

## **Sammendrag**

Denne oppgaven er stilt av Aptina Norway AS og tar for seg utviklingen av et FPGA-basert system for emulering av output fra en A/D-omformer i en CMOS-bildesensor. Dette systemet er ment å benyttes til verifisering av RTL-design til CMOS-bildesensorprodukter. Emulatoren bruker en tilnærming til normalfordelingen for å emulere foton-, rad- og kolonnestøy, og kan kjøre på frekvenser opp til 124.81 MHz. Dette gjør at emulatoren kan behandle 60 bilder i sekundet med full HD-oppløsning. Systemet lar brukeren bestemme oppløsning, støytyper og eventuelt standardavvikene til rad- og kolonnestøyen ved oppstart. Hastigheten bestemmes av frekvensen på klokken som påtrykkes. Simuleringene og testene som er utført viser at emulatoren gir et resultat som er visuelt likt reell foton-, rad- og kolonnestøy, men støyfordelingene er noe kunstige og kan forårsake uventede artifakter i bildene.



# Innhold

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduksjon</b>                    | <b>1</b>  |
| <b>2</b> | <b>Teori</b>                           | <b>3</b>  |
| 2.1      | Bildebrikker . . . . .                 | 3         |
| 2.1.1    | Støy i bildebrikker . . . . .          | 4         |
| 2.1.2    | Tidsvarierende støy . . . . .          | 4         |
| 2.1.2.1  | Termisk støy . . . . .                 | 5         |
| 2.1.2.2  | Haglstry . . . . .                     | 5         |
| 2.1.2.3  | 1/f-støy . . . . .                     | 5         |
| 2.1.3    | CMOS vs. CCD . . . . .                 | 6         |
| 2.2      | CMOS-bildebrikker . . . . .            | 6         |
| 2.2.1    | Passiv og aktiv pikselsensor . . . . . | 7         |
| 2.2.2    | Digital pikselsensor . . . . .         | 8         |
| 2.3      | Støy i CMOS-bildebrikker . . . . .     | 8         |
| 2.3.1    | FPN . . . . .                          | 9         |
| 2.3.2    | Tidsvarierende støy . . . . .          | 9         |
| 2.3.2.1  | Pikselstøy . . . . .                   | 9         |
| 2.3.2.2  | Kolonneforsterkerstøy . . . . .        | 10        |
| 2.3.2.3  | PGA-støy . . . . .                     | 10        |
| 2.3.2.4  | A/D-omformerstøy . . . . .             | 11        |
| 2.4      | SNR og dynamisk område . . . . .       | 11        |
| 2.5      | PSNR og MAE . . . . .                  | 12        |
| <b>3</b> | <b>Metode</b>                          | <b>15</b> |
| 3.1      | Matlabsimuleringer . . . . .           | 15        |
| 3.2      | Utstyr og programvare . . . . .        | 16        |
| 3.3      | Emulatoren . . . . .                   | 16        |
| 3.3.1    | Systemoppsett . . . . .                | 17        |
| 3.3.2    | Tausworthe URNG . . . . .              | 18        |
| 3.3.3    | Fotonstøygenerator . . . . .           | 19        |
| 3.3.4    | Radstøygenerator . . . . .             | 21        |
| 3.3.5    | Kolonnestøygenerator . . . . .         | 22        |
| 3.4      | Testsystem . . . . .                   | 24        |

|          |  |           |
|----------|--|-----------|
| <b>4</b> | <b>Resultater</b>  | <b>25</b> |
| 4.1      | Matlabsimuleringer . . . . .   | 25        |
| 4.1.1    | Simulering med Matlabfunksjoner for å generere Poisson- og normalfordelte pseudo-tilfeldige tall . . . . . | 26        |
| 4.1.2    | Simulering av algoritmene som er brukt til å emulere foton-, rad- og kolonnestøy på FPGAen . . . . .       | 31        |
| 4.1.3    | Fordelingsfunksjonene til algoritmene for emulering av foton-, rad- og kolonnestøy . . . . .               | 35        |
| 4.2      | Resultater fra testsystem . . . . .  | 37        |
| 4.3      | Quartus II-simulering . . . . .  | 41        |
| 4.4      | Kompileringsresultater fra Quartus II . . . . .  | 43        |
| <b>5</b> | <b>Diskusjon</b>   | <b>45</b> |
| <b>6</b> | <b>Konklusjon</b>  | <b>47</b> |
| <b>A</b> | <b>Verilogkoden til emulatoren</b>   | <b>51</b> |
| <b>B</b> | <b>Emulatoren i testsystemet</b>   | <b>67</b> |
| <b>C</b> | <b>Nios II C-kode</b>  | <b>79</b> |
| <b>D</b> | <b>Matlabsimuleringer</b>  | <b>84</b> |
| D.1      | Simuleringer av støy med Matlabmetodene <i>poissrnd</i> og <i>randn</i> . . . . .                          | 84        |
| D.2      | Simuleringer av algoritmene emulatoren bruker . . . . .  | 87        |
| D.3      | Metoden som brukes til å regne ut PSNR, MAE og MSE . . . . .   | 89        |
| <b>E</b> | <b>C-sharp-program</b>   | <b>90</b> |
| E.1      | Program . . . . .  | 90        |
| E.2      | Klient . . . . .   | 90        |
| <b>F</b> | <b>Generering av mif-fil</b>   | <b>93</b> |

# Figurer

|      |  |    |
|------|--|----|
| 2.1  | Generering av fotostrøm i en reversforspent fotodiode . . . . .  | 4  |
| 2.2  | Blokkdiagram av en CMOS-bildesensor . . . . .  | 7  |
| 2.3  | Passiv pikselsensor . . . . .  | 7  |
| 2.4  | Aktiv pikselsensor . . . . .   | 8  |
| 2.5  | Digital pikselsensor . . . . .   | 8  |
| 2.6  | CMOS-bildesensor pikselmodell . . . . .  | 11 |
| 2.7  | SNR kontra signalamplitude . . . . .   | 12 |
| 3.1  | Cyclone III FPGA Development-kortets blokkdiagram . . . . .  | 15 |
| 3.2  | Overordnet blokkdiagram for systemet. . . . .  | 16 |
| 3.3  | Tilstandsdiagram til FSMen. . . . .  | 18 |
| 3.4  | Arkitekturen til en Tausworthe URNG. . . . .   | 19 |
| 3.5  | De 10000 første verdiene som genereres med inngangsverdiene 1 og 128. . . . .  | 20 |
| 3.6  | Arkitekturen til oppslagstabellen. . . . .   | 21 |
| 3.7  | Blokkdiagram av fotonstøygeneratoren. . . . .  | 22 |
| 3.8  | Blokkdiagram av multiplikatoren i radstøygeneratoren. . . . .  | 23 |
| 3.9  | Arkitekturen til RAM-modulen. . . . .  | 24 |
| 4.1  | Testbildet uten støy. . . . .  | 25 |
| 4.2  | Nærbilde av øverste høyre hjørnet av testbildet. . . . .   | 26 |
| 4.3  | Resultatet etter å ha brukt Matlab-funksjonen <i>poissrnd</i> til å generere Poisson-fordelt støy på testbildet. . . . . | 26 |
| 4.4  | Nærbilde av simulert fotonstøy. . . . .  | 27 |
| 4.5  | Simulert radstøy. . . . .  | 27 |
| 4.6  | Nærbilde av simulert radstøy. . . . .  | 28 |
| 4.7  | Simulert kolonnestøy. . . . .  | 28 |
| 4.8  | Nærbilde av simulert kolonnestøy. . . . .  | 29 |
| 4.9  | Simulert foton-, rad- og kolonnestøy. . . . .  | 29 |
| 4.10 | Nærbilde av simulert foton-, rad- og kolonnestøy. . . . .  | 30 |
| 4.11 | Resultatet av simulering av algoritmen som emulerer fotonstøy. . . . .   | 31 |
| 4.12 | Nærbilde av emulert fotonstøy. . . . .   | 31 |
| 4.13 | Resultatet av simulering av algoritmen som emulerer radstøy. . . . .   | 32 |
| 4.14 | Nærbilde av emulert radstøy. . . . .   | 32 |
| 4.15 | Resultatet av simulering av algoritmen som emulerer kolonnestøy. . . . .   | 33 |
| 4.16 | Nærbilde av emulert kolonnestøy. . . . .   | 33 |
| 4.17 | Simulering av algoritmene som benyttes til å emulere foton-, rad- og kolonnestøy. . . . .                                | 34 |

|      |   |    |
|------|---|----|
| 4.18 | Nærbilde av figur 4.17. . . . .   | 34 |
| 4.19 | De 10000 første verdiene generert med Matlabs <i>poissrnd</i> og algoritmen som brukes til å generere pseudo-tilfeldige tall i fotonstøygeneratoren med $\mu = 1$ . . . . .   | 35 |
| 4.20 | De 10000 første verdiene generert med Matlabs <i>poissrnd</i> og algoritmen som brukes til å generere pseudo-tilfeldige tall i fotonstøygeneratoren med $\mu = 128$ . . . . .   | 36 |
| 4.21 | De 10000 første verdiene generert med Matlabs <i>randn</i> og algoritmen som brukes til å generere pseudo-tilfeldige tall i rad- og kolonnestøygeneratorene med forventningsverdi 128 og standardavvik lik 7. . . . . | 36 |
| 4.22 | Resultatet fra test av emulering av fotonstøy på FPGAen. . . . .  | 37 |
| 4.23 | Nærbilde av figur 4.22. . . . .   | 37 |
| 4.24 | Resultatet fra test av emulering av radstøy på FPGAen. . . . .  | 38 |
| 4.25 | Nærbilde av figur 4.24. . . . .   | 38 |
| 4.26 | Resultat fra test av emulering av kolonnestøy på FPGAen. . . . .  | 39 |
| 4.27 | Nærbilde av figur 4.26. . . . .   | 39 |
| 4.28 | Resultat fra test av alle de tre implementerte støytypene emulert på FPGAen. . . . .  | 40 |
| 4.29 | Nærbilde av figur 4.28. . . . .   | 40 |
| 4.30 | Simulering av emulatoren rett etter oppstart. . . . .   | 41 |
| 4.31 | Simulering av emulatoren i det den er ferdig med første raden i et bilde. . . . .   | 42 |

# Tabeller

|     |  |    |
|-----|--|----|
| 2.1 | Hovedfordelene med CCD- og CMOS-bildebrikker . . . . .               | 6  |
| 3.1 | Seed- og konstantverdier benyttet i Tausworthe URNG. . . . .         | 18 |
| 3.2 | Oppslagstabell. . . . .  | 20 |
| 4.1 | Kvalitetsmål på simuleringsresultatene . . . . .                     | 30 |
| 4.2 | Kvalitetsmål på simuleringsresultatene av de emulerte støytypene . . | 35 |
| 4.3 | Kvalitetsmål på resultatene fra test på FPGA . . . . .               | 41 |
| 4.4 | Maksfrekvensen til emulatoren . . . . .                              | 43 |
| 4.5 | Oppsummering av kompileringsresultatene . . . . .                    | 43 |

# Forkortelser

|      |   |
|------|---|
| A/D  | Analog til Digital                      |
| APS  | Active Pixel Sensor                     |
| CCD  | Charged-Coupled Device                  |
| CDS  | Correlated Double Sampling              |
| CMOS | Complementary Metal-Oxide-Semiconductor |
| DIP  | Dual In-line Package                    |
| DPS  | Digital Pixel Sensor                    |
| DSC  | Digital Still Camera                    |
| DSLR | Digital Single-Lens Reflex camera       |
| FPGA | Field-Programmable Gate Array           |
| FPN  | Fixed-Pattern Noise                     |
| FSM  | Finite State Machine                    |
| LED  | Light-Emitting Diode                    |
| LFSR | Linear Feedback Shift Register          |
| LSB  | Least Significant Bit                   |
| MAE  | Mean Absolute Error                     |
| MSB  | Most Significant Bit                    |
| MSE  | Mean Squared Error                      |
| PGA  | Programmable Gain Amplifier             |
| PPS  | Passive Pixel Sensor                    |
| PSNR | Peak Signal-to-Noise Ratio              |
| RAM  | Random-Access Memory                    |



|      |                                 |
|------|---------------------------------|
| RMS  | Root Mean Square                |
| RNG  | Random Number Generator         |
| ROM  | Read-Only Memory                |
| RTL  | Register Transfer Level         |
| RTS  | Random Telegraph Signal         |
| SNR  | Signal-to-Noise Ratio           |
| URNG | Uniform Random Number Generator |



# Kapittel 1

## Introduksjon

Tidligere ble CMOS-bildebrikker kun benyttet i lavkostnadsapplikasjoner, fordi den optiske ytelsen ikke var god nok til at sensorene kunne benyttes i applikasjoner som setter strengere krav til kvaliteten. Dagens CMOS-bildebrikker har kommet til et nivå hvor den optiske ytelsen er sammenlignbar med den til CCD-bildebrikker. Dette gjør CMOS til et attraktivt alternativ, særlig på grunn av lavere produksjonskostnader, lavere effektforbruk og enklere integrasjon, sammenlignet med CCD-bildebrikker.

Aptina er en aktør som utvikler forskjellige typer CMOS-bildesensorer og bildeprosessorer, med ulike oppløsninger og hastigheter. De ønsket et system for å verifisere RTL-design til CMOS-bildesensorprodukter, hvor man kan endre hastigheten, oppløsningen og SNRen til sensoren. Med et slikt system vil man kunne teste RTL-design til forskjellige typer sensorer uten at man trenger å ha disse sensorene for hånden. Denne oppgaven går ut på å utvikle et FPGA-basert system som emulerer output fra en A/D-omformer i en CMOS-bildesensor, som kan benyttes til dette formålet. Systemet som skal utvikles må derfor ha programmerbar oppløsning, hastighet og SNR.



# Kapittel 2

## Teori

Dette kapittelet starter med å fortelle litt generelt om bildebrikker før det går mer inn på CMOS-bildebrikker. Deretter kommer det en seksjon som forteller om typisk støy i CMOS-bildebrikker før de viktige godhetstallene dynamisk område og SNR til bildebrikker presenteres. Kapittelet avsluttes med en kort forklaring av PSNR, MAE og MSE som forteller noe kvaliteten på et bilde i forhold til et referansebilde.

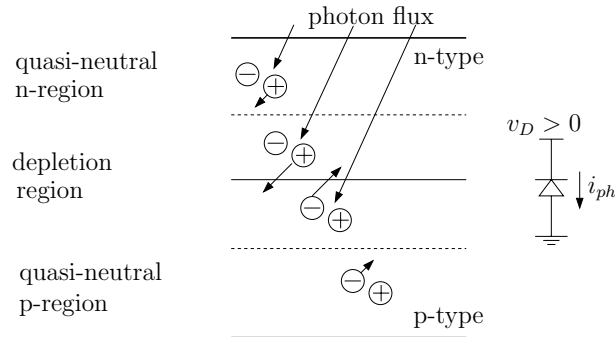
### 2.1 Bildebrikker

Alle digitalkameraer inneholder det vi kaller en bildebrikke (solid state image sensor). En bildebrikke består av  $n \times m$  piksler, fra  $320 \times 240$  til  $7000 \times 9000$  (benyttes til astronomiske observasjoner) [1]. Hver piksel inneholder en fotocelle og en utlesningskrets. Pikelstørrelsen varierer fra  $15\mu m \times 15\mu m$  ned til  $3\mu m \times 3\mu m$ , hvor minimum pikelstørrelse er begrenset av det dynamiske området og optikkostnader [1]. Hvor stor andel av arealet til pikselet som opptas av fotocellen kalles fyllingsgraden, og varierer fra 0,2 til 0,9. En høy fyllingsgrad er alltid ønskelig.

Fotocellen konverterer innkommende strålingseffekt (fotoner/s) til en fotostrøm som er proporsjonal med strålingseffekten [2]. Det finnes flere typer fotoceller, men de vanligste er fotodioden og fototransistoren. En fotodiode er satt sammen av en p-type halvleder og en n-type halvleder, og det er grenseområdet mellom disse, pn-overgangen, som er den aktive delen av dioden [3]. Normalt vil en diode kun lede strøm når den påtrykkes en spenning slik at p-siden blir positiv og n-siden negativ. Da vil frie elektroner fra n-siden bevege seg inn på p-siden, mens hull fra p-siden vil bevege seg inn på n-siden. Men når en fotodiode belyses i området rundt pn-overgangen vil det genereres elektronhullpar, og vi får frie elektroner i p-siden og frie hull i n-siden. Dioden vil dermed lede strøm i sperreretningen.

Figur 2.1 viser hvordan det genereres en fotostrøm i en reversforspent fotodiode [4]. Fotostrømmen,  $i_{ph}$ , er summen av tre komponenter:  $i_{ph}^{sc}$  - strøm som genereres i metningsområdet (space charge region),  $i_{ph}^p$  - strøm som følge av generering av hull i det kvasi-nøytrale n-type området, og  $i_{ph}^n$  - strøm som følge av generering av elektroner i p-type området [1]. Den totale genererte fotostrømmen er dermed:

$$i_{ph} = i_{ph}^{sc} + i_{ph}^p + i_{ph}^n \quad (2.1)$$



Figur 2.1: Generering av fotostrøm i en reversforspent fotodiode

Fotocellens mørkestrøm  $i_{dc}$  er en lekkasjestrøm. Den kalles mørkestrøm fordi den korresponderer med den fotostrømmen man har når fotocellen ikke er under belysning. Mørkestrøm kommer av defekter i silisiumet, og begrenser det dynamiske området til fotocellen, da den reduserer signalsvinget og introduserer haglstøy [1].

### 2.1.1 Støy i bildebrikker

Bildesensorer er i grunnen elektroniske, analoge kretser. Og i likhet med alle slike kretser er støy et problem [5]. Støy i en bildesensor kan defineres som en hver signalvariasjon som forverrer bildekvaliteten [6].

En bildesensor for stillbilder reproducerer todimensjonal bildeinformasjon. Støy som forekommer i faste romlige posisjoner i et reprodusert bilde, kalles gjerne fixed-pattern noise (FPN), mens støy som varierer med tiden gjerne kalles tilfeldig eller tidsvarierende støy. Det er ikke mulig å skille FPN fra tidsvarierende støy utifra ett stillbilde, men en sekvens med stillbilder vil kunne vise at den tidsvarierende støyen varierer fra bilde til bilde. Denne rapporten vil fokusere på tidsvarierende støy da emulatoren som er designet ikke emulerer noen form for FPN.

### 2.1.2 Tidsvarierende støy

Tidsvarierende støy er en tilfeldig variasjon i signalet som svinger over tid. Når signalet av interesse svinger rundt sin gjennomsnittsverdi, som man antar er konstant, er variansen definert som

$$\text{Varians} = \langle (N - \langle N \rangle)^2 \rangle = \langle N^2 \rangle - \langle N \rangle^2 \quad (2.2)$$

hvor  $\langle \rangle$  uttrykker det statistiske gjennomsnittet, som er et gjennomsnitt av en kvantitet over en tidsperiode,  $t$ , fra et sett av stikkprøver [6].

Variansen til et signal samsvarer med den totale støyeffekten til signalet. Når det eksisterer flere ukorrelerte støykilder, er den totale støyeffekten gitt av [6]

$$\langle n_{total}^2 \rangle = \langle \sum_{i=1}^N n_i^2 \rangle \quad (2.3)$$

Fra sentralgrenseteoremet vet man at sannsynlighetsfordelingen til en sum av uavhengige, tilfeldige variabler går mot en normalfordeling, Gausskurven, når antall

tilfeldige variabler går mot uendelig [7]. Normalfordelingen er gitt av

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left[ -\frac{(x-m)^2}{2\sigma^2} \right] \quad (2.4)$$

hvor  $m$  er middelveien eller gjennomsnittet og  $\sigma$  er standardavviket eller effektivverdien (RMS-verdien) til variabelen  $x$ . I dette tilfellet kan standardavviket,  $\sigma$ , brukes som et mål på tidsvarierende støy [6].

Det finnes tre typer fundamentale tidsvarierende støymekanismer i optiske og elektroniske systemer: termisk støy, haglstøy (shot noise) og  $1/f$ -støy (flicker noise).

### 2.1.2.1 Termisk støy

Termisk støy, også kjent som Johnson-Nyquist-støy [8], [9], skyldes vibrasjoner av ladningsbærere (som regel elektroner) i en elektronisk komponent. Dette fenomenet opptrer selv uten en påtrykt spenning. Effektspektraltettheten til termisk støy er gitt av

$$S_V(f) = 4kTR \text{ [V}^2/\text{Hz]} \quad (2.5)$$

hvor  $k$  er Boltzmanns konstant,  $T$  er absolutt temperatur og  $R$  er resistansen.

### 2.1.2.2 Haglstøy

Haglstøy genereres når det går en strøm gjennom et elektronrør eller en halvleder, og skyldes at antall ladningsbærere fluktuerer etter visse regler, statistisk, fra øyeblikk til øyeblikk [10]. I bildebrikker assosieres haglstøy med innkommende fotoner og mørkestrøm [6]. Statistiske studier av haglstøyen har vist at sannsynligheten for at  $N$  partikler, som fotoner eller elektroner, emitteres i et gitt tidsintervall, er gitt av Poissonfordelingen som kan representeres som

$$P_N = \frac{(\bar{N})^N \cdot e^{-\bar{N}}}{N!} \quad (2.6)$$

hvor  $N$  og  $\bar{N}$  er henholdsvis antall partikler og gjennomsnittet [6]. Poissonfordelingen har den interessante egenskapen at variansen er lik gjennomsnittsverdien, eller

$$n_{shot}^2 = \langle (N - \bar{N})^2 \rangle = \bar{N} \quad (2.7)$$

Effektspektraltettheten til termisk støy og haglstøy er konstant over alle frekvenser. Denne type støy kalles “hvit støy”, en analogi til hvitt lys som har en flat effektfordeling i det visuelle spektrum [6].

### 2.1.2.3 $1/f$ -støy

Effektspektraltettheten til  $1/f$ -støy er proporsjonal med  $1/f^\gamma$ , hvor  $\gamma$  er ca 1 [6]. Utgangsførsterkerne i bildebrikker har problemer med  $1/f$ -støy ved lave frekvenser, men støyen er for det meste undertrykket av “correlated double sampling” (CDS), så lenge CDS-operasjonen utføres slik at intervallet mellom to sampler er kort nok til at  $1/f$ -støyen kan sees på som en offset [6].

### 2.1.3 CMOS vs. CCD

CMOS-bildebrikker dukket opp så tidlig som i 1967, men etter oppfinnelsen av CCD i 1970 [5] dominerte denne teknologien markedet for bildegjengivende detektorer [11]. Årsaken til at CCD ble den herskende teknologien var et overlegent dynamisk område, lavere FPN, mindre pikselstørrelse og større lyssensitivitet [12].

På begynnelsen av 1990-tallet ble CMOS-bildebrikker igjen et alternativ til CCD på grunn av fordeler som lavt effektforbruk, lave kostnader, “on-chip”-funktjonalitet og stor frihet i valg av utlesningsarkitektur [11]. Første generasjons CMOS-bildebrikker benyttet passive pikselsensorer (se kapittel 2.2.1), men ved å sette inn en forsterker per kolonne eller per rad oppnår man en mye bedre SNR for fotodioden. Derfor begynte man å benytte sensorer som implementerer et buffer som oppfører seg som en enkel “source follower” per piksel. Denne typen sensor er blitt kjent som en aktiv piksel sensor, og representerer andre generasjon av CMOS-bildebrikker [11].

Mobiltelefoner, videokameraer og DSLR-kameraer har alle begynt å benytte CMOS-bildebrikker. Spesielt mobiltelefoner har blitt en viktig applikasjon. CMOS-bildebrikker med høy megapikseloppløsning har blitt utviklet slik at hvem som helst kan nyte å ta stillbilder med mobiltelefonen [13]. Til tross for at CMOS-bildebrikkene har erstattet CCD-bildebrikker i mange applikasjoner, er det fortsatt etterspørsel etter CCD-bildebrikker i høyoppløselige DSCs med høy bildekvalitet. Dette er fordi sensitiviteten til de små pikslene i CCDene er høyere enn de i CMOS-bildebrikkene. Men selv på dette området kan CMOS-bildebrikker snart erstatte CCDer, når CMOS-bildebrikkene nå er i ferd med å overgå menneskesynet, ved å realisere både høy oppløsning og høyhastighets ytelse [13].

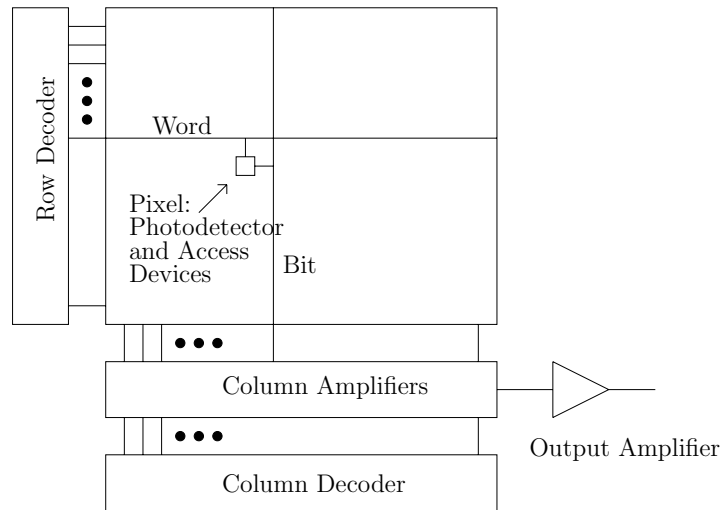
| CCD                    | CMOS                     |
|------------------------|--------------------------|
| Lavere støy            | Lavt effektforbruk       |
| Mindre pikselstørrelse | Singel strømforsyning    |
| Mindre mørkestrøm      | Høy integreringsmulighet |
| 100% fyllingsgrad      | Lavere kostnader         |
| Høy sensitivitet       | Singel masterklokke      |

Tabell 2.1: Hovedfordelene med CCD- og CMOS-bildebrikker

## 2.2 CMOS-bildebrikker

CMOS-bildebrikker framstilles ved å bruke standard CMOS-prosesser med små eller ingen modifikasjoner. Pikslene adresseres gjennom en horisontal ord-linje og ladingen eller spenningen leses ut fra hvert piksel gjennom en vertikal bit-linje. Utlesningen foregår ved å overføre en rad av gangen til kolonnellagringskondensatorene, for deretter å lese ut raden ved hjelp av en kolonnedekoder og en multiplekser. Figur 2.2 viser en CMOS-bildebrikkearkitektur. Det finnes tre pikselarkitekturer: Passiv piksel (PPS), aktiv piksel (APS) og digitalt piksel (DPS).

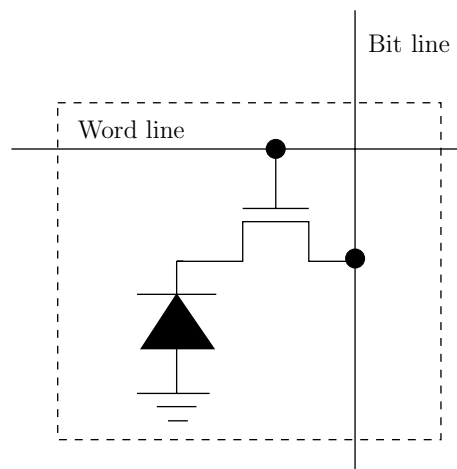




Figur 2.2: Blokkdiagram av en CMOS-bildesensor

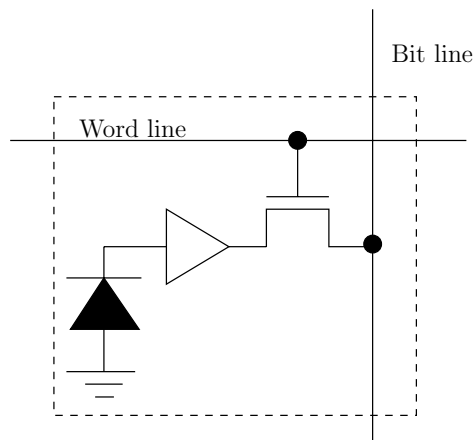
### 2.2.1 Passiv og aktiv pikselsensor

En passiv pikselsensor har bare en transistor per piksel som vist i figur 2.3. PPS har en liten pikselstørrelse og en høy fyllingsgrad, men ulempene er lav utlesningshastighet og lav SNR. Utlesningshastigheten til PPS er begrenset av tiden det tar å overføre en rad til utgangen av ladningsforsterkerne [1].



Figur 2.3: Passiv pikselsensor

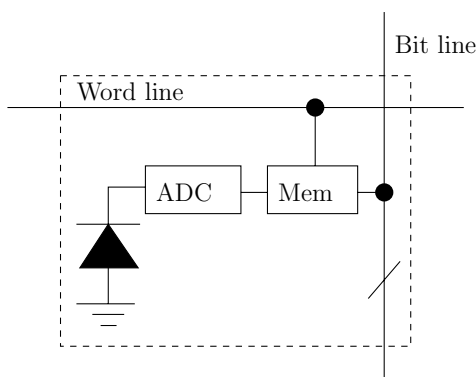
En aktiv pikselsensor har normalt tre eller fire transistorer per piksel, hvor en av transistorene fungerer som buffer og forsterker. Figur 2.4 viser hvordan utgangen til fotodioden bufres ved hjelp av en "pixel level follower"-forsterker. Sammenlignet med PPS har APS større pikselstørrelse og lavere fyllingsgrad, men utlesingen er kjappere og den har høyere SNR [1].



Figur 2.4: Aktiv pikselsensor

### 2.2.2 Digital pikselsensor

I en digital pikselsensor har hver piksel en egen analog til digital-omformer (A/D-omformer). A/D-omformerne jobber parallelt, og digitale data som lagres i minnet leses direkte ut av pikselsensorene som i et konvensjonelt digitalt minne (se figur 2.5).



Figur 2.5: Digital pikselsensor

En DSP-arkitektur har flere fordeler sammenlignet med analoge bildesensorer som APS og PPS. Blant disse fordelene er bedret skalering med CMOS-teknologien på grunn av reduserte krav til den analoge delen av kretsen, og eliminering av utlesningsrelatert kolonnestøy. Med en A/D-omformer og minne i hver piksel blir det praktisk med massiv parallell “snapshot”-avbildning, A/D-omforming og høyhastighetsutlesning, og de analoge flaskehalsene A/D-omforming og utlesning elimineres [1].

## 2.3 Støy i CMOS-bildebrikker

CMOS-bildebrikker påvirkes av forskjellige støykilder som setter fundamentale begrensninger på ytelsen, spesielt ved lav belysning og i videoapplikasjoner [14]. Poten-

sielle støykilder er til stede fra fotodioden i sensoren gjennom kolonneforsterkerne, PGAene (Programmable Gain Amplifier) og A/D-omformerne [15].

### 2.3.1 FPN

FPN har vært en stor begrensning ved CMOS-bildebrikker. FPN er den fikserte variasjonen i utgangssignalet når man påtrykker et uniformt inngangssignal [11]. I en perfekt bildesensor vil alle pikslene ha samme utgangsverdi ved samme belysning, men slik er det ikke i realiteten. FPN varierer ikke med tiden, men dersom man antar at en lineær pikselrespons, kan den karakteriseres som en variasjon i offset og forsterkning i hver piksel [11]. Eller sagt på en annen måte:

$$V_{ij}(t) = G_{ij}X_{ij}(t) + O_{ij} \quad (2.8)$$

Her er  $V_{ij}$  utgangssignalet,  $G_{ij}$  er forsterkningen,  $X_{ij}$  er inngangssignalet mens  $O_{ij}$  er offseten til pikselen i posisjon  $(i, j)$ .

FPN oppstår på grunn av mistilpasning i piksel og fargefiltre, variasjoner i kolonneforsterkere og mistilpasning mellom flere PGAer og A/D-omformere. FPN kan være enten koherent eller ikke-koherent [15].

### 2.3.2 Tidsvarierende støy

Tidsvarierende støy er en tidsavhengig variasjon i signalet og kan oppstå i pikselen, kolonneforsterkerne, PGAene og A/D-omformerne [15]. Eksempler på tidsvarierende støy er kTC-støy, Johnson-støy, 1/f-støy, RTS-støy (Random Telegraph Signal), mørkestrøm, fotonstøy, strømforsyningsstøy, fasestøy og kvantiseringsstøy [16].

#### 2.3.2.1 Pikselstøy

Fotonstøy, kTC-støy, mørkestrøm og MOS-støy er blandt støykildene i pikselen.

Fotonstøy oppstår som en følge av statistiske variasjoner i antall innfallende fotoner i eksponeringstiden [16]. Dette er en stokastisk prosess som kan beskrives ved hjelp av Poisson-statistikk. Dersom en piksel motter en mengde fotoner, lik  $\mu_{ph}$  i eksponeringsperioden, da er denne verdien gjennomsnittsverdien som også kan karakteriseres med en støykomponent  $\sigma_{ph}$ , som representerer fotonstøyen [16]. Relasjonen mellom gjennomsnittsverdien  $\mu_{ph}$  og dens assosierte støy  $\sigma_{ph}$  er gitt av

$$\sigma_{ph} = \sqrt{\mu_{ph}}. \quad (2.9)$$

Etter at de innfallende fotonene er absorbert inn i silisiumet, resulterer strømmen av  $\mu_{ph}$  fotoner i  $\mu_e$  elektroner i hver piksel, karakterisert av en støykomponent  $\sigma_e$ , som har den samme rot-relasjonen [16].

Signalet som integreres i en piksel måles relativt til dens reset-nivå. Usikkerheten knyttet til dette nivået på grunn av termisk støy kalles reset-støy eller kTC-støy [15]. Reset-arkitekturen kan kontrueres slik at denne typen støy stort sett blir borte, enten ved hjelp av CDS-teknikker eller ved å bruke enveis MOS-reset-svitsjer i pikselen [15].

Mørkestrøm assosiert med lekkasjestrøm i fotodioden,  $I(dark)$ , er veldig avhengig av eksponeringsperioden og er gitt av:

$$Noise = \frac{q}{C(pixel)} \times \sqrt{\frac{I(dark) \times \tau}{q}} [V] \quad (2.10)$$

hvor  $C(pixel)$  er den totale pikselkapasitansen og  $q$  er elektronladningen [15]. De beste CMOS-prosessene har som regel ekstremt lav mørkestrøm for å oppnå best mulig bildekvalitet [15].

Hovedkilden til forsterkerstøy i pikslene er termisk og 1/f-støy i MOS-transistorene [15]. 1/f-støy kan stort sett elimineres ved hjelp av hurtig dobbel-sampling av pikselen. Termisk støy kan også stort sett elimineres ved å begrense båndbredden til forsterkerne i pikslene.

### 2.3.2.2 Kolonneforsterkerstøy

Kolonneforsterkerne sampler reset og signalnivået til pikslene og bufrer eller forsterker deretter differansesignalet. De største støykildene i denne prosessen er kTC-støy assosiert med samplingsprosessen, og termisk og 1/f-støy i MOS-transistorene til forsterkeren [15].

De to ukorrelerte samplingsoperasjonene av resetnivået og signalet resulterer i et termisk støysignal som er gitt av

$$Noise = \sqrt{\frac{2kT}{C(Column)}} [V] \quad (2.11)$$

hvor  $C(Column)$  er kolonnesamplingskapasitansen,  $k$  er Boltzmanns konstant og  $T$  er absolutt temperatur [15].

Termisk og 1/f-støy er tilstede i kolonneforsterkernes MOS-transistorer, men effekten av disse er generelt neglisjerbar sammenlignet med kTC-støyen fra samplingsprosessen [15].

### 2.3.2.3 PGA-støy

Støykildene i PGAene er termisk kTC-støy i forbindelse med samplingsoperasjoner, og termisk og 1/f-støy i forbindelse med forsterkeroperasjonen. Som i kolonneforsterkerne, utføres det vanligvis to ukorrelerte samplingsoperasjoner i PGAene som resulterer i et termisk støysignal gitt av:

$$Noise = \sqrt{\frac{2kT}{C(pga)}} [V] \quad (2.12)$$

Termisk og 1/f-støy i MOS-transistorene i forsterkeren kan designes slik at de er mye mindre enn kTC-støyen fra samplingsoperasjonen [15].

### 2.3.2.4 A/D-omformerstøy

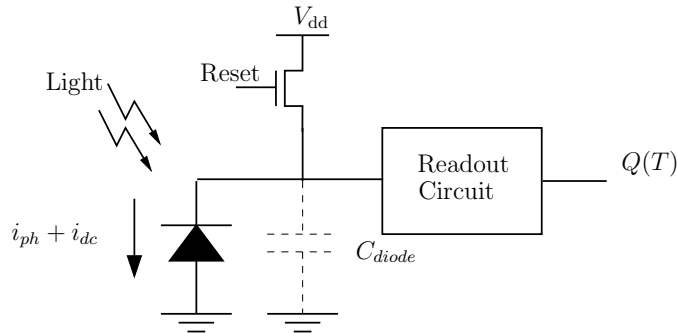
Hovedkilden til støy i A/D-omformerne er kvantiseringsstøy. En ideell A/D-omformers kvantiseringsstøy er gitt av:

$$Noise = \frac{LSB}{\sqrt{12}} = 0.288 \times LSB \quad (2.13)$$

En praktisk A/D-omformer har et støynivå som overgår 0.288 LSB på grunn av andre støykilder som termisk støy, forsterkerstøy og svitsjestøy. I tillegg vil tilfeldig mistilpasning i A/D-omformerkomponenter føre til FPN [15].

## 2.4 SNR og dynamisk område

Det brukes fortsatt mye ressurser på forskning og utvikling for å forbedre ytelsen til bildesensorer. Man forsøker å forbedre teknologien, optimalisere designet og utvikle nye sensorarkitekturer. Alt for å nå det samme målet, å forbedre signal-til-støy-forholdet (SNR) til en sensor for en gitt applikasjon eller oppløsning [17].



Figur 2.6: CMOS-bildesensor pikselmodell

Figur 2.6 viser en typisk piksel i en CMOS bildebrikke. Fotodioden resettes før begynnelsen av en eksponeringsperiode. Under eksponeringsperioden integreres den genererte fotostrømmen på den parasittiske kapasitansen  $C_{diode}$  og ladningen  $Q(T)$  (evt. spenningen) leses ut etter eksponeringstiden  $T$ . Mørkestrøm og additiv støy integreres også sammen med fotoladningen. Støyen kan uttrykkes som summen av tre uavhengige komponenter [1]:

- Haglstøy  $U(T)$ , genereres når det går strøm gjennom dioden, tilnærmet Gaussisk  $U(T) \sim \mathcal{N}(0, q \int_0^T (i_{ph}(t) + i_{dc}) dt)$  når fotostrømmen er stor nok. Her er  $q$  elektronladningen.
- Resetstøy (inkludert FPN), genereres ved resetting, har også en Gaussisk fordeling,  $C \sim \mathcal{N}(0, \sigma_C^2)$ .
- Utlekningsstøy  $V(T)$  (inkludert kvantiseringsstøy) med null i middelerdi og en varians på  $\sigma_V^2$ .

Utgangsladningen fra en piksel kan dermed skrives som

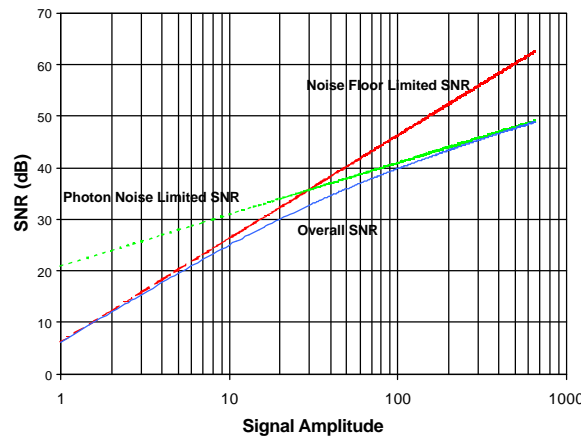
$$Q(T) = \int_0^T (i_{ph}(t) + i_{dc})dt + U(T) + V(T) + C, \quad (2.14)$$

gitt at  $Q(T) \leq Q_{sat}$ , brønnkapasiteten (saturation charge) [1].

Dersom fotostrømmen er konstant i eksponeringsperioden, er SNR gitt av [1]

$$\text{SNR}(i_{ph}) = 20 \log_{10} \frac{i_{ph}T}{\sqrt{q(i_{ph} + i_{dc})T + \sigma_V^2 + \sigma_C^2}}. \quad (2.15)$$

Likning (2.15) viser at SNRen øker med  $i_{ph}$ , først med 20 dB per dekad når resettingsstøy- og utlesningsstøyvariansen dominerer, deretter med 10 dB per dekad når haglstøyvariansen dominerer. Siden SNR også øker med  $T$  er det alltid ønskelig med så lang eksponeringstid som mulig, men brønnkapasiteten og endring i fotostrømmen som følge av bevegelse setter en øvre grense for eksponeringstiden. Figur 2.7 viser hvordan SNR endres med signalamplituden [15].



Figur 2.7: SNR kontra signalamplitude

Et annet viktig godhetstall for bildebrikker er det dynamiske området. Det dynamiske området sier noe om bildebrikkens evne til å vise både lyse lys og mørke skygger i en scene. Det er definert som ratioen av den høyeste ikke-mettede fotostrømmen  $i_{max}$  til den minste påviselige fotostrømmen  $i_{min}$ , typisk definert som standardavviket til støyen ved lav belysning [1]. For en sensor med en brønnkapasitet  $Q_{sat}$  er det metningen som begrenser det høyeste signalet, mens det er utlesningsstrømmen som begrenser det minste påviselige signalet. Ved å benytte sensormodellen kan det dynamiske området uttrykkes som [1]

$$DR = 20 \log \frac{i_{max}}{i_{min}} = 20 \log \frac{Q_{sat}}{\sqrt{q i_{dc} T + \sigma_V^2 + \sigma_C^2}}. \quad (2.16)$$

## 2.5 PSNR og MAE

Peak Signal-to-Noise Ratio (PSNR) og Mean Absolute Error (MAE) er de to enkleste og mest brukte kvalitetsmålene på bildekvaliteten til digitale bilder når man har et

referansebilde å sammenligne med [18]. De er tiltalende fordi de er enkle å beregne, har klare fysiske betydninger og er matematisk gunstig med tanke på optimalisering, men de trenger ikke nødvendigvis si så mye om den visuelle kvaliteten. I tillegg til disse benyttes også Mean Squared Error (MSE) som er gitt av

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i, j) - K(i, j)]^2 \quad (2.17)$$

hvor  $I$  og  $K$  er to  $m \times n$  monokrome bilder, og et av de er en tilnærming til det andre med støy. PSNR beregnes ved hjelp av MSE og er gitt av

$$PSNR = 10 \cdot \log_{10} \left( \frac{MAX_I^2}{MSE} \right) = 20 \cdot \log_{10} \left( \frac{MAX_I}{\sqrt{MSE}} \right) \quad (2.18)$$

Her er  $MAX_I$  maksverdien til en piksel, og vil med 8-bits pikselverdier være lik 255. MAE er gitt av

$$MAE = \frac{1}{n} \sum_{i=1}^n |f_i - y_i| = \frac{1}{n} \sum_{i=1}^n |e_i| \quad (2.19)$$

Som navnet tilsier er MAE et gjennomsnitt av de absolutte feilene  $e_i = f_i - y_i$ , hvor  $f_i$  er pikselverdi med støy mens  $y_i$  er den riktige pikselverdien.





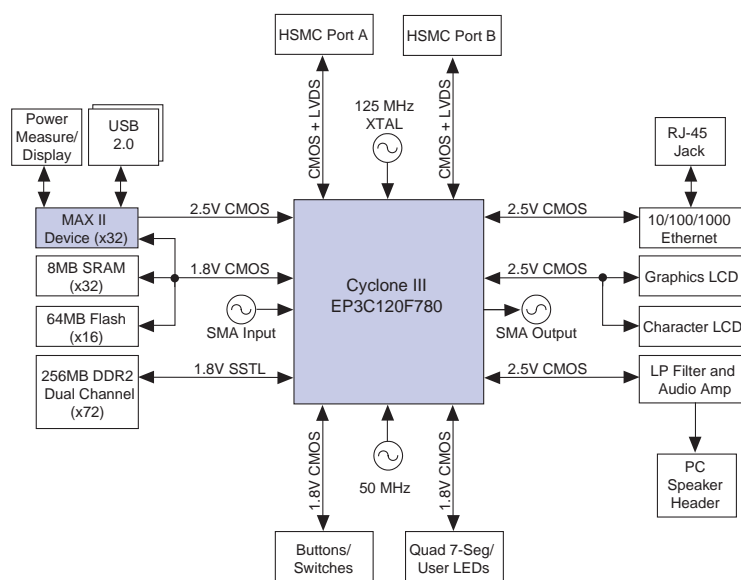
# Kapittel 3

## Metode

Dette kapitlet tar først for seg litt om Matlabsimuleringene som ble brukt i starten av utviklingen av systemet. Videre beskrives utstyret og programvaren som er brukt til å utvikle og implementere emulatoren. Deretter følger en beskrivelse av emulatoren som er designet og til slutt en beskrivelse av systemet som er utviklet for å teste emulatoren.

### 3.1 Matlabsimuleringer

Før de algoritmene som er valgt ble implementert i hardware, ble de simulert i Matlab. Resultatet ble sammenlignet med simuleringer av støyen hvor Matlab-funksjonene *poissrnd* og *randn* ble brukt til å generere henholdsvis Poisson- og normalfordelte pseudo-tilfeldige verdier. Når resultatet av simuleringene var tilfredsstillende ble det gått videre med å implementere algoritmene i hardware. Resultatet av simuleringene er i vist i kapittel 4.1, mens Matlab-koden er gjengitt i vedlegg D.

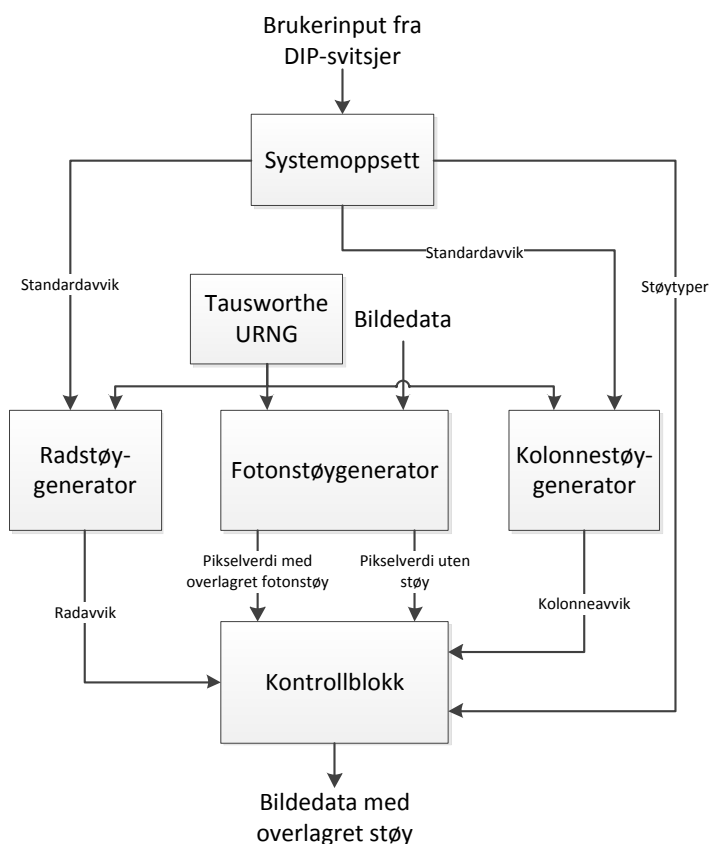


Figur 3.1: Cyclone III FPGA Development-kortets blokkdiagram

## 3.2 Utstyr og programvare

Systemet er utviklet og testet på *Cyclone III FPGA Development Kit*. Emulatoren er kodet i Verilog, og *Quartus II 9.1 sp1 Web Edition* er benyttet til kompilering av Verilog-koden, simulering av emulatoren, og for å programmere FPGAen. *Nios II 9.1 sp1 IDE* ble brukt til å compilere og laste over C-koden som ble brukt til å teste emulatoren.

Testsystemet implementerer en Nios II-prosessor og bruker ekstern DRAM, flash og et Ethernet PHY-grensesnitt, mens det endelige systemet består av DIP-svitsjer og LEDs i tillegg til systemet som er implementert på FPGAen.



Figur 3.2: Overordnet blokkdiagram for systemet.

## 3.3 Emulatoren

Emulatoren som er designet tar inn en 8-bits pikselverdi, en klokke, et reset-signal og en 8-bits verdi som er koblet til en DIP-svitsj (Dual In-Line Package). Utgangssignalene er en 8-bits pikselverdi med overlagret støy og 8 LEDs som er delt inn i to grupper. DIP-svitsjen og LEDene benyttes i første trinn i støygeneratoren hvor emulatoren settes opp. DIP-svitsjene brukes for å la brukeren sette opp systemet, mens LEDene viser statusen av denne prosessen. Systemet kan deles opp i seks blokker:

1. En del som lar brukeren velge oppløsning, støytype(r) og eventuelt standardavviket til rad- og kolonnestøyen.
2. Tausworthe Uniform Random Number Generator (URNG)
3. Fotonstøygenerator
4. Radstøygenerator
5. Kolonnestøygenerator
6. En kontrollblokk som overlagrer pikseldataen med de typene støy som er valgt og sender det ut.

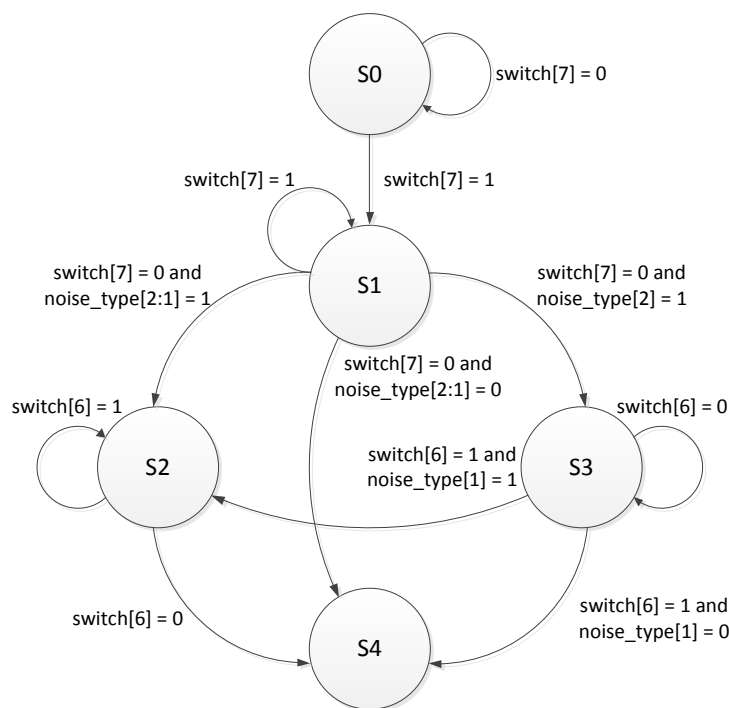
I tillegg til disse hovedblokkene inneholder emulatoren tre tellere som brukes for å holde orden på hvilken rad og kolonne pikselverdien som behandles hører til. Figur 3.2 viser et overordnet blokkdiagram over emulatoren.

### 3.3.1 Systemoppsett

Den delen av systemet som sørger for at brukeren får velge oppløsning, støytyper og SNR til rad- og kolonnestøyen består i hovedsak av en Finite-State Machine (FSM), hvor input fra DIP-svitsjer avgjør nestetilstanden og bestemmer utgangssignalene. FSMen er med andre ord av type Mealy. FSMen består av følgende tilstander:

- **S0:** Starttilstanden, her settes oppløsningen med DIP-svitsj 1 og 2. DIP-svitsj[2:1]=“01” gir VGA-oppløsning, “10” gir 720p (HD), mens “11” setter oppløsningen til 1080p (full HD).
- **S1:** I denne tilstanden velger man støytyper man ønsker å emulere med DIP-svitsj[3:1]. DIP-svitsj 1 er knyttet til fotonstøy, 2 til radstøy og 3 er knyttet til kolonnestøy. Støyen er aktiv når den tilhørende svitsjen er på (=1).
- **S2:** Her settes standardavviket til kolonnestøyen, og følgelig er det en forutsetning at kolonnestøy ble valgt i tilstand S1 for at FSMen kommer til denne tilstanden.
- **S3:** Denne tilstanden gir brukeren muligheten til å velge standardavviket til radstøyen, og slik som med tilstand S2, kreves det at radstøyen ble valgt i tilstand S1 for at FSMen skal nå denne tilstanden.
- **S4:** Når FSMen kommer til denne tilstanden har brukeren valgt oppløsning, støytyper og eventuelt standardavviket til rad- og kolonnestøyen, og det eneste som gjenstår er å sette *enable*-signalet til støygeneratorene høyt.

Figur 3.3 viser tilstandsdiagrammet til FSMen. Den viser hva som kreves for å komme til de forskjellige tilstandene, men det er en overgang fra hver tilstand som er utelatt fra figuren. Dersom *resetrn*-signalet går lavt vil nemlig FSMen hoppe til tilstand S0 uavhengig av hvilken tilstand den befinner seg i.



Figur 3.3: Tilstandsdiagram til FSMen.

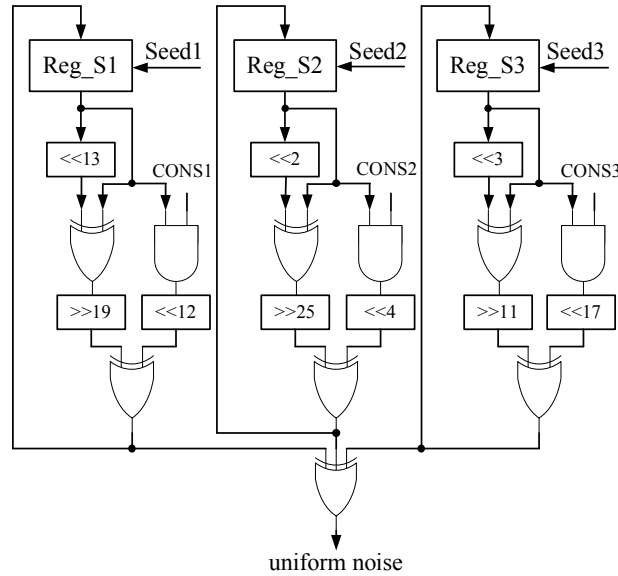
## 3.3.2 Tausworthe URNG

Det benyttes en Tausworthe URNG til å generere uniformtfordelte pseudo-tilfeldige tall. Vanligvis benyttes Linear Feedback Shift Registers (LFSR) til å generere uniformtfordelte tall i hardware. Selv om tradisjonelle LFSRs er tilstrekkelig til mange bruksområder, har en Tausworthe URNG bedre tilfeldighetskvaliteter med moderate hardware-kostnader [19].

Tausworthe-generatoren kombinerer tre LFSR-baserte URNGs for å oppnå bedre statistiske egenskaper. En generator med en periode  $\approx 2^{88}$  er foreslått i [20] og figur 3.4 viser hvordan denne generatoren er implementert i hardware. Generatoren genererer 32-bits verdier som representerer tall mellom 0 og 1. Tabell 3.1 viser de forskjellige konstantene og seed-verdiene generatoren bruker.

|            |              |
|------------|--------------|
| Seed 1     | 0xFFFFFFFF   |
| Seed 2     | 0xCCCCCC     |
| Seed 3     | 0xFF00FF     |
| Konstant 1 | 0xFFFFFFFFFE |
| Konstant 2 | 0xFFFFFFFFF8 |
| Konstant 3 | 0xFFFFFFFFF0 |

Tabell 3.1: Seed- og konstantverdier benyttet i Tausworthe URNG.



Figur 3.4: Arkitekturen til en Tausworthe URNG.

### 3.3.3 Fotonstøygenerator

Fotonstøy kan som tidligere nevnt beskrives ved hjelp av Poisson-statistikk, hvor den støy-frie pikselverdien er forventningsverdien og variansen (se kapittel 2.3.2.1). Standardavviket til den støy-frie pikselverdien er dermed lik roten av pikselverdien.

Poissonfordelingen kan tilnærmes med en normalfordeling hvor forventningsverdien og variansen er lik. Denne tilnærmingen blir bedre ved høyere forventningsverdier. I denne oppgaven er det gått ett skritt lenger og brukt en forenklet “normalfordeling” for å emulere fotonstøy. Støygeneratoren benytter den empiriske regelen (eller 3-sigma-regelen) som sier at omtrent 68 % av verdiene ligger innenfor ett standardavvik, omtrent 95 % av verdiene ligger innenfor to standardavvik og omtrent 99.7 % av verdiene ligger innenfor tre standardavvik i en normalfordeling. Dersom resultatet fra Tausworthe-generatoren er  $rnd \leq (2^{32} - 1) \cdot 0.68$  vil utgangsdataen være gitt av

$$output = input \pm \sqrt{input} \cdot rnd. \quad (3.1)$$

$2^{32} - 1$  er maksverdien til  $rnd$ , og siden  $rnd$  er uniformt fordelt vil dette sørge for at 68 % av verdiene som genereres vil være innenfor ett standardavvik. Det minst signifikante bitet til  $rnd$  brukes til å avgjøre om det skal være et negativt eller positivt avvik.

Hvis  $(2^{32} - 1) \cdot 0.68 < rnd \leq (2^{32} - 1) \cdot 0.95$  er utgangsdataen gitt av

$$output = input \pm \left[ \sqrt{input} \cdot rnd + \sqrt{input} \right]. \quad (3.2)$$

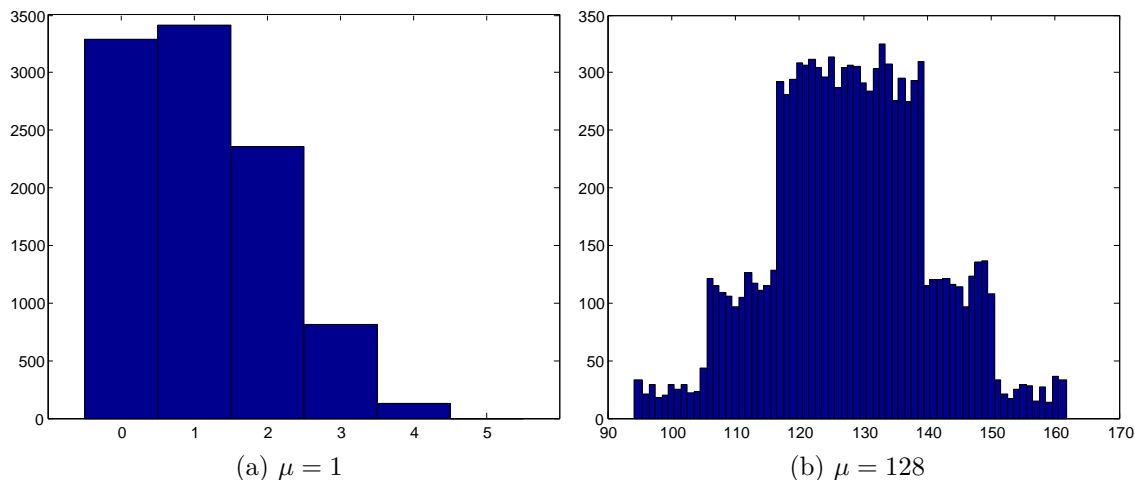
Dette gjør at 27 % av verdiene vil ligge mellom ett og to standardavvik fra forventningsverdien, og 95 % vil ligge innenfor to standardavvik.

Når  $rnd$  er større enn  $(2^{32} - 1) \cdot 0.95$  er utgangsverdien gitt av

$$output = input \pm \left[ \sqrt{input} \cdot rnd + 2 \cdot \sqrt{input} \right]. \quad (3.3)$$

## KAPITTEL 3. METODE

Dermed vil 5 % av verdiene ligge mellom to og tre standardavvik fra forventningsverdien, og 100 % vil ligge innenfor tre standardavvik. Figur 3.5a og 3.5b viser de 10000 første verdiene som genereres med denne generatoren når inngangsverdien er henholdsvis 1 og 128.



Figur 3.5: De 10000 første verdiene som genereres med inngangsverdiene 1 og 128.

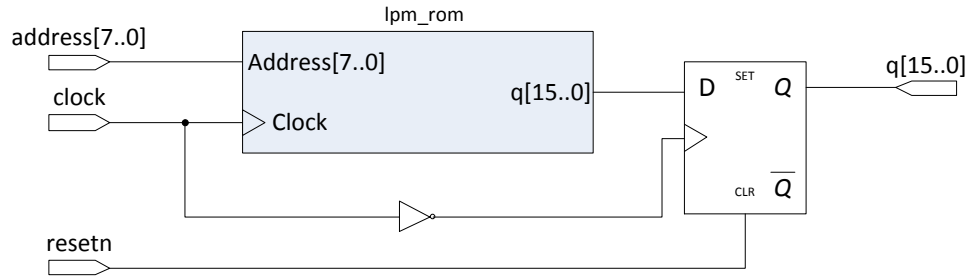
Standardavviket til alle de mulige inngangsverdiene er lagret i en oppslagstabell som vist i tabell 3.2. Som man kan se av denne tabellen er dataene i oppslagstabellen på 16 bit, hvor de 4 mest signifikante bitene (MSB) angir heltallsdelen, mens de 12 minst signifikante bitene (LSB) angir fraksjonsdelen.

| Adresse | Data         | Binær data          |
|---------|--------------|---------------------|
| 0       | $\sqrt{0}$   | 0000 0000 0000 0000 |
| 1       | $\sqrt{1}$   | 0001 0000 0000 0000 |
| 2       | $\sqrt{2}$   | 0001 0110 1010 0000 |
| .       | .            | .                   |
| .       | .            | .                   |
| .       | .            | .                   |
| 255     | $\sqrt{255}$ | 1111 1111 0111 1111 |

Tabell 3.2: Oppslagstabell.

Oppslagstabellen er implementert med en ROM-modul, som er en innebygget MegaWizard-funksjon i Quartus II. I tillegg inneholder den et register på utgangen som er klokke med en invertert klokke, slik at utgangen er gyldig på positive klokkeflanker (se figur 3.6). ROM-modulen initialiseres med en Memory Initialization File (.mif) som ble generert i Matlab. Denne Matlab-koden finnes i vedlegg F.

ROM-modulen er pipelined for å takle høye frekvenser. Derfor tar det 3 klokkesykler fra det påtrykkes en adresse til den tilhørende verdien ligger på utgangen. Når dataen fra ROMen er klar, multipliseres denne med resultatet fra Tausworthe-generatoren. Makskravet til systemet er 60 rammer per sekund med full



Figur 3.6: Arkitekturen til oppslagstabellen.

HD-oppløsning, som gir en frekvens på

$$\text{frekvens} = 1920 \cdot 1080 \cdot 60 = 124\,416\,000 \text{ Hz} = 124.416 \text{ MHz.} \quad (3.4)$$

For at systemet skulle takle så høye frekvenser var det nødvendig å pipeline multiplikasjonen. Dette ble løst ved å benytte “Shift and add”-algoritmen som vist i figur 3.8. Forskjellen mellom multiplikatoren i fotonstøygeneratoren og multiplikatorene i rad- og kolonnestøygeneratorene, er at fotonstøygeneratoren utfører en  $16 \times 32$ -bits multiplikasjon, mens de to andre generatorene utfører en  $8 \times 32$ -bits multiplikasjon. Dette gjør at fotonstøygeneratoren bruker en klokkesykel mer på multiplikasjonen og totalt 4 klokkesykler.

Når resultatet fra multiplikasjonen er klart, avrundes resultatet til nærmeste heltall, før det gjøres en sjekk mot *rnd* for å avgjøre om fotonstøyen skal ligge innenfor ett, to eller tre standardavvik og om avviket skal være negativt eller positivt.

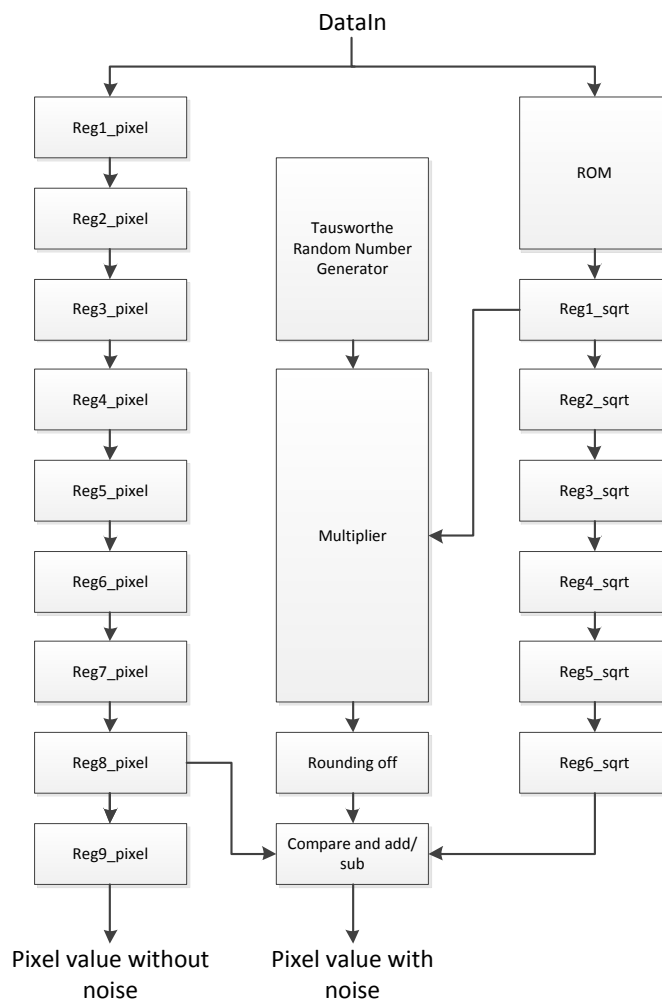
### 3.3.4 Radstøygenerator

Radstøy kan modelleres med en normalfordeling hvor forventningsverdien er lik null. Radstøyen emuleres ved hjelp av den samme tilnærmingen til normalfordelingen som benyttes i fotonstøygeneratoren, og standardavviket bestemmes av brukeren ved oppstart av systemet.

I radstøygeneratoren multipliseres verdien fra inngangssignalet *row\_std\_dev* med verdien fra registeret kalt *Reg\_S3* i figur 3.4. I likhet med utgangssignalet til Tausworthe-generatoren er dette et 32-bits tall, men det har noe dårligere tilfeldighetskvaliteter. Multiplikasjonen er pipelined på grunn av tidligere nevnte krav om at systemet må være i stand til å kunne kjøre på 124.416 MHz (se (3.4)), og utføres ved å benytte “shift and add”-algoritmen som vist i figur 3.8. AND-operasjonene i figuren er en bit-wise AND-operasjon.

Når multiplikasjonen er ferdig sjekkes verdien i registeret *Reg\_S2* for avgjøre om støyen skal ligge innenfor ett, to eller tre standardavvik og om avviket skal være positivt eller negativt. Avviket lagres i et 8-bits register, mens fortegnet lagres i et eget register (1 = negativ og 0 = positiv). Det siste trinnet i radstøygeneratoren ligner veldig på det siste trinnet i fotonstøygeneratoren og det siste trinnet i kolonnestøygeneratoren.

Radstøygeneratoren vil naturlig nok bare komme opp med et radavvik en gang per rad, og da i det den første pikselen på en rad behandles. Det siste trinnet vil



Figur 3.7: Blokkdiagram av fotonstøygeneratoren.

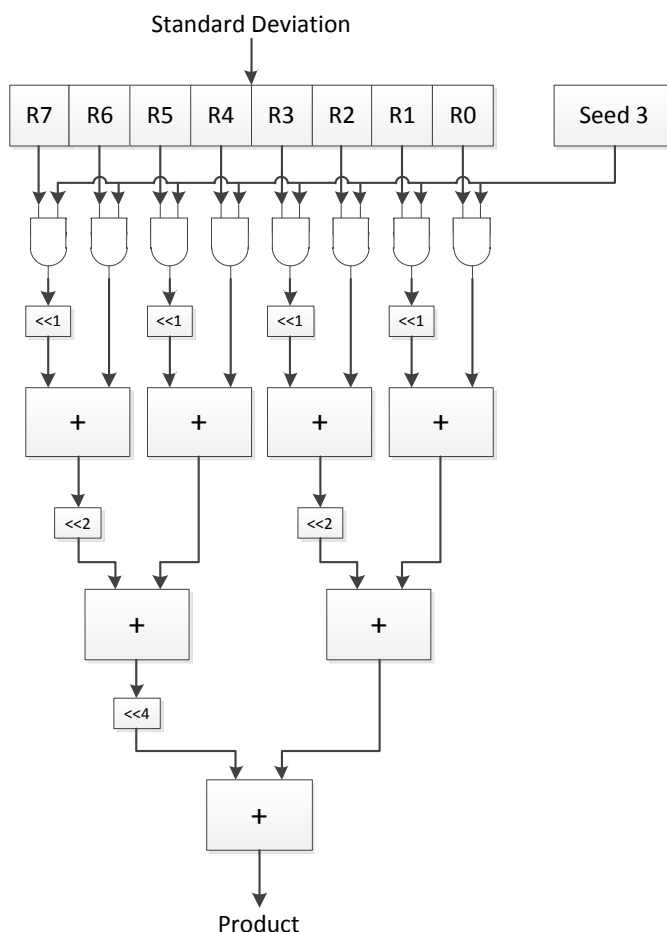
derfor sjekke en kolonneteller, og kun utføres dersom denne er lik null (kolonne null er den første). Multiplikasjonen vil allikevel beregne et nytt resultat hver klokkesykel. Dermed vil resultatet fra multiplikasjonen være klart i det kolonnetelleren igjen blir null.

## 3.3.5 Kolonnestøygenerator

Kolonnestøy kan i likhet med radstøy modelleres med en normalfordeling hvor standardavviket er lik null. Den samme algoritmen som gir en tilnærmet normalfordeling i foton- og radstøygeneratoren, benyttes også i kolonnestøygeneratoren. Og slik som med radstøygeneratoren bestemmes standardavviket til kolonnestøyen av brukeren ved oppstart.

Kolonnestøygeneratoren benytter en multiplikator lik den i radstøygeneratoren (se 3.8), men standardavviket multipliseres med *Seed 2* i stedet for *Seed 3*. I det multiplikasjonen er ferdig, sjekkes verdien i registeret *Reg\_S3* for avgjøre om støyen skal ligge innenfor ett, to eller tre standardavvik og om avviket skal være positivt eller negativt, på samme måte som i de to andre støygeneratorene. I motsetning til





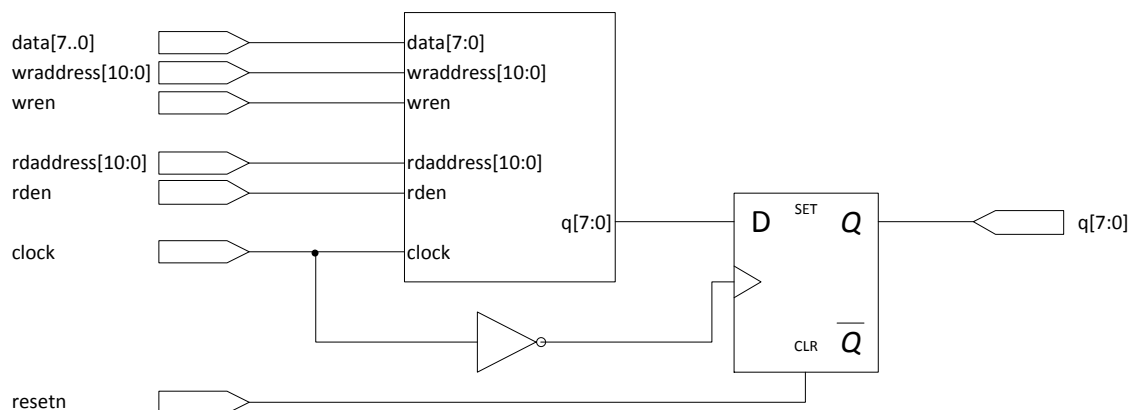
Figur 3.8: Blokkdiagram av multiplikatoren i radstøygeneratoren.

i radstøygeneratoren, lagres avviket og dets fortegn i det samme 8-bits registeret. Dette gjør at kolonneavviket har en maksverdi på 127 (7 bit).

En annen forskjell fra radstøygeneratoren er at siden bildene leses ut rad for rad, må kolonneavviket lagres til neste gang en piksel fra samme kolonne behandles. Dette er løst ved å benytte en dual-port RAM-instans på 2048 Byte. I likhet med ROM-modulen som brukes som oppslagstabell i fotonstøygeneratoren, er det også satt på et register på utgangen av dual-port RAM-enheten som klokkes på negative klokkeflanker. Grunnen til dette er som sagt for at verdien som leses ut skal være stabil på positive klokkeflanker. Figur 3.9 viser arkitekturen til RAM-modulen. To tellere brukes som lese- og skriveadresse til RAMen.

Kolonneavvikene blir bestemt i det den øverste raden i bildet leses ut. I det den siste pikselverdien på den første raden er sendt ut, bør kolonneavviket til den første kolonnen være tilgjengelig. Siden det tar 3 klokkesykler fra man påtrykker en adresse til den korresponderende verdien er tilgjengelig, er man nødt til å starte å lese ut av RAMen samtidig som man skriver de tre siste kolonneavvikene til den. Dermed er man nødt til å benytte en dual-port RAM, som skiller seg fra single-port RAM ved at den har denne evnen til å lese og skrive til forskjellige adresser samtidig.

I det den første raden av piksler er sendt ut, vil kolonnestøygeneratoren gå over til å kun lese ut kolonneavvikene av RAMen. I likhet med multiplikasjonen



Figur 3.9: Arkitekturen til RAM-modulen.

i radstøygeneratoren, utføres multiplikasjonen i kolonnestøygeneratoren selv om resultatet ikke benyttes.

### 3.4 Testsystem

Systemet som er brukt til å teste emulatoren består av en Nios II-prosessor, DDR2 SDRAM, flash og et fysisk Ethernet-grensesnitt. Prosessoren kjører *MicroC/OS-II* og en modifisert versjon av Altera sitt Simple Socket Server-program. De modifiserte metodene er gjengitt i vedlegg C.

Testsystemet er designet slik at en Nios II-prosessor tar imot bildedata over Ethernet og lagrer det i et buffer i DDR2 SDRAMen. I det CPUen har mottatt ett bilde og Ethernet-tilkoblingen er lukket, settes emulatoren i gang og prosessoren starter å sende bildedataen. CPUen vil vente til den har mottatt en pikselverdi med overlagret støy fra emulatoren før den sender en ny pikselverdi. Verdiene som prosessoren mottar fra emulatoren lagres i et nytt buffer, og når den har mottatt et helt bilde avventer den en ny Ethernet-tilkobling. Når denne Ethernet-tilkoblingen er på plass, vil CPUen sende bildet med overlagret støy.

Matlab ble brukt til å lese bildedata, og for å sende den til CPUen. For å motta data fra FPGAen ble det benyttet et *C#*-program som skrev resultatet til en tekstfil. Dette programmet finnes i vedlegg E. Matlab ble igjen brukt for å lese denne tekstfilen og vise resultatet.

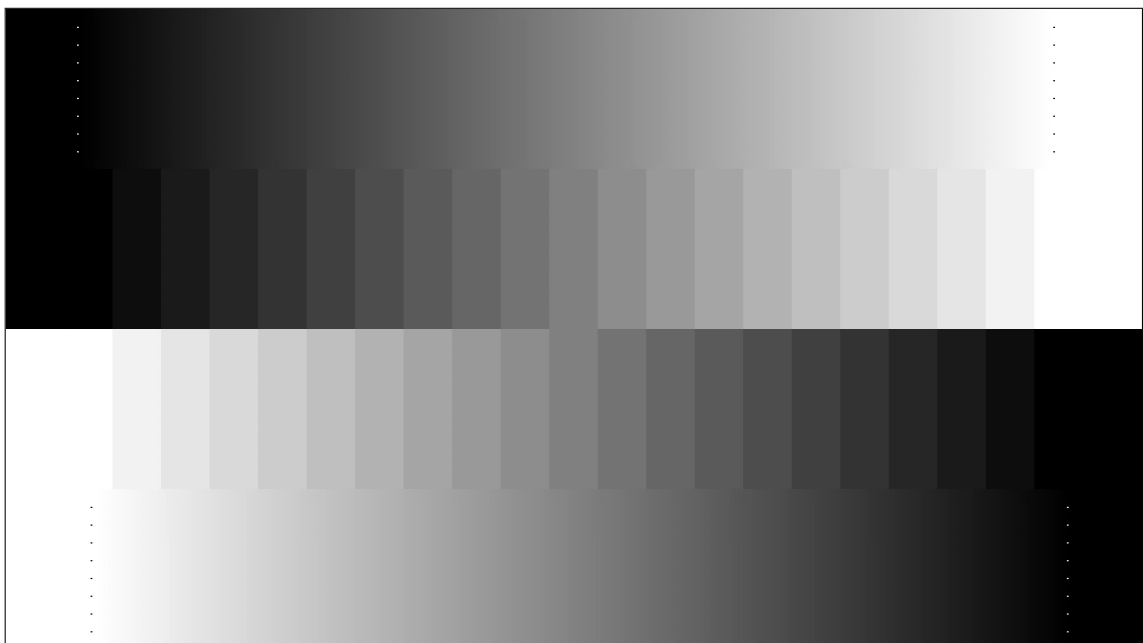
# Kapittel 4

## Resultater

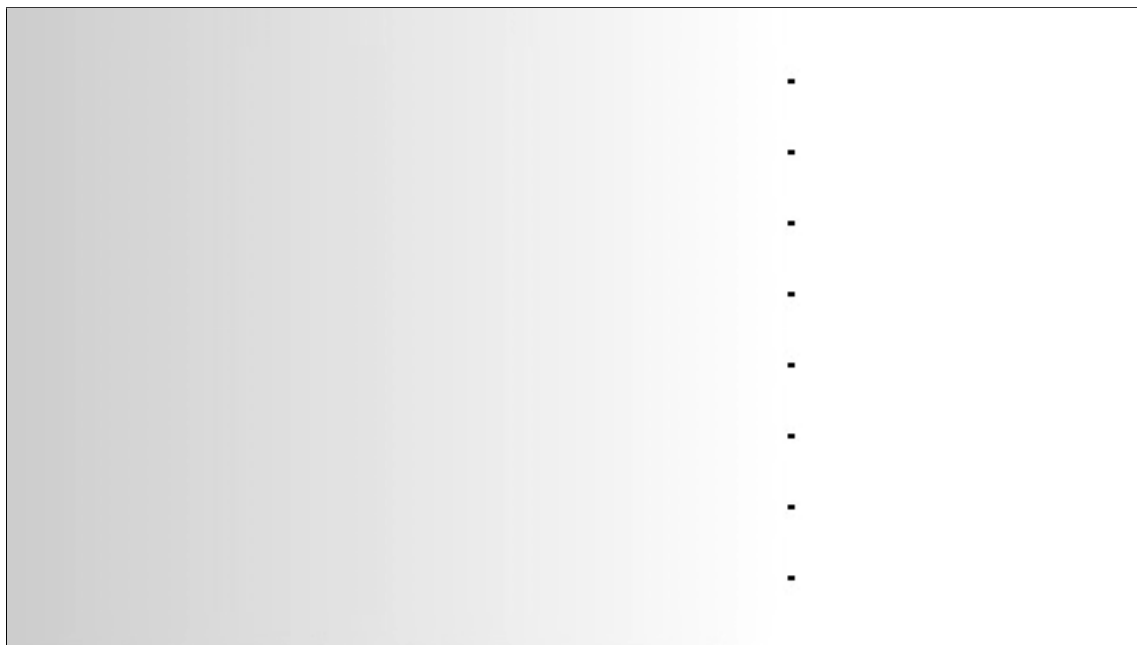
Dette kapittelet går først igjennom resultatene fra Matlabsimuleringene og testene på FPGAen som er utført. Videre tar kapittelet for seg Quartus II-simuleringer som viser noen av de viktigste funksjonene til emulatoren, før kapittelet avsluttes med en kort presentasjon av kompileringsresultatene til emulatoren fra *Quartus II* presenteres.

### 4.1 Matlabsimuleringer

Figurene 4.1 og 4.2 viser henholdsvis testbildet som er brukt og en forstørret del av testbildet, uten noen form for støy. Fordi testbildet har full HD-oppløsning er det litt vanskelig å se virkningen av støy. Det er derfor tatt med en forstørret del av alle test- og simuleringsbildene.



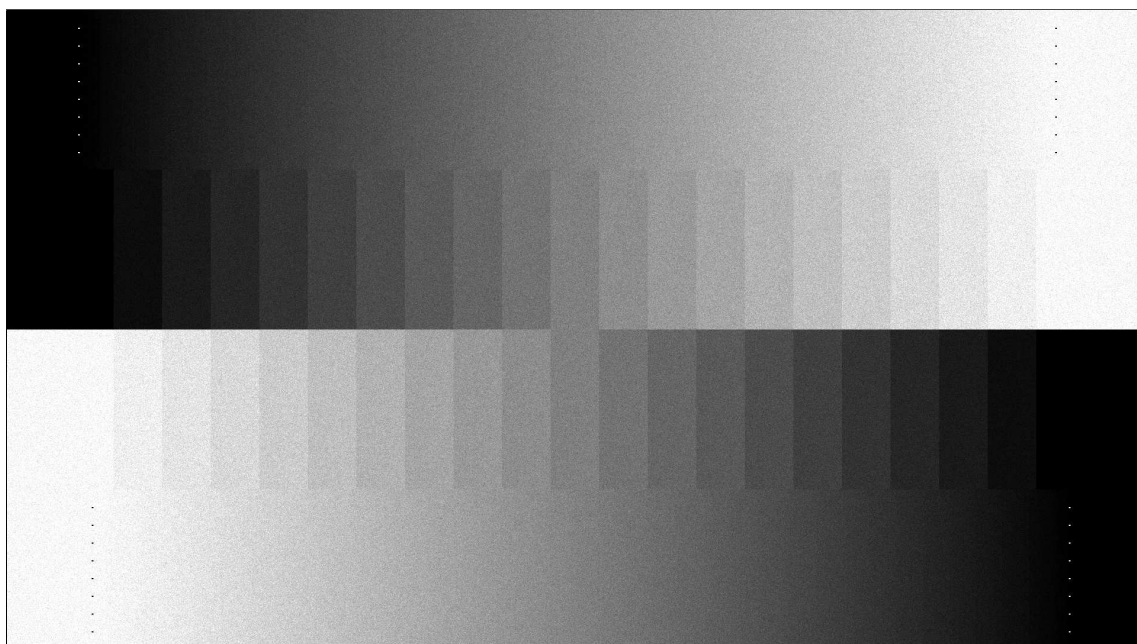
Figur 4.1: Testbildet uten støy.



Figur 4.2: Nærbilde av øverste høyre hjørnet av testbildet.

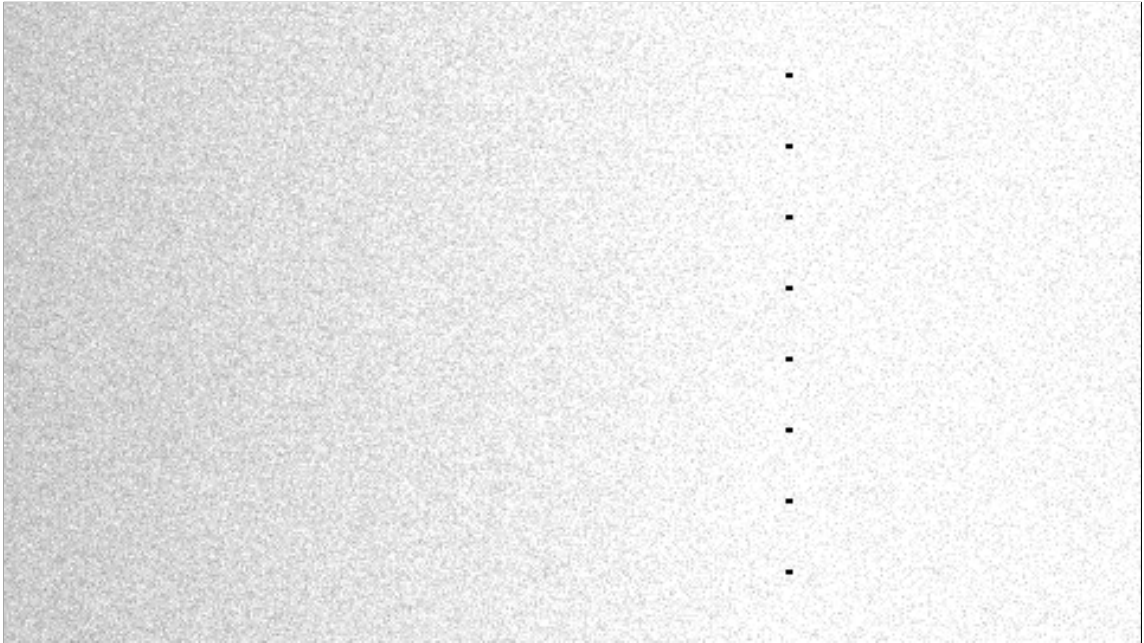
### 4.1.1 Simulering med Matlabfunksjoner for å generere Poisson- og normalfordelte pseudo-tilfeldige tall

Dette delkapittelet tar for seg simulering av foton-, rad- og kolonnestøy ved å benytte Matlabfunksjonene *poissrnd* og *randn* til å generere henholdsvis Poisson- og normalfordelte pseudo-tilfeldige verdier. Disse simuleringene er foretatt for å



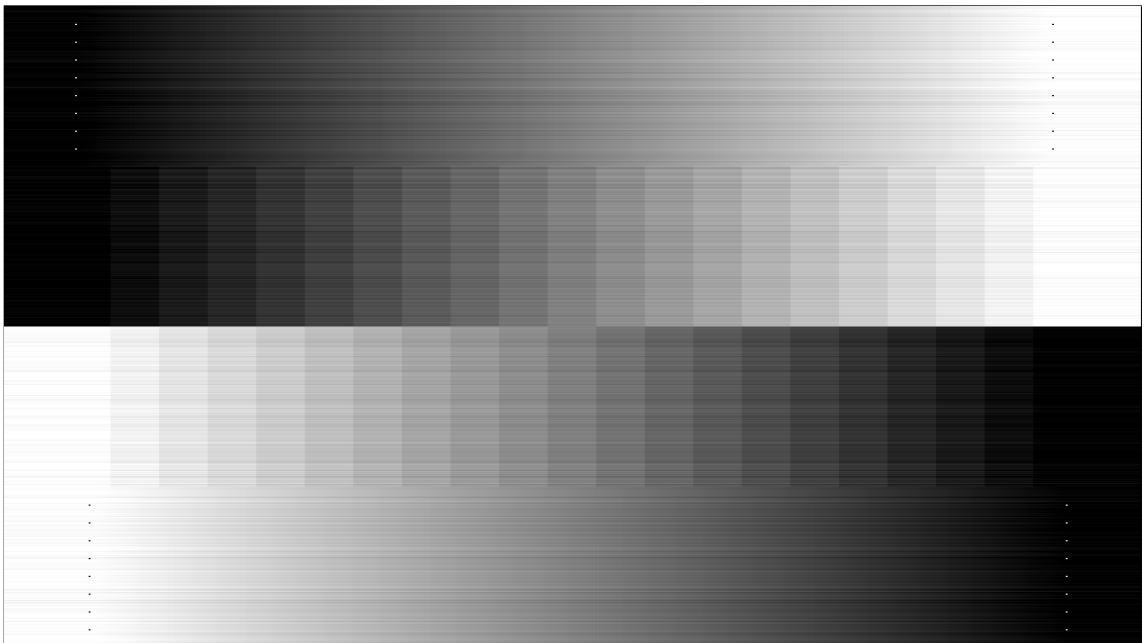
Figur 4.3: Resultatet etter å ha brukt Matlab-funksjonen *poissrnd* til å generere Poisson-fordelt støy på testbildet.

sammenlignes med simuleringene og resultatene av algoritmene emulatoren benytter til å emulere støy.

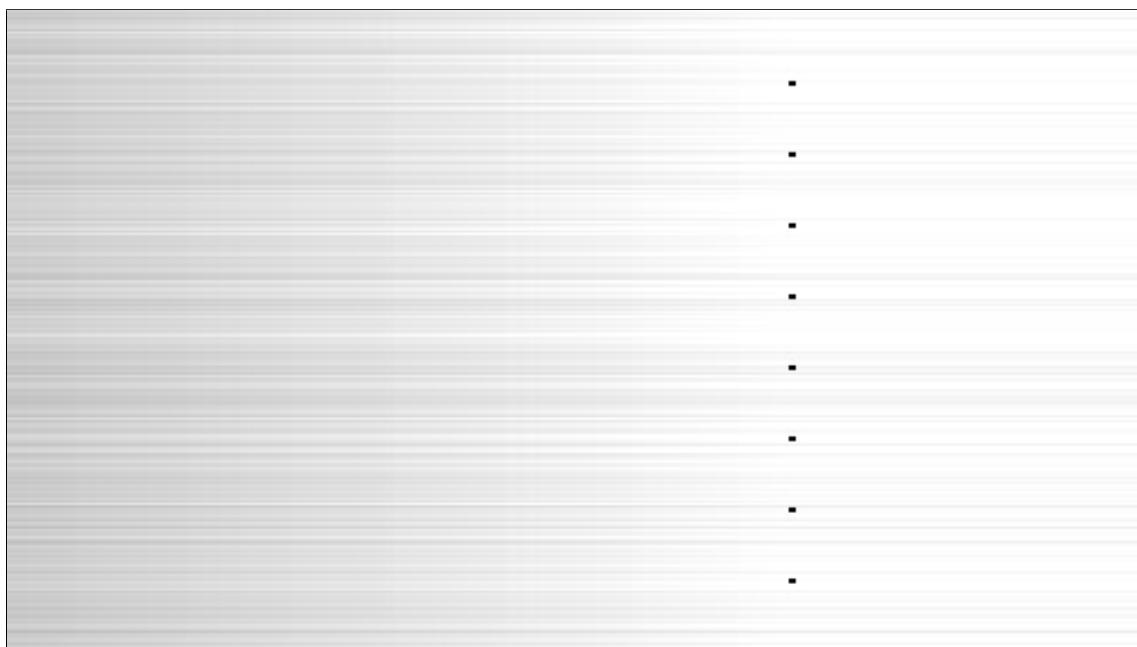


Figur 4.4: Nærbilde av simulert fotonstøy.

Figurene 4.3 og 4.4 viser simuleringen av fotonstøy. Det som er verdt å merke seg her, er at støyen øker med lysintensiteten (forventningsverdien) som forventet. Det må nevnes at i det pikselverdiene nærmer seg maksverdien 255 vil ikke støyen være Poissonfordelt lenger, da alle verdiene over maksverdien vil avrundes til 255.

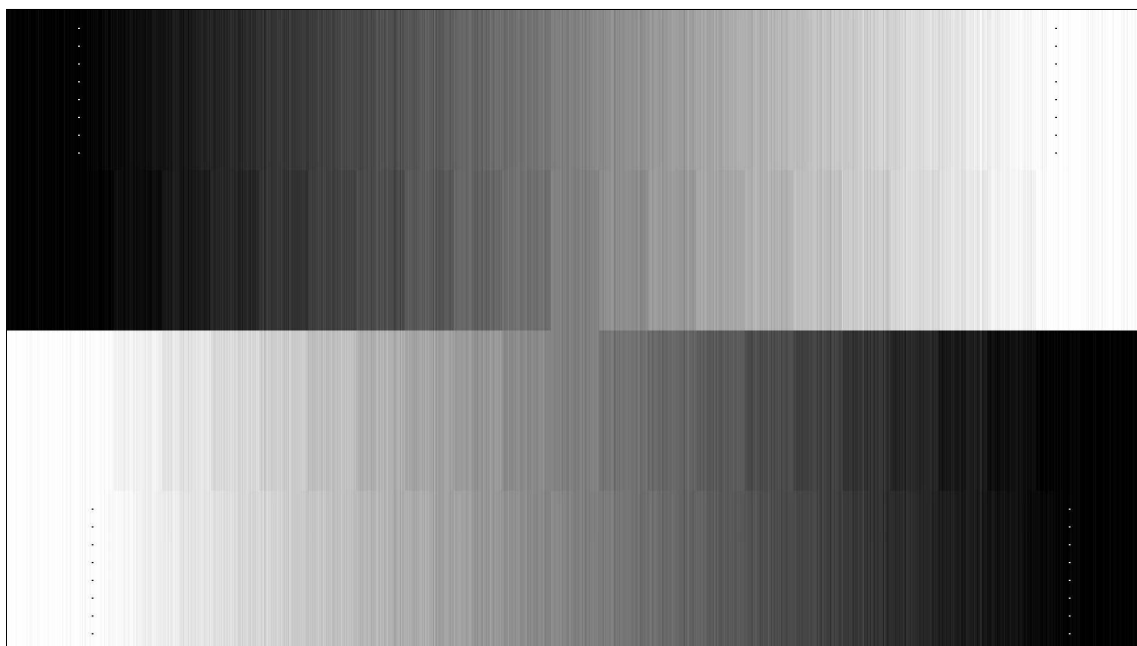


Figur 4.5: Simulert radstøy.



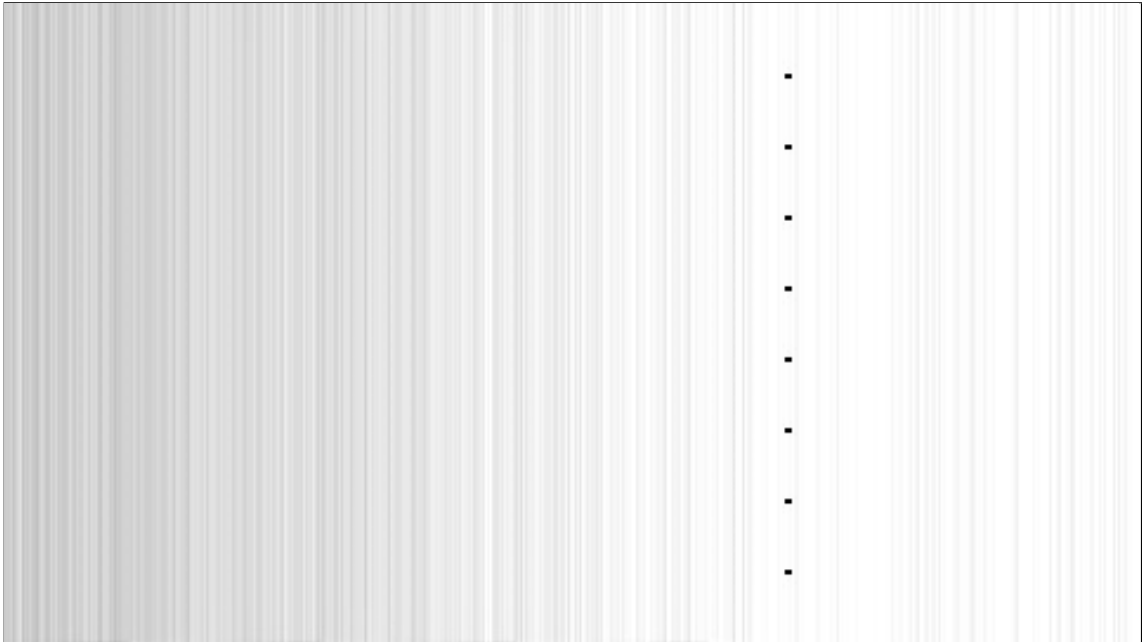
Figur 4.6: Nærbilde av simulert radstøy.

I figurene 4.5 og 4.6 ser man simulert radstøy med et standardavvik lik 7. Som man ser har noen rader et positivt avvik, mens andre har et negativt avvik. Summen av avvikene skal være lik null, men dette kan man ikke se ut fra bildene.



Figur 4.7: Simulert kolonnestøy.

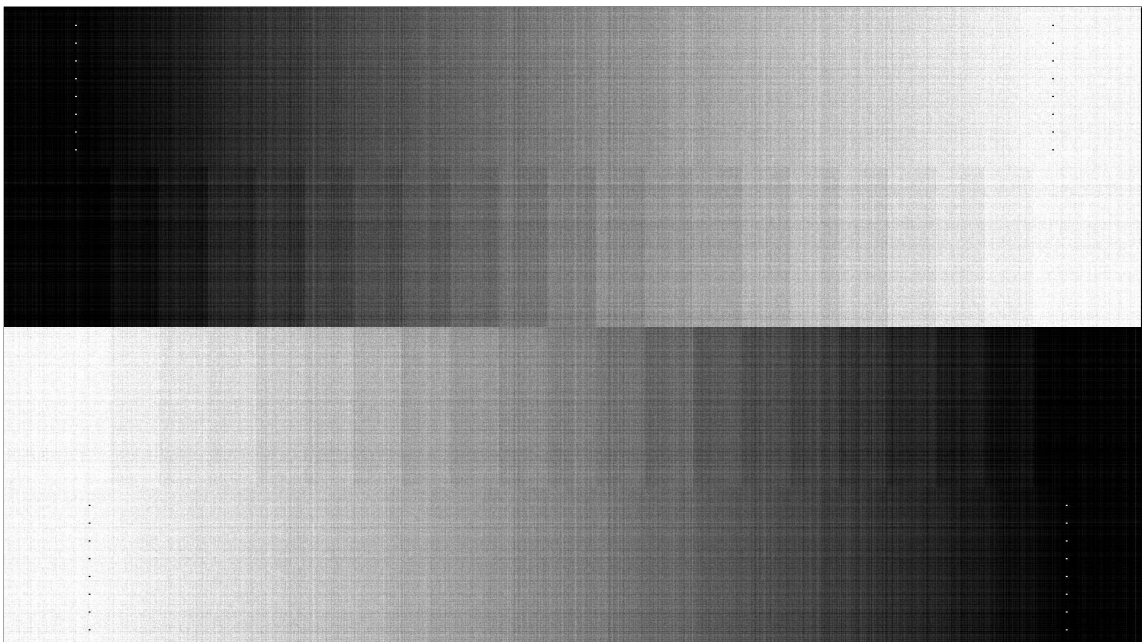
Simuleringene av kolonnestøy kan sees i figurene 4.7 og 4.8. Denne kolonnestøyen har et standardavvik lik 7. Slik som tilfellet er med rader som er beheftet med radstøy, har noen av kolonnene et positivt avvik, mens andre har et negativt avvik.



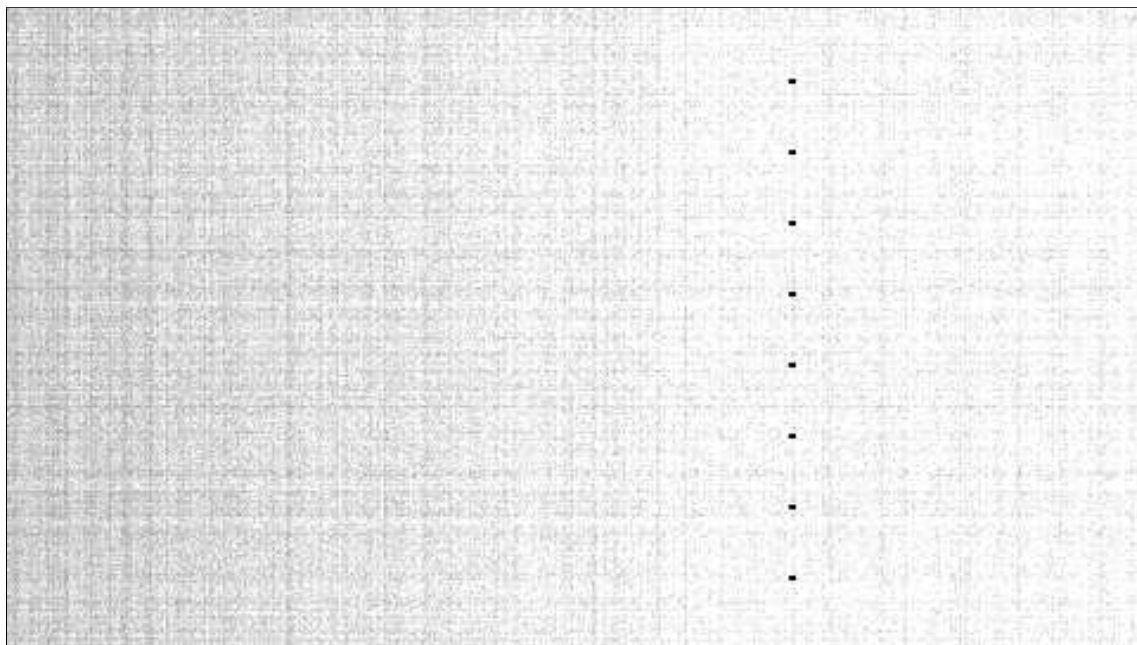
Figur 4.8: Nærbilde av simulert kolonnestøy.

Figurene 4.9 og 4.10 viser simuleringsresultatene når testbildet utsettes for simulert foton-, rad- og kolonnestøy. Standardavviket til rad- og kolonnestøyen er også her satt til 7. De tydelige linjene fra figurene 4.5 - 4.8 er fortsatt synlige, men her er de ikke like tydelige.

Tabell 4.1 viser noen utvalgte kvalitetsmål på simuleringsresultatene. Det er her verdt å merke seg at selv om rad- og kolonnestøyen er mer synlig enn fotonstøyen, så gir de bedre PSNR, MSE og MAE. Ikke overraskende er det summen av alle



Figur 4.9: Simulert foton-, rad- og kolonnestøy.



Figur 4.10: Nærbilde av simulert foton-, rad- og kolonnestøy.

Tabell 4.1: Kvalitetsmål på simuleringsresultatene

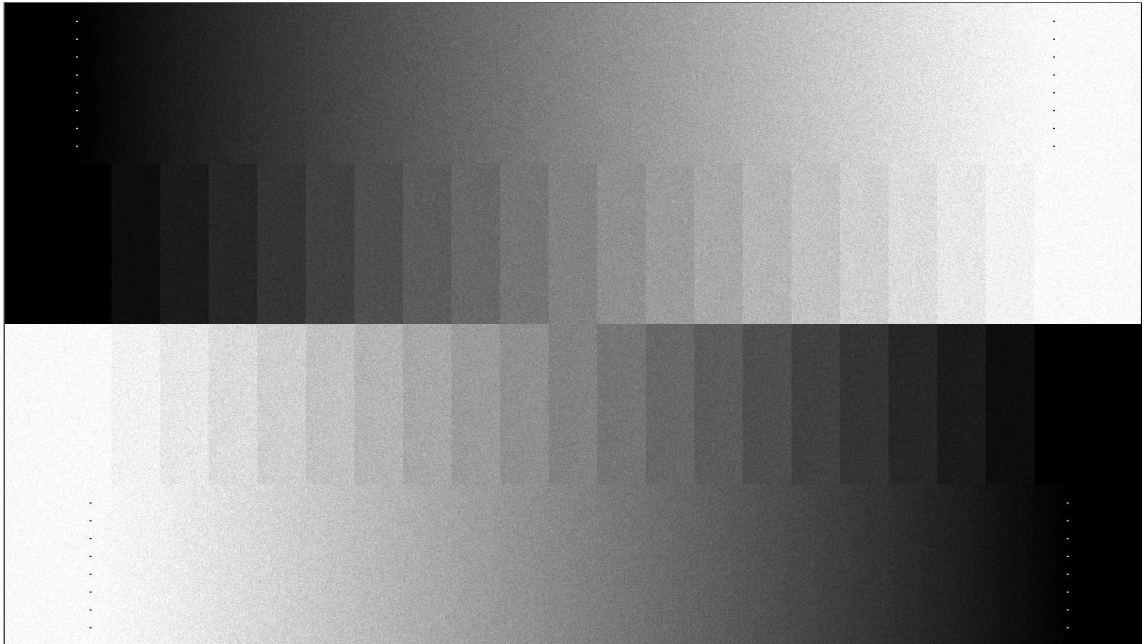
| Type støy            | PSNR (dB) | MSE     | MAE    |
|----------------------|-----------|---------|--------|
| Simulert fotonstøy   | 31.2403   | 48.8710 | 3.4059 |
| Simulert radstøy     | 34.3584   | 23.8364 | 2.6399 |
| Simulert kolonnestøy | 34.6527   | 22.2745 | 2.4849 |
| Alle                 | 28.4742   | 92.3976 | 5.1167 |

støytypene som gir dårligst PSNR.

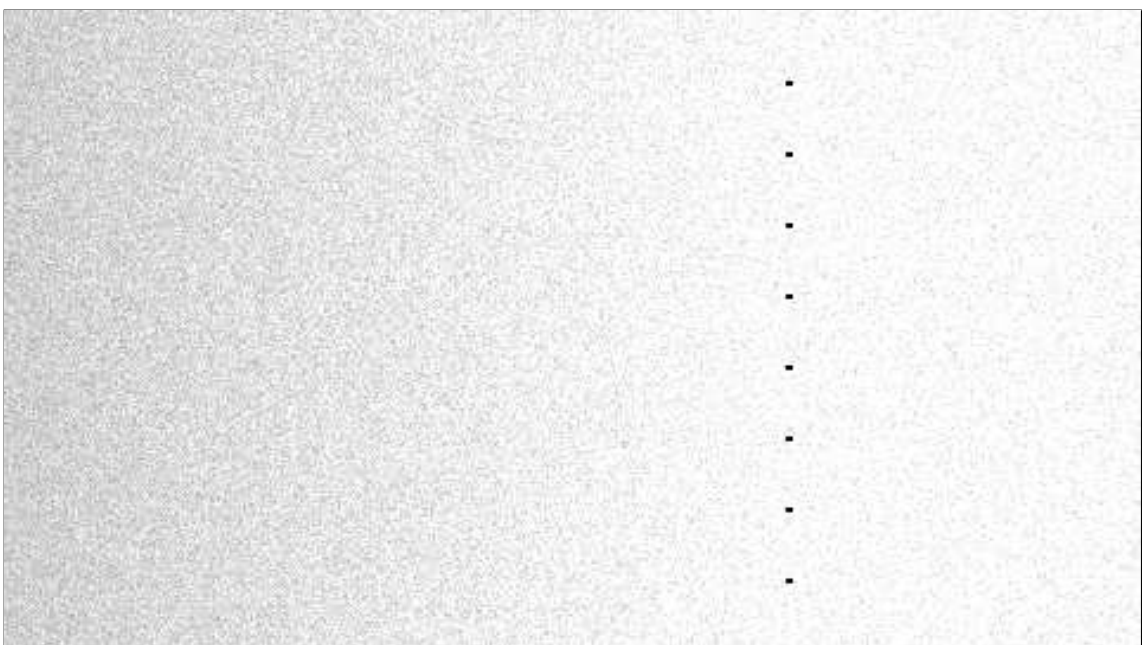


### 4.1.2 Simulering av algoritmene som er brukt til å emulere foton-, rad- og kolonnestøy på FPGAen

Dette kapitlet presenterer resultatene fra simuleringene av algoritmene som er brukt til å emulere foton-, rad- og kolonnestøy på FPGAen. Disse simuleringene ble gjort for å gi en pekepinn på hvordan resultatet ville bli, før algoritmene ble implementert på FPGAen.



Figur 4.11: Resultatet av simulering av algoritmen som emulerer fotonstøy.

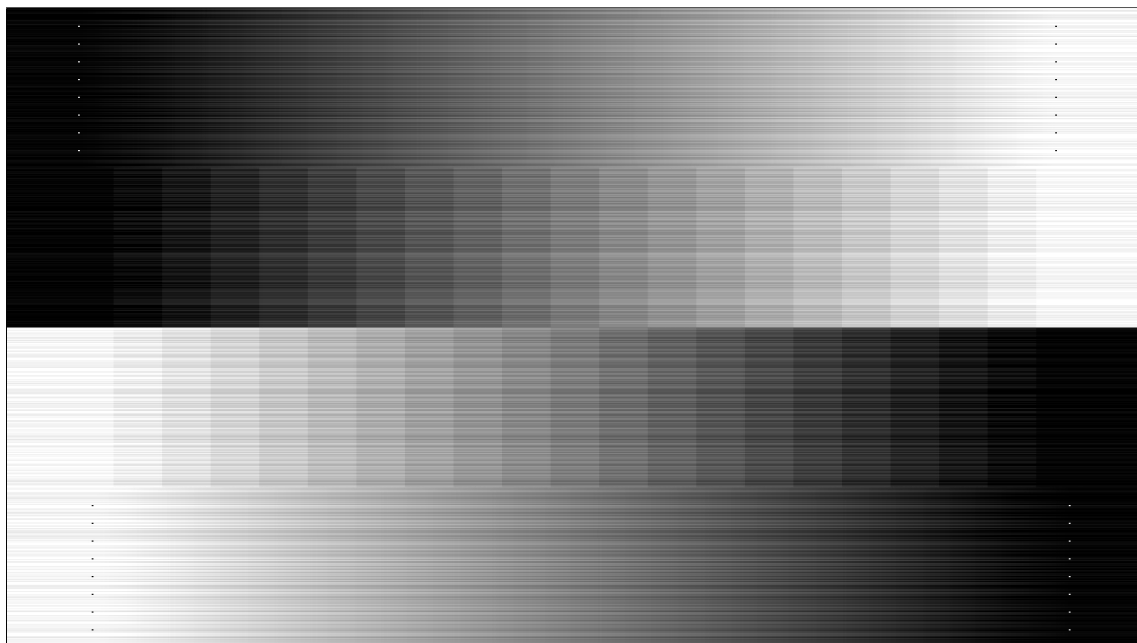


Figur 4.12: Nærbilde av emulert fotonstøy.

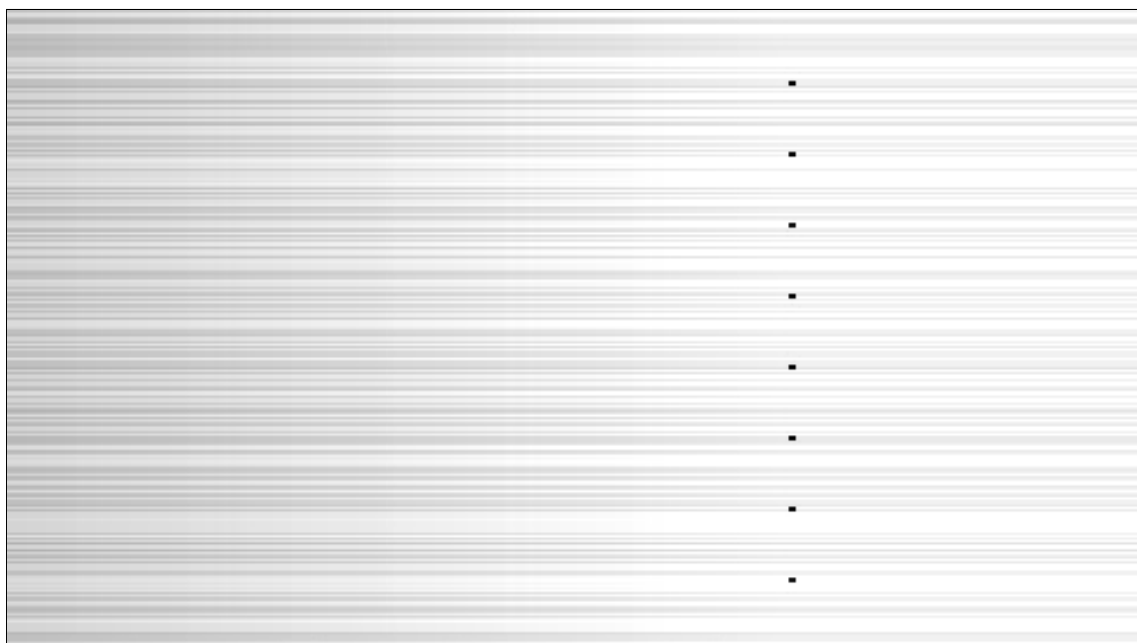
## KAPITTEL 4. RESULTATER

---

Resultatet fra simuleringen av algoritmen som er brukt til å generere fotonstøy kan man se i figur 4.11 og 4.12. Disse bildene kan sammenlignes med figurene 4.3 og 4.4.

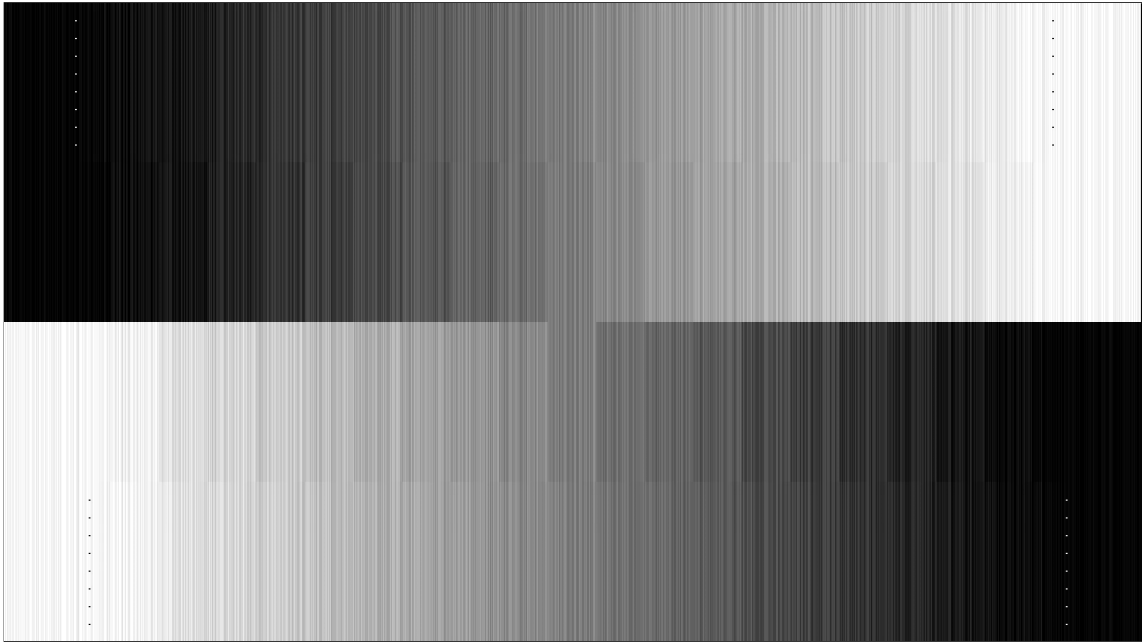


Figur 4.13: Resultatet av simulering av algoritmen som emulerer radstøy.



Figur 4.14: Nærbilde av emulert radstøy.

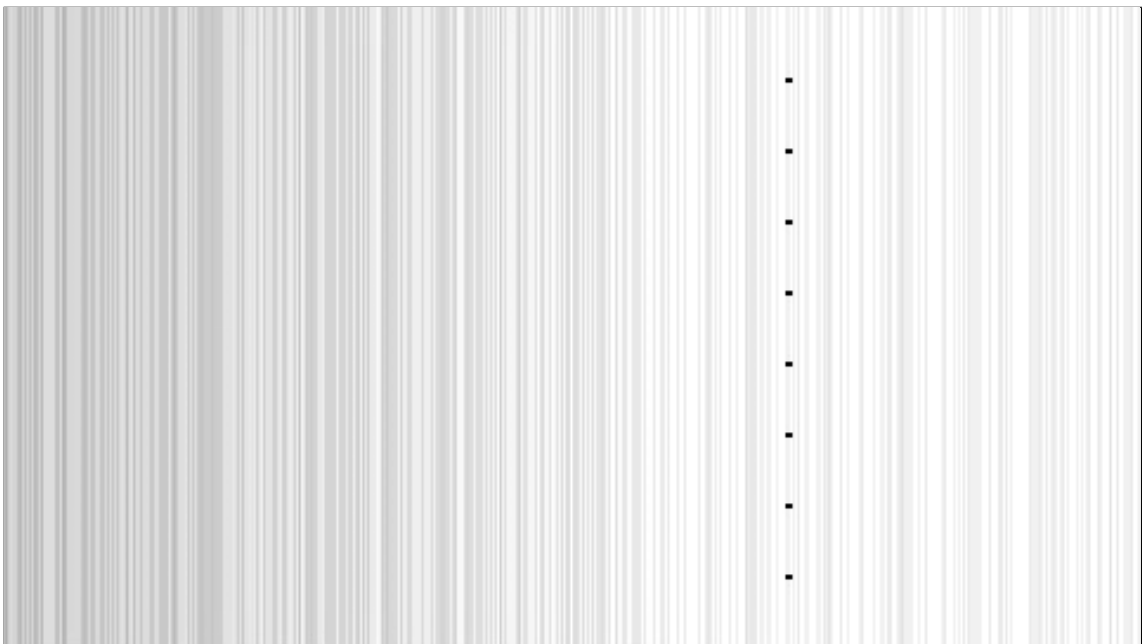
Figurene 4.13 og 4.14 viser emulert radstøy. Standardavviket til den emulerte radstøyen er satt 7. Sammenlignet med figur 4.5 og 4.6 ser man at radavvikene generelt er litt høyere i figurene som viser den emulerte radstøyen.



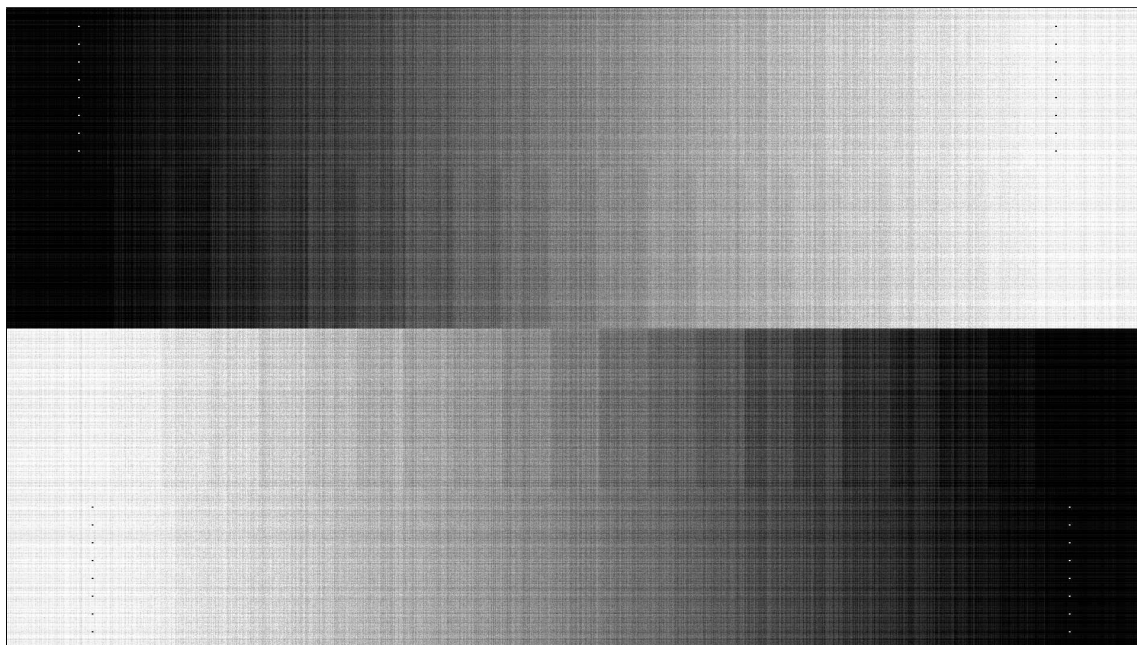
Figur 4.15: Resultatet av simulering av algoritmen som emulerer kolonnestøy.

I figur 4.15 og 4.16 ser man emulert kolonnestøy. Standardavviket til kolonnestøyen satt til 7, og i likhet med den emulerte radstøyen, ser det ut til at kolonneavvikene i disse figurene generelt er noe høyere enn den simulerte kolonnestøyen i figur 4.7 og 4.8.

Figurene 4.17 og 4.18 viser resultatet til simuleringen av algoritmene som er brukt til å emulere foton-, rad- og kolonnestøy i det samme bildet. Her er standardavvikene til rad- og kolonnestøyen satt til 7. Ved å sammenligne bildene

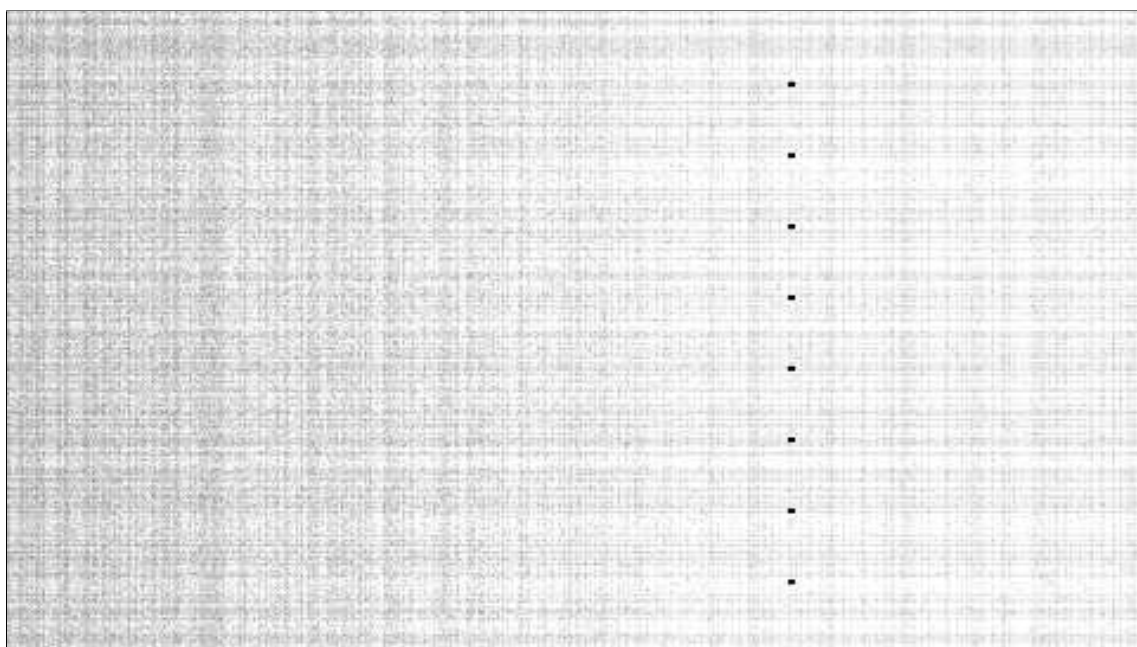


Figur 4.16: Nærbilde av emulert kolonnestøy.



Figur 4.17: Simulering av algoritmene som benyttes til å emulere foton-, rad- og kolonnestøy.

med figurene 4.9 og 4.10 ser man at simuleringene av emulatoren tilnærming til Poisson- og normalfordeling gir noe mer støy enn hvis man benytter en ekte Poisson- og normalfordeling. Ved å sammenligne tabell 4.1 med 4.2 ser man at simuleringene av algoritmene, som emulatoren bruker for å emulere støy, gir en PSNR som er 2-3 dB lavere enn tilsvarende simuleringer med Matlabs måte å generere normal- og



Figur 4.18: Nærbilde av figur 4.17.

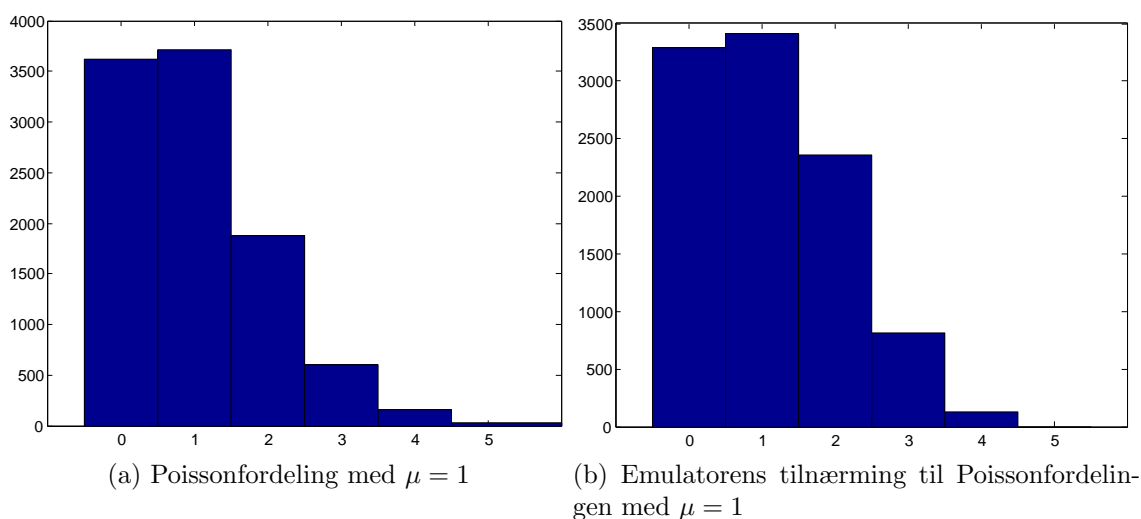
Poissonfordelte pseudo-tilfeldige tall.

Tabell 4.2: Kvalitetsmål på simuleringsresultatene av de emulerte støytypene

| Type støy           | PSNR (dB) | MSE      | MAE    |
|---------------------|-----------|----------|--------|
| Emulert fotonstøy   | 29.3189   | 76.0662  | 4.7570 |
| Emulert radstøy     | 31.0340   | 51.2487  | 4.5113 |
| Emulert kolonnestøy | 31.2444   | 48.8243  | 4.4129 |
| Alle                | 26.2118   | 155.5596 | 6.4108 |

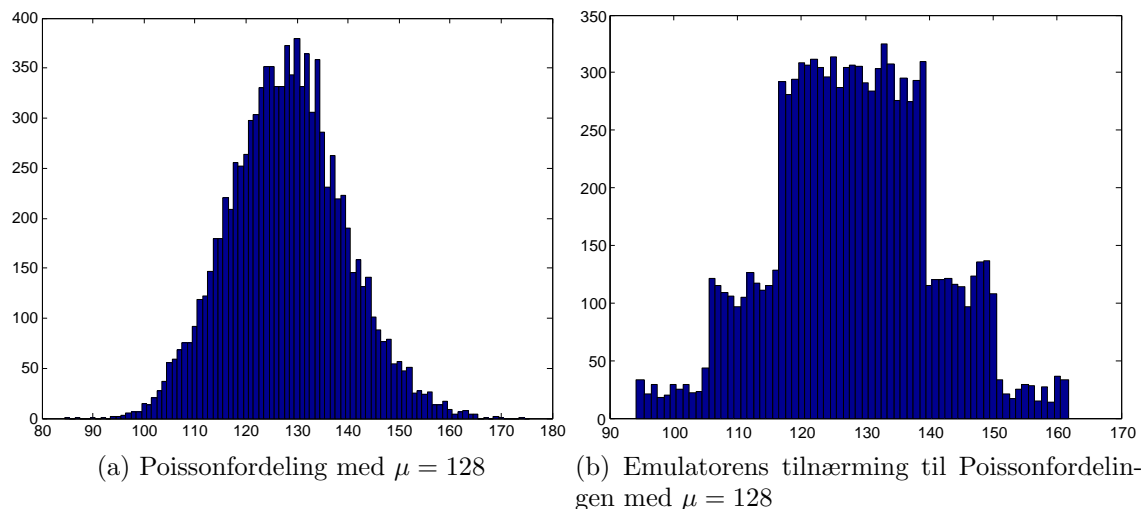
### 4.1.3 Fordelingsfunksjonene til algoritmene for emulering av foton-, rad- og kolonnestøy

Dette delkapittelet presenterer kort fordelingsfunksjonene til algoritmen som er brukt til å emulere fotonstøy og algoritmen som er brukt til å emulere rad- og kolonnestøy, og sammenligner disse med henholdsvis Poisson- og normalfordelingen. Fordelingsfunksjonen sier noe om hvor bra algoritmene emulerer foton-, rad- og kolonnestøy, og kan være med på å forklare hvorfor resultatene blir som de blir.

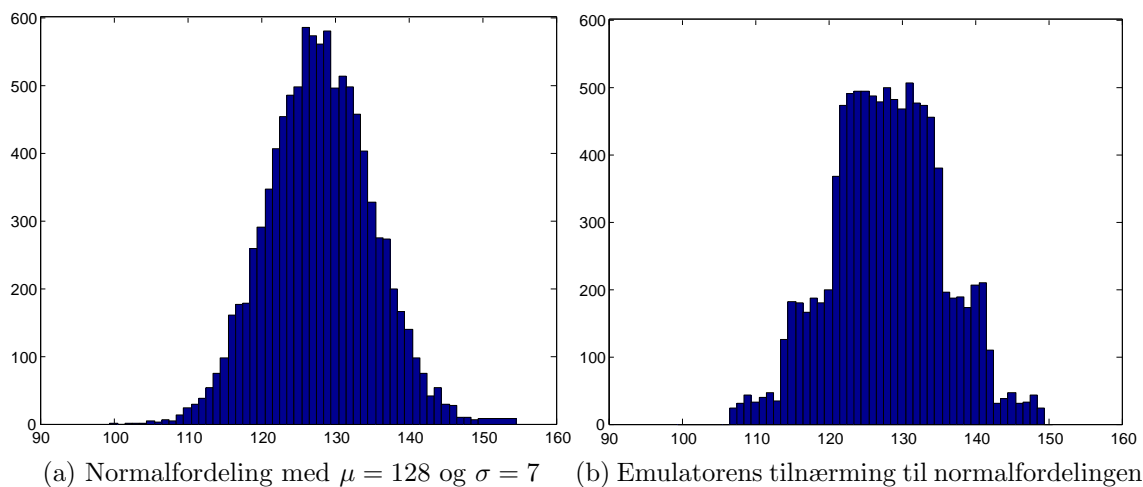


Figur 4.19: De 10000 første verdiene generert med Matlabs *poissrnd* og algoritmen som brukes til å generere pseudo-tilfeldige tall i fotonstøygeneratoren med  $\mu = 1$ .

Figur 4.19 og 4.20 viser en sammenligning av Poissonfordelingen og fordelingen til de pseudo-tilfeldige tallene som genereres i fotonstøygeneratoren. Som man ser har de pseudo-tilfeldige verdiene som genereres i emulatoren en noe kunstig fordeling. Sammenlignet med Poissonfordelingen er det flere verdier som er ulik forventningsverdien. Det vil dermed være færre pikselverdier som vil være uberørt av den emulerte fotonstøyen, enn det som er tilfellet i en reell bildebrikke. Det er i tillegg verdt å merke seg at alle verdiene til tallgeneratoren som brukes til å emulere fotonstøy ligger innenfor tre standardavvik.



Figur 4.20: De 10000 første verdiene generert med Matlabs *poissrnd* og algoritmen som brukes til å generere pseudo-tilfeldige tall i fotonstøygeneratoren med  $\mu = 128$ .

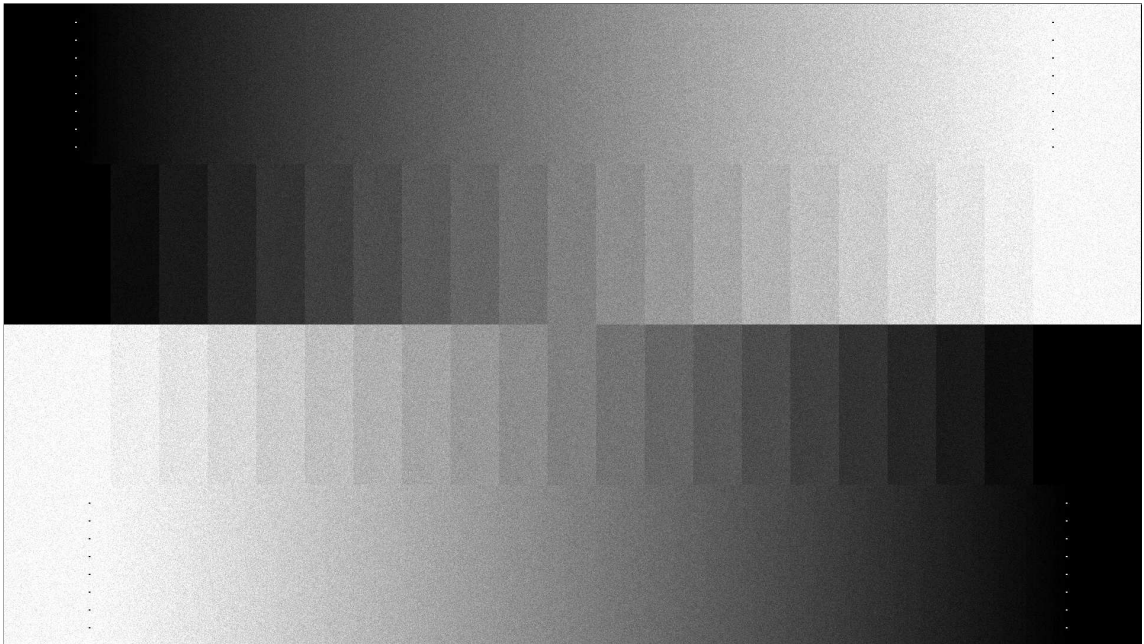


Figur 4.21: De 10000 første verdiene generert med Matlabs *randn* og algoritmen som brukes til å generere pseudo-tilfeldige tall i rad- og kolonnestøygeneratorene med forventningsverdi 128 og standardavvik lik 7.

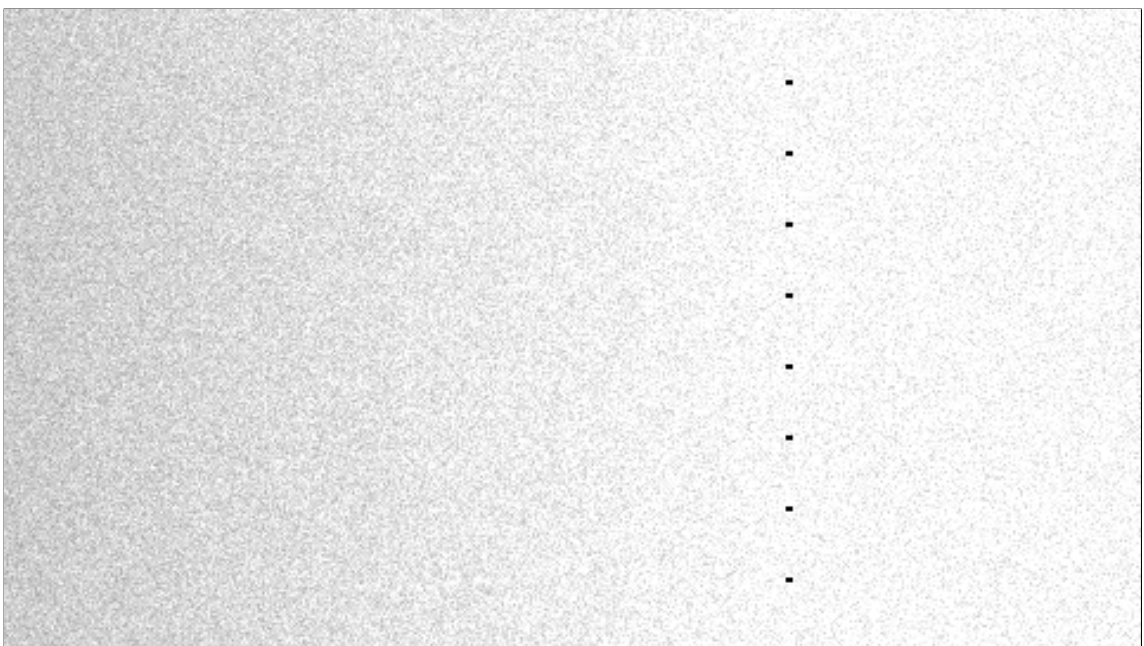
Figur 4.21 viser en sammenligning av Matlabs normalfordeling med en forventningsverdi lik 128 og standardavvik lik 7, og emulatorens tilnærming til den samme fordelingen. Det samme gjelder her som for emulatorens tilnærming til Poissonfordelingen, noe som er naturlig med tanke på at algoritmene som benyttes for å generere tallene i emulatoren er nesten helt like.

## 4.2 Resultater fra testsystem

Dette kapittelet presenterer resultatene fra testene som er utført på FPGAen. Se kapittel 3.4 for litt mer om hvordan systemet er satt opp og hvilke programmer som er brukt.



Figur 4.22: Resultatet fra test av emulering av fotonstøy på FPGAen.



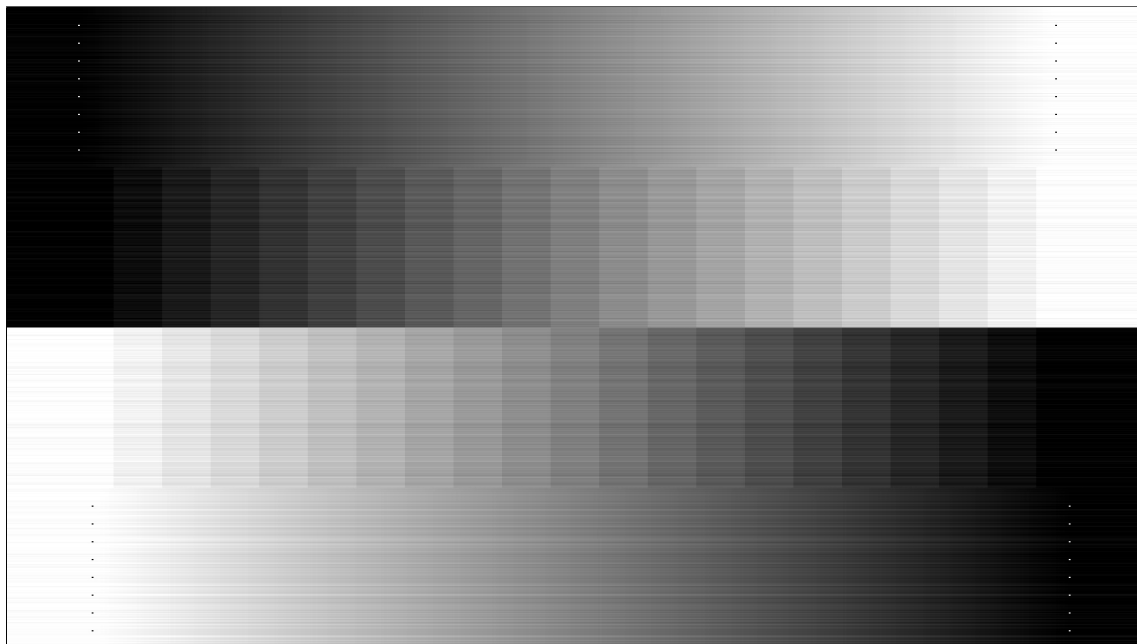
Figur 4.23: Nærbilde av figur 4.22.

Figur 4.22 og 4.23 viser resultatet av en test av emulatoren hvor den emulerer

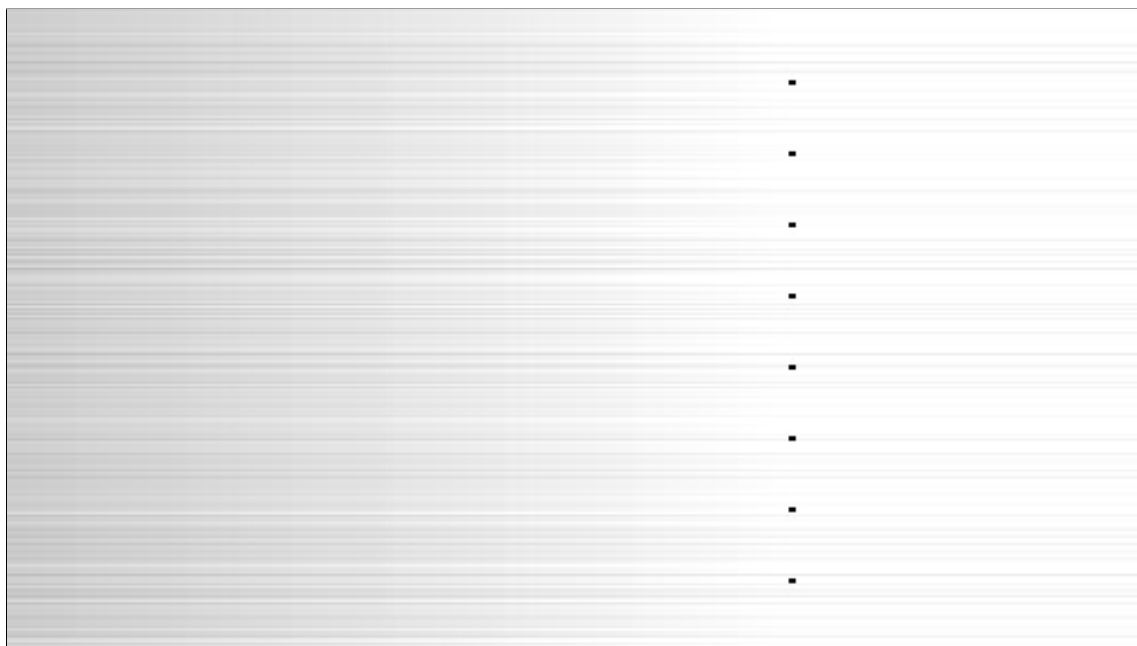
## KAPITTEL 4. RESULTATER

---

fotonstøy. Man ser at resultatet er omtrent som i figur 4.3 - 4.4 og 4.11 - 4.12, og det er vanskelig å skille disse figurene fra hverandre.



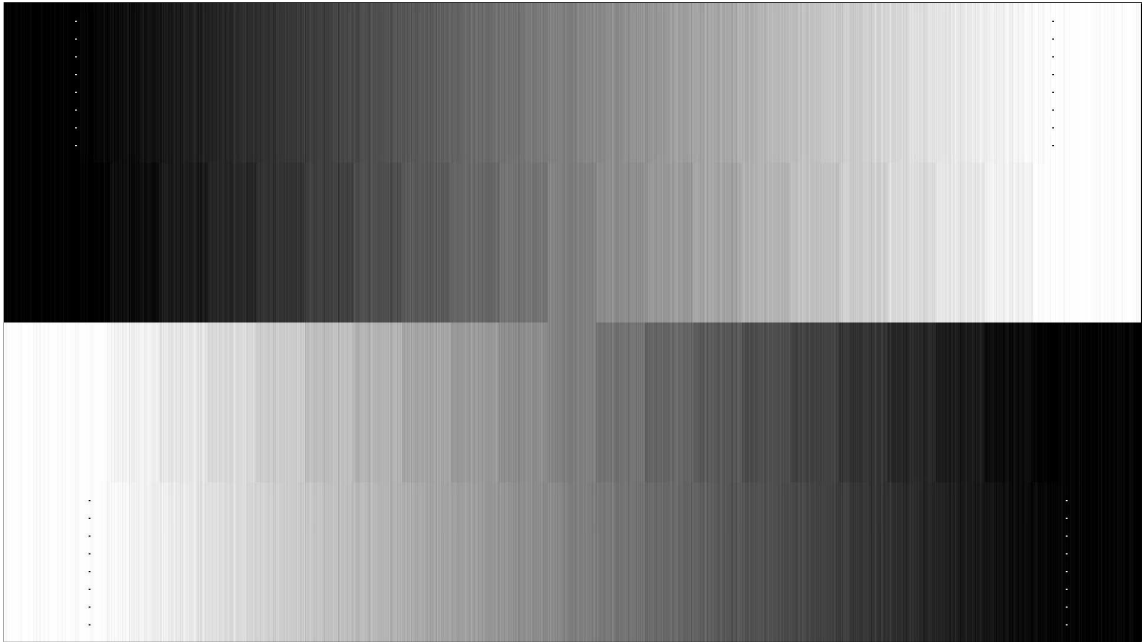
Figur 4.24: Resultatet fra test av emulering av radstøy på FPGAen.



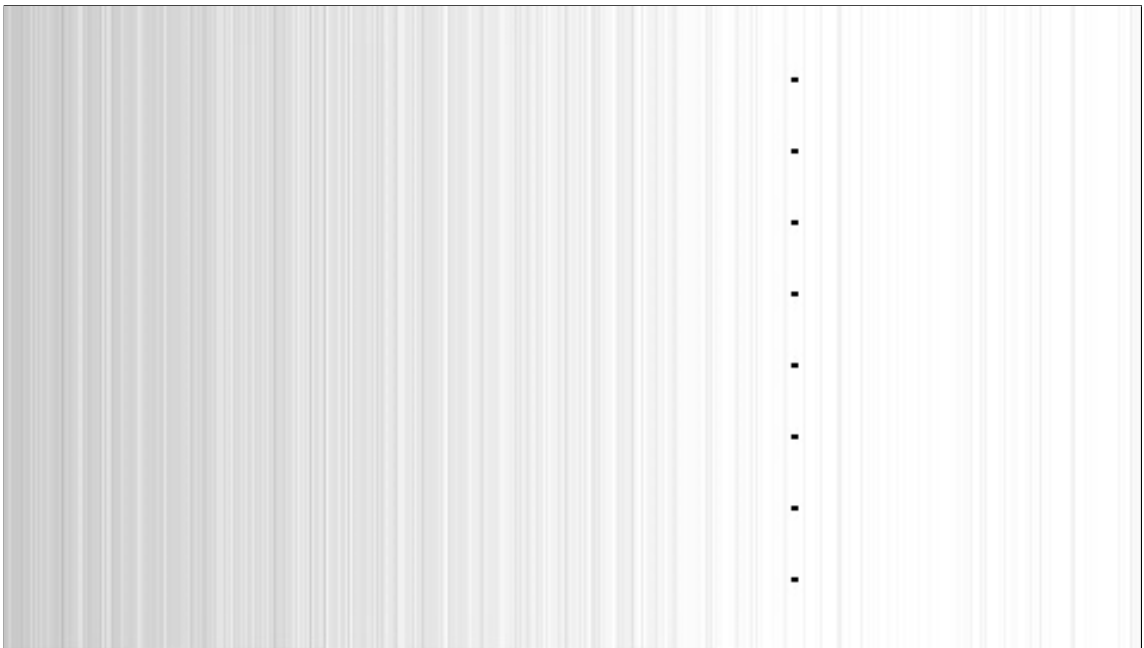
Figur 4.25: Nærbilde av figur 4.24.

Resultatet av testingen av emulering av radstøy ser man i figurene 4.24 og 4.25. I likhet med simuleringene av radstøy er standardavviket til radstøyen også her satt til 7 LSB. Som man ser er dette resultatet mer likt figurene 4.5 og 4.6 enn 4.13 og 4.14.





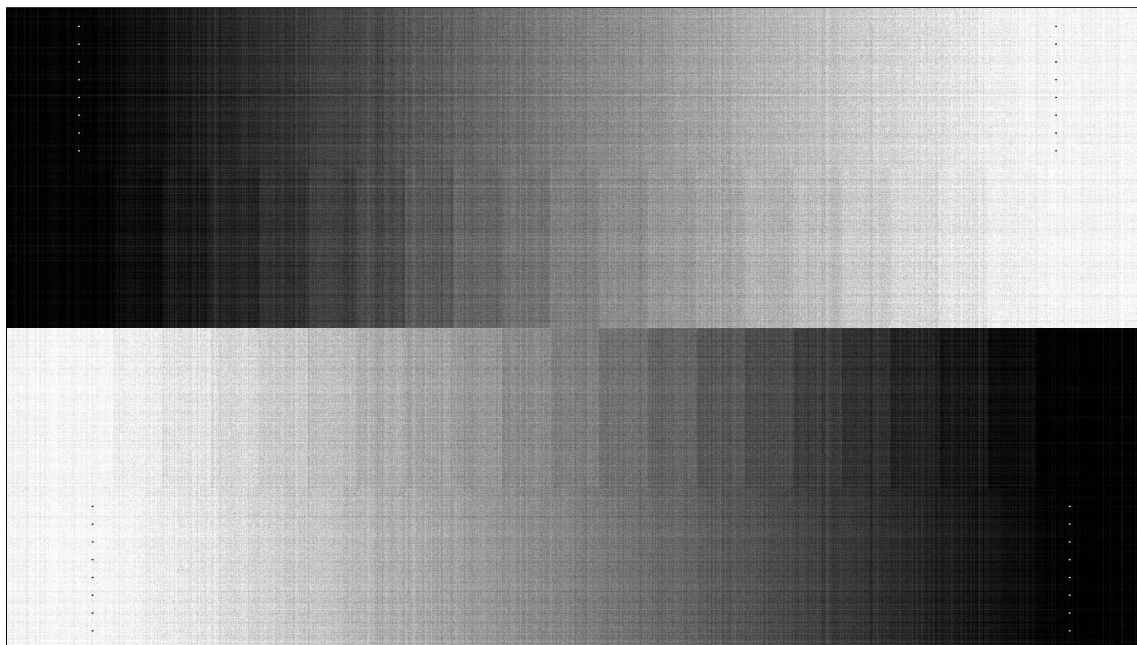
Figur 4.26: Resultat fra test av emulering av kolonnestøy på FPGAen.



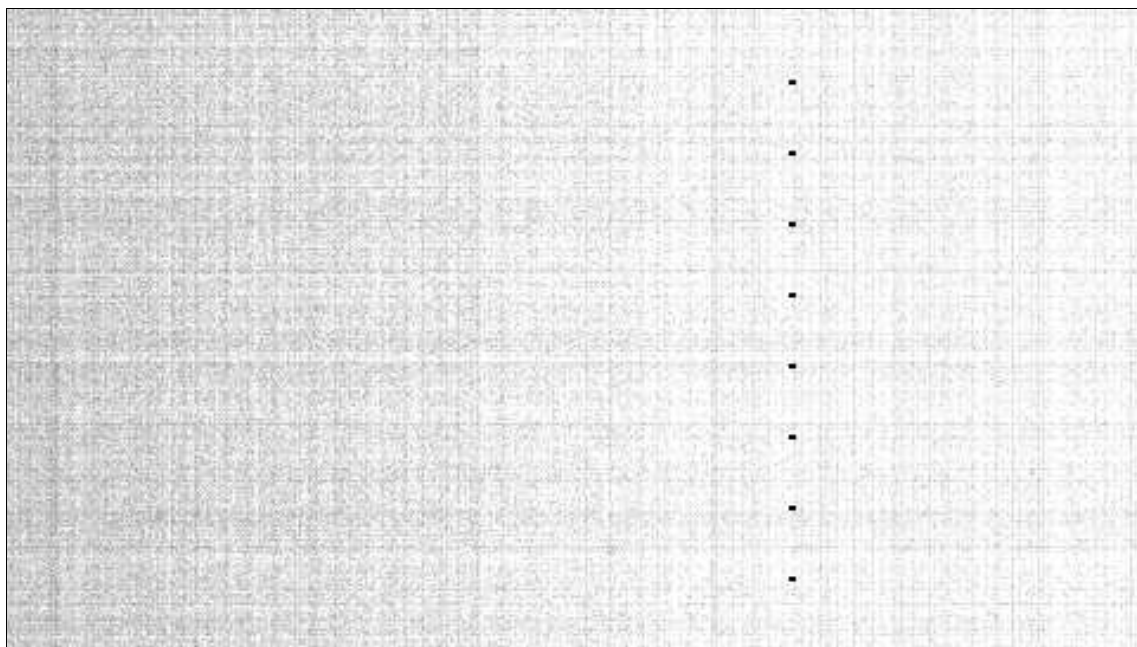
Figur 4.27: Nærbilde av figur 4.26.

I figurene 4.26 og 4.27 ser man resultatet av test av emulering av kolonnestøy. Også her er standardavviket til støyen satt til 7 LSB, og i likhet med resultatet av emulering av radstøy, ligner dette resultatet mest på simuleringene hvor Matlabfunksjonen *randn* ble brukt til å generere normalfordelt støy (se figur 4.7 og 4.8).

Figurene 4.28 og 4.29 viser resultatet fra testen hvor alle de implementerte støytypene ble emulert. Standardavviket til rad- og kolonnestøyen er som vanlig



Figur 4.28: Resultat fra test av alle de tre implementerte støytypene emulert på FPGAen.



Figur 4.29: Nærbilde av figur 4.28.

satt til 7 LSB. Sammenligner man disse figurene med figur 4.9 - 4.10 og 4.17 - 4.18, så ligner resultatet kanskje mest på figurene 4.9 og 4.10.

Tabell 4.3 viser en oversikt over kvalitetstallene til testresultatene. Sammenligner man denne med tabell 4.1 og 4.2 ser man at PSNRen ligger rundt den til de simuleringene utført med Matlabfunksjonene *randn* og *poissrnd* til å generere henholdsvis normal- og Poissonfordelt støy, og noe over de simuleringene av

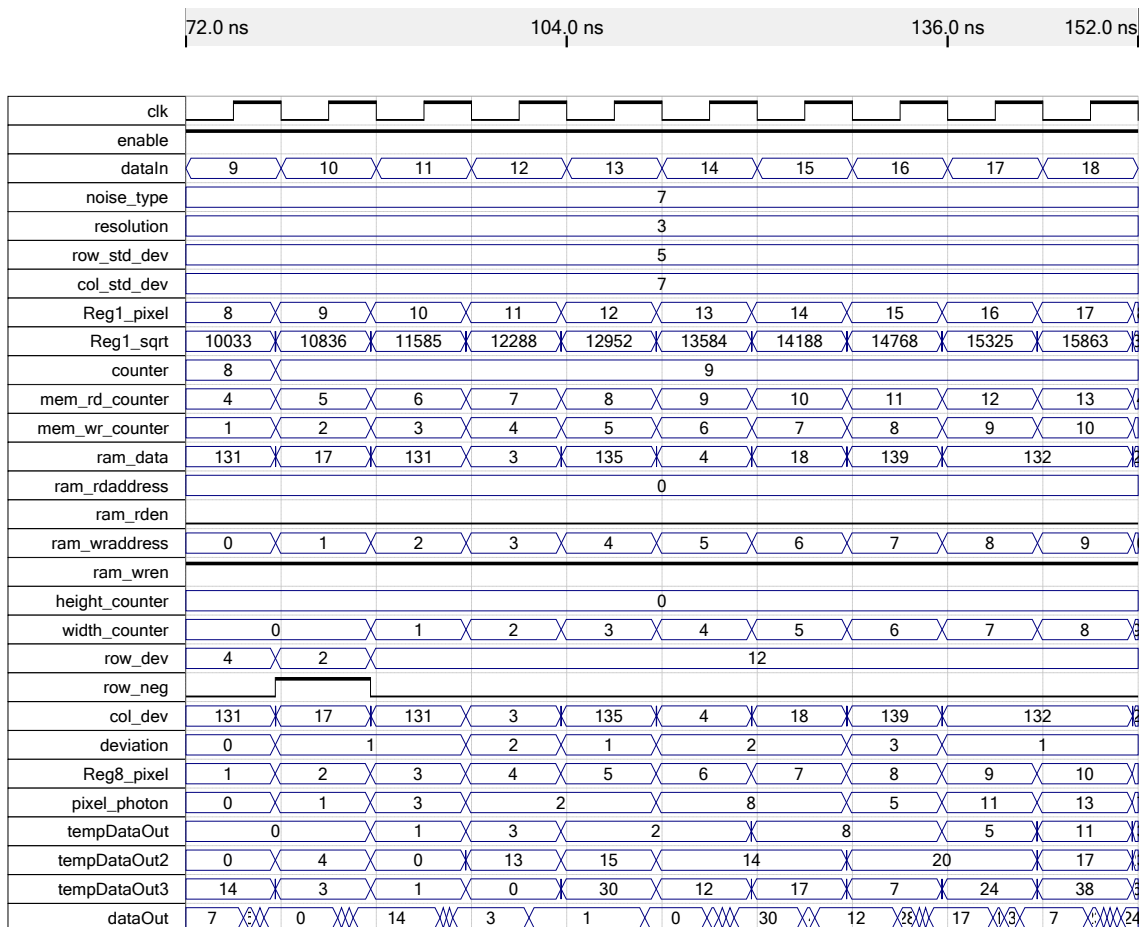
Tabell 4.3: Kvalitetsmål på resultatene fra test på FPGA

| Emulert støy | PSNR (dB) | MSE     | MAE    |
|--------------|-----------|---------|--------|
| Fotonstøy    | 30.6665   | 55.7736 | 3.7064 |
| Radstøy      | 34.9150   | 20.9691 | 2.3576 |
| Kolonnestøy  | 34.5557   | 22.7777 | 2.4749 |
| Alle         | 28.8158   | 85.4076 | 4.8183 |

algoritmene som brukes i emulatoren.

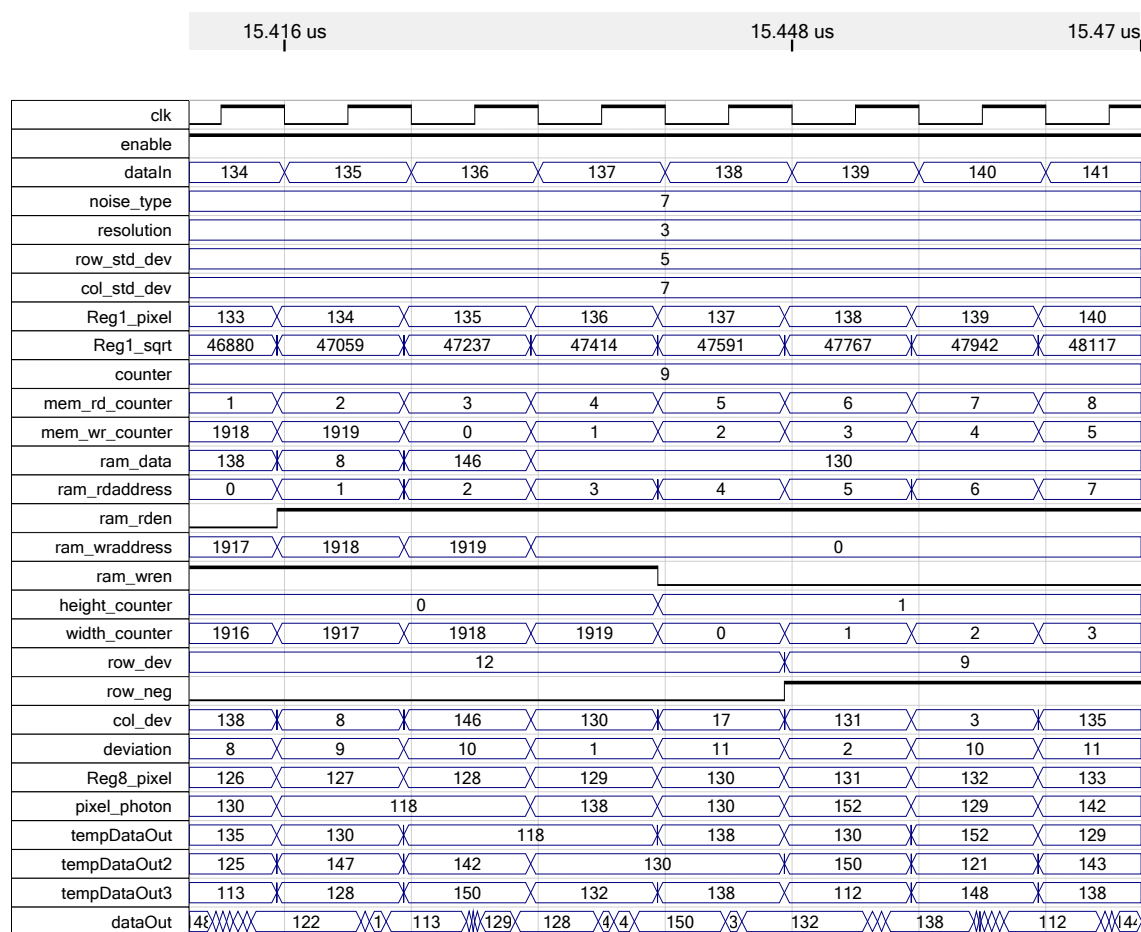
### 4.3 Quartus II-simulering

*Quartus II 9.1 sp1 Web Edition* ble brukt til å simulere systemet for å verifisere timing og funksjonalitet. Figurene 4.30 og 4.31 viser et utdrag av et bølgediagram som simulerer emuleringen av output fra en CMOS-bildebrikke med full HD-oppløsning som inneholder foton-, rad- og kolonnestøy. Kun de viktigste signalene og registerene er tatt med.



Figur 4.30: Simulering av emulatoren rett etter oppstart.

## KAPITTEL 4. RESULTATER



Figur 4.31: Simulering av emulatoren i det den er ferdig med første raden i et bilde.

Figur 4.30 viser en simulering av emulatoren rett etter oppstart. Til inngangssignalet *dataIn* er det brukt en 8-bits teller som startet på null og teller oppover. Man kan se at utgangssignalet *dataOut* skifter verdi hele veien, men utgangsverdien 30 er verdien til den første pikselen. Ved å se på *dataIn* ser man at det tar 14 klokkesykler fra den første pikselverdien ligger på inngangen av emulatoren, til den er overlappet med støy og er klar på utgangen.

Registeret *Reg8\_pixel* inneholder pikselverdier uten støy. Neste klokkesykel vil denne verdien overlappes med fotonstøy og lagres i registeret *pixel\_photon*, samtidig som pikselverdien uten støy lagres i registeret *Reg9\_pixel* (dette registeret er ikke tatt med i simuleringen). Dersom inngangssignalet *noise\_type[0]* er lik 1 vil registeret *tempDataOut* få verdien til *pixel\_photon*, og i motsatt tilfellet vil det få verdien som er lagret i registeret *Reg9\_pixel*. Ser vi på figur 4.30 ser vi at verdien 1 er lagret i registeret *Reg8\_pixel* etter 72 ns. Neste klokkesykel ser vi at registeret *pixel\_photon* også får verdien 1. Den emulerte fotonstøyen var altså mindre enn en halv LSB i dette tilfellet og dermed ikke synlig. Den neste klokkesykelen forplantes verdien i *pixel\_photon* til registeret *tempDataOut*. Deretter overlappes den med radstøy (så lenge *noise\_type[1] = 1*), og lagres i registeret *tempDataOut2*. Som man ser av figurene 4.30 og 4.31 fikk den første raden et positivt avvik på 12 LSB. Til slutt legges det til et kolonneavvik og resultatet lagres i registeret *tempDataOut3* før det

sendes ut på *DataOut* neste klokkesykel. Figuren viser at den første kolonnen fikk et positivt avvik på 17 LSB og hvordan dette resultatet lagres i RAM.

Figur 4.31 viser hvordan systemet oppfører seg i det emulatoren er ferdig med en rad, og begynner på den neste. Som man ser av figuren begynner kolonnestøygeneratoren å lese fra RAMen før den er ferdig å skrive til den. Figuren viser også at det beregnes et nytt radavvik, som denne gangen er negativt (vist ved at signalet *row\_neg* går høyt). I tillegg kan man observere at den første pikselen på rad nummer 2, har samme kolonneavvik som den første pikselen på den første raden, nemlig 17 LSB.

## 4.4 Kompileringsresultater fra Quartus II

Dette delkapittelet presenterer de viktigste kompileringsresultatene fra programmet *Quartus II 9.1 sp 1 Web Edition*.

Tabell 4.4: Maksfrekvensen til emulatoren

| Modell             | Maksfrekvens |
|--------------------|--------------|
| Slow 1200 mV 85 °C | 124.81 MHz   |
| Slow 1200 mV 0 °C  | 137.02 MHz   |

Tabell 4.4 viser maksfrekvensen til emulatoren, og som resultatene viser er emulatoren i stand til å behandle 60 full HD-bilder i sekundet (se (3.4)). En oppsummering av kompileringsresultatene er vist i tabell 4.5.

Tabell 4.5: Oppsummering av kompileringsresultatene

| Enhet                         | Antall |
|-------------------------------|--------|
| Totalt antall logiske enheter | 2491   |
| Kombinatoriske funksjoner     | 2326   |
| Dedikerte logiske registre    | 1251   |
| Totalt antall registre        | 1251   |
| Pinner                        | 34     |
| Minnebit                      | 20504  |



# Kapittel 5

## Diskusjon

Det er designet en emulator som emulerer output fra en A/D-omformer i en CMOS-bildebrikke, hvor fokuset har vært på funksjonalitet og ytelse. Systemet lar brukeren bestemme oppløsning, og støytyper som skal emuleres. Tilgjengelige oppløsninger er VGA ( $640 \times 480$ ), HD ( $1280 \times 720$ ) og full HD ( $1920 \times 1080$ ), og støytypene som er implementert er foton-, rad- og kolonnestøy. Hastigheten på systemet bestemmes av frekvensen på klokken systemet tar inn, og maksfrekvensen til emulatoren er 124.81 MHz.

Simuleringene av algoritmene emulatoren benytter for å emulere foton-, rad- og kolonnestøy viste at PSNRen til disse var noe lavere enn PSNRen til simuleringene av de samme støytypene hvor Matlabfunksjonene *randn* og *poissrnd* ble brukt til å generere henholdsvis normal- og Poissonfordelte tall. Dette kan best forklares ved å se på fordelingsfunksjonene vist i figur 4.20 og 4.21. Som man ser av disse histogrammene vil algoritmene som emulatoren bruker gi flere piksler med avvik enn ved å benytte virkelige normal- og Poissonfordelte tall. Så selv om maksavviket er større i en virkelig normal- eller Poissonfordeling enn de fordelingene til algoritmene emulatoren benytter, så blir summen av avvikene mindre. Dette gjenspeiles også i kvalitetstallet MAE for de forskjellige simuleringene.

Resultatene fra testene på FPGAen viste seg å ligne mer på simuleringene med Matlabs måte å generere normal- og Poissonfordelte tall, enn simuleringene av algoritmene som emulatoren bruker. Forskjellen på simuleringene av algoritmene til emulatoren og testingen av emulatoren på FPGA er at simuleringene benytter de  $1920 \times 1080$  første tallene fra Tausworthe-generatoren, mens testen på FPGAen plukker ut  $1920 \times 1080$  tall fra forskjellige plasser i rekken, avhengig av når CPUen er klar til å lagre resultatet og sende en ny pikselverdi. Dette viser at PSNR og MAE vil variere noe fra bilde til bilde selv med samme input, og annet kan ikke forventes når det brukes pseudo-tilfeldige tall.

Testene og simuleringene viser at emulatoren vil emulere støy som ligner visuelt på foton-, rad- og kolonnestøy. Simuleringen av algoritmene som brukes til å generere noe som minner om normal- og Poissonfordelte tall, viser at fordelingen til støyen er noe kunstig, og dette kan forårsake kunstige artifakter i bildet.

Som sagt har fokuset under designprosessen vært på funksjonalitet og ytelse. At systemet har en ventetid på 14 klokkesykler fra det påtrykkes en pikselverdi til den er overlagret med støy har lite å si, det viktigste er at systemet kan kjøre på

frekvenser opp til 124.81 MHz som gjør at systemet kan behandle 60 full HD-bilder i sekundet. De fleste klokkesyklene systemet bruker er nødvendig for å oppnå ønsket gjennomstrømning, men det er mulig å minske ventetiden noe.

Det kan også gjøres enkle grep for å minske effektforbruket. Mesteparten av prosessene som genererer foton-, rad- og kolonnestøy kjører uavhengig av støytypene som er valgt og hvilken piksel, det vil si posisjon i bildet, som behandles. Ved å kun kjøre de prosessene som trengs til enhver tid kan man spare mye effekt, men effektforbruket har aldri vært noe tema under arbeidet med denne oppgaven.

Kompileringsresultatene viser at systemet har et relativt beskjedent arealbruk. Systemet bruker for eksempel mindre enn 2% av de logiske enhetene som er tilgjengelig på *Cyclone III FPGA Development Kit*. Systemet kan dermed implementeres på de fleste FPGAene som er tilgjengelig i dag, selv om det ikke er gjort noen grep for å minske arealet. Men ved å benytte en FPGA med nok minne til å lagre testbildet internt, og nok PLLer til å implementere de ønskede frekvensene kan hele systemet implementeres på FPGAen, hvor man kun trenger å tilføre en referanseklokke til PLLene.

Vanligvis er det lagt inn en offset i bildebrikkene, på for eksempel 100 LSB, for å hindre at støyfordelingen ved laveste belysning blir feil, som følge av at pikselverdien ikke kan bli mindre enn null. For å sørge for at støyfordelingen blir riktig for de høyeste lysintensitetene øker man bit-bredden. Dette er ikke implementert i emulatoren, men det skulle ikke by på særlig store problemer å gjøre det.



# Kapittel 6

## Konklusjon

Denne rapporten tar for seg utviklingen av et system som emulerer output fra en A/D-omformer i en CMOS-bildebrikke på *Cyclone III FPGA Development Kit*. Systemet lar brukeren velge mellom oppløsningene VGA ( $640 \times 480$ ), HD ( $1280 \times 720$ ) og full HD ( $1920 \times 1080$ ), og støytypene fotonstøy, radstøy og kolonnestøy. SNRen er gjort programmerbar ved at brukeren velger standardavvikene til rad- og kolonnestøyen ved oppstart, dersom disse støytypene er valgt. Emulatoren takler frekvenser opp mot 124.81 MHz, noe som gjør at den er i stand til å behandle 60 full HD-bilder i sekundet.

Simuleringene og testene viser at emulatoren gir et resultat som er visuelt likt reell foton-, rad- og kolonnestøy, men fordelingen til emulatorens tilnærming til Poisson- og normalfordeling er noe kunstig og kan gi uventede artifakter i bildet. PSNR og MAE til bildene ut av emulatoren vil variere noe, men ligge i forventet område.

Videre arbeid kan bestå av å implementere en Gaussisk Random Number Generator (RNG) og en Poisson RNG for å få støyfordelinger som er mer lik de som observeres i CMOS-bildebrikker. Andre ting som kan implementeres er flere støytyper, en offset for å hindre feil støyfordeling ved lav lysintensitet og større bit-bredde for å hindre feil støyfordeling ved høy lysintensitet.



# Bibliografi

- [1] X. Liu, “CMOS Image Sensors Dynamic Range and SNR Enhancement via Statistical Signal Processing,” Ph.D. dissertation, Stanford University, 2002.
- [2] C. Mead, “A Sensitive Electronic Photoreceptor,” in *1985 Chapel Hill Conference on VLSI*, Chapel Hill, NC, 1985.
- [3] P. B. Andersen, “Fotodiode,” *Store Norske Leksikon*, 2009. [Online]. Available: <http://www.snl.no/fotodiode>
- [4] S. M. Sze, *Semiconductor Devices: Physics and Technology*. Wiley, 1985.
- [5] W. S. Boyle and G. E. Smith, “Charge-coupled semiconductor devices,” *Bell Systems Technical Journal*, vol. 49, no. 1, pp. 587–593, 1970.
- [6] J. Nakamura, *Image Sensors and Signal Processing for Digital Still Cameras*. Boca Raton: Taylor & Francis, 2006.
- [7] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability & Statistics for Engineers & Scientists*, 8th ed. Upper Saddle River: Pearson Education, Inc., 2007.
- [8] H. Nyquist, “Thermal Agitation of Electric Charge in Conductors,” *Phys. Rev.*, vol. 32, no. 1, pp. 110–113, Jul 1928.
- [9] J. B. Johnson, “Thermal Agitation of Electricity in Conductors,” *Phys. Rev.*, vol. 32, no. 1, p. 97, Jul 1928.
- [10] T. Hansen, “Haglstøy,” *Store Norske Leksikon*. [Online]. Available: <http://www.snl.no/haglstoy>
- [11] M. Bigas, E. Cabruja, J. Forest, and J. Salvi, “Review of CMOS image sensors,” *Microelectronics Journal*, vol. 37, no. 5, pp. 433 – 451, 2006. [Online]. Available: <http://www.sciencedirect.com/science/article/B6V44-4H21R3N-1/2/348a7b2ee3dfd1f4fcc6783d808fb7cb>
- [12] S. Kempainen, “CMOS image sensors: Eclipsing CCDs in visual information?” *EDN*, vol. 42, no. 21, pp. 101–102, 1997.
- [13] T. Suzuki, “Challenges of Image-Sensor Development,” in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 7-11 2010, pp. 27 –30.

## BIBLIOGRAFI

---

- [14] H. Tian, B. Fowler, and A. E. Gamal, “Analysis of Temporal Noise in CMOS Photodiode Active Pixel Sensor,” *IEEE Journal of Solid-State Circuits*, vol. 36, no. 1, pp. 92–101, 2001.
- [15] H.-P. C. Group, “Noise Sources in CMOS Image Sensors,” 1998. [Online]. Available: [http://www.stw.tu-ilmenau.de/~ff/beruf\\$\\_\\$cc/cmos/cmos\\$\\_\\$noise.pdf](http://www.stw.tu-ilmenau.de/~ff/beruf$_$cc/cmos/cmos$_$noise.pdf)
- [16] A. J. Theuwissen, “CMOS Image Sensors: State-of-the-art,” *Solid-State Electronics*, vol. 52, no. 9, pp. 1401 – 1406, 2008, papers Selected from the 37th European Solid-State Device Research Conference - ESSDERC’07. [Online]. Available: <http://www.sciencedirect.com/science/article/B6TY5-4SJR2HN-1/2/78b043cab23cdae1370392a63576d363>
- [17] A. J. P. Theuwissen, *Solid-State Imaging with Charge-Coupled Devices*. Dordrecht: Kluwer Academic Publishers, 1995.
- [18] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, “Image quality assessment: from error visibility to structural similarity,” *Image Processing, IEEE Transactions on*, vol. 13, no. 4, pp. 600 –612, april 2004.
- [19] G. Zhang and P. H. W. Leong, “Ziggurat-based hardware gaussian random number generator,” in *IEEE International Conf. Field-Programmable Logic and its Applications*, 2005, pp. 275–280.
- [20] P. L’Ecuyer, “Maximally equidistributed combined tausworthe generators,” *Math. Comput.*, vol. 65, no. 213, pp. 203–213, 1996.

# Tillegg A

## Verilogkoden til emulatoren

```
1 module system (
2
3   /* Input */
4   clk_50 ,
5   clk_125 ,
6   resetn ,
7   dataIn ,
8   switch ,
9
10  /* Output */
11  state_led ,
12  status_led ,
13  dataOut
14 );
15
16 input      clk_50 ;
17 input      clk_125 ;
18 input      resetn ;
19 input [ 7 : 0 ] dataIn ;
20 input [ 7 : 0 ] switch ;
21
22 output [ 2 : 0 ] state_led ;
23 output [ 4 : 0 ] status_led ;
24 output [ 7 : 0 ] dataOut ;
25
26 reg [2:0] state_led ;
27 reg [4:0] status_led ;
28
29 parameter SIZE = 3 ;
30 parameter S0 = 3'd0 ;
31 parameter S1 = 3'd1 ;
32 parameter S2 = 3'd2 ;
33 parameter S3 = 3'd3 ;
34 parameter S4 = 3'd4 ;
35
36 reg [SIZE-1:0] state ;
37 reg [SIZE-1:0] next_state ;
38
39 reg [2:0] noise_type ;
40 reg [1:0] resolution ;
```

```

41 reg [5:0] row_std_dev;
42 reg [5:0] col_std_dev;
43 reg enable;
44
45 noise_generator noise_generator_instance(
46     .enable (enable),
47     .clk (clk_125),
48     .dataIn (dataIn),
49     .noise_type (noise_type),
50     .resolution (resolution),
51     .row_std_dev (row_std_dev),
52     .col_std_dev (col_std_dev),
53     .dataOut (dataOut)
54 );
55
56 always @ (switch, state, noise_type)
57 begin
58     case (state)
59         S0: //Velger oppløsning
60             begin
61                 if (switch[7] == 1'b1) begin
62                     next_state <= S1;
63                 end else begin
64                     next_state <= S0;
65                 end
66             end
67         S1: //Velger støytype
68             begin
69                 if (switch[7] == 1'b0) begin
70                     if (noise_type[2] == 1'b1) begin
71                         next_state <= S2;
72                     end else if (noise_type[1] == 1'b1) begin
73                         next_state <= S3;
74                     end else begin
75                         next_state <= S4;
76                     end
77                 end else begin
78                     next_state <= S1;
79                 end
80             end
81         S2: //radstøy
82             begin
83                 if (switch[6] == 1'b1) begin
84                     if (noise_type[1] == 1'b1) begin
85                         next_state <= S3;
86                     end else begin
87                         next_state <= S4;
88                     end
89                 end else begin
90                     next_state <= S2;
91                 end
92             end
93         S3: //kolonnestøy
94             begin
95                 if (switch[6] == 1'b0) begin

```

```

96         next_state <= S4;
97     end else begin
98         next_state <= S3;
99     end
100 end
101 S4:
102     begin
103         next_state <= S4;
104     end
105 endcase
106 end
107
108 always @ (posedge clk_50 or negedge resetn)
109 begin
110     if (!resetn) begin
111         state <= S0;
112     end else begin
113         state <= next_state;
114     end
115 end
116
117 always @ (posedge clk_50)
118 begin
119     case (state)
120     S0:
121         begin
122             resolution <= switch[1:0];
123             status_led <= {3'b111, ~(resolution)};
124             state_led <= 3'b110;
125             enable <= 1'b0;
126         end
127     S1:
128         begin
129             noise_type <= switch[2:0];
130             status_led <= {2'b11, ~(noise_type)};
131             state_led <= 3'b101;
132             enable <= 1'b0;
133         end
134     S2:
135         begin
136             col_std_dev <= switch[5:0];
137             status_led <= ~col_std_dev[4:0];
138             state_led <= 3'b100;
139             enable <= 1'b0;
140         end
141     S3:
142         begin
143             row_std_dev <= switch[5:0];
144             status_led <= ~row_std_dev[4:0];
145             state_led <= 3'b011;
146             enable <= 1'b0;
147         end
148     S4:
149         begin
150             status_led <= 5'b11111;

```

```

151         state_led <= 3'b010;
152         enable <= 1'b1;
153     end
154     default:
155         begin
156             state_led <= 3'b000;
157             status_led <= 5'b11111;
158         end
159     endcase
160 end
161
162 endmodule

```

```

1 module noise_generator (
2     /* inputs */
3     enable,
4     clk,
5     dataIn,
6     noise_type,
7     resolution,
8     row_std_dev,
9     col_std_dev,
10
11     /* outputs */
12     dataOut
13 );
14
15 input    enable;
16 input    clk;
17 input    [ 7 : 0 ] dataIn;
18 input    [ 2 : 0 ] noise_type;
19 input    [ 1 : 0 ] resolution;
20 input    [ 5 : 0 ] row_std_dev;
21 input    [ 5 : 0 ] col_std_dev;
22
23 output    [ 7 : 0 ] dataOut;
24
25 wire    [ 7 : 0 ] dataIn;
26
27 reg      [ 7 : 0 ] dataOut;
28
29 reg [7:0] tempDataOut;
30 reg [7:0] tempDataOut2;
31 reg [7:0] tempDataOut3;
32
33 reg [7:0] ram_data;
34 reg ram_wren;
35 reg ram_rden;
36 reg [10:0] ram_wraddress;
37 reg [10:0] ram_rdaddress;
38 wire [7:0] ram_q;
39
40 wire [15:0] rom_q;
41
42 reg [7:0] Reg1_pixel;
43 reg [7:0] Reg2_pixel;

```



```

44 reg [7:0] Reg3_pixel;
45 reg [7:0] Reg4_pixel;
46 reg [7:0] Reg5_pixel;
47 reg [7:0] Reg6_pixel;
48 reg [7:0] Reg7_pixel;
49 reg [7:0] Reg8_pixel;
50 reg [7:0] Reg9_pixel;
51 reg [15:0] Reg1_sqrt;
52 reg [15:0] Reg2_sqrt;
53 reg [15:0] Reg3_sqrt;
54 reg [15:0] Reg4_sqrt;
55 reg [15:0] Reg5_sqrt;
56 reg [15:0] Reg6_sqrt;
57
58 reg [7:0] pixel_photon;
59
60 reg [7:0] temp;
61
62 reg [7:0] counter;
63 reg [10:0] mem_rd_counter;
64 reg [10:0] mem_wr_counter;
65
66 reg [7:0] deviation;
67 reg [7:0] row_dev;
68 reg [39:0] row_dev_pro;
69 reg [35:0] row_sum1;
70 reg [35:0] row_sum2;
71 reg [32:0] x1;
72 reg [32:0] x2;
73 reg [32:0] x3;
74 reg [32:0] x4;
75 reg row_neg;
76
77 reg [7:0] col_dev;
78 reg [7:0] col_dev_pipe;
79 reg [7:0] col_dev_pipe2;
80 reg [39:0] col_dev_pro;
81 reg [35:0] col_sum1;
82 reg [35:0] col_sum2;
83 reg [15:0] col_buffer;
84 reg [32:0] z1;
85 reg [32:0] z2;
86 reg [32:0] z3;
87 reg [32:0] z4;
88
89 reg [31:0] rnd;
90
91 reg [47:0] produkt;
92 reg [39:0] part_produkt1;
93 reg [39:0] part_produkt2;
94 reg [35:0] part1;
95 reg [35:0] part2;
96 reg [35:0] part3;
97 reg [35:0] part4;
98 reg [33:0] y1;

```

```

99 reg [33:0] y2;
100 reg [33:0] y3;
101 reg [33:0] y4;
102 reg [33:0] y5;
103 reg [33:0] y6;
104 reg [33:0] y7;
105 reg [33:0] y8;
106
107 reg [31:0] s1;
108 reg [31:0] s2;
109 reg [31:0] s3;
110
111 reg [10:0] width_counter;
112 reg [10:0] height_counter;
113 reg [10:0] width;
114 reg [10:0] height;
115
116 reg [7:0] row_dev_reg;
117 reg [7:0] col_dev_reg;
118
119 parameter [31:0] number1 = 32'hFFFFFFFE;
120 parameter [31:0] number2 = 32'hFFFFFFF8;
121 parameter [31:0] number3 = 32'hFFFFFFF0;
122
123 parameter [31:0] sigma = 32'hAE147AE1;
124 parameter [31:0] sigma2 = 32'hF3333333;
125
126 test_rom rom_instance(
127     .clock (clk),
128     .resetn (enable),
129     .address (dataIn),
130     .q (rom-q)
131 );
132
133 test_ram_dp ram_instance(
134     .data (ram_data),
135     .wraddress (ram_wraddress),
136     .wren (ram_wren),
137     .rdaddress (ram_rdaddress),
138     .rden (ram_rden),
139     .clock (clk),
140     .resetn (enable),
141     .q (ram-q)
142 );
143
144 //Tausworthe pseudo random generator
145 always @ (negedge enable or posedge clk)
146 begin
147     if (!enable) begin
148         s1 = 32'hFFFFFF;
149         s2 = 32'hCCCCCC;
150         s3 = 32'hFF00FF;
151     end else begin
152         s1 = (((s1<<13)^s1)>>19)^((s1&number1)<<12);
153         s2 = (((s2<<2)^s2)>>25)^((s2&number2)<<4);

```

```

154     s3 = (((s3<<3)^s3)>>11)^((s3&number3)<<17);
155     rnd = s1^s2^s3;
156 end
157 end
158
159 //Sets the resolution and the upper limit for the counters
160 //height_counter and width_counter
161 always @ (resolution)
162 begin
163     case (resolution)
164         2'b00:
165             begin
166                 width = 11'd1920;
167                 height = 11'd1080;
168             end
169         2'b01:
170             begin
171                 width = 11'd640;
172                 height = 11'd480;
173             end
174         2'b10:
175             begin
176                 width = 11'd1280;
177                 height = 11'd720;
178             end
179         2'b11:
180             begin
181                 width = 11'd1920;
182                 height = 11'd1080;
183             end
184     endcase
185 end
186
187 always @ (row_std_dev)
188 begin
189     row_dev_reg <= {2'b00, row_std_dev};
190 end
191
192 always @ (col_std_dev)
193 begin
194     col_dev_reg <= {2'b00, col_std_dev};
195 end
196
197 //Counters
198 always @ (posedge clk or negedge enable)
199 begin
200     if (!enable) begin
201         counter = 8'd0;
202     end else begin
203         if (counter >= 9) begin
204             counter = counter;
205         end else begin
206             counter = counter + 8'd1;
207         end
208     end

```

```

209 end
210
211 always @ (posedge clk or negedge enable)
212 begin
213     if (!enable) begin
214         width_counter = 11'd0;
215         height_counter = 11'd0;
216     end else begin
217         if (counter >= 9) begin
218             if (width_counter >= width-1) begin
219                 width_counter = 11'd0;
220                 if (height_counter >= height-1) begin
221                     height_counter = 11'd0;
222                 end else begin
223                     height_counter = height_counter + 11'd1;
224                 end
225             end else begin
226                 width_counter = width_counter + 11'd1;
227             end
228         end else begin
229             height_counter = height_counter;
230             width_counter = width_counter;
231         end
232     end
233 end
234
235 always @ (posedge clk or negedge enable)
236 begin
237     if (!enable) begin
238         mem_wr_counter = 11'd0;
239     end else begin
240         if (counter >= 7) begin
241             if (mem_wr_counter == width-1) begin
242                 mem_wr_counter = 11'd0;
243             end else begin
244                 mem_wr_counter = mem_wr_counter + 11'd1;
245             end
246         end else begin
247             mem_wr_counter = mem_wr_counter;
248         end
249     end
250 end
251
252 always @ (posedge clk or negedge enable)
253 begin
254     if (!enable) begin
255         mem_rd_counter = 11'd0;
256     end else begin
257         if (counter >= 4) begin
258             if (mem_rd_counter == width-1) begin
259                 mem_rd_counter = 11'd0;
260             end else begin
261                 mem_rd_counter = mem_rd_counter + 11'd1;
262             end
263         end else begin

```

```

264     mem_rd_counter = mem_rd_counter;
265     end
266 end
267 end
268
269 //Photon Noise
270 always @ (negedge enable or posedge clk)
271 begin
272     if (!enable) begin
273         y1 <= 34'd0;
274         y2 <= 34'd0;
275         y3 <= 34'd0;
276         y4 <= 34'd0;
277         y5 <= 34'd0;
278         y6 <= 34'd0;
279         y7 <= 34'd0;
280         y8 <= 34'd0;
281         Reg1_pixel <= 8'd0;
282         Reg2_pixel <= 8'd0;
283         Reg3_pixel <= 8'd0;
284         Reg4_pixel <= 8'd0;
285         Reg5_pixel <= 8'd0;
286         Reg6_pixel <= 8'd0;
287         Reg7_pixel <= 8'd0;
288         Reg8_pixel <= 8'd0;
289         Reg9_pixel <= 8'd0;
290         Reg1_sqrt <= 16'd0;
291         Reg2_sqrt <= 16'd0;
292         Reg3_sqrt <= 16'd0;
293         Reg4_sqrt <= 16'd0;
294         Reg5_sqrt <= 16'd0;
295         Reg6_sqrt <= 16'd0;
296         part1 <= 36'd0;
297         part2 <= 36'd0;
298         part3 <= 36'd0;
299         part4 <= 36'd0;
300         part_produkt1 <= 40'd0;
301         part_produkt2 <= 40'd0;
302         produkt <= 48'd0;
303         deviation <= 8'd0;
304         temp <= 8'd0;
305         pixel_photon <= 8'd0;
306     end else begin
307         /*----- STAGE 1 -----*/
308         Reg1_pixel <= dataIn;
309         /*----- STAGE 2 -----*/
310         Reg2_pixel <= Reg1_pixel;
311         /*----- STAGE 3 -----*/
312         Reg1_sqrt <= rom_q;
313         Reg3_pixel <= Reg2_pixel;
314         /*----- STAGE 4 -----*/
315         y1 <=
316             (Reg1_sqrt[0] ? {1'd0, rnd} : 0) +
317             (Reg1_sqrt[1] ? {rnd, 1'd0} : 0);
318         y2 <=

```

```

319      (Reg1_sqrt[2] ? {1'd0, rnd} : 0) +
320      (Reg1_sqrt[3] ? {rnd, 1'd0} : 0);
321      y3 <=
322      (Reg1_sqrt[4] ? {1'd0, rnd} : 0) +
323      (Reg1_sqrt[5] ? {rnd, 1'd0} : 0);
324      y4 <=
325      (Reg1_sqrt[6] ? {1'd0, rnd} : 0) +
326      (Reg1_sqrt[7] ? {rnd, 1'd0} : 0);
327      y5 <=
328      (Reg1_sqrt[8] ? {1'd0, rnd} : 0) +
329      (Reg1_sqrt[9] ? {rnd, 1'd0} : 0);
330      y6 <=
331      (Reg1_sqrt[10] ? {1'd0, rnd} : 0) +
332      (Reg1_sqrt[11] ? {rnd, 1'd0} : 0);
333      y7 <=
334      (Reg1_sqrt[12] ? {1'd0, rnd} : 0) +
335      (Reg1_sqrt[13] ? {rnd, 1'd0} : 0);
336      y8 <=
337      (Reg1_sqrt[14] ? {1'd0, rnd} : 0) +
338      (Reg1_sqrt[15] ? {rnd, 1'd0} : 0);
339
340      Reg4_pixel <= Reg3_pixel;
341      Reg2_sqrt <= Reg1_sqrt;
342
343      /*----- STAGE 5 -----*/
344      part1 <= {2'd0, y1} + {y2, 2'd0};
345      part2 <= {2'd0, y3} + {y4, 2'd0};
346      part3 <= {2'd0, y5} + {y6, 2'd0};
347      part4 <= {2'd0, y7} + {y8, 2'd0};
348
349      Reg5_pixel <= Reg4_pixel;
350      Reg3_sqrt <= Reg2_sqrt;
351      /*----- STAGE 6 -----*/
352      part_produkt1 <= {4'd0, part1} + {part2, 4'd0};
353      part_produkt2 <= {4'd0, part3} + {part4, 4'd0};
354
355      Reg6_pixel <= Reg5_pixel;
356      Reg4_sqrt <= Reg3_sqrt;
357
358      /*----- STAGE 7 -----*/
359      produkt <= {8'd0, part_produkt1} + {part_produkt2, 8'd0};
360
361      Reg7_pixel <= Reg6_pixel;
362      Reg5_sqrt <= Reg4_sqrt;
363
364      /*----- STAGE 8 -----*/
365      if (produkt[35] == 1) begin //avrunding
366          deviation <= {4'b0000, produkt[47:44]} + 8'b00000001;
367      end else begin
368          deviation <= {4'b0000, produkt[47:44]};
369      end
370
371      Reg8_pixel <= Reg7_pixel;
372      Reg6_sqrt <= Reg5_sqrt;
373

```

```

374 /*----- STAGE 9 -----*/
375     if (s1 <= sigma) begin
376         if (rnd[0] == 1) begin
377             if (Reg8_pixel + deviation >= 255) begin
378                 pixel_photon <= 8'd255;
379             end else begin
380                 pixel_photon <= Reg8_pixel + deviation;
381             end
382         end else begin
383             if (deviation >= Reg8_pixel) begin
384                 pixel_photon <= 8'd0;
385             end else begin
386                 pixel_photon <= Reg8_pixel - deviation;
387             end
388         end
389     end else if (s1 <= sigma2) begin
390         temp = Reg6_sqrt[11] ? {4'd0, Reg6_sqrt[15:12]} + 8'd1 : {4'd0,
391             Reg6_sqrt[15:12]};
392         if (rnd[0] == 1) begin
393             if (Reg8_pixel + temp + deviation >= 255) begin
394                 pixel_photon <= 8'd255;
395             end else begin
396                 pixel_photon <= Reg8_pixel + temp + deviation;
397             end
398         end else begin
399             if (deviation + temp >= Reg8_pixel) begin
400                 pixel_photon <= 8'd0;
401             end else begin
402                 pixel_photon <= Reg8_pixel - temp - deviation;
403             end
404         end
405     end else begin
406         temp = Reg6_sqrt[10] ? {3'd0, Reg6_sqrt[15:11]} + 8'd1 : {3'd0,
407             Reg6_sqrt[15:11]};
408         if (rnd[0] == 1) begin
409             if (Reg8_pixel + temp + deviation >= 255) begin
410                 pixel_photon <= 8'd255;
411             end else begin
412                 pixel_photon <= Reg8_pixel + temp + deviation;
413             end
414         end else begin
415             if (deviation + temp >= Reg8_pixel) begin
416                 pixel_photon <= 8'd0;
417             end else begin
418                 pixel_photon <= Reg8_pixel - temp - deviation;
419             end
420         end
421     end
422     Reg9_pixel <= Reg8_pixel;
423 /*-----*/
424 end
425 end
426

```

```

427 //Row Noise
428 always @ (negedge enable or posedge clk)
429 begin
430     if (!enable) begin
431         x1 <= 33'd0;
432         x2 <= 33'd0;
433         x3 <= 33'd0;
434         x4 <= 33'd0;
435         row_sum1 <= 36'd0;
436         row_sum2 <= 36'd0;
437         row_dev_pro <= 40'd0;
438         row_dev <= 8'd0;
439         row_neg <= 1'b0;
440     end else begin
441         /*----- STAGE 1 -----*/
442
443         x1 <=
444             (row_dev_reg[0] ? {1'd0, s3} : 0) +
445             (row_dev_reg[1] ? {s3, 1'd0} : 0);
446         x2 <=
447             (row_dev_reg[2] ? {1'd0, s3} : 0) +
448             (row_dev_reg[3] ? {s3, 1'd0} : 0);
449         x3 <=
450             (row_dev_reg[4] ? {1'd0, s3} : 0) +
451             (row_dev_reg[5] ? {s3, 1'd0} : 0);
452         x4 <=
453             (row_dev_reg[6] ? {1'd0, s3} : 0) +
454             (row_dev_reg[7] ? {s3, 1'd0} : 0);
455
456         /*----- STAGE 2 -----*/
457
458         row_sum1 <= x1 + {x2, 2'd0};
459         row_sum2 <= x3 + {x4, 2'd0};
460
461         /*----- STAGE 3 -----*/
462
463         row_dev_pro <= row_sum1 + {row_sum2, 4'd0};
464
465         /*----- STAGE 4 -----*/
466
467         if (width_counter == 0) begin
468             if (s2 <= sigma) begin
469                 row_dev <= row_dev_pro[39:32];
470             end else if (s2 <= sigma2) begin
471                 row_dev <= row_dev_reg + row_dev_pro[39:32];
472             end else begin
473                 row_dev <= (row_dev_reg << 1) + row_dev_pro[39:32];
474             end
475
476             if (s2[0] == 1) begin
477                 row_neg <= 1'b1;
478             end else begin
479                 row_neg <= 1'b0;
480             end
481         end else begin

```



```

482         row_neg <= row_neg;
483         row_dev <= row_dev;
484     end
485 end
486 end
487
488 //Column noise
489 always @ (negedge enable or posedge clk)
490 begin
491     if (!enable) begin
492         z1 <= 33'd0;
493         z2 <= 33'd0;
494         z3 <= 33'd0;
495         z4 <= 33'd0;
496         col_sum1 <= 36'd0;
497         col_sum2 <= 36'd0;
498         col_dev_pro <= 40'd0;
499         col_dev <= 8'd0;
500         ram_data <= 8'd0;
501         ram_wraddress <= 11'd0;
502         ram_rdaddress <= 11'd0;
503         ram_wren <= 1'b0;
504         ram_rden <= 1'b0;
505     end else begin
506         /*----- STAGE 1 -----*/
507         z1 <=
508             (col_dev_reg[0] ? {1'd0, s2} : 0) +
509             (col_dev_reg[1] ? {s2, 1'd0} : 0);
510         z2 <=
511             (col_dev_reg[2] ? {1'd0, s2} : 0) +
512             (col_dev_reg[3] ? {s2, 1'd0} : 0);
513         z3 <=
514             (col_dev_reg[4] ? {1'd0, s2} : 0) +
515             (col_dev_reg[5] ? {s2, 1'd0} : 0);
516         z4 <=
517             (col_dev_reg[6] ? {1'd0, s2} : 0) +
518             (col_dev_reg[7] ? {s2, 1'd0} : 0);
519
520         /*----- STAGE 2 -----*/
521
522         col_sum1 <= z1 + {z2, 2'd0};
523         col_sum2 <= z3 + {z4, 2'd0};
524
525         /*----- STAGE 3 -----*/
526
527         col_dev_pro <= col_sum1 + {col_sum2, 4'd0};
528
529         /*----- STAGE 4 -----*/
530
531         if (height_counter == 0 && width_counter < width-1) begin
532             if (s3 <= sigma) begin
533                 if (s3[0] == 1) begin
534                     ram_data <= {1'b1, col_dev_pro[38:32]};
535                     ram_wraddress <= mem_wr_counter;
536                     ram_wren <= 1'b1;

```

```

537         col_dev <= {1'b1, col_dev_pro[38:32]};
538     end else begin
539         ram_data <= col_dev_pro[39:32];
540         ram_wraddress <= mem_wr_counter;
541         ram_wren <= 1'b1;
542         col_dev <= col_dev_pro[39:32];
543     end
544 end else if (s3 <= sigma2) begin
545     if (s3[0] == 1) begin
546         ram_data <= {1'b1, col_dev_reg[6:0] + col_dev_pro[38:32]};
547         ram_wraddress <= mem_wr_counter;
548         ram_wren <= 1'b1;
549         col_dev <= {1'b1, col_dev_reg[6:0] + col_dev_pro[38:32]};
550     end else begin
551         ram_data <= col_dev_reg + col_dev_pro[39:32];
552         ram_wraddress <= mem_wr_counter;
553         ram_wren <= 1'b1;
554         col_dev <= col_dev_reg + col_dev_pro[39:32];
555     end
556 end else begin
557     if (s3[0] == 1) begin
558         ram_data <= {1'b1, (col_dev_reg[6:0]<<1) + col_dev_pro
559             [38:32]};
560         ram_wraddress <= mem_wr_counter;
561         ram_wren <= 1'b1;
562         col_dev <= {1'b1, (col_dev_reg[6:0]<<1) + col_dev_pro
563             [38:32]};
564     end else begin
565         ram_data <= (col_dev_reg<<1) + col_dev_pro[39:32];
566         ram_wraddress <= mem_wr_counter;
567         ram_wren <= 1'b1;
568         col_dev <= (col_dev_reg<<1) + col_dev_pro[39:32];
569     end
570 end
571 if (width_counter == width-4) begin
572     ram_rdaddress <= mem_rd_counter;
573     ram_rden <= 1'b1;
574 end else if (width_counter == width-3) begin
575     ram_rdaddress <= mem_rd_counter;
576     ram_rden <= 1'b1;
577 end else if (width_counter == width-2) begin
578     ram_rdaddress <= mem_rd_counter;
579     ram_rden <= 1'b1;
580 end else if (width_counter == width-1) begin
581     ram_wren <= 1'b0;
582     ram_rdaddress <= mem_rd_counter;
583     ram_rden <= 1'b1;
584 end else begin
585     ram_rden <= 1'b0;
586 end
587 end else begin
588     ram_wren <= 1'b0;
589     ram_rdaddress <= mem_rd_counter;

```

```

590     ram_rden <= 1'b1;
591     col_dev <= ram_q;
592     end
593 end
594 end
595
596 always @ (negedge enable or posedge clk)
597 begin
598     if (!enable) begin
599         dataOut <= 8'd0;
600         tempDataOut <= 8'd0;
601         tempDataOut2 <= 8'd0;
602         tempDataOut3 <= 8'd0;
603         col_dev_pipe <= 8'd0;
604         col_dev_pipe2 <= 8'd0;
605     end else begin
606         /*----- STAGE 1 -----*/
607
608         if (noise_type[0] == 1'b1) begin //Photon noise
609             tempDataOut <= pixel_photon;
610         end else begin //No photon noise
611             tempDataOut <= Reg9_pixel;
612         end
613
614         col_dev_pipe <= col_dev;
615
616         /*----- STAGE 2 -----*/
617
618         if (noise_type[1] == 1'b1) begin //Row noise
619             if (row_neg == 1) begin
620                 if (row_dev >= tempDataOut) begin
621                     tempDataOut2 <= 8'd0;
622                 end else begin
623                     tempDataOut2 <= tempDataOut - row_dev;
624                 end
625             end else begin
626                 tempDataOut2 <= tempDataOut + row_dev;
627             end
628         end else begin //No row noise
629             tempDataOut2 <= tempDataOut;
630         end
631
632         col_dev_pipe2 <= col_dev_pipe;
633
634         /*----- STAGE 3 -----*/
635
636         if (noise_type[2] == 1'b1) begin //Column noise
637             if (col_dev_pipe2[7] == 1) begin
638                 if (col_dev_pipe2[6:0] >= tempDataOut2) begin
639                     tempDataOut3 <= 8'd0;
640                 end else begin
641                     tempDataOut3 <= tempDataOut2 - col_dev_pipe2[6:0];

```

## TILLEGG A. VERILOGKODEN TIL EMULATOREN

---

```
642         end
643     end else begin
644         tempDataOut3 <= tempDataOut2 + col_dev_pipe2[6:0];
645     end
646 end else begin //No column noise
647     tempDataOut3 <= tempDataOut2;
648 end
649
650 /*----- STAGE 4
        -----*/
651
652     dataOut <= tempDataOut3;
653
654 end
655 end
656
657 endmodule
```

# Tillegg B

## Emulatoren i testsystemet

```
1 module noise_generator (
2   /* inputs */
3   enable ,
4   clk ,
5   dataIn ,
6   dataValid ,
7   resolution ,
8   noise_type ,
9   row_std_dev ,
10  col_std_dev ,
11  cpu_ready ,
12
13  /* outputs */
14  data_ready ,
15  finished ,
16  dataOut
17 );
18
19 input      enable;
20 input      clk;
21 input [ 7 : 0] dataIn;
22 input      dataValid;
23 input [ 1 : 0] resolution;
24 input [ 2 : 0] noise_type;
25 input [ 7 : 0] row_std_dev;
26 input [ 7 : 0] col_std_dev;
27 input      cpu_ready;
28
29 output      data_ready;
30 output      finished;
31 output [ 7 : 0] dataOut;
32
33 reg      data_ready;
34 reg      finished;
35 reg [ 7 : 0] dataOut;
36
37 reg [10 : 0] width;
38 reg [10 : 0] width_counter;
39 reg [10 : 0] height;
40 reg [10 : 0] height_counter;
```

## TILLEGG B. EMULATOREN I TESTSYSTEMET

---

```
41 reg      [20 : 0] total;
42 reg      [20 : 0] pixel_counter;
43 reg      [ 3 : 0] counter;
44
45 reg [7:0] rom_address;
46 reg [15:0] rom_q;
47
48 reg [7:0] ram_data;
49 reg ram_wren;
50 reg ram_rden;
51 reg [10:0] ram_wraddress;
52 reg [10:0] ram_rdaddress;
53 wire [7:0] ram_q;
54
55 reg [31:0] rnd;
56 reg [31:0] s1;
57 reg [31:0] s2;
58 reg [31:0] s3;
59
60 reg [47:0] produkt;
61 reg [39:0] part_produkt1;
62 reg [39:0] part_produkt2;
63 reg [35:0] part1;
64 reg [35:0] part2;
65 reg [35:0] part3;
66 reg [35:0] part4;
67 reg [33:0] y1;
68 reg [33:0] y2;
69 reg [33:0] y3;
70 reg [33:0] y4;
71 reg [33:0] y5;
72 reg [33:0] y6;
73 reg [33:0] y7;
74 reg [33:0] y8;
75
76 reg [7:0] deviation;
77 reg [7:0] row_dev;
78 reg [39:0] row_dev_pro;
79 reg [35:0] row_sum1;
80 reg [35:0] row_sum2;
81 reg [32:0] x1;
82 reg [32:0] x2;
83 reg [32:0] x3;
84 reg [32:0] x4;
85 reg row_neg;
86
87 reg [7:0] col_dev;
88 reg [7:0] col_dev_pipe;
89 reg [7:0] col_dev_pipe2;
90 reg [39:0] col_dev_pro;
91 reg [35:0] col_sum1;
92 reg [35:0] col_sum2;
93 reg [15:0] col_buffer;
94 reg [32:0] z1;
95 reg [32:0] z2;
```

```

96 reg [32:0] z3;
97 reg [32:0] z4;
98
99 reg [15:0] sqrt;
100
101 reg [7:0] pipe_pixel5;
102 reg [15:0] pipe_sqrt1;
103 reg [15:0] pipe_sqrt2;
104 reg [15:0] pipe_sqrt3;
105 reg [15:0] pipe_sqrt4;
106 reg [15:0] pipe_sqrt5;
107
108 reg [7:0] temp;
109
110 reg [7:0] tempDataOut;
111 reg [7:0] tempDataOut2;
112 reg [7:0] tempDataOut3;
113
114 reg [7:0] col_dev_reg;
115 reg [7:0] row_dev_reg;
116 reg [2:0] noise_type_reg;
117
118 parameter [31:0] number1 = 32'hFFFFFFFE;
119 parameter [31:0] number2 = 32'hFFFFFFF8;
120 parameter [31:0] number3 = 32'hFFFFFFF0;
121
122 parameter [31:0] sigma = 32'hAE147AE1;
123 parameter [31:0] sigma2 = 32'hF3333333;
124
125 test_rom rom_instance(
126     .clock (clk),
127     .resetn (enable),
128     .address (rom_address),
129     .q (rom_q)
130 );
131
132 test_ram_dp ram_instance(
133     .data (ram_data),
134     .waddress (ram_waddress),
135     .wren (ram_wren),
136     .rdaddress (ram_rdaddress),
137     .rden (ram_rden),
138     .clock (clk),
139     .resetn (enable),
140     .q (ram_q)
141 );
142
143 //Tausworthe pseudo random generator
144 always @ (negedge enable or posedge clk)
145 begin
146     if (!enable) begin
147         s1 = 32'hFFFFFFF;
148         s2 = 32'hCCCCCCC;
149         s3 = 32'hFF00FF;
150     end else begin

```

```

151     s1 = (((s1<<13)^s1)>>19)^((s1&number1)<<12);
152     s2 = (((s2<<2)^s2)>>25)^((s2&number2)<<4);
153     s3 = (((s3<<3)^s3)>>11)^((s3&number3)<<17);
154     rnd = s1^s2^s3;
155 end
156 end
157
158 always @ (resolution)
159 begin
160     case (resolution)
161         2'b00:
162             begin
163                 width = 11'd1920;
164                 height = 11'd1080;
165                 total = 21'd2073600;
166             end
167         2'b01:
168             begin
169                 width = 11'd640;
170                 height = 11'd480;
171                 total = 21'd307200;
172             end
173         2'b10:
174             begin
175                 width = 11'd1280;
176                 height = 11'd720;
177                 total = 21'd921600;
178             end
179         2'b11:
180             begin
181                 width = 11'd1920;
182                 height = 11'd1080;
183                 total = 21'd2073600;
184             end
185     endcase
186 end
187
188 always @ (row_std_dev)
189 begin
190     row_dev_reg <= row_std_dev;
191 end
192
193 always @ (col_std_dev)
194 begin
195     col_dev_reg <= col_std_dev;
196 end
197
198 always @ (noise_type)
199 begin
200     noise_type_reg <= noise_type;
201 end
202
203 //Counters
204 always @ (posedge dataValid or negedge enable)
205 begin

```



```

206  if (!enable) begin
207      width_counter <= 11'd0;
208      height_counter <= 11'd0;
209  end else begin
210      if (width_counter >= width) begin
211          width_counter <= 11'd1;
212          if (height_counter >= height-1) begin
213              height_counter <= 11'd0;
214          end else begin
215              height_counter <= height_counter + 11'd1;
216          end
217      end else begin
218          width_counter <= width_counter + 11'd1;
219      end
220  end
221 end
222
223 //Row Noise
224 always @ (negedge enable or posedge clk)
225 begin
226     if (!enable) begin
227         row_dev = 8'd0;
228         x1 = 33'd0;
229         x2 = 33'd0;
230         x3 = 33'd0;
231         x4 = 33'd0;
232         row_sum1 = 36'd0;
233         row_sum2 = 36'd0;
234         row_dev_pro = 40'd0;
235     end else begin
236 /*----- STAGE 1 -----*/
237
238         x1 <=
239             (row_dev_reg[0] ? {1'd0, s3} : 0) +
240             (row_dev_reg[1] ? {s3, 1'd0} : 0);
241         x2 <=
242             (row_dev_reg[2] ? {1'd0, s3} : 0) +
243             (row_dev_reg[3] ? {s3, 1'd0} : 0);
244         x3 <=
245             (row_dev_reg[4] ? {1'd0, s3} : 0) +
246             (row_dev_reg[5] ? {s3, 1'd0} : 0);
247         x4 <=
248             (row_dev_reg[6] ? {1'd0, s3} : 0) +
249             (row_dev_reg[7] ? {s3, 1'd0} : 0);
250
251 /*----- STAGE 2 -----*/
252
253         row_sum1 <= x1 + {x2, 2'd0};
254         row_sum2 <= x3 + {x4, 2'd0};
255
256 /*----- STAGE 3 -----*/
257
258         row_dev_pro <= row_sum1 + {row_sum2, 4'd0};
259
260 /*----- STAGE 4 -----*/

```

```

261
262     if (width_counter == 1 && counter == 1) begin
263         if (s2 <= sigma) begin
264             row_dev <= row_dev_pro[39:32];
265         end else if (s2 <= sigma2) begin
266             row_dev <= row_dev_reg + row_dev_pro[39:32];
267         end else begin
268             row_dev <= (row_dev_reg << 1) + row_dev_pro[39:32];
269         end
270
271         if (s2[0] == 1) begin
272             row_neg <= 1'b1;
273         end else begin
274             row_neg <= 1'b0;
275         end
276     end else begin
277         row_neg <= row_neg;
278         row_dev <= row_dev;
279     end
280 end
281 end
282
283 //Column noise
284 always @ (negedge enable or posedge clk)
285 begin
286     if (!enable) begin
287         col_dev = 8'd0;
288         z1 = 33'd0;
289         z2 = 33'd0;
290         z3 = 33'd0;
291         z4 = 33'd0;
292         col_sum1 = 36'd0;
293         col_sum2 = 36'd0;
294         col_dev_pro = 40'd0;
295     end else begin
296         /*----- STAGE 1 -----*/
297         z1 <=
298             (col_dev_reg[0] ? {1'd0, s2} : 0) +
299             (col_dev_reg[1] ? {s2, 1'd0} : 0);
300         z2 <=
301             (col_dev_reg[2] ? {1'd0, s2} : 0) +
302             (col_dev_reg[3] ? {s2, 1'd0} : 0);
303         z3 <=
304             (col_dev_reg[4] ? {1'd0, s2} : 0) +
305             (col_dev_reg[5] ? {s2, 1'd0} : 0);
306         z4 <=
307             (col_dev_reg[6] ? {1'd0, s2} : 0) +
308             (col_dev_reg[7] ? {s2, 1'd0} : 0);
309
310         /*----- STAGE 2 -----*/
311
312         col_sum1 <= z1 + {z2, 2'd0};
313         col_sum2 <= z3 + {z4, 2'd0};
314
315         /*----- STAGE 3 -----*/

```

```

316
317     col_dev_pro <= col_sum1 + {col_sum2, 4'd0};
318
319 /*----- STAGE 4 -----*/
320
321     if (height_counter == 0 && counter == 1) begin
322         if (s3 <= sigma) begin
323             if (s3[0] == 1) begin
324                 ram_data <= {1'b1, col_dev_pro[38:32]};
325                 ram_wraddress <= width_counter - 11'd1;
326                 ram_wren <= 1'b1;
327                 col_dev <= {1'b1, col_dev_pro[38:32]};
328             end else begin
329                 ram_data <= col_dev_pro[39:32];
330                 ram_wraddress <= width_counter - 11'd1;
331                 ram_wren <= 1'b1;
332                 col_dev <= col_dev_pro[39:32];
333             end
334         end else if (s3 <= sigma2) begin
335             if (s3[0] == 1) begin
336                 ram_data <= {1'b1, col_dev_reg[6:0] + col_dev_pro[38:32]};
337                 ram_wraddress <= width_counter - 11'd1;
338                 ram_wren <= 1'b1;
339                 col_dev <= {1'b1, col_dev_reg[6:0] + col_dev_pro[38:32]};
340             end else begin
341                 ram_data <= col_dev_reg + col_dev_pro[39:32];
342                 ram_wraddress <= width_counter - 11'd1;
343                 ram_wren <= 1'b1;
344                 col_dev <= col_dev_reg + col_dev_pro[39:32];
345             end
346         end else begin
347             if (s3[0] == 1) begin
348                 ram_data <= {1'b1, (col_dev_reg[6:0] << 1) + col_dev_pro
349                     [38:32]};
350                 ram_wraddress <= width_counter - 11'd1;
351                 ram_wren <= 1'b1;
352                 col_dev <= {1'b1, (col_dev_reg[6:0] << 1) + col_dev_pro
353                     [38:32]};
354             end else begin
355                 ram_data <= (col_dev_reg << 1) + col_dev_pro[39:32];
356                 ram_wraddress <= width_counter - 11'd1;
357                 ram_wren <= 1'b1;
358                 col_dev <= (col_dev_reg << 1) + col_dev_pro[39:32];
359             end
360         end
361     end
362
363     end else if (height_counter > 0 && counter == 1) begin
364         ram_wren <= 1'b0;
365         ram_rdaddress <= width_counter;
366         ram_rden <= 1'b1;
367         col_dev <= ram_q;
368     end else begin
369         col_dev <= col_dev;
370         ram_wren <= ram_wren;
371         ram_rden <= ram_rden;

```

```

369     ram_wraddress <= ram_wraddress;
370     ram_rdaddress <= ram_rdaddress;
371 end
372 end
373 end
374
375 //Photon noise and control block
376 always @ (posedge clk or negedge enable)
377 begin
378     if (enable == 0) begin
379         dataOut <= 8'd0;
380         counter <= 4'd0;
381         pixel_counter <= 21'd0;
382         finished <= 1'b0;
383         rom_address <= 8'd0;
384         data_ready <= 1'b0;
385         sqrt <= 16'd0;
386         pipe_sqrt1 <= 16'd0;
387         pipe_sqrt2 <= 16'd0;
388         pipe_sqrt3 <= 16'd0;
389         pipe_sqrt4 <= 16'd0;
390         pipe_sqrt5 <= 16'd0;
391         pipe_pixel5 <= 8'd0;
392         y1 <= 34'd0;
393         y2 <= 34'd0;
394         y3 <= 34'd0;
395         y4 <= 34'd0;
396         y5 <= 34'd0;
397         y6 <= 34'd0;
398         y7 <= 34'd0;
399         y8 <= 34'd0;
400         part1 <= 36'd0;
401         part2 <= 36'd0;
402         part3 <= 36'd0;
403         part4 <= 36'd0;
404         part_produkt1 <= 40'd0;
405         part_produkt2 <= 40'd0;
406         produkt <= 48'd0;
407         temp <= 8'd0;
408         tempDataOut <= 8'd0;
409         tempDataOut2 <= 8'd0;
410         deviation <= 8'd0;
411     end else begin
412         if (counter == 0 && dataValid == 1) begin
413             rom_address <= dataIn;
414
415             counter <= counter + 4'd1;
416         end else if (counter == 1) begin
417             counter = counter + 4'd1;
418         end else if (counter == 2) begin
419             counter = counter + 4'd1;
420         end else if (counter == 3) begin
421             sqrt <= rom_q;
422             counter = counter + 4'd1;
423         end else if (counter == 4) begin

```

```

424
425     y1 <=
426         (sqrt[0] ? {1'd0, rnd} : 0) +
427         (sqrt[1] ? {rnd, 1'd0} : 0);
428     y2 <=
429         (sqrt[2] ? {1'd0, rnd} : 0) +
430         (sqrt[3] ? {rnd, 1'd0} : 0);
431     y3 <=
432         (sqrt[4] ? {1'd0, rnd} : 0) +
433         (sqrt[5] ? {rnd, 1'd0} : 0);
434     y4 <=
435         (sqrt[6] ? {1'd0, rnd} : 0) +
436         (sqrt[7] ? {rnd, 1'd0} : 0);
437     y5 <=
438         (sqrt[8] ? {1'd0, rnd} : 0) +
439         (sqrt[9] ? {rnd, 1'd0} : 0);
440     y6 <=
441         (sqrt[10] ? {1'd0, rnd} : 0) +
442         (sqrt[11] ? {rnd, 1'd0} : 0);
443     y7 <=
444         (sqrt[12] ? {1'd0, rnd} : 0) +
445         (sqrt[13] ? {rnd, 1'd0} : 0);
446     y8 <=
447         (sqrt[14] ? {1'd0, rnd} : 0) +
448         (sqrt[15] ? {rnd, 1'd0} : 0);
449
450     pipe_sqrt1 <= sqrt;
451     counter = counter + 4'd1;
452 end else if (counter == 5) begin
453     part1 <= y1 + {y2, 2'd0};
454     part2 <= y3 + {y4, 2'd0};
455     part3 <= y5 + {y6, 2'd0};
456     part4 <= y7 + {y8, 2'd0};
457
458     pipe_sqrt2 <= pipe_sqrt1;
459     counter = counter + 4'd1;
460 end else if (counter == 6) begin
461     part_produkt1 <= part1 + {part2, 4'd0};
462     part_produkt2 <= part3 + {part4, 4'd0};
463
464     pipe_sqrt3 <= pipe_sqrt2;
465     counter = counter + 4'd1;
466 end else if (counter == 7) begin
467     produkt <= part_produkt1 + {part_produkt2, 8'd0};
468
469     pipe_sqrt4 <= pipe_sqrt3;
470     counter = counter + 4'd1;
471 end else if (counter == 8) begin
472     if (produkt[43] == 1) begin //avrunding
473         deviation <= {4'b0000, produkt[47:44]} + 8'b00000001;
474     end else begin
475         deviation <= {4'b0000, produkt[47:44]};
476     end
477
478     pipe_sqrt5 <= pipe_sqrt4;

```

```

479     pipe_pixel5 <= dataIn;
480     counter = counter + 4'd1;
481 end else if (counter == 9) begin
482     if (noise_type_reg[0] == 1) begin //Photon noise
483         if (s1 <= sigma) begin
484             if (rnd[0] == 1) begin
485                 if (pipe_pixel5+deviation > 255) begin
486                     tempDataOut <= 8'd255;
487                 end else begin
488                     tempDataOut <= pipe_pixel5 + deviation;
489                 end
490             end else begin
491                 if (deviation >= pipe_pixel5) begin
492                     tempDataOut <= 8'd0;
493                 end else begin
494                     tempDataOut <= pipe_pixel5 - deviation;
495                 end
496             end
497         end else if (s1 <= sigma2) begin
498             temp = pipe_sqrt5[11] ? {4'd0, pipe_sqrt5[15:12]} + 8'd1 :
499                 {4'd0, pipe_sqrt5[15:12]};
500             if (rnd[0] == 1) begin
501                 if (pipe_pixel5+temp+deviation > 255) begin
502                     tempDataOut <= 8'd255;
503                 end else begin
504                     tempDataOut <= pipe_pixel5 + temp + deviation;
505                 end
506             end else begin
507                 if (deviation + temp >= pipe_pixel5) begin
508                     tempDataOut <= 8'd0;
509                 end else begin
510                     tempDataOut <= pipe_pixel5 - temp - deviation;
511                 end
512             end
513         end else begin
514             temp = pipe_sqrt5[10] ? {3'd0, pipe_sqrt5[15:11]} + 8'd1 :
515                 {3'd0, pipe_sqrt5[15:11]};
516             if (rnd[0] == 1) begin
517                 if (pipe_pixel5+temp+deviation > 255) begin
518                     tempDataOut <= 8'd255;
519                 end else begin
520                     tempDataOut <= pipe_pixel5 + temp + deviation;
521                 end
522             end else begin
523                 if (deviation + temp >= pipe_pixel5) begin
524                     tempDataOut <= 8'd0;
525                 end else begin
526                     tempDataOut <= pipe_pixel5 - temp - deviation;
527                 end
528             end
529         end else begin //No photon noise
530             tempDataOut <= pipe_pixel5;
531         end

```

```

532     counter = counter + 4'd1;
533 end else if (counter == 10) begin
534     if (noise_type_reg[1] == 1) begin //Row noise
535         if (row_neg == 1) begin
536             if (row_dev >= tempDataOut) begin
537                 tempDataOut2 <= 8'd0;
538             end else begin
539                 tempDataOut2 <= tempDataOut - row_dev;
540             end
541         end else begin
542             if (tempDataOut + row_dev >= 255) begin
543                 tempDataOut2 <= 8'd255;
544             end else begin
545                 tempDataOut2 <= tempDataOut + row_dev;
546             end
547         end
548     end else begin //No row noise
549         tempDataOut2 <= tempDataOut;
550     end
551     col_dev_pipe2 <= col_dev;
552     counter <= counter + 4'd1;
553 end else if (counter == 11) begin
554     if (noise_type_reg[2] == 1) begin //Column noise
555         if (col_dev_pipe2[7] == 1) begin
556             if ({1'b0, col_dev_pipe2[6:0]} >= tempDataOut2) begin
557                 tempDataOut3 <= 8'd0;
558             end else begin
559                 tempDataOut3 <= tempDataOut2 - {1'b0, col_dev_pipe2[6:0]};
560             end
561         end else begin
562             if ({1'b0, col_dev_pipe2[6:0]} + tempDataOut2 >= 255) begin
563                 tempDataOut3 <= 8'd255;
564             end else begin
565                 tempDataOut3 <= tempDataOut2 + {1'b0, col_dev_pipe2[6:0]};
566             end
567         end
568     end else begin //No column noise
569         tempDataOut3 <= tempDataOut2;
570     end
571     counter <= counter + 4'd1;
572 end else if (counter == 12) begin
573     dataOut <= tempDataOut3;
574     counter <= counter + 4'd1;
575     pixel_counter <= pixel_counter + 21'd1;
576 end else if (counter == 13) begin
577     data_ready = 1'b1;
578     if (cpu_ready >= 1) begin
579         data_ready <= 1'b0;
580         counter <= 4'd0;
581         if (pixel_counter >= total) begin
582             finished <= 1'b1;
583         end else begin
584             finished <= 1'b0;
585         end
586     end else begin

```

## TILLEGG B. EMULATOREN I TESTSYSTEMET

---

```
587         counter <= counter;
588     end
589 end else begin
590     counter <= 4'd0;
591 end
592 end
593 end
594
595 endmodule
```



# Tillegg C

## Nios II C-kode

Koden under viser de modifiserte metodene i Alteras Simple Socket Server-program som brukes i testsystemet.

```
1 int width = 0;
2 int height = 0;
3
4 INT32U teller = 0;
5 INT32U counter = 0;
6 INT8U teller2 = 0;
7 int teller3 = 0;
8
9 INT16U width_counter = 0;
10 INT16U height_counter = 0;
11
12 INT32U totalt = 2073600;
13 INT32U res_teller = 0;
14
15 int enable = 0;
16 int mode = 0;
17
18 INT8U chosen = 0;
19 int test = 0;
20
21 int ferdig = 0;
22
23 int trans_size = 480;
24 int dimX = 1920;
25
26 INT8U buffer[2073600];
27 INT8U resultat[2073600];
28 INT8U temp1[480];
29 INT8U temp2[480];
30 INT8U temp3[480];
31 INT8U temp4[480];
32
33 volatile int ram_address = 0xF0;
34
35 volatile int var_width;
36 volatile int var_height;
37
38 int knapp_teller = 0;
```

```
39
40 INT32U data_teller = 0;
41 int ramme_teller = 0;
42
43 void setup()
44 {
45     //printf("Noise_type: %d\n", IORD(TEST_PIO_BASE, 0));
46
47     IOWR(LED_PIO_BASE, 0, 0x22);
48     IOWR(ENABLE_PIO_BASE, 0, 0x00);
49     IOWR(DATA_VALID_PIO_BASE, 0, 0x00);
50     IOWR(ENABLE_PIO_BASE, 0, 0xff);
51     IOWR(RESOLUTION_PIO_BASE, 0, 0x3);
52     IOWR(NOISE_TYPE_PIO_BASE, 0, 0x4);
53     IOWR(ROW_STD_DEV_PIO_BASE, 0, 0x07);
54     IOWR(COL_STD_DEV_PIO_BASE, 0, 0x07);
55     IOWR(DATA_PIO_BASE, 0, buffer[counter++]);
56     IOWR(DATA_VALID_PIO_BASE, 0, 0xFF);
57
58     while(ferdig == 0)
59     {
60         if (IORD_8DIRECT(DATA_READY_PIO_BASE, 0) >= 1)
61         {
62             if (width_counter >= 1920)
63             {
64                 width_counter = 1;
65                 if (height_counter >= 1080)
66                 {
67                     height_counter = 1;
68                 }
69                 else
70                 {
71                     height_counter++;
72                 }
73             }
74             else
75             {
76                 width_counter++;
77             }
78
79             IOWR(DATA_VALID_PIO_BASE, 0, 0x00);
80             resultat[res_teller++] = IORD(DATA_FROM_HW_PIO_BASE, 0);
81
82             if (res_teller < totalt)
83             {
84                 IOWR(DATA_PIO_BASE, 0, buffer[counter++]);
85             }
86             else
87             {
88                 ferdig = 1;
89                 break;
90             }
91             IOWR(CPU_READY_PIO_BASE, 0, 0xFF);
92         }
93         else
```

```

94         {
95             IOWR(CPU_READY_PIO_BASE, 0, 0x00);
96             IOWR(DATA_VALID_PIO_BASE, 0, 0xFF);
97         }
98     }
99
100     printf("Ferdig\n");
101
102 }
103
104 void result(SSSConn* conn)
105 {
106     int i;
107     int j;
108
109     if (mode == 1)
110     {
111         printf("Start sending\n");
112         for (i=0; i<1080; i++)
113         {
114             for (j=0; j<1920; j++)
115             {
116                 if (j<trans_size)
117                     temp1[j] = resultat[j+i*dimX];
118                 else if (j<2*trans_size)
119                     temp2[j-trans_size] = resultat[j+i*dimX];
120                 else if (j<3*trans_size)
121                     temp3[j-2*trans_size] = resultat[j+i*dimX];
122                 else
123                     temp4[j-3*trans_size] = resultat[j+i*dimX];
124             }
125             send(conn->fd, temp1, sizeof(temp1)*sizeof(INT8U), 0);
126             usleep(10);
127             send(conn->fd, temp2, sizeof(temp2)*sizeof(INT8U), 0);
128             usleep(10);
129             send(conn->fd, temp3, sizeof(temp3)*sizeof(INT8U), 0);
130             usleep(10);
131             send(conn->fd, temp4, sizeof(temp4)*sizeof(INT8U), 0);
132             usleep(10);
133         }
134         printf("Sendt\n");
135     }
136 }
137
138 /*
139  * sss_handle_accept()
140  *
141  * This routine is called when ever our listening socket has an
142  * incoming
143  * connection request. Since this example has only data transfer socket
144  * ,
145  * we just look at it to see whether its in use... if so, we accept the
146  * connection request and call the telent_send_menu() routine to
147  * transmit

```

```
145 * instructions to the user. Otherwise, the connection is already in
    use;
146 * reject the incoming request by immediately closing the new socket.
147 *
148 * We'll also print out the client's IP address.
149 */
150 void sss_handle_accept(int listen_socket, SSSConn* conn)
151 {
152     int socket, len;
153     struct sockaddr_in incoming_addr;
154
155     len = sizeof(incoming_addr);
156
157     if ((conn)->fd == -1)
158     {
159         if ((socket=accept(listen_socket, (struct sockaddr*)&incoming_addr, &
160             len)) < 0)
161         {
162             alt_NetworkErrorHandler(EXPANDED_DIAGNOSIS_CODE,
163                                     "[sss_handle_accept] accept failed");
164         }
165         else
166         {
167             (conn)->fd = socket;
168             printf("[sss_handle_accept] accepted connection request from %s\n",
169                 inet_ntoa(incoming_addr.sin_addr));
170             result(conn);
171         }
172     }
173     else
174     {
175         printf("[sss_handle_accept] rejected connection request from %s\n",
176             inet_ntoa(incoming_addr.sin_addr));
177     }
178     return;
179 }
180
181 /*
182 * sss_exec_command()
183 *
184 * This routine is called whenever we have new, valid receive data from
    our
185 * sss connection. It will parse through the data simply looking for
    valid
186 * commands to the sss server.
187 *
188 * Incoming commands to talk to the board LEDs are handled by sending
    the
189 * MicroC/OS-II SSSLedCommandQ a pointer to the value we received.
190 *
191 * If the user wishes to quit, we set the "close" member of our SSSConn
192 * struct, which will be looked at back in sss_handle_receive() when it
193 * comes time to see whether to close the connection or not.
```

```
194  */
195 void sss_exec_command(SSSConn* conn)
196 {
197     int bytes_to_process = conn->rx_wr_pos - conn->rx_rd_pos;
198
199     while(bytes_to_process --)
200     {
201         buffer[teller++] = *(conn->rx_rd_pos++);
202
203         if (teller >= totalt)
204         {
205             INT8U test = buffer[teller - 1];
206             printf("JIPPI!!! Data: %d \n", test);
207         }
208     }
209
210     return;
211 }
```

# Tillegg D

## Matlabsimuleringer

### D.1 Simuleringer av støy med Matlabmetodene *poissrnd* og *randn*

```
1 bilde = imread('daf.png');
2
3 a = bilde(:,:,1);
4 foton = a;
5 rad = a;
6 kolonne = a;
7 alt = a;
8
9 var = 7;
10 var2 = 7;
11 offset2 = zeros(1080, 1);
12
13 seed1 = uint32(16777215);
14 seed2 = uint32(13421772);
15 seed3 = uint32(16711935);
16
17 for y=1:1080
18     for x=1:1920
19
20         temp1 = bitshift(uint32(bitand(seed1, uint32(4294967294))), 12,
21                             32);
22         temp11 = bitshift(bitxor(bitshift(seed1, 13, 32), seed1), -19,
23                             32);
24
25         temp2 = bitshift(uint32(bitand(seed2, uint32(4294967288))), 4);
26         temp22 = bitshift(bitxor(bitshift(seed2, 2, 32), seed2), -25,
27                             32);
28
29         temp3 = bitshift(uint32(bitand(seed3, uint32(4294967280))), 17,
30                             32);
31         temp33 = bitshift(bitxor(bitshift(seed3, 3, 32), seed3), -11,
32                             32);
33
34         seed1 = bitxor(temp1, temp11);
35         seed2 = bitxor(temp2, temp22);
36         seed3 = bitxor(temp3, temp33);
37     end
38 end
39
40 offset2(y,x) = foton - rad - kolonne - alt + var + var2 + offset2(y,x);
```

```
31     seed3 = bitxor(temp3, temp33);
32
33     tall = uint32(bitxor(bitxor(seed1, seed2), seed3));
34
35     temp = 0;
36     for q=1:32
37         if (bitget(tall, q) == 1)
38             temp = temp + 2^-q;
39         end
40     end
41     ol = temp;
42
43     temp = double(a(y,x));
44
45     if (seed1 <= 0.68*2^32)
46         if (bitget(tall, 1) == 1)
47             if (foton(y,x) + sqrt(temp)*ol >= 255)
48                 foton(y,x) = 255;
49             else
50                 foton(y,x) = round(a(y,x) + sqrt(temp)*ol);
51             end
52         else
53             if (foton(y,x) <= sqrt(temp)*ol)
54                 foton(y,x) = 0;
55             else
56                 foton(y,x) = round(a(y,x) - sqrt(temp)*ol);
57             end
58         end
59     elseif (seed1 <= 0.95*2^32)
60         if (bitget(tall, 1) == 1)
61             if (foton(y,x) + sqrt(temp)*(1+ol) >= 255)
62                 foton(y,x) = 255;
63             else
64                 foton(y,x) = round(a(y,x) + sqrt(temp)*(1+ol));
65             end
66         else
67             if (foton(y,x) <= sqrt(temp)*(1+ol))
68                 foton(y,x) = 0;
69             else
70                 foton(y,x) = round(a(y,x) - sqrt(temp)*(1+ol));
71             end
72         end
73     else
74         if (bitget(tall, 1) == 1)
75             if (foton(y,x) + sqrt(temp)*(2+ol) >= 255)
76                 foton(y,x) = 255;
77             else
78                 foton(y,x) = round(a(y,x) + sqrt(temp)*(2+ol));
79             end
80         else
81             if (foton(y,x) <= sqrt(temp)*(2+ol))
82                 foton(y,x) = 0;
83             else
84                 foton(y,x) = round(a(y,x) - sqrt(temp)*(2+ol));
85             end
86         end
87     end
88 end
```

```
86         end
87     end
88
89     if (x == 1)
90         rnd = 0;
91         for t=1:32
92             if (bitget(seed2, 1) == 1)
93                 rnd = rnd + 2^(t-33);
94             end
95         end
96
97         if (seed3 <= 0.68*2^32)
98             rnd = rnd + 1;
99         elseif (seed3 <= 0.95*2^32)
100             rnd = rnd + 2;
101         else
102             rnd = rnd + 3;
103         end
104
105         if (bitget(seed2, 1) == 1)
106             rnd = -rnd;
107         end
108
109         offset = round(var*rnd);
110     end
111
112     if (y == 1)
113         rnd2 = 0;
114         for s=1:32
115             if (bitget(seed3, 1) == 1)
116                 rnd2 = rnd2 + 2^(s-33);
117             end
118         end
119
120         if (seed2 <= 0.68*2^32)
121             rnd2 = rnd2 + 1;
122         elseif (seed2 <= 0.95*2^32)
123             rnd2 = rnd2 + 2;
124         else
125             rnd2 = rnd2 + 3;
126         end
127
128         if (bitget(seed3, 1) == 1)
129             rnd2 = -rnd2;
130         end
131
132         offset2(x) = round(var2*rnd2);
133     end
134
135     if (foton(y,x)+offset+offset2(x) >= 255)
136         alt(y,x) = 255;
137     elseif (foton(y,x) + offset + offset2(x) <= 0)
138         alt(y,x) = 0;
139     else
140         alt(y,x) = foton(y,x) + offset + offset2(x);
```



```
141         end
142
143         if (a(y,x) + offset2(x) >= 255)
144             kolonne(y,x) = 255;
145         elseif (a(y,x) + offset2(x) <= 0)
146             kolonne(y,x) = 0;
147         else
148             kolonne(y,x) = a(y,x) + offset2(x);
149         end
150
151         if (a(y,x) + offset >= 255)
152             rad(y,x) = 255;
153         elseif (a(y,x) + offset <= 0)
154             rad(y,x) = 0;
155         else
156             rad(y,x) = a(y,x) + offset;
157         end
158
159     end
160 end
161
162 figure(1);
163 imshow(foton);
164 figure(2);
165 imshow(rad);
166 figure(3);
167 imshow(kolonne);
168 figure(4);
169 imshow(alt);
170
171 psnr(foton);
172 psnr(rad);
173 psnr(kolonne);
174 psnr(alt);
```

## D.2 Simuleringer av algoritmene emulatoren bruker

```
1 a = imread('daf.png');
2 a = a(:,:,1);
3 photon = a;
4 row = a;
5 col = a;
6 all = a;
7
8 r1 = randn(1, 1920);
9 r2 = randn(1, 1080);
10
11 col_dev = zeros(1, 1920);
12
13 col_std_dev = 7;
14 row_std_dev = 7;
15
16 for i=1:1080
```

```
17     for j=1:1920
18
19         if (i==1)
20             col_dev(j) = r1(j)*col_std_dev;
21         end
22
23         if (j==1)
24             row_dev = r2(i)*row_std_dev;
25         end
26
27         temp = double(a(i,j));
28         p = poissrnd(temp, 1, 1);
29
30         if (p >= 255)
31             p = 255;
32         elseif (p <= 0)
33             p = 0;
34         end
35
36         photon(i,j) = p;
37
38         if (row(i,j)+row_dev >= 255)
39             row(i,j) = 255;
40         elseif(row(i,j) + row_dev <= 0)
41             row(i,j) = 0;
42         else
43             row(i,j) = row(i,j) + row_dev;
44         end
45
46         if (col(i,j)+col_dev(j) >= 255)
47             col(i,j) = 255;
48         elseif(col(i,j)+col_dev(j) <= 0)
49             col(i,j) = 0;
50         else
51             col(i,j) = col(i,j) + col_dev(j);
52         end
53
54         if (p+row_dev+col_dev(j) >= 255)
55             all(i,j) = 255;
56         elseif (p+row_dev+col_dev(j) <= 0)
57             all(i,j) = 0;
58         else
59             all(i,j) = round(p + row_dev + col_dev(j));
60         end
61
62     end
63 end
64
65
66 figure(1);
67 imshow(photon);
68 figure(2);
69 imshow(row);
70 figure(3);
71 imshow(col);
```

```
72 figure(4);
73 imshow(all);
74
75 psnr(photon);
76 psnr(row);
77 psnr(col);
78 psnr(all);
```

### D.3 Metoden som brukes til å regne ut PSNR, MAE og MSE

```
1 function [psnr, mean] = psnr(a)
2
3 b = imread('daf.png');
4 b = b(:,:,1);
5
6 temp = 0;
7 temp2 = 0;
8
9 for i=1:1080
10     for j=1:1920
11         temp = temp + abs(double(a(i,j)-b(i,j)))^2;
12         temp2 = temp2 + abs(double(a(i,j)-b(i,j)));
13     end
14 end
15
16 mse = temp/(1920*1080);
17 mean = temp2/(1920*1080);
18
19 psnr = 10*log10(double(255^2/mse));
20
21 disp(psnr)
22 disp(mse)
23 disp(mean)
```

# Tillegg E

## C-sharp-program

### E.1 Program

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace ConsoleApplication1
7 {
8     class Program
9     {
10         static void Main(string[] args)
11         {
12             Client klient = new Client();
13             klient.Connect("192.168.1.234", 30);
14             klient.Send();
15             klient.Receive();
16             klient.WriteToFile("C:\\Users\\Rune\\Documents\\resultat.
17                             txt");
18             Console.ReadLine();
19             klient.Close();
20         }
21 }
```

### E.2 Klient

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 using System.IO;
7 using System.Net;
8 using System.Net.Sockets;
9
10 namespace ConsoleApplication1
11 {
```

```
12  class Client
13  {
14      Socket soc;
15      byte[] buffer;
16
17      const int buffersize = 2073600;
18      const int readstep = 480;
19
20      int index;
21
22
23      public Client()
24      {
25          buffer = new byte[ buffersize ];
26          index = 0;
27          Console.WriteLine(" Start");
28          soc = new Socket( AddressFamily.InterNetwork, SocketType.
                Stream, ProtocolType.Tcp);
29          soc.ReceiveTimeout = 30000;
30      }
31
32      public void Connect(string ip, int port)
33      {
34          IPAddress ipadresse = IPAddress.Parse(ip);
35          IPEndPoint ipe = new IPEndPoint(ipadresse, port);
36          Console.WriteLine(" Starting to connect...");
37          soc.Connect(ipe);
38      }
39
40      public void Send()
41      {
42          Console.WriteLine(" Send method");
43          byte[] buf = new byte[] { 49, 0 };
44          if (soc.Connected)
45          {
46              soc.Send(buf);
47          }
48          else Console.WriteLine(" Not connected");
49      }
50
51
52      public void Receive()
53      {
54          Console.WriteLine(" Receive method");
55          while (index < buffersize)
56          {
57              try
58              {
59                  index += soc.Receive(buffer, index, readstep,
                    SocketFlags.None);
60
61                  if (index % 10000 == 0)
62                  {
63                      Console.WriteLine(" Received 10000 byte");
64                  }
65              }
66          }
67      }
68  }
```

```
65         }
66         catch (Exception)
67         {
68
69             Console.WriteLine("Receive FAIL");
70             System.Environment.Exit(0);
71         }
72     }
73 }
74
75 public void WriteToFile(string navn)
76 {
77     Console.WriteLine("WriteToFile method");
78
79     FileStream stream = new FileStream(navn, FileMode.Create);
80     BinaryWriter w = new BinaryWriter(stream);
81     try
82     {
83         w.Write(buffer, 0, index);
84         w.Close();
85     }
86     catch (Exception)
87     {
88         Console.WriteLine("Write to file fail");
89         System.Environment.Exit(0);
90     }
91     Console.WriteLine("Done writing to file");
92 }
93
94 public void Close()
95 {
96     Console.WriteLine("The end");
97     try
98     {
99         soc.Close();
100     }
101     catch (Exception)
102     {
103         Console.WriteLine("Close fail");
104         System.Environment.Exit(0);
105     }
106     Console.ReadLine();
107 }
108 }
109 }
```

# Tillegg F

## Generering av mif-fil

```
1 a = zeros(255,1);
2 temp = zeros(255, 1);
3 komma = zeros(255,1);
4 file = fopen('lookup.mif', 'w');
5 tall = sprintf('DEPTH = 256;\n\nWIDTH = 16;\n\nADDRESS_RADIX = BIN;\n\nDATA_RADIX\n\nCONTENT\n\nBEGIN\n');
6 fprintf(file , tall);
7 for i=1:255
8     a(i) = sqrt(i);
9     temp(i) = round(a(i));
10    if (temp(i) > a(i))
11        temp(i)=temp(i)-1;
12    end
13
14    komma(i) = a(i)-temp(i);
15    bin = '';
16    for j=1:12
17        komma(i) = komma(i)*2;
18        if (komma(i) > 1)
19            komma(i) = komma(i)-1;
20            bin = sprintf('%s1', bin);
21        else
22            bin = sprintf('%s0', bin);
23        end
24    end
25    tall = sprintf('%s: %s%s;\n', dec2bin(i, 8), dec2bin(temp(i), 4),
26        bin);
27    fprintf(file , tall);
28 end
29 tall = sprintf('---\nEND;');
30 fprintf(file , tall);
31 fclose(file);
```