A 10bit, 100×100 pixel Photosensor

Carl Thyness, *Student, NTNU* and Daniel Vorhaug, *Student, NTNU*

***Abstract*—A 10-bit 100×100p photosensor is described. The design consisting of two parts that are simulated seperately. One part being a single pixel with a 10-bit digital-output, simulated in SPICE. The other being the digital system, including the array of pixels, all the necessary logic for correct timings, proper readout and control of the analog part. The digital part is simulated using SystemVerilog, and is scalable to any resoulution, bit depth and output bus width.**

I. Introduction

There are a number of different image sensor architectures that can be used to turn the signal on an array of photodiodes or phototransistors into a digital signal. One common method is to have one analog-to-digital converter (ADC) per column of pixels [1]. Another method, which the design discussed here uses, is a digital pixel sensor (DPS) [2], which has one ADC per pixel. Having one ADC per pixel has the advantage of avoiding column fixed-pattern noise that occurs with one ADC per column [2]. Unfortunately, as having an ADC in each pixel requires more transistor space, this could leave less space for a photodiode or phototransistor. However, using backside illumination, and using a stacked process which places the image sensors and ADCs on separate pieces of silicon that are then sandwiched together increases the usable image sensor area[3].

In this paper, we describe the design of a photosensor where the pixel-array resolution, the bit-depth of each pixel and output-bus-width is adjustable by changing a single parameter for each. The chip we will present has a resolution of 100x100 pixels, a bit depth of 10bit and an output-bus of 100bit (10 pixels). The design is based on the 2001 design by Kleinfelder, Lim, Liu and Gamal [2]. Our design mainly differs in the specifications it was made for while most of the implementation is similar if not the same. In implementation the main differences are the modularity of our chip and that we allow for less outside control of the chip. The last part will be done primarily for chip density and security. The resolution and bit depth will allow for use in more detail-demanding work.

II. Theory

Our design will be based on the one detailed in [2]. No further knowledge is required.

III. Implementation

For simplicity's sake the design has been split into two parts. An analog part consisting of a single pixel, and a digital part containing an array of a simplified model of the analog pixel and the necessary logic to read the pixel data.

A. Analog

Our analog design consist of three important subcircuits: The actual sensor, the per pixel comparator and the per pixel memory. Each playing their part in getting a digital signal from every pixel. All circuits are based on the ones detailed in [2] and specifically the models made by Carsten Wulff [4].

1) Photosensor: The photosensor consists of six transistors, one phototransistor to convert photon radiation into an electrical signal and five transistors to handle the Read/Erase logic. The circuit is as shown in figure 1.

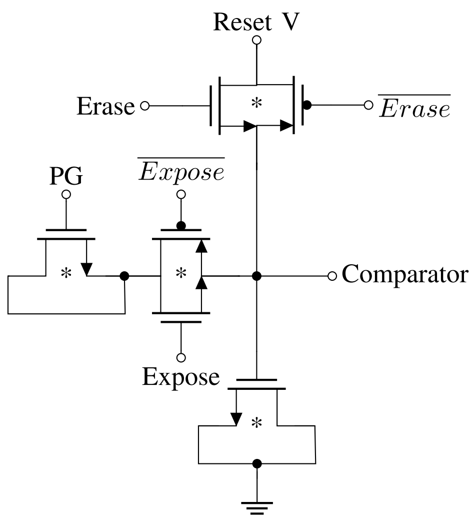


Fig. 1: Photosensor

The expose and erase transistors all have the same dimensions of w=0.5μm and l=0.15μm. The nmos are single transistors while the pmos are four transistors in parallel for as close to equal driving force as possible. We chose to model the phototransistor as a resistor and current source, while the lower transistor is used as a capacitor and we have therefore modelled it as such. When exposed a voltage will be built up over the phototransistor giving us a value dependent on the light levels hitting the pixel.

2) Comparator: The comparator is part of the per pixel ADC in our system. It is set up as a differential pair amplifier connected with two inverters. The two inverters are meant to amplify the signal from the differential pair, ensuring that even a small voltage difference flips the output completely between logical low and high. One input is the voltage from the sensor and one input is an analog ramp. The output of the comparator goes to the memory and is high when the sensor voltage is higher than the analog ramp. The circuit can be seen in figure 2.

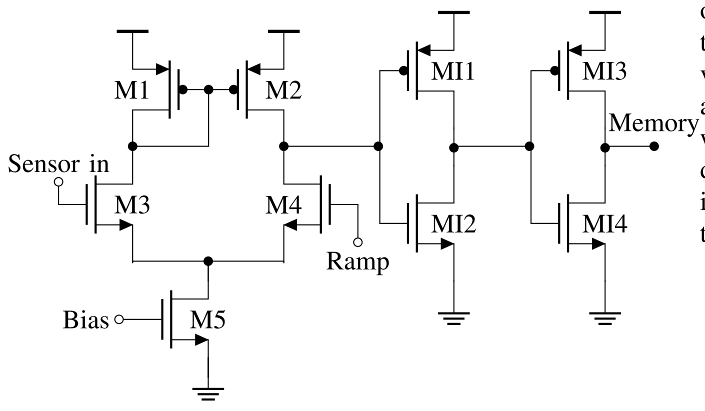


Fig. 2: Comparator.

The transistor M5 is part of a current mirror, allowing us to control the current passing through the comparator. We set the current through this transistor to 6μA. All transistors have the same width of 0.5μm. The transistor in current mirrors have lengths 0.5μm while the differential pair nmos and the inverter transistors all have length 0.15μm. The current mirror transistors M1, M2 and M5 are all single transistors, giving us a low ratio between width and length which again ensures that they are all in strong inversion. The differential pair transistors M3 and M4 are both two transistors in parallel. This keeps them all in weak inversion. For the inverters we made the nmos (MI2 and MI4) single transistors while the pmos (MI1 and MI3) are four in parallel. This is to ensure close to equal driving force.

3) Memory: Each pixel has a 10bit memory, each bit is held by a 3t memory cell. Each pixel therefore have 10 memory cells, giving us a total of 30 transistors for the memory of each pixel. The circuit diagram of our 3t memory cells can be seen in figure 3.

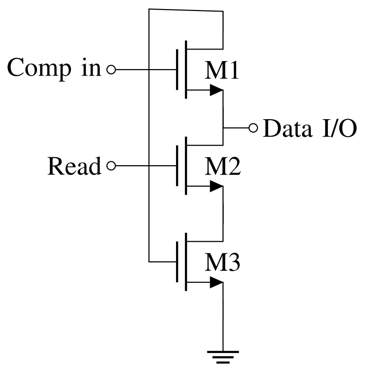


Fig. 3: 3t memory cell.

When the signal from the comparator goes low the current value in each memory cell is held until it is either read out or the comparator signal goes high again. The data held in the memory is from our gray code counter, giving us a digital value for the pixel. All transistors have a length of 0.13μm and are single transistors. When it comes to width M1 has w=0.2μm, M2 has w=0.4μm and M3 has w=1μm. These dimensions were chosen to minimise the footprint, especially important due to the large amount of transistors required by the memory.

B. Digital

The digital part of the design is made in the hardware description language SystemVerilog. The sensor is split into modules shown in figure 4. The *Counter* counts up on the vertical buses and the pixels use this to digitize their analog readings. When this data is to be read out of the pixels, the *Memory controller* uses the *Row pointer* to tell which pixel row should be read out and uses the *Output bus* to put the data on a smaller bus to be written to the outside.

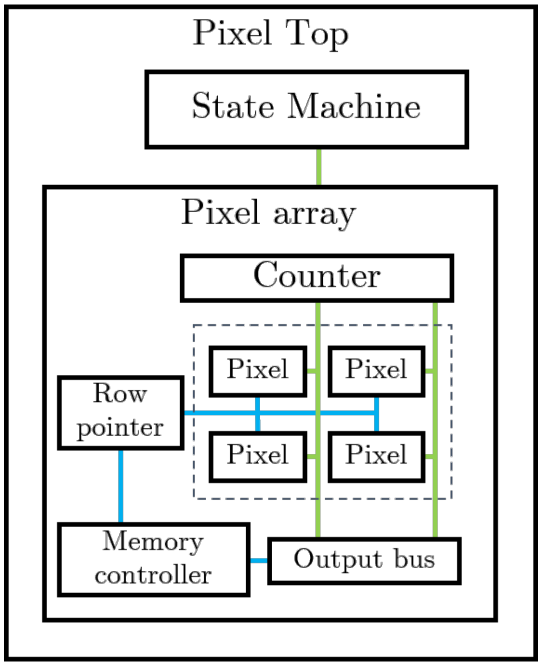


Fig. 4: Illustration of the modules in the SystemVerilog project and how they are connected. The pixel array in the dashed box shows for when it is compiled for a 2x2 array.

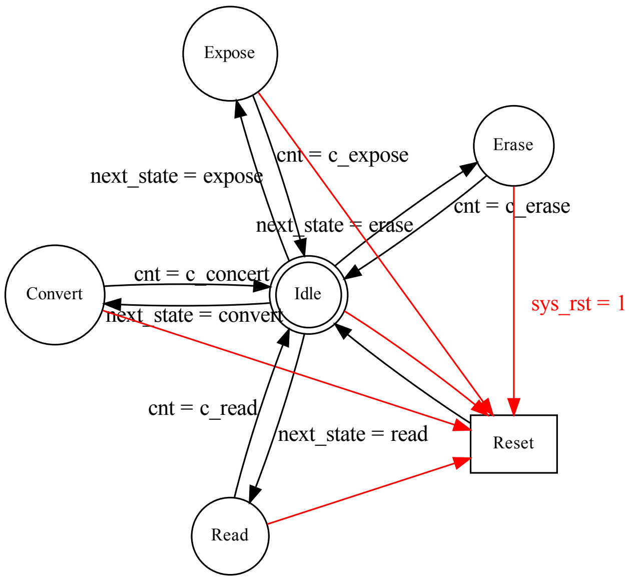


Fig. 5: State diagram of the main state machine.

1) Pixel Top: The system interfaces with the world through the top-level modules' inputs and outputs. The top module takes in a system clock, driving the modules' own, and a reset signal. It outputs a data-bus and a clock. The clock signals when data is put on the bus. Size of the output-bus is configurable by changing a single parameter, but is set to 100 bit which is 10 10-bit pixels wide. Apart from this, the module only connects the state machine to the pixel array module.

2) State machine: A state machine module controls the timing of an image capture. The states and transitions are shown in figure 5. It uses a counter to time how many clock cycles it has been in each state and to keep track on when to transition to the next state. The state machine uses the rising edge of the system clock to increase a counter and check if it should transition. Then it uses the falling edge to apply the output signals the current state entails.

3) Pixel array: The pixel array contains all the individual pixels and necessary logic to act upon the signals sent from the state machine module. In it we have each individual pixel (sorted in a grid), the counter, the memory controller, the row pointer and the output bus. The three last submodules are all used to control the output from the pixel onto the outpus bus.

4) Pixel: In Verilog we only have a model of how the actual analog pixel works. It is based on the model made by Carsten Wulff [4]. The model works by lowering the voltage value into the comparator by a constant value per clock cycle it is told to "expose" the phototransistor. The analog ramp is modeled as a value incrementing by a constant value per clock cycle we convert the analog signal. When the analog ramp gains a higher value than the one coming from the phototransistor we set the value out of the comparator low. The memory is continuously overwritten as long as the signal from the comparator is high and locks the current value when the comparator output signal goes low. Lastly the signal out of the pixel is set to high impedance when READ is low and to the inverted of the value currently in the memory when READ is high.

5) Counter: The counter give values in Gray code. It is written as a 10 bit full adder, adding 1 on the rising edge of the system clock. The current value is then converted into Gray code and sent out. When RESET is set high the counter is set to 0.

6) Memory controller: The memory controller is a four state finite state machine. It is an updated version of the one made by Carsten Wulff [4]. The states and their transitions can be seen in figure 6.

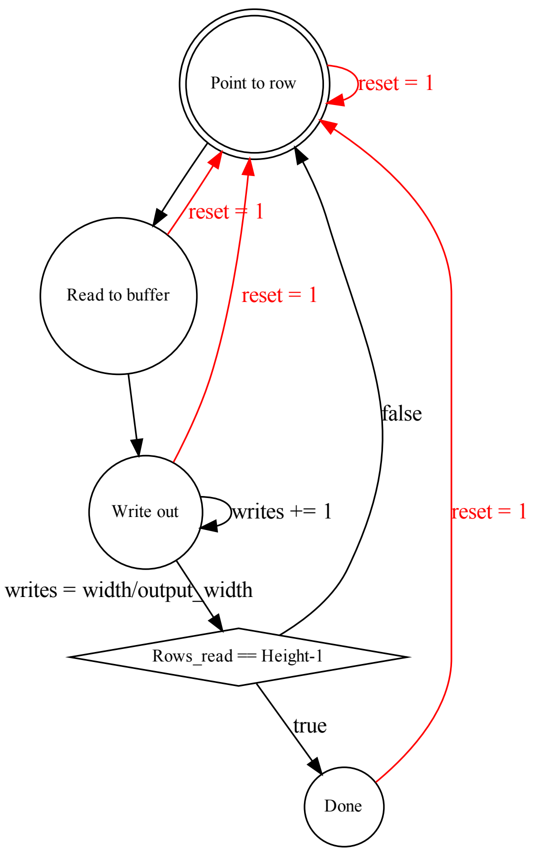


Fig. 6: State diagram of the memory controller.

First it sends a signal to the read pointer making it have one row write its' memory value onto the data bus. Second, during the next clock cycle, it has the output bus read this signal onto its' buffer. Third it has the output bus read the signal out sequentially (more on this in III-B8). Then the memory controller goes through these three states until all rows have been read out. Then it finally reaches the fourth state: Done. If the memory controller ever receives a reset signal it is reverted back to the Point to row state.

7) Row pointer: The row pointer takes in a binary value from the memory controller and through shifting sends out individual signals to each pixel row.

8) Output bus: The output bus is a parallel in, series out bus. It takes all values from a row into its buffer in parallel.. Then, as the data has been inverted when read out of the pixel memory, the data is inverted back in this module. After this the data is converted from Gray code back into natural binary code. Finally it reads the data out sequentially. The bus takes in 1kbit controlled by a read clock and then writes out 100 bit each clock cycle of a write clock coming from the memory controller.

IV. Result

A. Analog section

1) Current: Measurements of currents through one pixel can be seen in figure 7.

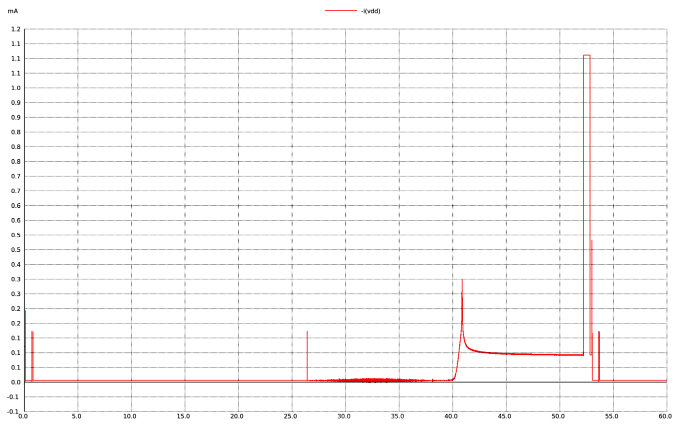


Fig. 7: Measured current through an individual pixel.

The current consumption varies a lot depending on the state, but most of the time there is simply an idle current, as can be seen in figure 8.

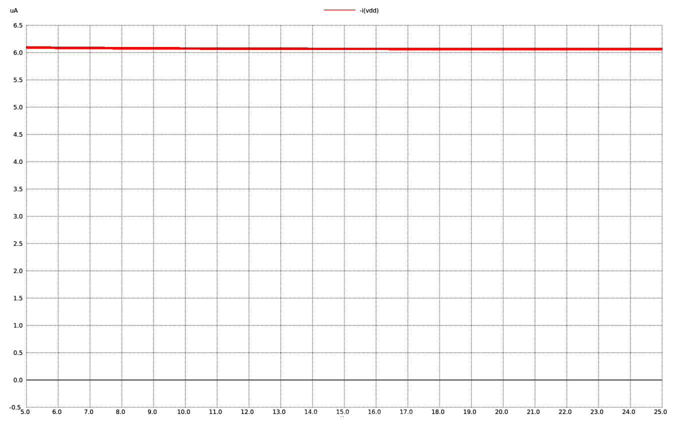


Fig. 8: Idle current of a single pixel.

The idle current is approximately 6μA while the peak current consumption reaches a bit above $1.1mA$. That specific peak is caused by the readout of the memory. This because high bits in the memory causes a short circuit when read out, more on this in IV-A4. A 100×100 array would because of this have an idle current consumption of about 60mA and a peak current consumption of 11A from the pixel array, in addition to the digital logic. Other noticable peaks are:

- At 41μs, which reaches 350μA and is caused by the comparator flipping the output value.
- At the start and at 54μs, these reach up to 175μA and are caused by the erasure of former values in the memory and the begining of a new exposure process.
- At 26μs which reaches 175μA and coincides with convert going high and the analog ramp starting.

The current between 41μs and 55μs, the peaks caused by the comparator flipping and the memory being read out, is between 90 and 100 μA. The absolute peak current is dependent on the value of the pixel as the more bits that are high the more short circuits and the more current drawn.

2) Photosensor verification: The storage capacitor should be charged to *Reset V*$=1.1$V during the erase phase. Figure 9 shows that this is the case, but that turning off the transmission gate causes a voltage jump of 12mV and $\frac{12\text{mV}}{1100\text{mV}} = 1.10\%$.

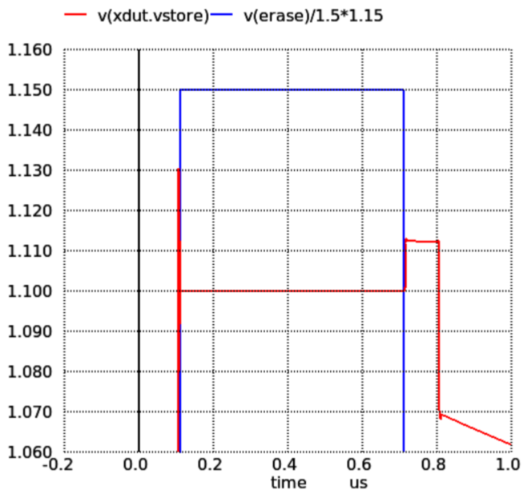


Fig. 9: Plot of storage capacitor voltage during the erase phase. Erase signal voltage is normalized to 1.15V.

During the expose part of taking a sample, the voltage across the storage capacitor should have time to converge. A plot of this is shown in figure 10. The figure shows voltage has almost converged, but also that turning on and off the expose transmission gate causes a ripple in the storage capacitor voltage. Turning it on causes a 40mV ($\frac{40\text{mV}}{1112\text{mV}} = 3.60\%$) drop and turning it off again causes a 22mV ($\frac{22\text{mV}}{853\text{mV}} = 2.58\%$) rise. When added with the jump from the erase phase these changes almost cancel each other out as $1.10\% - 3.60\% + 2.58\% = 0.08\%$

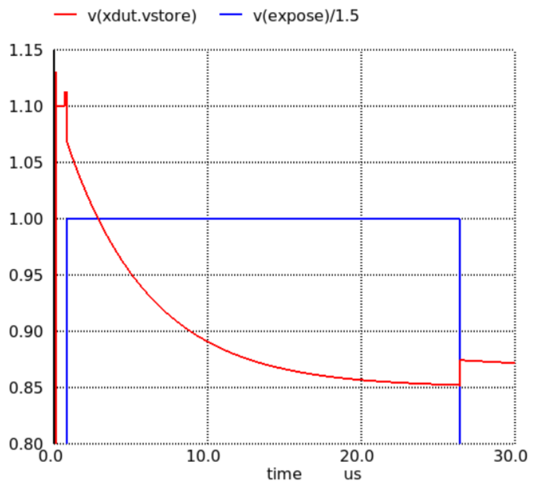


Fig. 10: Plot of storage capacitor voltage during the expose phase. Expose signal voltage is normalized to 1V.

The capacitor should also hold its charge during the convert phase. The plot in figure 11 indicates that the voltage falls by $875.0mV - 856.5mV = 18.5mV$ and $\frac{18.5mV}{875.0mV} = 2.1\%$.

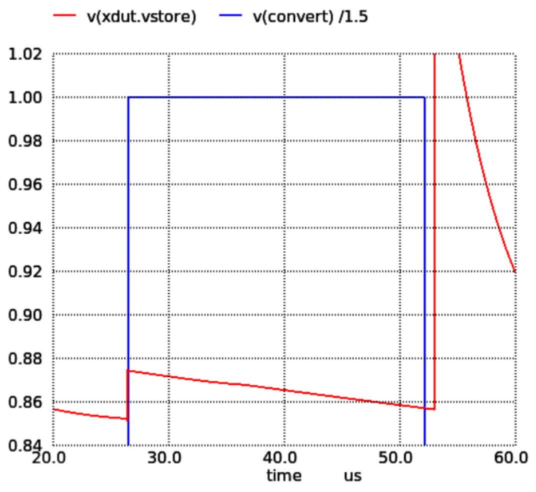


Fig. 11: Plot of storage capacitor voltage during the convert phase. Expose signal voltage is normalized to 1V.

3) Comparator delay: The comparator delay is measured from the time the *sensor in* voltage and the *ramp* voltage from figure 2 are equal, until the *Comp in* input on the memory from figure 3 has become low. *Comp in* is defined as low when it is 10% of the logic voltage $0.1 \cdot 1.5V = 0.15V$. A plot of *ramp* crossing *sensor in* is shown in figure 12 and *Comp in* reacting in figure 13. This shows that the comparator delay is $37.16\mu s - 36.96\mu s = 0.2\mu s$.

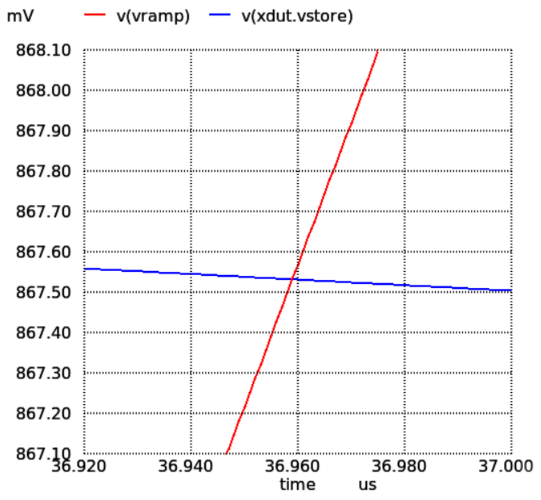


Fig. 12: Plot of *ramp* (vramp) crossing *sensor in* (vstore).

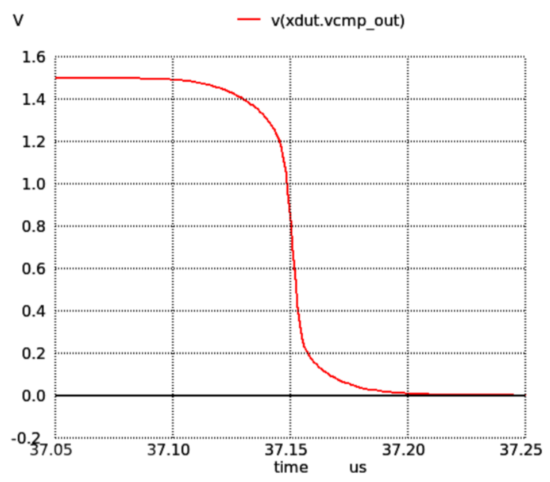


Fig. 13: Plot of *Comp in* (vcmp_out) going low after *ramp* crosses *sensor in*.

4) Digital value gets stored in memory: The memory needs to store the current data when the signal from the comparator goes low. Figure 14 demonstrates that the value in the memory is kept when the comparator triggers.

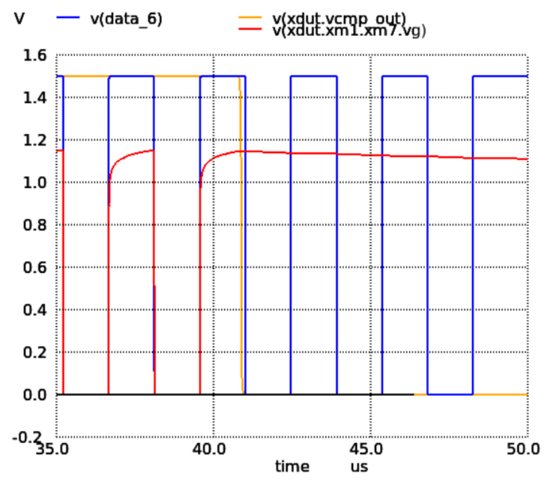


Fig. 14: Plot of a cell of memory when the comparator signal *vcomp_out* goes low. *vg* is data stored in the memory and *data_6* is the data in- and output.

When the read signal is set high, the data in the memory cells should be output. The memory is inverting, so if a high value is stored, the output will be low and vice versa. This process creates a short circuit when the stored value is high. Figure 15 shows that a high stored bit leads to a low output signal when *read* is high, and Figure 16 shows that a low stored bit leads to a high output signal when *read* is high.

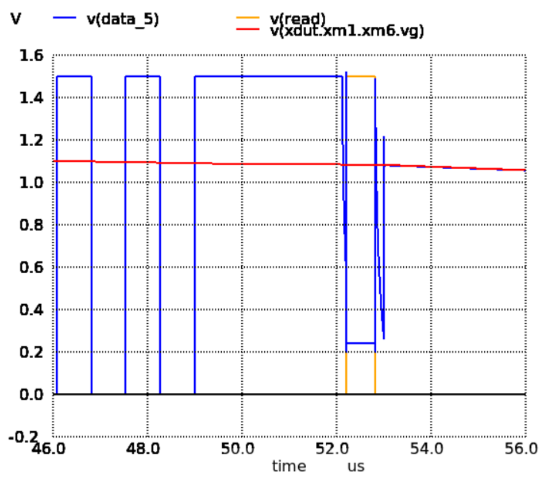


Fig. 15: Plot of a cell of memory when the read signal *read* goes high. *vg* is data stored in the memory and *data_5* is the data in- and output.

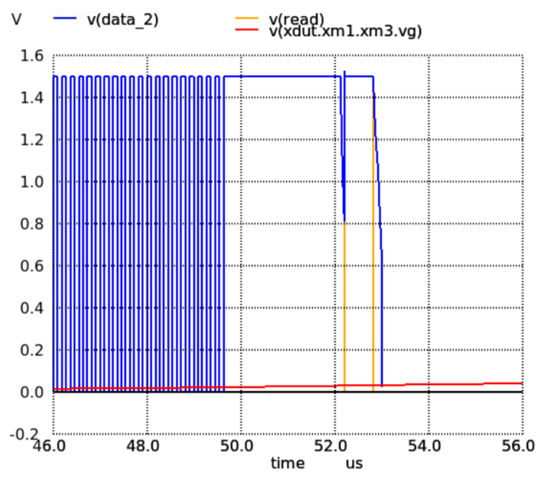


Fig. 16: Plot of a cell of memory when the read signal *read* goes high. *vg* is data stored in the memory and *data_2* is the data in- and output.

The sensor needs to work in the whole range of the ADC. Figure 17 shows a sample with a low current source in the photodiode model and 18 with a high current. All the phases of the sample cycle works as they should.

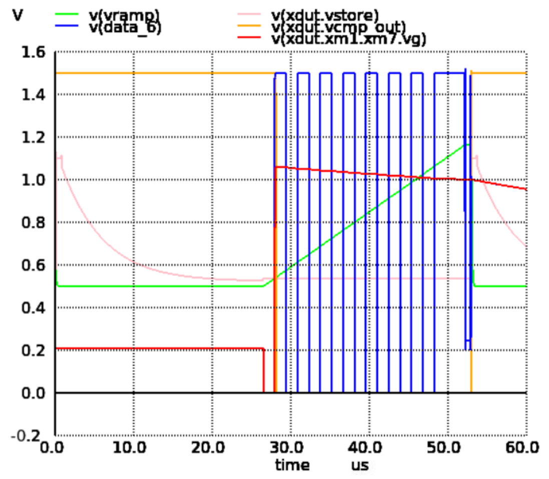


Fig. 17: One sample cycle with little light on the photodiode. *vg* is data stored in the memory, *data_6* is the data in- and output, *vstore* is the storage capacitor voltage, *vramp* is the ramp and *vcmp_out* is the output from the comparator.

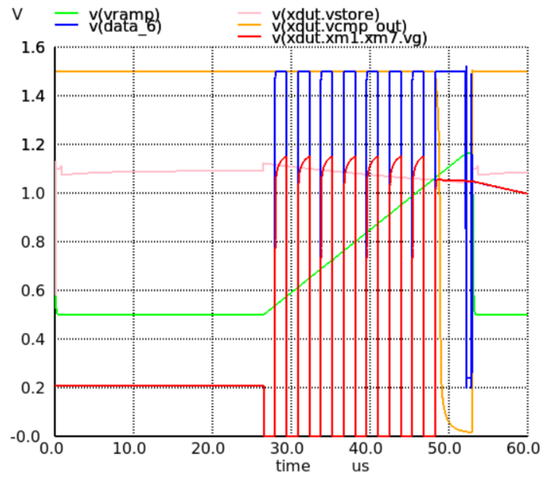


Fig. 18: One sample cycle with much light on the photodiode. *vg* is data stored in the memory, *data_6* is the data in- and output, *vstore* is the storage capacitor voltage, *vramp* is the ramp and *vcmp_out* is the output from the comparator.

B. Digital section

All modules that are a part of the digital section have its own test bench to determine that they work as intended. The test bench for Pixel Top tests the entire system.

Throughout the digital results we will show timing diagrams, some of these have been colour coded. Dark blue mean it is an input into the submodule, yellow means it is a clock, cyan means it is an internal signal only affecting the submodule, green means it is an output signal and purple signals denotes the current state of a state machine.

1) Pixel: The testbench for the pixel needs to show that it can do a sample cycle. As the pixel module is a simplified model of the analog pixel, it needs to behave as the pixel it represents. A state machine is used to cycle through the procedure of making a sample:

0 Erasing the pixel
1 Exposing/integrating the pixel
2 Converting the stored analog value into a digital value
3 Reading out the digital value

The testbench needs to verify that these phases are executed as they should on one individual pixel. Figure 19 shows that step 0, erasing, works. This and the following tests of the pixel are configured with a bit depth of 8 bits.

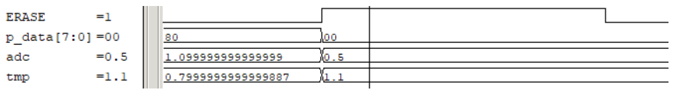


Fig. 19: Plot showing the effect on the *ERASE* signal going high. *p_data* is the pixel memory, *adc* is the analog ramp and *tmp* is the model of the storage capacitor voltage. All of them are successfully reset to their original value.

A successful step 1, exposing, is shown in figure 20.

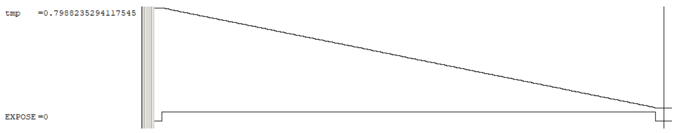


Fig. 20: The expose phase. *tmp* is the model of the storage capacitor voltage

Step 2, converting, is plotted in figure 21. The output from the comparator *cmp* goes low when *adc* and *tmp* and the data in the memory *p_data* at that moment is kept.

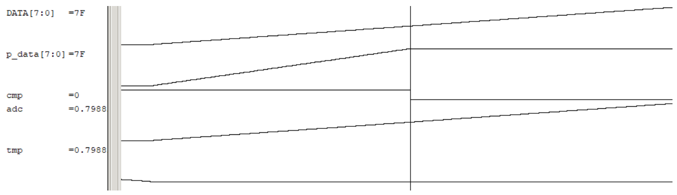


Fig. 21: *DATA* is the in- and output of the pixel, *p_data* is the pixel memory, *cmp* is the output from the comparator, *adc* is the analog ramp and *tmp* is the model of the storage capacitor voltage.

A step 3, readout, is shown in figure 22. The memory successfully puts its contents *p_data* inverted on the bus *DATA* when *READ* goes high.

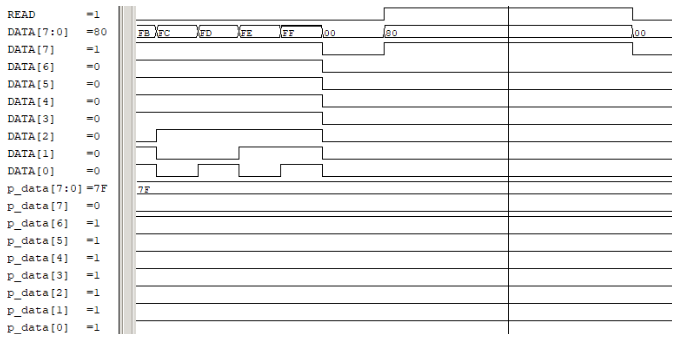


Fig. 22: *READ* is the read signal to the memory to put its content on the bus, *DATA* is the in- and output of the pixel and *p_data* is the pixel memory.

2) Counter: The counter module test bench runs a clock to make the counter run through its length and the reset signal is triggered to verify that it works properly. Figure 23 shows the reaction from the counter when a reset is triggered. The internal counter value returns to 0, and so does the output the following clock cycle.

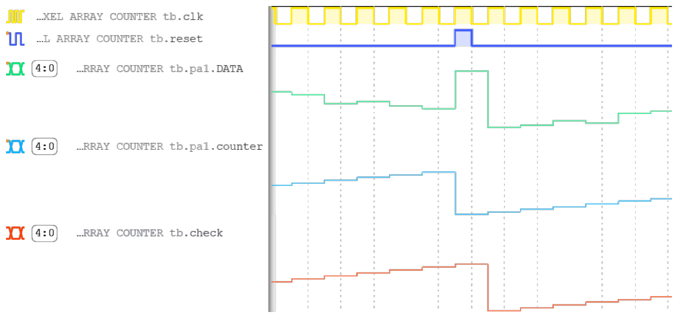


Fig. 23: Test results from the counter test bench. The figure is from a test with a bit depth of 5-bit to make the plots clearer. The plot ”DATA” is the gray-coded number outputed from the counter module, the plot ”counter” is the value of the counter before being gray-coded and the bottom plot ”check” is the gray-coded number transformed back into a regular number by the test bench.

A Gray-decoder is used in the test bench to verify that the Gray-coder in the counter works. The output of the counter module is one clock cycle behind the binary counter inside it. If the Gray-coder works, the decoded signal should be the same as the one from the counter, but one cycle behind. Figure 24 shows a part of a test-log where this is the case, which indicates the Gray counter works.

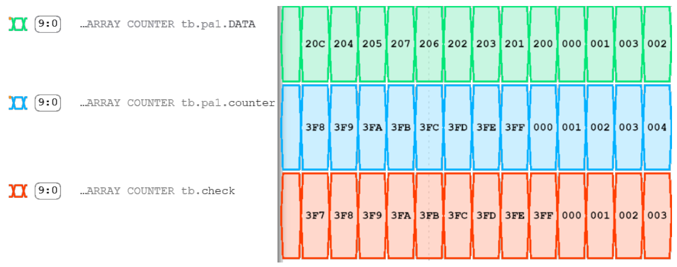


Fig. 24: Test results from the counter test bench. The top plot ”DATA” is the Gray-coded number outputed from the counter module, the middle plot ”counter” is the value of the counter before being Gray-coded and the bottom plot ”check” is the gray-coded number transformed back into a regular number by the test bench.

3) Memory Controller: For the purpose of having the results be presentable we will show a test of the memory controller where the parameters were quite different than in the full system. In this test the pixel height was set to 9, the width to 8 and the output bus was 40 bit, with other words 4 pixels, wide. The memory controller works as a state machine going through the following states:

0 Point to row
1 Read to buffer
2 Write out
3 Done

The test had the memory controller go through the states as if it controlled the write out of 8x9 pixel array through a 4 pixel wide output bus, but we will only show it process the last three rows. The results can be seen in figure 25.

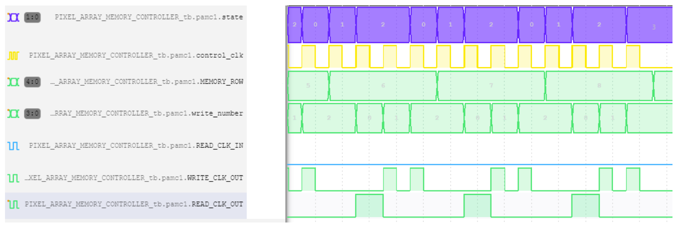


Fig. 25: Test of the memory controller. Showing readout of last three rows.

As can be seen in 25 the memory controller moves through all states. Using one cycle on the point to row, one cycle on read to buffer and two cycles on write out. It takes two cycles to read out as it can only read out 4 pixels per cycle and the chip had a width of 8 pixels. The memory controller sends out one pulse on read clk out and two pulses on write clk out. These pulses are used by the output bus to read into its’ buffer and then write the data out. For results from the full test see 31.

4) Row pointer: The row pointer takes in a control signal and uses bit shifting to send a read to one specific row. When it gets in 0 that means the first row should be read and when it gets in 5 that means the sixth row should be read. For presenting purposes the test shown will have the height set to 9 pixels. The results can be seen in figure 26.

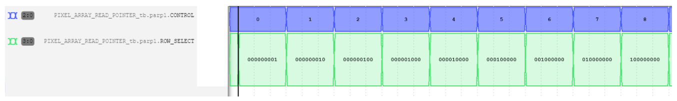


Fig. 26: Test of the row pointer

As can be seen, the row pointer sends out a row of 0’s with only a singular 1 in it. The position of the 1 is controlled by the control signal and as such the test is a success.

5) Output bus: The output bus takes in data from the pixels, inverts it back, converts it back from Gray code and writes it out. When it does this is dependent on the memory controller. One write out of a system with width 8, bit depth 10 and a bus width of 40bit can be seen in figure 27.

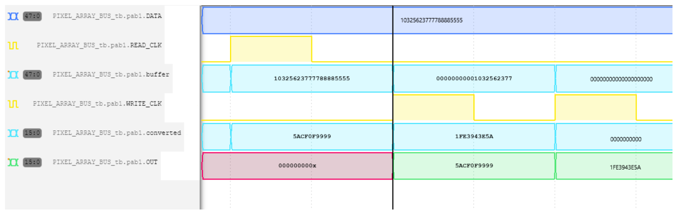


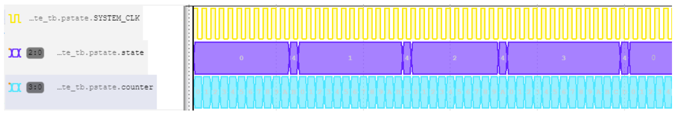
Fig. 27: Test of the output bus in a system with width 8, bit depth 10 and bus width 40bit.

As can be seen from the results the output bus only takes in signals and read them out when it gets the appropriate signals from the memory controller, namely read clk and write clk. It also converts the gray code correctly back into common binary values.

6) State machine: The state machine controlls the pixel array by sending signals to tell it what to do. The state machine iterates through 5 different states for this purpose:

0) Erase
1) Expose
2) Convert
3) Read
4) Idle

The states last vastly different duration's and as such it is quite difficult to present in a nice manner. Therefore the test we will show has altered duration's for each state except Idle. We set the duration of erase to 10 cycles, expose to 11 cycles, convert to 9 cycles and read to 12 cycles. Idle was kept at 1 cycle. The results can be seen in figure 28.


...te_tb.pstate.SYSTEM_CLK
2:0 ...te_tb.pstate.state
3:0 ...te_tb.pstate.counter
0
4
1
4
2
4
3
4
0
Fig. 28: Test of the state machine with the duration of each state altered.

In 28 it is also shown the counter that keeps track of how long the state machine has been in each state. As we can see the state machine switches states after the correct amount of cycles.

7) Pixel top: As all subsystems have been tested and verified we will here only concern ourselves with the most important factors. That means we will only look at the current state, the data line within the pixel array and the output. The results from a simulation at full resolution can be seen in figure 29.

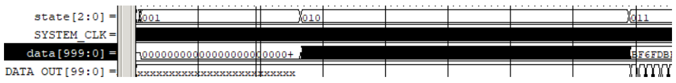
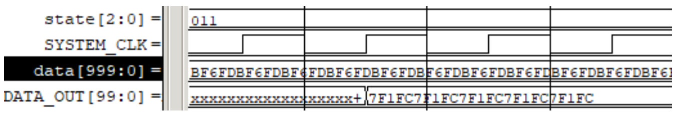


Fig. 29: Results from pixel top test.

As the results include thousands of clock cycles a lot details are hard to discern. The full VCD file from the simulation can be found in appendix C. For a more informative view we can zoom in on the write out. For this see figure 30.


state[2:0] =
SYSTEM_CLK =
data[999:0] =
DATA_OUT[99:0] =
011
BF6FDBF6FDBF6FDBF6FDBF6FDBF6FDBF6FDBF6FDBF6FDBF6FDBF6FDBF6
xxxxxxxxxxxxxxxxxxxxxx+
7F1FC7F1FC7F1FC7F1FC7F1FC
Fig. 30: Results from pixel top test, zoomed in on write out.

Here we can see that the data is properly converted and written out. As all pixels are exposed for the same amount of time and modelled with the exact same parameters they all give out the same value.

V. Discussion

As a whole our results are promising. A $0.2\mu\mathrm{s}$ delay through the comparator means that we can ensure that all values are correct plus minus one quantization with a frequency as high as 5MHz. Furthermore the gray code counter ensures only minimal issues can be caused by timing errors elsewhere on the chip. The 10 bit pixel depth and resolution ensures a good quality picture and at potentially high frame rates, if the frequency can be delivered upon. Two main issues with our chip is the high transistor count, which can limit the pixel density, and high current consumption. The comparator, the main continuous current drain, uses $6\mu\mathrm{A}$. Meaning our 100x100p chip uses about $60\mathrm{mA}$ even when in standby. And the peak current of $11\mathrm{A}$ would fry the chip in its current state. Lastly we have not synthesised and routed anything, meaning we have little to no knowledge on how parasitic effects would alter our results.

VI. Future work

- Tightening the current budget.
- Limiting the current when reading out of the on-pixel memory.
- Reduce transistor count and transistor size.
- Reducing comparator delay.
- Reducing the charge loss from the storage capacitor during the convert phase.
- Removal of waste cycles.
  There are a number of places where it could be possible to limit the amount of cycles used.
  - Removing the idle state from active use.
    Could be unnecessary to spend one cycle here for every state change.
  - Minimising exposure time to only what is necessary.
  - Combining the point to row and read to buffer actions into one state.
    If all systems were validated to have this not cause timing issues.
- Improving the DAC.
  Could be possible to make either a more accurate or smaller digital to analog converter, all depending on the purpose and requirements.
- Have the output work by its' own clock.
  Could allow for faster write speeds.
- Synthesis and routing.

VII. Conclusion

We have designed and detailed a 100×100 pixel photosensor utilising a per pixel ADC. Timings have been kept tight and the clock frequency could theoretically go as high as 5MHz with only minor issues. The standby current draw of the chip was about $60\mathrm{mA}$, but with a peak current consumption of about $11\mathrm{A}$. The transistor count was also quite high, incuring costs and limiting pixel density. The design was split into two parts, one analog and one digital. The digital system was detailed and simulated in systemVerilog, the analog system in SPICE. The analog system went into detail on the individual pixel while the digital system described the composition of the pixels and the logic used to get data out of the pixels in a systematic manner. The digital system was made to be modular with respect to resolution, bit depth and output bus width.

VIII. Appendix

Appendix A
Github repository

https://github.com/DanielVorhaug/TFE4152-Project–Digital-Pixel-Sensor

Appendix B
Full memory controller test

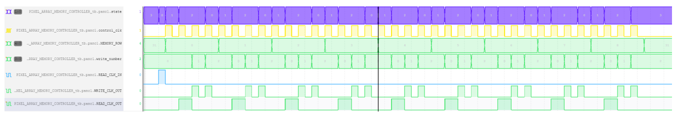


Fig. 31: Test of the memory controller, width of 8, height of 9, bit depth of 10 and output bus size of 40 bit.

Appendix C
VCD file from full resolution pixel top test

Note: In this file the frequency has been arbitrarily set to 20GHz. As the simulation only deals in logic, with no rise or fall times, the frequency could have been set to anything and it would not matter. The file can be downloaded here.

Appendix D
SPICE: pixelSensor.cir

```

* Pixel sensor
**
**        Copyright (c) 2021 Carsten Wulff Software, Norway
**
** Created       : wulff at 2021-7-22
**
**  The MIT License (MIT)
**
**  Permission is hereby granted, free of charge, to any person obtaining a copy
**  of this software and associated documentation files (the "Software"), to deal
**  in the Software without restriction, including without limitation the rights
**  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
**  copies of the Software, and to permit persons to whom the Software is
**  furnished to do so, subject to the following conditions:
**
**  The above copyright notice and this permission notice shall be included in all
**  copies or substantial portions of the Software.
**
**  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
**  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
**  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
**  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
**  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
**  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
**  SOFTWARE.
**
**

.SUBCKT PIXEL_SENSOR VBN1 VRAMP VRESET ERASE EXPOSE READ
+ DATA_9 DATA_8 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 COMP

XM1 READ VCMP_OUT DATA_9 DATA_8 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_9 DATA_8 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
XM9 VCMP_OUT DATA_8 READ VSS MEMCELL
XM10 VCMP_OUT DATA_9 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
MnReset VRESET ERASE VSTORE VSS nmos w=0.5u l=0.15u
XI1 ERASE ERASE_N VDD VSS INVERTER
MpReset VRESET ERASE_N VSTORE VDD pmos w=0.5u l=0.15u m=4
*Might want to drop m = 4 on the pmos

* Switch to expose pixel
MnExpose VPG EXPOSE VSTORE VSS nmos w=0.5u l=0.15u
XI2 EXPOSE EXPOSE_N VDD VSS INVERTER
MpExpose VPG EXPOSE_N VSTORE VDD pmos w=0.5u l=0.15u m=4

* Model photocurrent
* Set current source between 10n and 22n
* to test different light levels
Iphoto VSS VPG dc 17n
Rphoto VPG VSS 50Meg
.ENDS

.SUBCKT COMP VCMP_OUT VSTORE VRAMP VDD VSS


mp1 VP VP VDD VDD pmos w=0.5u l=0.5u
mp2 VO VP VDD VDD pmos w=0.5u l=0.5u

mn1 VP VSTORE VS VS nmos w=0.5u l=0.15u m=2
mn2 VO VRAMP VS VS nmos w=0.5u l=0.15u m=2

I3 0 VBN1 dc 6u
*This is just making a current mirror out of the lowest transistor
mb1 VBN1 VBN1 VSS VSS nmos w=0.5u l=0.5u
mb2 VS VBN1 VSS VSS nmos w=0.5u l=0.5u

*INVERTER
mp3 VO2 VO VDD VDD pmos w=0.5u l=0.15u m=4
mn3 VO2 VO VSS VSS nmos w=0.5u l=0.15u

*INVERTER
```

```
mp4 VCMP_OUT VO2 VDD VDD pmos w=0.5u l=0.15u m=4
mn4 VCMP_OUT VO2 VSS VSS nmos w=0.5u l=0.15u



.ENDS

.SUBCKT INVERTER VIN VOUT VDD VSS
M1 VOUT VIN VDD VDD pmos w=0.5u l=0.15u m=4
M2 VOUT VIN VSS VSS nmos w=0.5u l=0.15u
.ENDS
```

APPENDIX E
SYSTEMVERILOG: PIXELTOP.V

```

module PIXEL_TOP (
    input  logic SYSTEM_CLK,
    input  logic SYSTEM_RESET,
    output logic DATA_OUT_CLK,
    output logic [OUTPUT_BUS_PIXEL_WIDTH*BIT_DEPTH-1:0] DATA_OUT
);
    parameter WIDTH = 100;
    parameter HEIGHT = 100;
    parameter OUTPUT_BUS_PIXEL_WIDTH = 10;
    parameter BIT_DEPTH = 10;

    logic power_enable;
    logic write_enable;
    logic counter_reset;
    logic counter_clock;
    real  analog_ramp;
    logic reset;
    logic erase;
    logic expose;
    logic read_reset;
    logic read_clk_in;
    logic vbn1;

    PIXEL_STATE_MACHINE #(
            .WIDTH(WIDTH),
            .HEIGHT(HEIGHT),
            .OUTPUT_BUS_PIXEL_WIDTH(OUTPUT_BUS_PIXEL_WIDTH),
            .BIT_DEPTH(BIT_DEPTH))
        pstate(
            .SYSTEM_CLK (SYSTEM_CLK),
            .SYSTEM_RESET (SYSTEM_RESET),
            .POWER_ENABLE (power_enable),
            .WRITE_ENABLE (write_enable),
            .COUNTER_RESET (counter_reset),
            .COUNTER_CLOCK (counter_clock),
            .ANALOG_RAMP (analog_ramp),
            .RESET (reset),                      // Reset voltage in paper
            .ERASE (erase),                      // Pixel reset in paper
            .EXPOSE (expose),                    // PG in paper
            .READ_RESET (read_reset),
            .READ_CLK_IN (read_clk_in),
            .VBN1 (vbn1)                         // ana_bias1
    );

    PIXEL_ARRAY #(
            .WIDTH(WIDTH),
            .HEIGHT(HEIGHT),
            .OUTPUT_BUS_PIXEL_WIDTH(OUTPUT_BUS_PIXEL_WIDTH),
            .BIT_DEPTH(BIT_DEPTH))
        pa1(
            .POWER_ENABLE (power_enable),
            .WRITE_ENABLE (write_enable),
            .COUNTER_RESET (counter_reset),
            .COUNTER_CLOCK (counter_clock),
            .ANALOG_RAMP (counter_clock),
            .RESET (reset),
            .ERASE (erase),
            .EXPOSE (expose),
            .SYSTEM_CLK (SYSTEM_CLK),
            .READ_RESET (read_reset),
            .READ_CLK_IN (read_clk_in),
            .VBN1 (vbn1),
            .DATA_OUT_CLK (DATA_OUT_CLK),
            .DATA_OUT (DATA_OUT)
    );

endmodule
```

APPENDIX F
SYSTEMVERILOG: PIXELSTATE.V

```

//
//        Copyright (c) 2021 Carsten Wulff Software, Norway
//
// Created       : wulff at 2021-7-21
//
//  The MIT License (MIT)
//
//  Permission is hereby granted, free of charge, to any person obtaining a copy
//  of this software and associated documentation files (the "Software"), to deal
//  in the Software without restriction, including without limitation the rights
//  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//  copies of the Software, and to permit persons to whom the Software is
//  furnished to do so, subject to the following conditions:
//
//  The above copyright notice and this permission notice shall be included in all
//  copies or substantial portions of the Software.
//
//  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
//  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
//  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//  SOFTWARE.
//
//

module PIXEL_STATE_MACHINE (
    input  logic SYSTEM_CLK,
    input  logic SYSTEM_RESET,
    output logic POWER_ENABLE = 0,
    output logic WRITE_ENABLE = 0,
    output logic COUNTER_RESET = 0,
    output logic COUNTER_CLOCK,
    output real  ANALOG_RAMP,
    output logic RESET = 0,             // Reset voltage in paper
    output logic ERASE = 0,             // Pixel reset in paper
    output logic EXPOSE = 0,            // PG in paper
    output logic READ_RESET = 0,
    output logic READ_CLK_IN = 0,
    output logic VBN1                   // ana_bias1
);

    parameter WIDTH = 2;
    parameter HEIGHT = 2;
    parameter OUTPUT_BUS_PIXEL_WIDTH = 2;
    parameter BIT_DEPTH = 10;

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


    assign COUNTER_CLOCK    = ANALOG_RAMP;
    assign ANALOG_RAMP      = convert ? SYSTEM_CLK : 0;
    assign VBN1             = EXPOSE  ? SYSTEM_CLK : 0; // Amount of posedges decides how low photodiode voltage is

    //
    // State Machine
    //
    parameter STATE_ERASE=0, STATE_EXPOSE=1, STATE_CONVERT=2, STATE_READ=3, STATE_IDLE=4;

    //State duration in clock cycles
    parameter integer c_erase   = 5;
    parameter integer c_expose  = 2**BIT_DEPTH - 1;
    parameter integer c_convert = 2**BIT_DEPTH - 1;
    parameter integer c_read    = (2+WIDTH/OUTPUT_BUS_PIXEL_WIDTH)*HEIGHT + 1;

    parameter integer counter_max = (c_erase > c_expose ? c_erase : c_expose) > (c_convert > c_read ? c_convert : c_read) ? (c_erase > c_expose ? c_erase : c_expose) : (c_convert > c_read ? c_convert : c_read); // Finds which state duration is longest

    logic [$clog2(counter_max)-1:0] counter;                    // Delay counter in state machine, Assumes the longest state will be read
    logic                           convert = 0;
    logic                           convert_stop;
    logic [2:0]                     state, next_state;          // States


    // Control the output signals
    always_ff @(negedge SYSTEM_CLK ) begin
        case(state)
            STATE_ERASE: begin
                POWER_ENABLE    <= 0;
                WRITE_ENABLE    <= 0;
                COUNTER_RESET   <= 1;
                RESET           <= 0;
                ERASE           <= 1;
                EXPOSE          <= 0;
                convert         <= 0;
                READ_RESET      <= 0;
                READ_CLK_IN     <= 0;
            end

            STATE_EXPOSE: begin
                POWER_ENABLE    <= 1;
                WRITE_ENABLE    <= 1;
                COUNTER_RESET   <= 0;
                RESET           <= 0;
                ERASE           <= 0;
                EXPOSE          <= 1;
                convert         <= 0;
                READ_RESET      <= 0;
                READ_CLK_IN     <= 0;
            end

            STATE_CONVERT: begin
                POWER_ENABLE    <= 1;
                WRITE_ENABLE    <= 1;
                COUNTER_RESET   <= 0;
                RESET           <= 0;
                ERASE           <= 0;
                EXPOSE          <= 0;
                convert         <= 1;
                READ_RESET      <= 0;
                READ_CLK_IN     <= 0;
            end

            STATE_READ: begin
                POWER_ENABLE    <= 1;
                WRITE_ENABLE    <= 0;
                COUNTER_RESET   <= 0;
                RESET           <= 0;
                ERASE           <= 0;
                EXPOSE          <= 0;
                convert         <= 0;
                READ_RESET      <= 0;
                READ_CLK_IN     <= 1;
            end

            STATE_IDLE: begin
                POWER_ENABLE    <= 0;
                WRITE_ENABLE    <= 0;
                COUNTER_RESET   <= 0;
                RESET           <= 0;
                ERASE           <= 0;
                EXPOSE          <= 0;
                convert         <= 0;
                READ_RESET      <= 0;
                READ_CLK_IN     <= 0;
            end

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


    // Control the state transitions
    always_ff @(posedge SYSTEM_CLK or posedge SYSTEM_RESET) begin
        if(SYSTEM_RESET) begin
            state       = STATE_IDLE;
            next_state  = STATE_ERASE;
            counter     = 0;
            convert     = 0;
        end
        else begin
            case (state)
                STATE_ERASE: begin
                    if(counter == c_erase) begin
                        next_state  <= STATE_EXPOSE;
                        state       <= STATE_IDLE;
                    end
                end

                STATE_EXPOSE: begin
                    if(counter == c_expose) begin
                        next_state  <= STATE_CONVERT;
                        state       <= STATE_IDLE;
                    end
                end

                STATE_CONVERT: begin
                    if(counter == c_convert) begin
                        next_state  <= STATE_READ;
                        state       <= STATE_IDLE;
                    end
                end

                STATE_READ: begin
                    if(counter == c_read) begin
                        state       <= STATE_IDLE;
                        next_state  <= STATE_ERASE;
                    end
                end

                STATE_IDLE:
                    state <= next_state;

            endcase // case (state)

            if(state == STATE_IDLE)
                counter = 0;
            else
                counter = counter + 1;

        end
    end // always @ (posedge clk or posedge reset)

endmodule
```

APPENDIX G
SYSTEMVERILOG: PIXELARRAY.V

```

module PIXEL_ARRAY(
    input logic POWER_ENABLE,
    input logic WRITE_ENABLE,
    input logic COUNTER_RESET,
    input logic COUNTER_CLOCK,
    input real  ANALOG_RAMP,
    input logic RESET,       // Reset voltage in paper
    input logic ERASE,       // Pixel reset in paper
    input logic EXPOSE,      // PG in paper
    input logic SYSTEM_CLK,
    input logic READ_RESET,
    input logic READ_CLK_IN,
    input logic VBN1,

    output logic DATA_OUT_CLK,
    output logic [OUTPUT_BUS_PIXEL_WIDTH*BIT_DEPTH-1:0] DATA_OUT

);
    parameter WIDTH = 2;
    parameter HEIGHT = 2;
    parameter OUTPUT_BUS_PIXEL_WIDTH = 2;
    parameter BIT_DEPTH = 8;
    parameter dv_pixel = 0.5;

    wire [BIT_DEPTH*WIDTH - 1:0] data;
    logic [BIT_DEPTH - 1:0] counter;
    logic read_clk_out;
    logic [$clog2(HEIGHT):0] memory_row;
    logic memory_read_enable;
    logic [HEIGHT-1: 0] row_select;

    PIXEL_ARRAY_COUNTER #(
            .BIT_DEPTH(BIT_DEPTH))
        pac1(
            .COUNTER_RESET(COUNTER_RESET),
            .COUNTER_CLOCK(COUNTER_CLOCK),
            .DATA(counter)
    );

    genvar i;
    generate
       for (i = 0; i < WIDTH; i = i + 1) begin
           assign data[BIT_DEPTH*(i+1) - 1:BIT_DEPTH*i] = WRITE_ENABLE ? counter : ’Z;
       end
    endgenerate

    PIXEL_ARRAY_MEMORY_CONTROLLER #(
            .HEIGHT(HEIGHT),
            .WIDTH(WIDTH),
            .OUTPUT_BUS_PIXEL_WIDTH(OUTPUT_BUS_PIXEL_WIDTH))
        pamc1(
            .SYSTEM_CLK(SYSTEM_CLK),
            .READ_RESET(READ_RESET),
            .READ_CLK_IN(READ_CLK_IN),
            .READ_CLK_OUT(read_clk_out),
            .WRITE_CLK_OUT(DATA_OUT_CLK),
            .MEMORY_ROW(memory_row),
            .MEMORY_READ_ENABLE(memory_read_enable)
    );

    PIXEL_ARRAY_READ_POINTER #(
            .HEIGHT(HEIGHT))
        parp1 (
            .CONTROL(memory_row),
            .ENABLE(memory_read_enable),
            .ROW_SELECT(row_select)
    );

    PIXEL_ARRAY_BUS #(
            .BIT_DEPTH(BIT_DEPTH),
            .OUTPUT_BUS_PIXEL_WIDTH(OUTPUT_BUS_PIXEL_WIDTH),
            .WIDTH(WIDTH))
        pab1(
            .DATA(data),
            .READ_CLK(read_clk_out),
            .OUT(DATA_OUT),
            .WRITE_CLK(DATA_OUT_CLK)
    );


        // Pixels
    genvar column;
    genvar row;
    generate
        for(column = 0; column < WIDTH; column = column + 1)begin
            for(row = 0; row < HEIGHT; row = row + 1)begin
                PIXEL_SENSOR #(
                        .BIT_DEPTH(BIT_DEPTH),
                        .dv_pixel(dv_pixel))
                    ps(
                        .VBN1(VBN1),
                        .RAMP(ANALOG_RAMP),
                        .RESET(RESET),            // Reset voltage in paper
                        .ERASE(ERASE),            // Pixel reset in paper
                        .EXPOSE(EXPOSE),          // PG in paper
                        .READ(row_select[row]),   // Read in paper
                        .DATA(data[BIT_DEPTH*(column+1)-1:BIT_DEPTH*(column)])
                );
            end
        end
    endgenerate

endmodule
```

Appendix H
SystemVerilog: PixelSensor.v

```

//
//        Copyright (c) 2021 Carsten Wulff Software, Norway
//
// Created       : wulff at 2021-7-21
//
//  The MIT License (MIT)
//
//  Permission is hereby granted, free of charge, to any person obtaining a copy
//  of this software and associated documentation files (the "Software"), to deal
//  in the Software without restriction, including without limitation the rights
//  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
//  copies of the Software, and to permit persons to whom the Software is
//  furnished to do so, subject to the following conditions:
//
//  The above copyright notice and this permission notice shall be included in all
//  copies or substantial portions of the Software.
//
//  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
//  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
//  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
//  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
//  SOFTWARE.
//
//

//
// Model of pixel sensor, including
//  - Reset
//  - The sensor
//  - Comparator
//  - Memory latch
//  - Readout of latched value
//
module PIXEL_SENSOR
  (
   input logic      VBN1,        // Amount of posedges decides how low photodiode is
   input logic      RAMP,
   input logic      RESET,       // Reset voltage in paper, this variable is not used
   input logic      ERASE,       // Pixel reset in paper
   input logic      EXPOSE,      // PG in paper
   input logic      READ,        // Read in paper
   inout [BIT_DEPTH - 1:0] DATA

   );

   parameter integer BIT_DEPTH = 8;

   real             v_erase      = 1.1;
   real             v_min        = 0.5;
   real             lsb          = (v_erase-v_min)/(2**BIT_DEPTH);
```

```
    real              lsb_expose   = (v_erase-v_min)/(2**BIT_DEPTH);
    parameter real    dv_pixel     = 0.8;

    real              tmp;
    logic             cmp;
    real              adc;

    logic [BIT_DEPTH - 1:0]       p_data;

    //
    // ERASE
    //
    // Reset the pixel value on pixRst
    always @(ERASE) begin
       tmp = v_erase;
       p_data = 0;
       cmp  = 1;
       adc = v_min;
    end

    //
    // SENSOR
    //
    // Use bias to provide a clock for integration when exposing
    always @(posedge VBN1) begin
       if(EXPOSE)
         tmp = tmp - dv_pixel*lsb_expose;
    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 <= 0;
    end

    //
    // Memory latch
    //
    always_comb  begin
       if(cmp) begin
          p_data = DATA;
       end
    end

    //
    // Readout
    //
    // Assign data to bus when pixRead = 1
    // The readout of the memory inverts the data
    assign DATA = READ ? ˜p_data : ’Z;

endmodule // re_control
```

Appendix I SystemVerilog: PixelArrayCounter.v

```


module PIXEL_ARRAY_COUNTER (
    input   logic COUNTER_RESET,
    input   logic COUNTER_CLOCK,
    output  logic [BIT_DEPTH - 1:0] DATA
);

    parameter BIT_DEPTH = 10;

    logic [BIT_DEPTH - 1:0] counter = ’0;

    always_ff @(posedge COUNTER_CLOCK or posedge COUNTER_RESET) begin
        if (COUNTER_RESET) begin
            counter <= 0;
        end else begin
            counter <= counter + 1;
        end

            // Gray-coding the counter
        DATA[BIT_DEPTH-1]   <= counter[BIT_DEPTH-1];
        DATA[BIT_DEPTH-2:0] <= counter[BIT_DEPTH-1:1] ^ counter[BIT_DEPTH-2:0];
    end

endmodule
```

Appendix J SystemVerilog: PixelArrayMemoryController.v

```

module PIXEL_ARRAY_MEMORY_CONTROLLER (
    input   logic SYSTEM_CLK,
    input   logic READ_RESET,
    input   logic READ_CLK_IN,
    output  logic READ_CLK_OUT = 0,
    output  logic WRITE_CLK_OUT,
    output  logic [$clog2(HEIGHT):0] MEMORY_ROW = -1,
    output  logic MEMORY_READ_ENABLE = 0
);

    parameter integer WIDTH = 2;
    parameter integer HEIGHT = 2;
    parameter integer OUTPUT_BUS_PIXEL_WIDTH = 2; // Needs to go up in WIDTH


    parameter integer POINT_TO_ROW = 0, READ_TO_BUFFER = 1, WRITE_OUT = 2, DONE = 3;

    logic control_clk;
    logic [1:0] state       = DONE;
    logic state_enable      = 0;
    logic write_out_enable  = 0;

    logic [$clog2(WIDTH):0] write_number = 0;

    always_ff @(posedge READ_CLK_IN) begin
        state_enable        <= 1;
        state               <= 0;
        MEMORY_READ_ENABLE  <= 1;
        MEMORY_ROW          = -1;
    end

    assign control_clk   = state_enable     ? SYSTEM_CLK  : 0;
    assign WRITE_CLK_OUT = write_out_enable ? control_clk : 0;

    always_ff @(posedge control_clk) begin

        case (state)
            POINT_TO_ROW:
                begin
                    write_out_enable    <= 0;
                    MEMORY_ROW          <= MEMORY_ROW + 1;
                    state               <= READ_TO_BUFFER;
                end

            READ_TO_BUFFER:
                begin
                    READ_CLK_OUT        <= 1;
                    state               <= WRITE_OUT;
                    write_number        <= 0;
                end

            WRITE_OUT:
                begin
                    READ_CLK_OUT        <= 0;
                    write_out_enable    <= 1;
                    write_number        <= write_number + 1;

                    if (write_number == WIDTH/OUTPUT_BUS_PIXEL_WIDTH - 1) begin
                        state <= (MEMORY_ROW == HEIGHT - 1) ? DONE : POINT_TO_ROW;
                    end
                end

            DONE:
                begin
                    MEMORY_READ_ENABLE  <= 0;
                    MEMORY_ROW          <= -1;
                    state_enable        <= 0;
                end

            default:
                begin
                end
        endcase

    end
endmodule
```

APPENDIX K
SYSTEMVERILOG: PIXELARRAYREADPOINTER.V

```

module PIXEL_ARRAY_READ_POINTER (
    input   logic [$clog2(HEIGHT):0] CONTROL,
    input   logic ENABLE,
    output  logic [HEIGHT-1:0] ROW_SELECT
);

parameter integer HEIGHT = 2;

assign ROW_SELECT = ENABLE ? (1 << CONTROL) : '0;

endmodule
```

APPENDIX L
SYSTEMVERILOG: PIXELARRAYBUS.V

```

module PIXEL_ARRAY_BUS (
    input   logic [WIDTH*BIT_DEPTH-1:0] DATA,
    input   logic READ_CLK,
    input   logic WRITE_CLK,
    output  logic [OUTPUT_BUS_PIXEL_WIDTH*BIT_DEPTH-1:0]
        OUT
);

    parameter integer BIT_DEPTH = 8;
    parameter integer OUTPUT_BUS_PIXEL_WIDTH = 2;
    parameter integer WIDTH = 2;
    logic [WIDTH*BIT_DEPTH-1:0] buffer = '0;
    logic [OUTPUT_BUS_PIXEL_WIDTH*BIT_DEPTH-1:0] converted;


        // Reads data from one row into the buffer
    always_ff @(posedge READ_CLK) begin
        buffer <= ~DATA; // Data is inverted back after it
            has been inverted by the pixel-memory
    end



        // Shift register: Sequences data to be written out
            of the system
    genvar i;
    generate for(i = 0; i < WIDTH/OUTPUT_BUS_PIXEL_WIDTH-1;
          i++) begin
            always_ff @(posedge WRITE_CLK) begin
                buffer[BIT_DEPTH*OUTPUT_BUS_PIXEL_WIDTH*(i
                    +1)-1:BIT_DEPTH*OUTPUT_BUS_PIXEL_WIDTH*
                    i] <= buffer[BIT_DEPTH*
                    OUTPUT_BUS_PIXEL_WIDTH*(i+2)-1:
                    BIT_DEPTH*OUTPUT_BUS_PIXEL_WIDTH*(i+1)
                    ];
            end
        end
    endgenerate

    always_ff @(posedge WRITE_CLK) begin
        buffer[WIDTH*BIT_DEPTH-1:WIDTH*BIT_DEPTH-
            OUTPUT_BUS_PIXEL_WIDTH*BIT_DEPTH] <= '0;
    end



        // Gray decoder: Turns Gray coded data from the end
            of the shift register into regular binary
    genvar pixel;
    genvar index;
    generate

        for (pixel = 1; pixel < OUTPUT_BUS_PIXEL_WIDTH+1;
             pixel++) begin
            assign converted[pixel*BIT_DEPTH-1] = buffer[
                pixel*BIT_DEPTH-1];

            for (index = 2; index < BIT_DEPTH+1; index++)
                begin
                assign converted[pixel*BIT_DEPTH - index] =
                      converted[pixel*BIT_DEPTH - index+1] ^
                      buffer[pixel*BIT_DEPTH - index];
            end
        end

    endgenerate



        // Writes data to output
    always_ff @(posedge WRITE_CLK) begin
        OUT <= converted;
    end


endmodule
```

REFERENCES

[1] Y. Oike, K. Akiyama, L. D. Hung, W. Niitsuma, A. Kato, M. Sato, Y. Kato, W. Nakamura, H. Shiroshita, Y. Sakano, Y. Kitano, T. Nakamura, T. Toyama, H. Iwamoto, and T. Ezaki, "An 8.3m-pixel 480fps global-shutter cmos image sensor with gain-adaptive column adcs and 2-on-1 stacked device structure," in *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, 2016, pp. 1–2.

[2] S. Kleinfelder, S. Lim, X. Liu, and A. E. Gamal, "A 10 000 frames/s cmos digital pixel sensor," *JSSC*, vol. 36, no. 12, pp. 2049–2059, 2001.

[3] C. Liu, L. Bainbridge, A. Berkovich, S. Chen, W. Gao, T.-H. Tsai, K. Mori, R. Ikeno, M. Uno, T. Isozaki, Y.-L. Tsai, I. Takayanagi, and J. Nakamura, "A 4.6m, 512×512, ultra-low power stacked digital pixel sensor with triple quantization and 127db dynamic range," in *2020 IEEE International Electron Devices Meeting (IEDM)*, 2020, pp. 16.1.1–16.1.4.

[4] C. Wulff. (2021) Dicex (design of integrated circuits examples). [Online]. Available: https://github.com/wulffern/dicex