UNIVERSITETET I OSLO
Institutt for informatikk

En adresse-event-representasjon-fargesynsensor med pulstetthetsmodula-sjon og time-to-first modus

Masteroppgave

Dag Halfdan Bryn

15. september 2010

Innhold

Figurer

Sammendrag

I denne masteroppgaven har jeg lagd en laveffekt AER-fargebilledsensor ved hjelp av stablede fotodioder og nevroner med strømtilbakekobling i standard 90 nm CMOS teknologi.

0.1 Todo

Bruke eller slette referansene 4,5,6,7,8,9,12,14,15,16

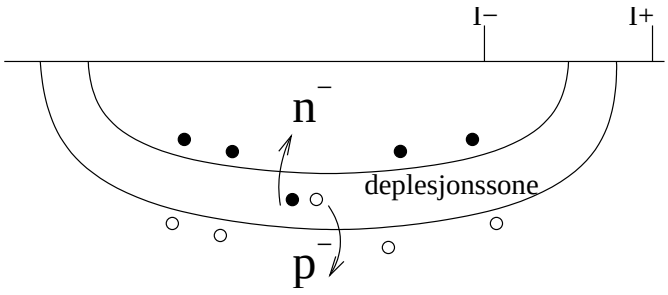
Kapittel 1

Introduksjon

De siste årene er det produsert et vell av billedsensorer. De finnes i alle mobiltelefoner og det kommer hele tiden nye modeller digitalkameraer. Den vanlige måten å lage fargebilledsensor er å lage hvert piksel ved hjelp av fire svart-hvite delpiksler og legge et filter som slipper gjennom bare én farge over hvert delpiksel. Ett fargepiksel består da av ett piksel som detekterer rødt lys, ett som detekterer blått lys og to som detekterer grønt lys. Dette heter en Bayer matrise. En ulempe med denne metoden er at den røde, den grønne og den blå delen av bildet blir forskjøvet i forhold til hverandre. Det tilsvarer å ta ett bilde med blått filter foran kameraet, ett bilde med rødt og ett med grønt filter, flytte kameraet litt mellom hvert av delbildene og så sette sammen delbildene [10]. Resultatet er at bildet blir litt tåkete eller som om bildet hadde vært tatt med et kamera med færre piksler. Det er brukt to metoder for å unngå dette problemet. Én er å separere bildet i tre ved hjelp av speilrefleks og bruke tre billedsensorer med hvert sitt filter foran. Denne metoden er plasskrevende og dyr og egner seg bare i proffesjonelle digitalkameraer. Den andre metoden er stablede fotodioder brukt i FoveonX3 og i denne oppgaven.

1.1 CMOS svart-hvitt lyssensor

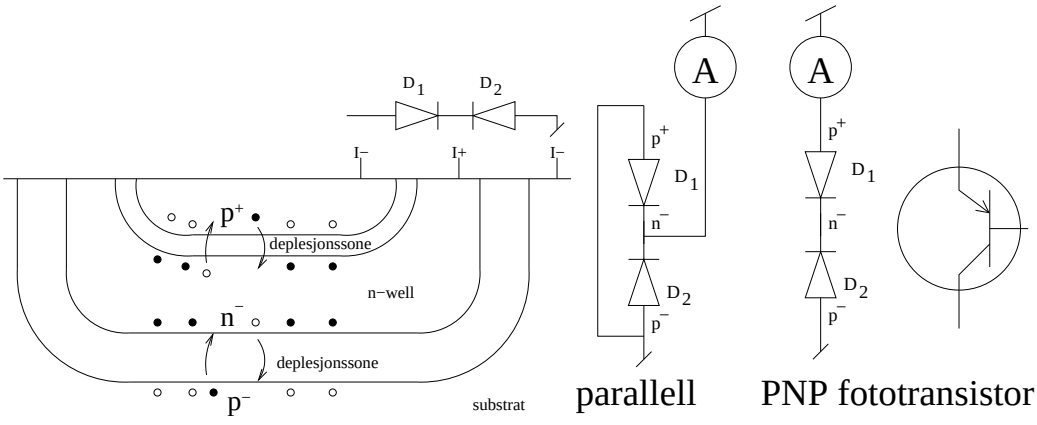
En CMOS lyssensor er en diode som er forspent i sperreretning [7]. Skulle et foton bli absorbert i deplesjonssonen, vil det forårsake en reversstrøm i dioden. Strøm oppstår ved at fotonet eksiterer et elektron fra valensbåndet til ledningsbåndet ved å gi fra seg energi tilsvarende båndgapet. Dette skjer i deplesjonssonen. Når elektronet er i ledningsbåndet, vil diodens forspenning



Figur 1.1: CMOS lyssensor

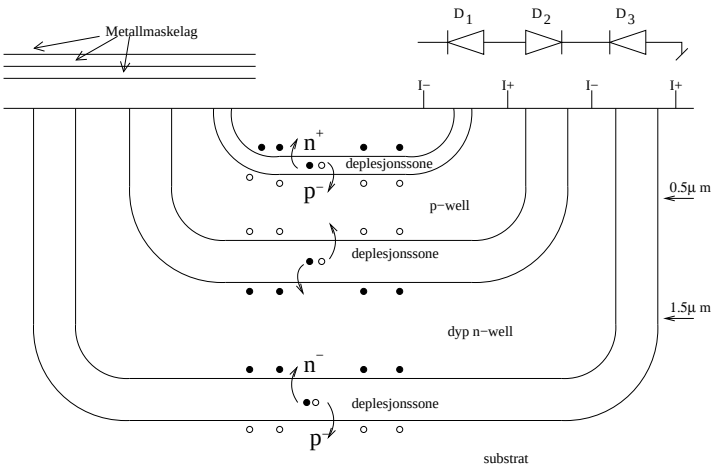
drive det gjennom deplesjonssonen, ut til terminalen og ut av dioden. Figur 1.1 viser tverrsnitt av en CMOS svart-hvitt lyssensor.

1.2 CMOS lyssensor med to PN-overganger

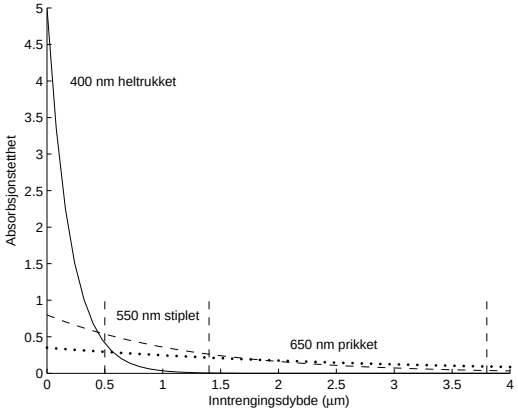


Figur 1.2: lyssensor med to fotodioder

En fotosensor med to dioder kan kobles på to måter: De to diodene kobles i parallell ved å koble p^+ til *gnd*. Alternativt kan de kobles i serie. Dioden D_1 vil da bli foroverforspent og fungere som en basis-emitterdiode, og fotostrømmen i D_2 vil forsterkes opp som kollektorstrøm med transistorens strømforsterkning β. Basisspenningen, over D_1, vil ha et logatritmisk forhold til fotostrømmen. Dette kan brukes for å øke det dynamiske nytteområdet.



Figur 1.3: Stablet CMOS lyssensor

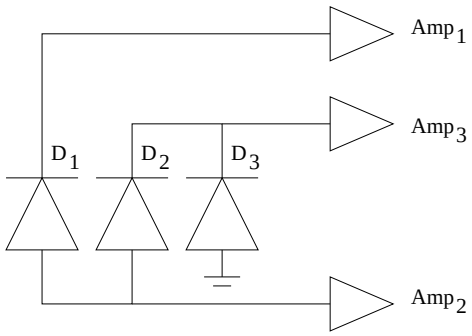


Figur 1.4: Absorbsjonssannsynlighetstetthet som funksjon av inntrengingsdybde

1.3 CMOS lyssensor med tre PN-overganger

I en fargelyssensor som bruker stablede fotodioder er det tre PN-overganger. Figur 1.3 viser tverrsnitt av en CMOS fargelyssensor. Denne teknikken er realisert i FoveonX3 og er nærmere beskrevet i [4, 5] Sensoren skiller mellom farger ved at forholdet mellom strømmene som genereres i de forskjellige diodene $D_1 - D_3$ varierer med fargen på lyset. Jo mer energi et foton har, jo høyere sannsynlighet er det for at fotonet blir absorbert pr. lengdeenhet i silisiumet. Figur 1.4 viser dette grafisk. Denne figuren og tallene kommer fra Foveon [4] og vil ikke nødvendigvis representere denne billedsensoren

siden Foveon kan ha (eller mer sannsynlig: har) brukt en annen CMOS-prosess. Tall og utregninger er tatt med for å vise at det er mulig å beregne forholdet mellom strøm i de tre fotodiodene og inn til nevronene som funksjon av bølgelengde og hvordan det gjøres. Tallene i tabell 1.1 viser den akkumulerte absorbsjonssannsynlighetstettheten mellom delelinjene mellom PN-overgangene. I virkeligheten vil deplesjonssonene utgjøre soner som indikert på figur 1.3 altså bare deler av arealet mellom de vertikale stiplede linjene.



Figur 1.5: Skjematikk som viser de tre fotodiodene

PN-overgang#	1	2	3
400 nm	91.79 %	8.12 %	0.09 %
550 nm	32.97 %	34.40 %	27.84 %
650 nm	16.05 %	22.68 %	34.81 %

Tabell 1.1: Overføringsmatrise som viser strøm i PN-overgangene som funksjon av lysets bølgelengde

Blått lys består av fotoner med høy energi. Disse fotonene vil med stor sannsynlighet bli absorbert i den første PN-overgangen. Det vil være igjen litt energi, men det som er igjen vil i regelen ikke være nok til å generere noen. Noen få blå fotoner passerer den første PN-overgangen uten å avgi energi og vil sannsynligvis bli absorbert i den andre og nesten ingen fotoner rekker fram til den siste PN-overgangen. Nesten all strømmen blir derfor generert i den første PN-overgangen.

Grønt lys består av fotoner med middels energi. En del av disse fotonene vil bli absorbert i den første PN-overgangen. Resten av fotonene passerer den første PN-overgangen uten å bli absorbert og vil sannsynligvis bli absorbert i den andre og noen fotoner rekker fram til den siste PN-overgangen.

Rødt lys består av fotoner med lav energi. Få av fotonene vil bli absorbert i den første PN-overgangen og mesteparten av fotonene fortsetter til de resterende PN-overgangene. Strømtettheten blir derfor tilnærmet lik hele veien gjennom detektoren.

Tabell 1.1 viser responsen til de forskjellige PN-overgangene for lys med tre bølgelengder i prosent. Lys med 400 nm bølgelengde vil gi fra seg 92 % av energien i den første PN-overgangen, 8 % i den andre og nesten ingenting i den siste. Lys med 550 nm gir fra seg omtrent like mye energi i alle PN-overgangene. Grunnen til at responsen ikke synker innover i detektoren, er at PN-overgangene øker i størrelse. De stiplede vertikale linjene i figur 1.4 viser skillene mellom PN-overgangene. Lys med 400 nm bølgelengde vil derfor gi fra seg mer energi jo lenger inn i detektoren det kommer.

Krysskobling av signal

Ideelt skulle lys forårsake en strøm i bare én PN-overgang avhengig av bølgelengde. Dette er en årsak til krysskobling. En annen er at PN-overgangene er koblet sammen som vist i figur 1.5. Derfor vil en strøm i D_1 bidra til signal i både Amp_1 og Amp_2. En strøm i D_2 vil på samme måte bidra til signal i både Amp_2 og Amp_3. Dette betyr at koeffisientene i tabellen ovenfor får litt andre verdier. Som vi ser av figuren, blir forholdet mellom strøm i diodene og strøm inn på forsterkerene som følger:

$$\begin{aligned} I_{Amp_1} &= I_{D_1} \\ I_{Amp_2} &= I_{D_1} + I_{D_2} \\ I_{Amp_3} &= I_{D_2} + I_{D_3} \end{aligned} \tag{1.1}$$

Den nye tabellen blir da:

	400 nm	550 nm	650 nm
I_{Amp_1}	91.79	32.97	16.05
I_{Amp_2}	99.91	67.37	38.74
I_{Amp_3}	8.21	62.25	57.5

Matematisk kan dettte uttrykkes slik:

$$A\vec{l} = \vec{i}$$

hvor $\vec{l}$ er lyset inn satt sammen av flere forskjellige bølgelengder, A er overføringsmatrisen i tabellen over og $\vec{i}$ er signalstrømmen inn på de tre inngangene på skjemaet på figur 1.5.

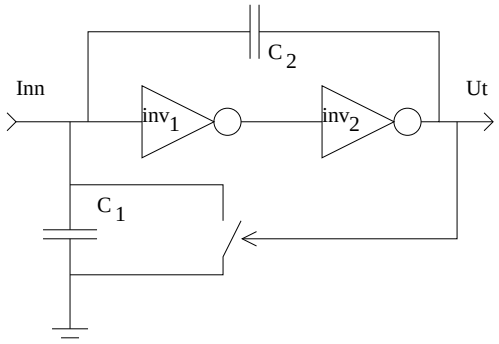
For å regne seg tilbake fra strømmen inn på inngangene til andelen lys i de forskjellige bølgelengder, inverterer vi tabellen og får:

$$\vec{l} \;=\; A^{-1}\vec{i}$$

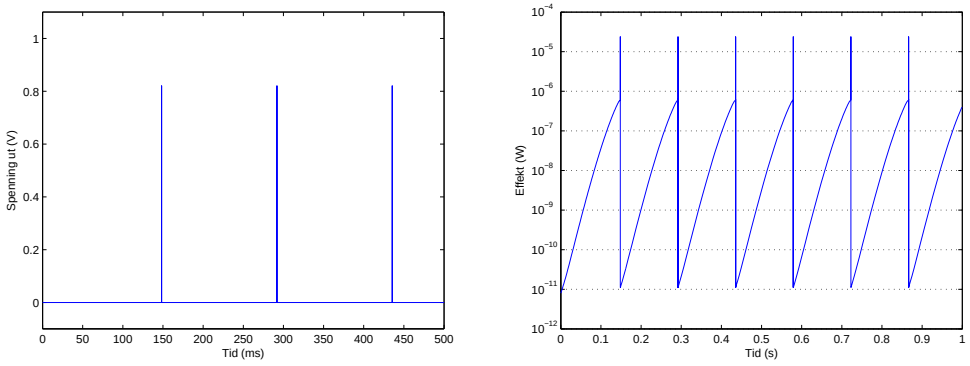
	I_{Amp_1}	I_{Amp_2}	I_{Amp_3}
400 nm	0.316	-0.194	0.042
550 nm	-1.172	1.112	-0.422
650 nm	1.224	-1.176	0.624

1.4 Carver & Mead integrate & fire nevron

Et Integrate & fire nevron integrerer en inngangsstrøm i en kondensator.



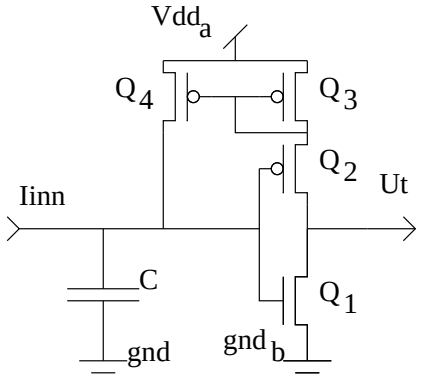
Figur 1.6: Carver Mead integrate & fire nevron



Figur 1.7: Signal ut fra og effekt omsatt i et originalt CM-nevron

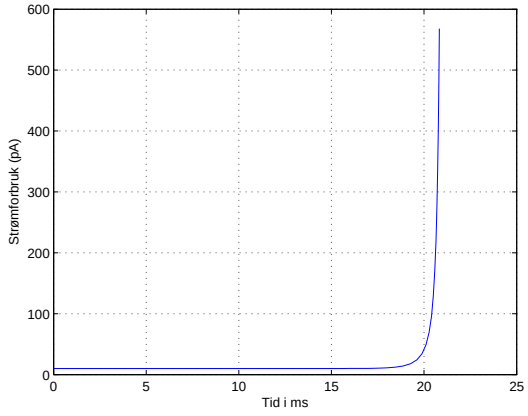
Når spenningen når et gitt nivå, gir nevronet ut en puls og kondensatoren lades ut. Intervallet fra reset til pulsen er da invers proporsjonalt med inngangsstrømmen. Skjematikk til et enkelt nevron ser vi i figur 1.6 av Carver Mead [2]. Signal ut ser vi til venstre i figur 1.7. Dette nevronet er enkelt og pålitelig, men bruker mye effekt som vi ser til høyre i samme figur. En sekvens kan begynne med at $V_{C_1} = V_{Ut} = 0$. En strøm inn (I_{in}) lader opp C_1 og C_2. Når V_{C_1} når switchepunktet til inverter inv_2, vil utgangen på inverter inv_1 gå fra '1' til '0' og inverter inv_2 gå fra '0' til '1'. Dette signalet sendes via C_2 tilbake til C_1. V_{C_1} gjør et hopp opp og en bryter, implementert v.h.a. en transistor, lader ut C_1 og C_2. Når V_{C_1} når switchepunktet til inverter inv_2, vil utgangene på inverterene og V_{Ut} bytte tilbake til utgangsposisjon. Igjen vil C_2 lede den denne gangen negative flanken tilbake til C_1. V_{C_1} gjør et hopp ned, og vi er tilbake til utgangsposisjon.

1.5 Current-feedback event generator

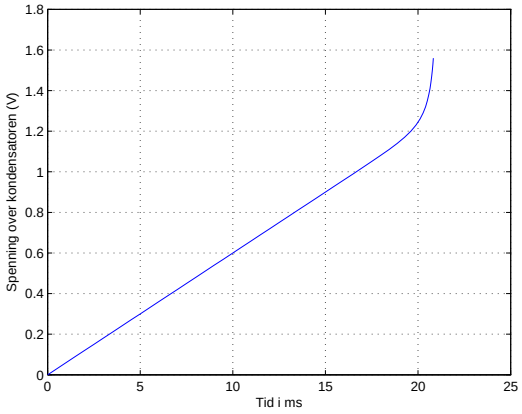


Figur 1.8: Forenklet utgave av "current-feedback event generator pixel"

Dette er beskrevet fullstendig i [3]. Her er en kortversjon. Figur 1.8 viser en forenklet utgave av "current-feedback event generator pixel". Den fungerer som følger: gnd_b har en spenning som er høyere enn gnd f.eks. 0.5 V. En sekvens starter med at V_C er 0 V. Q_1 får da en V_{gs} på -0.5 V og vil derfor ikke lede strøm. Q_2 vil lede, men strømmen blir 0 fordi Q_1 ikke leder. Strømspeilet Q_3 og Q_4 speiler strømmen som foreløpig er 0 tilbake til inngangen. V_{Ut} er lik Vdd_a. Når V_C blir høy nok d.v.s. at den nærmer seg $V_{gnd_b} + V_T$, begynner Q_1 å lede og strømmen blir speilet tilbake til inngangen. Dette fører til at V_C øker raskt, Q_2 slutter å lede og V_{Ut} blir lik V_{gnd_b}. Sekvensen består derfor av en lang periode hvor det nesten ikke går strøm og en kort periode hvor



Figur 1.9: Strømforbruket i et "current-feedback event generator pixel"

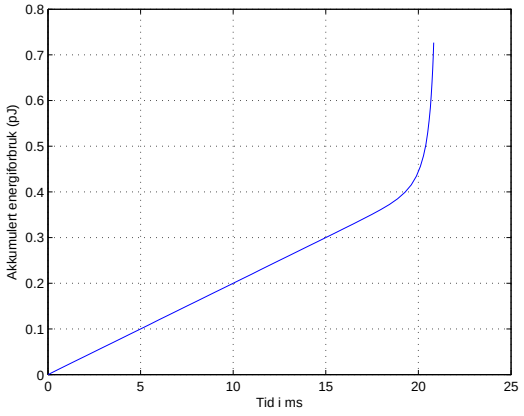


Figur 1.10: V_C i et "current-feedback event generator pixel"

det går strøm. Resultatet er et veldig lavt gjennomsnittlig strømforbruk og et utgangssignal med skarpe flanker.

Den forenklede kretsen er forenklet enda mer ved at Q_2 er erstattet av en kortslutning og Q_3,Q_4 er erstattet av et idéelt strømspeil. Kretsen er så simulert i Matlab og noen resultater er vist i figur 1.9 - 1.11.

$$\begin{aligned} V &\triangleq V_g - V_T \\ \dot{V_C} &\approx \frac{I_{in} + Ee^V}{C} \; V < 0 \end{aligned} \tag{1.2}$$

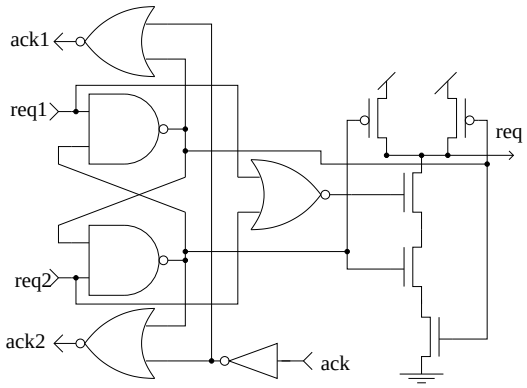


Figur 1.11: Akkumulert energiforbruk i et "current-feedback event generator pixel"

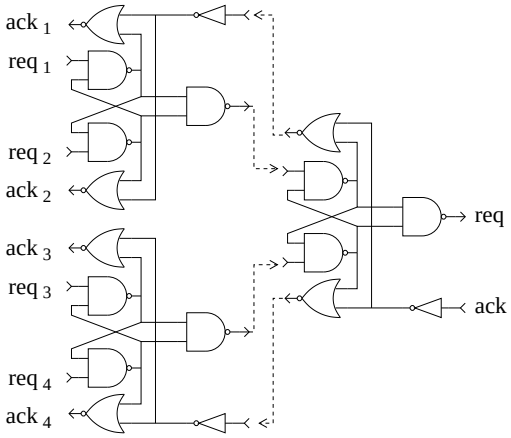
$$\dot{V_C} \approx \frac{I_{in} + EV^2}{C} \; V \geq 0 \tag{1.3}$$

Ligning 1.2 er en forenkling av differensialligningen for å regne ut V_C når V_C er under terskelspenningen til Q_1. Ligning 1.3 er en forenkling av differensialligningen for å regne ut V_C når V_C er over terskelspenningen til Q_1. Den er i prinsippet en tangensfunksjon.

1.6 Arbiter



Figur 1.12: Glitchfri arbitercelle med to innganger



Figur 1.13: Arbiter i kaskadekobling

Figur 1.12 viser en arbiter med to innganger og en kaskadeutgang. Den brukes på følgende måte: Når en eventgenerator genererer en event, sender den en forespørsel inn på sin req-inngang og venter på svar. Arbiteren sender forespørselen videre og venter også på svar. Når arbiteren får svar, sender den det videre til den av eventgeneratorene som sendte forespørsel først og eventuelt til den andre når den første har trukket tilbake forespørselen. Når ingen sender noen forespørsel til arbiteren, sender heller ikke arbiteren noen forespørsel videre. En komplett beskrivelse finnes i [9]. NAND-porten og de fem diskrete transistorene utgjør en NAND-port med glitch-sikring.

For å få en arbiter med flere innganger kaskadekobles de som i figur 1.13.

1.7 Pulsfrekvensmodulasjon og time-to-first

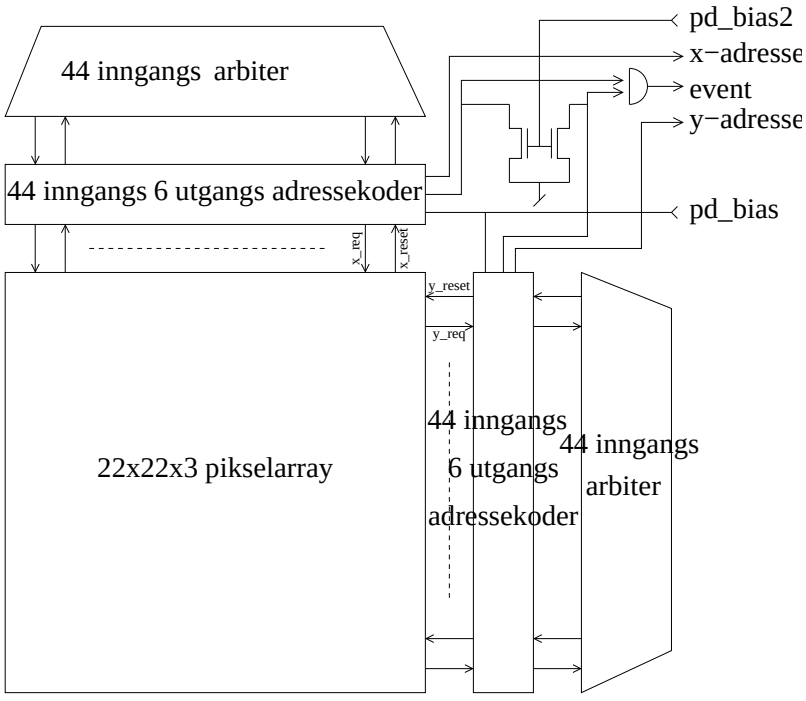
Ved pulsfrekvensmodulasjon ligger informasjonen i hvor hyppig pulsene blir sendt ut. Kombinert med AER betyr det hvor hyppig en adresse blir sendt. Jo oftere en adresse blir sendt, jo høyere verdi representeres i den gitte adressen.

Ved time-to-first modulasjon ligger informasjonen i tidsforsinkelsen fra reset til første event [6]. Jo kortere tidsforsinkelse, jo høyere verdi representeres i den gitte adressen.

Kapittel 2

Beskrivelse av designet

2.1 Oversikt over systemet

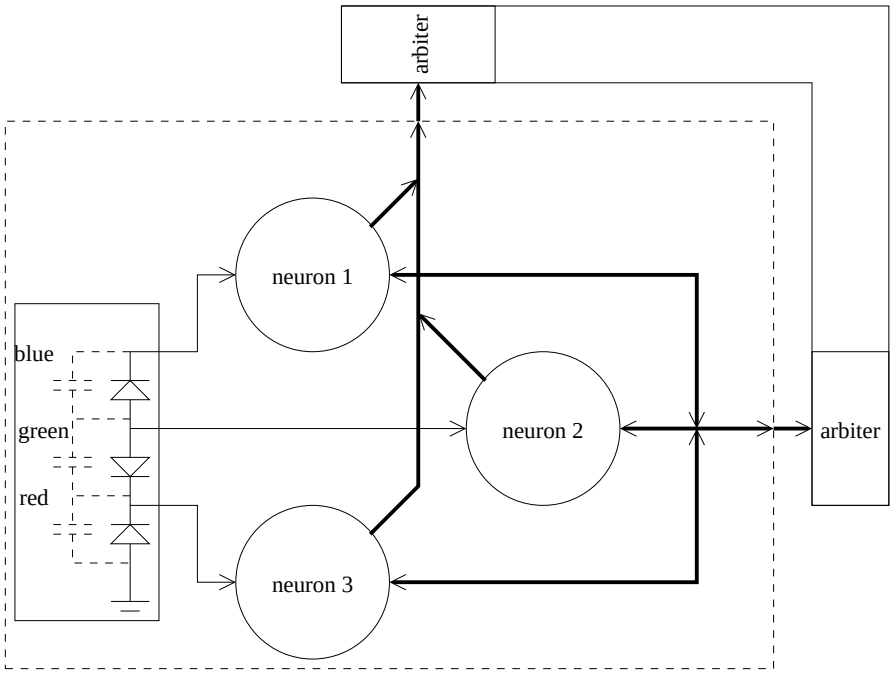


Figur 2.1: Oversikt over systemet

På figur 2.1 ser vi et blokkdiagram over systemet. Sentralt er en matrise med $22 * 22$ piksler hver med 1 fargesensorer og 3 eventgeneratorer. I pulstetthetsmodulasjonsmodus vil hver av de til sammen $22 * 22 * 3$

eventgeneratorene generere en jevn sekvens av eventer med en frekvens proporsjonal med lysstyrken i det aktuelle pikselet og avhengig av farge. I "time to first"-modus vil hver eventgenerator sende ut én enkelt puls en periode etter reset. Denne perioden er omvent proporsjonal med lysstyrken. Når et event genereres, sendes det en forespørsel til arbiteren. Arbiteren tar i mot forespørsler og skal sende ut kvitteringer i samme rekkefølge som forespørslene kommer inn etter at alle tidligere forespørslene fra andre eventgeneratorene er tatt hånd om. I praksis vil arbiteren prioritere forespørsler fra kanaler nær den som behandles hvis forespørsler kommer mer eller mindre samtidig. Hvis det for eksempel kommer forespørsler fra kanal 2 og 3 mens en forespørsel fra kanal 1 behandles, vil den fra kanal 2 behandles før den fra kanal 3 selv om den fra kanal 3 kommer først. Når en kvittering blir sendt fra arbiteren, setter adressekoderen opp en $2 * 6$ *bits* adresse avhengig av hvilken eventgenerator som fikk kvitteringssignal.

2.2 Et piksel

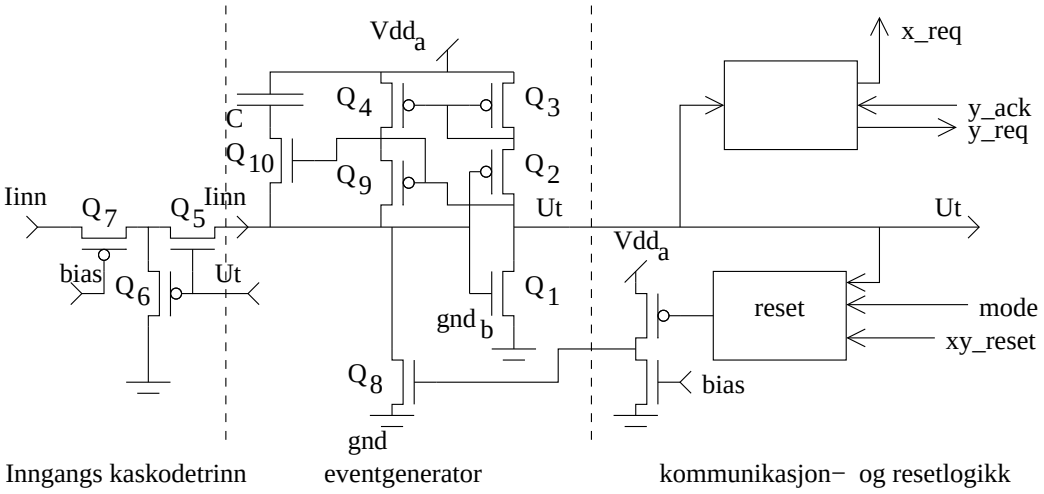


Figur 2.2: Blokkdiagram som viser et enkelt piksel

Figur 2.2 viser blokkdiagram av et piksel med en trippel stablet fotodiodesensor med parasittkapasitanser illustrert som stiplete kondensatorer til venstre

og tre nevroner.

2.3 Et nevron

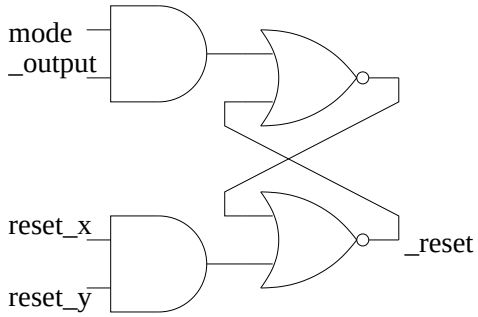


Figur 2.3: Blokkdiagram som viser et enkelt nevron

2.4 Tiltak mot krysskobling ved event

Når eventgeneratoren genererer en event, endrer spenningen på inngangen seg. I et monokromt piksel er ikke det noe problem. Der er det bare koblet én fotodiode til hver eventgenerator. I denne typen piksel derimot er det koblet fotodioder mellom de tre inngangene. Disse diodene har store parasittkapasitanser og spenningsvariasjoner smitter over fra en eventgenerator til en annen. Denne krysskoblingen er ikke lineær og kan derfor ikke kompenseres for i overføringsmatrisen. Krysskoblingen er avhengig av ladningen i C i eventgeneratoren som mottar krysskoblingen. Siden ladningen varierer over tid, vil krysskoblingen medføre temporær støy. Helt til venstre figur 2.3 er det satt inn tre transistorer som hindrer spenningsvariasjonene over C i å smitte over til de andre nevronene i samme pikselet. Q_7 er koblet som common gate med inngang på source. Dette gir lav inngangsimpedans og høy utgangsimpedans. Dette forutsetter at $V_{drain} \leq V_{inn} - V_{gt}$. En høy utgangsimpedans gjør at en gitt spenning over C forårsaker en liten strøm ut på inngangen. Lav inngangsimpedans reduserer spenningssvinget på inngangen og derved krysskoblingen.

2.5 Resetkrets



Figur 2.4: Resetkrets med to operasjonsmodi

Denne har til oppgave å gi kretsen "time-to-first"-modus ved at $\_reset$ forblir aktiv etter at $\_output$ er trukket tilbake. Den fungerer som følgende: En sekvens starter med at $reset_x$ og $reset_y$ er lave og $\_output$ er høy mens en kondensator C figur 2.3 integrerer opp strømmen inn. Når kondensatoren er tilstrekkelig oppladet og nevronet fyrer, går _output lav. Dette er utgangen fra nevronet. Når eventsignalet fra nevronet har kommet gjennom arbiternettverkene, først y-retning så x-retning, vil $reset_y$ og så $reset_x$ gå høy. $\_reset$ går lav og nevronet vil resettes. Når nevronet er resatt, går _output høy. Avhengig av nivået til mode skjer nå ett av to:

1. Hvis mode er høyt, vil $\_reset$ etter at enten $reset_x$ eller $reset_y$ forsvinner deaktiveres ved å gå høyt og nevronet initierer en ny sekvens.

2. Hvis mode er lavt, vil $\_reset$ ikke deaktiveres og nevronet holdes resatt inntil mode går høyt.

I prinsippet er resetkretsen en RS-vippe med to R-innganger og to S-innganger hvor begge R-inngangene må være aktive for at vippen skal resettes og begge S-inngangene må være aktive for at vippen skal settes.

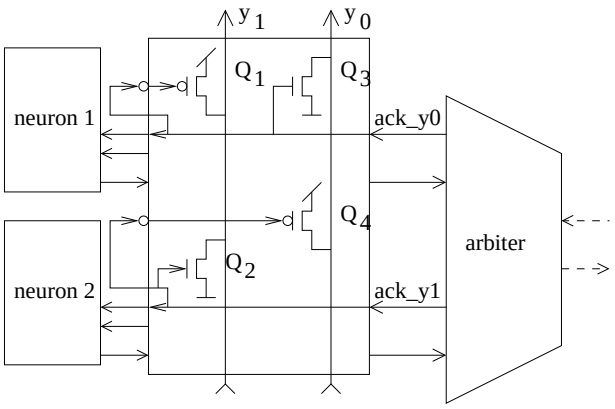
2.6 Ringing

Ringing som følge av at eventgeneratoren og arbiteren kommer i en ustabil tilstand hvor arbiteren sender resetsignal til eventgeneratoren. V_C lades delvis ut, nok til at arbiteren trekker tilbake eventforespørselen, men ikke nok til at strømtilbakekoblingen slutter å levere en høy strøm gjennom Q_4 se figur

1.8. P.g.a strømtilbakekoblingen vil eventgeneratoren umiddelbart sende ut en ny eventforespørsel som arbiteren godtar og kvitterer på med å sende resetsignal. Og slik står kretsen og ringer.

Da resetkretsen i avsnitt 2.5 ble sett inn, stoppet ringingen. Siden resetkretsen ikke skulle ha den funksjonen, er det sammsynligvis "pull down"-transistoren etter resetkretsen som ikke trekker ned fortere enn at transistoren som lader ut kondensatoren rekker å lade den helt ut.

2.7 Adressekoder

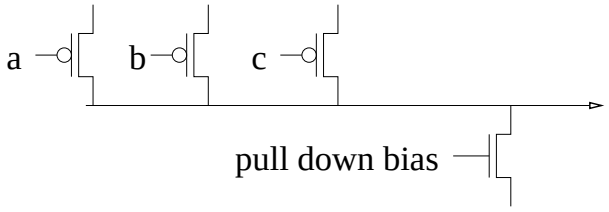


Figur 2.5: Adressekoder

Adressekoderen i figur 2.5 er et utsnitt av en komplett adressekoder med to acknowledge-signaler inn fra en to inngangs arbiter og to adressebit ut. Et system med to nevroner hadde selvfølgelig ikke trengt mer enn én adressebit ut, men her er det tatt med to for å illustrere hvordan ett acknowledge-signal genererer både 1-ere 0-er. Når arbiteren sender et acknowledge-signal til nevron 1 ved å legge ack_y_0 høy, vil Q_3 trekke y_0 til 0 og Q_1 vil trekke y_1 til 1. Når arbiteren sender et acknowledge-signal til nevron 2 ved å legge ack_y_1 høy, vil Q_4 trekke y_0 til 1 og Q_2 vil trekke y_1 til 0.

2.8 "Wired or"-logikk

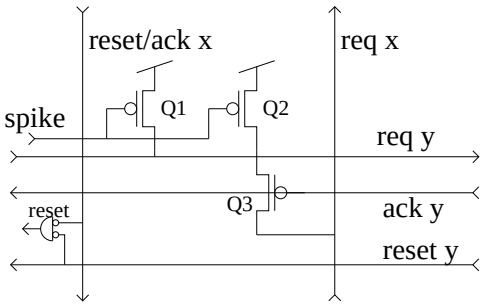
Dette er en meget enkel og arealbesparende måte for mange sendere å kommunisere med én mottaker. I dette eksempelet er det tre sendere. De senderne som skal gi ut 1 trekker den felles noden til 1. De som skal gi ut



Figur 2.6: Wired or logikk

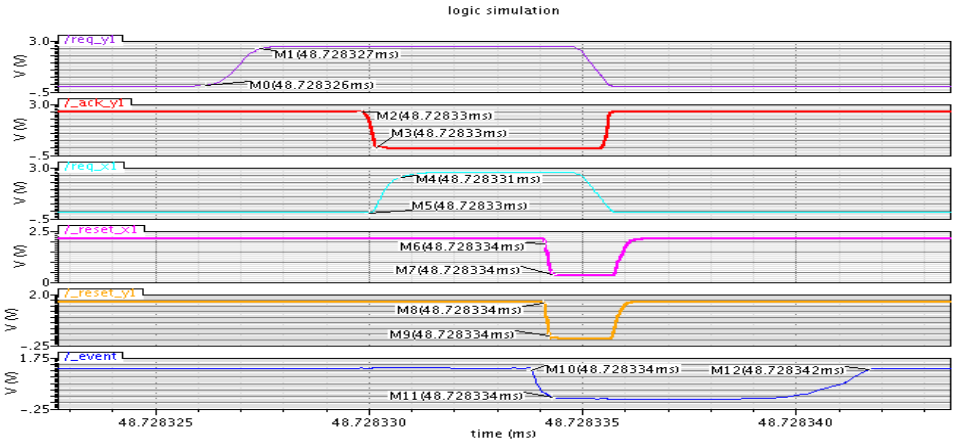
0 lar være. Inngangene er her aktiv lave så utgangen blir $\_a + \_b + \_c$. Som over alt ellers er det også her noen ulemper. Én er problemet med å identifisere avsenderen. Et annet problem er hastighet. Hver sender bidrar til kapasitansen på linjen med drain-kapasitansen i den transistoren som trekker linjen opp. Når antall sendere øker vil også den totale kapasitansen øke og hastigheten reduseres. Dette problemet er ikke trivielt å løse fordi den normale måten å løse denne typen problemer er å øke bredden på transistorene. Her vil ikke dette hjelpe fordi en økning av bredden på transistorene også øker drain-kapasitansen som jo var årsaken til problemet i første omgang. Identifisering av sender er beskrevet senere.

2.9 Kommunikasjon mellom piksel og arbiter



Figur 2.7: AER-read out-krets

Figur 2.7 viser den delen som formidler kommunikasjon mellom nevronet og arbiternettverkene, mens figur 2.8 viser en simulering av kommunikasjonen. Simuleringen er utført på skjematikk i som beskrevet i avsnitt 3.2. Noen av disse signalene er aktiv lave, andre aktiv høye og noen begge deler fordi de to typene nevroner krever forskjellig polaritet på reset-signalene. Når et nevron fyrer vil følgende skje: 1. Nevronet aktiverer spike-signalet ved å trekke det lavt. 2. Transistor Q_1 aktiverer req_y ved å trekke det høyt. Dette signalet



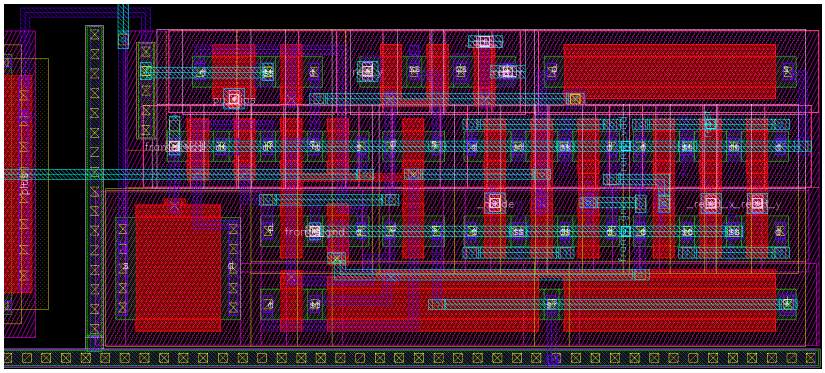
Figur 2.8: Plott av kommunikasjon mellom piksel og arbiter

er "wired or" sammen med alle andre piksler på samme rad ved at noden strekker seg over hele pikselmatrisens bredde. 3. Denne noden går inn på den ene arbiternettverket som kvitterer med å aktivere ack_y ved å trekke denne lavt straks alle tidligere eventer fra andre rader er håndtert. 4. Nå er både *spike* og ack_y lave og Q_2/Q_3 aktiverer req_x ved å trekke denne høy. Dette er et "wired or"-signal i en node som strekker seg over hele pikslematrisens høyde. 5. Når det andre arbiternettverket har håndtert alle tidligere eventforespørsler fra samme kolonne, sendes det et *event*-signal som forteller at en adresseevent er klar for eksport ut av chipen samtidig med at $reset_x$ og $reset_y$ aktiveres, nevronet resettes og sekvensen gjentas.

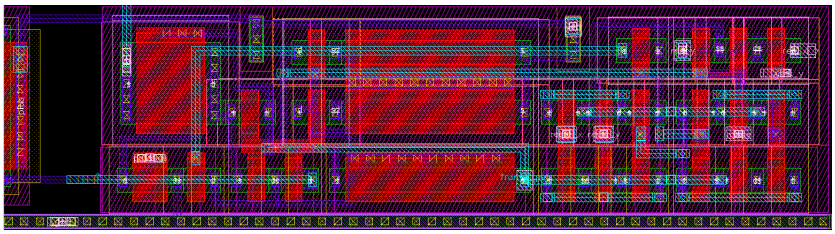
2.10 Utlegg

Figur 2.9 viser utlegg av hver av de nevronene som registrerer negativ strøm inn. Se også skjematikk i figur 1.5. Figur 2.10 viser utlegg av det nevronet som registrerer positiv strøm inn. Helt til venstre i figurene ser vi så vidt de store kondensatorene som akkumulerer strømmen inn. Disse fyller opp mye mer areal enn resten av nevronene. Derfor er ikke disse vist i sin helhet. De røde rektanglene er polysilisium og danner gate.

Det er to aspekter som er viktige å ta hensyn til i et slikt design. I alle analoge kretser er det viktig å minimere prosessvariasjoner. I kretser som inneholder både analoge og digitale signaler, er det viktig at de digitale signalene ikke smitter til de analoge. Digitale signaler har høy spenning og inneholder høyfrekvente komponenter. De smitter derfor lett ved kapasitiv



Figur 2.9: Utlegg av nevron 1 og 3

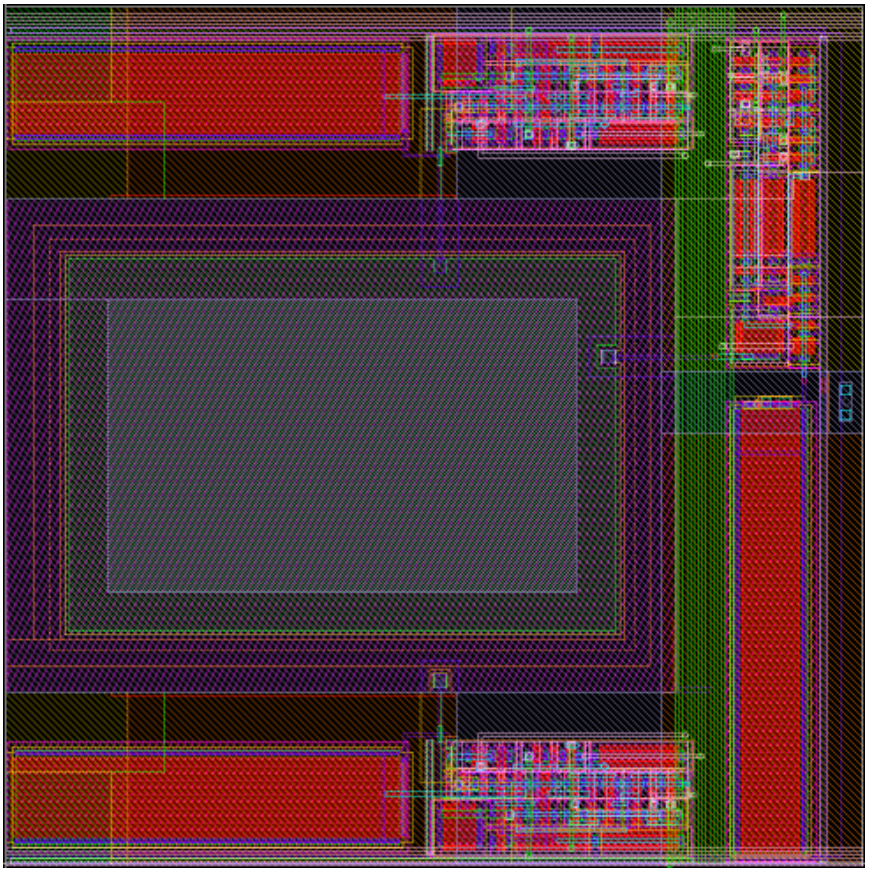


Figur 2.10: Utlegg av nevron 2

koblingsstøy. Dette er ivaretatt ved at de analoge signalene er plassert i den ene enden, d.v.s. lengst mulig til venstre. Det er ikke lagt noe mer vekt på det problemet fordi det eneste analoge signalet er avkoblet med en stor kondensator i hver ende, nemlig den tidligere nevnte ladekondensatoren i den ene enden og fotodiode i den andre. For å redusere prosessvariasjoner, er noen transistorer gjort ekstra lange og/eller brede. Tre av transistorene er ekstra lange. Den ene er den transistoren som den akkumulerte strømmen går inn på. Variasjoner i terskelspenningen her, vil direkte påvirke hvor lenge nevronet akkumulerer strøm inn før det fyrer. De to andre utgjør strømspeilet som speiler strømmen tilbake til inngangen. Strømspeil er generelt følsomme for mismatch. En mismatch her, vil også påvirke fyringstidspunktet.

Figur 2.11 viser et piksel. Den store rektangulære strukturen er de tre fotodiodene. Husk at de er stablet, så det meste som synes er den øverste dioden. Det er også mulig å se de tre tilkoblingspunktene med ledere til hvert sitt nevron. Nå ser man også de store kondensatorene. Vertikalt til høyre for fotodiodene går x-bussen illustrert i grønt. Y-bussen går dels helt i toppen og helt i bunnen.

Hele pikselet er 31.3 $\mu m * 31.3\ \mu m = 979.69\ \mu m^2$. Selve sensoren er 24.49 $\mu m * 18\ \mu m = 440.82\ \mu m^2$, eller 45 % av pikselet. Hullet i metallmasken

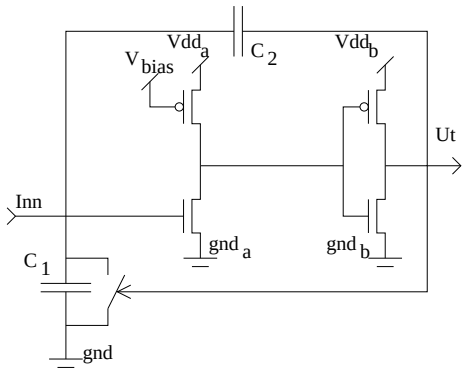


Figur 2.11: Utlegg av ett piksel(bildeelement)

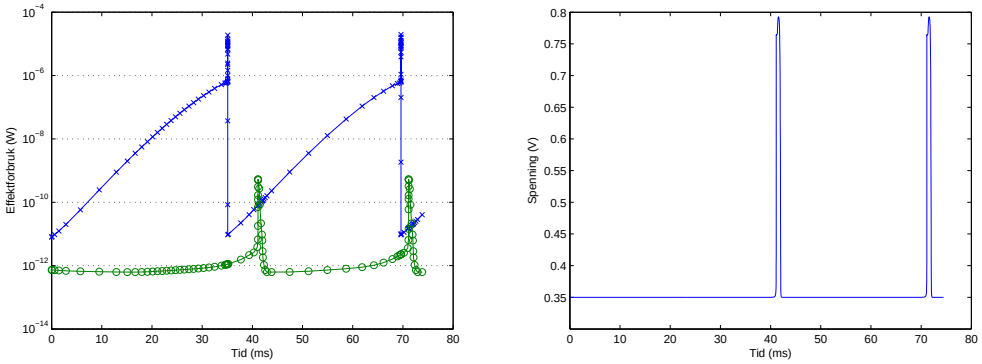
er 17.15 $\mu m * 10.735\ \mu m = 184\ \mu m^2$ eller 18.8 % av pikselet. Hvorvidt det er nødvendig at metallmasken dekker så store deler av selve sensoren, er heller tvilsomt. Den er der primært for å hindre metallpartikler fra bondeprosessen i å legge seg på og dermed kortslutte elektronikken. Masken skal også hindre lys fra å treffe den vertikale delene av PN-overgangene der de nærmer seg overflaten og derfor er mest følsomme for blått lys alle tre. Det hadde antageligvis holdt om masken gikk rundt katoden i den øverste dioden(n^+ *området*). Hullet ville da vært 19.995 μm*13.58 $\mu m = 271.53\ \mu m^2$ eller 27.72 % av pikselet.

2.11 Ikke implementerte varianter av nevronet

Én av ulempene med de forrige billedsensorene produsert ved denne forskningsgruppen, var at de trakk veldig mye strøm. En måte å redusere effektfor-



Figur 2.12: Modifisert utgave av samme nevron



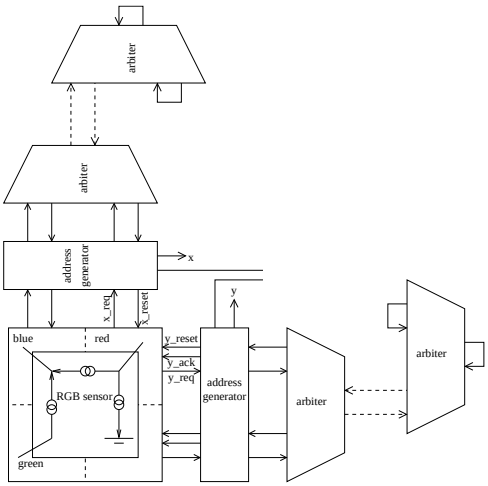
Figur 2.13: Sammenligning av effektforbruk i et tradisjonelt CM-nevron og et modifisert nevron til venstre og signal ut fra et modifisert CM-nevron til høyre

bruket, er å trimme tilførselsspenningene. Skjematikk til et modifisert Carver Mead (CM)nevron ser vi i figur 2.12 og signal ut fra dette i figur2.13. Til venstre i figur 2.13 er effektforbruket for begge plottet sammen for lettere sammenligning. Effektforbruket ser mye bedre ut, men pulsen ut er flere ms lang. Derfor ble ikke dette alternativet brukt.

Kapittel 3

Simuleringer

3.1 Simuleringsoppsett



Figur 3.1: Skjematikk for simulering

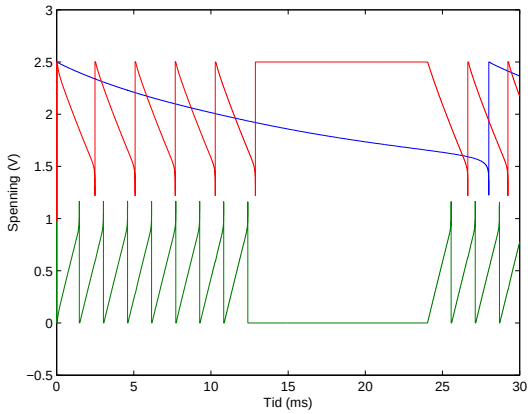
Simuleringsoppsettet er en utvidet utgave av oppsettet i [1]. Med i simuleringen er:

1. Ett piksel med tre nevroner
2. To adressegeneratorer: én i x- og én i y-retning. Dette gir to adressebit: x og y.
3. To arbitere som hver består av fem arbiterceller i serie. Dette for å få mer realistiske responstider. På figur 3.1 ser vi bare to celler i hver

arbiter. De tre siste skal være der de stiplede linjene er. I noen tidligere versjoner av nevronene var det nødvendig med et visst minimum av responstid for at nevronet skulle resettes ordentlig. I det implementerte nevronet er ikke dette nødvendig, men simuleringsoppsettet ble stående som det var fordi det fungerte.

For å undersøke om pikselet fungerer i begge operasjonsmodi og ved veksling mellom dem, ble pikselet simulert med en sekvens som består av 12 ms i kontinuerlig modus, 12 ms i "time to first"-modus og til slutt 6 ms i kontinuerlig modus når strømmen inn er satt til ca. 100 pA. Med 10 pA er tilsvarende tider 40, 40 og 20 ms. Simuleringen vil da vise ved hvilken frekvens de tre nevronene fyrer i kontinuerlig modus, om nevronene stopper i "time to first"-modus, og om de begynner å fyre igjen når operasjonsmodus settes tilbake til kontinuerlig.

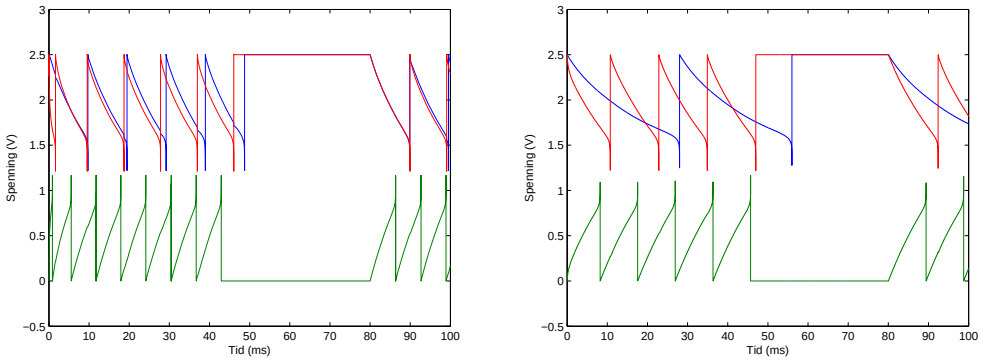
3.2 Simulering av et enkelt piksel, svak/middels belysning, hvitt lys



Figur 3.2: Plott av simulering med standard komponentverdier ved ca. 100 pA

I tillegg til selve pikselet, inngår signalveien til og fra pikselet. Dette vil vise et mest mulig nøyaktig tidsforløp. Chipen skal sende bildet til en PC gjennom en CPLD og en USB-kabel. Fordi datastrømmens hastighet er avhengig av belysning; svakere belysning - lavere hastighet, sterkere belysning - høyere hastighet, og datastrømmens hastighet er begrenset, må belysningen

begrenses. For at simuleringen skal være mest mulig realistisk, er da også pikselet simulert med strømmer i fotodiodene som tilsvarer svak belysning. Hvitt lys er simulert som tre fotonstrømmer på henholdsvis 400, 550 og 650 nm med like mange fotoner med hver bølgelengde. Dette blir ikke helt riktig. Det skulle ha vært like mye energi ved hver av tre bølgelengdene, men like mange fotoner blir nær nok fordi energien i fotonene bare varierer litt mer enn en fjerdedel. Figur 3.2 viser en simulering på utlegg av spenningen over kondensatoren C, se figur 1.8 i hver av de tre nevronene i et piksel med $I_{D1} = 141\ pA$, $I_{D2} = 65.2\ pA$ og $I_{D3} = 62.7\ pA$ med refeanse til figur 1.5. Denne strømmen regnes i denne sammenhengen som middels belysning. Den blå kurven viser spenningen over kondensatoren som akkumulerer strømmen i den første sensoren; den som hovedsakelig registrerer de blå komponentene i lyset. Den grønne viser tilsvarende for den andre sensoren som hovedsakelig registrerer de grønne komponentene i lyset. Tilsvarende for den røde kurven.



Figur 3.3: Plott av simulering på skjematikk og utlegg med standard komponentverdier ved ca. 10 pA

Figur 3.3 simulering på skjematikk til venstre og utlegg til høyre. Legg merke til den blå kurven som gjør et kraftig hopp idet det tredje nevronet, som hovedsakelig registrerer de røde komponentene i lyset, fyrer. Dette hoppet forekommer bare i simulering på skjematikk og ikke på utlegg.

Det skulle vært plott av en simulering av kommunikasjon mellom pikselet og arbiterene og effektbruket med 10 pA strøm inn, men simuleringsresultatet tok for mye plass til at programvaren klarte å plotte det.

3.3 Monte Carlo-simulering av et enkelt piksel, svak/middels belysning, hvitt lys

Statistiske simuleringer av nevroner i fire forskjellige utførelser med 10 pA, 100 pA og noen av dem med 1 nA gir følgende resultat: De tre først er simulert med det samme oppsettet som i [1]. Simuleringene er foretatt på utlegg for å få mest mulig realistiske resultater. En ulempe med å simulere på utlegg er at det tok over et døgn å simulere én sekvens med ett enkelt piksel og fordi simuleringstiden øker kvadratisk med kretsens størrelse, må pikslene simuleres ett og ett. Derfor ble hver variant av pikselet simulert med 40 kjøringer.

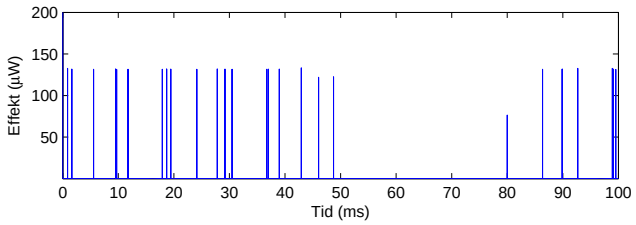
1. Et Carver Mead nevron med 10 pA fyrer 167 ganger/sekund med et standardavvik på 5. Samme nevron med 100 pA fyrer 1492 ganger/sekund med et standardavvik på 19.

2. Mitt nevron i to utgaver: ett for positiv strøm inn og ett for negativ strøm inn, implementert med minimumstransistorer og 10 pA fyrer 181/169 ganger/sekund med et standardavvik på 26/22. De samme nevronene med 100 pA fyrer 973/873 ganger/sekund med et standardavvik på 57/30 og med 1 nA: gjennomsnitt: 7966/6867 og standardavvik 252/216.

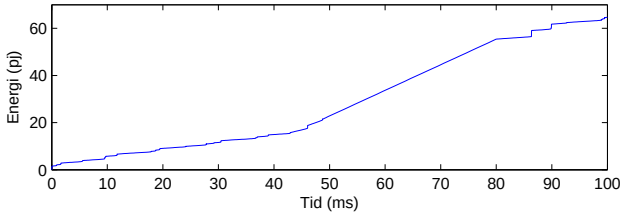
3. Mitt nevron hvor noen av transistorene er gjort 2.8μm i motsetning til 0.56μm tidligere for å redusere variasjoner mellom nevronene, lekkasjestrømmene og eventraten og dermed trafikken på bussen. Med 10 pA fyrer de 123/114 ganger/sekund med et standardavvik på 8/5. Med 100 pA fyrer de 722/667 ganger/sekund med et standardavvik på 29/9. Med 1 nA fyrer de 5942/5236 ganger/sekund med et standardavvik på 141/45.

4. Til slutt simulering på utlegg av mine nevroner. Dette er simulert med oppsettet i figur 3.1. Her har de tre nevronene forskjellige strømmer og det bli derfor tre resultater for hver simulert lysstyrke. Med 10 pA i hver av de tre PN-overgangene fyrer nevron 1: 35.6 ganger/sekund med et standardavvik på 3.1, nevron 2: 82 ganger/sekund med et standardavvik på 3.8 og nevron 3: 104 ganger/sekund med et standardavvik på 7.1. Med 100 pA i hver av de tre PN-overgangene fyrer nevron 1: 37.7 ganger/sekund med et standardavvik på 2.6, nevron 2: 625 ganger/sekund med et standardavvik på 16 og nevron 3: 384 ganger/sekund med et standardavvik på 9.8.

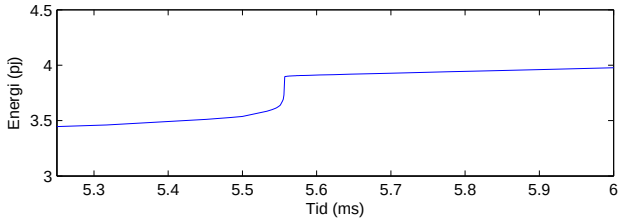
3.4 Effekt- og energiberegning i et piksel



Figur 3.4: Effektforbruk i et piksel



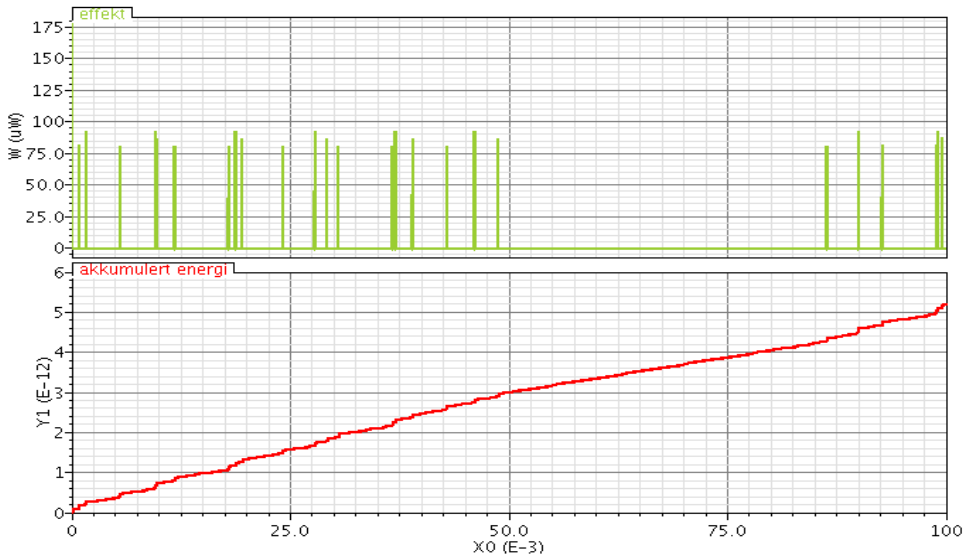
Figur 3.5: Energiforbruk i et piksel



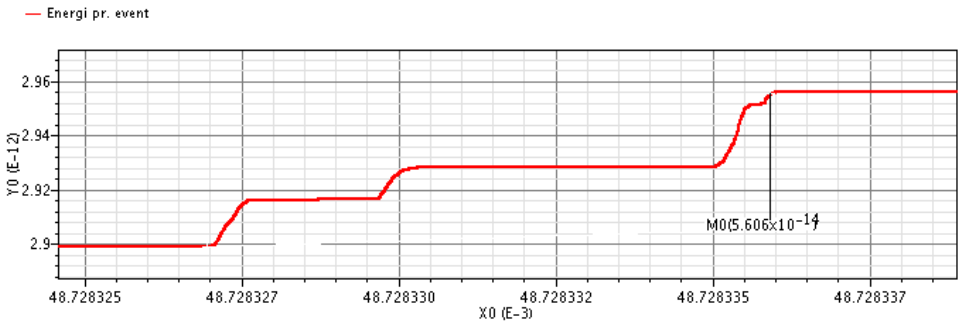
Figur 3.6: Energiforbruk for én event i et piksel

Denne simuleringen var det ikke mulig å få plottet etter å ha vært simulert på utlegg fordi resultatfilen fra simulatoren ble for stor og PCen sto og swappet i flere timer uten å produsere noe plott. Simuleringen er derfor foretatt på skjematikk. Som vi kan se av figur 3.4 er effektforbruk i korte perioder rundt 130 μW. Maksimalt effektforbruk ved en event er 133 μW. Mellom eventene er effektforbruket $150 - 160$ pW i kontinuerlig modus og 1086 pW etter eventene i "time-to-first"-modus. Grunnen til det høye effektforbruket i "time-to-first"-modus er "pull down"-transistoren Q_8, se figur 2.3, som trekker strøm hele den tiden resetkretsen er aktiv. Figur 3.5 viser det akkumulerte energiforbruket over en simulering på 100 ms i dette tilfellet 65 pj. Figur 3.6 viser et utsnitt av figur 3.6 zoomet inn til én enkelt event. Energiforbruket i et piksel er omtrent et 0.72 pj/event.

3.5 Effekt- og energiberegning i en arbiter



Figur 3.7: Effekt- og energiforbruk i en arbitercelle



Figur 3.8: Energiforbruk i en arbitercelle for én enkelt event

Selv om effektforbruket i en arbitercelle er mindre enn i et piksel, og det er færre arbitere enn piksler, er dette også beregnet. Simuleringen viser topper i effektforbruket på $75 - 100\ \mu W$. Mellom eventene er effektforbruket $35pW$. Figur 3.7 viser det akkumulerte energiforbruket over en simulering på 100 ms i dette tilfellet 5.2 pj. Figur 3.8 viser et utsnitt av den nederste kurven i figur 3.7 zoomet inn til én enkelt event. Energiforbruket i en arbitercelle er omtrent et 56 fj/event.

3.6 Beregning av effektforbruk for hele billedsensoren

Effektforbruket er beregnet utelukkende basert på simuleringer fordi det ikke ble tid til å rigge opp et måleoppsett som var nøyaktig nok. Denne kretsen består av $M * N$ piksler, $M - 1 + N - 1$ arbiterceller og $Mlog_2(2M) + Nlog_2(2N)$ adressekoderceller. Grunnen til at det er 2M og 2N er at hvert piksel har 2 x- og 2 y-adresser. Adressekodercellene er koblet slik at det aldri er mer enn én aktiv transistor som trekker i en node; enten mot V_{dd} eller mot gnd. Det vil aldri være en p-kanal- og en n-kanaltransistor i serie som er skrudd på samtidig og leder strøm fra V_{dd} til gnd. Det vil bare være lekkasjestrøm som bidrar til statisk strømforbruk og drain-kapasitans som bidrar til dynamisk strømforbruk. Dette burde vært simulert. Effektforbruket vil derfor domineres av piksler og arbiterceller. I denne type kretser skiller en som regel mellom det statiske og det dynamiske forbruket.

Det statisk effektforbruket utgjøres av summen av det statisk effektforbruket i alle komponentene. For en billedsensor med $22 * 22$ piksler blir: $M * N * 160\ pW + (M - 1 + N - 1) * 35\ pW = 78.9\ nW$

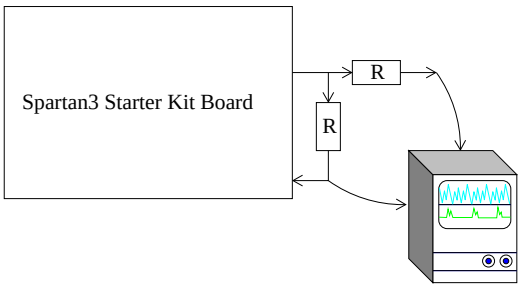
Det er bare de aktive komponentene som bidrar til det dynamiske effektforbruket. Det blir i dette tilfellet det pikselet som genererer eventen og de arbitercellene og adressekoderceller signalet går gjennom. Energiforbruk pr. event: $720\ fj + (\lceil log_2(2M) \rceil + \lceil log_2(2N) \rceil) * 56\ fj = 1.33\ pj$

Ved $4 * 10^5$ eventer pr. sekund(eps) blir det totale effektforbruket: $78.9\ nW + 1.33\ pj/event * 4 * 10^5\ eventer = 611\ nW$

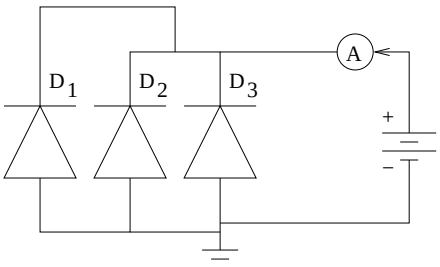
For at bergeningen skal bli lettere å sammenligne med billedsensorer med mer realistiske størrelser, er samme beregning gjort for en billedsensor med 128x128 piksler og 10 $Meps$. På det gamle printkortet klarer ikke kretsen 10 $Meps$, men den bør klare det med nytt printkort og en gjennomgang av kritiske signalveier. Statisk effektforbruk blir 2.63 μW og dynamisk blir 1.62 $pj/event$. Totalt 18.8 μW.

Kapittel 4

Målinger og måleoppsett



Figur 4.1: Måling av kapasitans i måleprobe og inngang på Spartan3



Figur 4.2: Måleoppsett for å teste alle fotodiodene på en gang

Først funksjonstestes kretsen for å se om den i det hele tatt virker. Deretter måles kvaliteten d.v.s. om alle nevronene fyrer, om nevroner av samme type fyrer like ofte ved lik belysning og høyeste/laveste eventrate.

Chipen ble designet for å kunne måle i tre nivåer;

1. Rett på de tre fotodiodene

   Figur 4.2 viser et meget enkelt måleoppsett. Første måling foregår med alle tre fotodiodene i parallell. Tilførselsspenning settes til 1.5 V og strømmen gjennom diodene måles med et ekstra følsomt amperémeter av typen Keithley 617. Dette instrumentet kan måle pA.

2. Ett piksel med tre fotodioder med eventgenerator og kommunikasjonslogikk

3. Hele billedsensoren med alle pikslene og adressegenerator

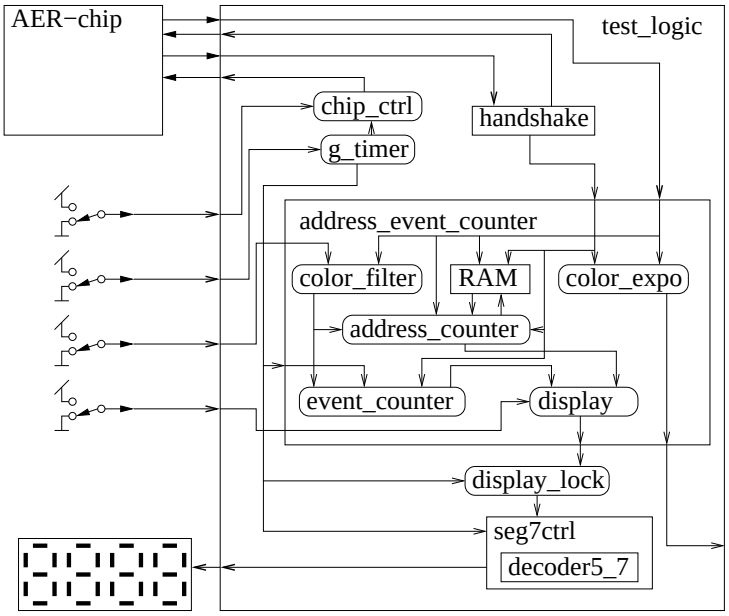
Måling 1: rett på de tre fotodiodene og 2: ett piksel med tre fotodioder med eventgenerator og kommunikasjonslogikk ble ikke foretatt på grunn av produksjonsfeil på chipen. Både det enkle pikselet med kommunikasjonslogikk og den ene bare fotosensoren var dekket av metall selv om det var spesifisert at det ikke skulle være det.

4.1 Måling av kapasitans i måleprobe og inngang på Spartan3

Motstanden R har en verdi på 21.5 $k\Omega$. Signalkilden var en digital utgang på Spartan3-kortet. Den har en stigetid på ca. 3 ns med måleprobe tilkoblet og mindre enn det uten. Måleproben var den som fulgte med oscilloskopet. Tiden ble målt fra utgangssignalet begynte å synke fra 3.3 V til den nådde $3.3\ V * \left(1 - \frac{1}{e}\right)$ d.v.s. en tidskonstant. Resultatene med h.h.v. fritthengende probe og probe+inngang tilkoblet var 242 ns og 440 ns. Se figur 4.1. Kapasitansene for proben og probe+inngangen blir derfor 11.25 pF og 20.5 pF. Inngangen utgjør differansen $20.5\ pF - 11.25\ pF = 9.25\ pF$. Proben er spesifisert til $6 - 15\ pF$, så måleresultatet virker rimelig.

4.2 Funksjonstesting av chipen

For å se om sensorenen sender ut data, bruker jeg et testoppsett vist i figur 4.3. Kortet er et Spartan-3 FPGA Starter Kit fra Digilent. Dette kortet inneholder bl.a. spenningsregulatorer, et bryterpanel med vippebrytere, et firesiffers sjusegmentdisplay, en Spartan-3 FPGA fra Xilinx



Figur 4.3: Testoppsett med Spartan 3-kort

og kantkontakter for tilkobling av periferalutstyr. Vippebryterene brukes til å velge testfunksjon. Sjusegmentdisplayet brukes til å vise måleresultater. Billedsensorbrikken settes på et egnet kretskort f.eks. det som er vist i figur A.1. Dette kortet har også kantkontakter og plugges rett inn i Spartan-3 kortet. Spartan-3 FPGA-en inneholder:

1. Kontrolllogikk med en 26 *bit* timer som resetter billedsensorbrikken med jevne mellomrom. Ved hjelp av en vippebryter, kan måleperiodens lengde velges til å være 2^{24} eller 2^{26} klokkeperioder. Dette tilsvarer h.h.v ca. 1/3 og 4/3 s. En annen vippebryter brukes til å velge om billedsensoren skal operere i "time to first" eller kontinuerlig modus.

2. Handshake-logikk som tar imot *request*-signal svarer med *acknowledge*-signal og sender et *event*-signal videre til telleverket.

3. Selve telleverket som inneholder:

   (a) Et fargefilter som slipper igjennom bare adresser med farger spesifisert ved hjelp av tre brytere på bryterpanelet.

   (b) Logikk som sender ut en puls med en klokkeperiodes lengde på én av tre utganger avhengig av fargen. De tre utgangene kan så kobles til et oscilloskop. Dette er gjort i avsnitt 4.3 på side 34.

(c) En eventteller som teller opp antall eventer.

(d) En adresseteller som teller opp antall unike adresser ved å sette et flagg for alle adresser som har kommet i en BRAM og telle opp bare de som ikke har kommet før.

(e) En multiplekser som velger hva som skal ut på displayet; antall eventer, antall adresser eller tidspunkt for "first spike".

4. Et register som holder på måleverdien fra forrige måleperiode så displayet ikke står og teller mens brukeren leser av måleresultatet.

5. En sjusegmentkontroller som sender binærverdiene fra registeret ut til sjusegmentdisplayet.

Tillegg C inneholder VHDL-kode brukt i Spartan3-kortet.

1. Alle eventer telles opp 1/3 sekund eller 4/3 sekund. Når sensoren er dekket til med f.eks. en musematte, viser displayet 0 eventer. I normal rombelysning, viser displayet 50-200 kiloeventer/sekund (kevnt/s).

2. Sensoren settes i "time-to-first"-modus for å se om den modusen virker og om alle nevronene fyrer. V.h.a. fargefilteret i FPGA-en og tre brytere kan de tre sensorfargene testes hver for seg. For alle de tre sensorfargene viste displayet 484 eventer. Siden billedsensoren består av 22∗22 piksler, forventer vi å få $22 * 22 = 484$ adresser for hver av de tre fargene.

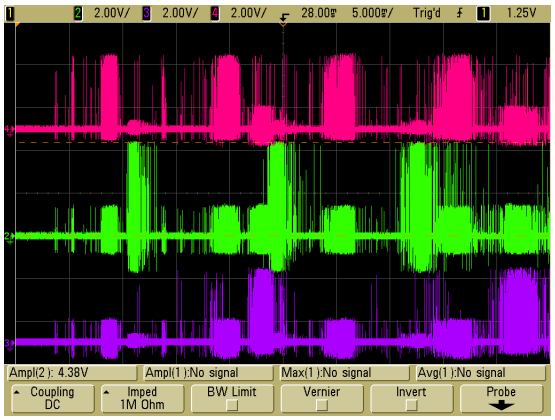
3. En variant av foregående test er å sette sensoren i kontinuerlig modus og telle opp unike adresser. Her benyttes også fargefilteret i FPGA-en. Opptelling fra de "blå" sensorene viser 484-485 unike adresser. Tilsvarende 484-490 for de "grønne" og 484 for de "røde". Grunnen til at det blir talt opp mer enn de forventede 484 adressene, er at adressebitene ut fra billedsensoren ikke har høy nok spenning. Det er ikke alltid at en '1'-er blir oppfattet som '1'. I adresserommet er det noen ledige adresser og det er noen av disse ledige adressene som blir registrert i stedenfor den riktige.

4. Fordeling av eventer fordelt på de tre fargene måles i kontinuerlig modus, med et fargefilteret implementert i FPGA-en. Opptelling fra de "blå" sensorene viser 24250-24530 eventer/sekund. Tilsvarende 23930-24280 for de "grønne" og rundt 52990 for de "røde". Tallet for de "røde" gikk sakte opp og ned, noe de andre ikke gjorde.

5. Samme test som 4. men med en leselampe satt helt nedtil sensoren gir følgende resultat: "Blå": 255-263 kevnt/s, "grønn": 296-301 kevnt/s og "rød": 421-423 kevnt/s. Tilsammen 1010 kevnt/s.

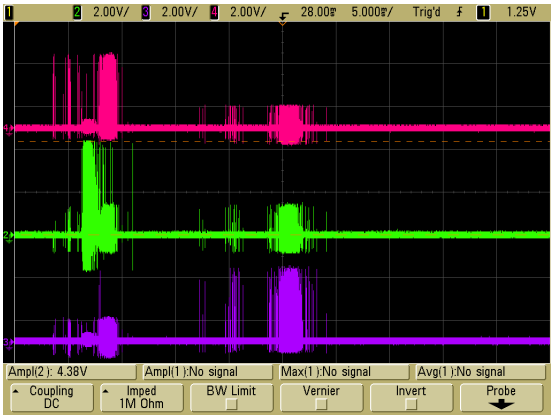
6. "Time-to-first": Sensoren settes i "Time-to-first"-modus. I stedenfor å telle opp antall eventer, måles tiden til halvparten av pikslene har fyrt med en stoppeklokke. Ved normal rombelysning, fåes følgende tider: "blå": 19.8-22.3 ms, "grønn": 15.5-21.1 ms og "rød": 8.6-11.4 ms. En leselampe satt helt nedtil sensoren gir følgende resultat: "blå": 1.5-2.6 ms, "grønn": 1.1-1.8 ms og "rød": 0.53-1.0 ms.

4.3 Illustrasjon av "time-to-first"-modus



Figur 4.4: Screen shot av oscilloskopet med sensoren i kontinuerlig modus

Figur 4.4 viser eventene fra de tre forskjellige nevronene i et oscilloskopbilde. Fortsatt er det testoppsettet fra funksjonstesten som brukes. Nå vil en event medføre en 50 ns puls ut fra én av tre utganger på Spartan3-kortet. En event fra et "rødt" nevron gir en puls på den ene utgangen, en event fra et "grønt" nevron gir en puls på den andre utgangen og en event fra et "blått" nevron gir en puls på den siste utgangen. Oscilloskopprobene er valgt slik at fargen på kurven på oscilloskopet korresponderer best mulig til fargen på nevronet. Vi ser at nesten alle de "røde" nevronene fyrer i en bolk og tilsvarende for de to andre fargene. Figur 4.5 viser det samme med unntak av at sensoren er satt i "time-to-first"-modus. Her ser vi én bolk med eventer fra hver av de tre fargene med noen få unntak. Vi ser også tydelig en krysskobling mellom de tre kanalene. Dette opptrer kun på utgangen av FPGA-en og skyldes ikke billedsensoren. I begge målingene er hele billedsensoren resatt 3 ms eller



Figur 4.5: Screen shot av oscilloskopet med sensoren i "time-to-first"-modus

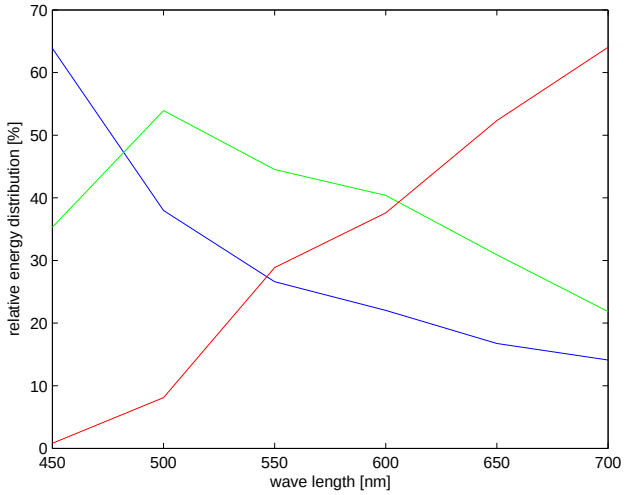
nesten én rute til venstre for bildets venstre ytterkant. Noe som ser litt pussig ut, er at den første bolken med eventer fra de "grønne" nevronene kommer tidligere i den andre målingen enn i den første. Dette skyldes antageligvis endring i lysforholdene d.v.s. forholdet mellom dagslys utenfra og lys fra lysstoffrørene i taket.

4.4 Måling av fargeseparasjon

	Blått nevron	Grønt nevron	Rødt nevron	Filterets båndbredde
400 nm	521.67	360.78	-	50
450 nm	118.62	86.67	344.06	80
500 nm	101.74	52.26	95.52	80
550 nm	34.41	16.26	16.47	80
600 nm	58.98	26.54	21.01	80
650 nm	79.30	35.55	18.74	80
700 nm	43.58	21.44	7.97	80
880 nm	58.41	163.18	7.52	50
950 nm	158.02	-	17.92	50

Tabell 4.1: Tid i ms til halvparten av nevronene har fyrt.

Fargeseparasjon måles ved å dekke til sensoren med en sort kloss. I klossen er det et hull og i hullet settes ett av ni optiske filtre med forskjellige bølgelengder. Gjennom filtrene sendes lys fra en bredspektret hallogénpære. Nevronaktiviteten er målt v.h.a. oppsettet i figur 4.4 og sensoren satt i "time-to-first"-modus. Grunnen til at denne modusen er valgt, er at i svak belysning



Figur 4.6: Relativ aktivitet pr. bølgelengde

fyrer nevronene i bolker og målingene viser i prinsippet bare antall bolker. Det blir for dårlig presisjon. Tabell 4.1 viser tiden fra reset til halvparten av nevronene av hver farge har fyrt. Båndbreddene er hentet fra [8]. Figur 4.6 viser aktivitetsfordeling mellom de tre typene nevroner normalisert for hver bølgelengde. Først regner man ut hvor mye av strømmen til hvert nevron som kommer fra hver av sensordiodene. Som vi ser av figur 1.5 vil strømmen fra D_1, kalt I_{D_1} registreres i både Amp_1 og Amp_2. På samme måte vil I_{D_2} registreres i både Amp_2 og Amp_3. (Samme ligningssett som 1.1)

$$\begin{aligned} I_{Amp_1} &= I_{D_1} \\ I_{Amp_2} &= I_{D_1} + I_{D_2} \\ I_{Amp_3} &= I_{D_2} + I_{D_3} \end{aligned}$$

I_{D_1}, I_{D_2} og I_{D_3} regnes ut som følger:

$$\begin{aligned} I_{D_1} &= I_{Amp_1} \\ I_{D_2} &= I_{Amp_2} - I_{Amp_2} \\ I_{D_3} &= I_{Amp_3} - I_{D_2} \end{aligned}$$

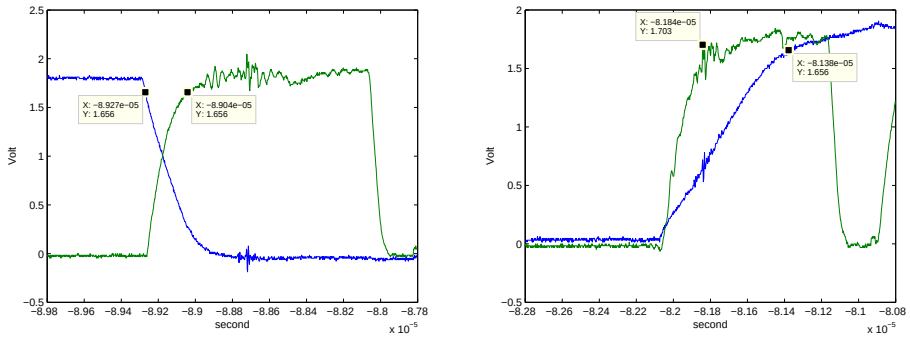
For hver bølgelengde og hver farge regnes det ut en verdi.

$$blå = \frac{I_{D_1}}{I_{D_1} + I_{D_2} + I_{D_3}}$$

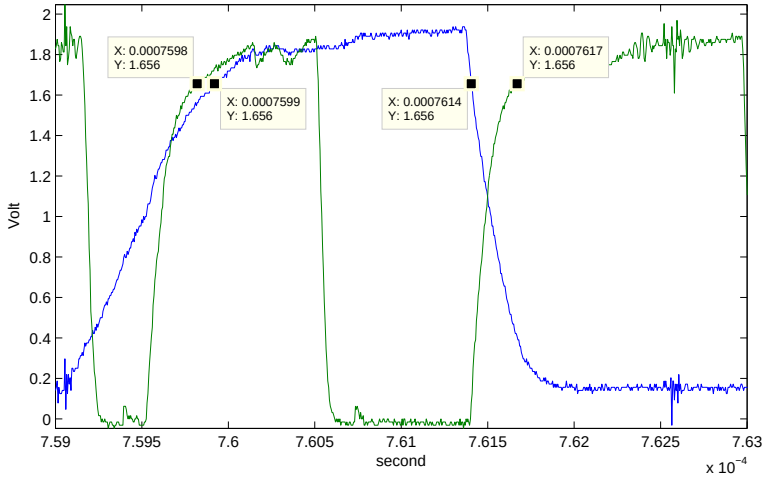
$$
\begin{aligned}
gr\phi nn &= \frac{I_{D_2}}{I_{D_1} + I_{D_2} + I_{D_3}} \\
r\phi d &= \frac{I_{D_3}}{I_{D_1} + I_{D_2} + I_{D_3}}
\end{aligned}
$$

Grafen er begrenset til det synlige området $450 - 700nm$.

4.5 Måling av signal- og spenningsnivåer



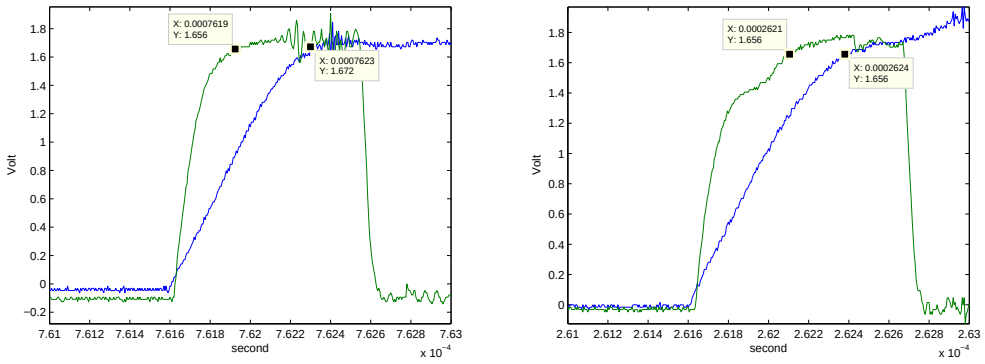
Figur 4.7: Fallende og stigende flanke på y_0 uten pullup



Figur 4.8: Stigende og fallende flanke på y_0 med pullup

I figurene 4.7 og 4.8 er antatt swtchepunkt for FPGA-en markert ved $Vdd/2 = 3.3\ V/2 = 1.65\ V$. Ved simulering på et enkelt piksel, får man

ikke noe inntrykk av hvordan signalene ser ut på utsiden av den ferdig produserte kretsen. Det er to grunner til det. Hele kretsen var for tung å simulere på. Simuleringen tok alt for lang tid og det var ikke tilgjengelig noen simuleringsmodell for tilkoblingspinnene på pakken. Derfor er det nødvendig og interessant å måle hvordan signalene ser ut i virkeligheten. I simuleringene genereres adressebitene før request-signalet som sier at en ny event er klar. Grafen til venstre i figur 4.7 viser at adressebit y_0, i grønt, passerer switchepunktet 230 ns før request-signalet, i blått, på vei fra 1 til 0. I grafen til høyre ser vi at adressebit y_0 passerer switchepunktet 460 ns etter request-signalet. Et tiltak for å prøve å kompensere for den lange stigetiden, er å sette på en pullup-motstand. på figur 4.8 er det satt på en pullup-motstand. Her ser vi en klar forbedring i det y_0 passerer switchepunktet bare 0.1 μs etter request-signalet på vei opp og 0.3 μs før på vei ned.

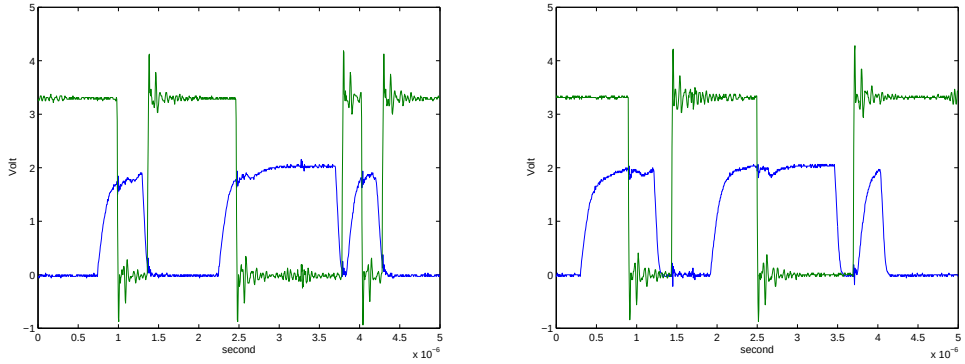


Figur 4.9: Stigende flanke på y_0 med bias på h.h.v. 550mV og 1.4V

Et annet tiltak kan være å bremse request-signalet litt på vei opp. Dette gjøres ved å øke en av bias-spenningene. Til venstre i figur 4.9 viser signalforløpet for y_0 og request-signalet med bias-spenningen satt til 550 mV. Til høyre i samme figur er bias-spenningen satt til $1.4V$. y_0 kommer fortsatt lenge etter request-signalet. Ved å sette bias-spenningen høyere slutter kretsen å fungere.

4.6 Glitcher (falske eventer)

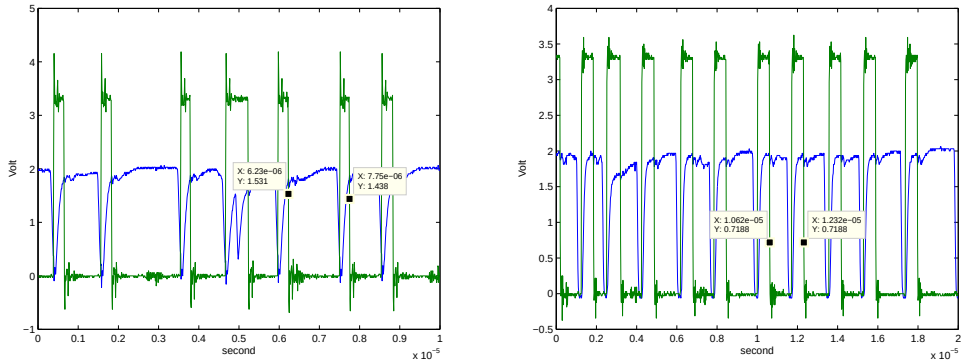
I figurene 4.10 ser vi praktisk talt det samme forløpet. Den blå kurven viser et aktivt høyt request-signalet fra billedsensorer mens den grønne viser et aktive lavt acknowledge-signal fra Spartan3-kortet. I det første forsøket svarer Spartan3-kortet umiddelbart med et lavt acknowledge-signal når det registrerer at request-signalet har blitt høyt. I det andre er Spartan3-kortet



Figur 4.10: Sekvens uten glitchfilter til venstre og med til høyre

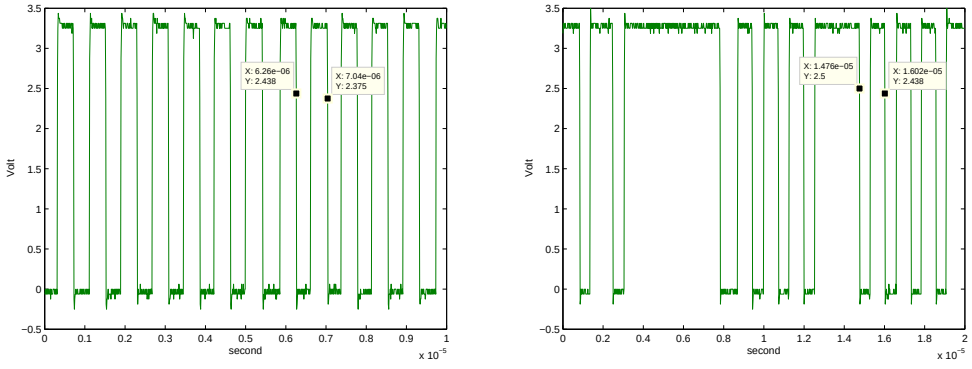
programmert til å vente 0.4 μs med å svare. Til høyre i figuren ser vi at request-signalet blir trukket tilbake uten å ha fått noe acknowledge. Dette er ikke i samsvar med kommunikasjonsprotokollen og kretsen fungerer tilsynelatende etter hensikten hvis signalet i dette tilfellet blir ignorert.

4.7 Trimming av biasspenninger



Figur 4.11: Stresstest uten trimmede biaser uten/med glitchfilter

Da billedsensoren ble satt i sokkelen og koblet til første gang, ble gnd_b satt til 0.5 V og Vdd_a satt til $Vdd - 0.5\ V = 2.0\ V$. Se figur 2.3. Alle biasspenninger unntatt den i kaskodetrinnet på inngangen ble tilsvarende satt til 0.6 V fra h.h.v. gnd og Vdd. Biasspenningene på kaskodeinngangene er koblet til h.h.v. gnd_b og Vdd_a og kan derfor ikke justeres separat. Spenningen pd_bias styrer



Figur 4.12: Stresstest med trimmede biaser og kort/langt glitchfilter

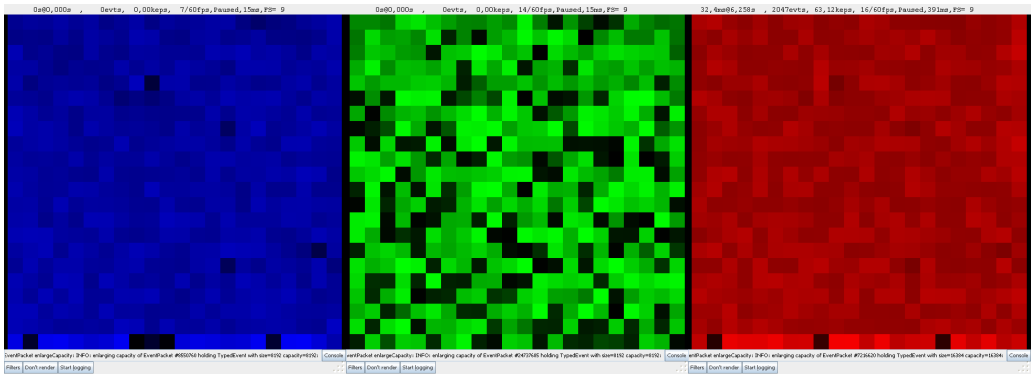
pull-down-transistorene i kommunikasjonen fra pikslene til arbiterne. Denne ble justert til 560 mV. Spenningen pd_bias2 styrer pull-down-transistorene i kommunikasjonen fra arbiterne og ut. Denne ble justert til 700 mV.

Pull-down-spenningen inne i det ene pikselet ble justert til 1.88 V; altså uforandret. Se figur 2.1. Pull-up-spenningen i de to andre nevronene ble satt til 350 mV; ikke så rart når man tenker på at en N-kanal transistor trenger lavere gate-source-spenning for å trekke like mye strøm som en tilsvarende P-kanal transistor. P- og N-kanal transistorene ble mot bedre vitende dimensjonert likt.

Lengden på glitchene ble også forandret. Før trimming var en stor del av glitchene på 31 klokkeperioder. Etter trimming er den lengste på 13 klokkeperioder.

Oscilloskopet har en funksjon for å telle opp eventer og regne ut en maksimal eventrate over en kort periode. Før trimming var maksimal eventrate 550 $keps$ og etterpå 2.2 $Meps$. Eventene på figur 4.12 varer kortere og med mindre variasjon i tidsforbruket enn de i figur 4.11. Dette er en fordel både fordi mer lys gjør at det blir mindre støy i bildet og for å unngå lange køer av eventer, da dette også medfører støy i bildet fordi mye av informasjonen i en AER-protokoll ligger i tidspunktet for eventene.

4.8 Analyse av loggede data



Figur 4.13: Bilde av det "blå", "grønne" og "røde" fargeplanet

I figur 4.13 er data fra billedsensoren sendt til en PC via en USB-adapter til et program som lager og viser fram bildet. USB-adapter og program er lagd til jAER-prosjektet. Igjen er fargefilteret i FPGA-en tatt i bruk for å splitte bildet i tre fargeplan.

4.9 Beregning av effektforbruk ved kommunikasjon ut av kretsen

Når vi beregner effektforbruket ut av kretsen, finner vi ut hvor mange utganger som endrer verdi fra 0 til 1 og multipliserer med kapasitans og speningsøkningen fra 0 til 1. Siden nevronene fyrer uavhengig av hverandre, forutsetter vi at det ikke er noen sammenheng mellom adressebitene fra én event til den neste. 1/4 av adressebitene vil endre verdi fra 0 til 1, 1/4 vil endre verdi fra 1 til 0, 1/4 vil forbli 0 og 1/4 vil forbli 1. De utgangene som forblir 0 eller 1 eller går fra 1 til 0 bruker ikke energi, mens handshakebitet endrer verdi hver gang. Kapasitansen på inngangene til FPGA-en er målt til 9.25 pF i avsnitt . V_{ut} er målt til 2 V, se figur 4.11. Med 12 adressebit og 1 handshakebit, blir $E = N * C * V_{ut} = \left(\frac{1}{4} * 12 + 1\right) * 9.25\ pF * 2\ V = 74\ pj/event$.

Kapittel 5

Diskusjon

5.1 Effektforbruk

Effektforbruket internt i billedsensoren er beskjedent, men kommunikasjon ut av kretsen krever mye energi, i følge beregningene over 50 ganger så mye. Skulle det vært mulig å måle effektforbruket internt i billedsensoren, måtte den hatt utgangsbuffere med separat tilførselspenning. Noe bør gjøres for å redusere maksimalt antall eventer ut av billedsensoren. Én måte å gjøre det på, er å gjøre eventer pr. sekund som funksjon av lysstyrke til en logaritmisk funksjon i stedenfor lineær. Dette krever flere transistorer i den analoge inngangskretsen, noe som alltid medfører fare for mye støy. En annen måte er å ha en frekvensdeler i hvert piksel med en konfigurerbar frekvensdelerverdi. Dette krever kommunikasjon inn i billedsensoren. I denne sensoren kan man bruke "time-to-first"-modus for å redusere effektforbruket ved å la hvert nevron fyre bare én gang hver gang bildet skal oppdateres. En annen metode er å la bare de sterkest belyste pikslene fyre og så resette hele billedsensoren. På denne måten kan en oppnå både høy billedoppfriskningsrate og lavt effektforbruk.

5.2 Sammenligning med lignende billedsensor

I tabell 5.1 er noen vitale data for denne billedsensoren sammenlignet med en annen nevromorf billedsensor.

	Denne billedsensoren	TEMPDIFF128
Funksjon		
Pikselstørrelse	$31.3 * 31.3\ \mu m$	$40 * 40\ \mu m$
Fillfaktor	18.8 %	8.1 %
Fabrikasjonsprosess	STM 90 nm	$4M\ 2P$ 0.35 μm
Pikselkompleksitet	87 transistorer (42 analoge) 3 kondensatorer	26 transistorer, (14 analoge) 3 kondensatorer
Arraystørrelse	$22 * 22$	$128 * 128$
Brikkestørrelse	$1 * 1\ mm$	$6 * 6.3\ mm$
Interface	12-bit word-parallell AER	15-bit word-parallell AER
Effektforbruk	Beregnet 611 nW 18.8 μW ved $128 * 128$ piksler	30 mW

Tabell 5.1: Sammenligning med en lignende billedsensor

5.3 Lyssensorer med en, to eller tre PN-overganger

Et viktig parameter for billedsensorer er areal pr. piksel. Dette bestemmer forholdet mellom oppløsning og pris. Arealet er også avgjørende for den optiske kvaliteten. Minimum areal[1] for lyssensor med en PN-overgang er $0.35\ \mu m * 0.35\ \mu m$ for det n^+ dopede området og 0.24 μm avstand mellom sensorene. Totalt minimum arealforbruk blir 0.348 μm^2. For en lyssensor med to PN-overganger er minimum areal $1.245\ \mu m * 1.245\ \mu m$ for det n^- brønnen og 1 μm avstand mellom pikslene. Totalt 5.04 μm^2 eller 14.5 ganger så stort som for en lyssensor med en PN-overgang. For en lyssensor med tre PN-overganger er minimum areal $6.0\ \mu m * 6.0\ \mu m$ for den dype n^- brønnen med n^- brønn langs ytterkanten og 3.3 μm avstand mellom pikslene. Totalt 86.49 μm^2 eller 17 ganger så stort som for lyssensor med to PN-overganger. Når vi tar hensyn til at vi må bruke fire ganger så mange sensorer pr. piksel med lyssensorer med en eller to PN-overganger som vi må med en sensor med tre PN-overganger, blir minimum areal for en lyssensor med tre PN-overganger bare drøyt fire ganger arealet ved bruk av lyssensor med to PN-overganger. Foveon har brukt lyssensorer med tre PN-overganger på 5.0 μm i sitt kamera modell F19 [10] noe som er mindre enn minimum for den prosessen brukt her, så Foveon har etter all sannsynlighet brukt en annen prosess.

[1] Areal og avstander er for en $90nm$ process fra ST Microelectronics

Temporær støy som består av termisk støy [11], shot-noise og kT/C-støy, og statisk støy p.g.a. lokale prosessvariasjoner er avhengig av arealet pr. piksel. Fordi fotonene kommer inn ukorrelert, vil de ha en Poisson-fordeling [<statistikkbok>]. Variansen i antall fotoner vil være lik antall fotoner og signal/støyforhold vil være proporsjonalt med kvadratroten av sensorens størrelse. Ved en dobling av sensorens størrelse vil signalstyrken øke 6 dB og signal/støyforholdet øke med halvparten d.v.s. 3 dB.

Kapittel 6

Avslutningen

6.1 Konklusjon

Det er mulig og relativt enkelt å lage en laveffekt AER-fargebilledsensor ved hjelp av stablede fotodioder og nevroner med strømtilbakekobling i standard 90 nm CMOS teknologi. Den er robust nok til å fungere uten noen initiell prøving og feiling med bias- og tilførselspenninger.

Stablede fotodioder ser ut til å være en god idé i teorien på grunn av arealbesparelsen ved å bygge i høyden istedenfor i bredden. Dessverre er presisjonen i produksjonen av den dype n-brønnen så dårlig at én stablet fotosensor bruker større areal enn de tradisjonelle fire tilsammen. Den lave prisen gjør at denne teknikken likevel kan brukes i spesielle applikasjoner.

6.2 Videre arbeid

En enkel forbedring av dette designet er å bruke pull-upmotstander og nivåkonvertere på alle utganger. Dette vil sikre at alle eventene blir registrert med riktig adresse. Dette er det satt av plass til på det nye kretskortet på figur A.1.

Ved å bruke selvresettende eventgeneratorer i kombinasjon med binærtellere i pikslene, kan størrelsen på kondensatorene reduseres og tellesekvensens lengde kan reguleres i forhold til belysningen. Ved mye lys, kan telleren telle opp mange eventer før en event sendes ut av pikselet.

En annen måte å separere farger, kunne være å detektere enkeltfotoner og klassifisere dem etter energinivået. Dette har et stort potensiale, men er

ikke nødvendigvis hverken praktisk eller mulig. Jeg har ikke undersøkt om noen har forsket på det tidligere. Klassfisering på denne måten blir brukt på gammastråler. Utfordringen er at mens et gammafoton ligger i keV og MeV området, f.eks. røntgen fra bly på 75 keV, befinner synlig lys seg i overkant av 1 eV.

Signalet fra denne chipen kan tenkes å bli sendt til forskjellige filtre for bildebehandling eller kobles til huden for å gi synet tilbake til ikke seende.

Måleresultatene kan brukes til å beregne egenskapene til den dype n-brønnen. Dette var det et ønske om, men tiden strakk ikke til.

Jeg har tenkt å undersøke om en current-feedback event generator kan brukes til å forbedre kvaliteten på digitale signaler der hvor wired-or-logikk er brukt. Typiske eksempler på applikasjoner er billedsensorer, RAM og andre kretser som består av store arrayer med celler som skal levere data ut. Kondensatoren i figur 1.8 modellerer parasittkapasitansen i transistorene som er med i wired-or-logikken og I_{inn} modellerer den ene transistoren som skal trekke noden fra 0 til 1.

For å demonstrere billedsensoren, er det mulig å koble til et kort med en 3x8 bit videoDAC og VGA-kontakt, skrive en skjermdriver i VHDL og dermed vise bildet fra billedsensoren på skjerm uten å gå veien om en PC.

Bibliografi

[1] Olsson, Jenny Anna Maria. Noise reduction in retinomorphic photo circuits. Master Thesis, 2007

[2] C. Mead. Analog VLSI and Neural Systems. Addison Wesley, 1989.

[3] Eugenio Culurciello, Ralph Etienne-Cummings, and Kwabena A. Boahen. A Biomorphic Digital Image Sensor. IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 38, NO. 2, FEBRUARY 2003

[4] R. F. Lyon and P. M. Hubel. Eyeing the camera: Into the next century. In IS&T/SID 10th Color Imaging Conf., pages 349355, 2002.

[5] Foveons hjemmeside, http://www.foveon.com/article.php?a=67, besøkt 23 august 2010.

[6] X. Qi, X. Guo, and J.G. Harris. A time-to-first spike CMOS imager. In Proceedings of the IEEE International Symposium on Circuits and Systems, volume 4, pages 824827, May 2004.

[7] Alf Olsen, CMOS Image Sensors, pages 2.13-2.33, spring term 2009.

[8] Bardello, J. A. L. , report, 2008.

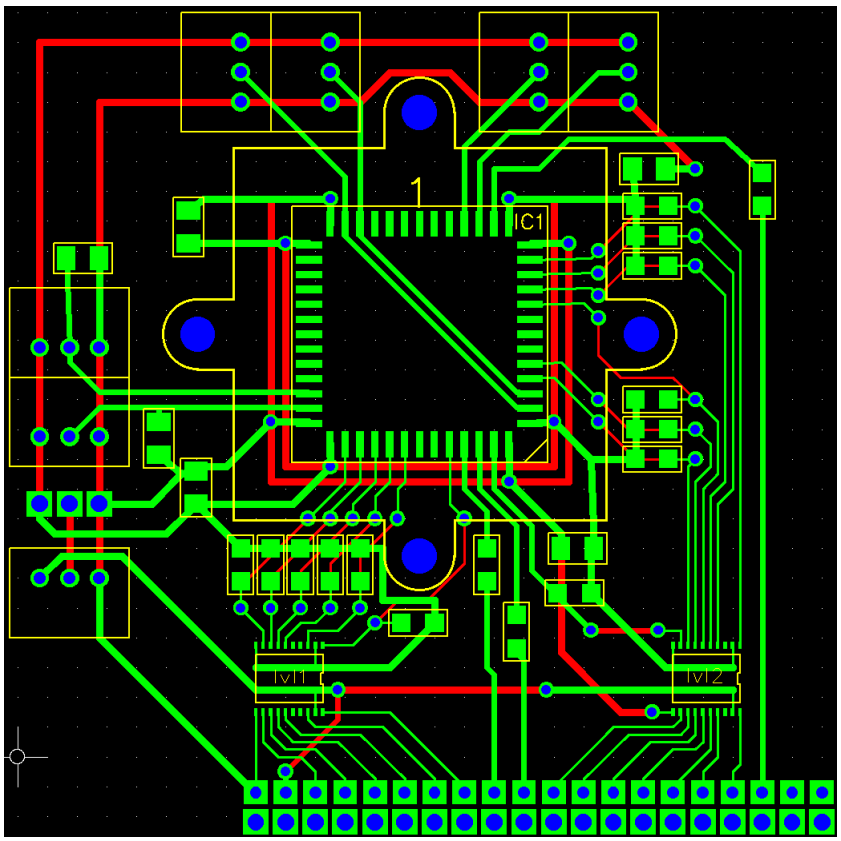
[9] P. Häfliger: Neuromorphic Electronics Lecture Notes INF5470, 2010

[10] Rudolph J. Guttosch, Investigation of Color Aliasing of High Spatial Frequencies and Edges for Bayer-Pattern Sensors and Foveon X3 Direct Image Sensors, 2005

[11] C.D. Motchenbacher: Low-Noise Electronic System Design, John Wiley & Sons, 1993.

Tillegg A

Kretskort for testing av chipen

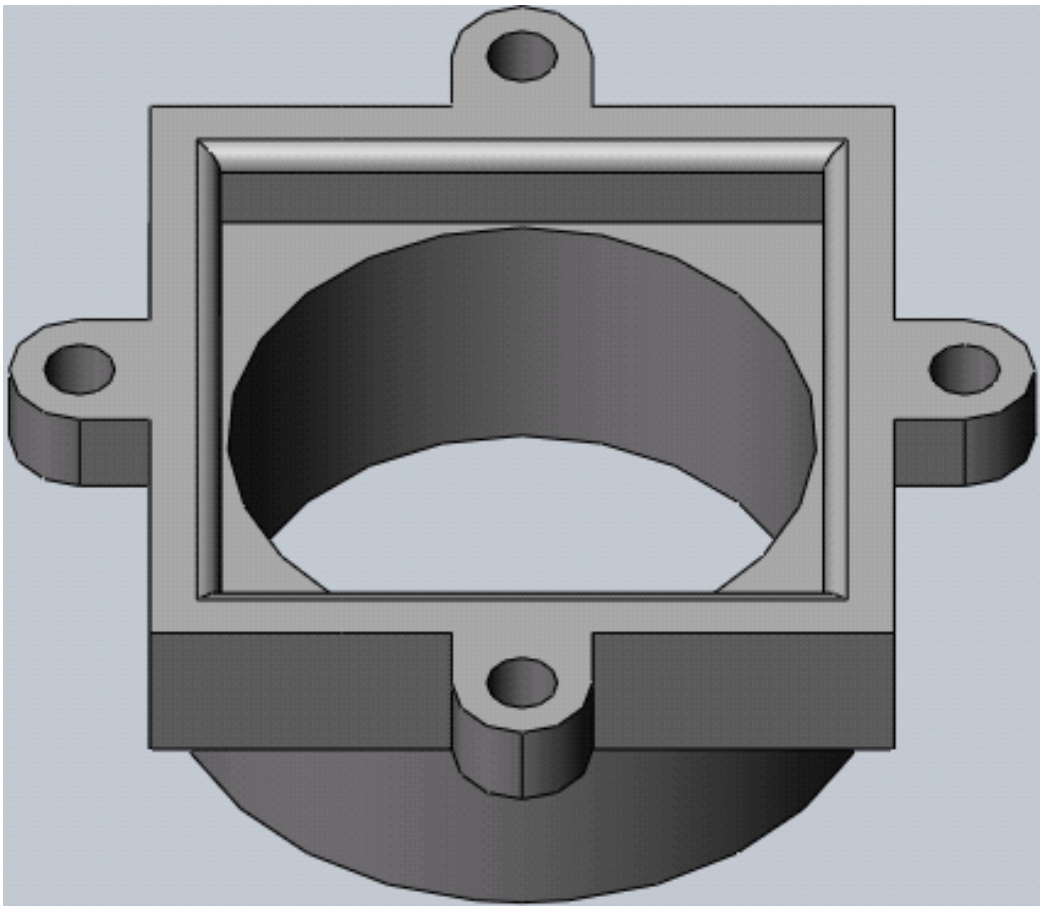


Figur A.1: Utlegg av kretskort

Figur A.1 viser den siste versjonen av kretskortet. Målingene er foretatt på en tidligere versjon.

Tillegg B

Linseholder



Linsen skrues inn i det runde hullet fra undersiden. Sokkelen til billedsensor-brikken passer i det firkantede hullet.

Tillegg C

VHDL kode for testing av chipen

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity test_logic is
  port (
    clk, reset, bus_req, transmit_ack_n : in  std_logic;
    bus_x, bus_y                        : in  std_logic_vector(5 downto 0);
    sw                                  : in  std_logic_vector(7 downto 0);
    bus_ack_n, sensor_mode, res         : out std_logic;
    ld2, transmit_req_n                 : out std_logic;
    ABCDEFG                             : out std_logic_vector(0 to 6);
    ld                                  : out std_logic_vector(0 downto 0);
    transmit_x, transmit_y              : out std_logic_vector(5 downto 0);
    an                                  : out std_logic_vector(3 downto 0));
end test_logic;

architecture Behavioral of test_logic is

  component seg7ctrl
    port(
      D0, D1, D2, D3 : in  std_logic_vector(4 downto 0);
      sel            : in  std_logic_vector(1 downto 0);
      ABCDEFG        : out std_logic_vector(0 to 6);
      AN             : out std_logic_vector(3 downto 0));
  end component;

  component address_event_counter
```

```
    port(
      reset, clk, bus_event                 : in  std_logic;
      red_enable, green_enable, blue_enable : in  std_logic;
      display_mode, high_time               : in  std_logic_vector (1 downto 0);
      bus_x, bus_y                          : in  std_logic_vector (5 downto 0);
      g_time                                : in  std_logic_vector;
      error                                 : out std_logic;
      D0, D1, D2, D3                        : out std_logic_vector(4 downto 0));
  end component;

  component handshake
    generic (
      up, down                        :     integer);
    port (
      reset, clk, req, glitch_filter : in  std_logic;
      ack_n, event                   : out std_logic
      );
  end component;

  signal event, bus_event, counter_reset, slow_motion : std_logic;
  signal D0, D1, D2, D3, D_0, D_1, D_2, D_3           : std_logic_vector(4 downto 0) := (ot
  signal g_time                                       : std_logic_vector(35 downto 0);
  signal high_time                                    : std_logic_vector(1 downto 0) := (ot
  alias sel                                           : std_logic_vector(1 downto 0) is g_t

begin

  seg7ctrl1 : seg7ctrl
    port map(
      D0      => D_0,
      D1      => D_1,
      D2      => D_2,
      D3      => D_3,
      sel     => sel,
      ABCDEFG => ABCDEFG,
      AN      => AN
      );

  address_event_counter1 : address_event_counter
    port map(
      clk             => clk,
      reset           => counter_reset,
      display_mode(0) => sw(1),
```

```
    display_mode(1) => sw(2),
    bus_event       => event,
    bus_x           => bus_x,
    bus_y           => bus_y,
    red_enable      => sw(5),
    green_enable    => sw(6),
    blue_enable     => sw(7),
    g_time          => g_time,
    high_time       => high_time,
    D0              => D0,
    D1              => D1,
    D2              => D2,
    D3              => D3,
    error           => ld2
    );

handshake1 : handshake
  generic map (
    up            => 20,
    down          => 10)
  port map(
    reset         => reset,
    clk           => clk,
    req           => bus_req,
    ack_n         => bus_ack_n,
    glitch_filter => sw(4),
    event         => bus_event
    );

event       <= bus_event;
res         <= counter_reset;
ld(0)       <= sw(3);
slow_motion <= sw(3);
high_time   <= g_time(g_time'high downto g_time'high-1) when slow_motion = '1' els

event_transmit   : process(clk, reset)
  type state_type is (wait_for_event, wait_for_ack_on, wait_for_ack_off);
  variable state : state_type := wait_for_event;
begin
  if reset = '1' then
    transmit_req_n <= '1';
    state                     := wait_for_event;
  elsif rising_edge(clk) then
```

```
    transmit_x     <= bus_x;
    transmit_y     <= bus_y;
    case state is

      when wait_for_event =>
        if event = '1' then
          state := wait_for_ack_on;
          transmit_req_n <= '0';
        end if;

      when wait_for_ack_on =>
        if transmit_ack_n = '0' then
          state := wait_for_ack_off;
          transmit_req_n <= '1';
        end if;

      when wait_for_ack_off =>
        if transmit_ack_n = '1' then
          state := wait_for_event;
        end if;

      when others => null;
    end case;
  end if;
end process event_transmit;

display_lock : process(clk, reset, high_time, g_time)
begin
  if reset = '1' then
    D_0 <= (others => '0');
    D_1 <= (others => '0');
    D_2 <= (others => '0');
    D_3 <= (others => '0');
  elsif rising_edge(clk) and (high_time & g_time(g_time'high-2 downto 0) = 0) then
    D_0 <= D0;
    D_1 <= D1;
    D_2 <= D2;
    D_3 <= D3;
  end if;
end process display_lock;

g_timer : process(clk, reset)
begin
```

```
    if reset = '1' then
      g_time <= (others => '0');
    elsif rising_edge(clk) then
      g_time <= g_time + 1;
    end if;
  end process g_timer;

  chip_ctrl : process(g_time, sw(0), high_time)
  begin
    counter_reset         <= '0';
    sensor_mode           <= sw(0);
    if (high_time & g_time(g_time'high-2 downto 7) = 0) then
      if sw(0) = '1' then               -- continous mode
--              0 1 2 3
--                _____
-- res        : _/     \_
--              _________
-- sensor_mode:
        case g_time(6 downto 5) is
          when "00"   =>
            counter_reset <= '0';
            sensor_mode   <= '1';
          when "01"   =>
            counter_reset <= '1';
            sensor_mode   <= '1';
          when "10"   =>
            counter_reset <= '1';
            sensor_mode   <= '1';
          when "11"   =>
            counter_reset <= '1';
            sensor_mode   <= '1';
          when others => null;
        end case;
      else                              -- one shot mode
--              0 1 2 3
--                _
-- res        : _/ \_____
--                    _
-- sensor_mode: _____/ \_
        case g_time(6 downto 5) is
          when "00"   =>
            counter_reset <= '0';
            sensor_mode   <= '0';
```

```
          when "01"   =>
            counter_reset <= '1';
            sensor_mode   <= '0';
          when "10"   =>
            counter_reset <= '0';
            sensor_mode   <= '0';
          when "11"   =>
            counter_reset <= '0';
            sensor_mode   <= '1';
          when others => null;
        end case;
      end if;
    end if;
  end process chip_ctrl;

end Behavioral;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity handshake2 is
  port (
    reset, clk                          : in  std_logic;
    caviar_ack_n, chip_req              : in  std_logic;
    chip_x, chip_y                      : in  std_logic_vector(5 downto 0);
    sw                                  : in  std_logic_vector(7 downto 0);
    caviar_req_n, chip_ack_n, new_data : out std_logic;
    data                                : out std_logic_vector(11 downto 0);
    caviar_x, caviar_y, glitch_length  : out std_logic_vector(5 downto 0)
    );
end handshake2;

architecture behavorial of handshake2 is

  signal data_write, data_read, data_present, valid : std_logic;
  signal x, y, counter, max_glitch_length           : std_logic_vector(5 downto 0);
  signal color                                      : std_logic_vector(1 downto 0);

begin

  data          <= y & x;
  glitch_length <= max_glitch_length;
  color         <= chip_x(0) & chip_y(0);
  new_data      <= '0';

  color_filter : process(sw, color)
  begin
    case color is
      when "00" =>
        valid <= sw(0);                 -- red

      when "01" =>
        valid <= sw(2);                 -- blue

      when "11"   =>
        valid <= sw(1);                 -- green
      when others =>
        valid <= '0';
```

```
  end case;
end process color_filter;

receiver         : process(reset, clk)
  type state_type is (wait_for_chip_req_active, wait_for_chip_req_passive);
  variable state : state_type := wait_for_chip_req_active;

begin
  if reset = '1' then
    counter           <= "000000";
    chip_ack_n        <= '1';
    data_write        <= '0';
    max_glitch_length <= "000000";
  elsif rising_edge(clk) then
    data_write        <= '0';
    case state is

      when wait_for_chip_req_active =>
        chip_ack_n            <= '1';
        if chip_req = '0' then
          if max_glitch_length < counter then
            max_glitch_length <= counter;
          end if;
          counter             <= "000000";
        elsif counter = sw(4 downto 3) & "0000" then
          if sw(7) = '1' or data_present = '0' then
            counter           <= "000000";
            chip_ack_n        <= '0';
            data_write        <= '1';
            state := wait_for_chip_req_passive;
          end if;
        else
          counter             <= counter + 1;
        end if;

      when wait_for_chip_req_passive =>
        chip_ack_n   <= '0';
        if chip_req = '1' then
          counter    <= "000000";
        elsif counter = sw(4 downto 3) & "000" then
          counter    <= "000000";
          state := wait_for_chip_req_active;
          chip_ack_n <= '1';
```

```
        else
          counter    <= counter + 1;
        end if;

      when others => null;
    end case;
  end if;
end process receiver;

databuffer : process(clk, reset)
begin
  if reset = '1' then
    data_present     <= '0';
  elsif rising_edge(clk) then
    if data_write = '1' and valid = '1' then
      if not(x(5 downto 0)&y(5 downto 0) = chip_x(5 downto 0)&chip_y(5 downto 0))
        x            <= chip_x;
        y            <= chip_y;
        data_present <= '1';
      end if;
    elsif data_read = '1' then
      if sw(5) = '0' then
        caviar_x     <= x;
        caviar_y     <= y;
      else
        caviar_x     <= x(0) & x(5 downto 1);
        caviar_y     <= y(0) & y(5 downto 1);
      end if;
      data_present   <= '0';
    end if;
  end if;
end process databuffer;

transmitter      : process(clk, reset)
  type state_type is (wait_for_data, wait_for_ack_on, wait_for_ack_off);
  variable state : state_type := wait_for_data;
begin
  if reset = '1' then
    state                    := wait_for_data;
    caviar_req_n <= '1';
    data_read    <= '0';
  elsif rising_edge(clk) then
    data_read    <= '0';
```

```
      case state is

        when wait_for_data =>
          caviar_req_n   <= '1';
          if data_present = '1' or sw(6) = '1' then
            state := wait_for_ack_on;
            data_read    <= '1';
            caviar_req_n <= '0';
          else
            state := wait_for_data;
          end if;

        when wait_for_ack_on =>
          caviar_req_n   <= '0';
          if caviar_ack_n = '0' then
            caviar_req_n <= '1';
            state := wait_for_ack_off;
          end if;

        when wait_for_ack_off =>
          caviar_req_n <= '1';
          if caviar_ack_n = '1' then
            state := wait_for_data;
          end if;

        when others => null;
      end case;
    end if;
  end process transmitter;

end behavorial;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity address_event_counter is
  port(
    reset, clk, bus_event                 : in  std_logic;
    red_enable, green_enable, blue_enable : in  std_logic;
    display_mode, high_time               : in  std_logic_vector (1 downto 0);
    bus_x, bus_y                          : in  std_logic_vector (5 downto 0);
    g_time                                : in  std_logic_vector;
    error                                 : out std_logic;
    D0, D1, D2, D3                        : out std_logic_vector(4 downto 0));
end address_event_counter;

architecture BEHAVORIAL of address_event_counter is

  signal event_count                                    : std_logic_vector(19 downt
  signal delayed_event, WEB, DOA, DIB, reset_int, passed : std_logic
  signal address, unique_addresses, bus_yx              : std_logic_vector(11 downt
  signal color                                          : std_logic_vector(1 downto
  signal median_time                                    : std_logic_vector(25 downt

  component RAMB16_S1_S1
    port (
      DIA   : in  std_logic_vector (0 downto 0);
      ADDRA : in  std_logic_vector (13 downto 0);
      ENA   : in  std_logic;
      WEA   : in  std_logic;
      SSRA  : in  std_logic;
      CLKA  : in  std_logic;
      DOA   : out std_logic_vector (0 downto 0);
--
      DIB   : in  std_logic_vector (0 downto 0);
      ADDRB : in  std_logic_vector (13 downto 0);
      ENB   : in  std_logic;
      WEB   : in  std_logic;
      SSRB  : in  std_logic;
      CLKB  : in  std_logic;
      DOB   : out std_logic_vector (0 downto 0)
      );
  end component;
```

```
  component ram
    port (
      DIA   : in  std_logic_vector (0 downto 0);
      ADDRA : in  std_logic_vector (13 downto 0);
      ENA   : in  std_logic;
      WEA   : in  std_logic;
      SSRA  : in  std_logic;
      CLKA  : in  std_logic;
      DOA   : out std_logic_vector (0 downto 0);
--
      DIB   : in  std_logic_vector (0 downto 0);
      ADDRB : in  std_logic_vector (13 downto 0);
      ENB   : in  std_logic;
      WEB   : in  std_logic;
      SSRB  : in  std_logic;
      CLKB  : in  std_logic;
      DOB   : out std_logic_vector (0 downto 0)
      );
  end component;

begin

-- ram1 : ram                           -- simulation
  ram1 : RAMB16_S1_S1                   -- synt
    port map (
      DIA(0)             => '0',        -- insert 1 bit data in bus (<0 downto 0>)
      ADDRA(11 downto 0) => bus_yx,     -- insert 14 bits address bus
      ADDRA(12)          => '0',
      ADDRA(13)          => '0',
      ENA                => '1',        -- insert enable signal
      WEA                => '0',        -- insert write enable signal
      SSRA               => '0',        -- insert set/reset signal
      CLKA               => clk,        -- insert clock signal
      DOA(0)             => DOA,        -- insert 1 bit data out bus (<0 downto 0>)
--
      DIB(0)             => DIB,        -- insert 1 bit data in bus (<0 downto 0>)
      ADDRB(11 downto 0) => address,    -- insert 14 bits address bus
      ADDRB(12)          => '0',
      ADDRB(13)          => '0',
      ENB                => '1',        -- insert enable signal
      WEB                => WEB,        -- insert write enable signal
      SSRB               => '0',        -- insert set/reset signal
```

```
      CLKB               => clk         -- insert clock signal
--      DOB   =>                         -- insert 1 bit data out bus (<0 downto 0>)
      );

  bus_yx <= bus_y & bus_x;
  color  <= bus_y(0) & bus_x(0);
  error  <= '1' when (event_count > unique_addresses + 1) else '0';

  color_filter : process(color, red_enable, blue_enable, green_enable)
  begin
    case (color) is
      when "00"   =>                     -- red first
        passed <= red_enable;
      when "10"   =>                     -- blue last
        passed <= blue_enable;
      when "11"   =>                     -- green in the middle
        passed <= green_enable;
      when others =>
        passed <= '0';
    end case;
  end process color_filter;

  addresses_counter : process(clk)
  begin
    if rising_edge(clk) then
      delayed_event      <= bus_event;
      if reset = '1' then
        address          <= (others => '0');
        reset_int        <= '1';
      elsif reset_int = '1' then
        address          <= address + 1;
        unique_addresses <= (others => '0');
        if (address + 1 = 0) then
          reset_int      <= '0';
        else
          reset_int      <= '1';
        end if;
        WEB              <= '1';
        DIB              <= '0';
      elsif (delayed_event = '1') and (DOA = '0') and (passed = '1') then
        WEB              <= '1';
        DIB              <= '1';
        address          <= bus_yx;
```

```
      unique_addresses <= unique_addresses + 1;
    else
      WEB              <= '0';
      DIB              <= '0';
    end if;
  end if;
end process addresses_counter;

event_counter : process(reset_int, clk)
begin
  if reset_int = '1' then
    event_count     <= (others => '0');
  elsif rising_edge(clk) then
    if (delayed_event = '1') and (reset_int = '0') and (reset = '0') and (passed = '1') t
      if event_count = x"F1" then
        median_time <= high_time & g_time(g_time'high-2 downto 0);
      end if;
      event_count   <= event_count + 1;
    end if;
  end if;
end process event_counter;

display : process(display_mode, event_count, unique_addresses, median_time)
begin
  case display_mode is
    when "00"       =>
      D3 <= (others => '0');
      D2 <= '0' & unique_addresses(11 downto 8);
      D1 <= '0' & unique_addresses(7 downto 4);
      D0 <= '0' & unique_addresses(3 downto 0);
    when "01"       =>
      D3 <= '0' & event_count(15 downto 12);
      D2 <= '0' & event_count(11 downto 8);
      D1 <= '0' & event_count(7 downto 4);
      D0 <= '0' & event_count(3 downto 0);
    when "10"       =>
      D3 <= '0' & median_time(25 downto 22);
      D2 <= '0' & median_time(21 downto 18);
      D1 <= '0' & median_time(17 downto 14);
      D0 <= '0' & median_time(13 downto 10);
    when "11"       =>
      D3 <= '0' & event_count(19 downto 16);
      D2 <= '0' & event_count(15 downto 12);
```

```
        D1 <= '0' & event_count(11 downto 8);
        D0 <= '0' & event_count(7 downto 4);
      when others     =>
        D3 <= (others => '0');
        D2 <= (others => '0');
        D1 <= (others => '0');
        D0 <= (others => '0');
    end case;
  end process display;

end architecture BEHAVORIAL;
```