Utvikling av et ekkoloddsystem

Andreas Fagerland Haavik

Oppgave for graden
Master i Mikroelektronikk og Sensorteknologi
60 studiepoeng

Fysisk Institutt
Det matematisk-naturvitenskapelige fakultet

UNIVERSITETET I OSLO

Våren 2022

Utvikling av et ekkoloddsystem

Andreas Fagerland Haavik

Utvikling av et ekkoloddsystem

http://www.duo.uio.no/

Trykk: Reprosentralen, Universitetet i Oslo

Takk til ...

Nå som arbeidet med denne oppgaven nærmer seg slutten, ønsker jeg å benytte anledningen til å takke de som har hjulpet å realisere den.

Først ønsker jeg å takke min veileder, førsteamanuensis Helge Balk, for at jeg fikk muligheten til å jobbe med denne oppgaven. Den har gitt meg bredere praktisk erfaring i arbeid med FPGA, i tillegg til å dyrke en liten interesse for hydroakustikk. Jeg vil også takke ham for å ha tatt seg tid til opplæring i sonarteori og for all hjelp underveis.

Videre ønsker jeg å takke mine medveiledere professor Sverre Holm og professor Ketil Røed. I tillegg til å ha gitt god hjelp med både teoretiske og praktiske problemer som har oppstått, har de tatt seg tid til samtaler og gjennomlesing av utkast. Jeg vil også takke senioringeniør Halvor Strøm på ELAB, for hjelp med produksjonen av kretskort og senere modifikasjoner på dette.

Takk til mine medstudenter på rom 333 for interessante faglige diskusjoner, gode tips og ellers hyggelige samtaler.

Til slutt vil jeg takke min samboer Ingvild, som sa seg villig til å lese korrektur, og som ellers har vært veldig hjelpsom og tålmodig.

Abstrakt

Denne oppgaven inngår som en del av flere oppgaver hvor målet er å lage et rimelig og fleksibelt forskningsekkolodd. Her fokuserer vi på ekkoloddmottakerens sluttrinn og bruker et utviklingskort med en Xilinx Zynq-7020 SoC FPGA. Fokuset er på å konstruere en mottaker med lite analog elektronikk og med stort dynamikkområde. Lite analogelektronikk oppnås ved å digitalisere det høyfrekvente ekkosignalet rett etter forforsterkerne. Stor dynamikk oppnås ved en form for kaskadekobling av to 12-bits AD-omformere. I oppgaven utvikles kaskadekoblingen med beskyttelseskretser og kanalvelger, moduler for uthenting av omhyllingskurve og for overføring av data til ekstern lagringsenhet.

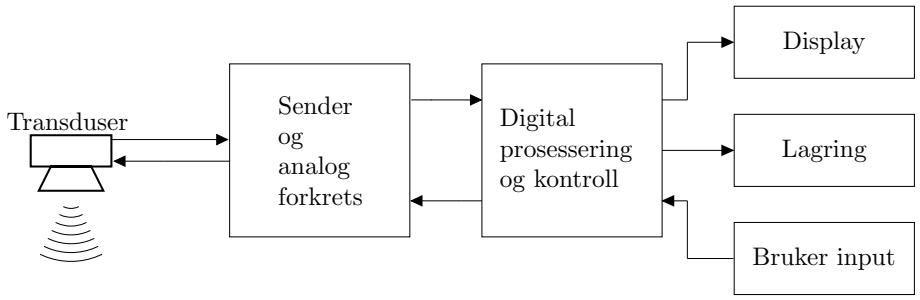
Innhold

1 Introduksjon

Det finnes i dag en mengde ulike ekkolodd fra mange forskjellige produsenter. Vi har billige fiskefinner ekkolodd laget for hobbymarkeder, med kort rekkevidde, bred stråle og lav følsomhet. Vi har også kostbare og kraftige ekkolodd som brukes av forskere rundt i verden til bestandsestimering, bunndeteksjon og lignende. De enkleste ekkoloddene bruker puls med kontinuerlig bølge (CW) mens de mer avanserte ekkoloddene bruker flere stråler og også ofte frekvensmodulerte signaler.

Blant firmaer som produserer ekkolodd er Simrad - Kongsberg Maritime, BioSonics, HTI, Furuno, Knudsen, Humminbird, Lowrance og Garmin. Simrads splittstråle ekkoloddserie som EK60 og EK80 er eksempler på svært populære vitenskapelige ekkolodd med høy kvalitet.

Denne oppgaven går ut på å utvikle deler av et fullstendig ekkoloddsystem for forskningsanvendelser. En grov oversikt over et generelt ekkoloddsystem er vist i figur 1.



Figur 1: Blokkskjema over generelt ekkoloddsystem.

Det består av:

- En sender/mottaker (transduser) for å sende lydpuls og motta ekko.
- Analog forkrets (forsterkere, anti-aliasing filtre osv.).
- Prosseseringselementer for å hente ut og tydeliggjøre ønsket data. Dette kan gjøres analogt og digitalt.
- Display for å vise data til operatør.
- Lagring for etteranalyse.
- Mulighet for manuell innstilling av nødvendige parametre.

Delene som fokuseres på i denne oppgaven er prosesseringsdelen og overføring av data til ekstern lagringsenhet. Prosesseringen skal gjøres digitalt, og FPGA ble valgt som basis for utvikling av systemet. I tillegg legges det frem forslag til løsninger for deler av den analoge forkretsen.

1.1 Problemstilling og motivasjon

Hovedproblemet med eksisterende ekkoloddsystem er at de kan være kostbare og vanskelige å frakte med seg innlands til bruk i for eksempel elver og innsjøer. Det vil ofte være ønskelig å sette ut flere system for å dekke et større område. Dette kan fort bli dyrt og logistisk utfordrende. For mindre forskningsgrupper med begrensede midler blir dermed slike løsninger ikke aktuelle.

1.2 Målsetting

Systemet presentert i oppgaven er utviklet for å dekke et behov for billige og kompakte løsninger, som muliggjør bruk av flere systemer og frakt til avsidesliggende områder. Målet er å skape et fleksibelt system som det skal være mulig å operere manuelt og som i tillegg kan settes ut for å samle data automatisk over lengre tidsperioder. Aktuelle bruksområder vil være fra fartøy, stasjonært montert på bøye eller senket ned på vannbunn.

Fleksibiliteten forsterkes ved at systemet er implementert på FPGA, som tillater delvis eller fullstendig rekonfigurasjon av systemet ved behov. Dette gjør det mulig med videreutvikling av systemet i form av for eksempel forbedring eller tilføring av ny funksjonalitet.

2 Teori

For å utvikle et system som skal ta imot og behandle akustisk data, er det nødvendig med grunnleggende kunnskap om hydroakustikk samt utstyret dataen skal behandles på. Det er viktig å vite hvordan signalet ser ut, hvordan det skal behandles for at man skal kunne tolke det riktig, og hva som er nødvendig for å realisere et slikt system.

2.1 Hydroakustikk

Ekkolodd er et måleinstrument som bruker lyd for å detektere objekter eller måle avstand. Man skiller ofte mellom to typer ekkoloddsystem; aktivt og passivt. Et aktivt ekkoloddsystem sender ut en lydpuls, også kalt et *ping*, og lytter etter refleksjoner (ekko). Et passivt ekkoloddsystem lytter etter bølger generert fra andre eksterne kilder [18]. Systemet presentert i denne oppgaven er tilpasset et aktivt system, men kan med små endringer også brukes i et passivt system.

Bruksområder varierer fra deteksjon av ubåter og sjøminer, til kartlegging og klassifisering av havbunn og overvåking av fiskebestander.

2.1.1 Viktige matematiske uttrykk

Lydens hastighet i vann er på rundt 1500 m/s avhengig av temperatur og saltinnhold. Avstanden til et objekt kan dermed uttrykkes som halvparten av tiden t det tar ekkoet å returnere multiplisert med hastigheten c [18]:

$$R = c\frac{t}{2} \tag{1}$$

Lyd er trykkbølger som propagerer gjennom et medium. På vei gjennom mediet avtar intensiteten. Dette skjer hovedsakelig av to grunner [18][27]:

- **Geometrisk spredning** : Ettersom trykkbølgene brer seg sfærisk, vil bølgefrontens intensitet bli fordelt over et stadig større område.
- **Absorpsjon** : En del av bølgefrontens intensitet vil gå tapt til omgivelsene grunnet mediumets viskositet, i tillegg til relaksasjon av molekyler i mediumet. Graden av aborbsjon er frekvensavhengig.

Intensiteten på det returnerte ekkoet er uttrykt matematisk i sonarligningene 2 og 3 [27]. Ligning 2 beskriver ekko fra et enkelt objekt, mens ligning 3 beskriver romklang i vannet. Kilden til romklang kan være fiskestim,

eller refleksjon fra flere flater som bunn og overflate eller kantete undervannsskjær. Begge ligningene er forenklede i den forstand at de ikke inkluderer støy.

$$EL = SL - 2TL + TS \tag{2}$$

$$RL = SL - 2TL + Sv + PV \tag{3}$$

- ***EL*** (Echo Level) / ***RL*** (Reverberation Level) er størrelsen på det returnerte signalet vi observerer. Det er *EL* og *RL* som ekkoloddet måler.
- ***SL*** (Source Level) er intensiteten på det utsendte signalet.
- ***TL*** (Transmission Loss) er størrelsen på intensitetstapet, som oppstår både til og fra, og gir dermed et dobbelt negativt bidrag.
- ***TS*** (Target Strength) er en størrelse avhengig av det akustiske tversnittet til et objekt.

Dermed kan amplituden til ekkoet si noe om størrelsen på objektet som returnerte det. Ettersom TL er bestemt av faktorer som frekvens, temperatur, trykk og saltinnhold, kan denne regnes ut. EL og SL er kjente størrelser, og vi kan dermed omformulere ligningen 2 til en utregning av TS, som betyr at vi også kan regne ut objektets akustiske tversnittsareal, og videre objektets størrelse.

$$TS = EL - SL + 2TL \tag{4}$$

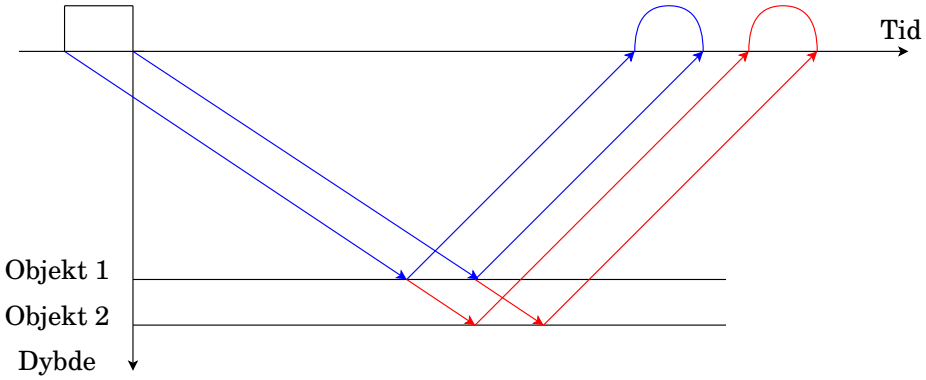
Dersom systemet skal være brukbart i overvåking av fiskebestand, vil kalibrering for å gi en nøyaktig verdi for EL være viktig.

2.1.2 Transduser og lydpulsen

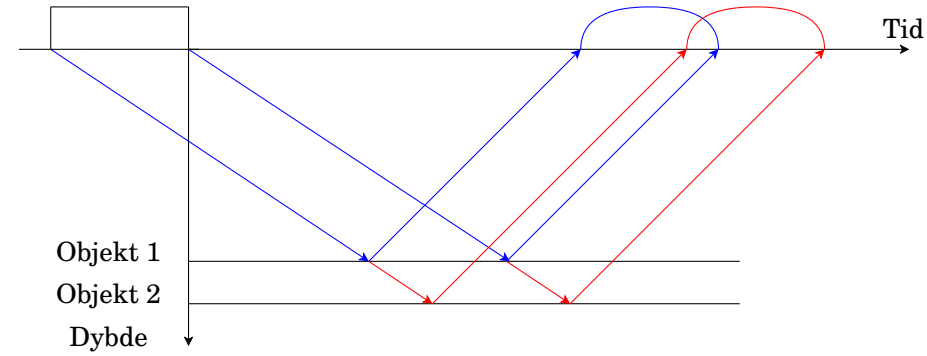
Ekkoloddsystemet er avhengig av en transduser. Ved sending oversetter den elektrisk spenning til akustiske bølger. I lyttemodus fungerer den som en hydrofon som oversetter akustiske bølger tilbake til elektriske signaler [18].

Lydpulsens varighet vil ha innvirkning på avstandsoppløsningen til systemet. Kortere varighet på pulsen betyr at objekter kan være nærmere hverandre før ekkoene fra dem overlapper. Overlappende ekko kan bli tolket som et sammenhengende ekko fra et mye større objekt. En visuell fremstilling er vist i figur 2 og 3. På en annen side vil kort varighet på pulsen bety

at mindre energi blir sendt ut. Den totale utsendte energien vil bestemme hvor små objekter som kan detekteres, og hvor dypt en kan se. Dette fører til at bestemmelse av pulslengden blir en avveiing mellom avstandsoppløsning og mengde energi som kan sendes ut.



Figur 2: Kort senderpuls gjør at ekko kan skilles fra hverandre.



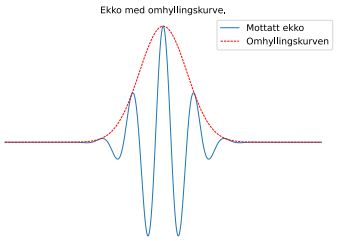
Figur 3: Lang senderpuls gjør at ekko overlapper, som gjør at objektene blir vanskeligere å skille.

En mulighet for å få både god avstandsoppløsning og mye utsendt energi er å bruke frekvensmodulerte bølger, kjent som et *sweep*. Ved å sende pulser med lang varighet og varierende frekvens får en både sendt ut mye energi, samtidig som en kan bruke pulskompresjon for å få en god avstandsoppløsning. En ulempe med en slik løsning er at mer signalprosessering kreves.

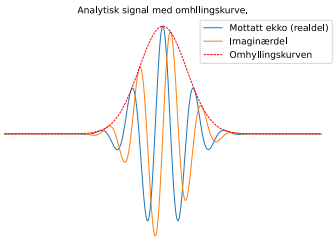
Til sist er det verdt å nevne at lydbølgens frekvens setter en nedre grense på hvor kort en puls kan være. Dersom lydpulsen blir for kort i forhold til bølgelengden, vil ikke transduseren klare å sende ut en tilstrekkelig mengde svingninger.

2.1.3 Omhyllingskurven

Det returnerte signalet vil ha en bærebølge med frekvens omtrent lik frekvensen til det utsendte signalet. Informasjonen vi ønsker avhenger av ekkoets bredde og amplitude. Dermed kan de høyfrekvente komponentene i ekkoet fjernes, og vi kan se bare på omhyllingskurven.



Figur 4: Signal med omhyllingskurve.



Figur 5: Komplekst signal med omhyllingskurve.

Omhyllingskurven, eller *envelopen*, kan finnes numerisk ved at en ser på magnituden av det komplekse signalet, som beskrevet i ligning 5 og visualisert i fig. 5. Signalet $x_c(t)$ er markert med c fordi den er kompleks, og består av det en realdel $x_r(t)$ og imaginærdel $x_i(t)$ [17].

$$E(t) = |x_c(t)| = \sqrt{x_r(t)^2 + x_i(t)^2} \tag{5}$$

Realdelen av signalet er det originale signalet som systemet mottar. Den imaginære delen er et resultat av en *Hilbert transformasjon*, som faseskifter det originale singalet med $\pm 90°$.

2.2 FPGA

Field Programmable Gate Array (FPGA) er en form for programmerbar maskinvare. Den består av et sett med logiske blokker som kan settes sammen for å danne forskjellige kretser. Systemets funksjonalitet beskrives i maskinvarebeskrivende språk som Very-High-Speed Integrated Circuit Hardware Description Language (VHDL) eller Verilog, som så blir oversatt til koblinger og lastet opp på FPGA-en. En FPGA tillater komplekse og spesialiserte kretser.

Dette kan bli sett på som et alternativ til et system basert på en mikro-

kontroller. Et slikt system har en fast maskinvarearkitektur som man kontrollerer med et sett instruksjoner gjennom programmering i språk som Python og C. Slike systemer er ofte også fleksible.

En annet alternativ er en Application Specific Integrated Circuit (ASIC), som er en integrert krets designet for et spesifikt formål. En ASIC kan tilby god ytelse i tillegg til å være billigere per enhet sammenlignet med en FPGA, men tilbyr ikke samme rekonfigurerbarhet som en FPGA gjør.

2.2.1 Hvorfor FPGA?

Hovedgrunnen til at FPGA ble valgt for denne oppgaven er fleksibiliteten og rekonfigurerbarheten som den tilbyr. På grunn av dette kan moduler forbedres eller flere moduler legges til ved behov gjennom reprogrammering av systemet. Funksjonelle utvidelser i et system utviklet på en ASIC innebærer at en ny krets må utvikles og produseres, noe som er en kostbar og lang prosess[19]. Den faste maskinvarearkitekturen i mikrokontrollere kan føre til at de må fysisk modifiseres eller erstattes ved behov for forbedring av systemet[11].

Den andre store grunnen til å velge en FPGA er ytelse. Instruksjonene gitt til en mikrokontroller utføres sekvensielt, som betyr at den konstant må bytte mellom oppgaver. På en FPGA har man muligheten til å designe og koble sammen moduler som alle kan arbeide parallelt. En stor del av de implementerte modulene vil være signalbehandlingsmoduler i form av digitale filtre. Disse består av en rekke multiplikatorer og adderere. På en FPGA kan operasjonene som inngår i signalbehandlingen parallelliseres. Dette fører til at resultater produseres raskere enn ved å kjøre tilsvarende programvare på mikrokontrollere eller spesialiserte digital signalprosessorer [19].

2.2.2 Valg av FPGA

Det enkleste er å velge et utviklingsbrett som inkluderer diverse tilkoblingsmuligheter, og som er klar til bruk rett fra boksen. FPGA-en må inneholde nok ressurser til at ønsket funksjonalitet kan implementeres, i tillegg til ressurser for fremtidig utvidelse. Valget falt til slutt på utviklingsbrettet *PYNQ-Z2* [1], som instituttet hadde erfaring med fra før.

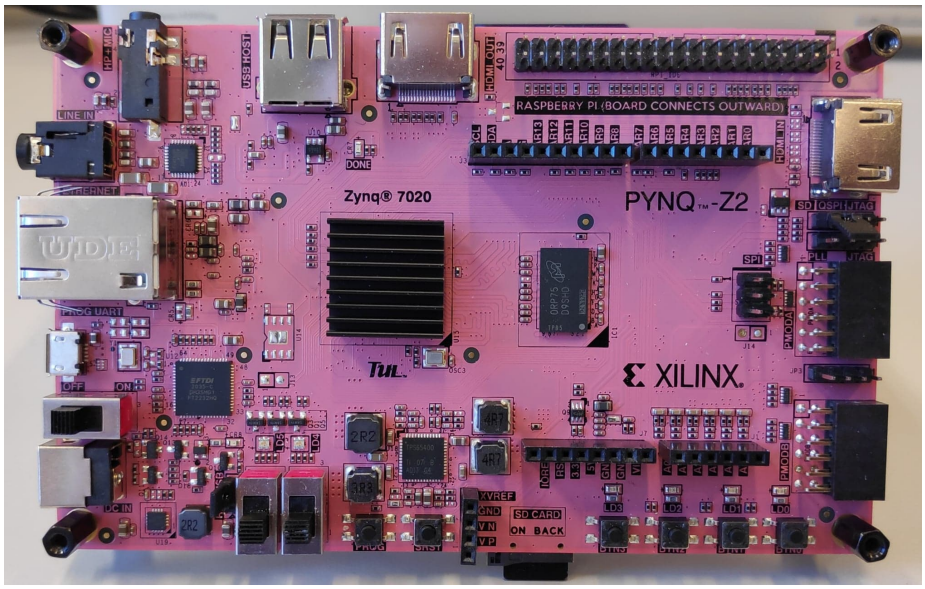
Utviklingsbrettet er bygget rundt et Xiling Zynq-7020 System-on-Chip (SoC), som inkluderer en ARM Cortex-A9 tokjerneprosessor og AD-omformere i tillegg til den programmerbare logikken. Brettet er laget for å støtte PYNQ,

som er et rammeverk for å interagere med den programmerbare logikken og grensesnittene ved hjelp av programmeringsspråket Python.

Tabell 1: Oversikt over tilgjengelige ressurser i FPGA-en.

LUT	53200
Flip-Flop	106400
BRAM	140*
DSP	220

*blokker på $36\ Kb$



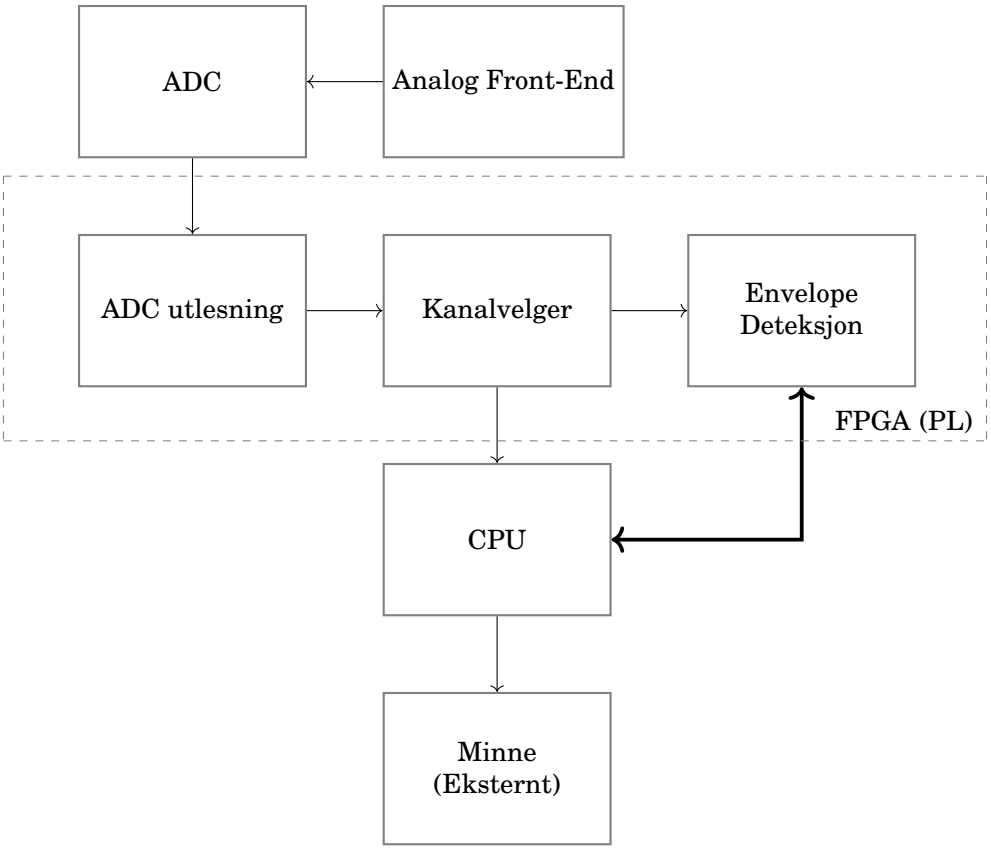
Figur 6: Bilde av PYNQ-Z2 utviklingsbrettet.

3 Metoder

3.1 Systemoversikt

Ekkoloddsystemet er satt sammen av en rekke moduler med forskjellige oppgaver. Figur 7 viser en grov oversikt over hoveddelene som inngår i systemet. Noen av disse består av flere undermoduler.

Modulene som inngår i systemet er en blanding av Xilinx IP-er og moduler skrevet i VHDL. Xilinx IP-er er ferdiglagde moduler, som er godt dokumenterte og ofte lette å ta i bruk. I tilfeller hvor en trenger høyere grad av kontroll og tilpasning er det likevel ofte greit å programmere egne moduler.



Figur 7: Blokkskjema som viser en oversikt over systemet.

3.1.1 Systemklokke

Ekkoloddsystemet bruker ikke eksterne enheter som krever egne klokker. Det betyr at vi kan styre systemet med den innebygde standardklokken på 100 MHz. AD-omformeren produserer data med en frekvens på 1 MHz. Dermed kan vi bruke 100 klokkesykluser på å behandle hver sample.

3.1.2 Systemspesifikasjoner

De veiledende spesifikasjonene for systemet er oppsummert i tabell 2. Disse inkluderer ønsket dybde, dynamikkområde, pingrate (hvor ofte ping sendes ut) og frekvenser for lydpulsen.

Dybdespesifikasjonene gir en indikasjon på nødvendig dynamikkområde , samt hvor mye data som må behandles og lagres for hvert ping. Et stort dyp vil bety større transmisjonstap, slik at forskjellen på ekkostyrken øker mellom et stort objekt på nært hold og lite objekt på lang avstand. I tillegg vil større dybder bety lengre lyttetid, og dermed større antall sampler som må behandles.

Pingraten setter en begrensning på systemet som helhet ved at det settes krav til hvor raskt data fra et ping må behandles og lagres før neste ping. En høy pingrate vil gjøre det mulig å spore passerende fisk og gjenstander ved at man får flere treff på dem. Dette er ønskelig ved for eksempel telling av laks i elv, for å skille forskjellige fisk og gjenstander.

Hvilke frekvenser som brukes i lydpulsen påvirker flere elementer i ekkoloddet, fra de analoge forkretsene til knekkfrekvenser i de digitale filtrene.

For å oppnå et rimelig og enkelt ekkolodd har vi valgt å bruke frekvenser mellom 50 og 200 kHz, en maksimal dybde på 60 meter, dynamikkområde på 100 dB og en maksimal pingrate på 10 ping/s. Dette forenkler elektronikken med tanke på signal-til-støyforhold samt mengde data vi trenger å håndtere for hvert ping.

Tabell 2: Oversikt over ønskede systemspesifikasjoner.

Dybde	60 m
Dynamikkområde	100 dB
Pingrate	10 Hz
Frekvenser	50 - 200 kHz

3.2 AD-Omformer

Analog-til-digital omformeren (AD-omformer, ADC) er første ledd i signalkjeden for prosesseringsdelen av ekkoloddsystemet presentert i denne oppgaven. Det analoge signalet, som blir mottatt gjennom front-end elektronikken, må bli konvertert til digitale verdier før vi kan drive videre signalbehandling digitalt. For å unngå tap av informasjon i løpet av konverteringen, stilles det krav til omformeren. Viktige krav er for eksempel dynamikk, signal-til-støy forhold (SNR, Signal-to-Noise Ratio) og samplinghastighet.

3.2.1 Krav til omformeren

Dynamikkområdet (dynamic range) til AD-omformeren bestemmer forholdet mellom det sterkeste og det svakeste signalet omformeren kan representere. For et ekkoloddsystem som både skal detektere små gjenstander over en stor distanse og større gjenstander over liten distanse, vil et stort dynamikkområde være viktig.

SNR er et mål på forholdet mellom signal og støy. Omformerens bidrag til total SNR kommer hovedsakelig fra kvantiseringsstøy. Dette oppstår ved at man deler inn et analogt signal i et gitt antall digitale verdier. Støyen vil være avviket mellom den nøyaktige analoge verdien og den avrundede digitale verdien. Forholdet mellom den minste spenningen omformeren kan representere (V_{LSB}) og kvantiseringsstøy ($V_{Q(rms)}$) er gitt i ligning 6 [9]. Siden V_{LSB} er avhengig av antall bit i omformeren som vist i ligning 7 [9] kan kvantiseringsstøyen, og dermed SNR, forbedres ved å øke antallet bit.

$$V_{Q(rms)} = \frac{V_{LSB}}{\sqrt{12}} \tag{6}$$

$$V_{LSB} = \frac{V_{ref}}{2^N} \tag{7}$$

Samplinghastigheten må være høy nok til at informasjonen vi ønsker ikke går tapt. Nyquist-Shannons samplingsteorem gir oss at samplingfrekvensen minst må være det dobbelte av den høyeste frekvensen i signalet.

3.2.2 Valg av omformer

FPGA-en valgt for dette prosjektet er utstyrt med en Xilinx XADC [3] hard IP, som inneholder to AD-omformere med 12-bit oppløsning og en samplehastighet på 1 mega-sampler per sekund (MSPS).

Samplehastigheten setter en øvre grense for signalfrekvensen som syste-

met kan motta. Denne frekvensen blir da 500 kHz i henhold til Nyquist, eller 100 kHz om man ønsker 10 ganger oversampling.

Med en maksimal inngangsspenning på 1 V, gir ligning 7 en V_{LSB} på 244 μV [3]. Dynamikkområdet til omformeren kan regnes ut med ligning 8.

$$\text{Dynamikkområde (dB)} = 20 \cdot log\left(\frac{\text{Fullskala utgangssignal}}{\text{Laveste utgangssignal}}\right) \tag{8}$$

Dynamikkområdet for en av de inkluderte omformerne blir med fullskala utgangssignal på 1 V og laveste utgangssignal på 244 μV omtrent 72 dB.

Som vi ser av beregningen ligger dynamikkområdet for en 12-bits omformer nesten 30 dB under ønsket dynamikkområde beskrevet i seksjon 3.1.2. Alternativer for å omgå dette problemet er blitt lagt frem i tidligere arbeid. Disse omfatter bruk av analog tidsvariabel forsterker før AD-omformeren, bruk av ADC med flere bit, eller inndeling i flere kanaler med ulik forsterkning [8].

Vi ønsker minst mulig analog elektronikk og vil derfor helst unngå en analog tidsvariabel forsterker. Utviklingsbrettet er utstyrt med to Peripheral Module (PMOD) tilkoblinger som gjør det mulig å bruke en ekstern ADC med 18-bit. Dette er tilstrekkelig til å nå den ønskede dynamikken, men slike omformere er relativt kostbare. Siden utviklingsbrettet allerede har to innebygde omformere, er det naturlig å velge å bruke disse. Inngangen deles opp i to kanaler, med en omformer i hver kanal. Vi lar en kanal med høy analog forsterkning behandle de svake ekkoene, mens en annen kanal med lavere forsterkning tar seg av større ekko. For å oppnå dynamikkområde på 100 dB må forskjellen i forsterkningen mellom de to kanalene ligge på ca. 30 dB.

Ved å bruke de inkluderte omformerene sparer man også plass og strømforbruk, som er ideelt for et system som skal være bærbart.

3.2.3 Beskyttelseskrets

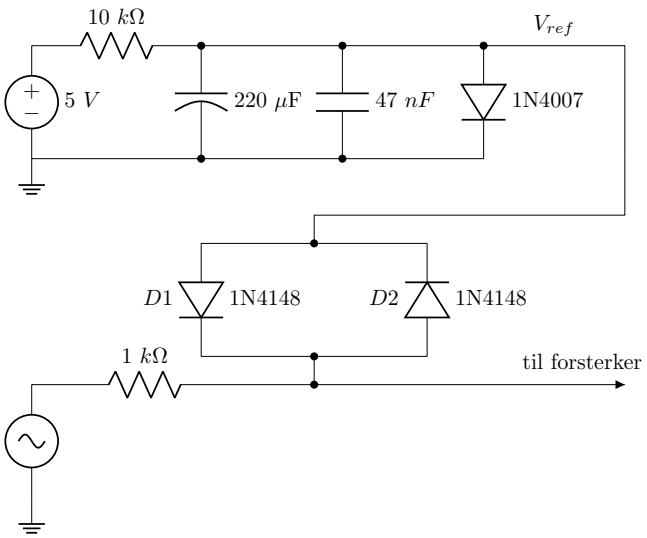
Brukerveiledningen [3] for XADC oppgir at inngangssignalet skal ligge i området $0-1$ V. Inngangsverdier utenfor disse grensene vil bli klippet. Veiledningen oppgir også en maksimal inngangsspenning på 1.9 V og en minimum innganssspenning på -0.1 V. Beveger man seg utenfor disse grensene risikerer man permanent skade på omformeren. En krets for å begrense inngangsspenningen blir dermed nødvendig for å hindre at utstyret blir ødelagt. For å kunne utnytte mest mulig av omformerens dynamikkområ-

de, burde kretsen sørge for en overgang til klipping uten stor påvirkning av signalet innenfor grensene. Utviklingsbrettet vi bruker har allerede plassert en spenningsdeler mellom inngangspinnene og AD-omformeren som skalerer ned fra 3.3 V [22], og gjør at vi ideelt skal ligge i området $0 - 3.3$ V.

Problemet med begrensning kan løses på flere måter. En mulighet er å sette sammen en krets med dioder for å klippe signalet. Fordelen med en slik løsning er at den er enkel og billig. En annen mulighet er å bruke spesialiserte komponenter med innebygd spenningsbegrensning, som for eksempel operasjonsforsterkeren OPA698 fra Texas Instruments [26]. Fordelen med denne løsningen er at slike komponenter er enkle å ta i bruk og er mer effektive ettersom de er designet spesifikt for formålet. I tillegg er operasjonsforsterkere allerede en nødvendig del av forkretsen for å forsterke inngangssignalet, og vi kan dermed slå to fluer i en smekk. Begge løsningene nevnt over ble simulert og testet ut.

Diodekrets

Kretsen bygget for å begrense spenningen ved hjelp av dioder er vist i figur 8. Den kan deles inn i to deler, hvor den øverste delen er laget for å gi en referansespenning på ca. 0.6 V, og den nederste delen bestående av to signaldioder skal begrense inngangssignalet til DC ± 0.6 V. Et inngangssignal som svinger rundt 0.5 V skal dermed bli begrenset til området -0.1 V til 1.1 V, som stemmer godt overens med omformerens spesifikasjoner.



Figur 8: Krets bygget med dioder for å begrense inngangsspenningen.

Denne kretsen ble designet før vi oppdaget at det var plassert en spenningsdeler foran inngangen til omformeren på utviklingsbrettet. Dette gjør at vi ikke får utnyttet hele omformerens spenningsområde med konfigurasjonen i figur 8. For å løse dette kan signalet forsterkes etter begrensningen, eller så kan spenningsdeleren på brettet fjernes. Ideelt sett vil fjerning av spenningsdeleren vært best for å unngå mye unødvendig forsterkning og redusering av signalet.

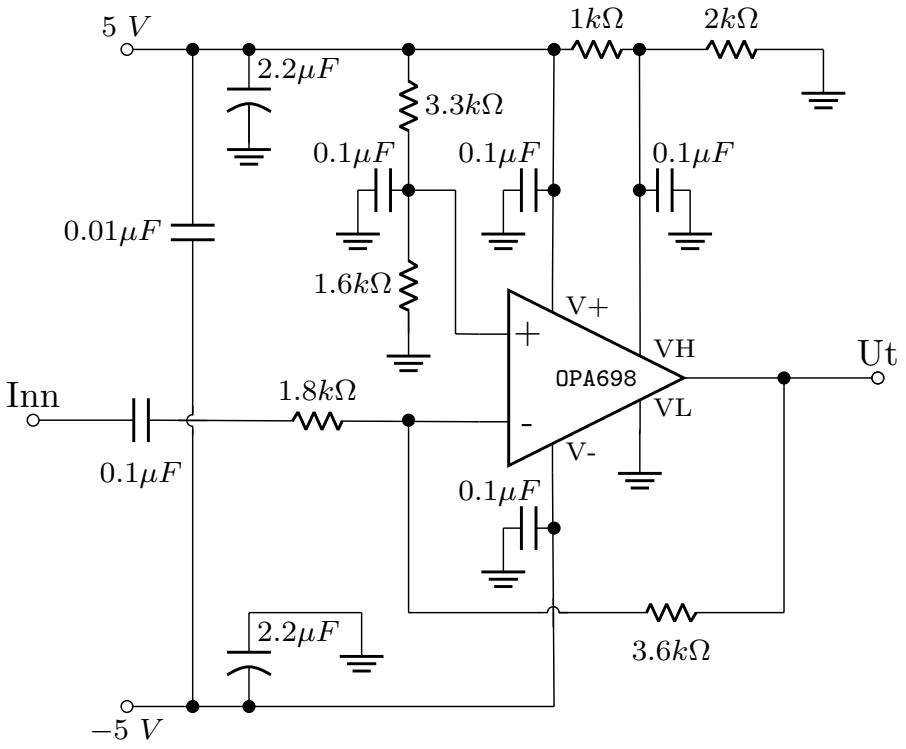
Krets med operasjonsforsterker

Operasjonsforsterkeren OPA698 er utstyrt med to ekstra pinner for referansespenninger som bestemmer øvre og nedre begrensning for utgangssignalet. En krets tilpasset vårt system er vist i figur 9, og er designet i henhold til eksempelkretsene i databladet [26]. Den øvre begrensningspinnen er koblet til en referansespenning på ca. 3.3 V, som kommer fra en spenningsdeling fra forsyningsspenningen på 5 V. Motstandsverdiene er kalkulert ved hjelp av ligning 9 som, med en inngangsspenning på 5 V og ønsket utgangsspenning på litt i overkant av 3.3 V, resulterte i motstandene $\mathrm{R}_1 = 1$ kΩ og $\mathrm{R}_2 = 2$ kΩ. Den nedre begrensningspinnen er koblet til jord. Dermed vil signalet inn til omformeren være begrenset til området mellom 0 V og 3.3 V, som ønsket.

$$V_{ut} = V_{in}\frac{R_2}{R_1 + R_2} \tag{9}$$

$$V_{ut} = V_{in}\frac{R_f}{R_i} \tag{10}$$

En kondensator er koblet på inngangen for å fjerne inngangssignalets offset. I tillegg er avkoblingskondensatorer koblet til jord fra alle forsyninger og referanser for å minimere støy. En tilbakekoblet motstand R_f på 3.6 kΩ sammen med inngangsmotstanden R_i på 1.8 $k\Omega$ gir en forsterkning på 2 ut ifra ligning 10. En referansespenning er koblet på den positive inngangen slik at utgangssignalet svinger rundt ønsket DC-offset på omtrent $\mathrm{V}_{maks}/2 = 1.65$ V. Referansespenningen er spenningsdelt ned fra forsyningsspenningen ved hjelp av motstandene $\mathrm{R}_1 = 3.3$ kΩ og $\mathrm{R}_2 = 1.6$ kΩ.



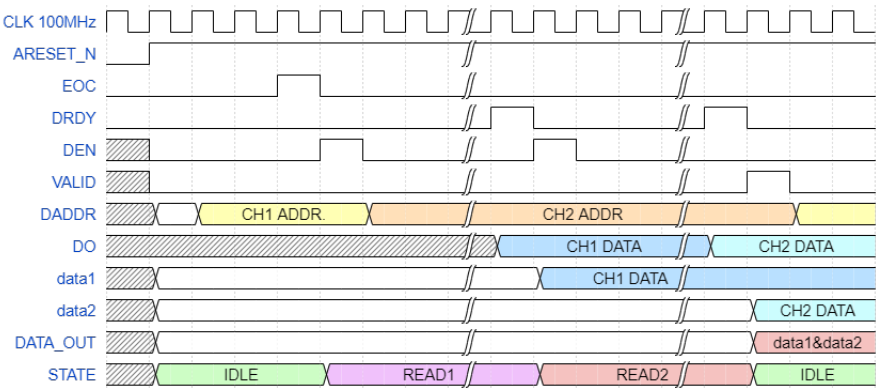
Figur 9: Krets bygget rundt operasjonsforsterkeren OPA698 for å begrense inngangsspenningen.

3.3 Utlesing fra AD-omformer

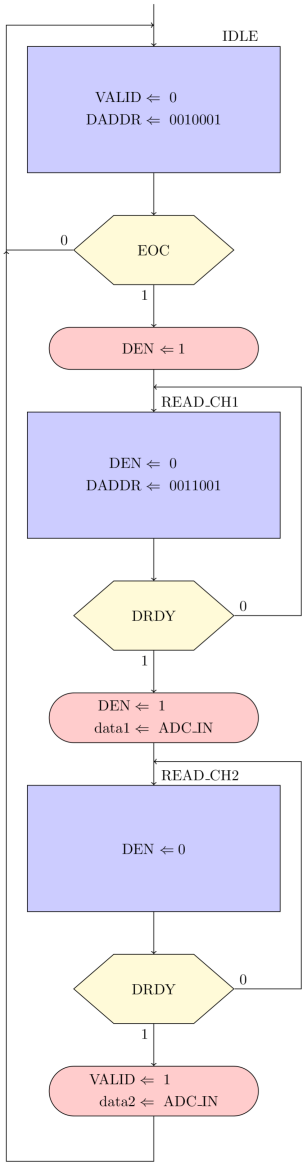
XADC IP-en tilbyr flere muligheter for utlesing. Disse inkluderer Advanced eXtensible Interface (AXI) grensesnittene AXI4-Stream og AXI4-Lite, i tillegg til Dynamic Reconfiguration Port (DRP). AXI4-Stream er unidireksjonell og brukes for rask strømming av data [6]. Ettersom den bare sender data en vei trengs en av de andre grensensittene i tillegg for sende kontrollsignaler til omformeren. Å endre innstillinger under kjøring kan være aktuelt for eksempel ved implementering av søvnmodus i systemet for å spare strøm. AXI4-Lite kan brukes til å lese ut data, men egner seg best for lavhastighets overføringer som skriving til kontroll- og statusregistre [6]. Ettersom DRP var godt dokumentert i [3] og kan brukes til både utlesing av data og kontroll av omformeren, ble denne valgt for prosjektet.

Utlesningsmodulen er designet som en tilstandsmaskin. Den venter på at XADC IP-en skal signalisere at den har en digital verdi klar ved hjelp av end of conversion (EOC)-signalet. Når tilstandsmaskinen mottar EOC er allerede adressen til registeret som inneholder data for første kanal satt

ut på adresseporten (DADDR), og enable (DEN)-signalet settes høyt slik at adressen blir lest inn på neste klokkesyklus. Når data er klar på data ut (DO)-porten til XADC, signaliserer den det med et ready (DRDY) signal. Tilstandsmaskinen leser da ut dataen, og setter ut en ny adresse sammen med DEN. Når verdier fra begge kanaler er lest ut, signaliserer tilstandsmaskinen til de etterfølgende modulene at den har gyldige verdier klare for behandling gjennom et VALID-signal. Omformeren produserer to 16-bit verdier, hvor de 12 mest signifikante bitene (MSB) er dataen [3].



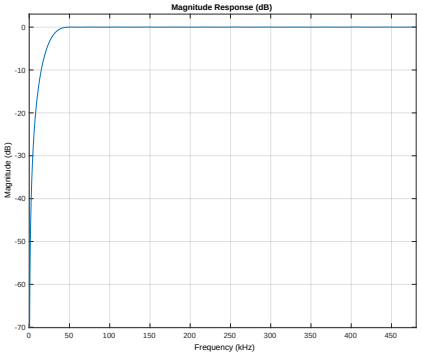
Figur 10: Timing for utlesningsmodul. Signaler med små bokstaver er interne, de med store bokstaver er inngangs og utgangssignaler.



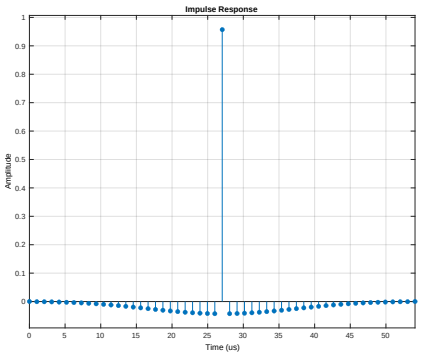
Figur 11: ASM diagram for utlesningsmodulen for AD-omformeren.

3.4 Høypassfilter

Etterfølgende algoritmer er avhengig av at signalet svinger rundt 0. Etter konvertering sendes derfor dataen gjennom høypassfilter for å fjerne DC-offset og lavfrekvent støy. Høypassfilteret er et FIR-filter med oppbygning nærmere forklart i seksjon 3.6.2. Filterkoeffisienter er laget i MATLAB. Frekvens- og impulsrespons er vist i figur 12 og 13.



Figur 12: Frekvensresponsen til høypassfilteret.



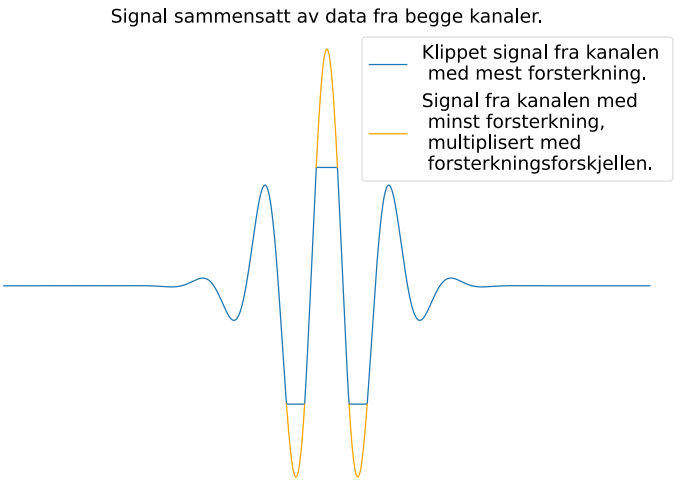
Figur 13: Impulsresponsen til høypassfilteret.

3.5 Kanalvelger

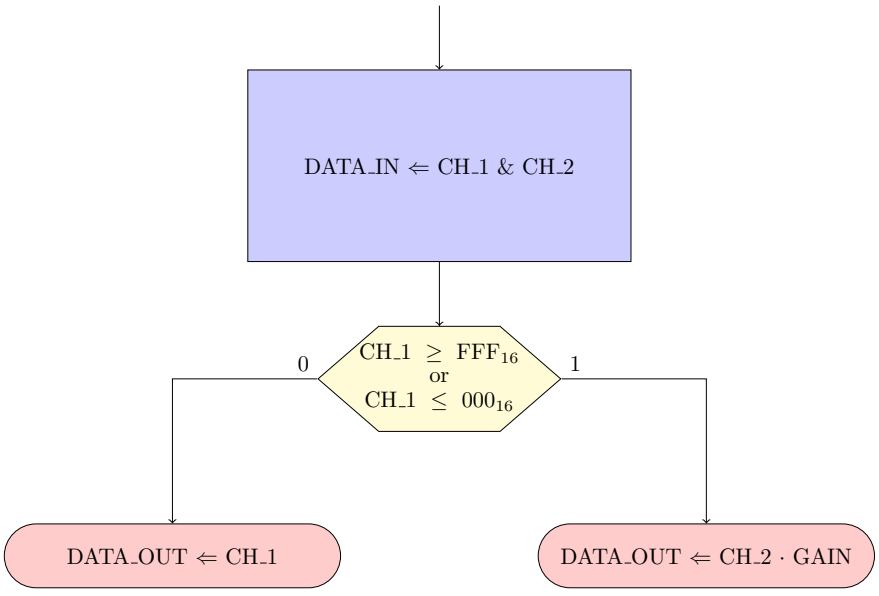
Siden systemet tar nytte av to kanaler for å øke det dynamiske området, trengs en modul for å detektere når kanalen med høyest forsterkning er gått i metning. Dette vil være tilfelle for ekko fra objekter nær eller større objekter lengre vekke fra senderen. Når kanalen med høyest forsterkning går i metning, brukes data fra den andre kanalen.

Måten dette blir gjort på er å bestemme om kanalen med størst forsterkning er i metning. I et slikt tilfelle brukes verdier fra den andre kanalen multiplisert med en faktor tilsvarende forskjellen i forsterkning mellom kanalene. Dette er illustrert i figur 14 og forklart i flytskjemaet i figur 15.

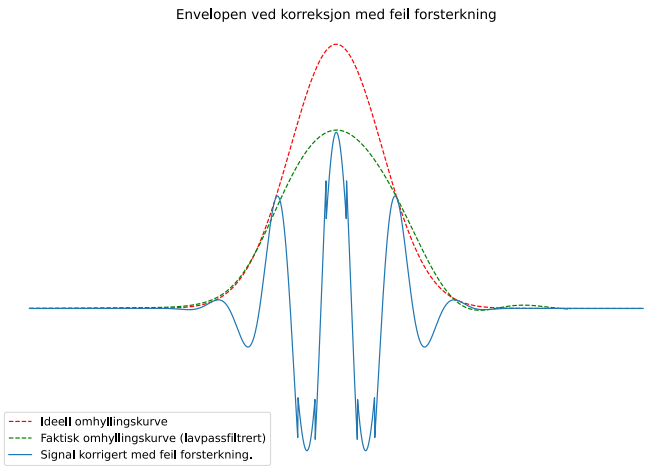
Det er viktig å være klar over at et ekko kan ende opp med å være sammensatt av sampler fra begge kanaler. Dersom samplene fra kanalen med minst forsterkning ikke blir multiplisert med korrekt kalibrert verdi (figur 16), eller det er en faseforskyvning mellom de to kanalene (figur 17), kan dette føre til feil i utlesningen av omhyllingskurven.



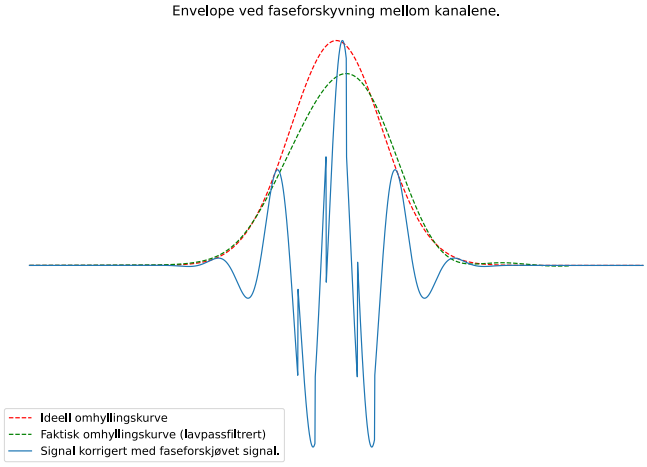
Figur 14: Visualisering av kanalvelgerens funksjon.



Figur 15: Dataflytdiagram over kanalvelgeren.



Figur 16: Omhyllingskurven til et signal satt sammen av signaler fra begge kanaler sammenlignet med den ideelle omhyllingskurven. Kanalen med minst forsterkning er her multiplisert med feil verdi. Omhyllingskurven er filtrert med et lavpass butterworth filter.



Figur 17: Omhyllingskurven til et signal satt sammen av signaler fra begge kanaler sammenlignet med den ideelle omhyllingskurven. Her er singalet på inngangen til den ene kanalen faseforskjøvet i forhold til den andre. Omhyllingskurven er filtrert med et lavpass butterworth filter.

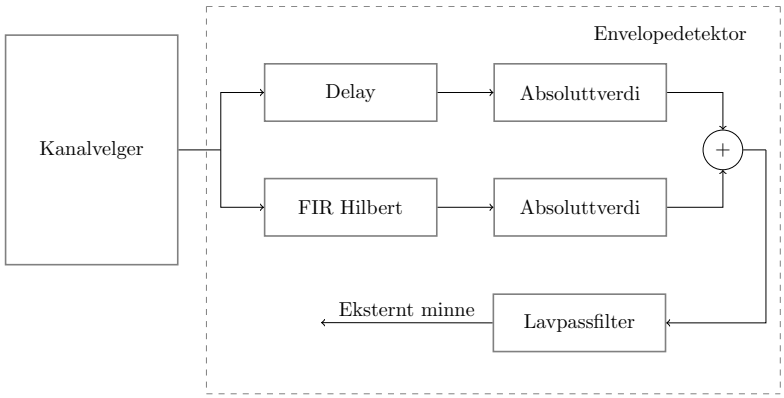
3.6 Uthenting av omhyllingskurven

Det finnes en rekke metoder for å hente ut omhyllingskurven ved hjelp av digital signalbehandling [16]. Disse varierer i kompleksitet fra enklere implementasjoner med likeretting og lavpassfiltrering, til mer avanserte detektorer som involverer Hilbert-transformasjoner og kvadratrotmetoder. De sistnevnte krever ofte mer ressurser, men kan også gi bedre resultater. Andre metoder som ikke krever like mye ressurser er testet i tidligere arbeid [23][25], og går ut på å sammenligne sampler å beholde toppverdiene.

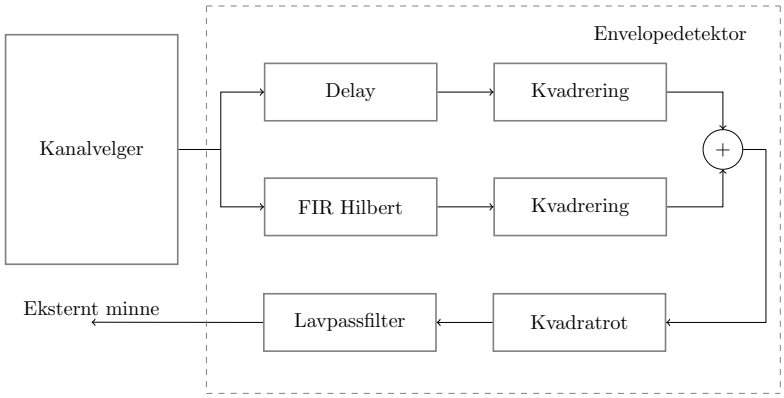
For dette prosjektet ble det valgt å bruke metoder mer lik den presentert i seksjon 2.1.3, og metodene presentert i [16] med høyest signal-til-støyforhold ble valgt for implementasjon.

- Metode 1 går ut på å finne det komplekse signalet ved hjelp av en Hilbert transformasjon, ta absoluttverdi av real og imaginærdel, og addere disse. Resultatet vil bli sendt gjennom et lavpassfilter for å tydeliggjøre omhyllingskurven.
- Metode 2 begynner også med å finne det komplekse signalet, men fortsetter med å kvadrere real og imaginærdel før addisjon. Man tar så kvadratroten av resultatet, og sender det gjennom et lavpassfilter.

Metode 2 minner mest om den eksakte metoden vist i ligning 5, men krever en ekstra modul for utregning av magnitude. Begge metodene ble testet for å kunne sammenligne og vurdere hvilken som var best. Testene er beskrevet i seksjon 3.12.4, og resultatene er presentert i seksjon 4.4. Testingen viste at metode 2 produserte de beste resultatene.



Figur 18: Oversikt over operasjonene som inngår i metode 1 for uthenting av omhyllingskurven.



Figur 19: Oversikt over operasjonene som inngår i metode 2 for uthenting av omhyllingskurven.

3.6.1 Hilbert filter

Hilbert transformasjonen kan gjøres på diskrete signaler i tidsdomenet ved at en konvolverer signalet med et sett koeffisienter som vist i ligning 11 [17].

$$x_i(n) = \sum_{k=-\infty}^{\infty} h(k)x_r(n-k) \tag{11}$$

Dette ser ut som et FIR filter (nærmere forklart i seksjon 3.6.2) med et uendelig antall tapper. Dermed brukes en FIR-filter struktur for Hilbert transformasjonen.

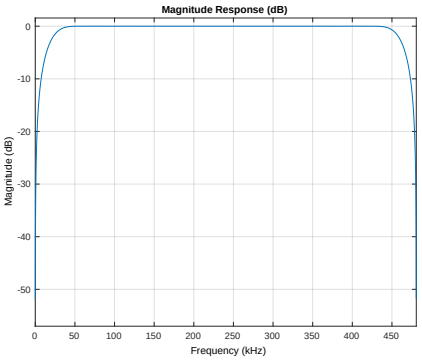
For å få rett forsinkning mellom samplene sendes sampler fra kanalvelgeren inn i et skiftregister, illustrert som bokser med piler i figur 23. Pilene indikerer at alle tidligere sampler forskyves når en ny sample er tilgjengelig. Hver av samplene i skiftregisteret blir multiplisert med en koeffisient. Filterkoeffisientene ble kalkulert som i ligning 12 [17]. Produktene blir summert til et resultat på utgangen til filteret.

$$h(n) = \frac{2sin^2(\pi n/2)}{\pi n} \tag{12}$$

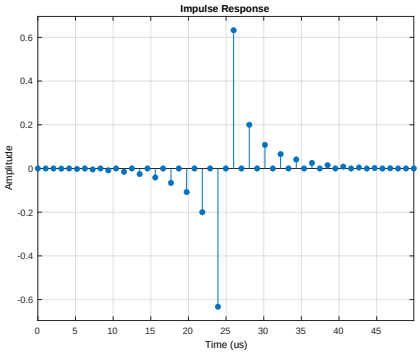
Det resulterende filteret hadde rippler i passbåndet. Disse kan imidlertid dempes ved å legge på et vindu [17]. Et Blackman-vindu ble valgt for dette.

Ettersom det ikke var realistisk å addere alle produktene i en operasjon, ble denne prosessen delt opp. Når en ny verdi kommer inn i skiftregisteret, produseres nye produkter som adderes parvis. Summene blir lagret i midlertidige registre, som igjen adderes parvis. Dette legger til en liten for-

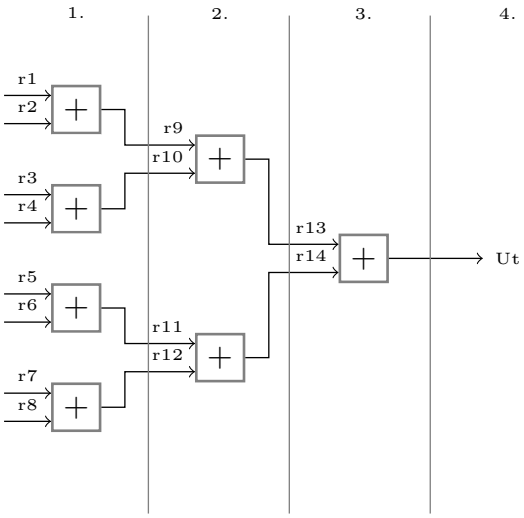
sinkelse som kommer i tillegg til filterets grupperforsinkelse. Prosessen er beskrevet visuelt i figur 22.



Figur 20: Frekvensresponsen til Hilbert filteret.



Figur 21: Impulsresponsen til Hilbert filteret.



Figur 22: Illustrasjon av oppdelingen av addisjonsoperasjoner. Tallene øverst indikerer klokkesykluser. Systemets filtre består av mer en 8 tapper, og bruker mer enn 4 sykluser.

Resultatet på utgangen av filteret er den imaginære delen av det komplekse signalet $x_c(t)$ beskrevet i ligning 5. For å få det fullstendige komplekse signalet må den reelle delen legges til. Det reelle signalet er i dette tilfellet en forsinket versjon av originalsignalet med en forsinkelse lik $z^{-N/2+1}$, hvor N er filterets lengde (gjelder for filter med oddetallslengde). Denne verdien kan hentes rett ut fra midterste plass i skiftregisteret [17].

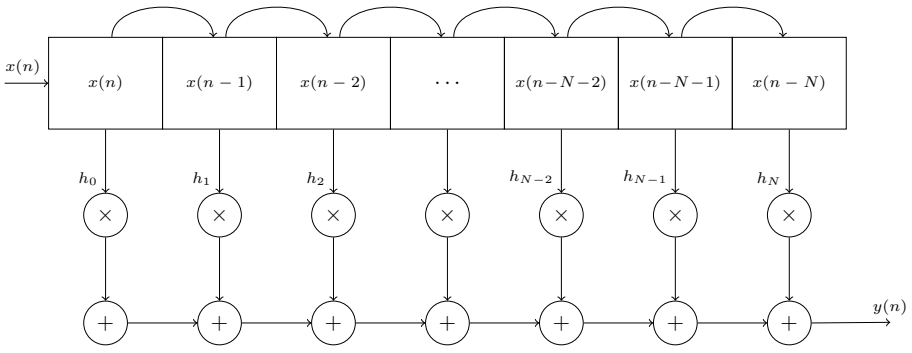
3.6.2 Lavpassfilter

FIR eller IIR?

Når det kommer til design av digitale lavpassfilter, står valget mellom å implementere et Infinite Impulse Response (IIR)-filter, eller et Finite Impulse Response (FIR)-filter.

Et FIR-filter har, som navnet impliserer, en endelig impulsrespons. Det vil si at et endelig antall sampler på inngangen gir et endelig resultat på utgangen. Dette kommer av at resultatet på utgangen bare er avhengig av tidligere sampler på inngangen. Ettersom filteret ikke har tilbakekobling (er avhengig av tidligere resultater), vil det alltid være stabilt. Alle frekvenskomponentene i et FIR-filter er forsinket like mye. Det betyr at filteret har en lineær faserespons [17]. Matematisk kan filtreringen med et FIR filter uttrykkes som i ligning 13 [17] hvor $h(k)$ er filterkoeffisientene, $x(n-k)$ er inngangssignalet og $y(n)$ er utgangssignalet. En visuell fremstilling er presentert i figur 23.

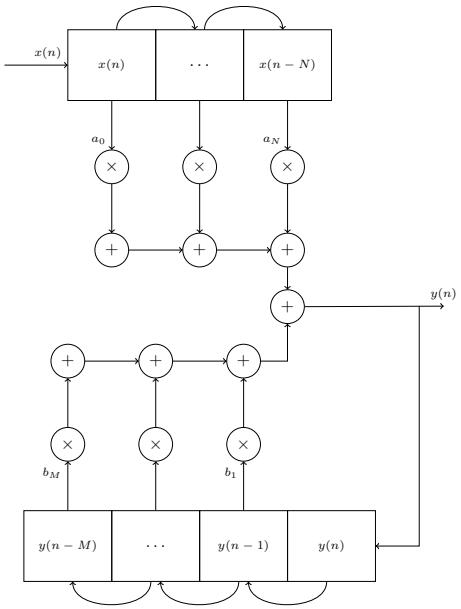
$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \tag{13}$$



Figur 23: Eksempel på oppbygning av digitalt FIR-filter.

Et IIR-filter kan ha en uendelig impulsrespons. Dette kommer av at filteret er avhengige av tidligere inngangsverdier i tillegg til tidligere resultater på utgangen. En slik tilbakekobling kan føre til at filteret blir ustabilt og oscillerer. IIR-filter har heller ikke lineær faserespons. Fordelen med et IIR-filter over et FIR-filter er at det kreves mindre utregninger for å produsere samme resultat [17]. Matematisk kan filtreringen med et IIR-filter uttrykkes som i ligning 14 [17]. En visuell fremstilling er presentert i figur 24.

$$y(n) = \sum_{i=0}^{N-1} a(i)x(n-i) - \sum_{j=1}^{M-1} b(j)y(m-j) \tag{14}$$



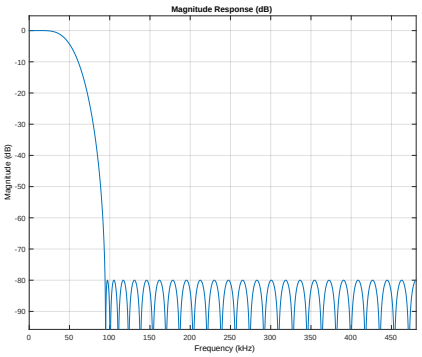
Figur 24: Eksempel på oppbygning av digitalt IIR-filter.

Det er fordeler og ulemper med begge implementasjonene. Stabilitet er en klar fordel. Lineær faserespons er ikke så viktig i vårt system, ettersom vi sender lydpulser med konstant frekvens. For lydpulser med en mer avansert utforming kan dette bli en viktig faktor. Når det kommer til mengden utregninger kan dette ha mye å si for tidsbruk og den kan være begrenset av mengden ressurser tilgjengelig.

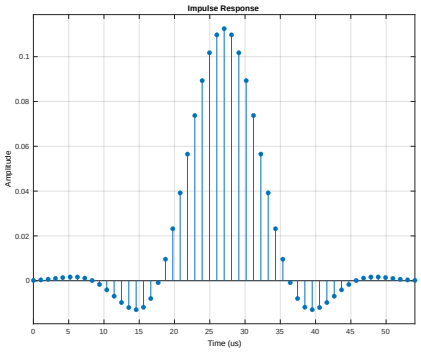
På grunn av stabilitet og en enklere struktur [17] ble et FIR filter valgt for implementasjon. Brikken som er brukt for utvikling av systemet har mange tilgjengelige signalbehandlingsblokker (220 DSP-slices [1]), så ressurser er ikke et problem. Ettersom en FIR filter struktur også ble brukt for Hilbert transformasjonen, kan mye av koden brukes igjen.

Implementasjon

Lavpassfilteret har veldig lik oppbygning som Hilbert filteret, med andre koeffisienter og noen optimaliseringer for å minke antallet multiplikasjoner. Koeffisientene ble produsert med funskjoner i MATLAB. Frekvensrespons og impulsrespons kan bli sett i figur 25 og 26.

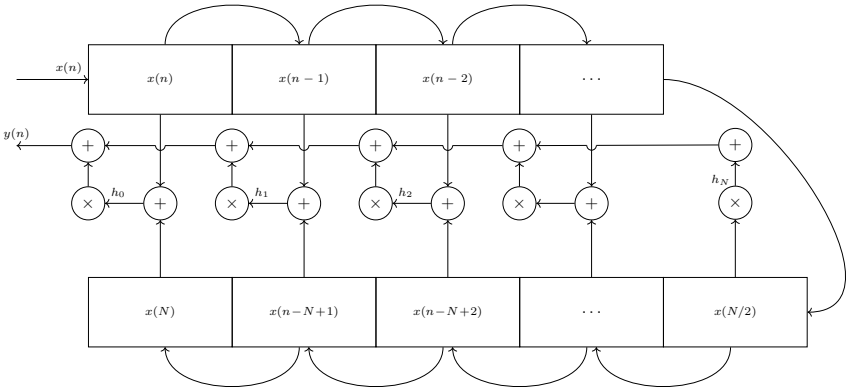


Figur 25: Frekvensresponsen til lavpassfilteret.



Figur 26: Impulsresponsen til lavpassfilteret.

Lavpassfilteret er implementert som et type 1 filter. Det vil si at antallet tapper på filteret er et oddetall, og koeffisientene er symmetriske om midten. Dette gjør at vi kan halvere antallet multiplikasjoner som trengs ved å addere sampler som skal multipliseres med samme koeffisient [17], som vist i figur 27. Samme optimalisering er gjort for høypassfilteret.



Figur 27: Digitalt FIR-filter, optimalisert med ferre multiplikasjoner.

3.6.3 Magnitudemodul

Magnitudemodulen tar inn realdel og imaginærdel av det komplekse signalet og regner ut magnituden. Delene kvadreres, adderes og sendes til slutt gjennom en iterativ prosess som regner ut kvadratroten. Den viktigeste delen av modulen er kvadratrotalgoritmen, som baserer seg på den modifiserte ikke-gjenopprettende algoritmen presentert i [24].

Tallet man ønsker å ta roten av deles opp i par. Første par brukes til å regne ut rotens MSB, så regnes rest og neste par legges til for å regne ut neste siffer i roten, som vist i figur 28.

1. Deler tallet man ønsker å ta roten av opp i par. Tallet 11010111 representert i titallsystemet er 215, som har roten ≈ 14.66.
2. Finner høyeste verdi for a_1 som tilfredsstiller $X_1 \geq a_1^2$, hvor X_1 er første par. $11 \geq a_1^2$ gir oss at den høyeste verdien for a_1 blir 1.
3. Regner ut rest og legger til neste par, slik at vi får X_2.
4. Finner høyeste verdi for a_2 som tilfredsstiller $X_2 \geq [a_1 \mid\mid 0 \mid\mid a_2] \Rightarrow 1001 \geq [10 \mid\mid a_2]$. Prøver vi $a_2 = 1$ får vi 101 på høyre side, som tilfredsstiller ulikheten.
5. Regner ut rest og legger til neste par, slik at vi får X_3.
6. Finner høyeste verdi for a_3 som tilfredsstiller $X_3 \geq [a_1 \mid\mid a_2 \mid\mid 0 \mid\mid a_3] \Rightarrow 10001 \geq [110 \mid\mid a_3]$. Prøver vi $a_3 = 1$ får vi 1101 på høyre side, som tilfredsstiller ulikheten.
7. Regner ut rest og legger til neste par, slik at vi får X_4.
8. Finner høyeste verdi for a_4 som tilfredsstiller $X_4 \geq [a_1 \mid\mid a_2 \mid\mid a_3 \mid\mid 0 \mid\mid a_4] \Rightarrow 10011 \geq [1110 \mid\mid a_4]$. Prøver vi $a_4 = 1$ får vi 11101 på høyre side, som ikke tilfredsstiller ulikheten. Dermed må vi velge $a_4 = 0$.
9. Om en ønsker svar som er heltall stopper prosessen her, og det som er igjen er rest. Svaret på kvadratroten blir dermed $a_1 + a_2 + a_3 + a_4 = 1110$ som representert i titallsystemet blir 14.

		1 1 1 0
1.		$\sqrt{11 \mid 01 \mid 01 \mid 11}$
2.	−	1
3.		10 01
4.	−	1 01
5.		1 00 01
6.	−	11 01
7.		1 00 11
8.	−	0
9.		1 00 11

Figur 28: Kvadratrotalgoritmen steg for steg. Uttrykket $a_1 \mid\mid a_2$ betyr at vi plasserer sifferet a_2 til høyre for a_1.

Ulikhetene i steg 2, 4, 6, og 8 (ref. fig. 28) kan vi uttrykke mer generelt som i ulikhet 15. Her er X_n på venstre side det som gjenstår av tallet man ønsker å ta roten av for hvert steg, P_{n-1} på høyre side er den partielle roten man har funnet så langt og a_n er sifferet man ønsker å finne. For hver iterasjon i prosessen oppdateres X_n og P_{n-1} slik at man kan finne en ny a_n.

$$X_n \geq [P_{n-1} \mid\mid 0 \mid\mid a_n] \tag{15}$$

Implementasjon

Magnitudemodulen er skrevet som en tilstandsmaskin. I første tilstand tar modulen inn real- og imaginærdel av det komplekse signalet når disse er tilgjengelige (signalisert av VALID-signalet fra utlesningsmodulen for AD-omformeren). Disse blir kvadrert hver for seg, før modulen beveger seg videre til neste tilstand.

I andre tilstand adderes de kvadrerte delene. I tredje tilstand settes første ulikhet opp, som beskrevet i ulikhet 15. Algoritmen er implementert slik at det alltid testes for $a_n = 1$. Dermed settes MSB a_1 til 1 slik at høyre side av ulikheten er 01, og venstre side av ulikheten er første par.

I fjerde tilstand subtraheres høyre og venstre side av ulikheten. Svaret vil være oppgitt som toerkomplement, slik at MSB kan sjekkes for å finne ut om svaret er negativt eller positivt.

Er svaret negativt betyr dette at ulikheten ikke er korrekt, og a_1 skulle vært 0. Partiellroten P blir dermed 0, og høyre side av ulikhet 15 oppdateres til å bli $[P_1 \mid\mid 0 \mid\mid a_2] = 001$. Venstre side oppdateres ved at forrige og neste par blir konkatenert.

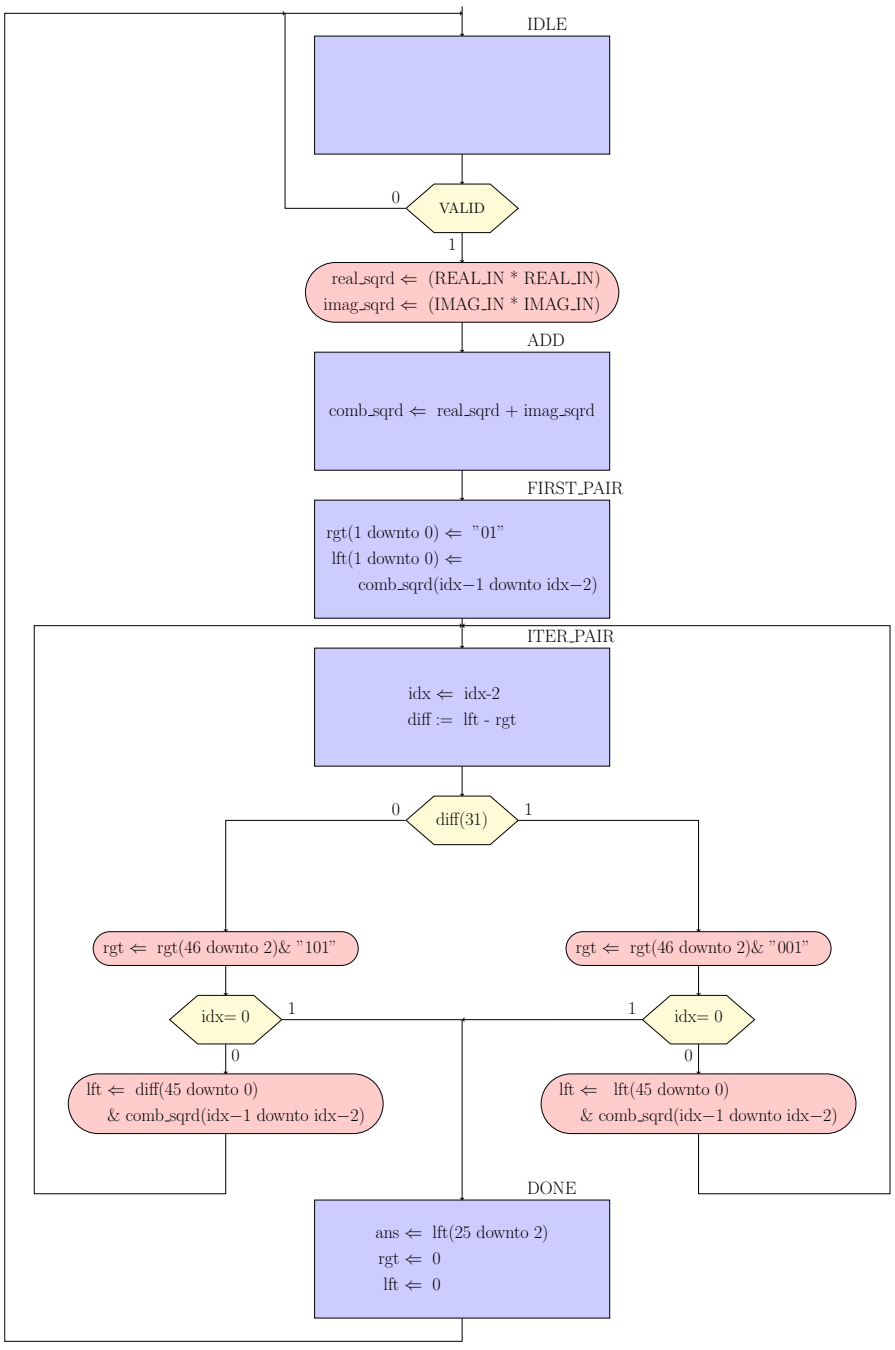
Er svaret positivt beholder vi $a_1 = 1$. Partiellroten P blir dermed 1, og høyre side av ulikhet 15 oppdateres til å bli $[P_1 \mid\mid 0 \mid\mid a_2] = 101$. Venstre side oppdateres ved at differansen fra fjerde tilstand og neste par blir konkatenert.

Etter at vi har iterert oss gjennom alle parene er prosessen ferdig.

3.7 LISTEN signal

Systemet er avhengig av et kontrollsignal, som har fått navnet LISTEN. Dette skal fortelle når systemet skal skrive til fil eller ikke. Bredden på pulsen bestemmer hvor lenge systemet skal lytte og dermed også hvor dypt en ser i vannet. En LISTEN-puls med bredde på 40 ms vil med en hastighet i vann på 1500 m/s bety at vi kan se objekter ned til en dybde på 30 m (beregnet med ligning 1).

En modul gitt navnet Ping FSM er brukt for å synkronisere LISTEN signalet med systemklokken. Denne modulen styres også av en bryter, som kan brukes til å sette LISTEN-signalet konstant lavt, og dermed midlertidig stanse systemet.



Figur 29: ASM diagram over magnitudemodulen.

3.8 Digital signalbehandling på FPGA

3.8.1 DSP48E1

Zynq-7020 brikken, som utviklingsbrettet er bygget rundt, er utstyrt med 220 DSP48E1 [2] blokker. Disse blokkene er dedikert til digital signalbehandling. Blokkene inkluderer blant annet pre-addere og 25x18-bit toerkomplement multiplikatorer (begrenset til 24x18-bit ved bruk av pre-addere). Større multiplikasjoner, som for eksempel kvadreringen i magnitudemodulen, kan utføres ved å kaskadekoble flere DSP-blokker. Databladet oppgir også at minst tre pipeline registre er nødvendig for å lagre midlertidige kalkulasjoner, slik at DSP48E1 blokken kan sette i gang med nye beregninger oftere og dermed operere ved full hastighet.

For å best mulig benytte hver enkelt blokk ble 18-bit brukt til å representere alle filterkoeffisientene. Data fra omformer ble utvidet fra 12- til 24-bit for å gjøre plass til skalering i kanalvelgeren.

3.8.2 Aritmetikk og bitbredder

Ved aritmetiske operasjoner må nok plass settes av til resultatet. En multiplikasjon mellom tall med lengde M og N kan resultere i et tall med lengde $M+N$. Ved summering av tall må registeret for å holde svaret ha en lengde på $b+log_2(G)$, hvor b er størrelsen på tallene som adderes og G er mengden tall som skal summeres [17].

Signalverdier og filterkoeffisienter i systemet representeres som toerkomplement, som tillater både positive og negative verdier.

Alle filterkoeffisientene ble skalert med 2^{17} og omgjort til 18-bits toerkomplement. Dermed må produktene fra multiplikasjon med disse koeffisientene deles på en faktor 2^{17} for å få korrekt skalert verdi. Multiplikasjonen mellom tall med størrelse på 24- og 18-bit produserer et tall med størrelse på $24+18 = 42$-bit. Skaleringen ble implementert som en trunkering av produktet ved å fjerne de nederste 17-bitene. Dette er det samme som et 17-bits venstreskift, eller en divisjon på 2^{17}. Produktet av en multiplikasjon mellom to toerkomplementer får et ekstra fortegnsbit som kan ignoreres [17]. Dermed står vi igjen med en 24-bit signalverdi. Etterfølgende Xilinx IP-er bruker hele ordlengder (16-bit, 32-bit, 64-bit osv.) slik at signalverdiene på 24-bit ble utvidet til 32-bit for å tilfredsstille disse kravene.

3.9 Lagring av data

Systemet produserer en 32-bit verdi hvert mikrosekund. Dette betyr at i løpet av ett sekund vil den produsere 4 megabyte med data som skal overføres til en ekstern lagringsenhet. Typiske eksterne lagringsenheter vil være minnekort, minnepenner eller eksterne harddisker. Førstnevnte finner man oftest i form av et Secure Digital (SD)-kort som man kommuniserer med via en Serial Peripheral Interface (SPI) eller en egen SD bus. Minnepenner og eksterne harddisker kommuniseres ofte med via Universal Serial Bus (USB). For å lagre data eksternt blir det dermed nødvendig å implementere drivere for en av disse protokollene i systemet.

3.9.1 PYNQ

PYNQ er rammeverket som gjør det mulig for en utvikler å interagere med den programmerbare logikken gjennom programmeringsspråket Python [21][20]. Det inneholder ferdiglagde og godt dokumenterte klasser og funksjoner som er designet for kommunikasjon Xilinx IP-er, og for kommunikasjon med eksterne kretser og moduler via grensesnittene som utviklingsbrettet tilbyr. I tillegg kan egne klasser, funksjoner og drivere skrives for selvdesignede moduler. Programmeringen foregår i det nettleserbaserte verktøyet Jupyter Notebook.

Dette rammeverket forenkler kommunikasjonen med eksterne lagringsenheter, og egner seg dermed godt for denne oppgaven. Utviklingsbrettet inneholder både SD kort leser og tilkoblingsmuligheter for USB, som begge kan interageres med via PYNQ. Den første utfordringen er å gjøre dataen lesbar av prosessorsystemet som kjører PYNQ, og derfra skrive den til valgt lagringsenhet. Den andre utfordringen er å skrive programvare som gjør disse operasjonene raskt nok til at ønsket pingrate kan opprettholdes. Python er et tolket språk som vil si at programmet tolkes av prosessoren linje for linje når det blir kjørt. Dette vil oftest føre til programvare som er tregere enn tilsvarende programmer skrevet i kompilerte språk [13] hvor programmet oversettes til maskinkode en gang ved å kompileres, og som deretter kan kjøres uten å måtte oversettes for hver gang.

Aternativt kan drivere implementeres direkte i logikken eller i en myk prosessorkjerne (en prosessor implementert i den programmerbare logikken) programmert i C. Ettersom Python var godt kjent fra før, falt valget på en PYNQ basert løsning.

3.9.2 AXI DMA og AXI4-Stream

Overføring av data fra den programmerbare logikken til prosessorsystemet kan gjøres via AXI Direct Memory Access (AXI DMA) [5], som er en Xilinx IP. Denne IP-en kan brukes til å skrive data fra modulene implementert i den programmerbare logikken direkte til prosessorsystemets dynamic random-access memory (DRAM), som kan leses ved hjelp av funksjoner i PYNQ [10].

Dataoverføringen mellom systemet og AXI DMA foregår over AXI4-Stream, som er en protokoll brukt til adresseløs dataoverføring mellom to punkter [6]. Den består av fire porter, men kan også inneholde flere:

- TREADY går fra mottaker til sender for å signalisere at den er klar for å motta data.
- TVALID går fra sender til mottaker for å signalisere at dataen som blir overført er gyldig.
- TLAST går fra sender til mottaker for å signalisere at transaksjonen er ferdig.
- TDATA går fra sender til mottaker og inneholder dataen som skal sendes.

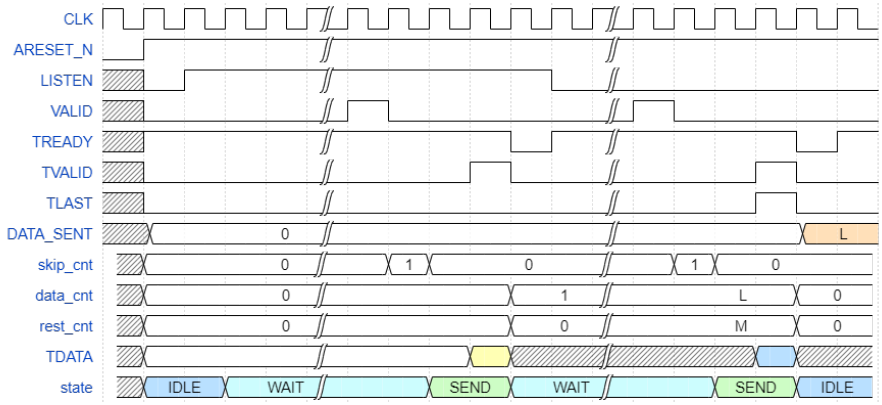
Implementasjon av AXI4-Stream

Modulen for å drive dataoverføringen ble skrevet som en tilstandsmaskin. Den starter i en inaktiv tilstand, og venter på at systemet skal gå i lyttemodus. Når dette skjer går den over i neste tilstand hvor den venter på VALID fra AD-utleseren. Ved å ignorere annen hver VALID puls henter vi bare ut hver andre sample, slik at signalet i praksis blir nedsamplet. Ettersom omhyllingskurven vil ha en lavere frekvens enn bærebølgen kan en nedsampling være fordelsmessig for å minke mengden data som må lagres. Velger vi å ignorere ett høyere antall VALID pulser, vil vi få en høyere grad av nedsampling.

Etter ønsket antall VALID pulser er mottatt går modulen til neste tilstand, setter data ut og TVALID høy. Dersom mottakeren har TREADY høyt, går tilstandsmaskinen tilbake til forrige tilstand, og venter på ny gyldig data. Hvis TREADY ikke er høy, venter tilstandsmaskinen i denne tilstanden frem til den blir det.

Når systemet går ut av lyttemodus, sørger modulen for at alle skiftregistrene får tømt seg før den går tilbake til inaktiv tilstand. Dette gjør den ved å

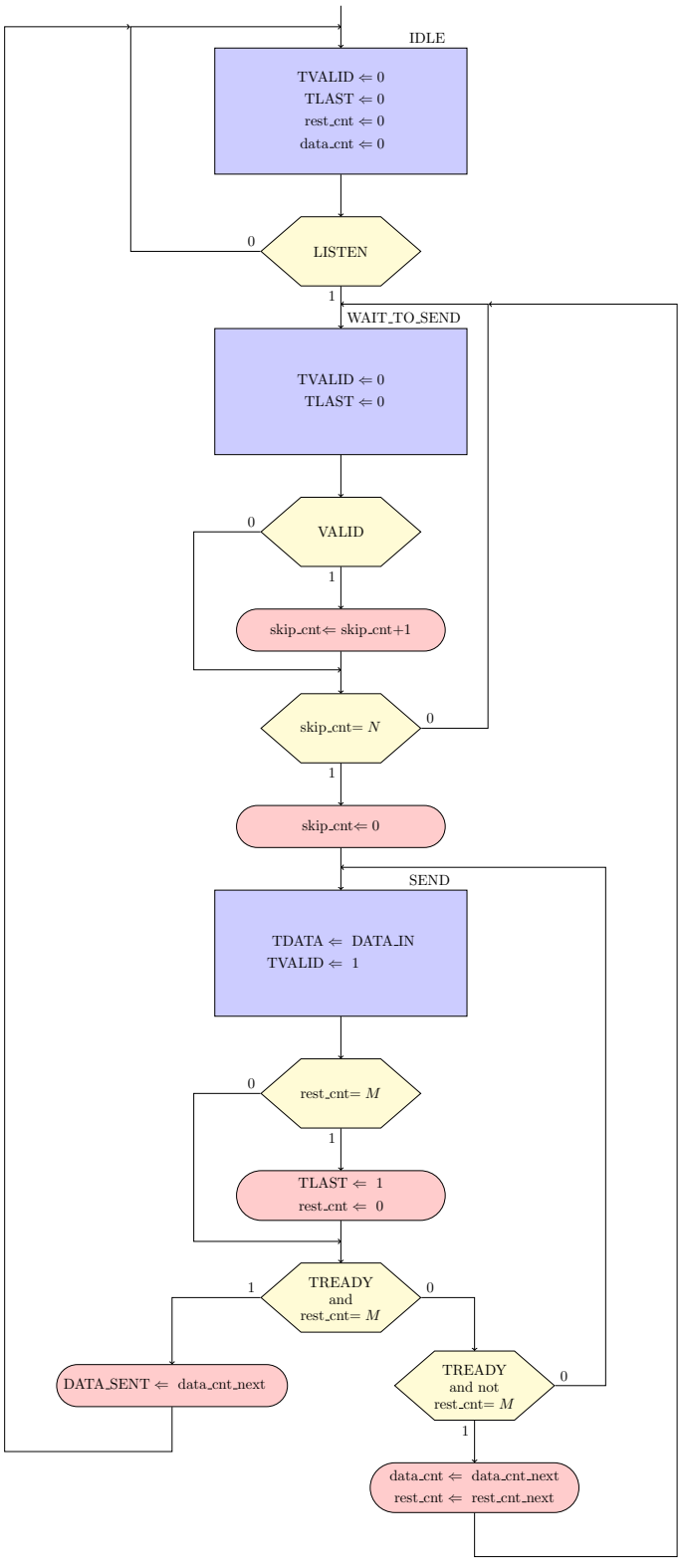
ha en rest-teller som har en størrelse lik summen av lengden til alle skiftregistrene og ytterlige forsinkelser i systemet. Når denne telleren er ferdig, settes TLAST høy for å signalisere at transaksjonen er ferdig. Ved slutten av transaksjonen sendes også DATA_SENT, som er ett tall på hvor mange sampler systemet har behandlet, slik at dette også kan skrives til fil. Modulen går så tilbake til inaktiv tilstand og venter på at systemet skal gå tilbake i lyttemodus. ASM diagram for modulen kan bli sett i figur 31.



Figur 30: Timing for AXI4-Stream modul. Signaler med små bokstaver er interne, de med store bokstaver er inngangs og utgangssignaler. Lengden til rest- telleren er her oppgitt som M.

3.9.3 AXI-GPIO

Signalet DATA_SENT produsert av systemet for hvert ping blir overført til prossessorsystemet via en AXI-GPIO [14]. Denne Xilinx IP-en har to konfigurerbare porter som kan kobles til modulene i den programmerbare logikken. IP-en er også koblet til prossessorsystemet via et AXI4-Lite grensesnitt. PYNQ inneholder funksjoner for å lese og skrive til AXI-GPIO. Dermed kan modulen brukes til å implementere kontroll- og statusregistre i designet raskt uten å måtte inkorporere AXI4-Lite grensesnitt direkte. For fremtidig utvidelse burde likevel direkte inkorporasjon vurderes. Alternativt kunne DATA_SENT bli overført via AXI4-Stream som siste data i pakken.



Figur 31: ASM diagram over AXI-Stream modulen.

3.9.4 AXI-Stream Data FIFO

For tilfeller hvor et nytt ping skulle behandles under lagringen av forrige, ble en First In First Out (FIFO) buffer plassert før DMA modulen. FIFO-en som er brukt er en Axi Stream Data FIFO [7] satt til maksimal størrelse på 32768 32-bit sampler. Dette tilsvarer $32.7\ ms$ med data med en samplingshastighet på 1 MSPS, som gir Python programmet tid til å skrive ferdig.

3.9.5 Asyncio

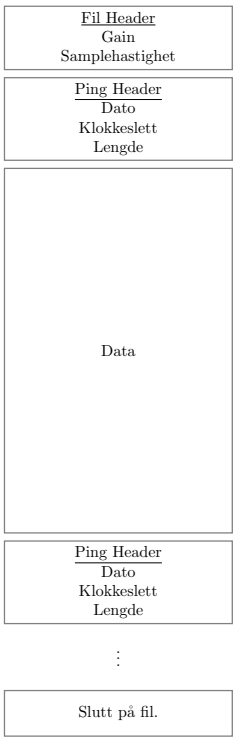
For å realisere kommunikasjon mellom den programmerbare logikken og prosessorsystemet ble Python biblioteket *Asynchronous I/O* (asyncio) brukt. Ved hjelp av dette biblioteket kan funksjoner kjøres samtidig ved at prosessoren bytter mellom dem [4]. Dette tillot også bruk av interrupts ved å skrive funksjoner som venter på disse.

Interrupt signaler blir sendt fra den programmerbare logikken til prosessorsystemet for å gjøre det oppmerksom på noe viktig. AXI DMA modulen vil for eksempel sende en interrupt når dataoverføringen er ferdig, og dermed gjøre prosessorsystemet oppmerksom på dette slik at det kan behandle dataen (i vårt tilfelle skrive den til fil).

3.9.6 Filformat

For å gjøre overføringen så rask og effektiv som mulig skrives data direkte i binært format til en datafil med .dat filendelse. Filen blir opprettet med en fil header bestående av litt generell systeminfo, som forsterkningen i den analoge kretsen og samplingshastigheten til systemet.

For hvert ping opprettes en ping header, som inneholder dato (som gjør etteranalyse enklere ved innsamling av data over flere dager), klokkeslett ned til millisekundpresisjon og lengde på pinget i antall sampler. Deretter kommer selve samplene fra pinget. For neste ping genereres ny ping header etterfulgt av dataen, frem til innsamling er ferdig. Alle informasjon og data blir skrevet til fil som 32-bit heltall.



Figur 32: Visuell fremstilling av oppbygningen til filene brukt av systemet for å lagre data.

Implementasjon av driver i Python

I PYNQ allokeres en buffer i minnet som dataen leses inn i. Bufferstørrelsen ble vilkårlig satt til 500000 32-bit heltall, som tilsvarer 0.5 s med data med en samplingshastighet på 1 MSPS. Med en ønsket pingrate på 10 Hz vil ikke denne bufferen fylles opp, og det vil være lengden på LISTEN som avgjør når transaksjonen er ferdig. Ved hjelp av asyncio ventes det på at DMA modulen signaliserer at transaksjonen skal bli ferdig via interrupt signal, før headere genereres og innholdet i bufferen blir skrevet til fil.

3.10 Fullstendig systemoversikt og ressursbruk

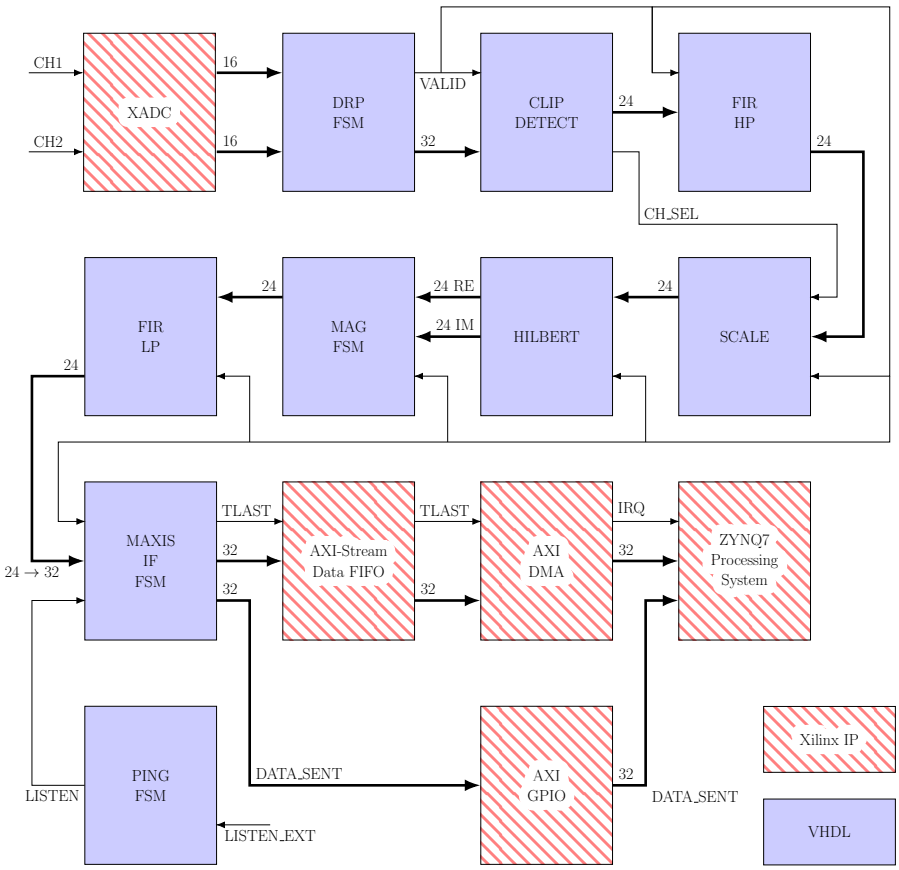
Fullstendig systemoversikt over modulene beskrevet i tidligere seksjoner er vist i figur 33. Legg merke til at kanalvelgermodulen har blitt delt opp i en modul for deteksjon av klipping (CLIP_DETECT) før høypassfilter og en modul for skalering (SCALE) etter høypassfilter. Grunnen til dette er diskutert i seksjon 5.2.1. Fullstendig blokkaskjema inkludert automatisk genererte moduler er inkludert i appendix (ref. fig. 60).

I tabell 3 er systemets ressursbruk oppgitt.

Tabell 3: Ressurser brukt av systemet, samt ledige ressurser til utvidelse.

	Brukt	Utnyttelsesgrad	Ledig
LUT	7846	14.75%	45354
Flip-Flop	11366	10.68%	95034
BRAM	131.5*	93.9%	8.5
DSP	78	35.45%	142

*96 (68.57%) av disse er brukt av ILA.



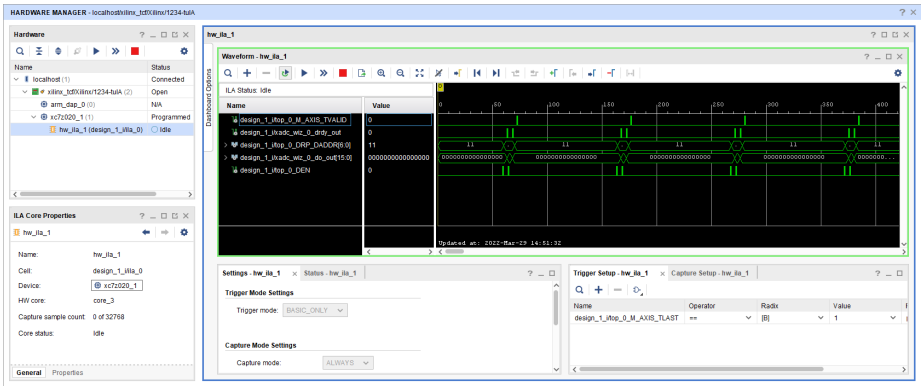
Figur 33: Fullstendig oversikt over modulene i systemet sammen med de viktigste koblingene.

3.11 Programvare

VHDL koden som er skrevet for prosjektet må oversettes til en fysisk implementasjon. Denne prosessen kan deles i to: *syntese* og *implementasjon*. Gjennom syntese oversettes koden til en fysisk beskrivelse av kretsen. Denne beskrivelsen består av komponentene som trengs for å realisere systemet, som for eksempel logiske porter og registre, samt koblingene som må gjøres mellom dem. Gjennom implementasjon allokeres logikken til fysiske logikkblokker på FPGA-en en jobber på. Ettersom brikken brukt i prosjektet kommer fra Xilinx ble syntese og implementasjon gjort i programmet Vivado [29] fra samme leverandør. Programmet er i tillegg til disse nødvendige funksjonene utstyrt med funksjonalitet for å forenkle designprosessen og for å analysere det ferdigstilte produktet.

Vivado inneholder en blokkskjemaoversikt. Ved hjelp av denne kan man kombinere egne moduler med Xilinx IP-er, og enkelt koble disse sammen ved hjelp av et dra og slipp brukergrensesnitt.

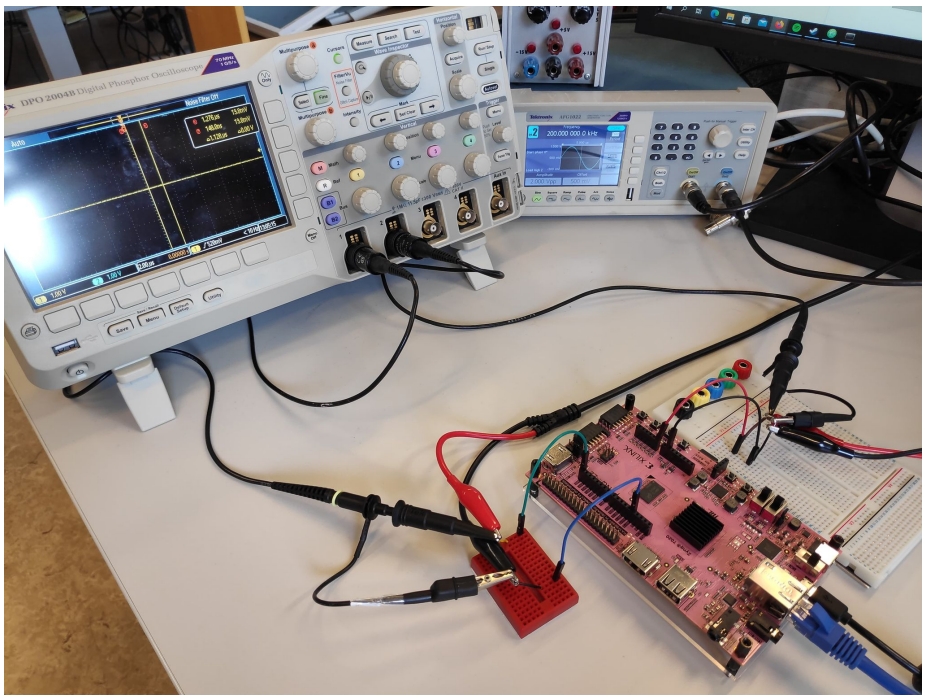
Ved hjelp av Xilinx IP-en Integrated Logic Analyzer (ILA)[12] kan man etter programmeringen av FPGA-en få en oversikt over interne signaler. Logikkanalysatoren kan settes på porter og koblinger mellom moduler man ønsker å analysere, og mens systemet kjører vil analysatoren fange, lagre og vise signalene i et vindu i Vivado.



Figur 34: Skjermbilde fra logikkanalysatoren i Vivado.

3.12 Testing

Testing ble gjort på deler av og hele systemet. Ved å teste de forskjellige modulene separat kan vi kartlegge hvilke deler at systemet som fungerer bra og hvilke som må forberdes. Testingen ble utført med en signalgenerator av typen Tektronix AFG1022 brukt til å generere testsignaler. Disse ble sendt inn til FPGA-en via et brødbrett. I tillegg er et Tektronix DPO 2004B oscilloskop brukt for å samle analog data og verifisere at inngangssignalene er korrekt. ILA er brukt for å verifisere og lagre indre signaler og utgangssignaler produsert av systemet. For å teste indre signaler, trekkes disse ut til toppnivå og kobles på ILA.



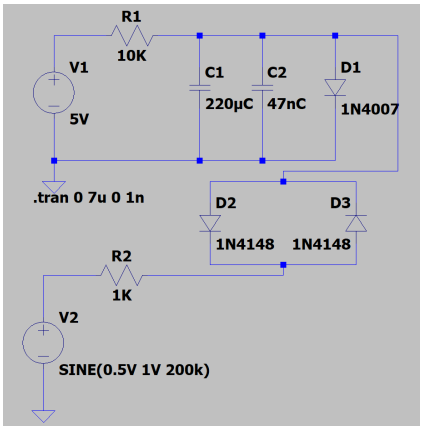
Figur 35: Laboppsett med utstyr.

Python er brukt til å plotte signaler og gjøre kalkulasjoner på resultat. ILA sampler en gang per klokkesyklus mens systemet produserer ny data hver 104 klokkesyklus. Dette betyr at ILA i praksis sampler samme verdi 104 ganger. For å se det korrekte signalet hentes hver 104. sample ut i Python ved plotting av signalene.

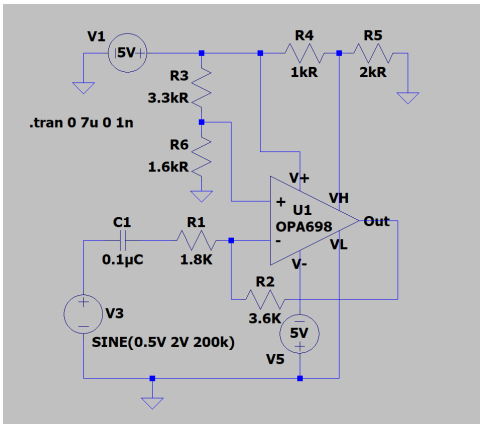
3.12.1 Testing og simulering av beskyttelseskrets

Kretsene ble simulert i LTspice [15] og testet fysisk med signalgenerator og oscilloskop. Simuleringsoppsettene kan bli sett i fig. 36 og 37. I begge tilfel-

lene kjøres en transientanalyse på 7 μs med en maksimal avstand på 1 ns mellom punktene. Diodekretsen ble for enkelhetsskyld simulert og testet uten forsterkning, og man ser dermed på området -0.1 V til 1.1 V. Operasjonsforsterkeren skal forsterke signalet, men begrense det til området 0 V til 3.3 V. Innstillingene for inngangssignal er de samme for både simulering og testing, og de er oppsummert i tabell 4.



Figur 36: Simuleringsoppsett for krets med signaldioder.



Figur 37: Simuleringsoppsett for krets med OPA698.

Tabell 4: Oppsummering av innstillingene for simulering og testing av beskyttelseskrets.

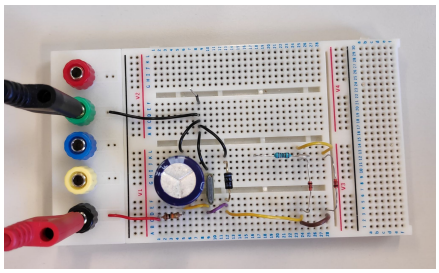
	Diodekrets	Operasjonsforsterkerkrets
Bølgeform	Sinus	Sinus
DC	0.5 V	1.65 V
V_{pp}	2 V	2 V
Frekvens	200 kHz	200 kHz

Diodekretsen ble koblet opp fysisk med komponenter på brødbrett, og kretskort for operasjonsforsterkerkretsen ble produsert på ELAB.

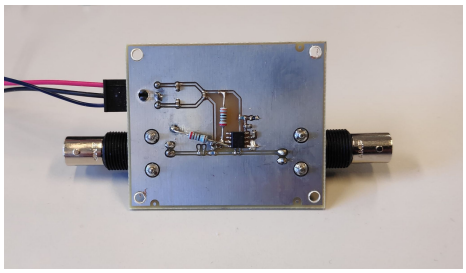
3.12.2 Testing av AD-omformer

Under initiell testing av systemet ble det observert en uventet frekvensrespons fra AD-omformeren. Derfor ble det bestemt å sette opp en test for å kartlegge denne.

For å utelukke feil i egenutviklede moduler, ble et system kun bestående av omformeren og andre nødvendige Xilinx IP-er satt sammen (figur 40) for å teste omformeren. ILA ble koblet på utgangen via AXI4-Stream for å



Figur 38: Diodekretsen koblet på brødbrett.

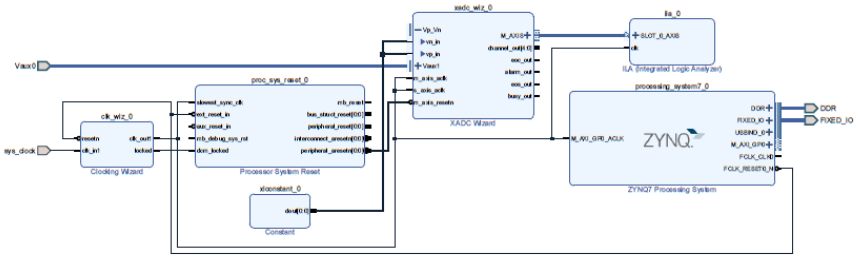


Figur 39: Kretskort med OPA698.

verifisere og lagre utgangssignalet. Innstillingene for signalgeneratoren er oppsummert i tabell 5.

Tabell 5: Oppsummering av innstillingene på signalgenerator for testing av AD-omformer.

Bølgeform	Sinus
DC	1.65 V
V_{pp}	2.5 V
Frekvens	10 - 450 kHz



Figur 40: Blokkskjema av testkrets for ADC med kun Xilinx IP.

3.12.3 Testing av kanalvelgeren

For å teste kanalvelgeren ble utgangen av modulen trukket ut til toppnivå og koblet til ILA. Begge utgangene på signalgeneratoren ble brukt ved testing av modulen. Signalet på den ene utgangen ble valgt til å ha dobbel så

stor V_{pp}, slik at multiplikasjonsfaktoren i kanalvelgerenen ble 2 (implementert som et bitskift til venstre). For ikke å risikere skade på omformeren ble V_{pp} amplituden til signalet plassert godt under maks på 3.3 V. På grunn av dette måtte grensene for å bytte kanal i modulen settes noe ned. Øvre grense ble satt til $9F4_{16}$ og nedre grense ble satt til $60C_{16}$. Innstillingene som er brukt på signalgenerator er oppsummert i tabell 6.

I tillegg ble kanalen med høyest forsterkning også trukket ut til toppnivå og koblet til ILA. Ettersom V_{pp} er bare er på 3 V vil ikke omformeren gå i metning, og kanalen vil kunne sample hele signalet. Kanalvelgerenens funksjon er å supplementere med data for å "fullføre" signalet når første kanal er gått i metning. Dermed kan det fullstendige signalet samplet med bare en kanal brukes som referanse for sammenligning.

Tabell 6: Oppsummering av innstillingene på signalgenerator for testing av kanalvelgeren.

	Utgang 1	Utgang 2
Bølgeform	Sinus	Sinus
DC	1.65 V	1.65 V
V_{pp}	3 V	1.5 V
Frekvens	50 kHz, 100 kHz, 200 kHz	50 kHz, 100 kHz, 200 kHz

3.12.4 Testing av modulene for uthenting av omhyllingskurve

Testing av modulene for uthenting av omhyllingskurven ble gjort ved å bare bruke en kanal. Signalet som ble sendt inn ble valgt slik at V_{pp} var lav nok til å omgå kanalvelgeren. Dette ble gjort for at ingen feil i kanalvelgeren skulle påvirke testingen av modulene. Innstillingene for signalgeneratoren er oppsummert i tabell 7. Andre utgang av signalgeneratoren ble brukt til å generere et LISTEN-signal.

ILA ble brukt for verifisering og uthenting av data til CSV-fil.

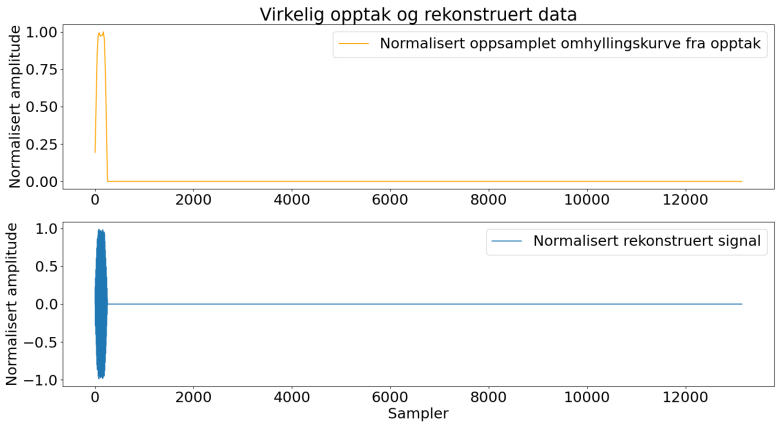
Tabell 7: Oppsummering av innstillingene på signalgenerator for testing av modulene for uthenting av omhyllingskurve.

	Utgang 1	Utgang 2
Bølgeform	Amplitudemodulert sinus	Firkant puls
DC	1.65 V	1.65 V
V_{pp}	2.5 V	3.3 V
Frekvens	50 kHz, 100 kHz, 200 kHz	10 Hz
AM-frekvens	10 kHz	-
Duty cycle	-	40%

3.12.5 Testing på reell data

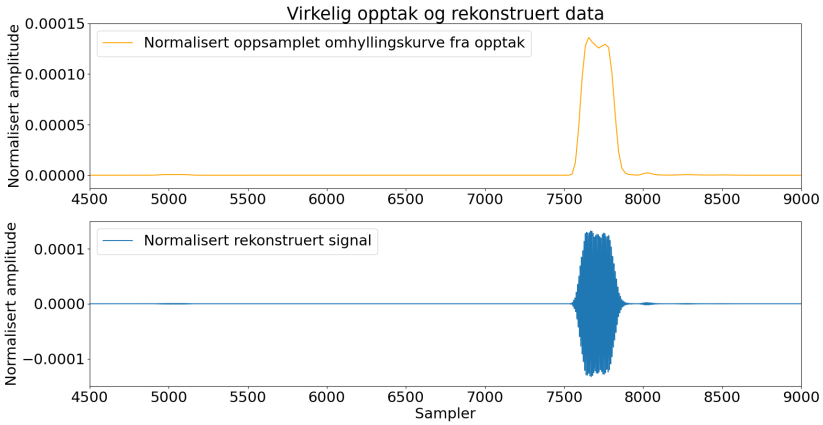
For å teste systemet som helhet ble tidligere opptak med ekkolodd brukt til å produsere kunstige digitale sampler som ble sendt inn i systemet. Dataen kommer fra måling med CW puls på kalibreringskule i tank gjort med et EK80 ekkolodd, og er gjenbrukt med tillatelse fra Frank Reier Knudsen ved Simrad - Kongsberg Maritime.

Opptaket hadde en rekkevidde på 10 m, og kom i form av en omhyllingskurve som bestod av 626 sampler. Oppgitt lydhastighet i tanken var på 1500 m/s som med ligning 1 gir en varighet på $1/75$ s. Med 626 sampler betyr dette at opptaket var nedsamplet til $626/(1/75)$ s = 46950 Hz. Derfor ble en oppsampling nødvendig for å lage et signal som passer systemets spesifikasjoner (bærebølge mellom 50 og 200 kHz). Ved å sette inn 21 nye sampler mellom hvert originalsample, og interpolere mellom disse ble omhyllingskurven oppsamplet til 985950 Hz. Signalsamplene ble generert ved at en normalisert omhyllingskurve ble multiplisert med en 120 kHz sinusbølge, som er lydfrekvensen brukt i opptaket. Signalet ble så skalert og omgjort til 16-bit heltall, hvor de 12 MSB-ene er signalverdier og de 4 LSB-ene ble satt til 0. Forskjellig skalerte versjoner av signalet ble konkatenert til et 32-bit tall for å etterligne utgang fra utlesningsmodulen til AD-omformeren.



Figur 41: Omhyllingskurve fra opptak og rekonstruert signal.

Originalopptaket inkluderte ringing fra transduseren, som har en mye større signalamplitude enn ekkoet fra kalibreringskulen, og setter et stort krav til systemets dynamikkområde. På grunn av dette ble to sett med testdata generert. Det første settet består av hele signalet, og har en forsterkningsforskjell på 50 (34 dB) mellom kanalene for å synliggjøre ekkoet fra kalibre-



Figur 42: Omhyllingskurve fra opptak og rekonstruert signal. Ekkoet fra kalibreringskulen er her forstørret.

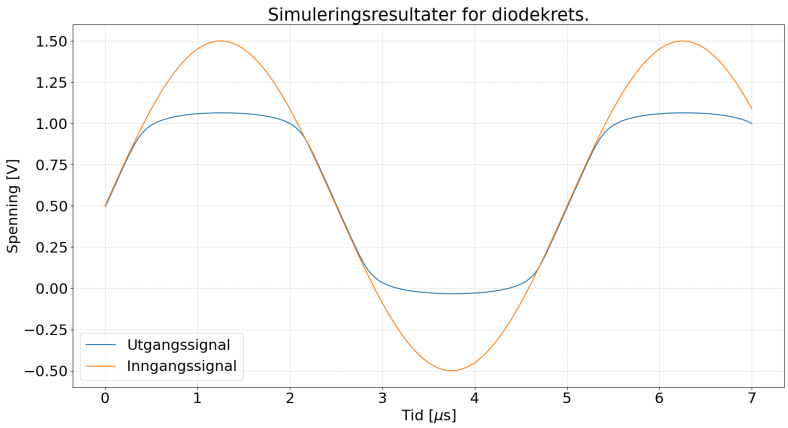
ringskulen. I det andre settet ble første del av signalet ignorert (ekvivalent med å starte lyttingen litt senere, og dermed overse et område på en ca. en meter fra transduseren), slik at kravet til dynamikkområde ble mindre og en forsterkning på 2 (6 dB) kunne brukes mellom kanalene.

En egen modul ble laget i VHDL for å sende ut ny data med $1\ MHz$ frekvens. Den er skrevet som en tilstandsmaskin med to tilstander, en inaktiv tilstand (IDLE) og en sendetilstand (SEND). I inaktiv tilstand venter modulen på LISTEN. Samtidig sender den en DC verdi på halvparten av maksimal utgangsverdi, slik at utgangen på høypassfilteret får stabilisert seg rundt 0. I sendetilstand sender modulen det rekonstruerte signalet. Datasamplene fra det rekonstruerte signalet er lagret i en array med en indeks som inkrementeres for hver sample som blir sendt. Denne modulen ble satt inn i stedet for AD-omformeren og utlesningsmodulen. På denne måten kunne den fysiske implementasjonen av systemet testes i stedet for å simulere via en testbenk, slik at resultatene reflekterer det implementerte systemets ytelse.

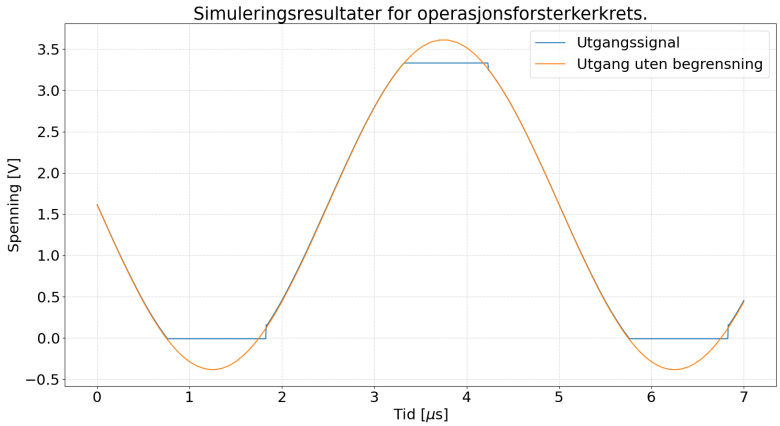
4 Resultater

4.1 Beskyttelseskrets

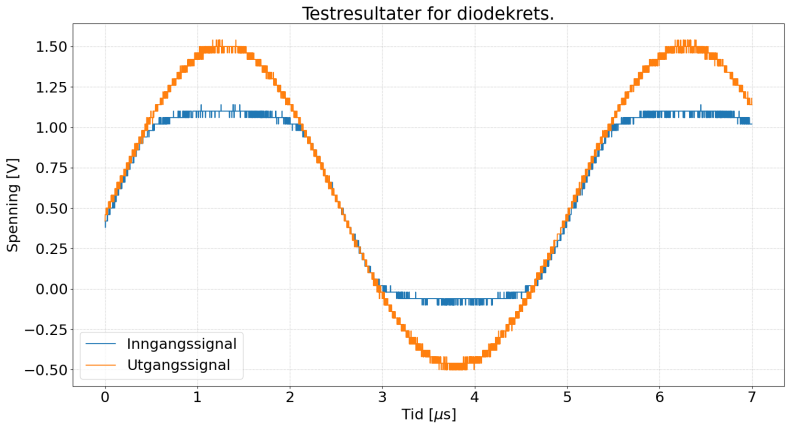
Simulasjonsresultatene er vist i fig. 43 og 44. På grunn av at vi har benyttet en inverterende operasjonsforsterker så er kurven i figur 44 $180°$ faseforskjøvet. De fysiske målingene er vist i fig. 45 og 46. Testdata er lagret fra oscilloskop.



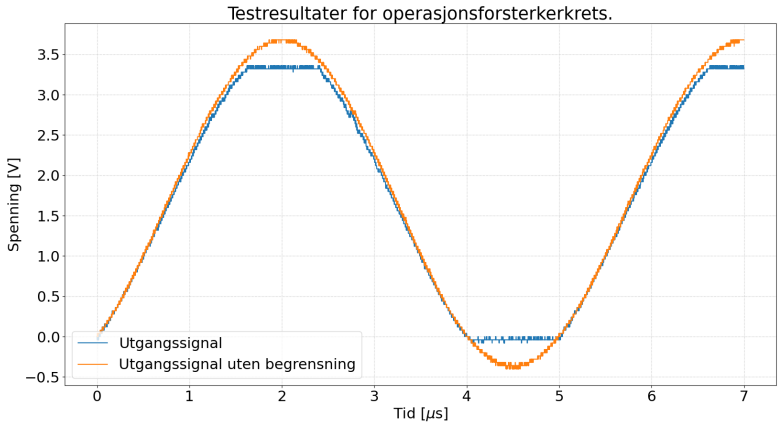
Figur 43: Resultatet av den simulerte transientanalysen av diodekretsen.



Figur 44: Resultatet av den simulerte transientanalysen av operasjonsforsterkerkretsen.



Figur 45: Resultatet av testen av diodekretsen med signalgenerator.



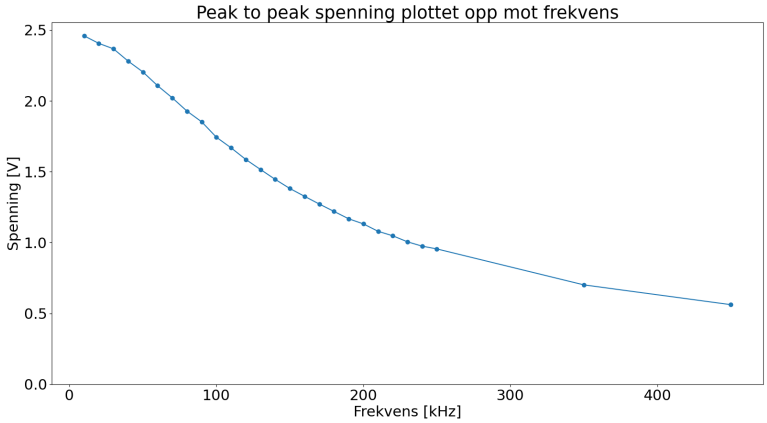
Figur 46: Resultatet av testen av operasjonsforsterkerkretsen med signalgenerator.

I figur 46 kan vi se at utgangssignalet fra operasjonsforsterkerkretsen ikke følger det ideelle utgangssignalet som i simulasjonen. Dette kan være grunnet ikke-idealiteter i spenningsdelerne i kretsen, som gjør at DC-offset på pluss inngangen blir litt lavere enn det forventede nivået. I tillegg ser det ut for at forsterkningen er litt lavere enn 2.

Til tross for dette viser simulasjonene og testene ganske like resultat. Diodekretsen gjør at utgangssignalet ”bøyes av” litt før det skal begrenses, slik at dynamikkområdet på signalet blir mindre. Operasjonsforsterkerkretsen viser en bedre overgang til begrensning uten stor påvirkning av signalet innenfor grensene, som gjør at vi i større grad kan bruke hele signalet.

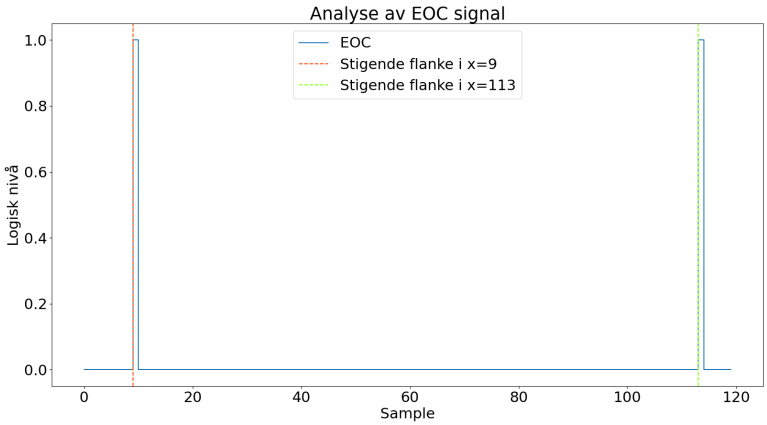
4.2 AD-omformer

Omformerens frekvensrespons er vist i figur 47. Resultatene viser en tydelig nedgang i signalamplitude for høyere frekvenser.



Figur 47: Data ut fra ADC. V_{pp} verdier er produsert ved å sende inn sinusbølger med DC-offset på 1.65 V og V_{pp} på 2.5 V. Frekvensen på signalet ble variert fra 10 kHz til 250 kHz med 10 kHz mellomrom. I tillegg ble det målt for 350 kHz og 450 kHz.

Første tanke var at dette var en form for aliasing, som oppstår når samplingfrekvensen er for lav. Ved hjelp av ILA kunne vi verifisere at omformeren produserte en verdi hver 104 klokkesyklus (fig. 48), som med en klokke på 100 MHz gir en samplingfrekvens på ≈ 961 kHz. Denne samplingfrekvensen tilsier at høyeste frekvensen på inngangssignalet skal være på ≈ 480 kHz, som er høyere enn systemets maksimalfrekvens.



Figur 48: Logikkanalyse av EOC pulsene til AD-omformeren. Analysen viser 104 klokkesykluser ellom hver EOC, som med en 100 MHz klokke gir samplingfrekvens på ≈ 961.5 kHz.

4.3 Kanalvelger

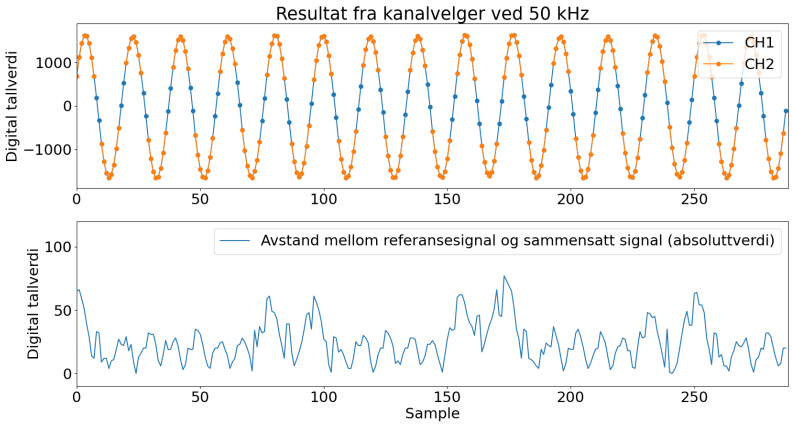
Resultatene fra testing av kanalvelgeren er vist i figur 49, 50 og 51. Figurene viser at det blir brukt skalert data fra kanal 2 på de stedene hvor sampler fra kanal 1 ville vært over de oppgitte grensene.

Effekten av AD-omformerens frekvensrespons kan sees tydelig på signalenes topp til topp amplitude fra figur 49 til figur 51. En langt større del av signalet havner under grensene for kanalvalg ved 200 kHz sammenlignet med 50 kHz. På grunn av dette blir det vanskeligere å karakterisere kanalvelgerens funksjonalitet ved høyere frekvenser.

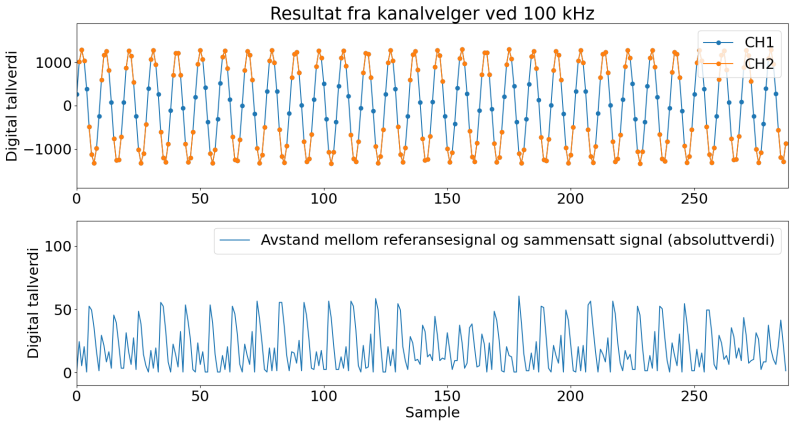
Den gjennomsnittlige og maksimale avstanden mellom det sammensatte signalet og referansesignalet ble beregnet, og er vist i tabell 8.

Tabell 8: Sammenligning av sammensatt signal og refearansesignal. Størrelsene er oppgitt i digital verdi, med prosentandel av topp til topp signalamplitude i parantes.

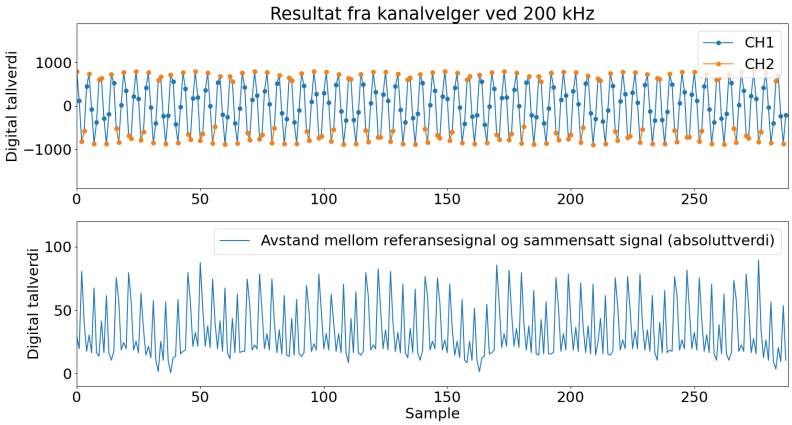
	50 kHz	100 kHz	200 kHz
Maksimal avstand	77 (2.38%)	61 (2.35%)	89 (5.45%)
Gjennomsnittlig avstand	25 (0.76%)	19 (0.75%)	34 (2.05%)



Figur 49: Resultater av test på kanalvelger ved 50 kHz. Øverste plot viser sammensetningen av signalet ut fra kanalvelger. Nederste plot viser avviket mellom sammensatt signal og referansesignal.



Figur 50: Resultater av test på kanalvelger ved 100 kHz. Øverste plot viser sammensetningen av signalet ut fra kanalvelger. Nederste plot viser avviket mellom sammensatt signal og referansesignal.

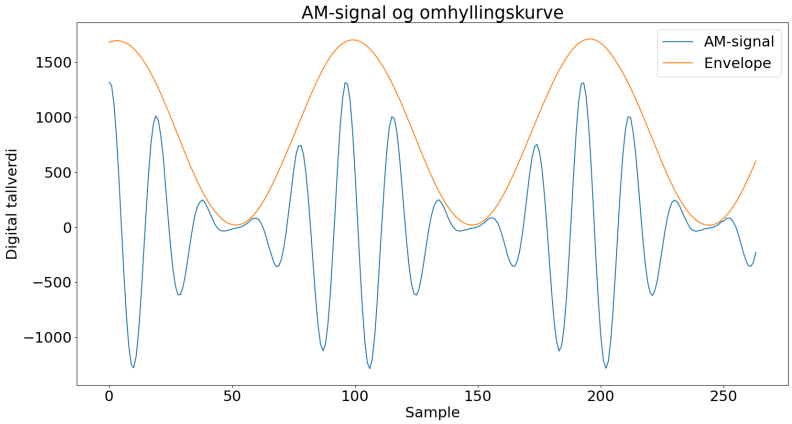


Figur 51: Resultater av test på kanalvelger ved 200 kHz. Øverste plot viser sammensetningen av signalet ut fra kanalvelger. Nederste plot viser avviket mellom sammensatt signal og referansesignal.

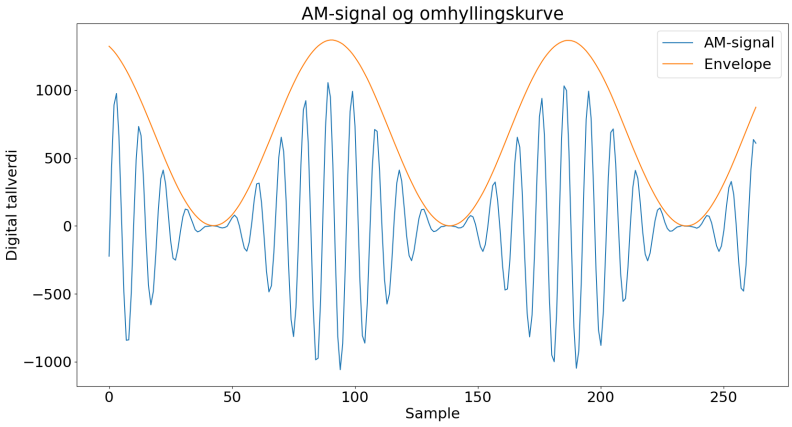
4.4 Uthenting av omhyllingskurven

4.4.1 Metode 1

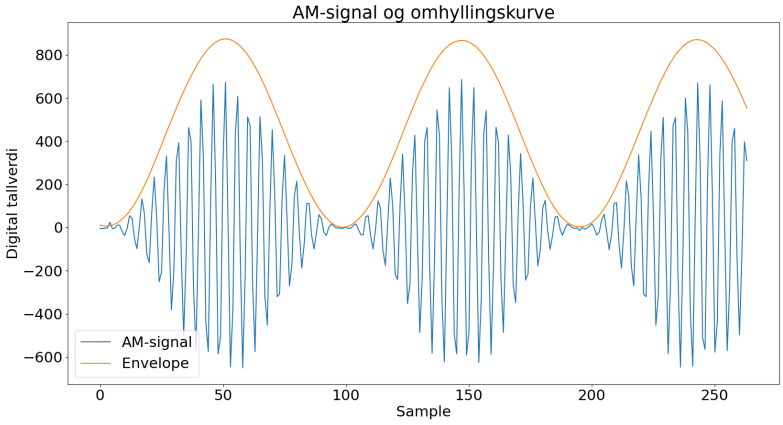
Figur 53, 53 og 54 viser målingene gjort med metode 1 for uthenting av omhyllingskurven (ref. fig. 18). Metoden fører tilsynelatende til at omhyllingskurven får en større amplitude enn den skal.



Figur 52: Resultat av uthentingen av envelopen til et amplitudemodulert signal med 50 kHz bærebølge. Envelopen har en frekvens på 10 kHz.



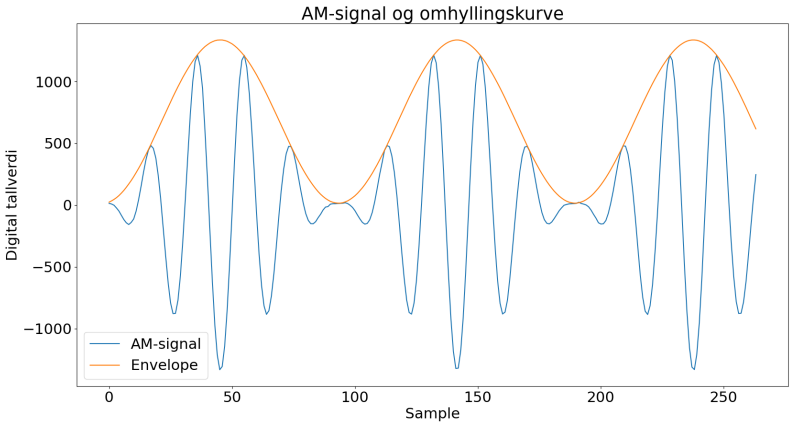
Figur 53: Resultat av uthentingen av envelopen til et amplitudemodulert signal med 100 kHz bærebølge. Envelopen har en frekvens på 10 kHz.



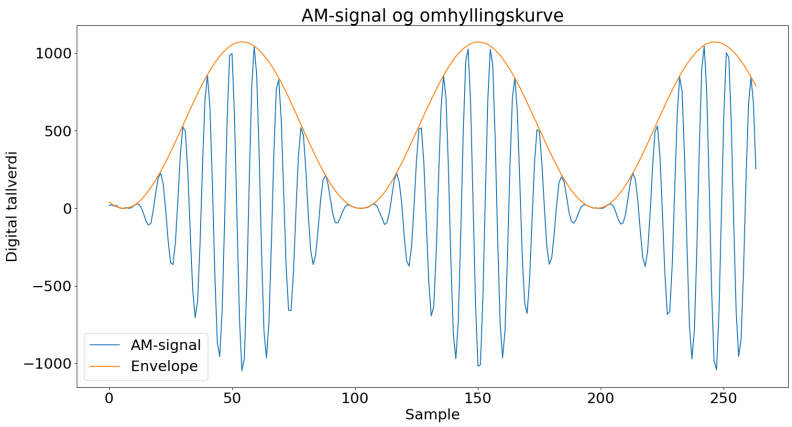
Figur 54: Resultat av uthentingen av envelopen til et amplitudemodulert signal med 200 kHz bærebølge. Envelopen har en frekvens på 10 kHz.

4.4.2 Metode 2

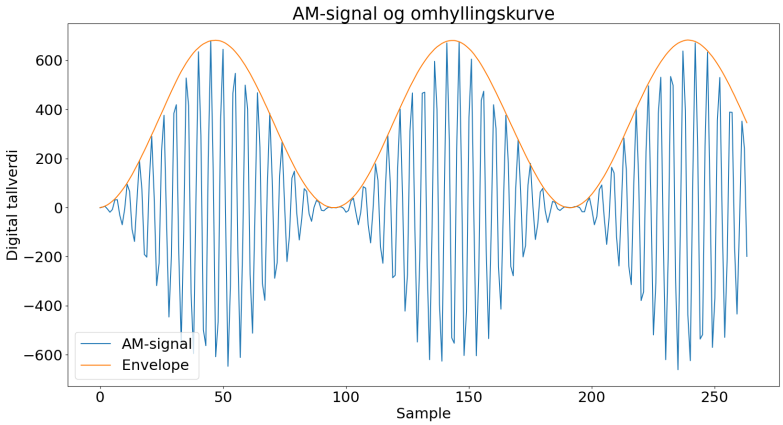
Figur 56, 56 og 57 viser målingene gjort med metode 2 for uthenting av omhyllingskurven (ref. fig. 19). Denne metoden produserer en omhyllingskurve som i større grad passer inngangssignalet.



Figur 55: Resultat av uthentingen av envelopen til et amplitudemodulert signal med 50 kHz bærebølge. Envelopen har en frekvens på 10 kH



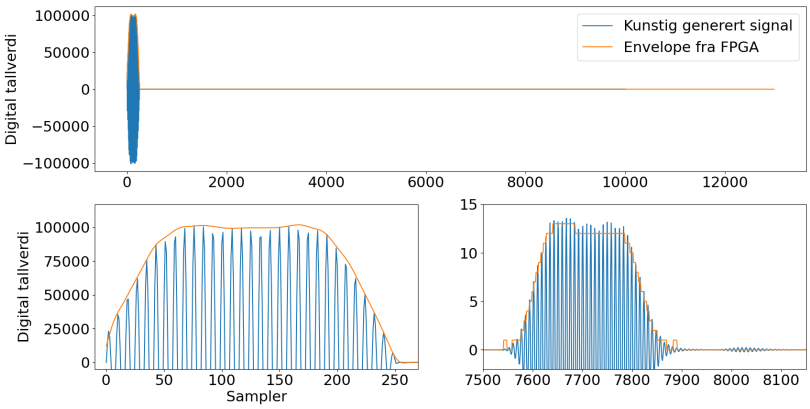
Figur 56: Resultat av uthentingen av envelopen til et amplitudemodulert signal med 100 kHz bærebølge. Envelopen har en frekvens på 10 kHz.



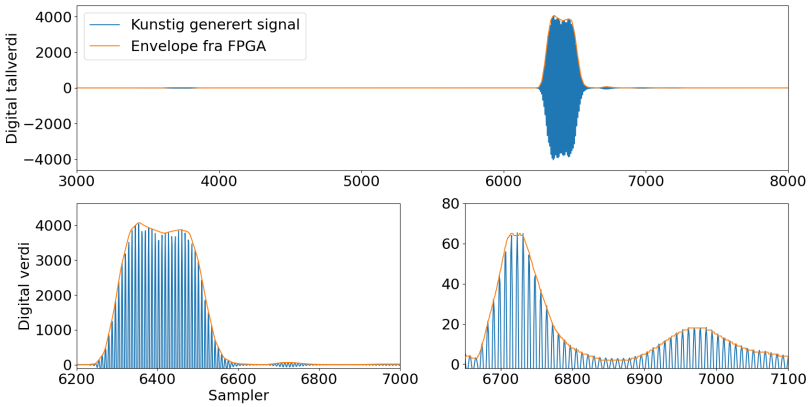
Figur 57: Resultat av uthentingen av envelopen til et amplitudemodulert signal med 200 kHz bærebølge. Envelopen har en frekvens på 10 kHz.

4.5 Resultat fra behandling av rekonstruerte data fra virkelig opptak

Figur 58 viser resultatene fra behandling av hele opptaket, mens figur 59 viser resultatene fra behandling av opptaket med ringing fjernet. Resultatene er hentet fra fil produsert av driveren i prosessorsystemet.



Figur 58: Resultat av uthentingen av envelopen til rekonstruert signal. Det øverste plottet viser hele resultatet. Plottet nede til venstre er pinget fra transduseren forstørret. Plottet nede til høyre er ekkoet fra kalibreringskulen forstørret.



Figur 59: Resultat av uthentingen av envelopen til reelt signal. Her er første meteren av det opprinnelige signalet ignorert for å senke kravet til dynamikkområde. Det øverste plottet viser hele resultatet. Plottet nede til venstre er ekkoet fra kalibreringskulen forstørret. Plottet nede til høyre er sannsynligvis gjenklang fra kalibreringskulen.

Fra resultatene kan man se at systemet klarer å hente ut omhyllingskurven til det rekonstruerte signalet i begge tilfeller. I testen med ringing inkludert får ekkoet fra kalibrerinskulen en litt lavere amplitude og dårligere oppløsning, som viser at dynamikkområdet ikke var stort nok.

5 Diskusjon

5.1 AD-Omformeren

På grunn av at AD-omformerens frekvensrespons fører til demping av frekvensene som benyttes av systemet anbefales bruk av eksterne omformere. Dette betyr også at modulen som brukes å lese ut av og kontrollere omformeren må byttes ut. Hvis disse likevel elementene skal byttes ut kan en litt dyrere høyhastighets 18-bit ADC vurderes som alternativ til to omformere med lavere oppløsning for å oppnå ønsket dynamikkområde.

5.2 Kanalvelgeren

5.2.1 DC problemet

Skaleringen av data i kanalvelgeren er avhengig av at signalet svinger rundt 0, slik at dette kan gjøres med en enkel multiplikasjon. I tillegg vil Hilberttransformasjonen produsere et faseforskjøvet signal som svinger rundt 0, samtidig som den reelle delen ikke nødvendigvis gjør det. For at disse skal kunne legges sammen til et komplekst signal må enten imaginærdelen heves til samme DC-nivå, eller så må DC-nivå fjernes fra realdelen.

Første tanke var å eliminere DC ved å fjerne en fast verdi tilsvarende halvparten av fullskala utgangssignal fra AD-omformeren ($2^{12}/2 = 2048$). Problemet med denne løsningen var at ytre påvirkninger av den analoge forkretsen vil kunne føre til at DC-verdien også endrer seg, og dermed at denne verdien regelmessig må kalibreres. Den enkleste løsningen ble dermed å inkludere et digitalt høypassfilter for å fjerne DC uavhengig av verdi.

Plasseringen av høypassfilteret skapte imidlertid problemer for kanalvelgeren. I tillegg til signalets DC spenning vil også klippede toppunkt og bunnpunkt ha faste verdier og bli dempet. Dette førte til at topp- og bunnpunkter havnet under modulens grenser for å detektere klipping. For å løse dette blir kanalvelgermodulen delt i to. Metning ble detektert og data fra kanalene ble satt sammen før høypassfiltrering. Skaleringen blir gjort etter filtrering. Dermed vil et signal, lignende det vist i figur 16, bli sendt inn i høypassfilteret. De plutselige overgangene til ikke-skalert signal fra den andre kanalen vil introdusere høyere frekvenser i signalet, og passere upåvirket gjennom filteret. For at de riktige samplene skal skaleres etter filtrering må informasjonen om kanalvalg sendes forbi filteret til skaleringsmodulen, og forsinkes med en tid tilsvarende høypassfilterets forsinkelse.

Et pulstog, som er lavt ved sampler fra første kanal og høyt ved sampler fra andre kanal, sendes fra metningsdetektoren inn i skaleringsmodulen, og forsinkes der ved hjelp av et skiftregister.

5.2.2 Svakheter i den implementerte kanalvelgeren

I tillegg til å få en mer avansert oppbygning på grunn av høypassfilteret, er kanalvelgeren avhengig av at signalene er i fase og at skaleringsfaktoren er korrekt kalibrert. Dette er forklart i seksjon 3.5. Skaleringsfaktoren skal reflektere forskjellen i forsterkning mellom kanalene, som på samme måte som DC-nivået kan endre seg grunnet ytre faktorer som påvirker analogelektronikken. Et slikt regelmessig behov for kalibrering kan gjøre systemet mindre brukervennlig.

Det kan tenkes at en form for automatisk kalibrering kan implementeres. Et testsignal kan sendes med jevne mellomrom og en algoritme kan utvikles for å identifisere hopp i signalamplituden. Systemet kan så øke eller senke multiplikasjonsfaktoren for å kompensere for hoppet, og deretter sende nytt testsignal. Denne prosessen fortsetter frem til skaleringsfaktoren er korrekt.

5.2.3 Andre løsninger til dynamikkproblemet

Ved å bruke en tidsvariabel forsterker eller en forsterker med ikke-lineær forsterkning, kan kanalvelgeren elimineres helt. Forsterkningen i slike komponenter er en kjent funksjon av enten tid eller amplitude, som kan tas med i betraktning under signalbehandling i prosesseringsenheten eller ved etteranalyse. Modulene for uthenting av omhyllingskurve vil kunne brukes uavhengig av hvilken løsning som velges for dynamikkproblemet.

5.3 Uthenting av omhyllingskurven

Resultatene viste at metode 1 produserte en tilsynelatende mindre nøyaktig omhyllingskurve enn metode 2. Videre testing av metode 1 ved forskjellige amplituder og frekvenser kan allikevel være fornuftig for å kartlegge om skaleringsfeilen er konstant. Dersom skaleringen er konstant kan denne kompenseres for, som betyr at magnitudemodulen kan fjernes for å gjøre plass til forbedring eller andre utvidelser.

5.4 Magnitudemodul

Magnitudemodulen er den modulen som setter størst begrensning på systemets totale samplinghastighet. Den iterative prosessen for å regne ut kvad-

ratrot i magnitudemodulen setter et minimumskrav på 28 klokkesykluser som trengs mellom hver sample. Med en systemklokke på 100 MHz betyr dette at omformere kan ha en maksimal samplehastighet på ca. 3.5 MSPS. Ved hastigheter over dette må flere magnitudemoduler avlaste hverandre i paralell, eller en ny modul må implementeres.

Høyere samplinghastigheter kan vurderes for å forbedre forholdet mellom signal og kvantiseringsstøy (SQNR). En samplerate på 2 ganger Nyquist-raten kan for eksempel forbedre SQNR med 3 dB [9].

5.5 Overføring til ekstern lagringsenhet

Driveren for overføring til ekstern lagringsenhet krever mer testing for å kartlegge hastigheten. Av grunner beskrevet i seksjon 3.9.1 kan en implementasjon med Python være for treg. Skulle det vise seg at driveren ikke klarer å opprettholde ønsket pingrate, kan en alternativ løsning med for eksempel dedikert myk prosessorkjerne programmert i C vurderes.

6 Fremtidig arbeid

I denne seksjonen presenteres forslag til forbedringer eller utvidelser.

6.1 Visning av ekkogram

Direkte visning av innkommende data på skjerm i form av ekkogram vil være til nytte ved manuell operasjon og muliggjøre kalibrering av systemet i felt. Utviklingsbrettet brukt i oppgaven er utstyrt med tilkoblingsmuligheter til skjerm via High-Definition Multimedia Interface (HDMI). I tillegg inneholder PYNQ-biblioteket en rekke funksjoner og IP-er for å drive disse. [28]

6.2 Brukerkontroll

Programvaren for å programmere FPGA og lagre data har til nå blitt kjørt gjennom Jupyter Notebook, som krever tilkobling til datamaskin. Dette kan endres til at programvaren kjører ved systemoppstart. Systemet mangler også muligheter for å endre innstillinger under kjøring. Aktuelle innstillinger er for eksempel å stille dybde og pingrate gjennom å bestemme pulslengde og frekvens på LISTEN signal. Implementasjon av kontroll og statusregistre kan gjøres via AXI-GPIO, som forklart i seksjon 3.9.3. Ettersom AXI-GPIO bare har to tilgjengelige konfigurerbare porter, kan en inkorporering av AXI4-Lite direkte vurderes ved behov for mange kontroll- og statusregistre.

Utviklingsbrettet er utstyrt med 4 knapper og 2 brytere, som gir begrensede muligheter for brukerinteraksjon i felt. Innstilling av systemet via bærbar datamaskin er et alternativ, men om mer kompakte løsninger ønskes, kan støtte for tastatur vurderes for implementasjon.

6.3 Strømsparing

For at systemet skal kunne samle data over lengre tid på batterikraft må det være strømeffektivt. Bedre utnyttelse av LISTEN-kontrollsignalet og implementasjon av søvn prosedyre er eksempler på forbedringer som kan gjøres. Den eneste modulen som aktivt benytter LISTEN er AXI4-Stream modulen, som bruker signalet til å gå mellom inaktiv tilstand og sendemodus. De andre modulene kjører uavhengig av dette signalet.

7 Konklusjon

I denne oppgaven ble deler av et ekkoloddsystem for forskningsanvendelser utviklet på FPGA. Fokuset var på dynamikkområde, prosesseringsmodul for uthenting av omhyllingskurve, samt overføring av data til ekstern lagringsenhet.

En løsning for å oppnå stort dynamikkområde ved bruk av to AD-omformere er implementert og testet. Dynamikkområdet er fordelt over to kanaler med ulik analog forsterkning og en AD-omformer for hver kanal. Digitale moduler detekterer metning, skalerer og setter sammen sampler fra begge kanaler til et fullstendig signal. Resultater fra testing på reell data med forskjellig krav til dynamikkområde viser at amplituden til det sammensatte signalet i stor grad er bevart. På en annen side kan modulen slik den er implementert føre til et regelmessig behov for kalibrering.

De implementerte signalbehandlingsmodulene for uthenting av omhyllingskurven er basert på å generere et komplekst signal ved hjelp av Hilbert transformasjon og ta magnituden av dette. Resultatene viser at denne metoden produserer omhyllingskurve som i stor grad bevarer signalets form og amplitude.

Overføringen til ekstern lagringsenhet har blitt gjort via prosessor til SD-kort ved hjelp av funksjoner i rammeverket PYNQ. Systemet produserer filer med 32-bit binærdata, men videre testing er nødvendig for å vurdere om systemet klarer å opprettholde ønsket pingrate.

For et komplett ekkoloddsystem mangler det fortsatt en analog front-end krets bestående av blant annet transduser, forsterkere og anti-aliasing filtre for sending og mottak av signal. I tillegg mangler brukervennlige og kompakte løsninger for å interagere med systemet i felt. Forslag til utvidelser er lagt frem i seksjon 6.

Referanser

[1] TUL. URL: https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html (sjekket 21.05.2022).

[2] *7 Series DSP48E1 Slice*. UG479. v1.10. Xilinx. Mar. 2018. URL: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1 (sjekket 21.05.2022).

[3] *7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter*. UG480. v1.10.1. Xilinx. Jul. 2018. URL: https://docs.xilinx.com/v/u/en-US/ug480_7Series_XADC (sjekket 21.05.2022).

[4] *asyncio - Asynchronous I/O*. URL: https://docs.python.org/3/library/asyncio.html (sjekket 21.05.2022).

[5] *AXI DMA v7.1 LogiCORE IP Product Guide*. PG021. Xilinx. Apr. 2022. URL: https://docs.xilinx.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide (sjekket 21.05.2022).

[6] *AXI Reference Guide*. UG761. v14.3. Xilinx. Nov. 2012. URL: https://docs.xilinx.com/v/u/en-US/ug761_axi_reference_guide (sjekket 21.05.2022).

[7] *AXI4-Stream Infrastructure IP Suite v3.0*. PG085. Xilinx. Nov. 2021. URL: https://docs.xilinx.com/v/u/en-US/pg085-axi4stream-infrastructure (sjekket 21.05.2022).

[8] Helge Balk. *Digitalisering av Sonar*. Masteroppg. Universitetet i Oslo, 1996.

[9] Tony Chan Carusone, David A. Johns og Kenneth W. Martin. *Analog Integrated Circuit Design*. eng. 2. utg. International Student Version. John Wiley & Sons, 2012. ISBN: 9781118092330.

[10] *DMA*. URL: https://pynq.readthedocs.io/en/latest/pynq_libraries/dma.html (sjekket 21.05.2022).

[11] *FPGA advantages and most common applications*. URL: https://hardwarebee.com/fpga-advantages-common-applications-today/ (sjekket 21.05.2022).

[12] *Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide*. PG172. v6.2. Xilinx. Okt. 2016. URL: https://docs.xilinx.com/v/u/en-US/pg172-ila (sjekket 21.05.2022).

[13] Hans Petter Langtangen. *A Primer on Scientific Programming with Python*. eng. 5. utg. Springer, 2016. ISBN: 9783662498866.

[14] *LogiCORE IP AXI GPIO (v1.00a)*. DS744. 1.0. Xilinx. Sep. 2010. URL: https://docs.xilinx.com/v/u/1.0-English/axi_gpio_ds744 (sjekket 21.05.2022).

[15] *LTspice*. Analog Devices. URL: https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html (sjekket 22.05.2022).

[16] Richard Lyons. Digital Envelope Detection: The Good, the Bad, and the Ugly [Tips and Tricks]. I: *IEEE Signal Processing Magazine* 34.4 (2017), s. 183–187. DOI: 10.1109/MSP.2017.2690438. (Sjekket 21.05.2022).

[17] Richard G. Lyons. *Understanding Digital Signal Processing*. eng. 3. utg. Pearson, 2010. ISBN: 0137028458.

[18] David N. MacLennan og E. John Simmonds. *Fisheries acoustics*. eng. 1. utg. Chapman & Hall, 1992. ISBN: 0412330601.

[19] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. eng. Newnes/Elsevier, 2004. ISBN: 0750676043.

[20] *PYNQ - Introduction*. URL: https://pynq.readthedocs.io/en/latest/ (sjekket 21.05.2022).

[21] *PYNQ - Python Productivity for Zynq*. URL: http://www.pynq.io/ (sjekket 21.05.2022).

[22] *PYNQ-Z2 Reference Manual v1.0*. Technology Unlimited. Mai 2018. URL: https://dpoauwgwqsy2x.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf (sjekket 21.05.2022).

[23] Mats Randgaard. *Ekkoloddsystem realisert med System-on-Chip på FPGA*. Masteroppg. Universitetet i Oslo, 2010.

[24] Tole Sutikno. An Efficient Implementation of the Non Restoring Square Root Algorithm in Gate Level. I: *International Journal of Computer Theory and Engineering* (2011), s. 46–51. DOI: 10.7763/ijcte.2011.v3.281. URL: https://doi.org/10.7763/ijcte.2011.v3.281 (sjekket 21.05.2022).

[25] Alexander Sveinall. *Embedded Sonar med GPRS*. Masteroppg. Universitetet i Oslo, 2012.

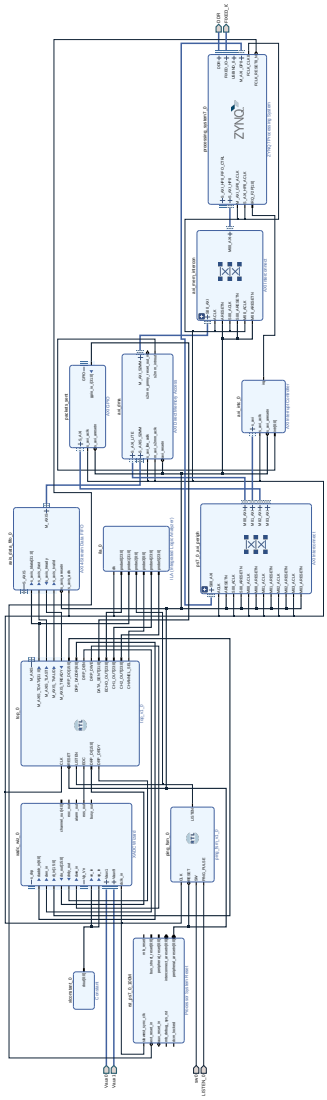
[26] *Unity-Gain Stable, Wideband Voltage Limiting Amplifier*. SBOS258D. Revidert Desember 2008. Texas Instruments. Nov. 2002. URL: https://www.ti.com/lit/ds/symlink/opa698.pdf?ts=1633513388584&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FOPA698 (sjekket 21.05.2022).

[27] Robert J. Urick. *Principles of Underwater sound*. eng. 3. utg. Peninsula Publishing, 2013 (opprinnelig 1983). ISBN: 0932146627.

[28] *Video*. URL: https://pynq.readthedocs.io/en/latest/pynq_libraries/video.html (sjekket 21.05.2022).

[29] *Vivado ML Overview*. Xilinx. URL: https://www.xilinx.com/products/design-tools/vivado.html (sjekket 22.05.2022).

A Blokkskjema fra Vivado



Figur 60: Blokkskjema over systemet hentet fra Vivado. Constant blokken er satt til 0 for å koble de dedikerte XADC inngangene til jord da disse ikke er i bruk.

B VHDL kode

Toppmodul (top.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity top is
6   generic(
7   -- AXI-STREAM generics
8     SKIP : integer := 0;
9     STREAM_WIDTH : integer := 32;
10    REST_DELAY : integer := 250;
11  -- DSP generics
12    DSP_DATA_WIDTH : integer := 24;
13    DSP_COEF_WIDTH : integer := 18
14  );
15  port (
16  -- Clock and Reset ports
17    CLK : in std_logic;
18    RESET : in std_logic;
19  -- Listen port
20    LISTEN : in std_logic;
21  -- ADC interface ports
22    EOC : in std_logic;
23    DRP_DI : in std_logic_vector(15 downto 0);
24    DRP_DRDY : in std_logic;
25    DRP_DO : out std_logic_vector(15 downto 0);
26    DRP_DADDR : out std_logic_vector(6 downto 0);
27    DRP_DEN : out std_logic;
28    DRP_DWE : out std_logic;
29  -- AXI-STREAM Master ports
30    M_AXIS_TREADY : in std_logic;
31    M_AXIS_TVALID : out std_logic;
32    M_AXIS_TLAST : out std_logic;
33    M_AXIS_TDATA : out std_logic_vector(STREAM_WIDTH-1 downto 0)
        ;
34  -- Number of data packets sent
35    DATA_SENT : out std_logic_vector(31 downto 0);
36  -- Test ports
37    ECHO_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
38    CH1_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
```

```
39      CH2_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
40      CHANNEL_SEL : out std_logic
41    );
42  end top;
43
44  architecture rtl of top is
45  -- Signal declarations
46    signal valid_i : std_logic;
47    signal tlast_i : std_logic;
48    signal ch_sel_i : std_logic;
49    signal adc_out_i : std_logic_vector(31 downto 0);
50    signal cd2hpf : signed(DSP_DATA_WIDTH-1 downto 0);
51    signal hpf2scale : signed(DSP_DATA_WIDTH-1 downto 0);
52    signal scale2ht : signed(DSP_DATA_WIDTH-1 downto 0);
53    signal ht2sqrt_re : signed(DSP_DATA_WIDTH-1 downto 0);
54    signal ht2sqrt_im : signed(DSP_DATA_WIDTH-1 downto 0);
55    signal ht2lpf : signed(DSP_DATA_WIDTH-1 downto 0);
56    signal mag2lpf : signed(DSP_DATA_WIDTH-1 downto 0);
57    signal lpf2stream : signed(DSP_DATA_WIDTH-1 downto 0);
58  -- Test signals
59  signal test_data : std_logic_vector(31 downto 0);
60  signal scale_ch1 : signed(DSP_DATA_WIDTH-1 downto 0);
61  signal scale_ch2 : signed(DSP_DATA_WIDTH-1 downto 0);
62
63
64  -- Component declarations
65
66    component DRP_IF is
67      port (
68        ADC_IN : in std_logic_vector(15 downto 0);
69        DRDY : in std_logic;
70        EOC : in std_logic;
71        CLK : in std_logic;
72        RESET : in std_logic;
73        VALID : out std_logic;
74        ADC_OUT : out std_logic_vector(15 downto 0);
75        DATA_OUT : out std_logic_vector(31 downto 0);
76        DADDR : out std_logic_vector(6 downto 0);
77        DEN : out std_logic;
78        DWE : out std_logic
79      );
80    end component;
81
```

```
82  component test_data_feeder is
83   port(
84    CLK : in std_logic;
85    RESET : in std_logic;
86    LISTEN : in std_logic;
87    TLAST : in std_logic;
88    VALID : out std_logic;
89    DATA : out std_logic_vector(31 downto 0)
90   );
91  end component;
92
93  component CD is
94   generic (
95    DATA_WIDTH : integer
96   );
97   port(
98    CLK : in std_logic;
99    RESET : in std_logic;
100   VALID : in std_logic;
101   DATA_IN : in std_logic_vector(31 downto 0);
102   CHANNEL : out std_logic;
103   DATA_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
104   -- Test ports
105   CH1_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
106   CH2_OUT : out signed(DSP_DATA_WIDTH-1 downto 0)
107  );
108 end component;
109
110 component FIR_HP is
111  generic (
112   DATA_WIDTH : integer;
113   COEF_WIDTH : integer
114  );
115  port (
116   CLK : in std_logic;
117   RESET : in std_logic;
118   VALID : in std_logic;
119   DATA_IN : in signed(DSP_DATA_WIDTH-1 downto 0);
120   DATA_OUT : out signed(DSP_DATA_WIDTH-1 downto 0)
121  );
122 end component;
123
124 component SCALER is
```

```
125     generic (
126       DATA_WIDTH : integer
127     );
128     port (
129       CLK : in std_logic;
130       RESET : in std_logic;
131       VALID : in std_logic;
132       CH_SEL : in std_logic;
133       DATA_IN : in signed(DSP_DATA_WIDTH-1 downto 0);
134       DATA_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
135       -- Test ports
136       CH_USED : out std_logic
137     );
138   end component;
139
140   component HT is
141     generic (
142       DATA_WIDTH : integer;
143       COEF_WIDTH : integer
144     );
145     port (
146       CLK : in std_logic;
147       RESET : in std_logic;
148       DATA_IN : in signed(DSP_DATA_WIDTH-1 downto 0);
149       VALID : in std_logic;
150       REAL_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
151       IMAG_OUT : out signed(DSP_DATA_WIDTH-1 downto 0);
152       ABS_OUT : out signed(DSP_DATA_WIDTH-1 downto 0)
153     );
154   end component;
155
156   component MAG is
157     generic (
158       DATA_WIDTH : integer
159     );
160     port (
161       CLK : in std_logic;
162       RESET : in std_logic;
163       VALID : in std_logic;
164       REAL_IN : in signed(DSP_DATA_WIDTH-1 downto 0);
165       IMAG_IN : in signed(DSP_DATA_WIDTH-1 downto 0);
166       DATA_OUT : out signed(DSP_DATA_WIDTH-1 downto 0)
167     );
```

```
168   end component;
169
170   component FIR_LP is
171     generic (
172       DATA_WIDTH : integer;
173       COEF_WIDTH : integer
174     );
175     port (
176       CLK : in std_logic;
177       RESET : in std_logic;
178       VALID : in std_logic;
179       DATA_IN : in signed(DSP_DATA_WIDTH-1 downto 0);
180       DATA_OUT : out signed(DSP_DATA_WIDTH-1 downto 0)
181     );
182   end component;
183
184   component M_AXIS_IF is
185     generic(
186       SKIP : integer;
187       STREAM_WIDTH : integer;
188       REST_DELAY : integer
189     );
190     port(
191       CLK : in std_logic;
192       RESET : in std_logic;
193       LISTEN : in std_logic;
194       VALID : in std_logic;
195       DATA_IN : in std_logic_vector(STREAM_WIDTH-1 downto 0);
196       DATA_SENT : out std_logic_vector(31 downto 0);
197       -- AXI-Stream ports
198       M_AXIS_TREADY : in std_logic;
199       M_AXIS_TVALID : out std_logic;
200       M_AXIS_TLAST : out std_logic;
201       M_AXIS_TDATA : out std_logic_vector(STREAM_WIDTH-1 downto
              0)
202     );
203   end component;
204
205 begin
206
207 -- Component port mapping
208
209   DRP_IF_inst : DRP_IF
```

```
210     port map(
211       ADC_IN => DRP_DI,
212       DRDY => DRP_DRDY,
213       EOC => EOC,
214       CLK => CLK,
215       RESET => RESET,
216       VALID => valid_i, -- For normal use
217   -- VALID => open, -- For testing with reconstructed data
218       ADC_OUT => DRP_DO,
219       DATA_OUT => adc_out_i,
220       DADDR => DRP_DADDR,
221       DEN => DRP_DEN,
222       DWE => DRP_DWE
223     );
224
225   test_data_feeder_inst :test_data_feeder
226     port map(
227       CLK => CLK,
228       RESET => RESET,
229       LISTEN => LISTEN,
230       TLAST => tlast_i,
231       VALID => open, -- For normal use
232   -- VALID => valid_i, -- For testing with reconstructed data
233       DATA => test_data
234     );
235
236   CD_inst : CD
237     generic map(
238       DATA_WIDTH => DSP_DATA_WIDTH
239     )
240     port map(
241       CLK => CLK,
242       RESET => RESET,
243       VALID => valid_i,
244       DATA_IN => adc_out_i, -- For normal use
245   -- DATA_IN => test_data, -- For testing with reconstructed data
246       CHANNEL => ch_sel_i,
247       DATA_OUT => cd2hpf,
248       CH1_OUT => scale_ch1,
249       CH2_OUT => scale_ch2
250     );
251
252   HPF_inst : FIR_HP
```

```
253     generic map(
254       DATA_WIDTH => DSP_DATA_WIDTH,
255       COEF_WIDTH => DSP_COEF_WIDTH
256     )
257     port map(
258       CLK => CLK,
259       RESET => RESET,
260       VALID => valid_i,
261       DATA_IN => cd2hpf,
262       DATA_OUT => hpf2scale
263     );
264
265   SCALER_inst : SCALER
266     generic map(
267       DATA_WIDTH => DSP_DATA_WIDTH
268     )
269     port map(
270       CLK => CLK,
271       RESET => RESET,
272       VALID => valid_i,
273       CH_SEL => ch_sel_i,
274       DATA_IN => hpf2scale,
275       DATA_OUT => scale2ht,
276       CH_USED => CHANNEL_SEL
277     );
278
279   HT_inst : HT
280   generic map(
281       DATA_WIDTH => DSP_DATA_WIDTH,
282       COEF_WIDTH => DSP_COEF_WIDTH
283     )
284     port map(
285       CLK => CLK,
286       RESET => RESET,
287       DATA_IN => scale2ht,
288       VALID => valid_i,
289       REAL_OUT => ht2sqrt_re,
290       IMAG_OUT => ht2sqrt_im,
291       ABS_OUT => ht2lpf
292     );
293
294   MAG_inst : MAG
295     generic map(
```

```
296       DATA_WIDTH => DSP_DATA_WIDTH
297     )
298     port map(
299       CLK => CLK,
300       RESET => RESET,
301       VALID => valid_i,
302       REAL_IN => ht2sqrt_re,
303       IMAG_IN => ht2sqrt_im,
304       DATA_OUT => mag2lpf
305     );
306
307   LPF_inst : FIR_LP
308     generic map(
309       DATA_WIDTH => DSP_DATA_WIDTH,
310       COEF_WIDTH => DSP_COEF_WIDTH
311     )
312     port map(
313       CLK => CLK,
314       RESET => RESET,
315       VALID => valid_i,
316       DATA_IN => mag2lpf,
317       DATA_OUT => lpf2stream
318     );
319
320   M_AXIS_inst : M_AXIS_IF
321     generic map(
322       SKIP => SKIP,
323       STREAM_WIDTH => STREAM_WIDTH,
324       REST_DELAY => REST_DELAY
325     )
326     port map(
327       CLK => CLK,
328       RESET => RESET,
329       LISTEN => LISTEN,
330       VALID => valid_i,
331       DATA_IN => std_logic_vector(resize(lpf2stream, STREAM_WIDTH
              )),
332       DATA_SENT => DATA_SENT,
333       M_AXIS_TREADY => M_AXIS_TREADY,
334       M_AXIS_TVALID => M_AXIS_TVALID,
335       M_AXIS_TLAST => tlast_i,
336       M_AXIS_TDATA => M_AXIS_TDATA
337     );
```

```
338
339   M_AXIS_TLAST <= tlast_i;
340
341 -- Test signal assignments
342
343   ECHO_OUT <= scale2ht;
344   CH1_OUT <= scale_ch1; --For channel selector testing
345   CH2_OUT <= scale_ch2; --For channel selector testing
346 -- CH1_OUT <= ht2sqrt_re; --For envelope extractor testing
347 -- CH2_OUT <= ht2sqrt_im; --For envelope extractor testing
348
349
350 end rtl;
```

Modul for synkronisering av LISTEN signal (ping_fsm.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity ping_fsm is
6    port(
7    CLK : in std_logic;
8    RESET : in std_logic;
9    SW : in std_logic;
10   PING_PULSE : in std_logic;
11   LISTEN : out std_logic
12   );
13 end ping_fsm;
14
15 architecture rtl of ping_fsm is
16
17   type t_state is (IDLE, PING);
18   signal state : t_state;
19
20 begin
21
22   FSM : process(state, CLK, RESET, SW, PING_PULSE) is
23   begin
24     if (RESET = '0') then
25       LISTEN <= '0';
26       state <= IDLE;
27     elsif rising_edge(CLK) then
28       case state is
```

```
29        when IDLE =>
30          if (SW = '1') and (PING_PULSE = '0') then
31            state <= PING;
32          end if;
33        when PING =>
34            LISTEN <= PING_PULSE;
35          if (SW = '0') and (PING_PULSE = '0') then
36            state <= IDLE;
37            LISTEN <= '0';
38          end if;
39      end case;
40    end if;
41  end process;
42
43 end rtl;
```

Modul for kommunikasjon over DRP (drp_if.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity DRP_IF is
6    port (
7      ADC_IN : in std_logic_vector(15 downto 0);
8      DRDY : in std_logic;
9      EOC : in std_logic;
10     CLK : in std_logic;
11     RESET : in std_logic;
12     VALID : out std_logic;
13     ADC_OUT : out std_logic_vector(15 downto 0);
14     DATA_OUT : out std_logic_vector(31 downto 0);
15     DADDR : out std_logic_vector(6 downto 0);
16     DEN : out std_logic;
17     DWE : out std_logic
18   );
19 end DRP_IF;
20
21 architecture rtl of DRP_IF is
22
23   type state_type is (IDLE, READ_CH1, READ_CH2);
24   signal state : state_type;
25   signal data1, data2 : std_logic_vector(15 downto 0);
26
```

```
27  begin
28
29    DRP_READ_FSM : process(state, CLK, RESET, EOC, DRDY, ADC_IN)
          is
30    begin
31      if (RESET = '0') then
32        state <= IDLE;
33        VALID <= '0';
34        DEN <= '0';
35        data1 <= (others => '0');
36        data2 <= (others => '0');
37      elsif rising_edge(CLK) then
38        case state is
39          when IDLE =>
40            VALID <= '0';
41            DADDR <= "0010001"; -- Address for Vaux1
42            if (EOC = '1') then
43              DEN <= '1';
44              state <= READ_CH1;
45            end if;
46
47          when READ_CH1 =>
48            DEN <= '0';
49            DADDR <= "0011001"; -- Address for Vaux9
50            if (DRDY = '1') then
51              data1 <= ADC_IN;
52              DEN <= '1';
53              state <= READ_CH2;
54            end if;
55
56          when READ_CH2 =>
57            DEN <= '0';
58            if (DRDY = '1') then
59              data2 <= ADC_IN;
60              VALID <= '1';
61              state <= IDLE;
62            end if;
63        end case;
64      end if;
65    end process DRP_READ_FSM;
66
67    DATA_OUT <= data1 & data2;
68
```

```
69 end rtl;
```

Modul for å detektere metning (clip_detect.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity CD is
6    generic (
7      DATA_WIDTH : integer
8    );
9    port(
10     CLK : in std_logic;
11     RESET : in std_logic;
12     VALID : in std_logic;
13     DATA_IN : in std_logic_vector(31 downto 0);
14     CHANNEL : out std_logic;
15     DATA_OUT : out signed(DATA_WIDTH-1 downto 0);
16     -- Test ports
17     CH1_OUT : out signed(DATA_WIDTH-1 downto 0);
18     CH2_OUT : out signed(DATA_WIDTH-1 downto 0)
19   );
20 end CD;
21
22 architecture rtl of CD is
23   signal data_ch1, data_ch2 : signed(DATA_WIDTH-1 downto 0);
24   signal channel_sel : std_logic;
25 begin
26   clip_detect : process (RESET, CLK, VALID) is
27   begin
28     if (RESET = '0') then
29       channel_sel <= '0';
30       data_ch1 <= (others => '0');
31       data_ch2 <= (others => '0');
32     elsif rising_edge(CLK) and (VALID = '1') then
33       if (unsigned(DATA_IN(31 downto 20)) >= x"FFF"
34       or unsigned(DATA_IN(31 downto 20)) <= x"000") then
35         channel_sel <= '1';
36       else
37         channel_sel <= '0';
38       end if;
39       data_ch1 <= signed(resize(unsigned(DATA_IN(31 downto 20)),
           DATA_WIDTH));
```

```
40      data_ch2 <= signed(resize(unsigned(DATA_IN(15 downto 4)),
            DATA_WIDTH));
41    end if;
42  end process clip_detect;
43
44  DATA_OUT <= data_ch1 when channel_sel = '0' else
45            data_ch2 when channel_sel = '1';
46  CHANNEL <= channel_sel;
47  CH1_OUT <= data_ch1;
48  CH2_OUT <= data_ch2;
49
50 end rtl;
```

FIR Høypassfilter (fir_hpf.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.user_types_pkg.all;
7  use work.coeff_pkg.all;
8
9  entity FIR_HP is
10   generic (
11     DATA_WIDTH : integer;
12     COEF_WIDTH : integer
13   );
14   port (
15   CLK : in std_logic;
16   RESET : in std_logic;
17   VALID : in std_logic;
18   DATA_IN : in signed(DATA_WIDTH-1 downto 0);
19   DATA_OUT : out signed(DATA_WIDTH-1 downto 0)
20   );
21 end FIR_HP;
22
23
24 architecture rtl of FIR_HP is
25   -- Delay registers
26   signal shift_reg : signed_array_type(0 to 52) (DATA_WIDTH-1
         downto 0);
27
28   -- Array to reduce number of mult_piplications
```

```
29    signal half_reg : signed_array_type(0 to 26) (DATA_WIDTH-1
          downto 0);
30
31    -- Coefficients
32    signal coeff : signed_array_type(0 to 26) (COEF_WIDTH-1 downto
           0) := hp_coeff;
33
34    -- mult_piplication result registers
35    signal mult_p : signed_array_type(0 to 26) (DATA_WIDTH+
          COEF_WIDTH-1 downto 0);
36    signal mult_m : signed_array_type(0 to 26) (DATA_WIDTH+
          COEF_WIDTH-1 downto 0);
37
38    -- Sum and temporary sums
39    signal tmpsum : signed_array_type(0 to 26) (DATA_WIDTH+
          COEF_WIDTH-1 downto 0);
40    signal sum : signed(DATA_WIDTH+COEF_WIDTH-1 downto 0);
41
42  begin
43
44    filter : process(CLK, RESET, VALID) is
45    begin
46      if rising_edge(CLK) then
47        if (RESET = '0') then
48          shift_reg <= (others => (others => '0'));
49          half_reg <= (others => (others => '0'));
50          mult_p <= (others => (others => '0'));
51          mult_m <= (others => (others => '0'));
52          tmpsum <= (others => (others => '0'));
53          sum <= (others => '0');
54        else
55          if (VALID = '1') then
56            shift_reg <= DATA_IN & shift_reg(0 to shift_reg'high-1);
57          end if;
58          for i in 0 to 25 loop
59            half_reg(i) <= shift_reg(i) + shift_reg(shift_reg'high -
                  i);
60            mult_m(i) <= half_reg(i) * coeff(i);
61          end loop;
62          half_reg(26) <= shift_reg(26);
63          mult_m(26) <= half_reg(26) * coeff(26);
64          mult_p <= mult_m;
65          --First Cycle
```

```
66          tmpsum(0) <= mult_p(0) + mult_p(1);
67          tmpsum(1) <= mult_p(2) + mult_p(3);
68          tmpsum(2) <= mult_p(4) + mult_p(5);
69          tmpsum(3) <= mult_p(6) + mult_p(7);
70          tmpsum(4) <= mult_p(8) + mult_p(9);
71          tmpsum(5) <= mult_p(10) + mult_p(11);
72          tmpsum(6) <= mult_p(12) + mult_p(13);
73          tmpsum(7) <= mult_p(14) + mult_p(15);
74          tmpsum(8) <= mult_p(16) + mult_p(17);
75          tmpsum(9) <= mult_p(18) + mult_p(19);
76          tmpsum(10) <= mult_p(20) + mult_p(21);
77          tmpsum(11) <= mult_p(22) + mult_p(23);
78          tmpsum(12) <= mult_p(24) + mult_p(25);
79          tmpsum(13) <= mult_p(26);
80          --Second Cycle
81          tmpsum(14) <= tmpsum(0) + tmpsum(1);
82          tmpsum(15) <= tmpsum(2) + tmpsum(3);
83          tmpsum(16) <= tmpsum(4) + tmpsum(5);
84          tmpsum(17) <= tmpsum(6) + tmpsum(7);
85          tmpsum(18) <= tmpsum(8) + tmpsum(9);
86          tmpsum(19) <= tmpsum(10) + tmpsum(11);
87          tmpsum(20) <= tmpsum(12) + tmpsum(13);
88          --Third Cycle
89          tmpsum(21) <= tmpsum(14) + tmpsum(15);
90          tmpsum(22) <= tmpsum(16) + tmpsum(17);
91          tmpsum(23) <= tmpsum(18) + tmpsum(19);
92          tmpsum(24) <= tmpsum(20);
93          --Fourth Cycle
94          tmpsum(25) <= tmpsum(21) + tmpsum(22);
95          tmpsum(26) <= tmpsum(23) + tmpsum(24);
96          --Fifth Cycle
97          sum <= tmpsum(25) + tmpsum(26);
98        end if;
99      end if;
100   end process filter;
101
102   DATA_OUT <= sum(DATA_WIDTH+COEF_WIDTH-2 downto COEF_WIDTH-1);
103
104 end rtl;
```

Modul for å skalere data (scale.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
```

```
3  use ieee.numeric_std.all;
4
5  entity SCALER is
6    generic (
7      DATA_WIDTH : integer
8    );
9    port (
10   CLK : in std_logic;
11   RESET : in std_logic;
12   VALID : in std_logic;
13   CH_SEL : in std_logic;
14   DATA_IN : in signed(DATA_WIDTH-1 downto 0);
15   DATA_OUT : out signed(DATA_WIDTH-1 downto 0);
16   -- Test ports
17   CH_USED : out std_logic
18   );
19 end SCALER;
20
21 architecture rtl of SCALER is
22
23   signal shift_reg_group : std_logic_vector(26 downto 0);
24   signal shift_reg_pipe : std_logic_vector(7 downto 0);
25
26 begin
27
28   delay : process(CLK, RESET, VALID) is
29   begin
30     if (RESET = '0') then
31       shift_reg_group <= (others => '0');
32       shift_reg_pipe <= (others => '0');
33     elsif rising_edge(CLK) then
34       -- Group delay
35       if (VALID = '1') then
36         shift_reg_group <= CH_SEL & shift_reg_group(26 downto 1);
37       end if;
38       shift_reg_pipe <= shift_reg_group(0) & shift_reg_pipe(7
           downto 1);
39     end if;
40   end process delay;
41
42   DATA_OUT <= DATA_IN when shift_reg_pipe(0) = '0' else
43             resize(DATA_IN*x"32", DATA_WIDTH) when shift_reg_pipe
                   (0) = '1'; -- 50 gain between channels
```

```
-- shift_left(DATA_IN, 1) when shift_reg_pipe(0) = '1';
  CH_USED <= shift_reg_pipe(0);

end rtl;
```

FIR Hilbert filter (hilb_tran.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.user_types_pkg.all;
use work.coeff_pkg.all;

entity HT is
  generic (
  DATA_WIDTH : integer;
  COEF_WIDTH : integer
  );
  port (
  CLK : in std_logic;
  RESET : in std_logic;
  DATA_IN : in signed(DATA_WIDTH-1 downto 0);
  VALID : in std_logic;
  REAL_OUT : out signed(DATA_WIDTH-1 downto 0);
  IMAG_OUT : out signed(DATA_WIDTH-1 downto 0);
  ABS_OUT : out signed(DATA_WIDTH-1 downto 0)
  );
end HT;


architecture rtl of HT is
  -- Delay registers
  signal shift_reg : signed_array_type(0 to 46) (DATA_WIDTH-1
      downto 0);

  -- Coefficients
  signal coeff : signed_array_type(0 to 23) (COEF_WIDTH-1 downto
       0) := ht_coeff;

  -- Multiplication result registers
  signal mult_p : signed_array_type(0 to 23) (DATA_WIDTH+
      COEF_WIDTH-1 downto 0);
```

```
35   signal mult_m : signed_array_type(0 to 23) (DATA_WIDTH+
         COEF_WIDTH-1 downto 0);
36
37   -- Temporary sums and delay registers
38   signal tmpsum : signed_array_type(0 to 23) (DATA_WIDTH+
         COEF_WIDTH-1 downto 0);
39
40   signal delay_real : signed_array_type(0 to 6) (DATA_WIDTH-1
         downto 0);
41
42   -- Real and Imaginary
43   signal signl : signed(DATA_WIDTH-1 downto 0);
44   signal imag : signed(DATA_WIDTH+COEF_WIDTH-1 downto 0);
45   signal env : signed(DATA_WIDTH+COEF_WIDTH-1 downto 0);
46
47 begin
48
49   filter : process(CLK, RESET, VALID) is
50   begin
51     if rising_edge(CLK) then
52       if (RESET = '0') then
53         shift_reg <= (others => (others => '0'));
54         mult_p <= (others => (others => '0'));
55         mult_m <= (others => (others => '0'));
56         tmpsum <= (others => (others => '0'));
57         imag <= (others => '0');
58         signl <= (others => '0');
59         delay_real <= (others => (others => '0'));
60       else
61         if (VALID = '1') then
62           shift_reg <= DATA_IN & shift_reg(0 to shift_reg'high-1);
63         end if;
64         for i in 0 to 23 loop
65           mult_m(i) <= shift_reg(2*i) * coeff(i);
66         end loop;
67         mult_p <= mult_m;
68         --First Cycle
69         tmpsum(0) <= mult_p(0) + mult_p(1);
70         tmpsum(1) <= mult_p(2) + mult_p(3);
71         tmpsum(2) <= mult_p(4) + mult_p(5);
72         tmpsum(3) <= mult_p(6) + mult_p(7);
73         tmpsum(4) <= mult_p(8) + mult_p(9);
74         tmpsum(5) <= mult_p(10) + mult_p(11);
```

```
75          tmpsum(6) <= mult_p(12) + mult_p(13);
76          tmpsum(7) <= mult_p(14) + mult_p(15);
77          tmpsum(8) <= mult_p(16) + mult_p(17);
78          tmpsum(9) <= mult_p(18) + mult_p(19);
79          tmpsum(10) <= mult_p(20) + mult_p(21);
80          tmpsum(11) <= mult_p(22) + mult_p(23);
81          --Second Cycle
82          tmpsum(12) <= tmpsum(0) + tmpsum(1);
83          tmpsum(13) <= tmpsum(2) + tmpsum(3);
84          tmpsum(14) <= tmpsum(4) + tmpsum(5);
85          tmpsum(15) <= tmpsum(6) + tmpsum(7);
86          tmpsum(16) <= tmpsum(8) + tmpsum(9);
87          tmpsum(17) <= tmpsum(10) + tmpsum(11);
88          --Third Cycle
89          tmpsum(18) <= tmpsum(13) + tmpsum(14);
90          tmpsum(19) <= tmpsum(15) + tmpsum(16);
91          tmpsum(20) <= tmpsum(17);
92          --Fourth Cycle
93          tmpsum(21) <= tmpsum(18) + tmpsum(19);
94          tmpsum(22) <= tmpsum(20);
95          --Fifth Cycle
96          imag <= tmpsum(21) + tmpsum(22);
97          -- Delay real part by six cycles
98          delay_real <= shift_reg(23) & delay_real(0 to delay_real'
                high-1);
99          signl <= delay_real(delay_real'high);
100       end if;
101     end if;
102   end process filter;
103
104   REAL_OUT <= signl;
105   IMAG_OUT <= imag(DATA_WIDTH+COEF_WIDTH-2 downto COEF_WIDTH-1);
106   ABS_OUT <= abs(signl) + abs(imag(DATA_WIDTH+COEF_WIDTH-2
          downto COEF_WIDTH-1));
107
108 end rtl;
```

Modul for å kalkulere magnitude (mag_fsm.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity MAG is
```

```
6   generic (
7     DATA_WIDTH : integer
8   );
9   port (
10    CLK : in std_logic;
11    RESET : in std_logic;
12    VALID : in std_logic;
13    REAL_IN : in signed(DATA_WIDTH-1 downto 0);
14    IMAG_IN : in signed(DATA_WIDTH-1 downto 0);
15    DATA_OUT : out signed(DATA_WIDTH-1 downto 0)
16  );
17 end MAG;
18
19
20 architecture rtl of MAG is
21
22  type state_type is (IDLE, SUM, FIRST_PAIR, ITER_PAIR, DONE);
23  signal state : state_type;
24  signal real_sqrd: signed(2*DATA_WIDTH-1 downto 0);
25  signal imag_sqrd : signed(2*DATA_WIDTH-1 downto 0);
26  signal comb_sqrd : signed(2*DATA_WIDTH-1 downto 0);
27  signal lft : signed(2*DATA_WIDTH-1 downto 0);
28  signal rgt : signed(2*DATA_WIDTH-1 downto 0);
29  signal ans : signed(DATA_WIDTH-1 downto 0);
30  signal idx : integer range 0 to 2*DATA_WIDTH;
31
32 begin
33  calculate : process(CLK, RESET, VALID) is
34    variable diff : signed(2*DATA_WIDTH-1 downto 0);
35  begin
36    if rising_edge(CLK) then
37      if (RESET = '0') then
38        real_sqrd <= (others => '0');
39        imag_sqrd <= (others => '0');
40        comb_sqrd <= (others => '0');
41        rgt <= (others => '0');
42        lft <= (others => '0');
43        ans <= (others => '0');
44        idx <= 2*DATA_WIDTH;
45        diff := (others => '0');
46        state <= IDLE;
47      else
48        case state is
```

```
49          when IDLE =>
50            if (VALID = '1') then
51              real_sqrd <= REAL_IN*REAL_IN;
52              imag_sqrd <= IMAG_IN*IMAG_IN;
53              state <= SUM;
54            end if;
55
56          when SUM =>
57            comb_sqrd <= real_sqrd + imag_sqrd;
58            state <= FIRST_PAIR;
59
60          when FIRST_PAIR =>
61            rgt(1 downto 0) <= "01";
62            lft(1 downto 0) <= comb_sqrd(idx-1 downto idx-2);
63            state <= ITER_PAIR;
64
65          when ITER_PAIR =>
66            idx <= idx - 2;
67            diff := lft - rgt;
68            if (diff(2*DATA_WIDTH-1) = '0') then
69              rgt <= rgt(2*DATA_WIDTH-2 downto 2) & "101";
70              if (idx = 0) then
71                state <= DONE;
72              else
73                lft <= diff(2*DATA_WIDTH-3 downto 0) & comb_sqrd(
                      idx-1 downto idx-2);
74              end if;
75            elsif (diff(2*DATA_WIDTH-1) = '1') then
76              rgt <= rgt(2*DATA_WIDTH-2 downto 2) & "001";
77              if (idx = 0) then
78                state <= DONE;
79              else
80                lft <= lft(2*DATA_WIDTH-3 downto 0) & comb_sqrd(idx
                      -1 downto idx-2);
81              end if;
82            end if;
83          when DONE =>
84            ans <= rgt(DATA_WIDTH+1 downto 2);
85            rgt <= (others => '0');
86            lft <= (others => '0');
87            idx <= 2*DATA_WIDTH;
88            state <= IDLE;
89        end case;
```

```
90      end if;
91    end if;
92  end process calculate;
93
94  DATA_OUT <= ans;
95
96 end rtl;
```

FIR Lavpassfilter (fir_lpf.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.user_types_pkg.all;
7  use work.coeff_pkg.all;
8
9  entity FIR_LP is
10   generic (
11     DATA_WIDTH : integer;
12     COEF_WIDTH : integer
13   );
14   port (
15   CLK : in std_logic;
16   RESET : in std_logic;
17   VALID : in std_logic;
18   DATA_IN : in signed(DATA_WIDTH-1 downto 0);
19   DATA_OUT : out signed(DATA_WIDTH-1 downto 0)
20   );
21 end FIR_LP;
22
23
24 architecture rtl of FIR_LP is
25   -- Delay registers
26   signal shift_reg : signed_array_type(0 to 52) (DATA_WIDTH-1
         downto 0);
27
28   -- Array to reduce number of multiplications
29   signal half_reg : signed_array_type(0 to 26) (DATA_WIDTH-1
         downto 0);
30
31   -- Coefficients
32   signal coeff : signed_array_type(0 to 26) (COEF_WIDTH-1 downto
```

```
        0) := lp_coeff;
33
34    -- Multiplication result registers
35    signal mult_p : signed_array_type(0 to 26) (DATA_WIDTH+
          COEF_WIDTH-1 downto 0);
36    signal mult_m : signed_array_type(0 to 26) (DATA_WIDTH+
          COEF_WIDTH-1 downto 0);
37
38    -- Sum and temporary sums
39    signal tmpsum : signed_array_type(0 to 26) (DATA_WIDTH+
          COEF_WIDTH-1 downto 0);
40    signal sum : signed(DATA_WIDTH+COEF_WIDTH-1 downto 0);
41
42  begin
43
44    filter : process(CLK, RESET, VALID) is
45    begin
46      if rising_edge(CLK) then
47        if (RESET = '0') then
48          shift_reg <= (others => (others => '0'));
49          half_reg <= (others => (others => '0'));
50          mult_p <= (others => (others => '0'));
51          mult_m <= (others => (others => '0'));
52          tmpsum <= (others => (others => '0'));
53          sum <= (others => '0');
54        else
55          if (VALID = '1') then
56            shift_reg <= DATA_IN & shift_reg(0 to shift_reg'high-1);
57          end if;
58          for i in 0 to 25 loop
59            half_reg(i) <= shift_reg(i) + shift_reg(shift_reg'high -
                  i);
60            mult_m(i) <= half_reg(i) * coeff(i);
61          end loop;
62          half_reg(26) <= shift_reg(26);
63          mult_m(26) <= half_reg(26) * coeff(26);
64          mult_p <= mult_m;
65          --First Cycle
66          tmpsum(0) <= mult_p(0) + mult_p(1);
67          tmpsum(1) <= mult_p(2) + mult_p(3);
68          tmpsum(2) <= mult_p(4) + mult_p(5);
69          tmpsum(3) <= mult_p(6) + mult_p(7);
70          tmpsum(4) <= mult_p(8) + mult_p(9);
```

```
71          tmpsum(5) <= mult_p(10) + mult_p(11);
72          tmpsum(6) <= mult_p(12) + mult_p(13);
73          tmpsum(7) <= mult_p(14) + mult_p(15);
74          tmpsum(8) <= mult_p(16) + mult_p(17);
75          tmpsum(9) <= mult_p(18) + mult_p(19);
76          tmpsum(10) <= mult_p(20) + mult_p(21);
77          tmpsum(11) <= mult_p(22) + mult_p(23);
78          tmpsum(12) <= mult_p(24) + mult_p(25);
79          tmpsum(13) <= mult_p(26);
80          --Second Cycle
81          tmpsum(14) <= tmpsum(0) + tmpsum(1);
82          tmpsum(15) <= tmpsum(2) + tmpsum(3);
83          tmpsum(16) <= tmpsum(4) + tmpsum(5);
84          tmpsum(17) <= tmpsum(6) + tmpsum(7);
85          tmpsum(18) <= tmpsum(8) + tmpsum(9);
86          tmpsum(19) <= tmpsum(10) + tmpsum(11);
87          tmpsum(20) <= tmpsum(12) + tmpsum(13);
88          --Third Cycle
89          tmpsum(21) <= tmpsum(14) + tmpsum(15);
90          tmpsum(22) <= tmpsum(16) + tmpsum(17);
91          tmpsum(23) <= tmpsum(18) + tmpsum(19);
92          tmpsum(24) <= tmpsum(20);
93          --Fourth Cycle
94          tmpsum(25) <= tmpsum(21) + tmpsum(22);
95          tmpsum(26) <= tmpsum(23) + tmpsum(24);
96          --Fifth Cycle
97          sum <= tmpsum(25) + tmpsum(26);
98        end if;
99      end if;
100   end process filter;
101
102   DATA_OUT <= sum(DATA_WIDTH+COEF_WIDTH-2 downto COEF_WIDTH-1);
103
104 end rtl;
```

Modul for sending over AXI4-Stream (m_axis_if.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity M_AXIS_IF is
6    generic(
7      SKIP : integer;
```

```
8     STREAM_WIDTH : integer;
9     REST_DELAY : integer
10   );
11   port(
12   CLK : in std_logic;
13   RESET : in std_logic;
14   LISTEN : in std_logic;
15   VALID : in std_logic;
16   DATA_IN : in std_logic_vector(STREAM_WIDTH-1 downto 0);
17   DATA_SENT : out std_logic_vector(31 downto 0);
18   -- AXI-Stream ports
19   M_AXIS_TREADY : in std_logic;
20   M_AXIS_TVALID : out std_logic;
21   M_AXIS_TLAST : out std_logic;
22   M_AXIS_TDATA : out std_logic_vector(STREAM_WIDTH-1 downto 0)
23   );
24 end M_AXIS_IF;
25
26 architecture rtl of M_AXIS_IF is
27
28   type t_state is (IDLE, WAIT_TO_SEND, SEND);
29   signal state : t_state;
30   signal skip_cnt : integer range 0 to SKIP + 1;
31   signal rest_cnt : integer range 0 to REST_DELAY;
32   signal rest_cnt_next : integer range 0 to REST_DELAY;
33   signal data_cnt : unsigned(31 downto 0);
34   signal data_cnt_next : unsigned(31 downto 0);
35
36 begin
37
38   MAXIS_FSM : process(state, CLK, RESET,
39              M_AXIS_TREADY, VALID,
40              DATA_IN) is
41   begin
42     if (RESET = '0') then
43       skip_cnt <= 0;
44       rest_cnt <= 0;
45       data_cnt <= (others => '0');
46       DATA_SENT <= (others => '0');
47       M_AXIS_TLAST <= '0';
48       M_AXIS_TDATA <= (others => '0');
49       state <= IDLE;
50     elsif rising_edge(CLK) then
```

```
51      case state is
52       when IDLE =>
53         rest_cnt <= 0;
54         data_cnt <= (others => '0');
55         M_AXIS_TLAST <= '0';
56         M_AXIS_TVALID <= '0';
57         if (LISTEN = '1') then
58           state <= WAIT_TO_SEND;
59         end if;
60       when WAIT_TO_SEND =>
61         M_AXIS_TLAST <= '0';
62         M_AXIS_TVALID <= '0';
63         if (VALID = '1') then
64           skip_cnt <= skip_cnt + 1;
65         end if;
66         if (skip_cnt = SKIP + 1) then
67           skip_cnt <= 0;
68           state <= SEND;
69         end if;
70       when SEND =>
71         M_AXIS_TVALID <= '1';
72         M_AXIS_TDATA <= DATA_IN;
73         if (rest_cnt = REST_DELAY-1) then
74           M_AXIS_TLAST <= '1';
75         end if;
76         if (M_AXIS_TREADY = '1') and (rest_cnt = REST_DELAY-1)
               then
77           state <= IDLE;
78           DATA_SENT <= std_logic_vector(data_cnt);
79         elsif (M_AXIS_TREADY = '1') and not (rest_cnt =
               REST_DELAY-1) then
80           state <= WAIT_TO_SEND;
81           data_cnt <= data_cnt_next;
82           rest_cnt <= rest_cnt_next;
83         end if;
84       end case;
85     end if;
86   end process MAXIS_FSM;
87
88   data_cnt_next <= (others => '0') when (state = IDLE) else
89                data_cnt + 1;
90   rest_cnt_next <= 0 when (rest_cnt = REST_DELAY) or (state =
         IDLE) else
```

```
91                rest_cnt + 1 when (LISTEN = '0') else
92                rest_cnt;
93  end rtl;
```

Testmoduler og egendefinerte pakker

Modul for sending av rekonstruert signal (data_feeder.vhd)

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   library work;
6   use work.user_types_pkg.all;
7   use work.test_data_pkg.all;
8
9   entity test_data_feeder is
10   port(
11   CLK : in std_logic;
12   RESET : in std_logic;
13   LISTEN : in std_logic;
14   TLAST : in std_logic;
15   VALID : out std_logic;
16   DATA : out std_logic_vector(31 downto 0)
17   );
18  end test_data_feeder;
19
20  architecture rtl of test_data_feeder is
21
22   type state_type is (IDLE, SEND);
23   signal state : state_type;
24   signal data_arr : slv_array_type(0 to 13145) (31 downto 0) :=
         data_full;
25   signal clk_cnt : unsigned(7 downto 0);
26   signal clk_cnt_next : unsigned(7 downto 0);
27   signal index : integer range 0 to 13145;
28  begin
29
30   feed : process(CLK, RESET) is
31   begin
32     if (RESET = '0') then
33       VALID <= '0';
34       DATA <= (others => '0');
35       clk_cnt <= (others => '0');
```

```
36      index <= 0;
37    elsif rising_edge(CLK) then
38      VALID <= '0';
39      clk_cnt <= clk_cnt_next;
40      case state is
41        when IDLE =>
42          DATA <= x"80008000"; -- Set IDLE input to DC (half of
                maximum output)
43          if (clk_cnt = x"63") then
44            VALID <= '1';
45          end if;
46          if (LISTEN = '1') then
47            state <= SEND;
48          end if;
49        when SEND =>
50          DATA <= data_arr(index);
51          if (clk_cnt = x"63") then
52            VALID <= '1';
53            index <= index + 1;
54          end if;
55          if (TLAST = '1') then
56            state <= IDLE;
57            index <= 0;
58          end if;
59      end case;
60    end if;
61  end process;
62
63 clk_cnt_next <= (others => '0') when (clk_cnt = x"63") else
64              clk_cnt + 1;
65
66 end rtl;
```

Pakke med array-typer (user_types_pkg.vhd)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  package user_types_pkg is
6
7    type signed_array_type is array (natural range <>) of signed;
8    type unsigned_array_type is array (natural range <>) of
         unsigned;
```

```
9   type slv_array_type is array (natural range <>) of
        std_logic_vector;
10
11  end package user_types_pkg;
```

Pakke med filterkoeffisienter (coeff_pkg.vhd)

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   library work;
6   use work.user_types_pkg.all;
7
8   package coeff_pkg is
9
10    constant ht_coeff : signed_array_type(0 to 23) (17 downto 0)
          := ("111111111111101011", "111111111110011110",
11    "111111111011110110", "111111110110111101",
12    "111111101110100011", "111111100000111111",
13    "111111001011111100", "111110101011101101",
14    "111101111001001101", "111100100010100100",
15    "111001100110000101", "101011110000101000",
16    "010100001111011000", "000110011001111011",
17    "000011011101011100", "000010000110110011",
18    "000001010100010011", "000000110100000100",
19    "000000011111000001", "000000010001011101",
20    "000000001001000011", "000000000100001010",
21    "000000000001100010", "000000000000010101");
22
23    constant lp_coeff : signed_array_type(0 to 26) (17 downto 0)
          := ("000000000000010101", "000000000000101110",
24    "000000000001010110", "000000000010000111",
25    "000000000010111001", "000000000011011011",
26    "000000000011011000", "000000000010011010",
27    "000000000000001100", "111111111100100101",
28    "111111110111101001", "111111110001110111",
29    "111111101100000100", "111111100111011101",
30    "111111100101100001", "111111100111110011",
31    "111111101111101110", "111111111110001111",
32    "000000010011101011", "000000101111011111",
33    "000001010000010001", "000001110011101111",
34    "000010010111000001", "000010110110111010",
35    "000011010000010101", "000011100000101111",
```

```
36    "0000111001100011001");
37
38    constant hp_coeff : signed_array_type(0 to 26) (17 downto 0)
          := ("000000000000100011", "111111111110111100",
39    "111111111110100000", "111111111101101000",
40    "111111111100011000", "111111111010110000",
41    "111111111000101101", "111111110110001110",
42    "111111110011010000", "111111101111110100",
43    "111111101011111010", "111111100111100010",
44    "111111100010101111", "111111011101100101",
45    "111111011000001001", "111111010010100000",
46    "111111001100110000", "111111000111000010",
47    "111111000001011101", "111110111100001010",
48    "111110110111001111", "111110110010110110",
49    "111110101111000101", "111110101100000010",
50    "111110101001110011", "111110101000011100",
51    "011110100111111111");
52
53  end package coeff_pkg;
```

Pakke med rekonstruert data (test_data_pkg.vhd)

Denne pakken ble lagt ved som eksternt vedlegg på grunn av størrelse.

C Python kode

Python driver for filoverføring (file_io_driver.py)

```
1   import pynq
2   import asyncio
3   import os
4   import sys
5   import numpy as np
6   import matplotlib.pyplot as plt
7   from IPython.display import display, clear_output
8   from datetime import datetime
9   import time
10
11  #Program FPGA
12  overlay = pynq.Overlay('/home/xilinx/pynq/overlays/Milepel1/
        sys_test1.bit')
13
14  class DataTransfer():
```

```
15      def __init__(self, ol):
16          self.ping_num = 1
17          self.dma = ol.axi_dma
18          self.data_cnt = ol.packets_sent
19
20      async def main(self):
21          #Open file
22          self.file = open("Ping/data_fail.dat", "ab+")
23          self.file.write((2).to_bytes(4, byteorder=sys.byteorder))
24
25          #Allocate buffers for transfer.
26          buffer = pynq.allocate(shape=(500000), dtype=np.int32)
27
28          while (True):
29              #Await transfer
30              self.dma.recvchannel.transfer(buffer)
31              try:
32                  await asyncio.wait_for(self.dma.recvchannel.
                        wait_async(), timeout=10)
33              except asyncio.TimeoutError:
34                  self.file.close()
35                  break
36
37              #Write to file
38              #start = time.time()
39              self.write_to_file(buffer)
40              #if (time.time()-start) > 0.0327:
41              # print(f"FAIL AT ping_number: {self.ping_num}")
42              self.ping_num += 1
43
44      def write_to_file(self, buffer):
45          self.file.write(int(datetime.now().strftime("%Y%m%d")).
                to_bytes(4, byteorder=sys.byteorder))
46          self.file.write(int(datetime.now().strftime("%H%M%S%f")
                [0:9]).to_bytes(4, byteorder=sys.byteorder))
47          L = self.data_cnt.register_map.GPIO_DATA.
                Channel_1_GPIO_DATA
48          self.file.write((L).to_bytes(4, byteorder=sys.byteorder))
49          buffer[:L].tofile(self.file, format="%b")
50
51
52  #Instantiate driver object and start transfer loop
53
```

```
dt = DataTransfer(overlay)

loop = asyncio.get_event_loop()
task1 = loop.create_task(dt.main())
loop.run_until_complete(task1)
```

Generering av Hilbert filterkoeffisienter og konvertering av filterkoeffisienter til toerkomplement (generate_coeff.py)

```
import numpy as np
import scipy.signal as si
import matplotlib.pyplot as plt

lp = [0.0001612793, 0.0003503660, 0.0006538771,
     0.0010300252, 0.0014077700, 0.0016671968,
     0.0016481037, 0.0011742540, 0.0000929953,
    -0.0016742066, -0.0040789359, -0.0069024035,
    -0.0097341179, -0.0119835065, -0.0129305792,
    -0.0118146109, -0.0079515959, -0.0008635891,
     0.0096021909, 0.0231879998, 0.0391941656,
     0.0565141149, 0.0737372544, 0.0893088073,
     0.1017245304, 0.1097305526, 0.1124961243,
     0.1097305526, 0.1017245304, 0.0893088073,
     0.0737372544, 0.0565141149, 0.0391941656,
     0.0231879998, 0.0096021909, -0.0008635891,
    -0.0079515959, -0.0118146109, -0.0129305792,
    -0.0119835065, -0.0097341179, -0.0069024035,
    -0.0040789359, -0.0016742066, 0.0000929953,
     0.0011742540, 0.0016481037, 0.0016671968,
     0.0014077700, 0.0010300252, 0.0006538771,
     0.0003503660, 0.0001612793]

hp = [0.0002638003, -0.0005195259, -0.0007352918,
    -0.0011629362, -0.0017706456, -0.0025651472,
    -0.0035622802, -0.0047774809, -0.0062220994,
    -0.0079015019, -0.0098138312, -0.0119491456,
    -0.0142888992, -0.0168057788, -0.0194639265,
    -0.0222195594, -0.0250219809, -0.0278149583,
    -0.0305384195, -0.0331304046, -0.0355291909,
    -0.0376754998, -0.0395146837, -0.0409987933,
    -0.0420884263, -0.0427542681, 0.9570217500,
    -0.0427542681, -0.0420884263, -0.0409987933,
    -0.0395146837, -0.0376754998, -0.0355291909,
    -0.0331304046, -0.0305384195, -0.0278149583,
```

```
    -0.0250219809, -0.0222195594, -0.0194639265,
    -0.0168057788, -0.0142888992, -0.0119491456,
    -0.0098138312, -0.0079015019, -0.0062220994,
    -0.0047774809, -0.0035622802, -0.0025651472,
    -0.0017706456, -0.0011629362, -0.0007352918,
    -0.0005195259, 0.0002638003]


f = open("fir_coeff.txt", "w")

n = -25
h = []
for i in range(0, 51):
   if (n == 0) or (n%2 == 0):
      h.append(0)
   elif (n > 0):
      h.append(((2*(np.sin((np.pi*n)/2)**2))/(n*np.pi)))
   elif (n < 0):
      h.append(((2*(np.sin((np.pi*n)/2)**2))/(n*np.pi)))
   n = n + 1
w = si.windows.blackman(len(h))
h_w = np.array(h)*w

plt.stem(h_w)
plt.stem(h)
plt.show()

ht_bin = []
for h in h_w:
   if (round(h*2**17) == 0):
      coeff_scale = 0
   elif (round(h*2**17) > 0):
      coeff_scale = round(h*2**17)
      ht_hex.append(format(coeff_scale, "#020b"))
   elif (round(h*2**17) < 0):
      coeff_scale = (1<<18) + round(h*2**17)
      ht_bin.append(format(coeff_scale, "#020b"))

lp_bin = []
for coeff in lp:
   if abs(coeff*2**17) < 1:
      coeff_scale = 0
   elif coeff < 0:
```

```
        coeff_scale = (1<<18) + round(coeff*2**17)
    else:
        coeff_scale = round(coeff*2**17)
    lp_bin.append(format(coeff_scale, "#020b"))

hp_bin = []
for coeff in hp:
    if abs(coeff*2**17) < 1:
        coeff_scale = 0
    elif coeff < 0:
        coeff_scale = (1<<18) + round(coeff*2**17)
    else:
        coeff_scale = round(coeff*2**17)
    hp_bin.append(format(coeff_scale, "#020b"))


f.write("HT: \n")
for i in range(len(ht_hex)):
    f.write(f'"{ht_hex[i][2:]}", ')
    if ((i+1)%5 == 0):
        f.write(f'\n')

f.write("\nLP: \n")
for i in range(int(len(lp_hex)/2) + 1):
    f.write(f'"{lp_hex[i][2:]}", ')
    if ((i+1)%5 == 0):
        f.write(f'\n')

f.write("\nHP: \n")
for i in range(int(len(hp_hex)/2) + 1):
    f.write(f'"{hp_hex[i][2:]}", ')
    if ((i+1)%5 == 0):
        f.write(f'\n')
```

D Matlab kode

Generering av høypassfilter (FIR_HP.m)

```
N = 52;
Fp = 20e3;
Fs = 961538;

h = firceqrip(N,(Fp/(Fs/2)),[1e-4 1e-3],'high');
```

```
fvtool(h,'Fs',Fs,'Color','White')
fmt=['[' repmat(' %1.10f,',1,numel(h)) ']'];
fprintf(fmt,h,length(h))
```

Generering av lavpassfilter (FIR_LP.m)

```
N = 52;
Fp = 20e3;
Fs = 961538;

h = firceqrip(N,(Fp/(Fs/2)),[1e-3 1e-4],'passedge');
fvtool(h,'Fs',Fs,'Color','White')
fmt=['[' repmat(' %1.10f,',1,numel(h)) ']'];
fprintf(fmt,h)
```