescale 1 ns / 1 ps

module pixelSensorFsm(
                       input  logic        clk,
                       input  logic      reset,
                       output logic      erase,
                       output logic     expose,
                       output logic       read,
                       output logic    convert

                      );


     //State duration in clock cycles
   parameter integer c_erase = 5;
   parameter integer c_expose = 255;
   parameter integer c_convert = 255;
   parameter integer c_read = 5;

   //------------------------------------------------------------
   // State Machine
   //------------------------------------------------------------
   parameter ERASE=0, EXPOSE=1, CONVERT=2, READ=3, IDLE=4;


   logic                 convert_stop;
   logic [2:0]           state,next_state;   //States
   integer               counter;
```

```
// Control the output signals
always_ff @(negedge clk ) begin
   case(state)
     ERASE: begin
        erase <= 1;
        read <= 0;
        expose <= 0;
        convert <= 0;
     end
     EXPOSE: begin
        erase <= 0;
        read <= 0;
        expose <= 1;
        convert <= 0;
     end
     CONVERT: begin
        erase <= 0;
        read <= 0;
        expose <= 0;
        convert = 1;
     end
     READ: begin
        erase <= 0;
        read <= 1;
        expose <= 0;
        convert <= 0;
     end
     IDLE: begin
        erase <= 0;
        read <= 0;
        expose <= 0;
        convert <= 0;

     end
   endcase // case (state)
end // always @ (state)


// Control the state transitions.
//TODO: The counter should probably be an always_comb. Might be a good idea
//to also reset the counter from the state machine, i.e provide the count
//down value, and trigger on 0
```

```
   always_ff @(posedge clk or posedge reset) begin
      if(reset) begin
         state = IDLE;
         next_state = ERASE;
         counter  = 0;
         convert  = 0;
      end
      else begin
         case (state)
           ERASE: begin
              if(counter == c_erase) begin
                 next_state <= EXPOSE;
                 state <= IDLE;
              end
           end
           EXPOSE: begin
              if(counter == c_expose) begin
                 next_state <= CONVERT;
                 state <= IDLE;
              end
           end
           CONVERT: begin
              if(counter == c_convert) begin
                 next_state <= READ;
                 state <= IDLE;
              end
           end
           READ:
             if(counter == c_read) begin
                state <= IDLE;
                next_state <= ERASE;
             end
           IDLE:
             state <= next_state;
         endcase // case (state)
         if(state == IDLE)
           counter = 0;
         else
           counter = counter + 1;
      end
   end // always @ (posedge clk or posedge reset)

endmodule // test
```

C System Verilog testbench

```
//EDITED BY ANDREW GLOVER MARTEY AND HENRIK N. DAHLE
//EACH PIXEL SENSOR IS CREATED AS A MODULE
//THE 2X2 ARRAY IS IMPLEMENTED AS A SINGLE MODULE USING THE INDIVIDUAL SENSORS

`timescale 1 ns / 1 ps

//======================================================================
// Testbench for pixelSensor
// - clock
// - instanciate pixel
// - State Machine for controlling pixel sensor
// - Model the ADC and ADC
// - Readout of the databus
// - Stuff neded for testbench. St+ore the output file etc.
//======================================================================
module pixelSensor_tb;

   //------------------------------------------------------------
   // Testbench clock
   //------------------------------------------------------------
   logic clk =0;
   logic reset =0;
   parameter integer clk_period = 500;
   parameter integer sim_end = clk_period*2400;
   always #clk_period clk=~clk;

   //------------------------------------------------------------
   // Pixel
   //------------------------------------------------------------
   parameter real    dv_pixel = 0.5;  //Set the expected photodiode current (0-1)

   //Analog signals
   logic              anaBias1;
   logic              anaRamp;
   logic              anaReset;

   //Tie off the unused lines
   assign anaReset = 1;

   //Digital
   logic              erase;
   logic              expose;
   logic              read;
   tri[7:0]         pixData1; //  We need this to be a wire, because we're tristating it
   tri[7:0]         pixData2;
```

```
//------------------------------------------------------------
// State Machine
//------------------------------------------------------------
parameter ERASE=0, EXPOSE=1, CONVERT=2, READ=3, IDLE=4;

logic               convert;
logic               convert_stop;
logic [2:0]         state,next_state;   //States
integer             counter;            //Delay counter in state machine

//State duration in clock cycles
parameter integer c_erase = 5;
parameter integer c_expose = 255;
parameter integer c_convert = 255;
parameter integer c_read = 5;

// Control the output signals
always_ff @(negedge clk ) begin
   case(state)
     ERASE: begin
        erase <= 1;
        read <= 0;
        expose <= 0;
        convert <= 0;
     end
     EXPOSE: begin
        erase <= 0;
        read <= 0;
        expose <= 1;
        convert <= 0;
        convert_stop <= 1;
     end
     CONVERT: begin
        erase <= 0;
        read <= 0;
        expose <= 0;
        convert <= 1;
        convert_stop <= 0;
     end
     READ: begin
        erase <= 0;
        read <= 1;
        expose <= 0;
        convert <= 0;
     end
     IDLE: begin
        erase <= 0;
        read <= 0;
        expose <= 0;
        convert <= 0;

     end
   endcase // case (state)
end // always @ (state)
```

```
// Control the state transitions.
//TODO: The counter should probably be an always_comb. Might be a good idea
//to also reset the counter from the state machine, i.e provide the count
//down value, and trigger on 0
always_ff @(posedge clk or posedge reset) begin
   if(reset) begin
      state = IDLE;
      next_state = ERASE;
      counter  = 0;
      convert  = 0;
      convert_stop = 1;
   end
   else begin
      case (state)
        ERASE: begin
           if(counter == c_erase) begin
              next_state <= EXPOSE;
              state <= IDLE;
           end
        end
        EXPOSE: begin
           if(counter == c_expose) begin
              next_state <= CONVERT;
              state <= IDLE;
           end
        end
        CONVERT: begin
           if(counter == c_convert) begin
              next_state <= READ;
              state <= IDLE;
           end
        end
        READ:
          if(counter == c_read) begin
             state <= IDLE;
             next_state <= ERASE;
          end
        IDLE:
          state <= next_state;
      endcase // case (state)
      if(state == IDLE)
        counter = 0;
      else
        counter = counter + 1;
   end
end // always @ (posedge clk or posedge reset)
```

```
//------------------------------------------------------------
// DAC and ADC model
//------------------------------------------------------------
logic[7:0] data1;
logic[7:0] data2;
logic[7:0] data3;
logic[7:0] data4;

// If we are to convert, then provide a clock via anaRamp
// This does not model the real world behavior, as anaRamp would be a voltage from the ADC
// however, we cheat
assign anaRamp = convert ? clk : 0;

// During expoure, provide a clock via anaBias1.
// Again, no resemblence to real world, but we cheat.
assign anaBias1 = expose ? clk : 0;

// If we're not reading the pixData, then we should drive the bus

assign pixData1 = read ? 8'bZ: data1|data2;
assign pixData2 = read ? 8'bZ: data3|data4;




// When convert, then run a analog ramp (via anaRamp clock) and digtal ramp via
// data bus.
always_ff @(posedge clk or posedge reset) begin
   if(reset) begin
      data1 =0;
      data2 =0;// Data registors(memory) is initialized to zero
      data3 =0;
      data4 =0;
   end
   if(convert) begin  //Data is computed by doing increments of 1
      data1 +=  1;
      data2 +=  1;/
      data3 +=  1;
      data4 +=  1;
   end
   else begin    // Data registors(memory) is initialized to zero
      data1 = 0;
      data2 = 0;
      data3 = 0;
      data4 = 0;
   end
end // always @ (posedge clk or reset)
```

```
   //------------------------------------------------------------
   // Readout from databus
   //------------------------------------------------------------
   logic [7:0] pixelDataOut1;
   logic [7:0] pixelDataOut2;
   always_ff @(posedge clk or posedge reset) begin
      if(reset) begin            // OutputData registors are initialized to zero
         pixelDataOut1 = 0;
         pixelDataOut1 = 0;
         pixelDataOut2 = 0;
      end
      else begin
         if(read)         //Data on the pixData buses are written to temporal registers
           pixelDataOut1 <= pixData1;
           pixelDataOut2 <= pixData2;
      end
   end

   //------------------------------------------------------------
   // Testbench stuff
   //------------------------------------------------------------
   //PIXEL_SENSOR DUT ();
   initial
     begin
        reset = 1;
        #clk_period  reset=0;

        $dumpfile("pixelSensor_tb.vcd");
        $dumpvars(0,pixelSensor_tb);

        #sim_end
          $stop;

     end

endmodule // test
```

```
//EDITED BY ANDREW GLOVER MARTEY AND HENRIK N. DAHLE
//EACH PIXEL SENSOR IS CREATED AS A MODULE
//THE 2X2 ARRAY IS IMPLEMENTED AS A SINGLE MODULE USING THE INDIVIDUAL SENSORS

`timescale 1 ns / 1 ps

//====================================================================
// Testbench for pixelSensor
// - clock
// - instanciate pixel
// - State Machine for controlling pixel sensor
// - Model the ADC and ADC
// - Readout of the databus
// - Stuff neded for testbench. Store the output file etc.
//====================================================================
module pixelSensor_tb;

   //------------------------------------------------------------
   // Testbench clock
   //------------------------------------------------------------
   logic clk =0;
   logic reset =0;
   parameter integer clk_period = 500;
   parameter integer sim_end = clk_period*2400;
   always #clk_period clk=~clk;

   //------------------------------------------------------------
   // Pixel
   //------------------------------------------------------------
   parameter real    dv_pixel = 0.5;  //Set the expected photodiode current (0-1)

   //Analog signals
   logic              anaBias1;
   logic              anaRamp;
   logic              anaReset;

   //Tie off the unused lines
   assign anaReset = 1;

   //Digital
   wire              erase;
   wire              expose;
   wire             read;
   wire             convert;

   tri[7:0]         pixData1; //  We need this to be a wire, because we're tristating it
   tri[7:0]         pixData2;
```

```
//Instanciate the pixel SENSOR ARRAY
PIXEL_SENSOR_ARRAY  #(.dv_pixel(dv_pixel))  PSS(anaBias1, anaRamp, anaReset, erase,
     expose, read,pixData1,pixData2);

pixelSensorFsm #(.c_erase(5),.c_expose(255),.c_convert(255),.c_read(5))
     fsm1(.clk(clk),.reset(reset),.erase(erase),.expose(expose),.read(read),
     .convert(convert));


//------------------------------------------------------------
// DAC and ADC model
//------------------------------------------------------------
logic[7:0] data1;
logic[7:0] data2;
logic[7:0] data3;
logic[7:0] data4;

// If we are to convert, then provide a clock via anaRamp
// This does not model the real world behavior, as anaRamp would be a voltage from the ADC
// however, we cheat
assign anaRamp = convert ? clk : 0;

// During expoure, provide a clock via anaBias1.
// Again, no resemblence to real world, but we cheat.
assign anaBias1 = expose ? clk : 0;

// If we're not reading the pixData, then we should drive the bus

assign pixData1 = read ? 8'bZ: data1|data2;
assign pixData2 = read ? 8'bZ: data3|data4;

// When convert, then run a analog ramp (via anaRamp clock) and digtal ramp via
// data bus.
always_ff @(posedge clk or posedge reset) begin
   if(reset) begin // Data registors(memory) is erased
      data1 =0;
      data2 =0;
      data3 =0;
      data4 =0;
   end
   if(convert) begin    // Data registors(memory) is initialized to zero
      data1 +=  1;
      data2 +=  1;
      data3 +=  1;
      data4 +=  1;
   end
   else begin   // Data registors(memory) is initialized to zero
      data1 = 0;
      data2 = 0;
      data3 = 0;
      data4 = 0;
   end
end // always @ (posedge clk or reset)
```

[breaklines,frame=single]

```
   //----------------------------------------------------------------
   // Readout from databus
   //----------------------------------------------------------------
   logic [7:0] pixelDataOut1;
   logic [7:0] pixelDataOut2;
   always_ff @(posedge clk or posedge reset) begin
      if(reset) begin           // OutputData registors are initialized to zero
         pixelDataOut1 = 0;
         pixelDataOut2 = 0;
      end
      else begin
         if(read)               //Data on the pixData buses are written to temporal registers
           pixelDataOut1 <= pixData1;
           pixelDataOut2 <= pixData2;
      end
   end
   //----------------------------------------------------------------
   // Testbench stuff
   //----------------------------------------------------------------
   initial
     begin
        reset = 1;
        #clk_period  reset=0;

        $dumpfile("pixelSensor_tb.vcd");
        $dumpvars(0,pixelSensor_tb);

        #sim_end
          $stop;

     end

endmodule // test
```