NTNU
Det skapende universitet

Dynamikkompresjon av høydynamiske bilder i hardware

Implementasjon av Reinhars Fotoreseptormodell

Svein Arne Jervell Hansen

Master i elektronikk

Oppgaven levert: Juni 2007
Hovedveileder: Bjørn B. Larsen, IET
Biveileder(e): Sohrab Yaghmai, Micron
Johannes Sølhusvik, Micron

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

Masteroppgaven bygger videre på prosjektoppgaven "Dynamikkompresjon av høydynamiske bilder" som gikk ut på å finne den dynamikkompresjonsalgoritmen som egner seg best for en hardwareimplementasjon. Målet med denne masteroppgaven er å implementere dynamikkompresjonsalgoritmen Reinhards Fotoreseptormodell [Reinhard05] i hardware. I implementasjonen av denne algoritmen er det viktig med effektiv logaritme, så det vil også bli sett på forskjellige implementasjonsmetoder av denne operasjonen.

For å få til dette må algoritmen først optimaliseres m.h.p. gitte kriterier.
Kriteriene som er viktige i denne sammenhengen er ytelse og areal. Det er ønskelig med forskjellige implementasjonsversjoner som veier mellom disse kriteriene.

Resultatene skal måles iform av:
* Ytelse (fps, VGA-bilde)
* Areal (gate-count)
* Visuelt bilde er også ønskelig

Implementasjonen skal brukes som en testkrets, så det er ønskelig å beholde alle inngangsparametre brukerstyrte på tross av at dette går på bekostning av ytelsen.

Oppgaven gitt: 15. januar 2007
Hovedveileder: Bjørn B. Larsen, IET

Sammendrag

Dagens bildestandarder har 8-bit oppløsning per farge, noe som er lite i forhold til reelle scener man observerer i hverdagen. Det blir stadig sterkere fokus på høydynamisk bildeteknologi, og overgangen fra lavdynamisk til høydynamisk bildeteknologi er spådd å bli like stor som overgangen fra svart-hvitt til fargebilder. Siden man ikke har blitt enige om en standard for lagring og fremvisning av høydynamiske bilder, er fokuset på å komprimere de høydynamiske bildene ned til et lavdynamisk format uten å miste den visuelle informasjonen i bildet. Ved å bruke et vanlig lavdynamisk bildeformat, som for eksempel JPEG, kan de høydynamiske bildene brukes på eksisterende utstyr. I et kamerasystem vil det være ønskelig at denne kompresjonen skjer direkte i kamerabrikken, men det er foreløpig ikke rapportert om noen slike ASIC-løsninger.

Denne masteroppgaven tar for seg implementasjonen av en slik ASIC-løsning, og bygger på prosjektoppgaven "Dynamikkompresjon av høydynamiske bilder" som finner den kompresjons-algoritmen som egner seg best til en hardwareimplementasjon. Først analyseres denne algoritmen før den modifiseres for å egne seg bedre til hardwareimplementasjon.

Planleggingsfasen har som mål å danne bakgrunn for arkitekturløsningene som skal implementeres. En dynamikkompresjonsalgoritme er bygd opp av aritmetiske operasjoner, og spesielt logaritme er problematisk å implementere i hardware. Logaritme blir derfor viet ekstra oppmerksomhet i analysen, og de forskjellige måtene å implementere logaritme på blir utforsket for å finne den metodikken som egner seg best.

Selve arbeidet dokumenteres ved å først skissere de forskjellige arkitekurene gjennom en bottom-up metodikk. Deretter beskrives undermodulene før arbeidet oppsummeres ved å beskrive de forskjellige valgene som er tatt i implementasjonsfasen.

Både de visuelle og de fysiske resultatene blir så presentert, og satt i sammenheng med forskjellige applikasjoner for å vurdere om løsningen i denne masteroppgaven er konkurransedyktig.

Til slutt oppsummeres diskusjonen ved å konkludere med at løsningen presentert i denne oppgaven er konkurransedyktig på ytelse og overlegen på ressursbruk sammenlignet med eksisterende løsninger. Det er fortsatt noe arbeid som kan gjøres for å perfeksjonere løsningen, og oppgaven avrundes med å beskrive videre arbeid for optimalisering.

Forord

Denne masteroppgaven er en fortsettelse på prosjektoppgaven "Dynamikkompresjon av høydynamiske bilder" **[Hansen06]** jeg skrev høsten 2006. Prosjektoppgaven er et studie innen dynamikkompresjonsalgoritmer hvor målet er å finne den dynamikkompresjonsalgoritmen som egner seg best til å implementeres i hardware. Konklusjonen i denne prosjektoppgaven er at det er Reinhards Fotoreseptormodell **[Reinhard05]** som egner seg best til en hardwareimplementasjon, og det er der denne masteroppgaven tar over.

Siden masteroppgaven bygger videre på arbeidet til prosjektoppgaven baserer arbeidet på masteroppgaven seg på de valg og avgjørelser som ble tatt i prosjektoppgaven. Algoritmen det er tatt utgangspunkt i har blitt litt modifisert for å passe bedre i en hardwareimplementasjon, men det har ikke hatt noe visuelt utslag på de komprimerte bildene.

Når det gjelder den visuelle kvaliteten på de komprimerte bildene er jeg litt overrasket over hvor bra den er. Flere av mellomregningene er utført med begrenset presisjon og matematiske tilnærminger, men dette gir ingen utslag i den visuelle kvaliteten.

Et problemet har vært å finne noe relevant å sammenligne arbeidet med, siden denne løsningen er unik i implementasjonsform av en slik algoritme. De løsningene jeg har sammenlignet med vil ikke være aktuelle for samme applikasjoner som den løsningen som er presentert her i masteroppgaven, men sammenligningen gir allikevel en pekepinn på om ytelsen i denne masteroppgaven er konkurransedyktig i forhold til aktuelle standarder for levende bilder.

Det er skrevet mye kode i Matlab og VHDL i denne masteroppgaven, og de mest relevante filene følger med som filvedlegg til innleveringen. Disse filene representerer den mest relevante koden, det vil si hele designet i VHDL og de Matlab-filene nevnt i kapittel 6.2. Det er skrevet mye kode utover dette også for å teste ut forskjellige løsninger, samt for å beregne feil i algoritmene. Den vedlagte koden utgjør i overkant av 6000 linjer i VHDL og ca 200 linjer i Matlab, inkludert kommentarer og testbenker.

Denne masteroppgaven danner et meget bra grunnlag å bygge videre på når man skal implementere en slik algoritme i en kamerabrikke. Ved å utføre arbeidet nevnt i kapittel 6.2 mener jeg at man får en god og allsidig løsning som kan brukes i flere forskjellige applikasjoner.

Jeg vil også rette en takk til faglærer Bjørn B. Larsen samt veilederne Sohrab Yaghmai og Johannes Sølhusvik som har hjulpet meg gjennom denne oppgaven.

Målgruppen for denne oppgaven er sivilingeniører innen elektronikk.

Innledning

Målet med denne masteroppgaven er å implementere dynamikkompresjonsalgoritmen Reinhards Fotoreseptormodell i hardware på en så effektiv måte som mulig, samt å vurdere hvilke arkitekturløsninger som lønner seg ut fra krav om ytelse og areal.

Masteroppgaven starter med å forklare svakheter med dagens bildeteknologi, og hvorfor høydynamisk bildeteknologi er på vei inn i markedet. Kapittel 1 avslutter med å beskrive hva som er hovedfokuset for masteroppgaven, samt en beskrivelse av selve dynamikkompresjonsalgoritmen som skal implementeres i denne oppgaven.

Kapittel 2 er en gjennomgang av teorien i planleggingsstadiet til implementasjonen. Først blir selve kompresjonsalgoritmen analysert og modifisert for å bedre passe i en hardware-implementasjon. Deretter blir den matematiske operasjonen logaritme viet ekstra oppmerksomhet, siden denne operasjonen tradisjonelt er meget problemfylt å implementere diskret i hardware.

Designfasen blir så presentert i kapittel 3. Først blir det presentert to forskjellige arkitekturer med forskjellig fokus iforhold til ytelse og areal. Her blir også alle undermoduler og de atomære operasjonene presentert. Deretter blir undermodulene beskrevet gjennom en bottom-up metodikk før arbeidet til slutt blir arbeidet oppsummert ved å beskrive arbeidsmetodikken og de forskjellige valgene som er tatt iløpet av designfasen i denne masteroppgaven.

Deretter følger kapittel 4 som oppsummerer resultatene samt beskriver resultater fra andre og sammenlignbare prosjekter. Både de visuelle og de fysiske synteseresultatene blir her gjennomgått.

Disse resultatene danner igjen grunnlag for diskusjonen som kommer i kapittel 5. Her blir både de visuelle og de fysiske resultatene vurdert, samt at ytelsesresultatene blir vurdert opp mot alternative løsninger og de mest aktuelle bruksapplikasjonene.

Til slutt oppsummeres disse vurderingene i kapittel 6 ved å trekke konklusjoner over arbeidet. Kapittelet avrundes ved å beskrive noen punkter for videre arbeid som burde gjennomføres før dynamikkompresjonsmodulen implementeres i et ferdig produkt.

Innholdsfortegnelse

Ord og uttrykk

Tabell 1: Ordliste som beskriver betydningen av enkelte ord og uttrykk som er brukt gjennom denne masteroppgaven.

Ord/Uttrykk	***Betydning***
ALU	Arithmetic Logic Unit. En modul i en prosesseringsenhet (f.eks en MCU eller CPU) som utfører de aritmetiske og logiske operasjonene.
Approksimasjon	Tilnærming.
ASIC	Application Spesific Integrated Circuit. En IC som er spesialdesignet for en spesiell applikasjon.
CORDIC	COordinate Rotation DIgital Computer. Enkel og effektiv algoritme som beregner hyperboliske og trigonometriske funksjoner ved hjelp av LUTer, shift og adderer.
CPU	Central Prosessing Unit. Den vanligste betegnelsen på en prosessor. Forbindes oftest med prosessorer som står i en PC.
DSP	Digital Signal Prosessor. En prosessor med spesielle instruksjoner beregnet for Digital Signal Prosessering.
DSP-applikasjon	Digital Signal Prosesserings – Applikasjon. Applikasjoner hvor prosessering av digitale signaler står sentralt.
Fps	Frames Per Second. Betegnelse for antall bilderammer per sekund i levende bilder.
GPU	Graphic Processing Unit. Grafikkprosessor. En prosessor med eget instruksjonssett for grafikkberegninger.
HDR	High Dynamic Range. Betegnelse for teknologi med mer enn 8-bits definisjoner per farge.
Interscene	En sammenligning av forhold mellom to eller flere scener.
Intrascene	En sammenligning av forhold innad i en og samme scene.
LDR	Low Dynamic Range. Betegnelse for teknologi med 8-bits (eller lavere) definisjoner per farge.
MCU	MicroController Unit. En liten prosessor med begrenset instruksjonssett som ofte implementeres internt i ASICer.
MVS	Mennesket Visuelle System. En fellesbetegnelse for det visuelle oppfattelsessystemet som består av øyet, hjernen og kommunikasjonen mellom disse.
Ytelse	I denne oppgaven er ytelsen et mål på hastighet, og kan måles i antall operasjoner per sekund. For logaritme-enheten vil det si antall logaritmer utført per sekund, mens for dynamikkompresjonsmodulen vil det si antall piksler komprimert per sekund.

1 Introduksjon

Dette kapittelet tar først for seg hvorfor høydynamisk bildeteknologi er på vei inn i markedet, samt grunner til at høydynamisk bildeteknologi på sikt vil ta over for lavdynamisk teknologi. Deretter blir fokuset for denne oppgaven presentert før den blir satt i system ved å vise hvor den hører hjemme i en informasjonskjede i et digitalt kamera. Til slutt blir selve dynamikkompresjonsalgoritmen presentert sammen med tidligere arbeid på dette feltet.

1.1 Høydynamisk bildeteknologi

Fokuset innen bildeteknologi har vært å få flere piksler inn på en kamerabrikke fremfor å få større dynamikk i hvert piksel. Antall piksler i kamerabrikkene er i ferd med å nå et metningspunkt iforhold til hva som er praktisk og ønskelig i konsumermarkedet, og høydynamisk bildeteknologi er derfor ventet å bli et av de neste fokusene for bransjen.

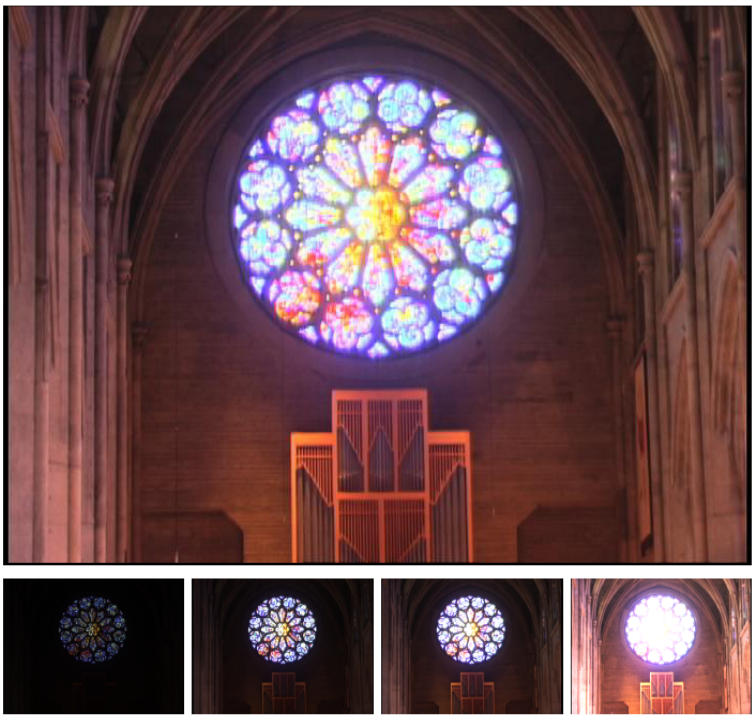
Dagens bildelagringsstandarder baserer seg på at bildene skal kunne vises frem på CRT-skjermer. CRT-skjermer inneholder fosfor som eksiteres, og kan på grunn av fysiske begrensninger kun avgi noen hundre forskjellige lysintensiteter. Tradisjonelt kombineres forskjellige lysintensiteter i fargene rødt, grønt og blått for å generere de fargene man er ute etter. Det er vanlig å lagre digitale bilder med maksimalt 8 bit i hver av disse fargene, noe som gir 256 forskjellige lysintensiteter i både rødt, grønt og blått. Kombinert vil dette kunne gi 256^3 forskjellige kombinasjoner, noe som tilsvarer over 16 millioner farger. 16 millioner farger høres imponerende ut, men i realiteten representerer bildet fortsatt bare 256 forskjellige intensiteter, noe som er lite i forhold til hva vi mennesker kan oppfatte **[Fukui01]**. Et bilde med 8 bits oppløsning eller mindre per farge er å betrakte som et lavdynamisk bilde og vil heretter bli referert til som et LDR bilde. Fra tabell 2 kan man se at lysstyrken i en innendørs scene med et vindu hvor sollys kommer inn vil ha et større dynamisk område enn det som kan lagres i et LDR bilde. Det vil derfor være en del informasjon som øyet oppfatter i denne scenen som ikke vil være representert i LDR-versjonen av bildet. Scener med høyere dynamisk område enn det kameraet støtter vil bli lagret med sterkt redusert kvalitet ved lineær respons.

*Tabell 2: Lysstyrke i noen vanlige scener. Verdiene er hentet fra **[Reinhard06]** og **[Fukui01]**.*

Lysforhold / Dynamiske områder	***Lysstyrke [candela/m^2]***
Stjernelys	10^{-3}
Månelys	10^{-1}
Innendørs belysning	10^{2}
Sollys	10^{5}
Maksimal lysstyrke i en CRT-skjerm	10^{2}
MVS intrascene dynamisk område	10^{5}-10^{6}
MVS interscene dynamisk område	10^{10}

LDR-representasjon av digitale bilder er veldig utbredt og de fleste digitalkameraer tar LDR bilder direkte. All informasjon som er utenfor det lagrede bildets dynamiske område vil dermed være tapt uten mulighet for gjenopprettelse. Det er per dags dato ingen kommersielle produkter tilgjengelig som tar HDR bilder direkte på en tilfredsstillende måte, men overgangen fra LDR til HDR er spådd til å bli like merkbar som overgangen fra svart-hvitt til fargebilder **[Reinhard06]**.

For å få med så mye informasjon som vi selv kan oppfatte i en scene må man ta bilder med et dynamisk område langt større enn 8 bit per farge. Et slikt bilde vil være å betrakte som et høydynamisk bilde og vil heretter bli referert til som et HDR bilde.



Figur 1: Høydynamisk bilde satt sammen fra fire lavdynamiske bilder med forskjellig lukkertid.

Det er flere måter å generere HDR bilder på, og de to mest utbredte måtene er datagenerasjon og sammenslåing av en bildesekvens tatt med konvensjonelt kamerautstyr med forskjellig eksponeringstid **[Reinhard06]**. Ved datagenerasjon blir syntetiske bilder generert ved hjelp av dataprogrammer som tar hensyn til alt fra 3D-modellering til lyssetting. Dette er et meget stort felt som ikke denne oppgaven vil gå nærmere inn på.

HDR bildegenerering ved hjelp av en bildesekvens med varierende eksponeringstid er den mest brukte metoden for å generere HDR bilder fra virkelige scener. Prinsippet bak denne metoden er at selv om alle bildene i bildesekvensen er LDR bilder, så representerer de forskjellige dynamiske områder og kan dermed kombineres for å lage et endelig HDR bilde. Hvert piksel i bildet vil være forskjellige eksponert i hvert bilde, enten undereksponerte, overeksponerte eller eksponerte i sitt lineære område. Ved å bruke en bildesekvens med de riktige eksponeringstidene vil alle pikselene være i det lineære området på minimum et av bildene. Under forutsetningen av at pikselene er perfekt lineære kan man dele pikselverdiene (som er i det lineære området) med eksponeringstiden for å få alle verdiene representert i det samme verdidomenet.

En av svakhetene med denne metoden er at den er meget tungvindt og krever en viss datakraft. En annen svakhet er følsomheten for bevegelse i både kameraet selv og i scenen man tar bilde av. Om noe i scenen beveger seg vil ikke de forskjellige bildene i sekvensen representere den samme scenen. I tillegg er det meget vanskelig å holde kameraet rolig nok, noe som resulterer i at man er avhengig av å bruke et kamerastativ eller avanserte stabiliseringsalgoritmer. Alternativt at bildene automatisk blir tatt i så rask rekkefølge at scenen ikke vil være forandret fra bilde til bilde, men om dette er mulig å gjennomføre eller ikke avhenger av hva slags scene man tar bilde av og lysforholdet i denne scenen. Figur 1 viser en illustrasjon av hvordan fire lavdynamiske bilder kan settes sammen til en høydynamisk scene, siden alle pikselene er i det lineære området i minst ett av de lavdynamiske bildene.

De fleste produsenter av kamerabrikker til digitalkameraer har nå begynt å se på muligheten til å produsere kamerabrikker som tar HDR bilder direkte. Dette kan for eksempel gjøres ved å lage ulineære piksler, piksler med en meget høy SNR eller ved å styre kamerabrikken slik at den tar flere bilder med forskjellig lukkertid som den setter sammen til et HDR bilde.

1.2 Fokuset for denne oppgaven

Fokuset i denne masteroppgaven vil ikke være hvordan HDR bilde blir generert, men selve hardwaremodulen hvor dynamikken i HDR-bildet blir komprimert. Masteroppgaven bygger på prosjektoppgaven **[Hansen06]**, og de avgjørelsene som ble tatt i den prosjektoppgaven blir her lagt til grunn og bygd videre på. Dynamikkompresjonen vil derfor utføres etter prinsippene i Reinhards Fotoreseptormodell og hovedfokuset for denne oppgaven vil være å implementere Reinhards Fotoreseptormodell i hardware på en så effektiv måte som mulig.

For å få en så effektiv hardwareimplementasjon som mulig vil dynamikkompresjonsalgoritmen bli analysert og delt inn undermoduler, som igjen vil bli analysert for å finne den beste balansen mellom areal og ytelse for hver av undermodulene. Ved arealbesparelser vil blant annet gjenbruk av moduler være i fokus der dette er hensiktsmessig.

I tillegg er den matematiske operasjonen logaritme viet ekstra oppmerksomhet i denne oppgaven. Dynamikkompresjonsalgoritmen er avhengig av logaritmeoperasjonen, så det er viktig å få denne implementert så optimalt som mulig.

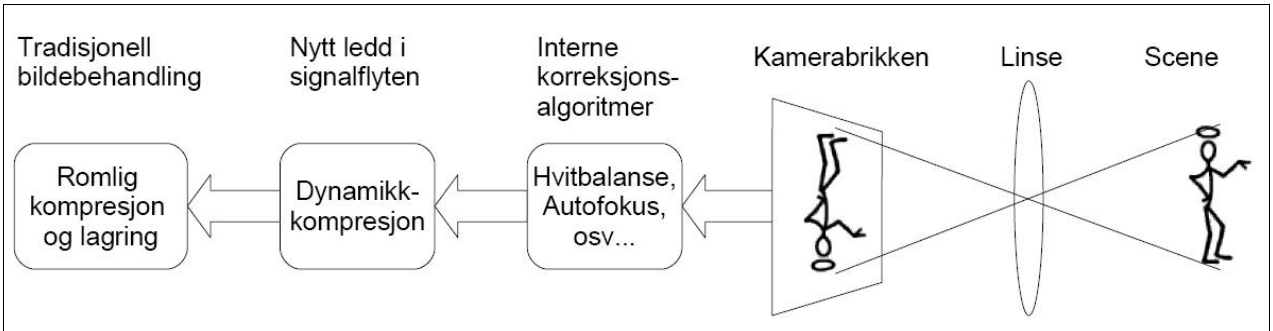
Areal, ytelse og effektiv arkitektur vil hele veien være fokuset i implementasjonen, og vil også bli diskutert i slutten av oppgaven.

1.3 Modulen sin plass i et kamerasystem

Et lavdynamisk digitalkamera må ha en SNR på minimum 48dB for å få dynamisk område i størrelsesorden 10^2 samt en feilfri 8bits representasjon av hvert piksel. Noen kameraer har en marginalt høyere dynamikk enn dette og komprimerer dynamikken internt i sensoren for å få en 8bits representasjon ut av sensorbrikken.

Ved høydynamisk bildetagning ser man for seg et system hvor en bildesensor skal kunne fange et bilde med like stor dynamikk som MVS kan oppfatte. Om man velger å representere absolutt luminans med et kamera må man belage seg på å oppnå en dynamikk på 200dB siden dette er det interscene dynamiske området til MVS **[Fukui01]**. Denne dynamikken tilsvarer et dynamisk område på 10^{10} som trenger en 34bits representasjon per farge for å dekke hele området.

Den mest nærliggende løsningen vil være å representere det intrascene dynamiske området til MVS, noe som vil si et område i overkant av 10^5 verdier. Dette tilsvarer en dynamikk i overkant av 100dB og en representasjon på 17-20bit. Det intrascene dynamiske området kan sees på som et



Figur 2: Signalflyten i et digitalkamera.

relativt dynamisk område som beveger seg langs skalaen for absolutt dynamikk ved å justere lukkertiden på kameraet.

Den 17-20bits representasjon som kommer ut av sensorbrikken må prosesseres for å komprimere dynamikken før kompresjon ned til det ønskede LDR-representasjonsformatet (for eksempel JPEG). Figur 2 viser signalflyten i et digitalkamera, og plasseringen til dynamikkompresjonsmodulen som skal implementeres i denne masteroppgaven. Dynamikkompresjonsmodulen er en implementasjon av Reinhards Fotoreseptormodell som beskrevet i **[Hansen06]**. Reinhards Fotoreseptormodell er også oppsummert senere i dette kapittelet.

1.4 Tidligere arbeid innenfor fagområdet

Det er gjort en del arbeid innen feltet dynamikkompresjon av høydynamiske bilder, hvor fokuset stort sett har vært på bildebehandling i datamaskiner med stor regnekraft **[Hansen06]**. De fleste kompresjonsalgoritmene er av den grunn meget ressurskrevende både i form av regnekraft og minnebruk.

Det er foreløpig ikke rapportert om noen ASIC implementasjoner av slike algoritmer. I en ASIC vil det være meget begrenset hvor store minneressurser en har tilgjengelig, da dedikert minne bruker stort areal, og dermed er forholdsvis dyrt og implementere. Det er også store begrensninger innen regnekraft, så sant man skal holde areal og effektforbruk innenfor et akseptabelt nivå.

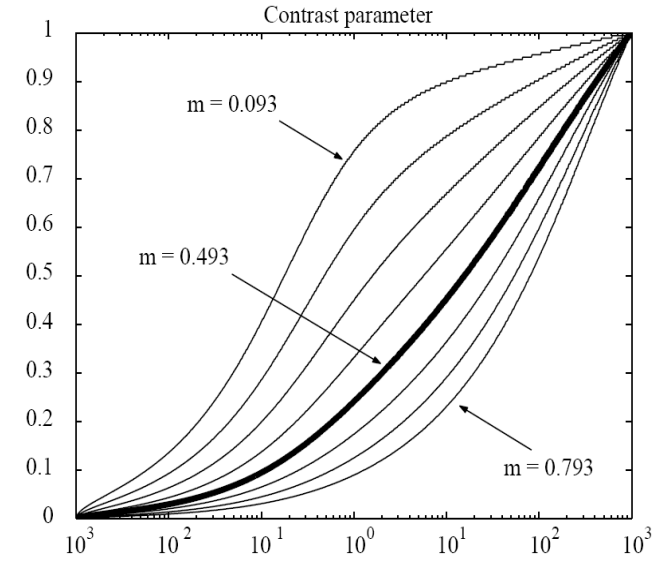
En annen grunn til at det ikke har vært interesse for ASIC implementasjon av slike algoritmer før nå, er at fokuset innen bildeteknologi har vært å få flere piksler inn på en kamerabrikke fremfor å få større dynamikk i hvert piksel. Dette har ført til at digitale kameraer ikke har hatt behov for en dynamikkompresjon, og det eneste reelle bruksområdet for dette fagområdet har derfor vært datagrafikk.

Når det gjelder implementasjonsmetodikk er det flere måter dette kan gjennomføres på. Algoritmen kan enten implementeres fullt i dedikert hardware som i en ASIC, implementeres i en mikrokontroller (MCU), kjøres på en dedikert prosessor (CPU) eller den kan deles opp slik at noen deler kjører i dedikert hardware og noen deler i software på en CPU. En annen mulighet er å implementere den i en dedikert grafikkprosessor (GPU), som er mer regneeffektiv enn en CPU. Hva slags metode som egner seg best for denne applikasjonen vil bli diskutert på et senere tidspunkt.

1.5 Reinhards Fotoreseptormodell

Reinhards Fotoreseptormodell er en dynamikkompresjonsalgoritme for høydynamisk bilder som bygger på de fysiologiske egenskapene og begrensingene til fotoreseptorene inne i øyet. Fotoreseptorenes respons blir i denne algoritmen modellert med en s-kurve ut fra teorien til Kleinschmidt og Dowling **[Kleinschmidt75]**. Metningskonstanten for s-kurven er beregnet ut fra bildets maksimum-, minimum- og gjennomsnittsverdi samt inngangsparametre. Kompresjonskurven danner en s-kurve i det semilogaritmiske planet og tilpasses hvert enkelt piksel med utgangspunkt i pikselets egen verdi. Hver farge blir forøvrig behandlet separat, noe som åpner for modellering av kromatisk adapsjon i tillegg til luminansadapsjon. Figur 3 illustrerer forskjellige kompresjonskurver generert med forskjellige parameterverdier.

Både den kromatiske- og luminansadapsjonen styres av inngangsparametre som vekter mellom globale verdier og pikselets egen verdi, både når det gjelder luminans og fargestyrke. I tillegg har **[Reinhard05]** inngangsparametre som styrer kontrast og intensitet, noe som gjør denne algoritmen meget allsidig. Det kan sees på som en ulempe at algoritmen kun tar for seg dynamikkompresjon og ingen av de andre effektene i MVS, men det er valget Reinhard og Devlin tok for å forsikre seg om at algoritmen ble regneeffektiv.



Figur 3: Eksempler på kompresjonskurver styrt av parameteren m. ***[Reinhard05]***

Bildene denne algoritmen produserer blir meget gode selv med standardverdier på inngangsparametrene, og siden parametrene har en intuitiv effekt er det fort gjort å vite hva man skal stille på om man ikke er helt fornøyd med bildet.

For nærmere gjennomgang av Reinhard Fotoreseptormodell se **[Reinhard05]**, eller se **[Hansen06]** for en gjennomgang av algoritmen med utgangspunkt i visuell kvalitet og ressursbruk.

Dette kapittelet har omhandlet fordelen med høydynamisk bildeteknologi og hvorfor høydynamisk bildeteknologi er etterspurt i markedet. Kapittelet har også tatt for seg hva som er fokuset i denne masteroppgaven og hvor denne oppgaven passer inn i et høydynamisk bildesystem. Neste kapittel vil ta for seg teorien som designfasen baserer seg på.

2 Teori

Dette kapittelet tar for seg teorien bak implementasjonen av Reinhards Fotoreseptormodell. Først blir algoritmen analysert for å finne ut hva som burde endres i algoritmen for å gjøre den mer optimal for en hovedargumentasjonenes. Logaritme blir betraktet som en av de mest problemfylte DSP-operasjonene å implementere i hardware, og blir derfor gjennomgått grundig i det påfølgende delkapittelet for å finne ut hvilken implementasjonsmetodikk som passer best for denne oppgaven.

2.1 Reinhards Fotoreseptormodell

Algoritmen for dynamikkompresjonen er i prinsippet hentet fra **[Reinhard05]** og er presentert som Matlabkode i vedlegg 1. Denne algoritmen er ikke optimalisert for hardware, men med små justeringene blir denne algoritmen mer hardwarevennlig. Dette delkapittelet presenterer hvilke justeringer som skal til for å gjøre algoritmen mer optimal for hardwareimplementasjon.

2.1.1 Det dynamiske området

Det dynamiske området til inngangssignalet styrer mange av parametrene til algoritmen, siden kompresjonsfaktoren må være i samme størrelsesorden som selve signalet. Denne egenskapen er ikke definert i den originale algoritmen, men Reinhard og Devlin valgte istedenfor å lage en inngangsvariabel som justerer kompresjonsfaktoren, og dermed styrer den generelle lysheten i bildet. Inngangsvariabelen som styrer den generelle lysheten bestemmer dermed også hvilken størrelsesorden det er på kompresjonen av pikselverdiene. Denne inngangsparameteren er variabelen *f_inn* i vedlegg 1. Variabelen *f_inn* bestemmer størrelsesorden på kompresjonen, og må dermed være i samme størrelsesorden som logaritmen til inngangsverdiene som komprimeres.

Dynamikken på parameteren *f* er i samme størrelsesorden som dynamikken på inngangssignalet, og ved kompresjon av høydynamiske bilder kan det være vanskelig å finne ut hvilken størrelsesorden parameteren *f* skal være i. For å gjøre algoritmen mer brukervennlig er det derfor hensiktsmessig å beregne en initiell verdi for *f*, slik at *f_inn* isteden blir brukt for å inkrementere eller dekrementere denne initialverdien. Initialverdien til parameteren *f* tar utgangspunkt i gjennomsnittlig luminansverdi i bildet. Ved eksperimentering viser det seg at den logaritmiske gjennomsnittsluminansen multiplisert med en faktor tilsvarende 1.375 er et godt

utgangspunkt for inngangskompresjonsparameteren *f_inn*. Totalverdien for f blir da:

$$f = f_{inn} + (Llav * 1.375)$$

Det er i likhet med parameteren *f* vanskelig å finne en initialverdi for parameteren *m*. **[Reinhard05]** foreslår en initialverdi som kan overstyres med en brukerparameter. Istedenfor å ha en slik enten-eller fremgangsmåte er det mer hensiktsmessig å la initialverdien til *m* bli beregnet i henhold til **[Reinhard05]**, mens den brukerstyrte parameteren *m_inn* kan, i likhet med *f_inn*, inkrementere eller dekrementere initialverdien etter brukerens ønske.

2.1.2 Logaritme

Den matematiske operatoren logaritme brukes flere steder i algoritmen for å beregne parametre som brukes underveis i kompresjonen. En logaritme med base 2 vil være mer hardwarevennlig enn logaritme med base 10, men gir en litt annen kompresjonskurve. Denne kurveendringen kan kompenseres ved å justere variabelen *f* ytterligere siden både de logaritmiske verdiene og inngangsvariablene sammen styrer kompresjonsparameteren σ, og dermed også den endelige kompresjonskurven. En slik justering er lagt inn i utregningen av initialverdien til *f*.

Når logaritmen blir byttet til base 2 vil det også være naturlig å bytte antilogaritmen til base 2. Dermed blir gjennomsnittsluminansen beregnet ved regne ut

$$Lav = \exp(Llav) \approx Lav = 2^{Llav}$$

istedenfor å operere med base e som i den originale algoritmen. Hardwareimplementasjon av logaritme blir forøvrig studert nærmere i kapittel 2.2.

2.1.3 Estimering av parametre

Denne masteroppgaven er som tidligere nevnt en videreføring av **[Hansen06]**, og de avgjørelsene som ble tatt i den prosjektoppgaven er derfor gjeldende også i denne masteroppgaven. I **[Hansen06]** ble det vist at det er forsvarlig å estimere de globale parametrene ved å beregne de basert på en bilderamme i forveien. Parametrene blir dermed beregnet parallelt med kompresjonen av bildet, og parametrene beregnet av forrige bilderamme brukes da til komprimering av nåværende

ramme.

De verdiene som blir brukt fra forrige ramme for å beregne kompresjonsparametrene er gjennomsnittsverdien til hver av fargekanalene, gjennomsnittsverdien til luminansen samt maksimal- og minimumsverdien til luminansen. Det er altså bare globale verdier som blir estimert, og det vil kun bli utslag i bildekvaliteten ved store svingninger i disse verdiene siden de er forholdsvis stabile.

Ved å bruke de globale verdiene fra forrige bilderamme øker ytelsen samt at arealbruken går ned. Om disse parameterverdiene skulle blitt regnet ut fra den reelle bilderammen, måtte først hele rammen lagres i minne før disse globale verdiene kunne blitt beregnet. Deretter måtte kompresjonsparametrene blitt beregnet før bilderammen hadde blitt komprimert piksel for piksel. En slik fremgangsmåte hadde gitt et meget stort minnebehov, samt at en slik sekvensiell gjennomgang av rutiner ville senket ytelsen siden ingenting ville blitt beregnet parallelt. Ved å estimere parametrene derimot, vil beregningen av de globale verdiene skje parallelt med komprimeringen av pikselverdiene.

I vedlegg 2 er det presentert en ny kode for kompresjonsalgoritmen som tar hensyn til de endringene nevnt i dette delkapittelet. I denne koden blir inngangssignalet i tillegg skalert for å passe inn i området $[0, 2^{18}]$ for å simulere en 18-bits heltallsverdi.

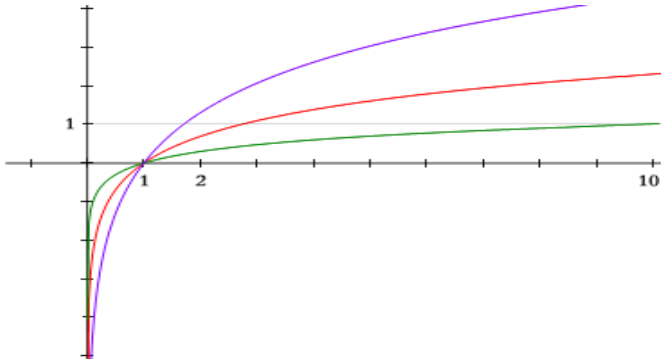
Dette delkapittelet har omhandlet justeringer i dynamikkompresjonsalgoritmen for å gjøre den mer hardwarevennlig enn den algoritmen som opprinnelig er skissert i **[Reinhard05]**. Neste delkapittel handler om hardwareimplementasjon av logaritme, og forskjellige metoder å estimere denne matematiske operasjonen på.

2.2 Logaritme for bruk i DSP-applikasjoner

Logaritme er en av de funksjonene som det tradisjonelt er problemfylt å implementere i hardware. En av grunnene til dette problemet er at en digital implementasjon vil ha en endelig oppløsning og et endelig dynamisk område, mens den matematiske definisjonen av logaritme har en uendelig oppløsning og et uendelig stort dynamisk område. Denne begrensningen innen elektronikk utnyttes i flere kodingsalgoritmer, som for eksempel RSA. Disse kodingsalgoritmene baserer seg på at det ikke er mulig å beregne logaritme av et meget stort tall uten ekstremt store mellomlagrings- og regneressurser, så dekodingen må derfor skje ved hjelp av andre metoder som forutsetter at man har informasjon om enkelte digitale nøkler isteden.

De vanligste måtene å implementere logaritme på er ved hjelp av rekkeutvikling, polynom approksimasjon eller lineær approksimasjon. Logaritme kan også implementeres ved hjelp av en LUT, og da slå opp tilsvarende verdi for hver nye inngangsverdi.

Vanskelighetsgraden i implementasjonen av logaritme er avhengig av det dynamiske området til inngangssignalet og den ønskede presisjonen. Et inngangsignal med dynamisk område <0,1] vil være vanskeligere å beregne logaritme av enn et signal med dynamisk område innenfor [1,2] med lik presisjon. Grunnen til dette er formen på kurven kurven til logaritmefunksjonen. Som vist i figur 4 er logaritmekurven mye brattere og beveger seg over et større verdiområdet ved inngangsverdier mellom [0,1] enn ved inngangsverdier mellom [1,2]. Dette gjør det enklere å tilnærme logaritmefunksjonen ved hjelp av en approksimasjon i området over 1 enn i området [0,1]. Denne egenskapen gjelder samme hva slags logaritmebase som er i bruk, noe som vises i figur 4.



Figur 4: Logaritmekurver med forskjellig base

Presisjonen som kreves i utregningen varierer stort fra applikasjon til applikasjon. Hvis man skal regne ut en kodingsalgoritme eller noe annet som krever en meget stor presisjon vil man sannsynligvis velge en iterativ algoritme som bruker x-antall klokkesykler på utregningen av logaritme, og hvor presisjonen blir forbedret for hver iterasjon. En slik løsning kan oppnå meget

god presisjon men vil ha en forholdsvis svak ytelse, samtidig som at den ofte vil være ganske arealkrevende siden de fleste av disse løsningene forutsetter en forholdsvis stor multiplikator. Mange applikasjonen innenfor lyd- eller bildebehandling utnytter derimot det faktum at menneskets syns- og hørselsevnene har begrenset presisjon. Med begrenset presisjon menes det at approksimasjonsfeil ikke vil bli oppdaget, så lenge feilen er liten nok. I slike applikasjoner går ofte presisjonen på bekostning av ytelsen siden småfeil ikke er detekterbare for brukeren.

I dette kapittelet vil de aktuelle implementasjonsmetodene bli presentert hver for seg. Først vil selve tankesettet bak metodene bli presentert, før konkrete hardwareløsninger blir beskrevet der dette er aktuelt. Aktuelle metoder er metoder som regner ut logaritmen av et heltall eller fasttall i en klokkesykel, og som har potensiale til å være konkurransedyktige iforhold til arealbruk og ytelse. Ut fra disse kriteriene er det valgt å utelate metoder som er avhengige av store multiplikatorer, siden disse multiplikatorene alene gjør at et slikt design ikke vil bli konkurransedyktig på areal og ytelse (se vedlegg 3).

Dette kapittelet er en innføring i forskjellige tankesett, og ikke en dyptgående matematisk analyse av logaritmefunksjonen.

2.2.1 Polynomapproksimasjon og Rekkeutvikling

Polynom approksimasjon er en generell approksimasjonsmetode som går ut på å lage et polynom P(x) av grad *n*, som passer overens med funksjonen f(x) i *n* punkter. En slik approksimasjon er kun gyldig innenfor et begrenset område av x, og en høyere grad n vil gjøre approksimasjonen bedre innenfor det gitte dynamiske området. Rekkeutvikling er et spesialtilfelle innenfor polynom approksimasjon, og er den polynomapproksimasjonsmetoden som egner seg best til logaritme **[Deschamps06]**.

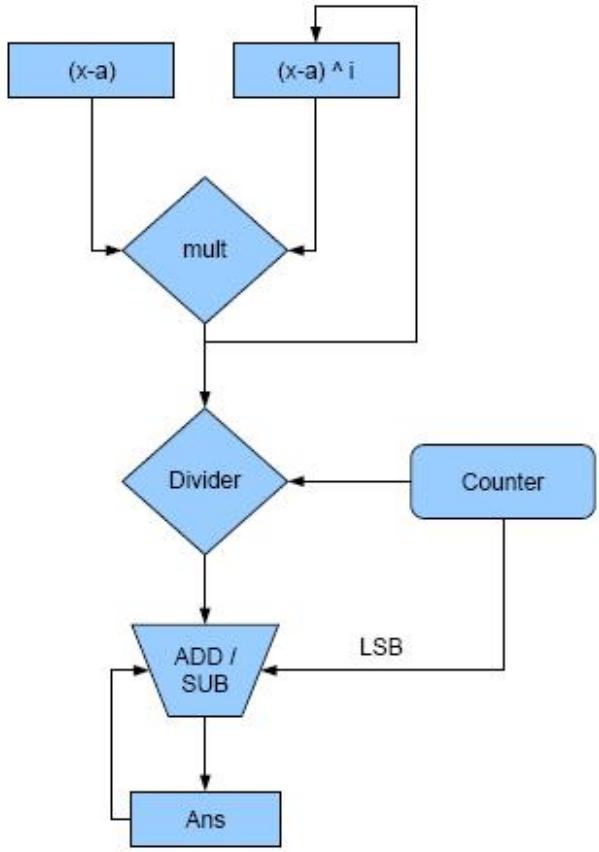
Den vanligste rekkeutviklingen av logaritme er Taylor-rekken

$$\ln(x) = \sum_{i=0}^{n} (-1)^{(i+1)} \frac{(x-a)^i}{i} + R_n$$

hvor R_n er approksimasjonsfeilen som går mot null når antall ledd *n* går mot uendelig. Taylor-rekken konvergerer mot *f(x)* rundt punktet *a*, og er en dårligere tilnærming jo lengre vekk fra punktet *a* inngangsverdien x beveger seg. Av den grunn så sies det at denne formelen bare er gyldig

for inngangsverdier innenfor et gitt område, noe som gjør at inngangsverdiene må normaliseres til å passe innenfor dette området før de kan behandles med en Taylor-rekke.

Denne rekken bruker veldig mange ledd for å konvergere mot riktig verdi og blir dermed sett på som meget ineffektiv. Gjennom forsøk i Matlab har det vist seg at for inngangsverdier i med dynamisk område [1,2] så holder det med 72 ledd for å få en nøyaktighet på 99% . Dette antallet gjelder uansett oppløsning på grunn av den forutsigbare formen på kurven i dette verdiområdet. Om inngangssignalet har verdier i området [0,1] derimot så er antall ledd som trengs for å oppnå en viss presisjon sterkt avhengig av oppløsningen på signalet. Ved en 16-bits oppløsning trengs det i dette tilfellet 93780 ledd for å oppnå 99% nøyaktighet. Denne egenskapen kommer av at den minste verdien i inngangssignalet tilsvarer en LSB, som er nærmere null jo flere bits presisjon det er i desimaldelen.



Figur 5: Flytskjema for en hardwareimplementasjon av Taylor rekkeutviklingen til logaritme.

En av de vanligste måtene å skrive rekken på er å tilpasse funksjonen for å konvergere rundt punktet x=1. Funksjonen blir da

$$\ln(x) = \sum_{i=1}^{n} (-1)^{(i+1)} \frac{(x-1)^i}{i}$$

med gyldighetsområde tilsvarende

$$0 < x \leq 2$$

2.2.1.1 Hardware

En rekkeutvikling kan enten implementeres som en iterativ algoritme som vist i figur 5, eller den kan pipelines som en lang rekke. Ved å lage en iterativ arkitektur blir de matematiske operatorene gjenbrukt, og man sparer på denne måten mye areal. I tillegg så er en slik implementasjon fleksibel i forhold til nøyaktighet, siden det er valgfritt hvor mange ledd man velger å ta med. Ved å implementere denne rekkeutviklingen som en lang pipeline vil man kunne få en langt høyere ytelse, men selve implementasjonen vil være langt mindre fleksibel og oppta et mye større areal.

Men siden en slik ren bitparallell-implementasjon forutsetter en multiplikator og en divisjonskrets, vil denne implementasjonen bli meget tung målt i både tid og areal. En en-sykel multiplikator er alene tregere og mer arealkrevende enn den lineære approksimasjonsmetoden for logaritme (se vedlegg 3). I tillegg kommer divisjonen som tradisjonelt har dårligere ytelse enn multiplikasjon i hardware. På grunn av disse ulempene er en ren Taylor-rekke uegnet for hardwareimplementasjon ved de gitte kriteriene, men brukes isteden noe ganger som en softwareimplementasjon i systemer som har multiplikasjon og divisjon tilgjengelig, eller i systemer som krever høy presisjon og hvor kravet til presisjon kan gå på bekostning av areal og ytelse. Av den grunn er Taylor-rekken ikke implementert i hardware i denne oppgaven.

2.2.2 Logaritme ved konvergens

Konvergensmetoder består av to parallelle prosesser med to relaterte sekvenser hvor den ene sekvensen konvergerer mot 1 (multiplikativ normalisering) eller 0 (additiv normalisering), mens den andre sekvensen konvergerer mot funksjonstilnærmingen. Funksjonstilnærmingen er i dette tilfellet logaritmen av inngangssignalet. Som et eksempel på en konvergensmetode kan man forklare multiplikativ normalisering med å først definere den *multiplikative normaliseringsfunksjonen* som

$$c(i) = 1 + a_i * 2^{-i} \quad , \quad a_i \in [-1, 0, 1]$$

hvor a_i er valgt på en slik måte at

$$x(i+1) = x(i) * c(i) \quad , \quad x(i) \in B(2^n)$$

konvergerer mot 1 ved økende *i* **[Deschamps06]**. Da kan sekvensen

$$y(i+1)=y(i)-\ln(c(i))$$

bli satt til å konvergere mot resultatet ln(x). Hvis y(0) og x(0) blir satt til respektivt 0 og x, samtidig som vi antar at x(p) ≈ 1 , så blir

$$x(p)=x*\Pi_i c(i)=1\rightarrow\frac{1}{x}=\Pi_i c(i)$$

$$y(p)=y-\Sigma_i c(i)=-\ln\Pi_i c(i)=\ln(x)$$

For å gjøre denne konvergeringen gyldig må x ha en verdi slik at

$$[x*min(\lim_{(p\rightarrow\infty)}\Pi_{(1\leq i\leq p)}c(i))]\leq 1$$

og

$$[x*max(\lim_{(p\rightarrow\infty)}\Pi_{(1\leq i\leq p)}c(i))]\geq 1$$

For å oppfylle disse kriteriene må x være innenfor området

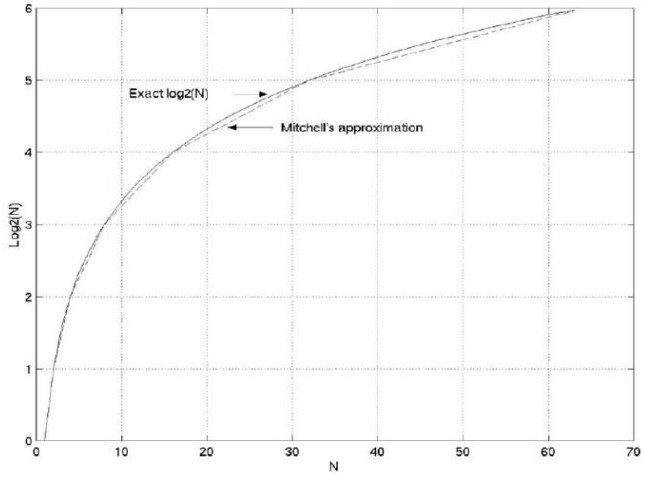
$$0.42\leq x\leq 3.45$$

Dette betyr at det kan være behov for å skalere inngangsverdien x dersom denne går utover dette verdiområdet, men et tall innenfor [1,2] (som for eksempel x definert i området [0,1] i formelen log(1+x)) passer utmerket.

En konvergensmetode er en iterativ algoritme hvor man beveger seg nærmere det korrekte svaret for hver iterasjon. På denne måten kan man selv bestemme presisjonen ved å justere antall iterasjoner man velger å bruke. Presisjonen man oppnår ved en slik metoden er typisk ett bit per iterasjon **[Deschamps06]**. Siden dette er en metode som brukes for å oppnå høy presisjon er det vanlig å ha forholdsvis mange iterasjoner i en slik utregning, og av den grunn har denne algoritmen meget lav ytelse. En hardwareimplementasjon av en konvergensmetode vil ha stor logisk

forsinkelse på grunn av at begge normaliseringsmetodene er avhengige av en multiplikator **[Deschamps06]**, samt at metoden bruker et visst antall klokkepulser på en utregning. Disse egenskapene gjør at denne metoden ikke er regnet som konkurransedyktig innen ytelse og areal, og er derfor ikke implementert i VHDL i denne masteroppgaven.

2.2.3 Lineær approksimasjon



Figur 6: Illustrasjon av Mitchells tilnærming. ***[Abed03]***

Lineær approksimasjon er en meget hardwarenær måte å ta logaritmen av et tall på. Denne metoden bygger ikke direkte på matematiske beviser, men heller en intuitiv tolkning av logaritme. Ideen er at man kan telle antall siffer i heltallsdelen, og så bruke dette antallet som heltallsdelen i svaret. Deretter setter man et komma etter den ledende eneren i inngangssignalet, og bruker alt til høyre for kommaet som desimaltall i svaret. På denne måten lager man en ren stykkevis lineær tilnærming som vist i figur 6. Med denne fremgangsmåten tar man utgangspunkt i en logaritme med base 2. Hvis det er ønskelig med en annen base vil det enkleste være å multiplisere med en konstant for å skifte base etter at konverteringen er fullført. En slik ren lineær tilnærming blir kalt Mitchells tilnærming, siden Mitchell var den første til å foreslå en slik approksimasjonsmetode **[Mitchell62]**. Mitchell sitt arbeid ble gitt ut i 1962, og de fleste logaritmeimplementasjoner som baserer seg på lineær approksimasjon bygger på Mitchell sitt arbeid. De fleste reelle logaritmeimplementasjoner har en eller flere korreksjonsmekansimer som gjør tilnærmingen bedre enn Mitchell sin tilnærming. Mitchell sin tilnærming er bare korrekt i punktene 2^N og er helt lineær mellom disse punktene.

Mitchell oppsummerer binær logaritme på følgende måte:
La N være et binært tall i området $2^j \leq N \leq 2^k$, hvor j og k er heltall, og $k \geq j$. Da kan N skrives som $N=z_k$ ..... $z_3\ z_2\ z_1\ z_0\ z_{-1}\ z_{-2}$ ... z_j. Matematisk uttrykkes da N ved

$$N = \sum_{i=j}^{k} 2^i * z_i$$

hvor z er en binær verdi, altså med verdien '1' eller '0'. Om vi antar at MSB er 1 så kan man skrive om N til

$$N = 2^k \sum_{i=j}^{k-1} 2^i * z_i = 2^k (1 + \sum_{i=j}^{k-1} 2^{(i-k)} * z_i)$$

Ved å representere

$$m = \sum_{i=j}^{k-1} 2^{(i-k)} * z_i$$

vil m være alle bit'ene til høyre for den ledende eneren i N. Siden $k \geq j$ vil m være i intervallet $0 \leq m < 1$

Logaritmen kan da skrives om til:

$$\log_2(N) = \log_2(2^k(1+m)) = k + \log_2(1+m)$$

Mitchell definerte i **[Mitchell62]** en lineær approksimasjon som sier at

$$\log_2(N)' = k + m$$

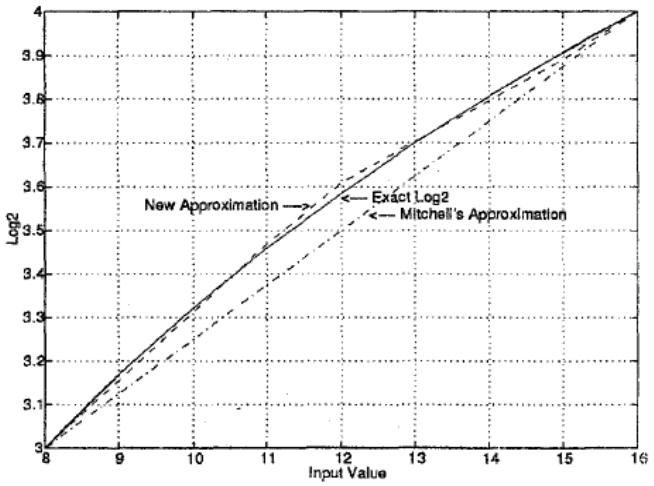
og erstattet $\log_2(N)$ med $\log_2(N)'$ i sine beregninger. Forskjellen mellom disse kurvene kommer frem i figur 6 og i figur 7. Ved å bruke den lineære approksimasjonen $\log_2(N)'$ istedenfor $\log_2(N)$ innføres det en approksimasjonsfeil tilsvarende

$$Error = \log_2(N) - \log_2(N)' = \log_2(1+m) - m$$

Denne feilen kan i noen tilfeller være betydelig, så en viss feilkorreksjon kan med fordel benyttes i en slik implementasjon.

2.2.3.1 Hardware

Lineær approksimasjon er en av de mer populære hardwareimplementasjons-løsningene innenfor DSP, så det er publisert flere arbeider innenfor dette området. Felles for de alle er at de bygger på Mitchell sin binær-logaritme teori som her blir presentert først.



*Figur 7: Forskjellige approksimasjonskurver, hvor **[SanGregory99]** er merket 'New Approximation'.*

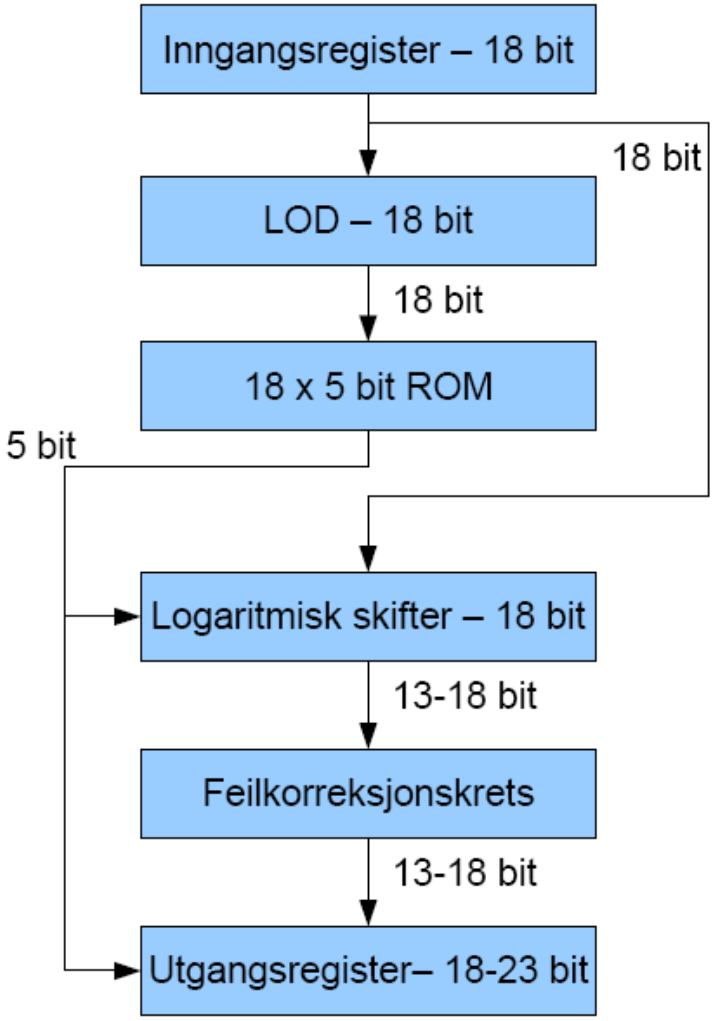
2.2.3.1.1 Mitchell sin binær-logaritme teori

Allerede i 1962 kom den første teorien om hvordan man kunne forenkle logaritme med lineær approksimasjon i et 2-tallsystem for å forenkle utregningen på en datamaskin. Denne metoden ble presentert av Mitchell som et ledd i en logaritmisk multiplikasjons og divisjons-metode som forenklet multiplikasjon og divisjon til henholdsvis addisjon og subtraksjon **[Mitchell62]**. Mitchell sin metode baserte seg på en lineær tilnærming mellom alle punktene 2^N hvor N er heltall. Dette ble utført ved å først finne den ledende eneren, for så å bruke posisjonen til den ledende eneren som heltall i svaret. Alle siffer bak den ledende eneren ble brukt som desimaltall i svaret, og man oppnådde da en lineær tilnærming mellom heltallspunktene i log2. Denne metoden gir en logaritme med base 2, men konvertering mellom baser er bare en multiplikasjon med en konstant som vist i formelen under.

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

Det er dermed ganske lett å skifte fra en base til en annen base om dette skulle være ønskelig. **[SanGregory99]** viser at det kun trengs tre addisjoner for å ga fra base 2 til base 10 med en akseptabel presisjon.

De fleste lineære approksimasjonsmetoder for logaritme bygger på Mitchell sin metode. Men **[Mitchell62]** innfører en ganske betydelig feil i svaret, og algoritmene som bygger på denne har som regel et ekstra ledd på slutten for å korrigere denne feilen. Det er viktig å merke seg at alle metodene som bygger på Mitchell sin lineære approksimasjonsmetode innfører en viss feil i logaritmeberegningen, så denne metoden er kun egnet i applikasjoner der slike små feil tolereres for å prioritere ytelse, som for eksempel i visse DSP-applikasjoner.



Figur 8: Flytskjema for en logaritmekrets basert på Mitchells approksimasjonsmetode. Bitbredde er angitt som eksempler.

2.2.3.1.2 SanGregorys korreksjonsmetode

SanGregory, Brothers og Gallagher (SBG) utga i 1999 en logaritmeapproksimasjon med fokus på lav effekt **[SanGregory99]**. Algoritmen er en lineær approksimasjonsalgoritme og bygger dermed direkte på Mitchell sitt arbeid fra 1962. Denne approksimasjonsmetoden innfører i likhet med **[Mitchell62]** en liten feil i svaret, men feilen som blir innført er vesentlig mindre. Metoden presentert i **[SanGregory99]** er ikke avhengig av noen multiplikasjoner, men trenger en liten ROM tilsvarende (N x $\log_2$N)-bit for en N-bit logaritme.

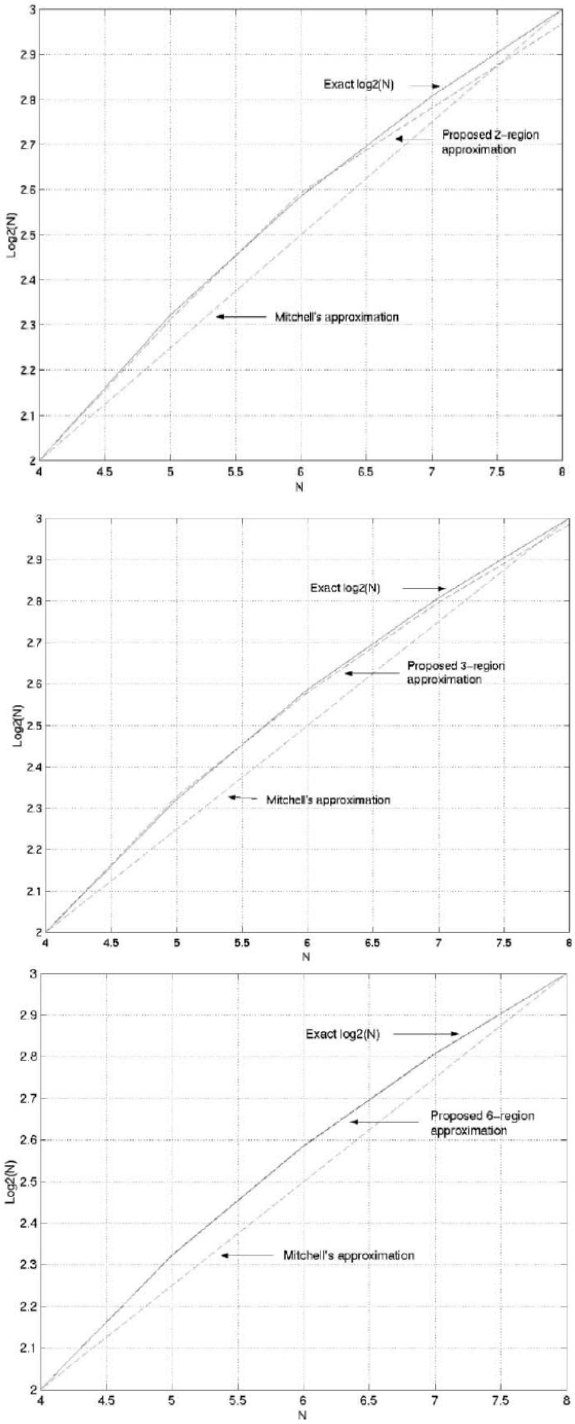
SBG sin algoritme bruker flere lineære linjestykker enn Mitchell i sin tilnærming. Disse ekstra linjestykkene innføres ved å dele inn hvert område tilsvarende 2^i til $2^{(i+1)}$ i to separate områder med hver sin lineære tilnærming. Deretter økes stigningstallet til approksimasjonskurven i den første halvdelen ved å multiplisere Mitchellapproksimasjonen med en konstant faktor på 1.125. Dette gjøres ved å addere desimaldelen med seg selv etter å ha skiftet denne delen tre plasser til høyre. SBG fant ut ved empiriske forsøk at denne faktoren var den mest passende og fjernet det meste av feilen som Mitchell sin tilnærming innførte. I den andre halvdelen må stigningstallet være mindre enn for Mitchell sin tilnærming. Dette gjøres ved å subtrahere den skiftede verdien fra Mitchellapproksimasjonen fremfor å addere den skiftede verdien med Mitchellapproksimasjonen. Ved å invertere bit'ene før addering oppnår man denne effekten. Forskjellen mellom approksimasjonskurven til Mitchell og til **[SanGregory99]** kommer tydelig frem i figur 7. **[SanGregory99]** beskriver også en måte å dele inn hvert 2^i til $2^{(i+1)}$ område i fire separate områder, men en slik ytterligere oppdeling ga tilnærmet ingen gevinst med SBG sin feilkorreksjonsmetode, så den ble forkastet før implementasjonen. Vedlegg 4 viser oppbygningen til feilkorreksjonskretsen og LOD-kretsen til **[SanGregory99]**.

SBG sin metode baserer seg på at MSB i desimaldelen av svaret til Mitchell skifter fra 0 til 1 akkurat der SBG ville ha knekkpunktet i kurven sin, så i algoritmen presentert i **[SanGregory99]** bestemmer dette bit'et stigningstallet på den lineære kurven. **[SanGregory99]** kan derfor sees på som en korreksjonsalgoritme til Mitchell sin approksimasjon.

Feilen som denne algoritmen innfører har en mye mindre RMS-verdi enn Mitchell sin algoritme, både på grunn av at feilen har en mindre amplitude og på grunn av at feilen har et alternerende fortegn.

2.2.3.1.3 Abeds korreksjonsmetoder

Året etter kom Abed og Siferd ut med en ny logaritmeimplementasjon som også bygde på Mitchell sin algoritme fra 1962 **[Abed00]**. Abed og Sifred hadde fullt fokus på hastighet, og de lagde både en logaritmisk og en antilogaritmisk konverter basert på **[Mitchell62]** uten ytterligere feilkorreksjon. Det tregeste steget i Mitchell sin algoritme er å finne den ledende eneren. Abed og Sifred foreslo en ny og raskere arkitektur for denne modulen. Mens tidligere implementasjoner, som for eksempel **[SanGregory99]**, baserte seg på ren seriell behandling av inngangsverdiene, er Abed og Sifred sin metode seriell/parallell. En slik metode går ut på å dele inn inngangssignalet i grupper, for så å behandle signalene serielt innad i gruppene, og alle gruppene parallelt med hverandre. Denne arkitekturen er vist i vedlegg 5. **[Abed00]** er altså meget rask, men innfører i likhet med **[Mitchell62]** en betydelig feil i svaret siden den ikke benytter seg av et feilkorreksjonssteg.



*Figur 9: Illustrasjon av approksimerte stykkevis lineære kurver med henholdsvis 2, 3 og 6 regioner. Den reelle logaritmekurven samt Mitchell sin approksimasjon er plottet inn for sammenligning. Fra **[Abed03]** .*

I 2003 lanserte Abed og Sifred en forbedret versjon **[Abed03]** hvor de tok med et feilkorrigeringsledd på sisten av signalkjeden, slik som vist i figur 8. Denne versjonen var i realiteten en kopi av **[Abed00]** frem til korreksjonsenheten. **[Abed03]** beskriver flere forskjellige versjoner av korreksjonsenheten, som alle bygger på tanksettet til **[SanGregory99]**. Korreksjonsenheten i **[SanGregory99]** gikk ut på å dele inn hvert område mellom to tilstøtende 2^n punkter i to nye stykkevis lineære områder. På denne måten ble approksimasjonskurven bedre tilpasset den reelle

logaritmekurven. **[Abed03]** tar for seg tre nye feilkorreksjonsmetoder, i tillegg til **[SanGregory99]** som blir brukt som sammenlikningsgrunnlag. De tre metoden deler hvert lineære område fra Mitchell sin approksimasjon inn i henholdsvis to, tre og seks nye stykkevis lineære områder. Figur 9 illustrerer hvordan disse approksimasjonskurvene ser ut sammenlignet med den reelle log2-kurven og Mitchell sin stykkevis lineære approksimasjonskurve. Vedlegg 6 viser hvordan de forskjellige korreksjonskretsene er bygd opp, og dermed også hvor komplekse de er.

Korreksjonskretsene bygger på det samme prinsippet som **[SanGregory99]**. For å øke stigningstallet på kurven så legges det til en variabel, som hele tiden teller oppover. Denne variabelen er inngangssignalet selv, men multiplisert med en konstant (1/8). For å senke stigningstallet på kurven så adderes en variabel som hele tiden teller nedover. For å generere denne variabelen så brukes inngangssignalet som nok en gang er multiplisert med den samme variabelen, men som nå er invertert for dermed å utføre en subtraksjon istedenfor en addisjon. Hvordan dette løses implementasjonsmessig er vist i vedlegg 6, og grundig forklart i **[Abed03]**.

Alle korreksjonsenhetene baserer seg på å korrigere stigningstallet ved å multiplisere svaret fra Mitchellapproksimasjonen med en konstant, men ingen av korreksjonsenheten bruker alle bit'ene i Mitchellapproksimasjonen til denne korrigeringen. Denne beslutningen er en avveining mellom presisjon og ytelse/areal. Presisjonen i korrigeringen øker ved antall bit som er i bruk, men samtidig så øker også arealet og ytelsen går ned. Hvor mange korreksjonsbit de forskjellige metodene bruker kan lese ut fra tabell 9.

2.2.4 Lookup Table (LUT)

En LUT er en enkel ROM-mapping hvor inngangsignalet blir brukt som indeks i en ROM for å slå opp riktig verdi. Om man bruker en-til-en kobling i en slik LUT vil man måtte lagre 2^n forskjellige dataord i ROMen, hvor n står for antall bit i inngangssignalet. Siden lagringskapasiteten, og da også arealet, er eksponentielt avhengig av antall bit i inngangssignalet, vil dette være en dårlig løsning for inngangssignal med stort dynamisk område. I realiteten vil man i en logaritmefunksjon ha flere like verdier, og ved å utnytte denne egenskapen vil kravet til lagring bli mindre enn 2^n forskjellige dataord. Men lagringskapasiteten som behøves er fortsatt så stor at en ren LUT-implementasjon sjelden vil være en god løsning.

2.2.5 Alternative metoder

Om det ikke er ønskelig å dedikere hardware til å regne ut logaritmen så er det fullt mulig å tilnærme logaritmen på en mikrokontroller, DSP eller eventuelt en CORDIC-enhet.

En mikrokontroller inneholder ofte meget avanserte ALUer som har god ytelse. Ved å skrive en kompakt mikrokontrollerkode vil det være mulig å regne ut logaritme på en mikrokontroller forholdsvis effektivt, og en slik implementasjon vil også være meget fleksibel siden koden ikke er like statisk som en ferdig syntetisert hardware-modul. Ulempen er at ytelsen vil bli mye dårligere enn en dedikert hardwaremodul siden ALUen ikke er så godt egnet for den type operasjoner, men om ytelsen ikke er kritisk så kan en mikrokontroller fortsatt være en god løsning. En fordel med en mikrokontrollerløsning er at man kan velge en iterativ algoritme, og på den måten oppnå ønsket presisjon på utregningen.

En DSP er en generell, men mindre fleksibel enhet enn en mikrokontroller. DSPer har også et større strømtrekk og opptar et større areal i designet. På den annen side vil den kunne utføre en logaritme-approksimasjon raskere enn en mikrokontroller, mye på grunn av en mer parallellisert struktur og en mer optimalisert arkitektur for raskere behandling av bitstrøm. Dette kan være en løsning om en DSP allerede er tilgjengelig i systemet som har behov for en logaritmefunksjon og ytelsen ikke er kritisk. En fordel med en DSP-løsning er at man i likhet med mikrokontrollerløsningen kan velge en iterativ algoritme, og på den måten oppnå ønsket presisjon på utregningen.

En CORDIC-enhet er en iterativ kontroller med en innebygd ALU som kun utfører shift, add/sub og sammenligning. Basert på disse operasjonene har en CORDIC-enhet en struktur som gjør den meget effektiv til å tilnærme trigonometriske funksjoner. Den har også muligheter for å utføre logaritme-beregninger, men er mindre effektiv på dette området da den er optimalisert med hensyn på trigonometri.

Det er også verdt å nevne her at det finnes flere forskjellige hardwareløsninger som ikke er nevnt i dette delkapittelet, og som er utelatt på grunn av at de ikke passer inn i profilen som er valgt. Det er kun en-sykel logaritmeløsninger for inngangsverdier i heltall og fastpunkts formater som her er beskrevet. Metode for høy radiks som **[Pineiro02]** og **[Pineiro04]** og metoder for flyttall som **[Detrey05]** samt bit-serielle metoder er derfor ikke tatt med i denne gjennomgangen.

2.2.6 Oppsummering av logaritmebetraktningene

De aktuelle løsningene skal bli implementert i hardware og sammenlignet med hverandre med tanke på feiltoleranse, timing- og arealresultater. Ut fra kriteriene om at presisjon kan offres på bekostning av areal og ytelse samt at det ikke skal brukes multiplikatorer, er lineær approksimasjon den mest relevante implementasjonsmetodikken for logaritme i denne masteroppgaven. Disse kriteriene er veldig vanlige å sette i audio- og bildeapplikasjoner og den lineære approksimasjonsmetoden er derfor den mest utbredte metoden innenfor DSP-applikasjoner **[Abed03]**. Alle metodene som er beskrevet under lineær approksimasjon er derfor implementert i VHDL, og disse implementasjonene blir nærmere gjennomgått i kapittel 3.2.1.4.

Dette kapittelet har omhandlet teorien rundt planleggingsstadiet til hardwareimplementasjonen av dynamikkompresjonsalgoritmen med fokus på analyse av kompresjonsalgoritmen samt forskjellige implementasjonsmetoder for logaritme. Neste kapittel vil ta for seg arkitekturen til hardwaremodulen som skal designes, og hvordan de forskjellige undermodulene bygges opp.

3 Design

Dette kapittelet beskriver designprosessen som er gjennomført for å få realisert en hardwareimplementasjon av Reinhard Fotoreseptormodell. Først blir de overordnede arkitekturene skissert gjennom en top-down metodikk. Deretter blir designet dokumentert på modulnivå gjennom en bottom-up metodikk ved å beskrive de atomære operasjonene før de sammensatte modulene. Til slutt oppsummeres arbeidet ved å beskrive arbeidsmetodikken som er brukt i denne masteroppgaven, samt de forskjellige valgene som er tatt underveis i arbeidet.

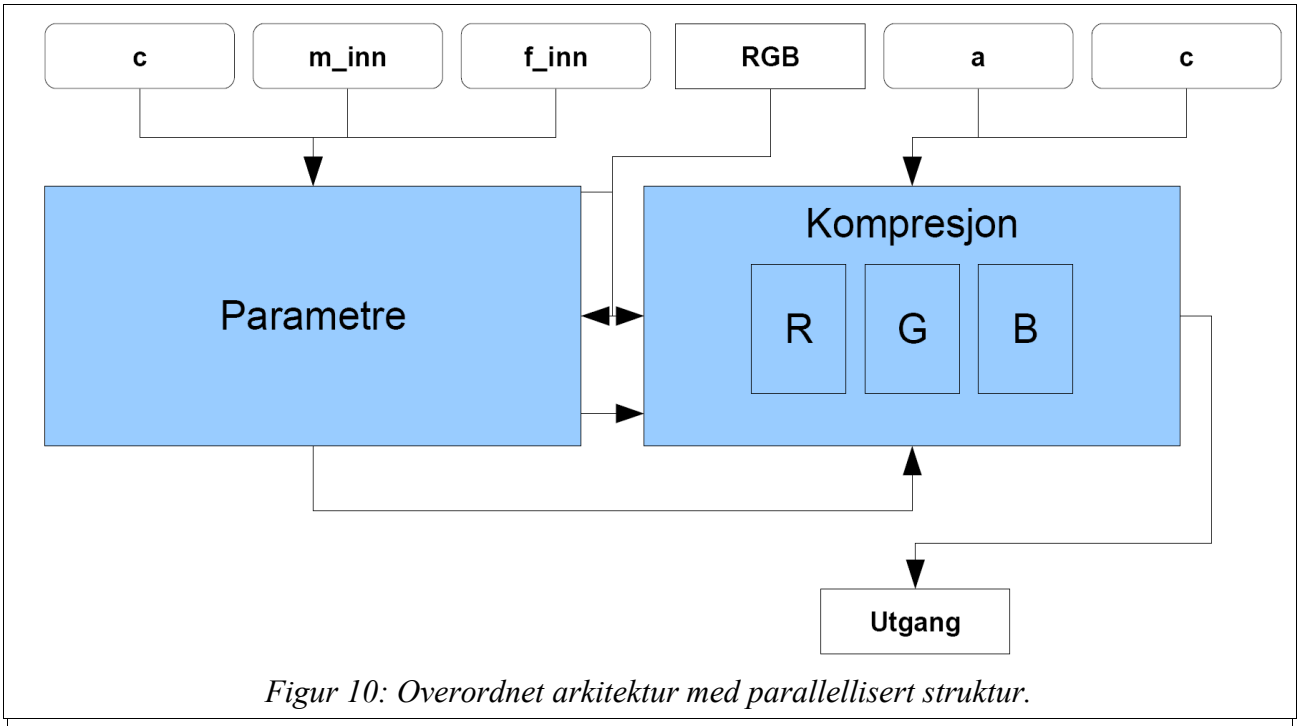
3.1 Arkitektur

Arkitekturen til hardwareenheten er avgjørende for ytelse og arealbruk, og for å ivareta størst mulig fleksibilitet skal arkitekturen til dynamikkkompresjonsenheten være modulbasert. På denne måten blir det lettere å sette sammen forskjellige versjoner og å sammenligne de forskjellige konstellasjonene med tanke på areal og ytelse.

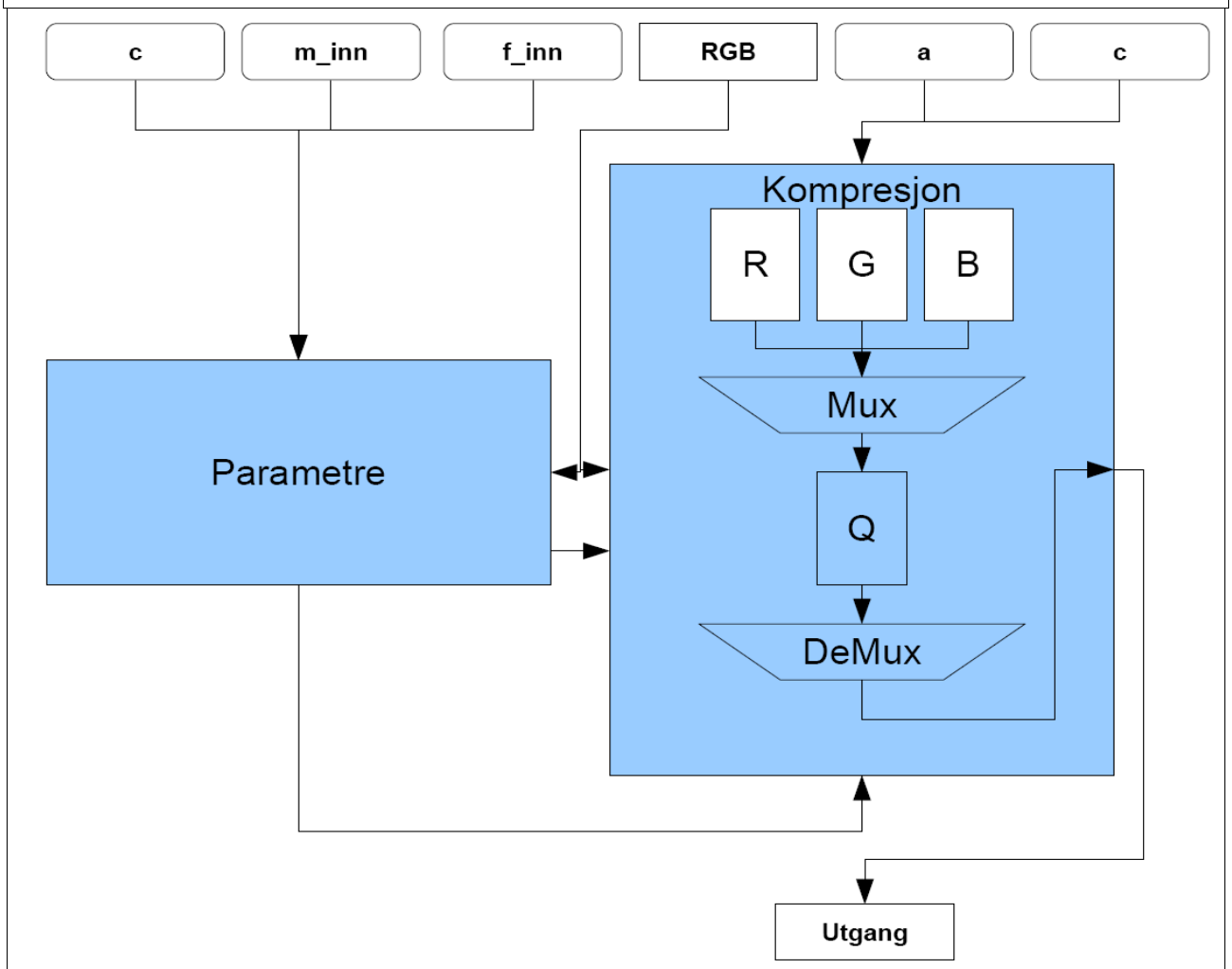
I dette delkapittelet blir først de overordnede arkitekturene presentert før signalkompresjonsdelen og parameterberegningene blir presentert hver for seg. Til slutt blir undermodulene i kompresjonskjeden og parameterberegningene presentert teoretisk.

3.1.1 Overordnet arkitektur

Som nevnt innledningsvis vil det være flere versjoner av den overordnede arkitekturen, for så å sammenligne areal- og ytelsesresultatene til slutt. Ved å gjøre en slik analyse er det lettere å se hvor mye parallellisering som lønner seg i en bestemt applikasjon med bestemte ytelses- og/eller arealkrav. Den ene versjonen vil ha en parallellisert struktur hvor hver farge har en egen kompresjonsenhet, mens den andre versjonen vil ha en enkelt kompresjonsenhet som vil bli delt mellom de tre forskjellige fargene. På denne måten vil den ene arkitekturen prioritere ytelse på bekostning av areal, men den andre strukturen prioriterer motsatt. Figur 10 og figur 11 viser de overordnede arkitekturene med henholdsvis parallellisert struktur og seriell tidsmultiplekset struktur. I disse figurene representerer de hvite rammene registre, mens de andre blokkene representerer forskjellige operasjoner. Enheten som genererer kompresjonsparametrene er lik i begge tilfellene, og tilsvarer de blokkene innenfor den stiplede firkanten i figur 13.



Figur 10: Overordnet arkitektur med parallellisert struktur.



Figur 11: Overordnet arkitektur med en enkelt kompresjonsenhet.

I den parallelliserte enheten i figur 10 representerer den blå kompresjonsboksen tre separate kompresjonsenheter, hvor hver av enheten inneholder de operasjonene som er innenfor den stiplede firkanten i figur 12. Disse kompresjonsenhetene er dedikert til hver sin farge, og alle de tre enhetene får hvert sitt sett med inngangsparametre.

Arkitekturen med den enkle kompresjonsenheten i figur 11 er lik arkitekturen i figur 10 bortsett fra selve kompresjonen. Siden denne versjonen bare inneholder en kompresjonsenhet, merket Q i selve figuren, må denne enheten deles mellom de tre fargene. Dette gjøres ved tidsmultipleksing slik at fargene blir komprimert etter hverandre i en forhåndsbestemt rekkefølge.

3.1.2 Signalflyt og kompresjonen

I dynamikkompresjonen er det selve verdien for hvert piksel i bildet som komprimeres, og for å oppnå høy ytelse er det derfor meget viktig at denne signalveien er optimalisert. Figur 12 illustrerer hvilke operasjoner som gjøres i dynamikkkompresjonen av signalet, og hvilke parametre denne kompresjonen er avhengig av. Matlabkoden i vedlegg 2 viser hvilke matematiske operatorer som er i bruk i hver enkelt operasjon i figur 12. Formlene nedenfor fremhever hvilke matematiske operasjoner som skjer i hvert av de forskjellige leddene som er illustrert som separate bokser i figur 12.

Selve kompresjonen er en enkel divisjon hvor kompresjonsfaktoren σ bestemmer andelen kompresjon for hvert enkelt piksel, og som selv er avhengig av informasjonsinnholdet i bildet.

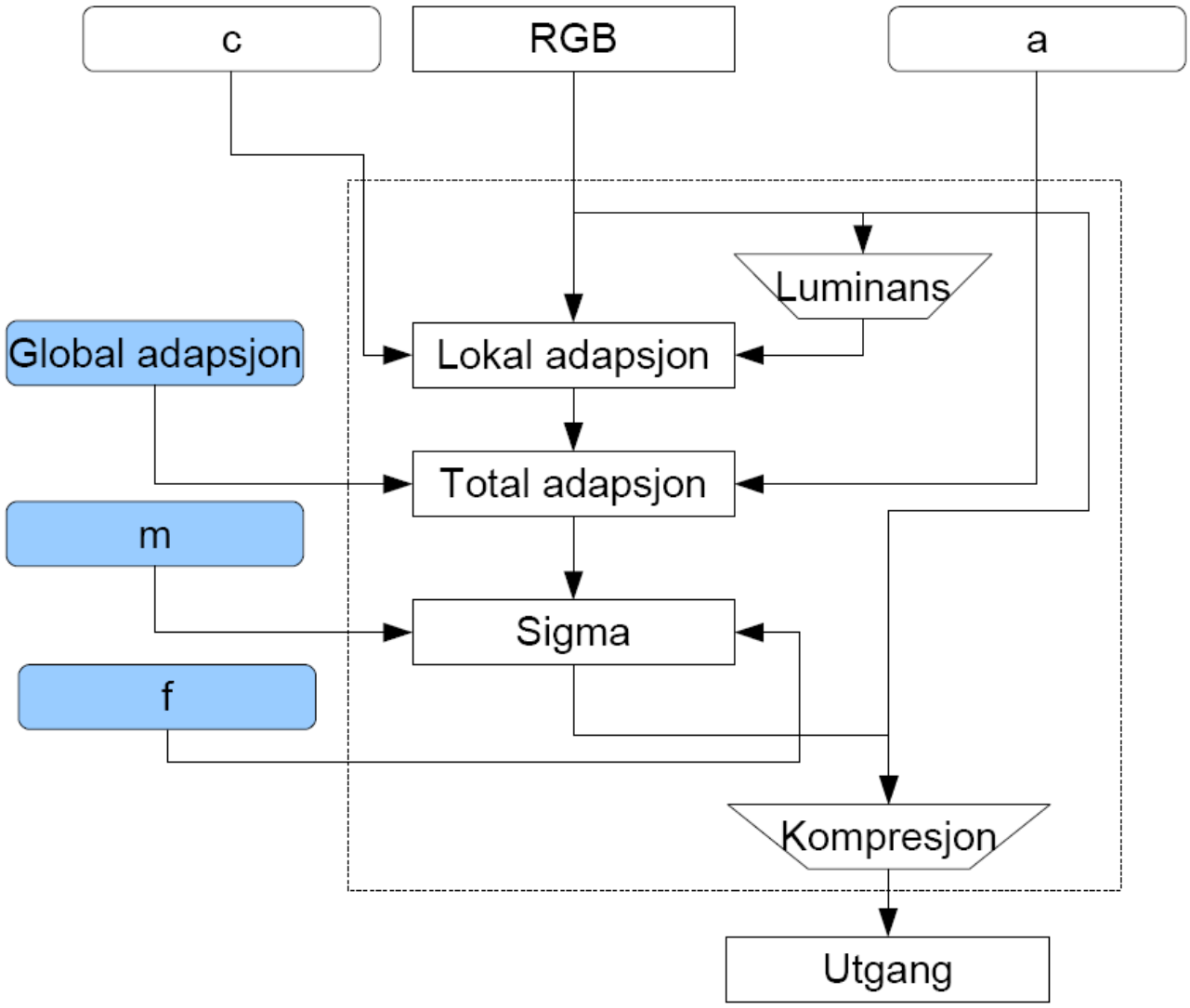
$$KomprimertPiksel_i = \frac{PikselRådata_i}{PikselRådata_i + \sigma_i}$$

$$Sigma = \sigma_i = ((2^f) * Ia_i)^m$$

$$Total\ adapsjon = Ia_i = a * Ia_{lokal,i} + (1-a) * Ia_{global}$$

$$Lokal\ adapsjon = Ia_{lokal,i} = c * RGB_i + (1-c) * Luminans_i$$

$$Luminans_i = 0.2125 * R + 0.7154 * G + 0.0721 * B$$



Figur 12: Viser signalflyten for inngangssignalet som skal komprimeres (RGB). Den delen innenfor de stiplede linjene implementeres i en felles hardwaremodul som står for selve signalflyten. Modulene utenfor de stiplede linjene er registre og parametre som enten er brukerstyrte eller estimerte verdier. De hvite parametrene er inngangsparametre mens de blå er estimerte parametre.

Separat fargebehandling sikrer god fargebalanse i bildet, men siden hver farge i hvert piksel må komprimeres separat bruker denne fremgangsmåten mye ressurser. En av avgjørelsene som må taes i designprosessen er om disse ressursene skal brukes i form av tid eller areal. Ved parallellisering av fargebehandlingen kan alle de tre fargene komprimeres parallelt på bekostning av arealet. På den annen side kan algoritmen implementeres med kun en fargekompresjonsenhet (de operasjonene innenfor de stiplede linjene i figur 12) og isteden tidsmultiplekse bruken av denne. En slik løsning vil gå på bekostning av ytelsen siden antall piksler komprimert per sekund vil gå ned med en faktor på tre. I denne masteroppgaven er det valgt å implementere begge løsningene, for så å sammenligne

ytelses- og arealresultatene til slutt.

3.1.3 Parametre

Dette delkapittelet tar for seg inngangsparametrene og kompresjonsparametrene til algoritmen. Inngangsparametrene til algoritmen er beskrevet i tabell 3, og de interne kompresjonsparametrene er beskrevet i tabell 4. Presisjonen til parametrene er ikke definert i den opprinnelige algoritmen, og de presisjonene som er definert her er valgt på bakgrunn av en avveining mellom bitbredde/arealbruk og nødvendig presisjon for å oppnå ønsket visuell effekt. Den visuelle effekten er vurdert ved forsøk i Matlab.

Tabell 3: Inngangsparametre i Reinhards Fotoreseptormodell

Parameter	***Beskrivelse***	***Initiell verdi***	***Operasjonsintervall***	***Presisjon***
m_inn	Kontrast	0 {0.3 + 0.7*$k^{1.4}$ [1]}	[-1, 1]	2^{-13}
f_inn	Intensitet	0 {1.375 * Lav}	[-18, 18]	2^{-13}
c	Kromatisk adapsjon	0.0	[0.0, 1.0]	2^{-3}
a	Lys adapsjon	1.0	[0.0, 1.0]	2^{-3}

[1] k er en variabel som regnes ut fra maksimum, minimum og gjennomsnittlig luminans i bildet.

Effekten av inngangsvariablene til kompresjonsalgoritmen er intuitive men operasjonsintervallet og initialverdiene er ikke like intuitive. Spesielt når inngangssignalet har et stort dynamisk område, så vil også inngangsvariabelen *f_inn* ha et forholdsvis stort operasjonsintervall. *f_inn* bestemmer andelen kompresjon og dermed den generelle oppfattede lysheten i bildet. På grunn av hvordan parameteren *f_inn* blir brukt i algoritmen, må *f_inn* være i størrelsesorden antall bit til gjennomsnittsluminansen i bildet. Det er lagt til en funksjon i algoritmen som setter *f* til 1.375 ganger den logaritmiske gjennomsnittsluminansen i bildet, og *f_inn* justerer *f* ut fra denne initialverdien. Initialverdien er bestemt ut fra hvilke verdier som gir best visuelt resultat gjennom forsøk i Matlab. Algoritmen selv setter ingen begrensning for operasjonsintervallet til parameteren *f_inn*, men for hardwareimplementasjonen er det definert et operasjonsintervall tilsvarende heltall innenfor området [-18, 18] siden dette stemmer overens med den definerte bitbredden på inngangssignalet.

Kromatisk adapsjon og lysadapsjon er parametere som bestemmer hvor mye av bildebehandlingen som skal være bestemt av globale farge- og lysforhold iforhold til lokale farge-

og lysforhold. Vedlegg 7 illustrerer hvordan utseendet på bildet varierer i takt med endringer i disse parametrene. For hardwareimplementasjonen er skrittlengden til disse parametrene satt til 2^{-3}.

For å beregne kontrastparameteren *m* er det gitt en funksjon i **[Reinhard05]**. Denne funksjonen er det anbefalt å holde seg til i **[Reinhard05]**, da de initielle tallverdiene til denne parameteren ikke er intuitive. Inngangsparameteren *m_inn* vil da inkrementere eller dekrementere denne initialverdien etter brukerens ønske.

Både kontrastparameteren *m* og intensitetsparameteren *f* har beregningsmetoder for å finne initialverdier, mens brukerparametrene *f_inn* og *m_inn* justere disse initialverdiene. Denne justeringen skjer ved at brukeren bestemmer relative verdier som inkrementerer eller dekrementerer de initielle verdiene.

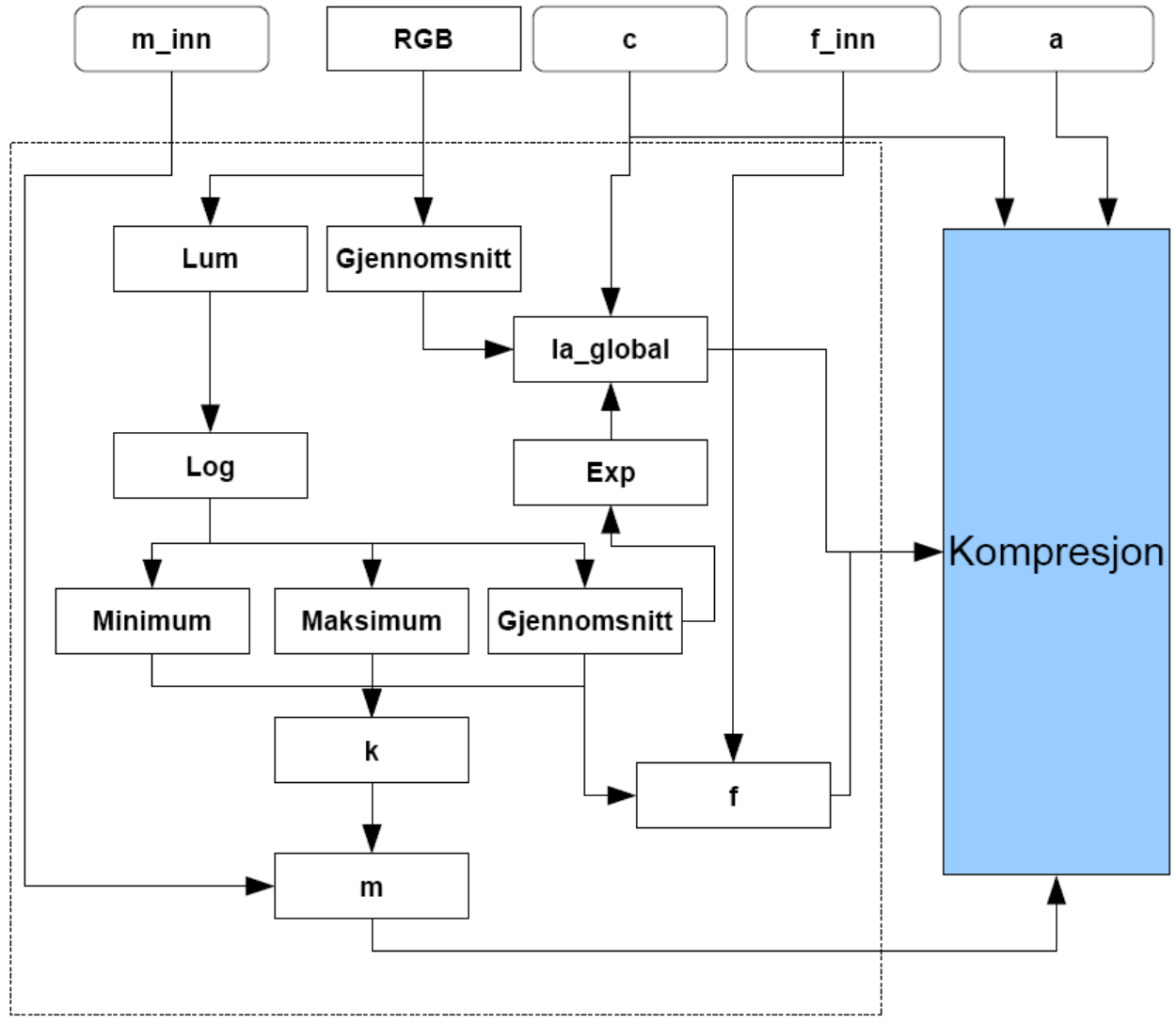
Tabell 4: Interne parametre i Reinhards Fotoreseptormodell

Parameter	***Beskrivelse***	***Verdi***
sigma	Bestemmer kompresjonskurven	$(2^{f} * I_a)^m$
f	Intensitet	$1.375*Llav + f\_inn$
m	kontrast	$0.3 + 0.7*k^{1.4} + m\_inn$
k	Logaritmisk forholdstall	$\frac{Llmax - Llav}{Llmax - Llmin}$
Llmax	Maksimal logaritmisk luminans	Log2(*maks*(luminans))
Llmin	Minimum logaritmisk luminans	Log2(*min*(luminans))
Llav	Logaritmisk gjennomsnittlig luminans	$\frac{1}{height * width} \sum_{i=1}^{height * width} Log2(Luminans_i)$

De interne kompresjonsparametrene er som vist i tabell 4 styrt av globale verdier i bildet og av inngangsparametrene. Figur 13 viser sammenhengen mellom disse parametrene samt sammenhengen mellom kompresjonsparametrene og inngangsparametrene. For en mer detaljert beskrivelse se Matlabkoden i vedlegg 2.

For å begrense minneforbruket samt å øke ytelsen blir disse kompresjonsparametrene estimert istedenfor å bli beregnet nøyaktig. Denne estimeringen skjer ved å bruke forrige rammes verdier til å beregne kompresjonsparametrene, fremfor å først lagre hele bilderammen for så å beregne kompresjonsparametrene. På denne måten vil utregningen av de nye parametrene og komprimeringen av bildet basert på forrige rammes parametre skje parallelt og dermed øke ytelsen til algoritmen. Ved å parallellisere på denne måten vil ytelsen til algoritmen gå på bekostning av

presisjon, siden de estimerte parametrene vil inneholde visse unøyaktigheter.



Figur 13: De operasjonene innenfor den stiplede firkanten er de operasjonene som er nødvendige for å beregne kompresjonsparametrene, og som blir estimert fra ramme til ramme.

3.1.4 Undermoduler

De forskjellige modulene vil bli implementert med forskjellig fokus ut fra hvor i arkitekturen de befinner seg. I signalflyten til inngangssignalet som skal komprimeres er det mest hensiktsmessig å prioritere hastighet, siden ytelsen i denne delen av designet blir avgjørende for ytelsen på hele kompresjonsmodulen. Ytelsen på utregningen av parametrene er ikke like kritisk, siden det ikke har noe stort utslag om det tar noen klokkepulser ekstra å regne ut disse. Optimalisering av arealbruken ved gjenbruk av ressurser vil derfor være et vektlagt tema i

implementasjonen av parameterberegningen, mens ytelse vil være fokus for selve signalkjeden i kompresjonsdelen.

3.1.4.1 Kompresjonen

I signalkjeden til selve kompresjonen er følgende funksjoner i bruk:

$$KomprimertPiksel_i = \frac{PikselRådata_i}{PikselRådata_i + \sigma_i}$$

$$\sigma_i = ((2^f) * Ia_i)^m$$

$$Ia_i = a * Ia_{lokal,i} + (1-a) * Ia_{global}$$

$$Ia_{lokal,i} = c * RGB_i + (1-c) * Luminans_i$$

For å begynne med den siste operasjonen i signalkjeden, består selve kompresjonen av en enkelt divisjon og en addisjon. Problemet med divisjon er at det er en av de mest tungvinte matematiske operasjonene å implementere effektivt i hardware så lenge ytelsen er i fokus. De to vanligste implementasjonsmetodene for divisjon er siffer-gjentakelse (digit recurrence) og konvergensmetoder **[Deschamp06]**. Siffer-gjentakelse er en metode som beregner ett siffer i svaret av gangen, og antall iterasjoner i denne algoritmen er da lik antall siffer (bit) som er ønsket i svaret. I høydynamisk bildebehandling er nettopp dette et problem siden det dynamiske området er meget stort, og antall bit blir da også meget høyt. Men akkurat i dette tilfellet så er svaret lik den komprimerte pikselverdien og dermed bare 8 bit. 8 klokkesykler er allikevel for lenge å vente i denne sammenhengen siden ytelsen er i fokus i denne signalkjeden. Konvergensmetoder er metoder som øker presisjonen for hver klokkesykel slik at svaret fra konvergensmetoden konvergerer mot det reelle svaret fra divisjonen. En slik metode bruker gjennomsnittlig halvparten så mange klokkesykler som siffer-gjentakelsesmetodene, men de er meget komplekse og trege siden de er avhengige av to multiplikatorer i serie for hver iterasjon. Denne metoden er derfor også lite egnet i signalstier med fokus på høy ytelse **[Deschamp06]**.

Det er også mulig å bruke logaritme til å forenkle divisjonen, siden

$$log2(\frac{a}{b}) = log2(a) - log2(b)$$

og

$$\frac{a}{b} = 2^{log2(\frac{a}{b})}$$

Som nevnt i delkapittelet om logaritme for bruk i DSP-applikasjoner er det forholdsvis ukomplisert å beregne 2'er logaritmen til et binært tall effektivt, med litt over 99 prosent nøyaktighet. Eksponentieringen med grunntall to er også ukomplisert å implementere, men med en litt mer begrenset nøyaktighet. Om denne unøyaktigheten tolereres vil dette være en meget rask måte å beregne en divisjon på, og en metode som er mye lettere å pipeline enn en iterativ algoritme. Alternativt kan en iterativ algoritme rulles ut før implementasjon, slik at hver iterasjon får en egen dedikert hardwareblokk som kan utføre beregninger parallelt. En slik løsning vil kunne ta forholdsvis mye areal, men vil kunne være enkel å pipeline.

Kompresjonsfaktoren σ kan også være problematisk å beregne, siden eksponentiering med variabel base er meget tungvindt. Men også her kan regnestykket forenkles betraktelig ved hjelp av logaritmeberegninger.

$$\sigma = ((2^f) * Ia)^m$$

kan skrives om til

$$\sigma = 2^x$$

hvor

$$x = log2(((2^f) * Ia)^m) = m * log2((2^f) * Ia) = m * (log2(2^f) + log2(Ia)) = m * (f + log2(Ia))$$

Ankepunktet her kan være multiplikasjonen med m, men denne multiplikasjonen er uansett mye mindre komplisert enn den opprinnelige eksponentieringen. En annen ulempe med denne metoden er tap i presisjonen. Men siden den komprimerte pikselverdien har 8-bits presisjon, og

inngangssignalene har 18-bits presisjon så er det litt å gå på i denne sammenhengen. Robustheten til algoritmen iforhold til dette tapet i presisjon vil bli sjekket iform av visuell inspeksjon av det endelige komprimerte resultatet.

De to siste leddene som beregner totalt adapsjonsnivå er meget enkle addisjoner og multiplikasjoner. Både *a* og *c* er definert til å ha 3-bits bredde, så multiplikasjon med disse konstantene kan forenkles til noen addisjoner og multipleksere.

3.1.4.2 Parametrene

I implementasjonen av parameterberegningene er ikke ytelsen like kritisk, men minimumskravet er at modulen som skal beregne parametrene holder tritt med bitstrømmen inn, samt at parametrene som skal brukes til kompresjonen er klare når kompresjonen starter. Men det er ingenting i veien med å definere at det må være x antall klokkepulser mellom hver bilderamme som skal komprimeres, for så å bruke disse klokkepulsene til å beregne noen av parametrene. Det som må beregnes kontinuerlig er gjennomsnitts, maksimum og minimumsverdiene i figur 13. Parametrene beregnes på følgende måte:

$$Ia_{global}(i) = c * RGB_{gjennomsnitt}(i) + (1-c) * LogaritmiskGjennomsnitt_{Luminans}$$

$$m = 0.3 + (0.7 * k^{1.4}) + m_{inn}$$

$$f = 1.375 * LogaritmiskGjennomsnitt_{Luminans} + f_{inn}$$

$$k = \frac{Log2(Luminans_{maks}) - LogaritmiskGjennomsnitt_{Luminans}}{Log2(Luminans_{maks}) - Log2(Luminans_{min})}$$

De parametrene som må beregnes kontinuerlig er maksimum, minimum og gjennomsnittlig luminans, samt gjennomsnittlig fargeverdi. Maksimum og minimum luminans er forholdsvis ukomplisert å beregne, da de midlertidige verdiene bare blir lagret i et register mens disse verdiene kontinuerlig blir sjekket opp mot de nye verdiene som genereres ut fra bitstrømmen. Å beregne gjennomsnittsverdi er derimot litt mer krevende. Gjennomsnittsverdien beregnes opprinnelig ved å summere alle verdiene, for så å dividere på antall verdier.

$$LogaritmiskGjennomsnitt_{Luminans} = \frac{1}{n}\sum_{i=1}^{n} Log2(Luminans_i)$$

En ulempe med denne fremgangsmåten er at mellomregningene potensielt kan bli meget store verdier, og man trenger derfor både store registre og store aritmetiske enheter for å beregne gjennomsnittet på denne måten. Alternativt kan summen deles opp i flere delsummer slik at

$$LogaritmiskGjennomsnitt_{Luminans} = \frac{1}{n} * \sum_{i=1}^{n} Luminans_i = \frac{1}{c} * \sum_{j=0}^{c-1} \sum_{i=1}^{b} \frac{Luminans_{(i+j*b)}}{b} \quad , \quad n = c * b$$

Problemet her kan være å finne det optimale forholdet mellom *c* og *b,* som vil gi det mest effektive forholdet med tanke på registerstørrelse og bitbredden inn på de forskjellige aritmetiske enhetene. Det er i tillegg en fordel om *b* er en orden av 2

$$b = 2^p \quad , \quad p \in heltall$$

slik at det å dele på *b* er en ren skiftoperasjon. Med denne oppdelingen så blir det kun en problematisk operasjon i gjennomsnittsberegningen, hvilket er deling med *c* (så sant ikke *c* også er en orden av 2). Divideringen av *c* kan deretter utføres etter komprimeringen av hele bilderammen ved å låne divisjonselementet i kompresjonsdelen. Alternativt kan man multiplisere med konstanten (1/c).

Beregningen av parameteren *k* består også av en vanskelig divisjon. For å beregne denne divisjonen er det mest hensiktsmessig å låne divisjonsenheten til kompresjonsmodulen. Subtraheringene er forholdsvis trivielle, og vil ikke by på noen problemer.

Parameteren *m* derimot byr på litt flere problemer. I dette tilfellet vil det kunne hjelpe å forenkle regnestykket med logaritme på lik linje med hvordan parameteren *σ* beregnes under kompresjonen. Hardwaremessig vil det være mest lønnsomt om utregningen av *m* og *σ* deler på en enhet som gjør denne utregningen. Siden σ beregnes kontinuerlig under kompresjonen og m skal beregnes mellom bildene vil dette ikke by på problemer.

Det er viktig å merke seg her at den enheten som beregner σ i kompresjonsmodulen benytter seg av funksjonsapproksimasjon, og har derfor en begrenset presisjon.

3.2 Beskrivelse av hardwareimplementasjonen på modulnivå

I dette delkapittelet blir selve implementasjonen av undermodulene som algoritmen er delt inn i presentert. De aritmetiske modulene er implementasjon av enkle matematiske operatorer, og er de modulene som blir presentert først. De logiske modulene større-enn og mindre-enn er også i bruk i kompresjonsalgoritmen, men istedenfor å implementere disse logiske operatorene i egne moduler brukes de innebygde større-enn og mindre-enn funksjonene i VHDL. Til slutt presenteres de sammensatte modulene som representerer blokkene i figur 10 og figur 11. Disse modulene inneholder noe egen funksjonalitet, samt en eller flere aritmetiske moduler. De av modulene som representerer operasjoner som går over flere klokkepulser er pipelinet, og har derfor klokkeinngang.

3.2.1 Aritmetiske moduler

Her presenteres de aritmetiske modulene som representerer de matematiske funksjonene som er i bruk i dynamikkompresjonsalgoritmen.

3.2.1.1 Adderer / Subtraherer

For å få en oversikt over hvilke adderere og subtraherere som er tilgjengelige, er de forskjellige adderer-kretsene og subtraherer-kretsene i **[Deschamps06]** som passer til formatet i algoritmen implementert i VHDL. Det er her tatt utgangspunkt i VHDL-koden presentert i **[Deschamps06]** og gjort noen små justeringer ut fra dette for å gjøre modulene generiske iforhold til bitbredde. I tillegg til disse addererene og subtrahererene er det også implementert en enkel rippel-carry adderer som er med i sammenligningen, samt en adderer som er hentet fra **[Maryland]**.

Det er som tidligere nevnt bare de rene kombinatoriske modulene som behandler to inngangsverdier som er tatt med i denne sammenligningen. Moduler som er sekvensielle eller som er beregnet på tre eller flere verdier er utelatt i denne sammenligningen. Synteseresultatene kan lese ut fra tabell 5, hvor alle modulene er syntetisert med 18-bits bitbredde i Leonardo Spectrum med studentbiblioteket til NTNU.

Tabell 5: Synteseresultater fra forskjellige adderer- og subtrahererkretser.

Modulbeskrivelse	***Kilde***	***Logisk forsinkelse [nS]***	***Gate-count***
Enkel Rippel-Carry adderer	Egendesignet	9,50	403
Carry-Skip (1)	**[Deschamps06]**	9,04	520
Carry-Skip (2)	**[Deschamps06]**	10,09	410
Carry-Select (1)	**[Deschamps06]**	9,46	410
Carry-Select (2)	**[Deschamps06]**	33,10	374
Subtraktor	**[Deschamps06]**	8,71	427
Rippel-carry-loop	**[Maryland]**	8,43	316

Noen av modulene har generiske parametre som justerer den fysiske arkitekturen innad i modulene. Disse implementasjonsmetodene har derfor to felter i tabell 5 der den ene er optimalisert for ytelse og den andre er optimalisert for areal.

Fra tabell 5 kommer det tydelig frem at rippel-carry addereren fra **[Maryland]** er både den raskeste og den mest arealeffektive addereren i sammenligningen. Av den grunn er det denne addereren som blir brukt videre. Det var bare en subtraktor i **[Deschamps06]** som oppfylte kriteriene satt i begynnelsen av dette avsnittet, og siden den implementasjonen gir tilfredsstillende synteseresultater brukes denne subtraktoren videre.

3.2.1.2 Multiplikator

[Deschamps06] beskriver flere forskjellige multiplikatorer, og disse multiplikatorene er modifisert til 18-bits bitbredde på begge inngangene og syntetisert i Leonardo Spectrum med det samme biblioteket for å få et reellt sammenligningsgrunnlag. Synteseresultatene fra dette arbeidet er presentert i tabell 6. Hovedsakelig er vi her ute etter en multiplikator som skal multiplisere to tall på kortest mulig tid. Første prioritet er derfor at den skal utføre multiplikasjonen på en klokkesykel, og andre prioritet er den logiske forsinkelsen innenfor denne klokkesykelen. Arealbruken er i dette tilfellet nedprioritert for å få en rask nok krets. Carry_Save_mult er dermed den multiplikatoren som vil bli brukt videre i designet.

Tabell 6: Synteseresultater fra forskjellige multiplikatorkretser.

Modulbeskrivelse	***Logisk forsinkelse [nS]***	***Gate-count***	***# clk***
Booth_1	45,72	6944	1
Booth_2	83,76	5635	1
Booth_3	64,74	6805	1
Base2_mult	36,15	4460	1
Ripple_Carry_mult	46,29	6880	1
Carry_Save_mult	24,45	6108	1
signed_mult_seq	10,23	1694	18

3.2.1.3 Dividerer

Divisjon er tradisjonelt vanskeligere å utføre raskt enn multiplikasjon i hardware. I denne oppgaven er det ønskelig med en divisjonskrets som ikke er iterativ, siden iterative kretser demmer opp signalflyten og ikke kan pipelines. Samtidig så er parametrene som skal divideres med hverandre av en slik karakter at divisjonskretsen må behandle fraksjonale tall, siden dividenden er større enn divisoren. Som et siste kriterie så må det heller ikke være noen begrensinger på inngangsverdiene på dividenden og divisoren utover det at dividenden skal være større enn divisoren. **[Deschamps06]** beskriver flere forslag til kombinatoriske divisjonskretser, men det er kun to av disse kretsene som oppfyller de nevnte kriteriene. Disse to kretsene er modifisert slik at begge har 18 bits bitbredde inn og 8 bits bitbredde ut, siden dette er de bitbreddene som er ønskelige i kompresjonsalgoritmen. Synteseresultatene fra de to kretsene er listet opp i tabell 7.

Table 7: Synteseresultater fra forskjellige divisjonskretser.

Modulbeskrivelse	***Logisk forsinkelse [nS]***	***Gate-count***
Serial_divider	68,41	2171
NonRestoring_divider	73,3	2425

Begge de to aktuelle kretsene er vurdert til å være for trege, så den kretsen med de beste synteseresultatene, serial_divider, er brukt som utgangspunkt for å lage en pipelinet divisjonskrets. Arkitekturen i divisjonskretsen er beholdt, men det er innført ett pipelinetrinn midt i signalflyten. Tabell 8 gjengir synteseresultatene til den pipelinede divisjonskretsen. Om den logiske forsinkelsen

fortsatt blir vurdert til å være for stor, så er det mulig å innføre to pipelinetrinn til i divisjonskretsen.

Table 8: Synteseresultatene til den pipelinede divisjonskretsen.

Modulbeskrivelse	***Logisk forsinkelse [nS]***	***Gate-count***	***Pipelinetrinn***
Serial_divider_pipelinet	34,85	2454	1

3.2.1.4 Logaritme

Kapittel 2.2 tar for seg logaritmeberegning i hardware, samt de forskjellige implementasjonsmetodene for å utføre denne beregningen. Flere av disse løsningene er implementert i hardware i denne masteroppgaven, og resultatene fra disse implementasjonene presenteres her.

Ut fra kriteriene om at presisjon kan gå på bekostning av areal og ytelse, så er lineær approksimasjon den mest relevante implementasjonsmetodikken for logaritme. Det er vanlig å sette disse kriteriene i audio- og bildeapplikasjoner, så den lineære approksimasjonsmetoden er derfor den mest utbredte metoden innenfor DSP-applikasjoner **[Abed03]**. Alle metodene som er beskrevet under lineær approksimasjon er derfor implementert i VHDL. Det er også laget en VHDL-testbenk som automatisk logger inngangs- og utgangsverdier fra logaritme-enhetene. Denne logen leses så inn i et Matlabscript som beregner feilprosent iforhold til en ideell logaritmefunksjon. Presisjons- og synteseresultatene til disse implementasjonene er gjengitt i tabell 9, og VHDL og Matlabkode for beregning av feilprosent ligger i filvedlegg. Alle disse resultatene er generert med 18-bits inngangssignal med heltallsverdier fra 1 til 262144.

Mens det er forholdsvis små variasjoner i areal (gate count) så er det desto større forskjeller i presisjon og hastighet. Hvilken logaritmeenhet man velger å implementere blir derfor først og fremst en avveining mellom presisjon og hastighet. Alle resultatene i tabell 9 kommer fra Leonardo Spectrum etter å ha syntetisert egen VHDL-kode.

Ut fra resultatene i tabell 9 er det valgt å bruke **[Abed03]** med en 3-regions feilkorreksjonskrets. Denne kretsen er valgt på bakgrunn av at den har forholdsvis lav feilprosent og lav logisk forsinkelse i forhold til gate-count.

Tabell 9: Resultater innen innført feil og synteseresultater av de aktuelle logaritmeimplementasjonene. Tallene i parentes er indeksen hvor ekstremalverdien i feilen oppstår.

	Mitchell 62	***SanGregory 99***	***Abed 00***	***Abed 03 SanGregory***	***Abed 03 2-region***	***Abed 03 3-region***	***Abed 03 6-region***
# Regioner	0	2	0	2	2	3	6
# korreksjonsbit	0	4	0	4	3	4	6
Max positiv prosentfeil	5,36	0,431398 (9)	5,360537 (3)	0,431398 (9)	0,929876 (7)	0,431398 (3)	0,152886 (43)
Max negativ prosentfeil	0	-1,540257 (3)	0	-1,540257 (3)	-0,554429 (3)	-0,268367 (55)	-0,153816 (17)
Feil % område	5,36	1,971655	5,360537	1,971655	1,484305	0,699765	0,306702
Delay [ns]	-	18,01	11,06	14,66	14,72	16,03	18,73
Frekvens [MHz]	-	55,5	90,4	68,21	67,93	62,38	53,39
Gate count	-	1567	1362	1448	1433	1454	1606

3.2.1.5 Eksponentiering/Antilogaritme

[Abed00] presenterer en løsning på antilogaritme med base 2. I likhet med logaritmeløsningen presentert i den samme publikasjonen (Se kapittel 2.2.3), er denne antilogaritmen en lineær tilnærming uten feilkorreksjon. Antilogaritmekretsen fra **[Abed00]** er tatt i bruk uten store forandringer, bortsett fra at bitbredden er forandret fra 16 bit til 18 bit.

3.2.2 Sammensatte moduler

De fleste blokkene i figur 12 og figur 13 er blokker som beregner parametre, og er derfor sammensatt av de aritmetiske modulene beskrevet i forrige delkapittel. Dette delkapittelet beskriver de sammensatt blokkene og hvordan de er bygd opp.

Modulene som står for selve kompresjonene har ytelse som høyest prioritet, på grunn av at signalveien fra inngang til utgang gjennom kompresjonsalgoritmen bestemmer ytelsen til kretsen. Modulene som beregner parametrene har areal som høyeste prioritet, og bruker derfor ofte ressurser fra kompresjonsdelen når disse ikke er i bruk, selv om det kan medføre litt venting på ressurser.

3.2.2.1 Luminans

Luminansen er et mål for lysstyrken i hvert enkelt piksel, og er en lineær sammenheng mellom fargene i det pikselet. Kompresjonsalgoritmen som er definert i **[Reinhard05]** bruker opprinnelig en luminanssammenheng som er:

$$Luminans = 0.2125 * Rød + 0.7154 * Grønn + 0.0721 * Blå$$

Hardwaremodulen som beregner luminansen bruker en shift-add fremgangsmåte, og tilnærmer luminansformelen ved å beregne:

$$Luminans = (2^{-3} + 2^{-4} + 2^{-6} + 2^{-7}) * Rød + (2^{-1} + 2^{-3} + 2^{-4} + 2^{-5}) * Grønn + (2^{-4} + 2^{-7} + 2^{-9}) * Blå$$

Hvilket resulterer i formelen

$$Luminans = 0.2109 * Rød + 0.7188 * Grønn + 0.0723 * Blå$$

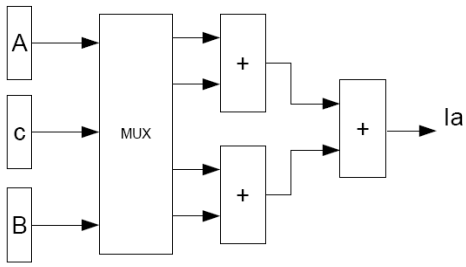
som gir en tilnærming til den originale luminansformelen med en feilmargin på 0.2 prosent. Det er uansett ingen universiell formel for Luminans, og forskjellige applikasjoner bruker forskjellige sammenhenger mellom fargene for å beregne luminansen.

Alle inngangsverdiene og utgangsverdien er 18-bits verdier, og alle addisjonene er implementert som 18-bits adderere. Det vil derfor være fare for enkelte avrundingsfeil i denne modulen, men forsøk viser at den totale feilen innført ved avrundingsfeil i addisjonene og estimeringen av parametrene tilsvarer mindre enn 0.5 prosent.

3.2.2.2 Ia

Både Ia_local, Ia_global og Ia_total beregnes på den samme måten. Formelen for disse beregningene kan generaliseres til:

$$Ia = c * A + (1 - c) * B$$



Figur 14: Flytskjema over modulen som beregner Ia.

Figur 14 viser hvordan modulen som beregner Ia er bygd opp. Parameteren c er en 3-bits parameter som bestemmer oppførselen til den 18-bits multiplekseren, som skifter A og B motsatt vei for å oppnå den ønskede effekten. Siden c er en 3-bits parameter, må multiplekseren ha fire utganger som til slutt adderes sammen for å oppnå riktig presisjon.

3.2.2.3 Sigma

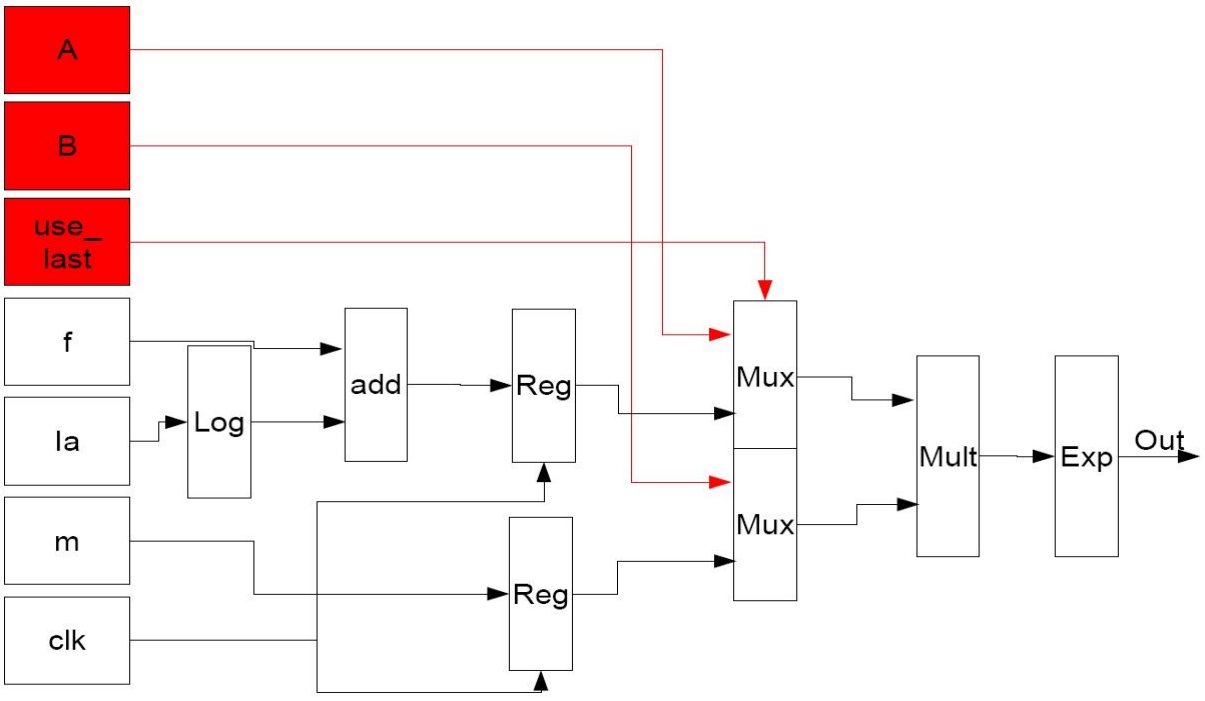
Kompresjonsparameteren sigma beregnes på følgende måte:

$$Sigma = \sigma_i = ((2^f) * Ia_i)^m$$

Til denne modulen er *f*, *Ia* og *m* inngangsparametre og blir beregnet tidligere i signalflyten. For å beregne sigma mest mulig effektivt er formelen skrevet om til:

$$\sigma_i = ((2^f) * Ia_i)^m = 2^{m*(f+log2(Ia_i))}$$

Ved å beregne sigma på den omskrevne formen kan sigma beregnes ved hjelp av de aritmetiske



Figur 15: Flytskjema over modulen som beregner kompresjonsparameteren sigma.

modulene beskrevet i forrige kapittel, og hvordan disse aritmetiske modulene er satt sammen for å danne sigma-modulen er illustrert i figur 15. De blokkene som er markert med rødt i figur 15 er implementert for å gjøre sigma-modulen mer fleksibel. Ved å sette use_last–signalet høyt vil sigmamodulen regne ut

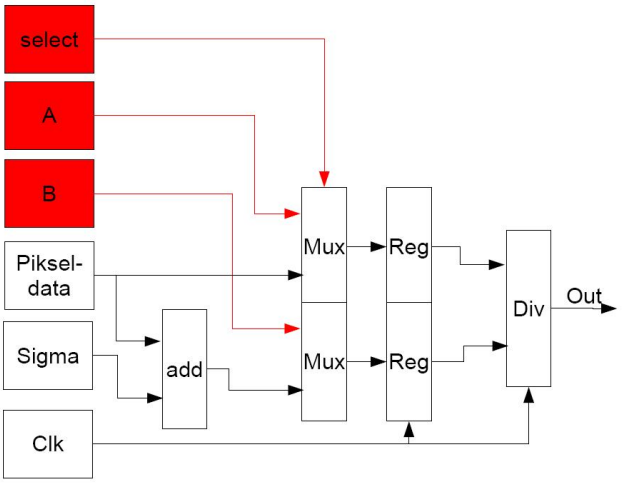
$$Out = 2^{(A*B)}$$

På denne måten kan disse ressursene deles slik at andre deler av systemet kan bruke multiplikatoren og eksponentieringsenheten når sigmamodulen ikke brukes til kompresjon.

3.2.2.4 Kompresjonen

Selve kompresjonen skjer ved at pikselverdien divideres med en faktor som er bestemt av kompresjonsparameteren sigma.

$$KomprimertPiksel_i = \frac{PikselRådata_i}{PikselRådata_i + \sigma_i}$$

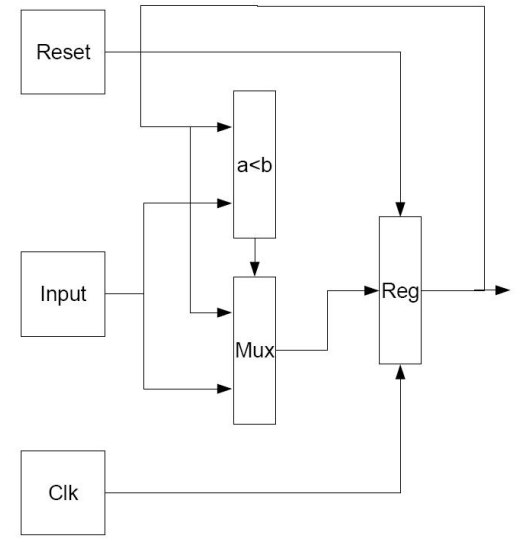


Figur 16: Flytskjema over kompresjonsmodulen.

Kompresjonsmodulen består som vist i figur 16 av en adderer og en divisjonskrets, hvorav disse aritmetiske modulene er beskrevet i forrige delkapittel. For å gi andre deler av designet tilgang til divisjonskretsen er det i denne modulen tre ekstra innganger, hvorav inngangen *select* velger hvilke innganger som skal ha tilgang til divisjonskretsen internt i kompresjonsmodulen.

3.2.2.5 Minimum og Maksimum

Modulene som beregner maksimum og minimum verdi i en bitstrøm er som vist i figur 17 klokkede moduler med synkron, aktiv høy reset. Ved



Figur 17: Flytskjema over modulene som beregner minimums- og maksimumsverdi.

reset så slettes registeret som bevarer verdien med den hittil største ekstremalverdien. Ved hver stigende klokkeflanke så sammenlignes inngangsverdien med den lagrede ekstremalverdien. Om inngangsverdien er en større ekstremalverdi enn den lagrede verdien så blir den lagrede verdien overskrevet med inngangsverdien.

3.2.2.6 Gjennomsnitt

Gjennomsnittet kan beregnes på følgende måte:

$$Gjennomsnitt = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{p}\sum_{j=1}^{p}\frac{1}{q}\sum_{k=1}^{q} x_{j,k}$$

$$Forutsatt,\ p*q = n$$

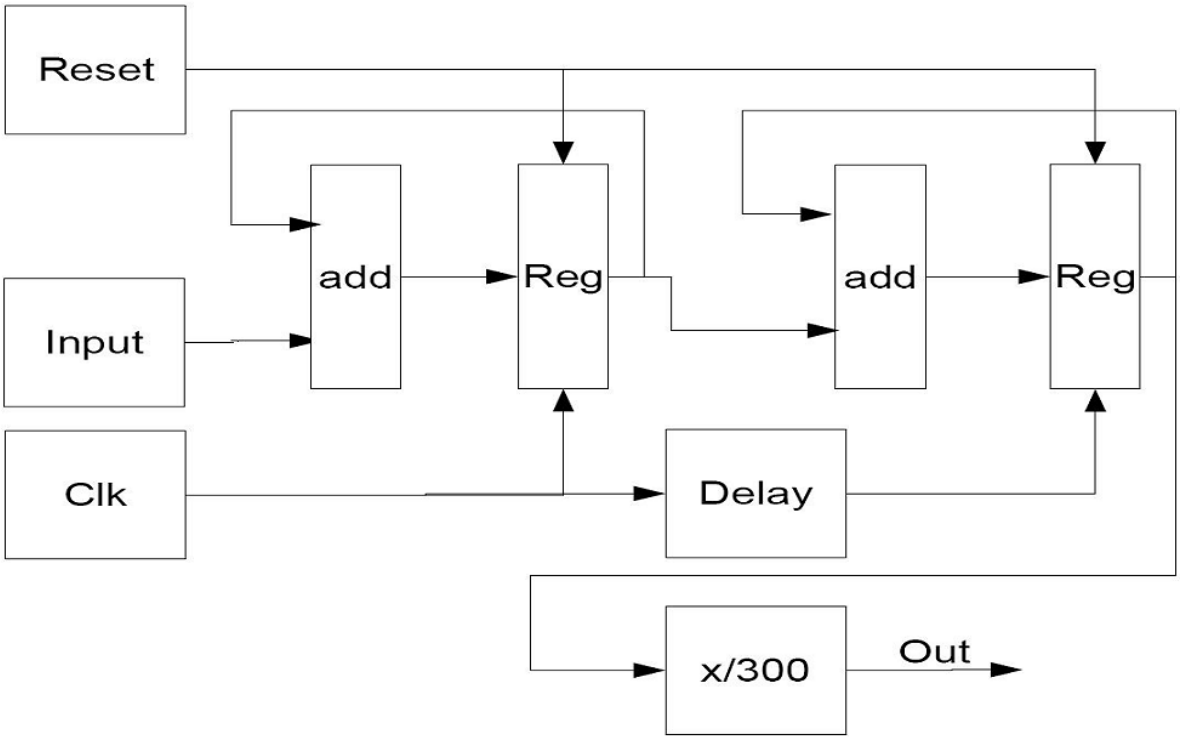
For å beregne gjennomsnittet har man valget mellom å gjøre dette på en mest mulig effektiv måte eller å gjøre det mest mulig fleksibelt. Ved å gjøre designet fleksibelt vil man kunne beregne gjennomsnittet av bitstrømmer med forskjellig antall sampler, altså med varierende *n*. Men siden divisjonene med *n* kan være krevende så er det mest effektivt å sette *n* fast, og så optimalisere designet iforhold til dette.

I denne implementasjonen er det valgt å la *q* være en multippel av 2, slik at denne divisjonen blir meget enkel. Som multippel er det valgt å bruke 10, det vil si at q blir lik 1024. Dette oppnås ved å skifte summen 10 bit til høyre mellom det første registeret og den andre addereren i figur 18. Delay-blokken som forsinker klokkesignalet er da en 10-bits teller slik at den andre addereren summerer summen av hver 1024-sampel blokk. Siden dette designet er beregnet på å komprimere VGA-bilder så blir parameteren *p* i formelen over lik 300. Divisjon med en konstant er mer komplekst enn multiplikasjoner med en konstant, og det er derfor valgt å multiplisere med konstanten (1/300). Denne konstanten er tilnærmet med uttrykket

$$\frac{1}{300} \approx \frac{1}{2^9} + \frac{1}{2^{10}} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{1}{2^{15}} = \frac{1}{300.6}$$

hvilket innfører en feil på 0.2 prosent på gjennomsnittet.

Denne effektiviseringen går som nevnt på bekostning av fleksibiliteten til designet. Om



Figur 18: Flytskjema over gjennomsnittsmodulen.

denne modulen skal brukes til å beregne gjennomsnittet av en bitstrøm med et annet antall sampler så må en eller begge parametrene *p* og *q* skifte verdi siden *n* også skifter verdi. I prinsippet kunne divisjonsenheten i kompresjonsdelen bli brukt til denne divisjonen, men dette var ikke mulig da den er konstruert som en fraksjonell divisjonskrets, det vil si at dividenden må være større enn divisoren. Dette kriteriet er ikke oppfylt i formelen til gjennomsnittet, og divisjonskretsen kan derfor ikke brukes her.

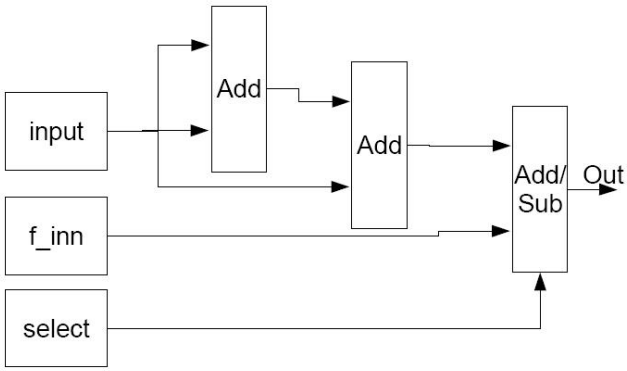
3.2.2.7 f

Parameteren *f* styrer lysheten i bildet, og blir beregnet ut fra bildets luminans samt en brukerdefinert parameter. Gjennomsnittsluminansen i bildet bestemmer initialverdien til *f*, mens brukeren kan inkrementere eller dekrementere denne verdien etter eget ønske. Initialverdien til *f* er gitt av

$$f = Llav * 1.375$$

hvor *Llav* er det logaritmiske gjennomsnittet til luminansen. Den endelige verdien på *f* etter at brukeren har justert er

$$f = f_{inn} + (Llav * 1.375)$$



Figur 19: Flytskjema over hvordan modulen som beregner parameteren f_inn er bygd opp.

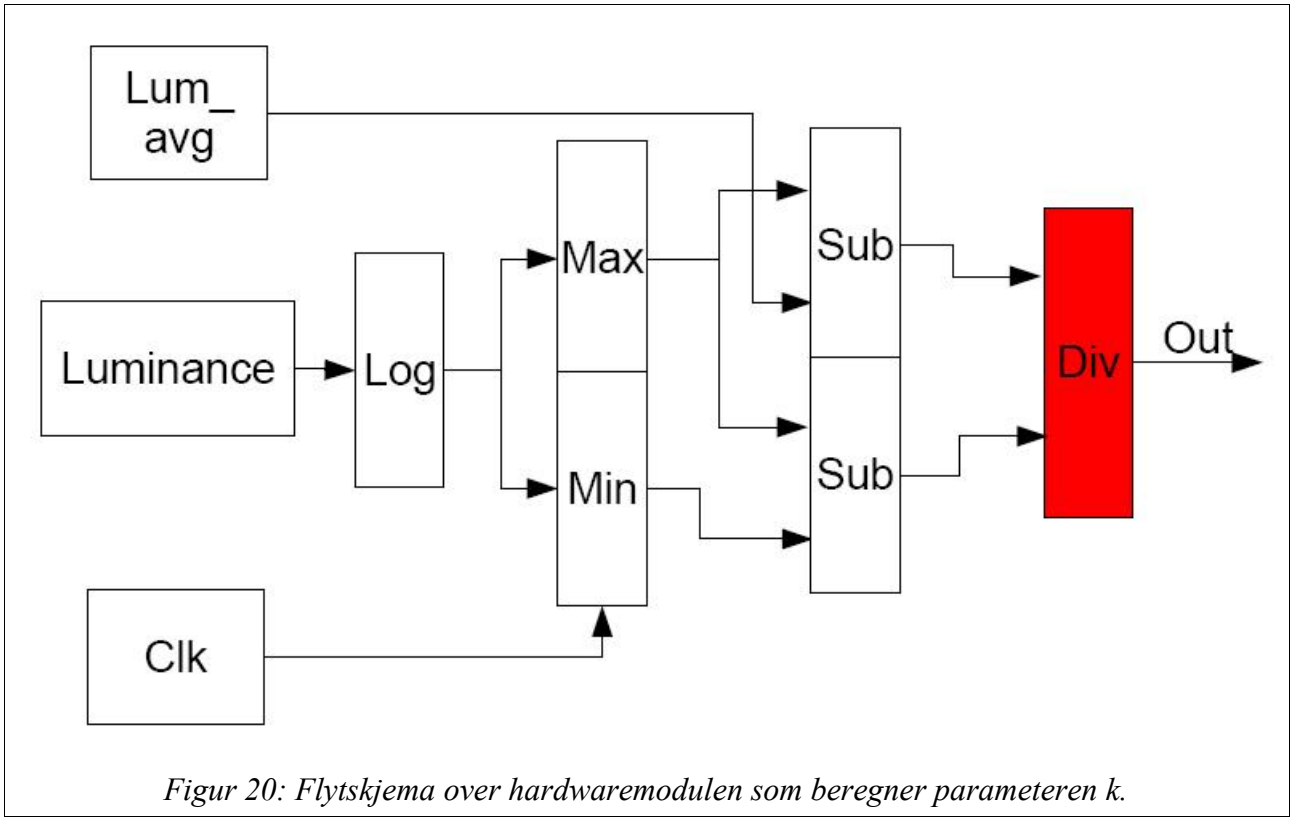
hvor *f_inn* kan være enten positiv eller negativ. Figur 19 viser hvordan selve modulen som beregner *f* er bygd opp. Inngangsverdien blir addert tre ganger med henholdsvis null, to og tre høyreskift, slik at resultatet blir 1.375 ganger inngangsverdien. *Select* bestemmer så om *f_inn* skal adderes eller subtraheres fra den initielle *f*-verdien. Select angir dermed fortegnsbitet til *f_inn*.

3.2.2.8 k

k er en parameter som beregnes ut fra de logaritmiske verdiene til minimum, maksimum og gjennomsnittlig luminans i bildet:

$$k = \frac{Log2(Luminans_{maks}) - LogaritmiskGjennomsnitt_{Luminans}}{Log2(Luminans_{maks}) - Log2(Luminans_{min})}$$

Figur 20 illustrerer hvordan hardwaremodulen som beregner parameteren *k* er bygd opp. Parameteren *k* blir regnet ut en gang for hver ramme, og har av den grunn ikke en dedikert hardware for alle operasjonene. Den røde blokkene i figur 20 er divisjonskretsen som er lånt fra kompresjonsmodulen. Siden *k* blir regnet ut mellom bilderammene vil ikke denne ressursdelingen by på noen problemer.



Figur 20: Flytskjema over hardwaremodulen som beregner parameteren k.

Luminance, Lum_avg og *clk* til venstre i figur 20 illustrerer inngangssignalene til modulen.

3.2.2.9 m

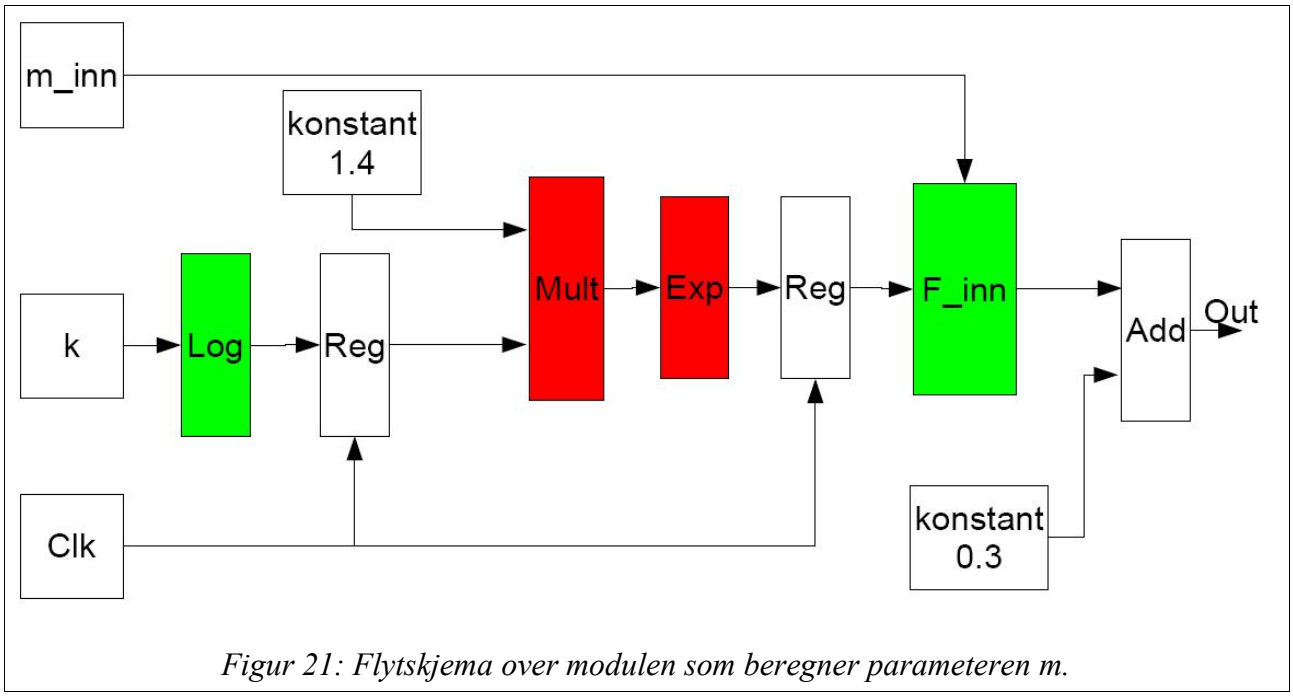
Parameteren *m* blir i likhet med parameteren *k* beregnet mellom bilderammene, og modulen som beregner *m* har derfor ikke all hardware dedikert men låner isteden enkelte deler av andre moduler. *m* beregnes på følgende måte:

$$m = 0.3 + (0.7 * k^{1.4}) + m_{inn}$$

For å beregne denne parameteren ved hjelp av de aritmetiske modulene beskrevet i forrige delkapittel er formelen skrevet om til:

$$m = 0.3 + (0.7 * k^{1.4}) + m_{inn} = 0.3 + 0.7 * 2^{(1.4 * log2(k))} + m_{inn}$$

Figur 21 illustrerer hvordan modulen som beregner *m* er bygd opp. De enhetene som er markert med rødt er lånt fra sigma-modulen, mens de enhetene som er markert med grønt er frittstående



Figur 21: Flytskjema over modulen som beregner parameteren m.

moduler på toppnivå som blir benyttet i denne beregningen. Modulen f_inn multipliserer inngangsverdien med en faktor på 1.375, og ved å høyreskifte utgangen en gang blir inngangen multiplisert med 0.6875.

Faktoren $2^{(1.4*log2(k))}$ blir i denne implementasjonen en tilnærming, siden både logaritmemodulen og eksponentieringsmodulen er lineære tilnærminger. Den endelige feilen som blir innført i dette leddet er beregnet i Matlab til å være maksimalt 2%. Mesteparten av den innførte feilen kommer av at den lineære approksimasjonen i eksponentieringsenheten ikke har feilkorreksjon.

Dette delkapittelet har tatt for seg hvordan undermodulene er bygd opp, og hvilke løsninger som er valgt for alle de aritmetiske modulene. Neste delkapittel oppsummerer arbeidet ved å beskrive de endelige arkitekturene og avgjørelsene tatt underveis.

3.3 Oppsummering av arbeidet

Dette delkapittelet er en oppsummering av arbeidet som er gjort, og oppsummerer arkitekturbeskrivelsene samt beskrivelser avgjørelsene tatt underveis som ikke er nevnt andre steder.

3.3.1 Beskrivelse av de forskjellige arkitekturene

Siden dynamikkompresjonsalgoritmen benytter seg av separat fargebehandling hvor hver fargekanal er uavhengig av hverandre, åpner dette for at fargene kan behandles enten serielt eller parallelt. Det er derfor implementert to forskjellige arkitekturer som beskrevet i kapittel 3.1, hvor den ene arkitekturen behandler fargene parallelt mens den andre arkitekturen behandler fargene serielt. Figur 10 og figur 11 illustrerer de to arkitekturene og forskjellen mellom disse.

Begge arkitekturene er modulbasert, slik at signalkompresjonen og parameterberegningen benytter de samme modulene i begge arkitekturene. Det er bare kontroll-modulen som er forskjellig, siden det er denne modulen som styrer signalflyten og parameterberegningene. På denne måten blir det enkelt å bytte mellom de to arkitekturene, og alle modulene kan gjenbrukes uten modifikasjoner bortsett fra kontrollmodulene.

3.3.2 Tallformater

Totalt i algoritmen er det tre forskjellige tallformater som er tatt i bruk. Disse tre tallformatene er heltall, logaritmisk fastpunkttall og lite fastpunkttall.

Pikselverdiene både før og etter kompresjon er gitt som heltall. Rådataene som kommer inn er høydynamisk 18-bits heltall mens den ferdig komprimerte pikselverdien er et 8-bits heltall.

Alle mellomregninger som innebærer logaritmiske verdier er gitt i 18-bits logaritmisk fastpunkttall. Dette tallformatet er definert slik at de 5 første bit'ene er heltallsdelen mens de 13 siste bit'ene er desimaldelen. Det dynamiske området på dette tallformatet er altså $[0, 32\text{-}2^{13}]$, med en presisjon tilsvarende 2^{-13}.

Enkelte parametre har operasjonsintervall mellom 0 og 1, som for eksempel parametrene m og k. Disse parametrene beregnes i lite fastpunkttall,noe som vil si at samtlige bit definerer desimal-talldelen. k som er 13 bit har da et dynamisk område mellom $[0, 1\text{-}2^{13}]$ og en presisjon på 2^{-13}.

Rådataene som blir komprimert er i denne oppgaven definert til å ha 18-bits dynamisk område. Deler av designet er gjort generisk, men siden dette ikke gjelder hele designet så er bitbredden inn i modulen låst til å være 18 bit. Det er de logaritmiske enhetene (logaritme og antilogaritme) som ikke er gjort generiske.

3.3.3 Oppløsningen på bildet

I et reellt design burde oppløsningen på bildet som skal komprimeres være en generisk parameter som kan settes i designet. I denne oppgaven er det bestemt at bildet er et VGA-bilde med 640x480 oppløsning. Dette er gjort for å forenkle oppgaven, og for å beholde fokuset på selve hardwareimplementasjonen av dynamikkompresjonsmodulen.

3.3.4 Koding av moduler

Samtlige undermoduler bortsett fra modulene som beregner parametrene m og k er kodet i egne VHDL-filer. Deretter er filhierarkiet bygd opp på samme måte som arkitekturen i modulen, hvor undermodulene blir tatt inn som komponenter i nivået over. Hvordan arkitekturen er bygd opp er beskrevet i kapittel 3.1. Vedlegg 9 viser en oversikt over alle VHDL-filer som er lagt ved oppgaven, samt en liten beskrivelse av hver fil.

3.3.5 Timing

Siden dynamikkompresjonen er avhengig av estimerte verdier vil den første bilderammen som sendes inn i modulen kun bli brukt som basis for parameterestimering. Det er altså først bilderamme nummer to som komprimeres med reelle parameterverdier, og som dermed vil ha tilfredsstillende visuelle resultater. Figur 22 illustrerer timingen av prosesser og når utgangen er gyldig. Illustrasjonen viser tre bilderammer som kommer inn til modulen hvorav den første kun brukes til å beregne kompresjonsparametre, mens de to siste både blir komprimert og brukes som grunnlag for nye parametre. De parametrene som beregnes kontinuerlig er maksimum luminans, minimum luminans, gjennomsnittlig luminans samt gjennomsnittet av de tre fargekanalene.

Inngang	Ramme1	Idle	Ramme2	Idle	Ramme3
Kontinuerlig parameter-beregning	Estimerer for Ramme2		Estimerer for Ramme3		Estimerer for Ramme4
Dedikert parameter-beregning		X		X	
Kompresjon	Ikke gyldig utgang		Ramme2		Ramme3
Sum sykler	*Antall piksler (x3)*	*8*	*Antall piksler (x3)*	*8*	*Antall piksler (x3)*

Figur 22: Illustrasjon av timingen til modulen som viser når de forskjellige interne prosessene er aktive.

Det er definert i designet at parameter-beregningene trenger 8 klokkepulser mellom hver bilderamme for å beregne ferdig parametrene. Disse klokkesyklene brukes til å fordele tidsmultipleksede ressurser til de modulene som beregner ferdig kompresjonsparametrene.

Antall klokkesykler som brukes per ramme er avhengig av hvilken arkitektur som er i bruk. Arkitekturen med parallell fargebehandling komprimerer ett piksel per klokkesykel og bruker dermed like mange sykler som antall piksler som skal komprimeres. Den serielle tidsmultipleksede arkitekturen komprimerer derimot en farge per klokkesykel, og siden det er tre farger per piksel så bruker denne arkitekturen tre ganger så mange klokkesykler som den parallelle arkitekturen.

3.3.6 Testbenker

Hvert nivå i designet har egne testbenker. Det er laget testbenker for de aritmetiske modulene, for de sammensatte modulene og for begge de to ferdige arkitekturene.

Testbenkene for undermodulene tester flere scenarioer, men forutsetter manuell kontroll for å avdekke feil. De aritmetiske modulene har testbenker som automatisk sjekker svaret opp mot den ideelle svarverdien. Flere av modulene representerer funksjonstilnærminger istedenfor eksakte funksjonsberegninger, og i disse tilfellene beregnes feilprosenten til estimeringen kontinuerlig underveis i testen.

Testbenkene for de to ferdige arkitekturene er bygd opp likt, men med forskjellig timing. Først henter de inn pikselverdier fra to tekstfiler, for så å mate disse verdiene inn i dynamikkompresjonsmodulen. Grunnen til at det hentes verdier fra to tekstfiler er at det skal være mulighet til å bruke to bilder med små forskjeller som skal kunne illustrere to påfølgende bilderammer i en filmsekvens. På denne måten kan man sjekke om estimeringen av parametrene fra en ramme til den neste gir tilfredsstillende resultater. De fire inngangsparametrene til dynamikkompresjonsalgoritmen er gitt som signaler i tekstbenken som kan justeres etter ønske. Mens bildet komprimeres blir resultatet logget i en egen tekstfil. Denne tekstfilen leses så inn i

Matlab som konverterer tekstfilen med pikselverdiene til et bitmap-bilde, og viser frem dette bildet som resultat.

Det er lagt mest arbeid i systemtestbenkene til toppnivået til modulene som sjekker det endelige resultatet samt samspillet mellom Matlab og Modelsim. Matlab brukes først til å finne de ønskede parameterverdiene og det ønskede visuelle resultatet. Deretter justeres parameterverdiene i VHDL-testbenken til å samsvare med parameterinnstillingene i Matlab og fullfører simuleringen med disse innstillingene. Deretter sjekkes resultatet fra testbenken i Matlab opp mot Matlab sine egne resultater visuelt.

Alle testbenkene er lagt ved oppgaven som filvedlegg.

3.3.7 Matlab

Gjennom denne masteroppgaven er Matlab blitt brukt til å generere pikseldata til VHDL-testbenkene, lese inn ferdig komprimert pikseldata fra VHDL-testbenkene og til å generere bilder fra disse dataene. I tillegg har Matlab blitt brukt til å generere testbilder for å sammenligne resultatene fra VHDL-testbenkene og for å finne verdier til dynamikkompresjonsmodulens parametre. Tabell 10 viser en oversikt over Matlabfilene som er lagt ved oppgaven som filvedlegg og hvilke oppgaver de funksjonene utfører. I tillegg til disse filene har Matlab blitt brukt aktivt til å beregne feilprosent til forskjellige moduler samt å behandle resultatdataene fra testbenkene til logaritmemodulene, men det er ikke laget script til alle operasjonene. Alle Matlabfilene er skrevet selv i forbindelse med denne masteroppgaven bortsett fra read_rle_rgbe.

Tabell 10: Oversikt over Matlabfilene.

`read_rle_rgbe`	Leser inn bilder i hdr-format i Matlab. Skrevet av Lawrence Taplin
`cut_picture`	Leser inn et hdr-bilde og lager en piksel-fil til VHDL-testbenken. Størrelsen på bildet som lagres i tekstfilen angis som parametre til funksjonen.
`PhPh_array`	Tar inn piksel-arrayet generert fra cut_picture og komprimerer dynamikken etter Reinhards Fotoreseptormodell og lagrer resultatet som en bmp-bildefil. For å teste ut forskjellige parameterinnstillinger.
`read_vhdl_output`	Leser inn resultatfila fra testbenken til den store arkitekturen og lagrer resultatet som en bmp-bildefil.
`read_vhdl_output_small`	Leser inn resultatfila fra testbenken til den lille arkitekturen og lagrer resultatet som en bmp-bildefil.
`find_error`	Tar inn resultatfila til en logaritme-testbenk og sammenligner de estimerte verdiene med ideelle verdier for å kartlegge estimeringsfeilen.

3.3.8 Designmetodikk

I planleggingsfasen av implementasjonen er det benyttet en top-down metodikk. Her ble algoritmen analysert og delt inn i stadig mindre deler, helt til det ble klart hvilke matematiske operasjoner som trengtes samt kravene til disse. Denne inndelingen ble så konkretisert iform av arkitekturbeskrivelsene i kapittel 3.1.

Utover i designfasen derimot er det benyttet en bottom-up metodikk hvor det er tatt utgangspunkt i de allerede bestemte arkitekturene. Her ble de matematiske operasjonene implementert først og testet hver for seg før de ble satt sammen til de sammensatte modulene som beskrevet i kapittel 3.2.2. Til slutt ble disse sammensatte modulene brukt til å bygge opp arkitekturene som beskrevet i kapittel 3.1.

3.3.9 Betraktninger rundt det dynamiske området

Høydynamisk bildeteknologi er en forholdsvis ung teknologi, og man har enda ikke blitt enige om hvor stort det dynamiske området bør være og hva slags kompresjonsalgoritmer som skal brukes for å komprimere ned dynamikken når bildet skal vises på et vanlig lavdynamiskt medie. På bakgrunn av dette er det valgt et dynamisk område som representerer det intrascene dynamiske området til MVS som vist i tabell 2, siden det vil være logisk for et høydynamisk kamera å ha dette området som et minimum det kan representere. 18-bit passer bra overens med det intrascene dynamiske områpdet til MVS, og er derfor valgt som bitbredde på inngangssignalet i designet.

3.3.10 SW brukt i denne oppgaven

I denne oppgaven er det brukt forskjellige SW-verktøy i de forskjellige fasene. Matlab versjon 7.4.0.287 er brukt som beskrevet i kapittel 3.3.7. Modelsim SE versjon 6.2d er brukt til å skrive selve VHDL-filene, samt til å simulere alle testene. Til syntetiseringen er Leonardo Spectrum 2006a.59 tatt i bruk med NTNU sitt eget studentbibliotek for 0.5-ASIC. OpenOffice 2.1 er brukt til å skrive selve oppgaven samt til å generere figurene.

Dette kapittelet har handlet om arbeidet som har blitt gjort i designfasen og hvilke valg som er tatt underveis i dette arbeidet. Neste kapittel omhandler resultatene til masteroppgaven, og vil gå gjennom både de visuelle resultatene og de fysiske synteseresultatene.

4 Resultater

Dette kapittelet er en gjennomgang av resultatene som er generert i denne masteroppgaven. Først blir de visuelle resultatene gjennomgått før de fysiske resultatene blir presentert i form av synteseresultater. Målet med dette kapittelet er å danne et grunnlag for diskusjonen som kommer i neste kapittel.

4.1 Visuelt resultat

Siden dynamikkompresjonsmodulen skal stå i et kamerasystem er det visuelle resultatet meget viktig for å vurdere om konseptet i det helet tatt har en fremtid i det kommersielle markedet. Modulen som er designet i denne oppgaven er beregnet på VGA-bilder, så samtlige visuelle resultater har VGA-oppløsning (640x480). I tillegg er de visuelle resultatene sterkt avhengig av inngangsparametrene til algoritmen. Inngangsparametrene er oppført opp i tabell 3, og ut fra denne tabellen kan man se at parameter-rommet er meget stort.

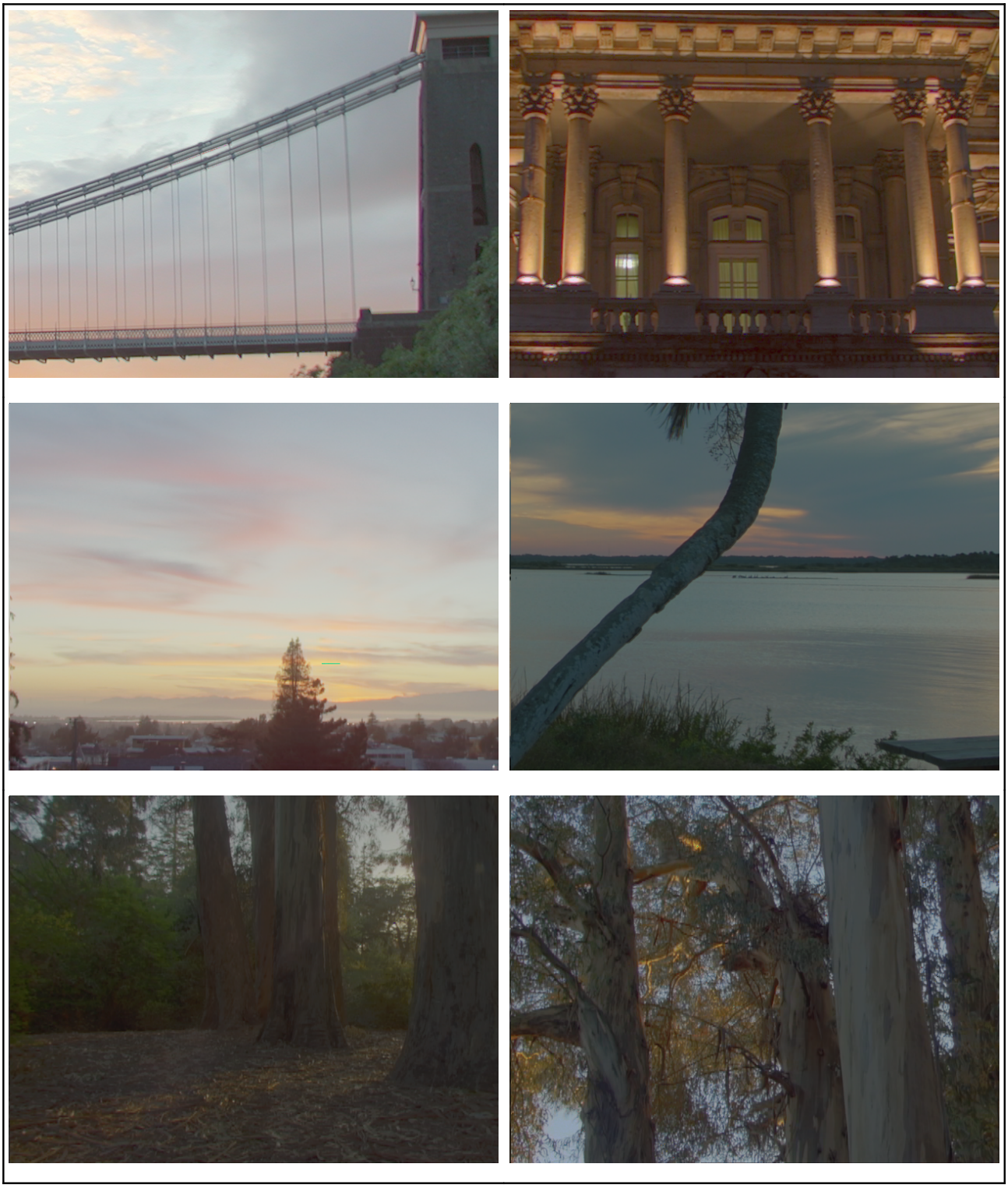
Parametrene f_inn, m_inn, a og c er henholdsvis 18, 13, 3 og 3 bit. De to første parametrene har i tillegg fortegnsbit, noe som teoretisk gir et totalt parameter-rom tilsvarende

$$2^{19} * 2^{14} * 2^{3} * 2^{3} \approx 550\, milliarder$$

forskjellige kombinasjoner av inngangsparametre. Det reelle antallet er i praksis en del mindre, siden verdier ut i grenseområdene på parameterne m_inn og f_inn vil kunne gjøre at kompresjonsparametrene f og m går utenfor sitt definerte område.

Figur 23 viser forskjellige resultater fra dynamikkompresjonsmodulen hvor det er scener med forskjellige egenskaper og karakteristikker som blir komprimert. Resultatene i figur 23 er generert med forskjellige parameterinnstillinger, og det er en subjektiv vurdering hvordan man skal justere parametrene for å få best mulig bilde. Hva man ser på som best mulig bilde kommer ann på hvilke kvaliteter man vil fremheve i bildet. Det må for eksempel forskjellige justeringer for å få frem stemning, detaljer eller dynamikk i et høydynamisk bilde.

Hardwaremodulen gir visuelt sett like resultater som dynamikkompresjonsalgoritmen i Matlab. For å sammenligne de visuelle resultatene fra denne algoritmen opp mot andre dynamikkompresjonsalgoritmer, se **[Hansen06]**.



Figur 23: En illustrasjon av flere scener med forskjellige egenskaper. Alle disse scenene er komprimert med hardwaremodulen via testbenk.

4.2 Fysiske resultater

De fysiske resultatene blir her målt i ytelse og i areal og er gjengitt i tabell 11.

Tabell 11: Synteseresultater for dynamikkompresjonsmodulen generert i Leonardo Spectrum.

Arkitektur	**Gate count**	**Piksler per sekund**	**Fps (VGA)**
Optimalisert for areal:			
Seriell fargebehandling (1	42427	9 M	29,3 fps
Parallell fargebehandling (1	72229	27,1 M	88,2 fps
Optimalisert for ytelse:			
Seriell fargebehandling (2	56774	10,3 M	33,5 fps
Parallell fargebehandling (2	75806	29,9 M	97,3 fps
Estimert resultat for 0.18 – prosess:			
Seriell fargebehandling (3	42427	~ 27 M	~ 90 fps
Parallell fargebehandling (3	72229	~ 81 M	~ 265 fps

Ved å sette enkelte betingelser i Leonardo Spectrum kan man velge mellom å syntetisere for optimal arealbruk eller for optimal ytelse. De linjene i tabell 11 som er merket med (1 er syntetisert for optimal arealbruk mens de linjene i tabell 11 som er merket med (2 er syntetisert for optimal ytelse.

Feltene i tabell 11 som er merket med (3 er et forsiktig estimat over hvilken ytelse som kan oppnåes med dynamikkompresjonsmodulen i en moderne 0.18 prosess, hvor estimatet baserer seg på at ytelsen øker med en faktor på ca tre. Studentbiblioteket som Leonardo Spectrum benytter seg av til syntesen baserer seg på en 0.5µ prosess, og gir av den grunn forholdsvis dårlige synteseresultater iforhold til dagens moderne teknologi. Hvor mye ytelsen vil øke ved at designet blir syntetisert med et moderne bibliotek er vanskelig å bedømme, og ytelsen vil uansett bli forskjellig om designet syntetiseres i en prosess beregnet på rene digital kretser eller med en prosess beregnet for blandet design. Ifølge roadmapen for CMOS-kretser presentert i **[Wanhammer99]** vil ytelsen til en 0.18-prosess være mellom 3 og 4 ganger større enn en 0.5 prosess, mens resultatene fra en 0.13 prosess vil være mellom 4 og 5 ganger raskere enn en 0.5 prosess.

4.3 Ytelsesresultater fra sammenlignbare løsninger

Som nevnt i kapittel 1.4 er alternativene til en hardwareimplementasjon å implementere algoritmen i en MCU, CPU eller en GPU.

Vedlegg 8 viser et grovt estimat over ytelsen til en MCU-implementasjon. Den reelle ytelsen vil bli lavere siden dette estimatet ikke tar hensyn til minnehåndtering, men estimatet kan allikevel sees på som en pekepinn på hvor konkurransedyktig en slik løsning er for en dynamikkompresjonsalgoritme. Den estimerte ytelsen er oppført i tabell 12. Arealbruken i en mikrokontroller ligger på ca 75K gates **[ARM_2]**, noe som gjør den sammenlignbar med den største arkitekturen som er designet i denne masteroppgaven. Men på lik linje med at minnebruken ikke er tatt med i ytelsesestimatet så er heller ikke minnet til MCU-koden med i denne arealbruken.

[Goodnight03] viser ytelsen til en lignende dynamikkompresjonsalgoritme implementert på CPU og på GPU. Kompresjonsalgoritmen som implementeres i **[Goodnight03]** er Reinhards Fotografiske Reproduksjonsmodell **[Reinhard02]**, og som vist i **[Hansen06]** er **[Reinhard02]** og **[Reinhard05]** sammenlignbare både når det gjelder visuelt resultat og når det gjelder ressursbruk. Av den grunn er resultatene fra **[Goodnight03]** tatt med i tabell 12 som viser ytelsen til forskjellige implementasjonsmetodikker.

Tabell 12: Oversikt over ytelsen til alternative implementasjonsmetodikker.

Metodikk	**Ytelse**	**Fps (VGA)**
GPU-implementasjon	65,5 Mpiksel / sek	213,2
CPU-implementasjon	1,7 Mpiksel / sek	5,5
MCU-implementasjon	0,6 Mpiksel / sek	1,9

Dette kapittelet har vist frem hvilke resultater som er oppnådd i denne masteroppgaven samt noen eksempler på sammenligningsgrunnlag. I neste kapittel vil disse resultatene bli gått gjennom og vurdert.

5 Diskusjon

I dette kapittelet vil resultatene som ble presentert i kapittel 4 bli vurdert, og sammenlignet med de sammenlignbare resultatene fra kapittel 4.3 i de tilfellene det er naturlig. Det er spesielt de visuelle- og de fysiske resultatene som skal vurderes, men også brukervennligheten, potensialet og allsidigheten til dynamikkompresjonsmodulen blir her vurdert.

5.1 Visuell vurdering

Generelt er det viktig å fremheve at en visuell vurdering er veldig subjektiv og at hva vi ser etter i et bilde er helt avgjørende for hvilket bilde vi syns er best når vi sammenligner flere bilder. **[Ledda05]** presenterer en studie av forskjellige dynamikkompresjonsalgoritme som konkluderer med at de dynamikkompresjonsalgoritmene som gir de visuelt beste resultatene ikke er de samme algoritmene som nødvendigvis reproduserer den originale scenen på den beste måten. Dette studiet understreker den subjektive betydningen av visuell kvalitet og hvor vanskelig denne er å måle.

Dynamikkompresjonsmodulen gir visuelt sett like resultater som Matlabscriptet fra **[Hansen06]**, mye på grunn av at det er dette scriptet som er lagt til grunn for hardwareimplementasjonen. I **[Hansen06]** blir Reinhards Fotoreseptoralgoritme valgt på bakgrunn av disse visuelle resultatene, og resultatene fra dette scriptet blir derfor brukt som sammenligningsgrunnlag i denne masteroppgaven. Ut fra dette sammenligningsgrunnlaget er de visuelle resultatene meget bra, siden estimeringen av matematiske funksjoner ikke gir visuelt utslag i de ferdig komprimerte bildene.

Som nevnt i **[Hansen06]** vil estimeringen av parametrene mellom bilderammene kunne gi utslag ved store forandringer mellom rammene. Denne begrensningen vil også gjelde for hardwareimplementasjonen, og denne modulen egner deg derfor ikke i kamerasystemer med veldig lav bilderate hvor de globale verdiene i bildet vil variere mye fra bilderamme til bilderamme. Men den vil derimot egne seg godt i kamerasystemer med ytelse som tilsvarer levende bilder, det vil si hvor rammeraten er 25 bilderammer eller mer i sekundet. Dette kriteriet gjelder forøvrig også for standardene i romlig kompresjons, som for eksempel MPEG-standarden. Et slikt kriterie blr derfor sett på som uproblematisk.

5.2 Vurdering av de fysiske resultatene

Tabell 13 viser en oppsummering av ytelsesresultatene til dynamikkompresjonsmodulen som er designet i denne masteroppgaven, samt alternative løsninger. Tabellen viser også kravene til de mest aktuelle formatene for levende bilder, for å få et visst innblikk hvilke applikasjoner de forskjellige løsningene kan brukes i.

Tabell 13: Oversikt over ytelsen til de forskjellige implementasjonsmetodikkene. Standardene som står i kursiv er tatt med i tabellen for sammenligning.

Metodikk / Standard		**Ytelse**	
ASIC - parallell fargebehandling	(0.18 prosess)	Estimert	81,0 Mpiksel / sek
GPU-implementasjon			65,5 Mpiksel / sek
HDTV			*55,3 Mpiksel / sek*
ASIC - parallell fargebehandling	(0.5 prosess)		29,9 Mpiksel / sek
ASIC - seriell fargebehandling	(0.18 prosess)	Estimert	27,0 Mpiksel / sek
PAL			*11,1 Mpiksel / sek*
ASIC - seriell fargebehandling	(0.5 prosess)		10,3 Mpiksel / sek
NTSC			*7,6 Mpiksel / sek*
CPU-implementasjon			1,7 Mpiksel / sek
MCU-implementasjon			0,6 Mpiksel / sek

Ut fra Tabell 13 ser man at det kun er GPU-løsningen som er konkurransedyktig med ASIC-løsningen når det gjelder ytelse. Men med tanke på at GPU-løsningen er implementert på et dedikert grafikkort i en PC, så er den definitivt ikke konkurransedyktig med tanke på ressurs- og arealbruk. På grunn av de store forskjellene i ressursbruk vil ikke disse to løsningene uansett være aktuelle for de samme applikasjonene, men det er uansett greit å se at ytelsen til denne ASIC-implementasjonen er i samme størrelsesorden som ytelsen til en så tung implementasjon som en GPU-implementasjon er. Sett på bakgrunn av denne sammenligningen, vurderes ytelsen på denne ASIC-implementasjonen til å være meget bra.

De fysiske resultatene til de to arkitekturene er gjengitt i tabell 11. Hvilken av disse arkitekturene som er å foretrekke er avhengig av hvilken applikasjon de skal brukes til og hvilke ytelseskrav den applikasjonen har. Det er generelt sett ønskelig å minimere gate count i ASIC-implementasjoner, så om ytelsen til den serielle tidsmultipleksede arkitekturen møter ytelseskravene til den aktuelle applikasjonen, så er denne arkitekturene å anbefale siden den har den laveste gate

counten. Siden arkitekturen med parallell fargebehandling oppnår resultater langt høyere enn HDTV (estimert med 0.18 prosess) så vil ytelsen på denne arkitekturen nå kravet til de fleste aktuelle applikasjoner per dags dato.

Som nevnt i kapittel 1.3 er denne ASIC-implementasjonen beregnet for et kamerasystem, og sett i sammenheng med dette bruksområdet burde dynamikkompresjonsmodulen nå de ytelsesmål som kameraprodusentene selv setter for høydynamiske produkter. Digitalkameraer for konsumermarkedet vil sannsynligvis bli målgruppen på sikt, men foreløpig er bruksområdene for høydynamiske kameraer er begrenset til overvåkning og sikkerhet **[Micron07]**. Sett iforhold til denne bruken vil PAL/NTSC og HDTV være en god målestokk for hvilken ytelse kompresjonsmodulen burde nå. Fra tabell 13 ser man begge arkitekturene når kravet til PAL og NTSC forholdsvis enkelt, mens den parallelle arkitekturen når kravet til HDTV med god margin ved en estimert 0,18 prosess.

Brukervennlighet og potensiale/allsidighet på en slik kompresjonsmodul kan tolkes som motstridende faktorer. Brukerpotensialet og allsidigheten til dynamikkompresjonsmodulen er meget bra siden brukeren har et meget stort parameterrom å justere bildet innenfor. Dette gjør at brukeren selv kan fremheve akkurat den stemningen eller de detaljene i bildet som brukeren selv er ute etter. Noen brukere finner derimot et slikt mangfold av muligheter lite brukervennlig, og brukeren er uansett avhengig av visuell tilbakemelding for å se om justeringene gir den ønskede effekten. Dersom dynamikkompresjonsmodulen blir plassert i et digitalt kamerasystem med en skjerm som viser bildet til enhver tid, og på den måten gir brukeren kontinuerlig tilbakemelding, vil antakeligvis de fleste brukere finne dynamikkompresjonsmodulen brukervennlig. En slik løsning er idag den vanligste på både digitale stillbildekameraer, digitale videokameraer og kameratelefoner, så dette kriteriet vil antakeligvis være oppfylt i de fleste kommersielle kamerasystemer.

Det er også en mulighet å stille inn kontrasten til maksimal verdi, og på den måten fremheve alle gradientene i bildet. Dette vil gi et visuelt dårlig bilde, men et bilde som er meget lett å lese maskinelt. Denne dynamikkompresjonsmodulen har derfor også potensiale til å gjøre stor nytte for seg i helautomatiske systemer, som for eksempel automatiske overvåkningssystemer og lignende.

Fordelene til kompresjonsmodulen som er presentert her er unikheten, de fysiske egenskapene, fleksibiliteten og potensialet. Siden det ikke er registrert noen andre ASIC-implementasjoner av dynamikkompresjonsalgoritmer er denne løsningen unik. Så fort høydynamiske kameraer nærmer seg introduksjon på markedet kommer sikkert slike løsninger til å bli mer vanlig, men for øyeblikket er det ingen tilsvarende løsninger tilgjengelig. Om man da ser på

digitale kameraer som bruksområde er de fysiske resultatene til denne dynamikkompresjonsmodulen bra. Det er selvfølgelig et forbedringspotensiale siden dette er et førsteutkast, men den når kravene til de mest aktuelle standardene som for øyeblikket dominerer markedet for levende bilder. Fleksibiliteten til dynamikkompresjonsmodulen kommer til syne ved to egenskaper. Arkitekturen er fleksibel på den måten at ved implementasjon av modulen i et større system, er det lett å bytte mellom seriell og parallell arkitektur i implementasjonsfasen. Denne fleksibiliteten gjør at man kan velge om man skal prioritere ytelse eller areal i en implementasjon. Mangfoldet i brukerparametere er også en fleksibilitet som kan være en stor fordel om man bruker parametrene riktig. Det store parameterrommet til inngangsparametrene representerer også potensialet til dynamikkompresjonsmodulen.

Ulempene til denne implementasjonen er begrensningene som den faste dynamikken på inngangssignalet og den faste bildestørrelsen gir. Logaritme- og antilogaritmeenheten er ikke gjort generisk, noe som gjør at bitbredden til inngangssignalet er låst til 18 bit. Modulen som beregner gjennomsnittet til inngangssignalet er satt til å beregne gjennomsnittet av 307200 sampler, noe som gjør at modulen er låst til å behandle VGAbilder.

I dette kapittelet har de forskjellige resultatene blitt vurdert og sammenlignet med alternative løsninger. På bakgrunn av disse vurderingene blir det i neste kapittel trukket konklusjoner om arbeidet samt foreslått videre arbeid.

6 Konklusjon og videre arbeid

Dette kapittelet avslutter vurderingen av arbeidet med dynamikkompresjonsmodulen ved å oppsummere hovedpunktene i diskusjonen og trekke konklusjon basert på disse. Deretter diskuteres det hvilken retning videre arbeid burde ta for å videreutvikle dynamikkompresjonsmodulen best mulig.

6.1 Konklusjon

Algoritmen Reinhards Fotoreseptormodell ble i prosjektoppgaven **[Hansen06]** valgt til å være best skikket til en hardwareimplementasjon ut fra kriteriene visuell kvalitet og kompleksitet. Den visuelle kvaliteten på resultatene etter kompresjonen i dynamikkompresjonsmodulen er like bra som etter kompresjon i Matlab, og algoritmens kompleksitet har gjennom både designprosessen og gjennom synteseresultatene vist seg å være meget overkommelig i en ASIC-implementasjon. Valget av algoritme har derfor vist seg å være et bra valg.

Siden den visuelle kvaliteten til Matlabimplementasjonen i **[Hansen06]** er beholdt i hardwareimplementasjonen, er det visuelle kravet tilfredsstilt.

Hardwareimplementasjonen av **[Reinhard05]** som er utført i denne masteroppgaven møter som nevnt i kapittel 5.2 ytelseskravene til primærapplikasjonene for høydynamisk bildetagning, så ytelsen på implementasjonen må i lys av dette sies å være bra.

Ressursbruken til dynamikkompresjonsmodulen, målt i gate count, er i den størrelsesorden man kan forvente av implementasjon av en slik kompleksitet **[Micron07]**. Det er vanskelig å vurdere denne ressursbruken opp mot konkurrerende løsninger, siden den eneste løsningen som kan konkurrere i ytelse er å dedikere et helt grafikkort i en pc til å utføre dynamikkompresjonen. Ut fra dette vurderes ressursbruken også til å være på et bra nivå, spesielt med tanke på at det er presentert to løsninger hvor man kan velge mellom fokus på ytelse eller arealbesparelse.

Generelt er det vanskelig å vurdere resultatet fra denne ASIC-implementasjonen opp mot konkurrerende løsninger siden det per dags dato ikke finnes noen andre løsninger som kan konkurrere på visuell kvalitet, ytelse og ressursbruk samtidig. Løsningen i denne masteroppgaven er derfor et meget bra utgangspunkt å bygge videre på når man skal utvikle et system for høydynamisk bildetagning. Denne løsningen når ytelseskravene til dagens bildestandarder til en overkommelig kostnad, men kan trenge litt videre arbeid for å finne optimale parameterverdier tilpasset de forskjellige applikasjonene.

6.2 Videre arbeid

Det er fortsatt noe videre arbeid som burde gjøres for å perfeksjonere dynamikkompresjonsmodulen implementert i denne masteroppgaven.

Den mest åpenbare utbedringene er å gjøre hele dynamikkompresjonsmodulen helt generisk, både med tanke på dynamikken på inngangssignalet og på bildeformat. Ved å gjøre dynamikken til inngangssignalet generisk vil man ikke være låst til 18 bits dynamikk, men vil kunne bruke denne modulen til flere forskjellige kamerabrikker med forskjellig dynamikk. Dette vil være en bra egenskap siden forskjellige applikasjoner kan kreve forskjellig dynamikk. Tilsvarende vil generisk bildestørrelse gjøre modulen mye mer fleksibel. Det er hovedsakelig kontrollmodulen og gjennomsnittsberegningen som må modifiseres for å gjøre modulen generisk for bildestørrelse (antall piksler per bilde).

En annen utbedring kan være å pipeline kompresjonsmodulen ytterligere. Det er hovedsakelig divisjonsenheten og multiplikatoren som da burde pipelines, siden det er disse enhetene som har den største logiske forsinkelsen. Dette vil føre til at ytelsen vil øke, samtidig som at arealet vil øke marginalt på grunn av innføringen av pipelineregistere. Dette er forøvrig bare en utbedring så sant man trenger bedre ytelse. I de applikasjonene hvor ytelsen til kompresjonsmodulen i denne oppgaven er tilstrekkelig, er det viktigere å begrense gate counten for å redusere kostnaden i en eventuell ASIC-implementasjeon.

Hardwaremodulen som er designet i denne masteroppgaven er som tidligere nevnt syntetisert i Leonardo Spectrum med NTNU sitt studentbibliotek som baserer seg på en 0.5-prosess. Hvor mye ytelsen øker ved å gå ned i prosess er bare estimert i denne oppgaven, og burde sjekkes ut ved å syntetisere modulen med et mer moderne bibliotek. Når man går ned i prosess vil ytelsen øke, og det vil være greit å vite med sikkerhet hvor mye den øker. Ved å bytte bibliotek vil sannsynligvis også gate counten forandre seg, og det vil være interessant å vite hva den ender på med et moderne bibliotek.

Ved å implementere kompresjonsmodulen i en FPGA som kan kobles opp mot en pc, kan man utforske parameterrommet til algoritmen på en interaktiv måte. Inngangen til kompresjonsmodulen kobles da inn mot et fiktivt kamera i FPGAen, som henter inn høydynamiske bilder fra RAM og sender de inn i kompresjonsmodulen. Samtidig kan parameterinngangene til kompresjonsmodulen styres fra pcen mens de ferdig komprimerte bildene som kommer ut fra kompresjonsmodulen vises på pc-skjermen. Om man setter sammen et slikt system vil man enkelt kunne utforske parameterrommet mens man ser resultatene på skjermen underveis. Dette vil være

meget nyttig for å se om man kan finne noen bedre algoritmer for beregning av initialverdier til parametrene, for å fastsette grenser til f_inn og m_inn, samt å finne optimale parameterverdier for faste applikasjoner. Et eksempel på en slik applikasjon kan være maskinell lesing av bilder.

En annen optimaliseringsmulighet kan være å se på hele kamerasystemet som en helhet, og eventuelt implementere flere av algoritmene nevnt i figur 2 inn i denne modulen. Ved å flytte denne modulen lenger frem i signalkjeden i figur 2, vil man kunne komprimere dynamikken rett ut fra kamerasensoren. Fordelen med dette er at bildet ut fra kamerabrikken er representert i bayer-format istedenfor RGB-format, noe som vil si at hvert piksel bare inneholder en farge. Ved å komprimere bildet i dette formatet vil man da kunne komprimere tre ganger så mange piksler per sekund med den samme kompresjonsmodulen. Ulempen er at algoritmene for hvitbalanse, autofokus med flere er avhengig av et lineært RGB-signal. Det vil altså være en krevende signalbehandlingsoppgave og få disse algoritmene til å skifte rekkefølge og fortsatt virke like bra, men om man greier dette så vil man kunne øke ytelsen på dynamikkompresjonen betraktelig.

Det er mange emner som må utforskes videre for å perfeksjonere dynamikkompresjonen ytterligere, og de punktene som er nevnt her vil gi en god start på arbeidet.

7 Litteraturliste

[Abed00] K. H. Abed, R. E. Sifred, CMOS VLSI Implementation of 16-bit Logarithm and Anti-logarithm Converters, Proc. 43rd IEEE Midwest Symp. on Circuits and Systems, Lansing MI, Aug 8-11.2000

[Abed03] K. H. Abed, R. E. Sifred, CMOS VLSI Implementation of a Low-Power Logarithmic Converter, IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 11, NOVEMBER 2003

[ARM_1] http://www.arm.com/products/CPUs/ARM7TDMI.html, 07.06.2007

[ARM_2] http://www.arm.com/support/faqip/3718.html, 07.06.2007

[ARM7] http://www.arm.com/pdfs/DDI0029G_7TDMI_R3_trm.pdf, 07.06.2007

[Deschamps06] J-P Deschamps, G. J. A. Bioul, G. D. Sutter, Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems John Wiley & Sons incorproated, 2006

[Detrey05] J. Detrey and F. de Dinechin. A parameterizable floating-point logarithm operator for FPGAs. In Asilomar Conference on Signals, Systems & Computers. IEEE, November 2005.

[Fukui01] Human Eyes as an Image Sensor, Feinberg School of Medicine, Northwestern University, Illinois, USA, 2001

[Goodnight03] N. Goodnight, R. Wang, C. Woolley, and G. Humphreys. Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware, Eurographics Symposium on Rendering 2003

[Hansen06] S-A Jervell Hansen, Dynamikkompresjon av høydynamiske bilder, Institutt for elektronikk og telekommunikasjon, NTNU, 2006

[Kleinschmidt75] J. Kleinschmidt and J. E. Dowling, Intracellular recordings from gecko photoreceptors during light and dark adaptation, J gen Physiol, vol. 66, pp. 617–648, 1975.

[Ledda05] P.Ledda, A.Chalmers, T.Troscianko, H.Seetzen. Evaluation of Tone Mapping Operators using a High Dynamic Range Display. ACM SIGGRAPH 2005, LA., ACM Press, August 2005.

[Maryland] http://www.csee.umbc.edu/help/VHDL/samples/add_g.vhdl, University of Maryland, Baltimore County, 07.06.2007

[Micron07] Personlig mailkorrespondanse, S. Yaghmai, J. Solhusvik, 2007.

[Mitchell62] J. N. Mitchell, Jr., "Computer multiplication and division using binary logarithms," IRE Transactions on Electronic Computers, side. 512-517, August 1962.

[Pineiro02] J.-A. Pineiro, M.D. Ercegovac, J.D. Bruguera Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors . pp 101, 2002

[Pineiro04] J.-A. Pineiro, M.D. Ercegovac, J.D. Bruguera, "Algorithm and Architecture for Logarithm, Exponential, and Powering Computation," IEEE Transactions on Computers, vol. 53, no. 9, pp. 1085-1096, Sept., 2004.

[Ramaswamy96] S. Ramaswamy and R. E. Siferd, "CMOS VLSI implementation of a digital logarithmic multiplier," in Proc. IEEE 1996 Nat. Aerospace Electron. Conf. NAECON 1996, Dayton, OH, May 1996, pp. 20-23

[Reinhard02] E.Reinhard, M.Stark, P.Shirley, J.Ferwerda. Photographic Tone Reproduction for Digital Images. In Proceedings of ACM SIGGRAPH 2002, Computer Graphics Proceedings, Annual Conference Series. ACM Press / ACM SIGGRAPH, July 2002.

[Reinhard05] E.Reinhard, K.Devlin. Dynamic Range Reduction inspired by Photoreceptor Physiology. IEEE Transactions on Visualization and Computer Graphics. 2005

[Reinhard06] E.Reinhard, G.Ward, S.Pattanaik og P.Debevec. High dynamic range imaging Acquisition, Display and image-based Lighting, Morgan Kaufmann series in computer graphics, 2006

[SanGregory99] S. L. SanGregory, C. Brothers, D. Gallagher, R Siferd, A Fast, Low-Power Logarithm Approximation with CMOS VLSI Implementation, 42nd Midwest Symposium on Circuits and Systems, 1999

[Wanhammer99]L. Wanhammer, DSP Integrated Circuits, Academic Press Series in Engineering, 1999

8 Vedlegg

8.1 Vedlegg 1 – Reinhards Fotoreseptormodell

```
% % % % Parameter   Description             Initial value       Operating range
% % % % m_inn       Contrast                0.3 + 0.7*k^1.4     [0.3;   1.0]
% % % % f_inn       Intensity               0.0                 [-8.0;  8.0]
% % % % c           Chromatic adaptation    0.0                 [0.0;   1.0]
% % % % a           Light adaptation        1.0                 [0.0;   1.0]

function out = PhPh(input_file, m_inn, f_inn, c, a)

% Leser inn pikselverdiene fra bildet
% read_rle_rgbe() er laget av Lawrence A. Taplin og er tilgjengelig på
% webadressen http://www.cis.rit.edu/mcsl/icam/hdr/read_rle_rgbe.m
I = read_rle_rgbe(input_file);

% Regner ut luminansen i hvert piksel samt finner minimumsveriden og den
% logaritmiske gjennomsnittsverdien
[height width dim] = size(I);
Lmin = 9999999;
Llav = 0;
for(i=1:1:height)
    for(j=1:1:width)
        Luminance(i,j) = 0.2125*I(i,j,1) + 0.7154*I(i,j,2) + 0.0721*I(i,j,3);
        Llav = Llav + log(1 + Luminance(i,j));
        if((Luminance(i,j) ~= 0)  && (Luminance(i,j)<Lmin))
            Lmin = Luminance(i,j);
        end
    end
end
Llmin = log(Lmin);
Llav = Llav/(height*width);
Lav  = exp(Llav);

% regner ut noen kompresjonsparametre
Lmax = max(max(Luminance));
Llmax = log(Lmax);
f=exp(-f_inn);
k = (Llmax-Llav)/(Llmax-Llmin);
if(m_inn == 0)
   m = 0.3 + (0.7*k^(1.4));
else
   m = m_inn;
end

% regner ut den globale adapsjonen
mean_intensity = mean(mean(I));
for(d=1:1:dim)
    Ia_global(d) = c*mean_intensity(d) + (1-c)*Lav;
end
```

```
% Går igjennom bildet og komprimerer piksel for piksel
for(i=1:1:height)
    for(j=1:1:width)
        for(d=1:1:dim)
            if(I(i,j,d)==0)
%               Setter verdien til 0 om pikselverdien er null
%               (for å slippe å dele på null i kompresjonen)
                out(i,j,d) = I(i,j,d);
            else
%               regner ut fargen ut fra kromatisk adapsjon
            Ia_local = c*I(i,j,d) + (1-c)*Luminance(i,j);
%               Regner ut adapsjonsnivå ut fra vekting mellom
%               Lokal og global adapsjon
            Ia = a*Ia_local + (1-a)*Ia_global(d);
%               Regner pikselets ut kompresjonsfaktor
            sigma = (f*Ia)^m;
%               Komprimerer
            out(i,j,d) = (I(i,j,d)/(I(i,j,d)+sigma));
            end
        end
    end
end
```

8.2 Vedlegg 2 – Reinhards Fotoreseptormodell – modifisert for hardwareimplmenetasjon

```
% % % % Denne funksjonen komprimere bilder etter [Reinhard05]
% % % % Parameter   Description             Initial value       Operating range
% % % % m_inn       Contrast                0.3 + 0.7*k^1.4     [-1.0 ;  1.0]
% % % % f_inn       Intensity               1.375*Lav           [-18.0; 18.0]
% % % % c           Chromatic adaptation    0.0                 [0.0  ;  1.0]
% % % % a           Light adaptation        1.0                 [0.0  ;  1.0]

function out = PhPh_2(input_file, m_inn, f_inn, c, a, output_file)

% Leser inn pikselverdiene fra bildet
% read_rle_rgbe() er laget av Lawrence A. Taplin og er tilgjengelig på
% webadressen http://www.cis.rit.edu/mcsl/icam/hdr/read_rle_rgbe.m
I = read_rle_rgbe(input_file);

% Skalerer inngangssignalet
I = I/max(max(max(I)));
Maks_inngang = 2^18;
I = round(I*Maks_inngang);

% Regner ut luminansen i hvert piksel samt finner minimumsveriden og den
% logaritmiske gjennomsnittsverdien
[height width dim] = size(I);
Lmin = 9999999;
Llav = 0;
for(i=1:1:height)
    for(j=1:1:width)
        Luminance(i,j) = 0.2125*I(i,j,1) + 0.7154*I(i,j,2) + 0.0721*I(i,j,3);
        Llav = Llav + log2(1 + Luminance(i,j));
        if((Luminance(i,j) ~= 0)  && (Luminance(i,j)<Lmin))
            Lmin = Luminance(i,j);
        end
    end
end
Llmin = log2(Lmin);
Llav = Llav/(height*width)
Lav  = 2^(Llav)

% regner ut noen kompresjonsparametre
Lmax = max(max(Luminance));
Llmax = log2(Lmax);

f_temp = floor(Llav * 1.375) + f_inn

f=2^(f_temp);
k = (Llmax-Llav)/(Llmax-Llmin);

m = 0.3 + (0.7*k^(1.4)) + m_inn;
```

```
% regner ut den globale adapsjonen
mean_intensity = mean(mean(I));
for(d=1:1:dim)
    Ia_global(d) = c*mean_intensity(d) + (1-c)*Lav;
end


% Går igjennom bildet og komprimerer piksel for piksel
for(i=1:1:height)
    for(j=1:1:width)
        for(d=1:1:dim)
            if(I(i,j,d)==0)
%               Setter verdien til 0 om pikselverdien er null
%               (for å slippe å dele på null i kompresjonen)
                out(i,j,d) = I(i,j,d);
            else
%               regner ut fargen ut fra kromatisk adapsjon
            Ia_local = c*I(i,j,d) + (1-c)*Luminance(i,j);
%               Regner ut adapsjonsnivå ut fra vekting mellom
%               Lokal og global adapsjon
            Ia = a*Ia_local + (1-a)*Ia_global(d);
%               Regner pikselets ut kompresjonsfaktor
            sigma = (f*Ia)^m;
%               Komprimerer
            out(i,j,d) = (I(i,j,d)/(I(i,j,d)+sigma));
            end
        end
    end
end
```

8.3 Vedlegg 3 – Syntese - Multiplikasjon vs logaritme

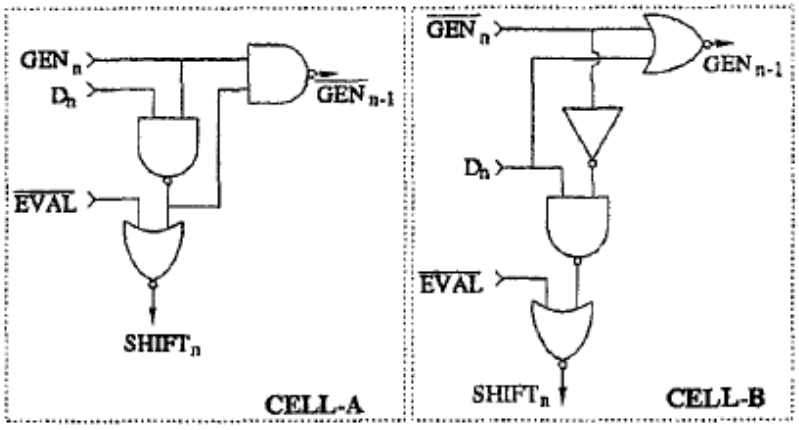
Synteseresultatene for logaritme-enhetene er oppført i tabell 9, og gjengitt i tabell 14. VHDL-koden for alle multiplikatorene er hentet fra **[Deschamps06]**, men er modifisert slik at alle har 18 bits bitbredde på inngangene. Logaritme-enhetene har også 18 bits bitbredde på inngangen. Både logaritme-enhetene og multiplikatorene er syntetisert i Leonardo Spectrum med det samme biblioteket for å få mest mulig sammenlignbare resultater.

Tabell 14: Synteseresultater for noen logaritme-enheter og noen multiplikatorer gitt i tilfeldig rekkefølge.

Logaritme-enheter	***SanGregory 99***	***Abed 00***	***Abed 03 SanGregory***	***Abed 03 2-region***	***Abed 03 3-region***	***Abed 03 6-region***
Delay	18.01	11.06	14.66	14.72	16.03	18.73
Gate-Count	1567	1362	1448	1433	1454	1606
Multipli-katorer	***Base2 - Multiplier***	***Ripple-Carry-multiplier***	***Carry-Save-Multiplier***	***Booth_1-multiplier***	***Booth_2-multiplier***	***Booth_3-multiplier***
Delay	36.15	46.29	24.45	45.72	83.76	64.74
Gate-Count	4460	6880	6108	6944	5635	6805

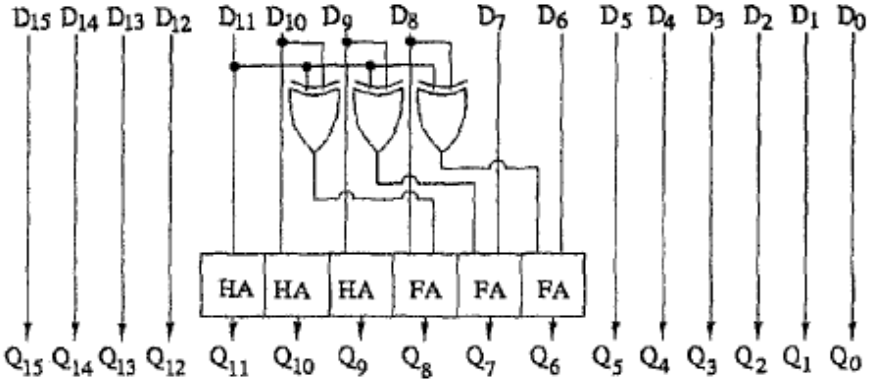
Fra tabell 14 ser man at samtlige multiplikatorer er vesentlig større og tregere enn logaritme-enhetene basert på lineær approksimasjon, og av den grunn blir logaritme-enheter som er avhengig av en slik multiplikator i implementasjonen ikke regnet for å være konkurransedyktig med tanke på areal og ytelse.

8.4 Vedlegg 4 – Deteksjons- og feilkorreksjonskrets fra SanGregory99



*Figur 24: Deteksjonskretsen for ledende ener er bygd opp av de to cellene Celle-A og Celle-B **[SanGregory99]**.*

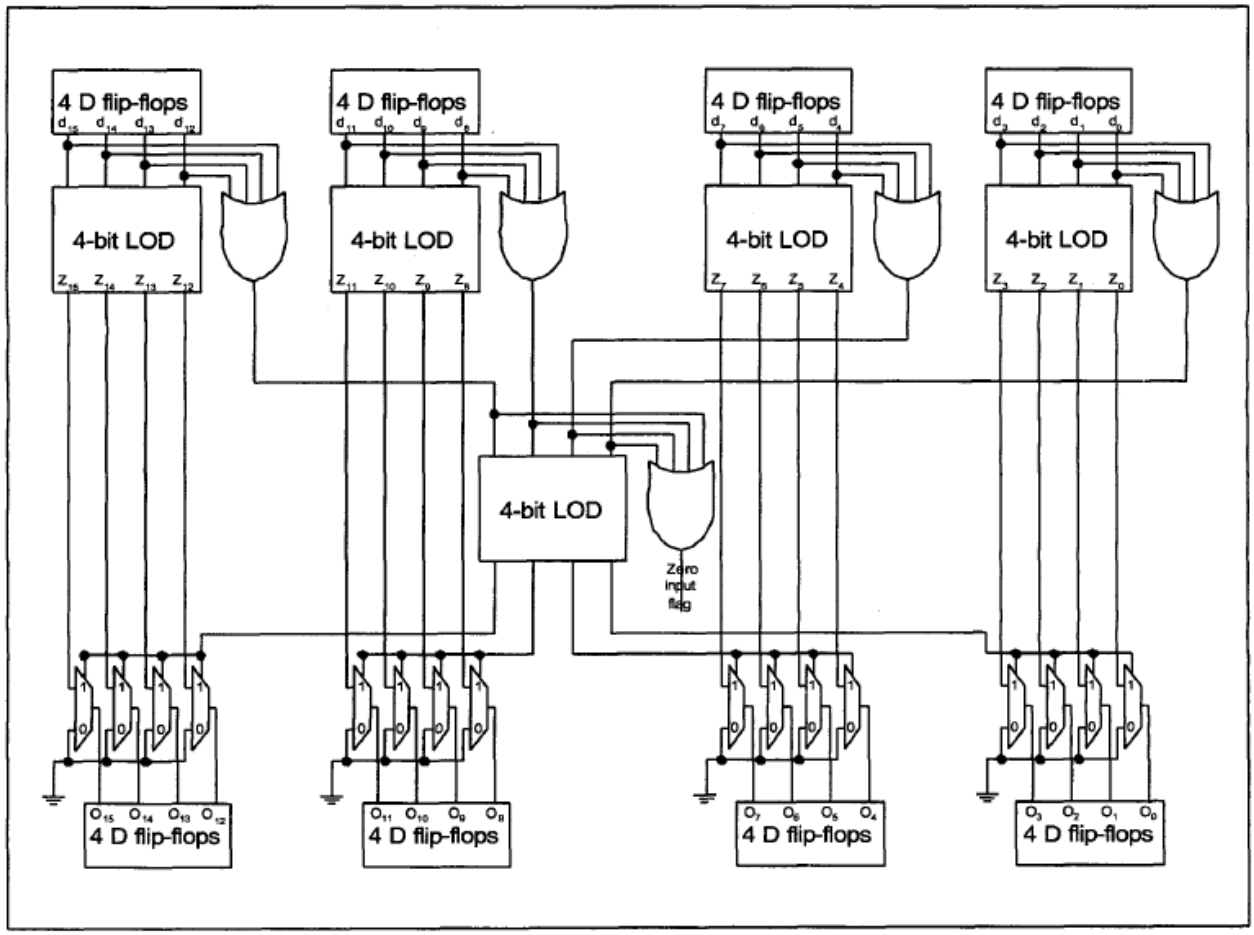
Cellene A og B fra figur 24 alternerer i et deteksjonsarray for å finne ledende ener. Dette vil si at de står annenhver gang i en rekke, og at det er en celle per bit i inngangssignalet. Et n bits inngangssignal trenger altså n antall celler for å detektere ledende ener. En ulempe med denne fremgangsmåten er at siden alle cellene kobles etter hverandre så vil deteksjonen skje serielt som i en ripple-carry adderer, og denne deteksjonskretsen vil derfor være ganske treg.



*Figur 25: Feilkorreksjonskrets fra **[SanGregory99]**.*

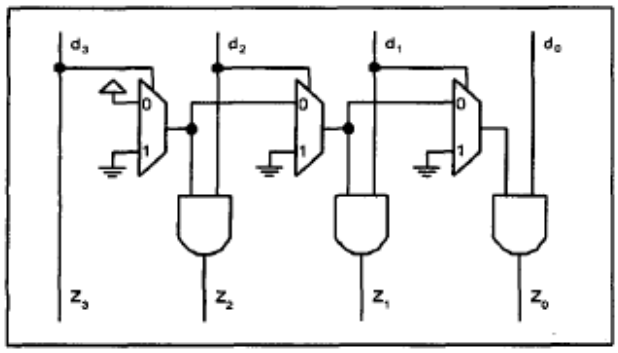
Feilkorreksjonskretsen fra figur 25 er feilkorreksjonskretsen fra **[SanGregory99]** som endrer approksimasjons-kurven fra Mitchell sin approksimasjonskurve til den nye approksimasjons-kurven vist i figur 6. I figur 25 er de fire bit'ene til venstre heltallsdelen i den 16-bits korreksjonskretsen, så bit D_{11} (MSB i desimaldelen) styrer formen på kurven. I VHDL-implementasjonen av denne kretsen er bitbredden 18 bit i motsetning til kretsen i figur 25 som har en bitbredde på 16 bit.

8.5 Vedlegg 5 – Deteksjonskrets for ledende ener fra Abed00



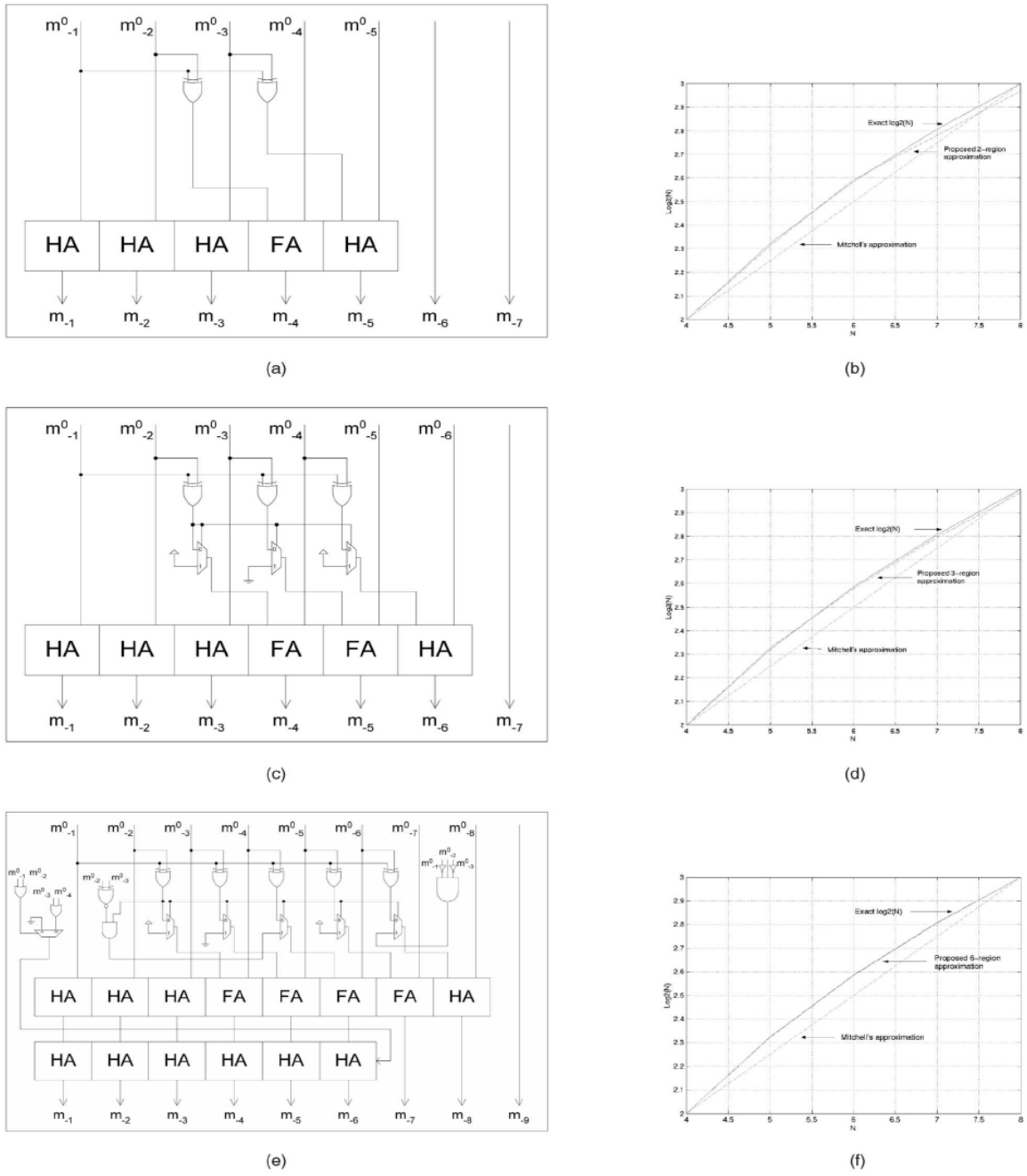
Figur 26: Leading One Detector - 16 bit ***[Abed00]***

Figur 27 viser en enkel 4 bits deteksjonskrets hvor ledende ener vil tilsvare det eneste høye signalet ut. Figur 26 viser et oppsett for hvordan man kan sette sammen en 16 bits deteksjonskrets ut fra den enkle byggblokken i figur 27. I den 18-bits versjonen som er implementert i forbindelse med denne oppgaven så er oppsettet i figur 26 tatt i bruk, men det er brukt 5-bits byggeblokker istedenfor 4-bits. De 5-bits deteksjonskretsene er laget etter prinsippet vist i figur 27, men utvidet med ett bit.



Figur 27: Leading One Detector - 4 bit ***[Abed00]***

8.6 Vedlegg 6 – Feilkorreksjonskretser fra Abed03



Figur 28: (a) Feilkorreksjonskrets for 2-regioner. (b) 2-regions approksimasjonskurve. (c) feilkorreksjonskrets for 3-regioner. (d) 3-regions approksimasjonskurve. (e) Feilkorreksjonskrets for 6 regioner. (f) 6-regions approksimasjonskurve.
m^0_{-1} representerer i dette tilfellet MSB i desimaldelen. ***[Abed03]***

8.7 Vedlegg 7 – Kromatisk- og lysadapsjon

Figur 29: Hvordan kromatisk- og lysadapsjon påvirker bildet.

Kromatisk- og *lysadapsjon* er parametre som enten kan være brukerstyrt eller de kan fastesettes før implementasjon. Ved å definere de som brukerstyrte parametre vil designet bli mer fleksibelt, men ved å sette de konstante før implementasjonen vil det lette arbeidet i selve implementasjonen, samt at algoritmen vil bruke mindre ressurser under eksekvering. Designet blir derimot mindre fleksibelt. Dette er avveininger som må vurderes i en designprosess. Figur 29 viser hvordan disse parametrene påvirker bildet ved å justere *kromatisk-* og *lysadapsjon*. *Kromatisk adapsjon* har i figuren verdiene 0, 0.5 og 1 i stigende rekkefølge fra topp til bunn. *Lysadapsjon* har i figuren verdiene 0,0.5 og 1 i stigende rekkefølge fra venstre mot høyre.

8.8 Vedlegg 8 – Dynamikkompresjon på mikrokontroller

Et alternativ til å implementere dynamikkompresjonsalgoritmen i hardware vil være å implementere algoritmen på en mikrokontroller. Algoritmen er ikke implementert på mikrokontroller i denne masteroppgaven, men ytelsen til en mikrokontrollerimplementasjon kan allikevel estimeres. Tabell 15 er hentet fra **[Hansen06]**, og viser antall operasjoner dynamikkompresjonsalgoritmen bruker på å komprimere ett VGA-bilde.

*Tabell 15: Antall matematiske operasjoner i **[Reinhard05]***

	Add	***Sub***	***Mult med Konstant***	***Mult***	***Divisjon***	***Logaritme***	***Eksponentiering***
Før kompresjon	921603	2	921607	0	2	307204	2
Under kompresjon	2764800	0	2364803	921600	921600	0	921600

Tabell 16 viser hvor mange klokkepulser en ARM7 bruker på å utføre de forskjellige operasjonene. Add, sub, operasjon+shift og mult er operasjoner som står i instruksjonssettet til ARM7, og informasjonen om disse er hentet fra dette instruksjonssettet **[ARM7]**. For enkelthetens skyld taes det ikke hensyn til forskjellen mellom sykel-typene, men det antas at alle syklene er av den hurtigste, sekvensielle typen. Forskjellen mellom sykeltypene er forøvrig beskrevet i **[ARM7]**. Divisjon, logaritme og eksponentiering er ikke standard-operasjoner på en mikrokontroller, og

Tabell 16: Antall sykler per operasjon på en ARM7. Normal dataprosessering er logiske operasjoner, sammenligning av verdier og flytting av data mellom forskjellige registre.

Operasjon	**Sekvensiell-sykler**	**Intern-sykler**	**Totalt antall sykler**
Normal dataprosessering	1	0	1
Add	1	0	1
Sub	1	0	1
Operasjon+Shift	1	1	2
Mult	1	1-16	2-17
Divisjon			96
Logaritme			56
Eksponentiering			3

antall sykler som trengs for å beregne disse operasjonene er derfor estimert.

Divisjon er ikke en standardrutine i en mikrokontroller, men databladet til ARM7 foreslår en algoritme for divisjon som er slik ut:

```
                            ; enter with numbers in Ra and Rb
      MOV   Rcnt,#1         ; bit to control the division
Div1  CMP   Rb,#0x80000000  ; move Rb until greater than Ra
      CMPCC Rb,Ra
      MOVCC Rb,Rb,ASL#1
      MOVCC Rcnt,Rcnt,ASL#1
      BCC   Div1
      MOV   Rc,#0
Div2  CMP   Ra,Rb           ; test for possible subtraction
      SUBCS Ra,Ra,Rb        ; subtract if ok
      ADDCS Rc,Rc,Rcnt      ; put relevant bit into result
      MOVS  Rcnt,Rcnt,LSR#1 ; shift control bit
      MOVNE Rb,Rb,LSR#1     ; halve unless finished
      BNE   Div2
                            ; divide result in Rc
                            ; remainder in Ra
```

Denne divisjonsalgoritmen er iterativ, og har en iterasjon per bit i svaret. Siden divisjonen i dynamikkompresjonsalgoritmen genererer et 8-bits svar, vil denne divisjonsalgoritmen ha 8 iterasjoner. Hver iterasjon inneholder på det meste 12 en-sykels operasjoner, så en divisjon vil derfor i verste fall bruke 96 klokkesykler.

Logaritme er heller ingen standardoperasjon på en mikrokontroller, men den lineære approksimasjonsmetoden som er beskrevet i kapittel 2.2.3 kan også brukes på en mikrokontroller. Først teller man antall ledende nullere for å finne den ledende eneren. Så bruker man dette antallet som heltallsdel og skifter inngangsverdien slik at sifferet etter den ledende eneren blir MSB i desimaldelen til svaret. Algoritmen vil da bruke 3 sykler på hver iterasjon mens den teller seg frem til den ledende eneren, en sykel for å sjekke, en sykel for å inkrementere telleren og en sykel for å hoppe tilbake til starten av løkka. Når den har telt seg ferdig vil den bruke to sykler for å sette sammen desimaldelen og heltallsdelen i svaret siden desimaldelen må skiftes x-antall plasser før de kan settes sammen. Siden inngangssignalet er 18 bit vil en logaritme da i verste fall bruke

$$18 * 3 + 2 = 56 \, sykler$$

Den siste operasjonen som ikke er standard i mikrokontrolleren er antilogaritme. Antilogaritmen kan estimeres på samme måte som logaritmen, noe som er beskrevet i **[Abed00]**. Dette innebærer at desimaldelen skilles ut fra inngangsverdien med en ledende ener foran seg, og venstreskiftes så mange ganger som heltallsverdien tilsier. Her vil mikrokontrolleren bruke en sykel på å lese ut heltallsverdien, en sykel på å lage den midlertidige verdien med en ledende ener foran desimal-verdien, før den bruker den siste sykelen på å skifte den midlertidige verdien det antall

ganger som heltallsverdien tilsier. Tre sykler tilsammen på en antilogaritme. En mer grundig innføring i denne metoden er gitt i **[Abed00]**.

Hvilke operasjoner som må gjøres før kompresjonen og hvilke som må gjøres under kompresjonen blir irrelevant for en mikrokontroller, siden den ikke kan parallellisere oppgavene men må gjøre alt sekvensielt. Verdiene i de to radene i tabell 15 blir derfor summert, og tabell 17 viser antall sykler som må til totalt sett for å komprimere ett VGA-bilde på en mikrokontroller.

Tabell 17: Antall klokkesykler en mikrokontroller vil bruke på å komprimere dynamikken i ett VGA-bilde.

	Add	**Sub**	**Mult m/ konstant**	**Mult**	**Div**	**Log**	**Eksp**
Før komp	921603	2	921604	0	2	307204	2
Under komp	2764800	0	2364803	921600	921600	0	921600
Sum operasjoner	3686403	2	3286410	921600	921602	307204	921602
# clk per operasjon	1	1	9	17	96	56	3
Sum sykler	3686403	2	29577690	15667200	88473792	17203424	2764806

Multiplikasjonen blir her regnet ut fra verste-tilfelle scenario, mens det er antatt at en multiplikasjon med konstant kan forenkles slik at det gjennomsnittlig brukes ca halvparten av klokkesyklene på en slik multiplikasjon.

Antall klokkesykler som blir brukt på å komprimere dynamikken i ett VGA-bilde blir da tilsammen 157.373.317, altså ca 157 millioner klokkesykler per VGA-bilde. En ARM7TDMI kan ifølge ARM sin webside **[ARM_1]** yte opp til 236 Mhz i en 90 nanometers prosess med en gate-count på ca 75K **[ARM_2]**. Tabell 18 viser hvordan ytelsen varierer med prosess:

*Tabell 18: Klokkefrekvens til en ARM7 ved forskjellige prosesser. dataene er hentet fra ARM sin webside **[ARM_2]**.*

Prosess:	**0.18, Speed opt.**	**0.13, Speed opt.**	**90 nm, speed opt**
Frekvens [MHz]	115	133	236

Den raskeste varianten av en ARM-prosessor vil med disse tallene kunne komprimere 1.5 VGA-bilde i sekundet.

Det er viktig å merke seg her at dette kun er et estimat basert på antall matematiske operasjoner, og at en mikrokontroller vil måtte håndtere en stor mengde minneoperasjoner i tillegg til de matematiske operasjonene i en reell implementasjon. Hvor krevende disse minneoperasjonene vil være i dette tilfellet er vanskelig å estimere uten en reell mikrokontrollerimplementasjon av algoritmen, men det er grunn til å tro at mikrokontrolleren sin ytelse vil kunne bli merkbart redusert.

8.9 Vedlegg 9 – Oversikt over VHDL-filer

Table 19: Oversikt over alle VHDL-filene som er lagt ved oppgaven. Kapittel-referansen henviser til hvilket kapittel i denne oppgaven som modulen er beskrevet.

Filnavn (.vhdl)	**Kapittel**	**Beskrivelse**
abed03_2region_error_ correction	2.2.3.1.3	2-regions korrkesjonskrets til bruk i lineære approksimasjons logaritme-moduler.
abed03_3region_error_ correction	2.2.3.1.3	3-regions korrkesjonskrets til bruk i lineære approksimasjons logaritme-moduler.
abed03_6region_error_ correction	2.2.3.1.3	6-regions korrkesjonskrets til bruk i lineære approksimasjons logaritme-moduler.
add_g	3.2.1.1	Adderer, fra **[Maryland]**.
add_g_tb		Testbenk til add_g.
addsub	3.2.1.1	Enhet som kan addere og subtrahere. Brukes som subtraherer.
addsub_tb		Testbenk til addsub. Tester både addereren og subtrahereren.
antilog_shifter18		Submodul i antilogarithm_18 modulen. Står for selve den logaritmiske skiftingen av inngangssignalet.
antilogarithm_18	3.2.1.5	18-bit antilogaritme-modul som benytter seg av lineær approksimasjon etter prinsipp beskrevet i [Abed00].
antilogarithm_18_tb		Testbenk til antilogarithm_18. Logger inngang og utgang i en txt-fil som kan lese inn i matlab for å beregne estimeringsfeilen der.
avg	3.2.2.6	Beregner gjennomsnittet av 307200 sampler (tilpasset et VGA-bilde) .
avg_tb		Testbenk for avg-modulen. Beregner estimeringsfeil.
carry_save_mult	3.2.1.2	Multiplikatoren fra **[Deschamps06]**. Filen inneholder også en testbenk fra **[Deschamps06]** som er kommentert ut for syntetisering.
comp	3.2.2.4	Kompresjonsmodulen.
comp_tb		Testbenk for kompresjonsmodulen.
control_big		Kontrollmodulen for den parallelliserte arkitekturen.
control_small		Kontrollmodulen for den serielle tidsmultipleksede arkitekturen.
f_inn	3.2.2.7	Modul som beregner initialverdien til f. *Ut = 1.375*Inn.*
f_inn_tb		Testbenk til modulen f_inn.

full_adder		2-bits fulladderer som er en undermodul til feilkorreksjonskretsene i **[Abed03]**.
half_adder		2-bits halvadderer som er en undermodul til feilkorreksjonskretsene i **[Abed03]**.
Ia_add	3.2.2.2	Modul som beregner adapsjonsnivået. Brukes til å beregne både lokalt, globalt og totlat adapsjonsnivå.
Ia_tb		Testbenk til modulen Ia.
leading_one_18	2.2.3.1.2	Deteksjonskrets for ledende ener. Til bruk i SanGregory99.
leading_one_abed00		Deteksjonskrets for ledende ener. Til bruk i Abed00 og Abed03.
leading_one_sub		Undermodul til bruk i leading_one_18 modulen.
leading_one_sub_abed00		Undermodul til bruk i leading_one_abed00 modulen.
logarithm_abed00	2.2.3.1.3	Modul som beregner logaritme ved hjelp av lineær approksimasjon. Ingen feilkorreksjon.
logarithm_abed00_tb		Testbenk for logarithm_abed00.
logarithm_abed03_ 2region	2.2.3.1.3	Modul som beregner logaritme ved hjelp av lineær approksimasjon. 2-regions feilkorreksjon fra Abed03.
logarithm_abed03_ 2region_tb		Testbenk for logarithm_abed03_2region.
logarithm_abed03_ 3region	2.2.3.1.3	Modul som beregner logaritme ved hjelp av lineær approksimasjon. 3-regions feilkorreksjon fra Abed03.
logarithm_abed03_ 3region_tb		Testbenk for logarithm_abed03_3region.
logarithm_abed03_ 6region	2.2.3.1.3	Modul som beregner logaritme ved hjelp av lineær approksimasjon. 6-regions feilkorreksjon fra Abed03.
logarithm_abed03_ 6region_tb		Testbenk for logarithm_abed03_6region.
logarithm_abed03_ sangregory		Modul som beregner logaritme ved hjelp av lineær approksimasjon. Feilkorreksjon fra SanGregory99.
logarithm_abed03_ sangregory_tb		Testbenk for logarithm_abed03_sangregory.
logarithm_sangregory	2.2.3.1.2	Modul som beregner logaritme ved hjelp av lineær approksimasjon.
logarithm_sangregory_tb		Testbenk for logarithm_sangregory.
luminance	3.2.2.1	Modul som beregner luminansen i hvert piksel.
luminance_tb		Testbenk for luminans-modulen.
maksimum	3.2.2.5	Modul som finner og tar vare på den største verdien i en bitstrøm.

maksimum_tb		Testbenk til maksimum-modulen.
minimum	3.2.2.5	Modul som finner og tar vare på den minste verdien i en bitstrøm.
minimum_tb		Testbenk til minimum-modulen.
param	3.1.2	En overordnet modul som inneholder hele parameterberegningen. Undermodulene m og k i kapittel 3.2.2.8 og 3.2.2.9 er implementert som prosesser i denne modueln, siden m og k bruker mye funksjonalitet fra andre moduler.
RGB_mux		En multiplekser som brukes i toppnivået til den serielle tidsmultipleksede modulen for å gi signalkompresjonen riktig fargesignal.
ROM18x5		18 til 5 bits dekoder til bruk i logaritme-enhetene.
sangregory_error_ correction	2.2.3.1.2	Feilkorreksjonskrets til bruk i en logaritmemodul som benytter seg av lineær approksimasjon.
Shifter18		18-bits logaritmisk skifter til bruk i logaritme-enhetene.
sigma	3.2.2.3	Modul som beregner kompresjonsparameteren sigma.
sigma_tb		Testbenk til sigma-modulen.
signal_pipe	3.1.2	En overordnet modul som inneholder hele signalkompresjons-pipelinen.
signal_pipe_tb		Testbenk til den overordnede modulen signal_pipe.
tb		Testbenk til toppnivået til den store, parallelle arkitekturen. Tar inn modulen top_level_big som DUT.
tb_small		Testbenk til toppnivået til den lille, serielle arkitekturen. Tar inn modulen top_level_small som DUT.
top_level_big	3.1.1	Øverste toppnivåmodulen for den parallelle arkitekturen. Her blir undermodulene knyttet sammen.
top_level_small	3.1.1	Øverste toppnivåmodulen for den serielle arkitekturen. Her blir undermodulene knyttet sammen.

Filene er fordelt i mapper slik at hver logaritmeløsning har sin egen mappe, og hver av arkitekturene til dynamikkompresjonsmodulen har sin egen mappe. Denne løsningen er valgt slik at filene lett kan sees i sammenheng med hverandre, selv om dette betyr at flere av filene er representert i flere kataloger.

Testbenkene til de aritmetiske og sammensatte modulene er lagt i en egen mappe for at mappene med selve designet skal se mer ryddig ut.