NTNU
Det skapande universitetet

DSP for Lågeffekt Programvaredefinert Radio

Martin Maråk

Master i elektronikk
Oppgåva levert: Juni 2011
Hovudrettleiar: Per Gunnar Kjeldsberg, IET

Noregs teknisk-naturvitskapelege universitet

Oppgåvetekst

Kandidatens namn: Martin Maråk
Tittel: Low Power Software-Defined Radio

SDR or Software-Defined Radio is typically a highly flexible radio platform where traditional radio components as modulators, demodulators, filters, etc. are realized in a programmable device like an FPGA, DSP or even a general purpose CPU. Traditionally, such systems have been confined to military, instrumentation and other high-performance applications. However, the technology is entering other application areas as well, but for battery-powered applications the main obstacle is still power consumption in particular if existing processors are to be used.

Benefits of SDR are among others, 1) total performance (link/network), 2) extended equipment lifetimes, 3) flexibility both with respect to the development process as well as to emerging radio standards and effective spectrum utilization. The last point is possible since such a system may actually be programmed to take the application's needs as well as traffic and channel conditions into account when claiming (limited) radio spectrum resources. For Nordic Semiconductors, SDR is interesting from several viewpoints (product/prototyping/test).

The assignment consists in a search of the available literature for SDR activities, with focus on low-power architectures and techniques, in particular used for GFSK / PSK modulation. Based on this, architecture for a low-power digital signal processor, specifically tuned for a software defined radio. When selecting the architecture, a well-founded trade-off between flexibility, area and power consumption should be performed.

Vegleiarar: Sverre Wichlund, Nordic Semiconductor ASA , Førsteamanuensis Lars Lundheim
Faglærar: Professor Per Gunnar Kjeldsberg

Samandrag

Programvaredefinert radio (SDR) er ein ny måte å implementere radiosystem på. Hovudtanken er at delar av radioen som tidlegare har vore implementert med låste analoge og digitale løysingar skal erstattast med programvare som køyrer på ein prosessor. Dette kan forbetre mellom anna fleksibilitet, tilpassingsdyktigheit og produksjonskostnadar.

Denne oppgåva tek for seg ein DSP-arkitektur spesielt tilpassa lågeffekt modulasjon og demodulasjon av radiosignaler med låg kompleksitet. Bluetooth er vald som døme og demodulasjonsdelen av denne er analysert for å belyse kva krav ein slik applikasjon vil stille til ein digital signalprosessor.

Gjennom denne analyse kjem det fram at ein del av applikasjonen, estimasjon av arcus tangens, bør akselererast for å oppnå optimal yting og effektforbruk. Dette blir realisert gjennom å introdusere ei CORDIC-eining i systemet. Denne er så satt i samanheng med resten av prosessorarkitekturen, og det er sett på korleis dette påverkar krava til arkitekturen.

Det blir mellom anna konkludert med at DSP-en bør innehalde ein dedikert løkkehandterar samt ein cache for programminnet, og desse blir beskrivne i detalj.

Forord

Denne oppgåva er den avsluttande delen av ei mastergrad ved Instituttet for Elektronikk og Telekommunikasjon ved NTNU. Den blei skreve våren 2011 og bygger vidare på ei prosjektoppgåve utført hausten før. Noko av det arbeidet eg utførde i prosjektoppgåva er gjenbrukt i denne, særskild metoden for applikasjonsanalyse.

Oppgåva er gitt av Nordic Semiconductor og var opprinneleg eit litteratursøk som omhandla "state of the art" SDR, men den blei endra til ein beskriving av ein DSP spesielt tilpassa SDR. Dette fordi det var meir fagleg interessant for meg personleg.

Det har vore givande å sette seg inn i og skrive om temaet og eg har fått meget god vegleiing av Per Gunnar Kjeldsberg, Lars Lundheim og Sverre Wichlund. Eg vil takke dei for at dei alltid var tilgjengelege både for å svare på faglege og praktiske spørsmål.

Dette er som sagt den avsluttande delen av ei mastergrad og fem veldig fine år ved NTNU med trivelege og gode klassekameratar. Tusen takk til mor og far for all støtta under utdanninga, og takk til lillebroren min for støtte og hjelp og for at han orkar å høyre på når eg pratar fag.

Martin Maråk

Trondheim

18.06.2011

Stikkordliste

ADC: Analog-til-digital konverter ... 3
AGU: Adressegeneringseining ... 16
ALU: Aritmetisk-logisk eining ... 18
ASIC: Applikasjonsspesifikk integrert krets ... 1
ASIP: Prosessor med applikasjonsspesifikt instruksjonssett ... 6
begrensar ... 35
Bluetooth ... 33
cache: hurtigbuffer ... 17
CORDIC: Coordinate Rotation Digital Computer ... 19
DMA: Direkte minneaksess ... 19
DPSK: Differential Phase Shift Keying ... 37
DSP: Digital Signalprosessor ... 1
FFT: Fast Fourier Transform ... 6
GFSK: Gaussian Frequency Shift Keying ... 36
GPU: Generell prosessor ... 1
Hard desisjon ... 39
ID: Instruksjonsdekoder ... 15
instruksjonssett ... 29
Kontrollflytoperasjon ... 31
limiter ... *sjå begrensar*
LNA: Lågstøyforsterkar ... 4
Løkker ... 26
løkketellar ... 26
MAC: Multiply Accumulate ... 6
MIMD: Multiple Instruction Multiple Data ... 25
MIPS: Millionar Instruksjonar per Sekund ... 11
mjuk desisjon ... 39
Moduloadressering ... 29
PC: Programtellar ... 15
PM: Programminnet ... 15
Registeradressering ... 29
RISC: Reduced Instruction Set Computer ... 24
RRI: Register-Register-Immediate ... 57
RRR: Register Register Register ... 57
scratchpad: Arbeidsminne ... 17
SDR: Programvaredefinert radio ... 1
SIMD: Single Instruction Multiple Data ... 13
Taylor-rekker ... 41
Umiddelbar adressering ... 29
VLIW: Very Long Instruction Word ... 13

Innhald

Oppgåvetekst I
Samandrag III
Forord V
Stikkordliste VII
Innhald IX
1. Introduksjon 1
1.1 Motivasjon 1
1.1.1 Programvaredefinert radio 1
1.1.2 Kvifor DSP? 3
1.2 Antakingar 4
1.3 Tekstens oppbygnad 5
1.4 Viktige bidrag 5
2. DSP-arkitektur 7
2.1 Generelt om DSP-design 7
2.2 Låg-effekt DSP-design 8
2.3 ASIP 10
2.3.1 Analyse av aktuelle applikasjonar 12
2.3.2 Utforsking av det arkitektoniske løysingsrommet 14
2.3.3 Generering av instruksjonssett 15
2.4 Komponentar 16
2.4.1 Programminne 16
2.4.2 Kontrolleining 17
2.4.3 Dataminne og dataadressering 18
2.4.4 Datasti 20
2.4.5 I/U – DMA, avbruddsdriven I/U eller programert I/U 20
2.5 Akseleratorar 21
2.5.1 CORDIC 21
2.6 Parallell databehandling 26
2.6.1 Samlebandsarkitektur 26
2.6.2 Ulike parallelle prosessortypar 27
2.7 Løkkehandtering 28
2.7.1 Løkkehandtering i programvare 28
2.7.2 Løkkehandtering i maskinvare 29

3. Instruksjonssett 31
3.1 Kva er eit instruksjonssett? 31
3.2 Minneadressering 31
3.3 Operasjonar i instruksjonssettet 32
3.3.1 Aritmetiske og logiske operasjonar 32
3.3.2 Kontrollflytoperasjonar 33
3.4 Koding av instruksjonar 33
4. Bruksområdet til prosessoren 35
4.1 Bluetooth 35
4.2 Algoritmar som nyttast i desse protokollane 38
4.2.1 GFSK 38
4.2.2 DPSK 39
4.3 Analyse og profilering 39
4.3.1 Analyse av GFSK 40
4.3.2 Analyse av DPSK 40
4.4 Minimumskrav til nøyaktigheit i approksimasjon av fase 41
5. Implementasjon av akselleratorar 43
5.1 Plassering av ADC 43
5.1.1 Begrensar 43
5.2 Approksimasjon av arcus tangens 43
5.3 Samanlikning av arealbruk 47
5.4 Implementasjon av CORDIC 48
5.5 Implementasjon med oppslagstabell 51
5.6 Implementasjon av GFSK 52
5.6.1 Implementasjon utan akselerasjon 52
5.6.2 Implementasjon med akselerasjon 54
5.6.3 Implementasjon med CORDIC 56
5.7 Implementasjon av DPSK 56
5.7.1 Implementasjon utan akselerasjon 56
5.7.2 Implementasjon med akselerator 56
5.7.3 Implementasjon med CORDIC 57
5.8 Samanlikning og diskusjon 57
5.9 Programstorleik 58
6. Forslag til arkitektur 61
6.1 Instruksjonssett 61
6.2 Arkitektur 62

6.2.1 Parallellitet 62
6.2.2 Minne 62
6.2.3 Datasti..65
6.2.4 Kontrollsti 66
7. Konklusjon og vidare arbeid 69
7.1 Konklusjon 69
7.2 Vidare arbeid..69
8. Referansar 71
9. Vedlegg..77
9.1 Vedlegg 1: Kode og pseudokode..77
9.1.1 Vedlegg 1.1 : Pseudokode for GFSK..77
9.1.2 Vedlegg 1.2 : Pseudokode for DPSK..78
9.1.3 Vedlegg 1.3: CORDIC C-kode..79
9.1.4 Vedlegg 1.4: Matlab-kode for utrekning av maksimal feil for CORDIC-algoritma 80
9.2 Vedlegg 2: ADL-basert ASIP-design: 81

1. Introduksjon

1.1 Motivasjon

1.1.1 Programvaredefinert radio

Kva er programvaredefinert radio?
Ein programvaredefinert radio, eller SDR ("Software Defined Radio"), er ein radio der nokre eller alle delane av dei fysiske funksjonslaga er definert i programvare. Det vil seie at delar som ville vore implementert som dedikert maskinvare i ein "tradisjonell" radio er erstatta med programmer som køyrer på ein slags form for prosessor, i dei fleste tilfelle ein digital signalprosessor (DSP), men det kan også vere ein generell prosessor (GPU) eller ein FPGA [**1**]. Med det fysiske funksjonslaget meiner ein her den delen av ein radioprotokoll der ein behandlar IF-, RF- eller baseband-signaler samt kanalkoding [**2**].

Korleis dette gjerast er opent for ein del variasjon. I den ein enden av skalaen finn ein programvare-kontrollert radio. I denne forma for SDR er det fysiske laget i stor grad låst, og programvaren kan berre endre på ulike parameter innanfor eit lite område [**2**]. Dette gjev noko høgare fleksibilitet enn ein tradisjonell radio, men lite fleksibilitet og programmerbarheit i forhold til dei andre typane SDR.

I den andre enden av skalaen finn ein ein programvareradio der alle delar av det fysiske laget er definert i programvare. Dette kan kallast ein idell programvaredefinert radio [**3**].

Mellom desse finnes det ei nærast uendeleg rekke variasjonar. Radioen kan til dømes vere kognitiv og adaptiv, det vil seie at den kan analysere og tilpasse seg til omgjevnadane den opererer i [**2**]. Alternativt kan den kun tilpassast ved produksjon eller ved hjelp av oppdateringar etter den er tatt i bruk.

Fordelar:
ASIC-implementasjonar ("Application Specific Integrated Circuit") har implisitt lågare effektforbruk og høgare yting enn ein programvareimplementasjon av same funksjonalitet [**4**], dette er fordi ein slik implementasjon idellt sett vil vere ei optimal løysing på den applikasjonen som skal realiserast.

På tross av dette blir SDR brukt i stor grad i mellom anna forsvars-applikasjonar og den tar stadig større del av det mobile kommunikasjonsmarknaden, både på tenesteleverandør-sida og på brukarsida. Mange spår at SDR etterkvart vil dominere dette markedet [**1**]. Dette er fordi SDR byr på ei rekke fordelar over dei tradisjonelle maskinvareløysingane:

Fleksibilitet:
SDR går ut på å gjere signalbehandlinga i eit radiosystem i programvare. Dermed er det mogleg å endre måten radioen operererar på etter at den er tatt i bruk. Til dømes kan ein SDR tilpasse det frekvensbandet den nyttar eller modulasjonsmetoden utan at ein treng å produsere ei ny eining [**5**].

Dette kan gje fleire fordelar. Mellom anna kan det føre til ei meir effektiv utnytting av den tilgjengelege bandbreidda. Ulike radiosystem/protokollar er lisensiert til ulike frekvensbandbreidder og på grunn av det store spekteret av ulike radiosystem er bandbreidde og gode frekvensar ein knapp resurs. Radioar er ikkje på heile tida og dermed blir ikkje heile bandbreidda utnytta. Med SDR kan radiosendararar og mottakarar til ei kvar tid justere dei frekvensane dei opererar på, og dermed utnytte bandbreidda meir effektivt. Ein kan tenke seg at dei kan søke gjennom frekvensbandet og finne frekvensar som ikkje er nytta og så stille seg inn på desse [**5**].

Ein annan fordel er at den kan tilpasse seg nye standardar etter at radioen er tatt i bruk. Dette fører til at det er enklare å forbetre standardane for kommunikasjon, fordi ein ikkje treng å bytte ut all maskinvare både på brukarnivå og leverandørnivå for å innføre endringar eller rette på feil [**5**].

Vidare kan den tilpasse seg endringar i sendeforhold (støy, rekkevidde o.s.b) ved til dømes å endre demodulasjonsmetode, bølgjeform og liknande. Den kan også gje betre sikkerheit enn eit maskinvaresystem, til dømes gjennom å bytte frekvens når den oppdagar brudd på sikkerheita [**5**].

Fleksibiliteten er i teorien total, fordi ein kan bytte ut programvaren systemet køyrer med kva som helst anna programvare. I realiteten vil den naturlegvis vere begrensa av ytinga til prosessoren som køyrer programmet. I ein mobil eller handhalden applikasjon er effektforbruk særs viktig og sidan høgare yting kan ha samanhang med høgare effektforbruk må ein ofre ein del av ytinga for å få eit godt nok effektforbruk.

Lågare kostnad og utviklingstid:
Når ein lagar ein maskinvareradio vil ei kvar endring i funksjonaliteten medføre at maskinvara må endrast. For det første er dette særs tidskrevande. Arkitekturen må i verste fall designas på nytt heilt frå botnen av, den må testast og så må den gjerast klar for produksjon. Med andre ord betyr det at ein kanskje må utvikle eit heilt nytt system. SDR kan potensielt gje moglegheita til å nytte ei felles maskinvareplattform for ei rekke ulike produkter, samt over fleire generasjonar av same produkt [**6**].

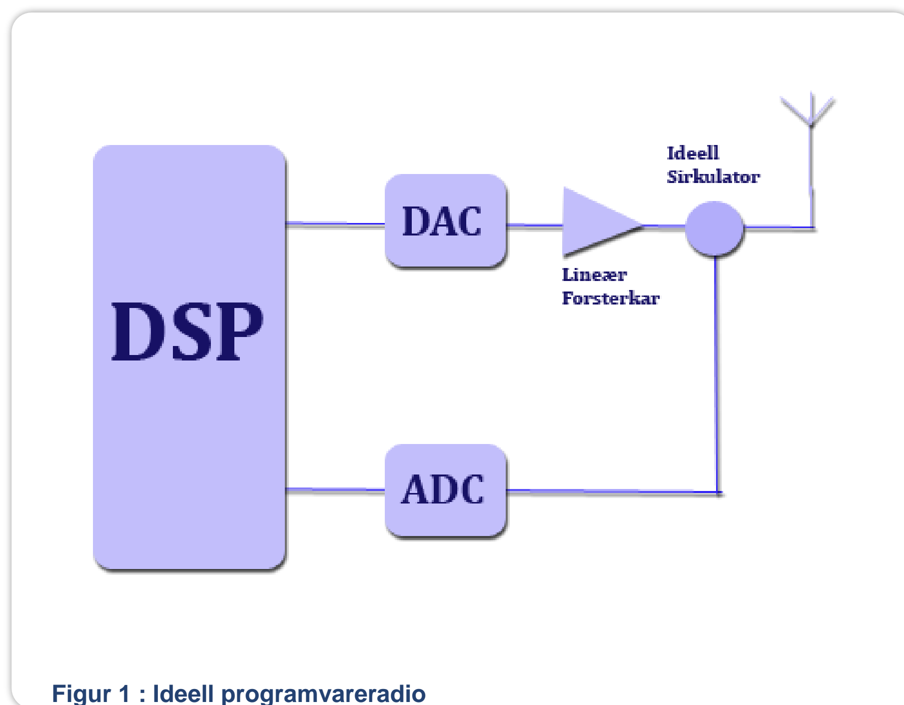
Dette kan føre til at ein aukar produksjonsvolumet på maskinvaren betrakteleg, noko som fører med seg reduserte produksjonskostnadar. Vidare fører det også til kortare utviklingstid, noko som også fører til lågare kostnad. I tillegg kjem det at eventuelle feil i eit programvaresystem er mindre kostbare i eit maskinvaresystem fordi dei i stor grad kan rettast opp ved endringar i programvare (“patches”) [**6**] [**7**].

Vidare kan ein sjå for seg eit system som skal støtte ei rekke ulike radioprotokollar og operasjonsmodusar. Lågare produksjon- og utviklingskostnad fordi det berre trengs eit system for ei eining som støttar mange sendemodi/protokollar [**7**]. Sidan det sjeldan er aktuelt at alle protokollar skal behandlast samtidig, kan desse dele prosesseringsresursar. Dette kan vere ein måte å redusere effektforbruket, og også ein måte å redusere storleiken på systemet i forhold til separate maskinvareimplementasjonar av alle dei ulike protokollane [**8**].

Ideell SDR

I ein ideell programvare-radio er modulasjon, demodulasjon, kanalisering (“channelization”), protokollar og ekvalisering (“equalization”) alle implementert med programvare i ei digital signalprosesseringseining. Denne eininga kan anten vere ein DSP, ein GPU, ein FPGA eller ein ASIC med programmerbare delar. [**9**] Denne oppgåva baserar seg på ein DSP-implementasjon.

I teikninga kan ein anta at ADC-en (“analog til digital konverter”) inkluderar eit anti-aliaseringfilter og at DAC inkluderar eit rekonstruksjonsfilter.



Figur 1 : Ideell programvareradio

Den ideelle sirkulatoren er nytta for å separere sending og mottaking. For å gjere dette er den avhengig av perfekt “matching” mellom antenna og forsterkaren. Dette er urealistisk, og i verkelegheita kan ein nytte filterbaserte løysingar, men desse er dårleg egna til programvareradio fordi dei ikkje er fleksible nok til å støtte variasjonen i programvareradio. [**9**]

1.1.2 Kvifor DSP?

Som nemnt har ASIC betre yting og lågare effektforbruk enn ein DSP eller GPU. Det er mellom anna er det nærmast umogleg å utnytte parallelitet på same måte i ein DSP som i dedikert maskinvare. Dette gjev dedikert maskinvare store fordelar med tanke på både gjennomstrøyming og effektforbruk, men dei manglar fleksibilitet. Dei kan berre utføre ein førehandsbestemd funksjonalitet. DSP-ar og GPU-ar har i teorien moglegheita til utføre eit kvart program [**10**].

I tillegg til denne fleksibiliteten har generelle prosessorar fleire fordelar over dedikert maskinvare [**10**]:

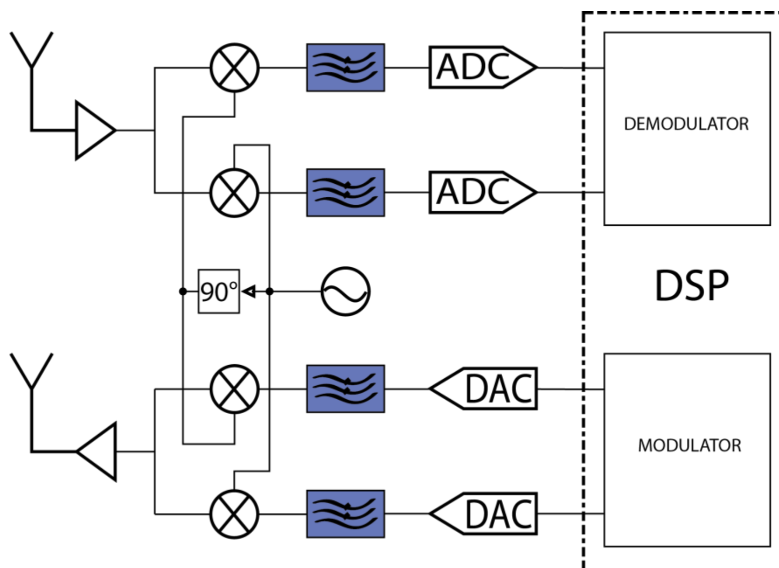
- Dei er forutsigbare og velprøvde og gjev derfor eit system som har mindre sjanse for feil.
- Dei kan til ei kvar tid omprogrammerast for å rette opp eventuelle feil.
- Produksjonsvoluma er store, fordi dei kan nyttast til ei rekke ulike applikasjonar utan å endrast.
- Dei kan brukast på nytt for nye teknologiar, noko som gjev lågare kostnadar enn å lage ny dedikert maskinvare.

1.2 Antakingar

Dette avsnittet tek for seg dei antakingane og avgrensingane som resten av oppgåva baserar seg på. Desse er gjorde for å gje eit avgrensa grunnlag for å beskrive prosessoren.

RF-front-end

I ein programvareradio er det ønskeleg å ha det digitale signalprosessoren "nærast mogleg" antenna, fordi dette aukar fleksibiliteten til systemet. Det vil seie at ein vil ha minst mogleg analoge komponentar mellom prosessoren og antenna. I denne oppgåva antek ein at det finnes nokre analoge komponentar, fordi desse ikkje gjev god nok yting når ein implementerer dei digitalt. Dei komponentane som er implementert analogt er ein låg-støy-forsterkar ("low noise amplifier - LNA") og to eller fleire filter før og etter LNA som fjernar uønskte frekvensar. Etter dette miksast signalet før ADC omformar det til digitale punktprøver ("samples").



Figur 2: RF-delen av prosessoren

Figuren over viser ein forenkla modell av den analoge RF-delen kopla til den digitale delen, som er markert som området med stipla linje. Som ein kan sjå liknar sendedelen på mottakaren, men ein vil ha ein annan type forsterkar enn for mottakaren. Ein kan anta at samplefrekvensen på ADC-ane er 16 MHz.

Applikasjonen

Ei Bluetooth-eining må kunne både sende og motta data for å kunne fungere. Sending og mottaking har ulike krav og eigenskapar. Denne oppgåva brukar mottakardelen av Bluetooth til profilering (kap. 4.1). Det er vald å berre fokusere på mottakardelen av applikasjonen. Grunnen til at sendardelen ikkje blir brukt som døme er at ein kan anta at denne har så mykje lågare kompleksitet og krav til yting relativt til mottakaren at ein trygt kan anta at ein prosessor som oppfyller krava til ein mottakar også vil kunne oppfylle krava til ein sendar. Vidare vil desse aldri bli nytta parallelt og vil difor heller ikkje "konkurrere" om resursane til prosessoren.

1.3 Tekstens oppbygnad

Kapittel 2, 3, og 4 fram til 4.3 presenterer teorien som er nødvendig for å kunne diskutere prosessorens oppbygnad og funksjon. Kapittel 2 tek for seg den generelle oppbygnaden til ein DSP, og presenterer teori om designmetodar for ASIP, typiske komponentar i ein DSP, akseleratorar, parallellitet i prosessorarkitektur og løkkehandtering.

Kapittel 3 tek for seg teori rundt instruksjonssett, og presenterer mellom anna typiske instruksjonar for DSP-ar.

Kapittel 4 til og med 4.2 tek for seg applikasjonen og det tenkte bruksområdet for prosessoren

Frå og med 4.3 og til og med kapittel 6 er forfattaren sitt eige bidrag og er ein diskusjon rundt applikasjonen til prosessoren og dei konsekvensane dette får for prosessorarkitekturen.

4.3 og 4.4 gjer analyse og profilering av applikasjonen. Kapittel 5 diskuterer implementasjon av akseleratorar for dei ulike demodulasjonsmetodane presentert i teoridelen.

Kapittel 6 sett dei foreslåtte akseleratorane inn i ein fullstendig prosessorarkitektur og diskuterer også kort korleis instruksjonssettet bør vere bygd opp.

Kapittel 7 presenterer kort dei konklusjonane som er gjort og ser også på potensielt vidare arbeid rundt det som er presentert i dei tidlegare kapitla.

1.4 Viktige bidrag

Det viktigaste bidraget og hovudformålet med denne oppgåva er ein applikasjon av ASIP-designmetodologi på applikasjonen lågeffekt programvareradio. To former for demodulasjon er analysert og profilert og ut frå dette er det gjort ein diskusjon rundt ulike måtar å approksimere arcus tangens på, der hovudvekta i vurderinga er på fleksibilitet og lågt effektforbruk. Konklusjonen her er at ei CORDIC-eining er særs egna for formålet.

Vidare er denne eininga satt inn i ein prosessorarkitektur og det er sett på kva konsekvensar dette får. Det er konkludert med at systemet bør ha ein dedikert løkkehandterar og ein instruksjonscache på om lag 40 byte.

1.5 Relatert arbeid

Det finnes ein del arbeid som tek for seg noko av den same tematikken som denne oppgåva. To tidlegare masteroppgåver frå NTNU, av Hallvard Næss [**11**] og Roger Koteng [**12**] har gjort nokre av dei same vurderingane, men for ein annan applikasjon.

Av eksperimentelle DSP-ar for SDR er SODA [**13**] verdt å nemne, medan av dei meir "ferdige" generelle SDR-prosessorane er Sandbridge sin Sandblaster [**14**] viktig å trekke fram.

2. DSP-arkitektur

2.1 Generelt om DSP-design

Digitale signalprosessorar er prosessorar som er særskild tilpassa digital signalbehandling generelt og ofte spesifikt den applikasjonen og dei algoritmane som skal køyrast på den. Med digital signalprosessering meiner ein at ein ved hjelp av eit digitalt system behandlar eit innkomande signal og sender det ut igjen i "den andre enden [**11**]." Det at ein DSP er spesialtilpassa gjer at den kan operere meir effektivt enn ein GPU med tanke på spesifikke applikasjonar eller avgrensa sett av applikasjonar, som til dømes digital kommunikasjon [**12**].

Det finnes mange ulike typar DSP, men dei delar nokre felles karakteristikkar:

1. Spesiell høghastigheits aritmetikk
2. Dataoverføring til og frå den verkeleg verda ("sanntidssystem")
3. Minnearkitektur med fleire kjelder

Dette krever typisk operasjonane:

1. Addisjon og multiplikasjon
2. Forseinkingar
3. Behandling av tabellar

For å gjere desse operasjonane effektivt har DSP-ar ofte: parallell addisjon og multiplikasjon, fleire minneaksessar og stort antal register for midlertidig datalagring. Denne effektiviseringa kan oppnåas ved å introdusere spesialiserte aritmetiske einingar, til dømes "mulitply-accumulate" eller MAC, som er ein kombinasjon av raske multiplikatorar og akkumulatorar som kan utføre multiplikasjon og akkumulasjon på ein klokkesyklus. Desse er beskrivne i meir detalj seinare i avsnitt 2.4.4. [**12**].

Effektiv behandling av tabellar involverer vanlegvis at ein DSP har meir avansert generering av nye adresser enn ein GPU. Nye adresser kan genererast samstundes som ein utfører ein "fetch"- eller "store"-operasjon, og dette skjer utan at det kostar ekstra syklusar for prosessoren [**12**], også kjend som "overhead".

Generelle DSP-ar har ein stor grad av fleksibilitet og er det er relativt enkle å utvikle programmer for. Denne typen prosessorar fungerer godt når ein har låge krav til effektforbruk og yting, men for ein lågeffekt mobil applikasjon er dette uakseptabelt. Når desse krava er strenge er det naudsynt å nytte ein DSP som er spesielt tilpassa den applikasjonen eller applikasjonane den skal køyre, til dømes ein sokalla "application specific instruction-set processor" (ASIP). For ein ASIP er ikkje målet å ha absolutt fleksibilitet, men heller at ein skal ha fleksibilitet som er akkurat tilstrekkeleg for applikasjonsområdet til ASIP-en [**13**].

Arkitekturen til DSP-en må derfor spesialtilpassast for oppnå god nok yting og effektforbruk. Som ein illustrasjon på dette kan ein prosessor som er laga for å køyre mykje FFT ("fast fourier transform") inkludere den spesialiserte adressegenereringsoperasjonen

"bitreversering" samt moduloadressering. Dette er omtrent berre naudsynt for FFT, og ville vore svært ineffektivt i ein prosessor som sjeldan utfører denne operasjonen [**14**] [**15**].

2.2 Låg-effekt DSP-design

Dei seinaste åra har det blitt viktigare for maskinvare- og programvare-designarar å ta hensyn til effektforbruk. Dette har fleire ulike årsakar. Mellom anna:

1. Ny teknologi innanfor batteri og transistorar har mogleggjort ei rekke nye mobile applikasjonar [**10**].
2. Store system (serverar o.l.) brukar store mengder energi, samanliknbar med luftfart og industri [**10**].

I dette tilfellet er det første punktet som tvinger fram eit hensyn til effektforbruk. Det tenkte bruksområdet for DSP-en er i mobile/handhaldne applikasjonar og det er difor kritisk å ha eit så lågt effektforbruk som mogleg. Dette gjerast gjennom ei rekke teknikkar som blir beskrivne i dette avsnittet.

Det ideelle er å auke ytinga samtidig som ein minkar effektforbruket. Dette kan mellom anna gjerast ved å introdusere spesielt tilpassa instruksjonar. Desse gjer at systemet kan fungere effektivt under visse arbeidslaster, det vil seie arbeidsoppgåver som blir utførd ofte [**10**].

Dynamiske effektbruksmodi:

Den må kunne støtte programvare-styrte modus for effektforbruk. Det vil seie å kunne "slå av" delar av prosessoren. Dette kan gjerast ved å dele opp prosessorarkitekturen i ulike blokker som individuelt kan settast i låg-effekt "idle"-tilstandar. Desse tilstandane er tilgjengelege for programmeraren gjennom eit kontrollregister [**12**]. Ein kan gjere dette ved å stoppe klokka for enkelte delar("clock gating") eller ved å slå av forsyningsspenninga til deler av prosessoren ("power gating").

Døme: TI TMS320C55 er delt opp i blokkene CPU, DMA, perifere einingar, klokkegenerering, instruksjonscache og ekstern minnetilgang. Når kvar av desse er i "idle" er funksjonaliteten deira ikkje tilgjengeleg [**12**].

Val av algoritmer:

Gjeve ein funksjonalitet systemet skal realisere finnes det mange ulike måtar å løyse dette på. Ulike algoritmar kan realisere funksjonalitet på ulike måtar og har ulike effektforbruk og ytingar knytt til seg. Å velje algoritmar er ei kompleks oppgåve, fordi ein må sjå på korleis ei algoritme oppfører seg saman med den programvaren den køyrer på. Eit viktig begrep er **kjerner** , små delar av koden/algoritmen som programmet brukar mesteparten av køyretida si på. Ved å optimalisere maskinvara for å ta hand om desse kan ein få eit kraftig og energieffektivt system. Denne optimaliseringa kan til dømes implementerast ved at ein har dedikert maskinvare for nett denne kjerna og skrur av spenningsforsyninga til resten av systemet [**10**].

Begrensing av driftsspenning og klokkefrekvens:

Ein enkel metode for å senke effektforbruket for eit system er å senke driftsspenninga. Det ser ein klart om ein ser på formelen for dynamisk effektforbruk [**16**]:

$$P_{dyn} = \alpha c V_{dd}^2 f \quad \textbf{Formel 1}$$

Der α er "aktiviteten" på kretsen, C er kapasitansen, V er driftsspenninga, og F er frekvensen. Som ein ser vil det vere mest effektivt å senke spenninga på kretsen, sidan denne er kvadrert. Problemet med å senke driftsspenninga er at det gjer at systemet får ein lågare maksimal operasjonsfrekvens. Ein god metode for å redusere effekten er difor å

1 Optimalisere systemet slik at det blir raskare enn dei krava spesifikasjonen for systemet setter
2 Senke driftspenninga til systemet til den maksimale frekvensen svarar til det opprinnlege kravet.

Denne optimaliseringa skjer ofte ved at ein nyttar meir areal til å lage spesialiserte einingar for enkelte instruksjonar, rullar opp løkker og liknande [**10**].

Denne metoden er særs effektiv, men det finnes nokre problemer med den. Hovudproblemet er at den blir mindre effektiv dess mindre teknologien (transistorane) blir. I tillegg blir driftspenninga blir mindre og mindre i nyare teknologiar, noko som gjev mindre rom for nedskalering [**10**].

For å optimalisere vidare må ein nytte mellom anna spekulasjon og andre liknande metodar. Desse er ikkje effektsparande fordi dei fører til ein del redundant databehandling. Med andre ord gjev dei høgare yting, men dei kan ikkje nyttast til å senke effektforbruket fordi den reduserte driftsspenninga ikkje veg opp for den redundante databehandlinga som oppstår [**10**].

Eit system som køyrer på lågare klokkefrekvens vil bruke lenger tid på å behandle ei viss mengde data. Dermed har ein lågare effektforbruk, men sidan det tek lenger tid vil den totale energien forbli den same. Om ein antek at heile systemet har same klokke er det lite å tene på å redusere klokkefrekvensen. For å utnytte det lågare effektforbruket knytt til redusering av klokkefrekvens må ein anten fordele klokka utover slik at ulike delar av systemet får ulike klokkefrekvensar, og/eller så må ein variere klokkefrekvensen over tid, sokalla **"clock gating"** [**10**], det er også mogleg å stenge av spenninga til delar av kretsen totalt, sokalla **"voltage gating".**

"Clock gating" er ei form for dynamisk klokkesetting der ulike delar av systemet får begrensa eller inga klokkehastigheit når dei ikkje gjer noko nyttig arbeid. Dette kan også utførast statisk, ved at klokkefrekvensen til ulike delar av kretsen begrensast ved produksjonstid. Dette forenklar den komplekse dynamiske styringa av klokkefrekvens betraktelig og kan redusere effektforbruket ytterlegare i forhold til dynamisk klokkesetting [**10**].

Minnehierarki

Begrepet minnehierarki omhandlar korleis minne er fordelt i forhold til kor nær det er sjølve databehandlinga. Minne som "ligger nære" er raskare og meir effektivt enn minne som er lengre borte. Dermed kan ein anta at ein vil ha mest mogleg minne nærast mogleg databehandlinga, men minne blir også tregare og meir effektkrevjande dess større det er. Difor er det viktig å finne ein balanse mellom storleik og nærleik for å kunne få eit system som er energieffektivt og samstundes har stor nok yting. Generelt er minne svært effektkrevjande både når det gjeld statisk effekt, det vil seie effekt som "lek" ut av systemet når det er inaktivt, og når det gjeld effekt knytt til minneaksessar. Minne kan ta opp 50% av

arealet i ein typisk DSP-applikasjon og bruke 25 til 45% av effekten [**16**]. Det er difor viktig å minimalisere både storleiken på minnet og frekvensen av minneoperasjonar [**10**] [**17**].

Når ein designar systemet er det viktig at resultatet frå utrekningar /operasjonar blir brukt av andre operasjonar så fort som mogleg. Ved å auke temporaliteten gjer ein behovet for minneoperasjonar mindre, og ein minskar dermed effektforbruket til systemet [**10**].

Ein metode for å gjere minneaksess meir effektiv er å dele opp alle nivå av minnet i blokker. Dette gjer ein ved å dele adresserommet opp i mindre rom. Mindre blokker tilsvarar mindre effektforbruk per aksess, men også meir effektforbruk på kommunikasjon, fordi denne blir meir komplisert. Dette er to hensyn som må balanserast mot kvarandre for å oppnå optimal yting [**10**] [**17**].

Kommunikasjonen mellom algoritmane i SDR er prega av "streaming". Det vil seie at medan prosessoren/prosessorane arbeidar med data blir samstundes den neste dataen som skal behandlast flytta frå minne til register. Om ein nyttar denne forma for dataminne vil ein unngå ei stor samanhengande blokk, noko som sparar effekt. I staden for vil ein få eit hierarki som består av ei rekke buffrar. På lågaste nivå vil ein finne registerfila, som er beskrive seinare i avsnitt 2.4.4, nivået over dette består av anten cachar eller "scratchpad"-minne. Om det er naudsynt med eit større minneområde enn dette vil ein ha eit nivå over dette igjen, og om kravet til minne er stort nok må ein nytte seg av eit eksternt minne, det vil seie eit utanfor "chipen [**13**]."

Henting ("fetching") av instruksjonar har også effektforbruk knytt til seg. Det finnes fleire ulike metodar for å optimalisere desse for lågt effektforbruk: viktige dømer er **prekoda instruksjonsbuffer** ("pre-coded instruction buffer") og **løkkecache** ("loop cache"). Begge utnyttar at ein prosessor brukar mykje tid på kjerner i koden, og ved å lagre desse på førehand unngår ein mykje henting av instruksjonar, og dermed sparar ein effektforbruk i forhold til å hente ut programdata direkte fra minnet, eller frå ein "normal" cache [**10**].

Fastpunkt-aritmetikk("Fixed-point artihmetic"):

Fastpunkt-aritmetikk går ut på at ein i motsetnad til flytande-punkt-aritmetikk ikkje inkluderer eksponenten i binære tal i kvart einskild tal. Denne eksponenten er i staden i ein separat variabel. Dette gjer at ei rekke binære tal kan dele ein eksponentverdi, det sparar logikk og gjer systemet meir effektivt. Ein fast-punkt-DSP kan nytte mindre enn halvparten so mykje areal som ein tilsvarande flytande-punkt-DSP. Mange DSP-ar har nokre ekstra breie register for å forhindre avrundingsfeil [**13**] [**12**].

Statisk planlegging:

Statisk planlegging går ut på at all bruk av maskinvare resursar blir bestemd/planlagd når ein kompilerer koden [**12**]. I eit system der avhengigheiter, tidsfristar og kompleksitet er kjend på førehand kan ein nytte statisk planlegging til å optimalisere bruken av maskinvare-ressursane [**13**].

2.3 ASIP

For å få til den ønska fleksibiliteten samtidig som ein opprettheld eit akseptabelt effektforbruk og god nok yting er det naudsynt å ofre noko av fleksibiliteten til ein prosessor for det låge effekt-forbruket til ein ASIC. Dette kan gjerast med ein ASIP("application specific instruction-set processor") [**18**].

Som namnet antydar er dette ein type prosessor der instruksjonsettet er spesielt tilpassa bruksområdet til prosessoren [**19**]. ASIP passar godt til digitale kommunikasjons-applikasjonar, dette er fordi dei forskjellige databehandlignsoppgåvene i desse systema kan delast opp i ei rekke oppgåver som krever større eller mindre grad av fleksibilitet. Dette gjev ein gode moglegheiter til å akselerere visse delar av applikasjonen [**20**].

Denne metodikken gjev ein digital signalprosessor som framleis er generell, i motsetnad til ASIC, men som er særs tilpassa til visse applikasjonar [**13**].

Det er verdt å nevne at det finnes formelle metodar for ASIP-design, men desse er rekna som relativt umodne og er ikkje tatt i bruk i denne oppgåva. Nokre av dei er beskrivne i Vedlegg 2, men det er ikkje nødvendig for lesar å sette seg inn i dei med tanke på resten av rapporten.

Designmetode for ASIP:

Det finnes fleire måtar å angripe designet av ein ASIP, og det er viktig å starte med ein klar spesifikasjon på det systemet ein skal lage. Det vil seie at ein har eit klart bilete av applikasjonen og dei krava og avgrensingane den medfører. Denne oppgåva fokuserer særskild på effektforbruk, men i ein kommersiell applikasjon er sjølvsagt til dømes kostnad og designtid kritisk, og det er her ASIP har fordelar over til dømes ein ASIC [13]. Når ein vel å sjå bort frå hensyn til marknad, kostnad og designtid, kan ein i hovudsak dele opp designprosessen for ein ASIP i fem steg [21] [13]:

1. **Analyse av aktuelle applikasjonar:**
   For å kunne tilpasse arkitekturen til det bruksområdet som er ønska er det naudsynt å gjere ei analyse av den applikasjonen eller applikasjonane som skal nytte ASIP-en.
2. **Utforsking av det arkitektoniske løysingsrommet:**
   Ved hjelp av parameterar frå analysen i første steg finn ein eit sett moglege arkitekturar. Vidare gjer ein eit estimat av ulike parameterar (t.d. effektforbruk, yting og areal) for desse arkitekturane og velden mest ønskelege av dei som oppfyller dei krava som er stilt til arkitekturen.
3. **Generering av instruksjonssett:**
   Her genererer ein eit instruksjonssett som skal nyttast når maskinvare og programvare skal syntiserast.
4. **Kodesyntisering:**
   Ein genererar anten ein eigen kompilator for arkitekturen eller ein sokalla “retargetable” kode.
5. **Maskinvaresyntisering:**
   Maskinvare blir generert ved hjelp av ein ASIP-arkitektur beskrive i steg 2 og eit maskinvarebeskrivande språk, t.d. VHDL eller Verilog.

Det er viktig å merke seg at denne prosessen ikkje er lineær. Den krever mange iterasjonar av nokre av stega. Særleg steg to og tre må sannsynlegvis gjennomførast mange gongar før ein kan oppnå eit tilfredsstillande resultat. Det er kritisk at ein utfører testar (“benchmarks”) for å undersøke korleis ulike arkitekturar og instruksjonssett oppfører seg [**13**].

Kodesyntetisering og maskinvaresyntisering ligger noko utanfor omfanget til denne oppgåva og vil ikkje bli beskrive.

2.3.1 Analyse av aktuelle applikasjonar

Det er viktig å få eit bilete av dei krava applikasjonen stiller, i kommunikasjon kan det vere t.d. bitrater og bandbreidde. Ein må også sjå på dei karakteristiske trekka ved applikasjonen, t.d. kva type algoritmar som blir nytta og programkjerner og typiske operasjonar i desse, det er også viktig å finne ut kor ofte dei ulike instruksjonane/operasjonane blir brukt slik at ein kan avgjersler om kva delar av applikasjonen som bør akselererast. I tillegg til å avdekke karakteristiske trekk kan applikasjonsanalyse også avsløre moglegheiter for parallellisering, noko som gjer at ein kan auke ytinga til systemet [**13**]. Dette kan ein gjere ved å skrive applikasjonen i eit høgnivå programmeringsspråk, eller i assemblykode, og så analysere den, anten med eit analyseprogram, eller som i fordjupingsprosjektet [**4**] knytt til denne masteroppgåva: “for hand.”

Den informasjonen ein prøver å få fram er **funksjonsdekning**, altså kva slags funksjonar som blir nytta i programmet, **databehandlingskostnad**, kostnaden av behandling av algoritmar og minneaksess- og synkroninseringskostnad, og til slutt **minnekostnad**, til dømes storleik på minne og kor mykje som blir brukt til ei kvar tid.

Metoden som nyttast for å analysere heiter **kodeprofilering** (“source code profiling”), dette går enkelt sagt ut på at ein stiller seg spørsmåla:

1. Kva funksjonar er det som krever mest MIPS og difor bør aksellererast?
2. Kva funksjonar blir nytta mest og difor bør aksellererast?
3. Korleis ser datavegen ut etter punkt 1 og 2?
4. Korleis må minne- og buss-arkitekturen vere for å kunne gjennomføre punkt 1 og 2 og samstundes halde seg innanfor begrensingane på minnestorleik?

Problemet med metoden er at det er vanskeleg å sjå utifrå kjeldekoden kor stor “overhead” blir. Dette må ein gjere gjennom sjå på ein referansearkitektur, eller ein tidlegare implementasjon [**13**].

Det går raskt å profilere høgnivå kode, som til dømes C eller C++, men det har låg presisjon, mellom anna fordi høgnivå-språk skjuler ein del assembly-instruksjonar og optimalisering frå kompilator. Profilering av assembly-kode er presist, men det går sakte, dessutan kan det vere vanskeleg å gjere dette på eit tidleg stadium i designprosessen, fordi ein ikkje har definert arkitekturen og instruksjonsettet og difor ikkje veit heilt korleis instruksjonsettet kjem til å sjå ut [**22**]. Ein kan grovt sett seie at når eit språk har høgare nivå, altså meir abstraksjon, vil det gje eit mindre nøyaktig bilete av funksjonaliteten.

Resultatet av denne analysa nyttast seinare som inngongsparametere i dei neste stega i designet. [**21**] [**23**] [**13**]. Nokre dømer på slike parametere inkluderar:

- Kva slags datatypar som blir nytta og korleis desse aksesserast
- Frekvensen til grunnleggande operasjonar
- Gjennomsnittleg blokkstorleik
- Antal MAC-operasjonar
- Ratio mellom adressegenereringsinstruksjonar og databehandlingsinstrukjonar
- Kjerner i koden

Poenget med å hente ut desse verdiane er å kunne ta avgjersler på kva slags spesialiserte einingar som bør inkluderast i arkitekturen.

Korleis profilerar ein?:
Første steg er statisk profilering, dette går ut på at ein går gjennom koden utan å køyre den og skaper ein presentasjon av koden som ein kontrollflytgraf ("control flow graph") . Blokkene i grafen får tilknytta nokre verdiar for køyretid. Ved å gå gjennom grafen kan den totale kostnaden ved å køyre programmet. Dette synleggjer kjerner i koden, typisk brukar programmet 90% av køyretida på 10% av koden. I tillegg gjer statisk profilering at ein kan identifisere kritisk sti i koden, noko som viser den delen av koden som tek lengst tid /krever høgst yting [**13**].

Når ein har ein kontrollflytgraf må ein sjå på dei individuelle blokkene og profilere desse på aritmetisk nivå. Det vil seie at ein ser på kva aritmetiske og logiske operasjonar som blir nytta, kva slags og kor mange minneaksessar som blir nytta og også kommunikasjon mellom ulike blokker.

Når ein har gjort dette må ein byrje å dele opp funksjonaliteten i systemet i programvare og maskinvare. Ein funksjon som er realisert i maskinvare får si eiga funksjonelle eining og sin eigen instruksjon, medan ein funksjon som er realisert i programvare nyttar allereie eksisterande instruksjonar. For å få god nok yting aksellererar ein dei mest brukte funksjonane/kjernene i maskinvare og implementerar dei mindre brukte i programvare [**13**].

Ei analyse av kva datatypar som er nytta i ein applikasjon kan gje ein indikasjon på kva ord-storleiken i prosessoren bør vere og aksess-analyse kan gje indikasjonar på korleis minnehierarkiet bør vere oppbygd [**23**].

Her er det først viktig å lage gode parameteriserte modellar for dei moglege arkitekturane. Ein slik modell kan vere bygd opp av ei rekke funksjonelle einingar. Desse blir beskrivne ved hjelp av dei eigenskapane dei har, til dømes nyttar Gupta et al [**24**] tilknytta operasjonar og forseinkingseigenskapar , t.d. gjennomstrøyming og latens, for å beskrive einingar.

Med unntak av antal og type funksjonseiningar finnes det ikkje brei einigheit om kva parmametere som skal inkluderast i vurderinga av ein arkitektur og antal og kva type parametere vil påverke kor stort løysingsrommet for prosessorarkitekturen vil vere [**23**]. Nokre slike parametere er [**24**] [**23**] :

- Antal einingar av kvar type
- Antal registere
- Instruksjonsbreidde
- Lagringseiningar : storleik på cache for data og instruksjonar
- Koplingar mellom einingar
- Paralelle "load/store" operasjonar

For å velge ut den mest høvelege av arkitekturane må ein ha eit estimat over ytinga deira, slik at ein kan samanlikne. Dei to vanlegaste måtane for å oppnå dette er simulator-metoden og tidsplan-metoden ("scheduler"). Simulator er mest tidkrevande og difor blir ofte tidsplan-metoden føretrekt [**23**].

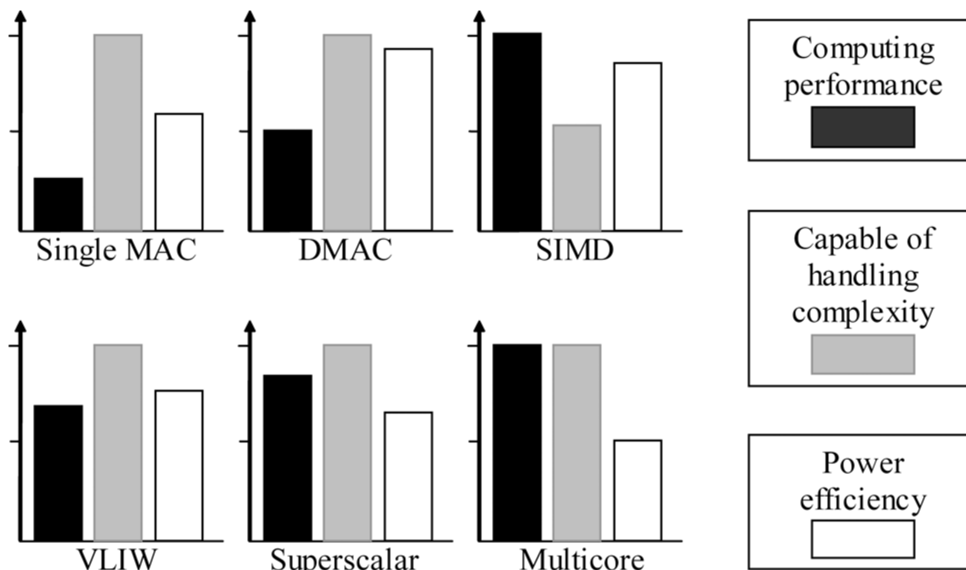
Tidsplan-metoden går ut på at ein ser på den oppgåva som prosessoren skal løyse som eit problem der det finnes begrensa resursar og tid til å utføre ei rekke operasjonar. Til dømes, om det finnes ei MAC-eining i prosessoren må ein planlegge korleis denne skal nyttast om det utføres fleire MAC-operasjonar i paralell. Ved å estimere/telle antal klokkesyklusar som

er nødvendige for å oppnå den ønska gjennomstrøyminga for ei gitt oppgåve kan ein også estimere kva klokkefrekvens som krevjast for å få til dette. Om denne estimerte klokkefrekvensen til dømes skulle bli for høg må ein kanskje introdusere ei til MAC-eining for å kunne møte krava [**21**].

For å designe **akseleratorar** kan ein oppnå gode resultat med ein sokalla **"oppgåve-flyt"**-arkitektur ("task-flow architecture"), også kjend som "funksjons-kartlagd"-maskinvare ("function-mapped hardware") [**13**].

2.3.2 Utforsking av det arkitektoniske løysingsrommet

Ein kan byrja med ein i utgangspunktet ganske generisk arkitektur, som til dømes dei i Figur 3 Gjennom gjentekne iterasjonar av instruksjonssettgenerering og utforsking av det arkitektoniske løysingsrommet kan ein etter kvart optimalisere arkitekturen slik at den oppfyller krava til applikasjonen [**13**].



Figur 3 : Arkitektur [13]

(DMAC- Dual Mac, SIMD – Single Instruction Multiple Data, VLIW – Very Long Instruction Word)

Figur 3 gjev eit overblikk over dei vanlegaste DSP-arkitekturar saman med grove estimat for yting, effektforbruk og evna til å handtere kompleksitet. Enkel –MAC-arkitekturen består av ei enkel databehandlingskjerne med ei enkel MAC-eining. Denne typen arkitektur var vanleg før, men det har vist seg at arkitekturar med større grad av parallellitet i dei fleste tilfelle har både betre ytelse og effektforbruk per instruksjon fordi dei gjer at ein kan ha ei lågare driftsspenning i systemet. Per i dag er det vanlegare å nytte seg av arkitekturar med fleire enn ei MAC-eining [**25**].

Typisk vil ein ta utgangspunkt i ein av desse formene for arkitektur når ein skal finne ein egna arkitektur. Det er vanskeleg å gjere ei formell vurdering av kva arkitektur som er mest høveleg for ein gitt applikasjon, løysingsrommet er svært stort. Ein må til ei viss grad stole på intuisjon og erfaring.

Når ein har vald ein av arkitekturane over kan ein byrje å sjå på korleis ein kan endre prosessoren for å tilpasse den applikasjonen. Gjennom den applikasjonsanalysa ein har gjort kan ein byrje å finne ut kva konsekvensar det vil ha å legge til eller fjerne funksjonalitet i prosessoren. Dette gjeld alle aspekt av arkitekturen, og ein må truleg gjere dette i fleire omgangar for å finne ein arkitektur som oppfyller krava til applikasjonen.

Begrepa VLIW, SIMD, multikjerne-arkitektur og superskalar arkitektur er greia ut i avsnitt 2.6 som tar for seg parallellitet. Valg av arkitektur er diskutert i

For "Ultra-lågeffekt"-applikasjonar er det mykje som tydar på at den beste løysinga er ein relativt enkel og liten DSP som opererer saman med mange dedikerte einingar som tek seg av dei ulike funksjonane [**25**]. Einingane tek seg av databehandlinga, medan DSP-kjerna tek seg av kontroll og behandlar "uventa" funksjonar, altså funksjonar som ikkje har dedikerte akseleratorar. Dette gjev minimalt effektforbruk, men det betyr også at ein har ganske lågfleksibilitet. Her fungerer DSP-en stort sett berre som ein kontroller som styrer dei ulike funksjonelle einingane. Ein annan måte å lage denne typen arkitektur på er å integrere akseleratorane i kjerna, direkte i datastien. Dette gjev ein mellom anna moglegheit for at dei ulike akseleratorane kan dele nokre funksjonelle einingar, til dømes multiplikatorar. Med andre ord kan dette gje mindre arealbruk [**25**].

(Re)konfigurerbar arkitektur

Ein metode for ASIP-design er sokalla "rekonfigurerbar arkitektur." Samanlikninga med å bygge prosessoren frå grunnen av utforskar denne metoden eit mindre design-rom, men kan gje raskare og meir effektiv utvikling. Metoden går ut på at ein nytter/ kjøper ein konfigurerbar prosessor frå ein produsent (t.d. ARC eller Tensilica) og konfigurerar den for eit spesielt bruksområde, anten med konfigurasjonar som er tilgjengeleg på førehand, eller med eigne programmer [**18**].

Sidan denne metoden ikkje har eit like stort designrom gjev den ikkje ein like interessant diskusjon og difor nyttast den ikkje i denne oppgåva, og vil ikkje beskrivast vidare.

2.3.3 Generering av instruksjonssett

Når ein har analysert applikasjonen og fått på plass ein grunnleggande arkitektur er det mogleg å byrje å utforme eit instruksjonssett.

Eit instruksjonssett består av:

- Aritmetiske instruksjonar og akselererte aritmetiske instruksjonar
- Minneaksess og adresseringsalgoritmar
- Instruksjonar for programkontroll
- Instruksjonar for I/U

Instruksjonssettet til ein ASIP består både av "standard-instruksjonar" som ein vil finne i ein kvar DSP, og også akselererte instruksjonar spesielt tilpassa applikasjonen. Instruksjonar og instruksjonssett blir omtalt grundig i kapittel 3.

2.4 Komponentar

I dette avsnittet presenterast nokre av dei typiske bestanddelane til ein DSP. Denne oppdelinga er gjort for å kunne gje eit meir oversikteleg og modulær bilete av prosessoren.

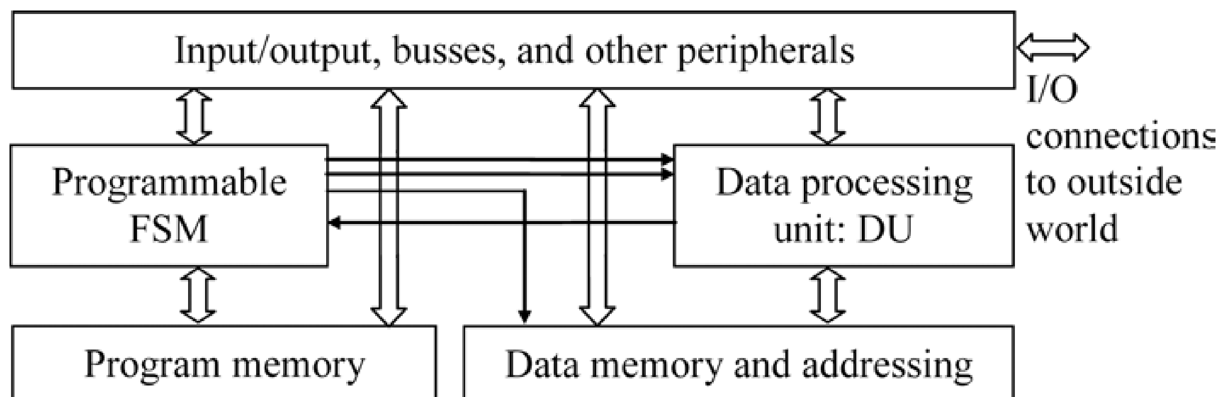
1 Programminne:
Programminne (PM) nyttast til å lagre programmer som skal utførast i prosessoren. PM er ein del av kontrollstien, og programma er lagra i maskinkode [**13**].

2 Kontrolleining:
Dette kan vere ei programmerbar tilstandsmaskin ("FSM") som består ein programteller ("program counter "- PC), forgreiningskontroll ("branch-control") og instruksjonsdekodar (ID) [**13**]. I tillegg er det vanleg å inkludere handtering av maskinvare-løkker, noko som er beskrive utfyllande i avsnitt 2.7.

3 Dataminne og dataadressering:
Dette minnet inneheld den informasjonen som skal prosesserast. Dette kan vere I/U-data, midlertidig data og koeffisientar og parameterar [**13**].

4 Datasti:
Denne eininga utfører dataprosseseringa av data. Det vil seie at den utfører aritmetiske og logiske operasjonar. I ein DSP består den minst av ei registerfil, ein MAC og ein ALU, men den kan ha fleire av desse og mange andre meir spesialiserte einingar [**13**].

5 Inngang/utgang(I/U):
Denne eininga tek seg av all interaksjon med eksterne einingar. Dette inkluderer minnebussar og perifere einingar [**13**].



Figur 4 : DSP-Oppbygnad [13]

2.4.1 Programminne

Programminnet ("PM") er eit minneområde som er ein del av kontrollstien og inneheld dei instruksjonane som dei programma som skal utførast av prosessoren består av. I tillegg til dette er det også mogleg å lagre koeffisientar som skal nyttast i datastien i PM [**13**].

For å unngå at ein må gjere mange kostbare minneaksessar til eit stort minne kan det lønne seg å introdusere ein cache mellom instruksjonsdekodaren og programminnet. I denne kan ein inkludere kjerna til det eller dei programma som prosessoren arbeidar med til ei kvar tid. Dermed kan ein redusere.

Storleiken til programminnet

PM må vere så stort at det inneheld alle dei applikasjonane som skal køyrast på prosessoren. Det må derimot ikkje vere så stort at det kan innehalde alle applikasjonar som kan køyrast. Dess større det er dess fleire applikasjonar kan ein ha tilgjengeleg i systemet samtidig. Kor mange dette er vil vere avhengig av kor stor køyretidsfleksibilitet ein ønskjer, med fleire applikasjonar lagra i systemet blir det mogleg å veksle mellom desse medan systemet køyrer. Om det held at systemet er fleksibelt i ved konfigurering, det vil seie når systemet blir produsert eller ved oppdatering, kan ein ha eit mindre PM med plass til berre ein eller nokre få program. Dette fører til ein lågare statisk effektforbruk fordi ein reduserer storleiken på minnet [**13**], men det vil også føre til lågare fleksibilitet ved køyretid.

2.4.2 Kontrolleining

Kontrolleininga kan reknast saman med programminnet som ein del av ein "kontrollsti" i DSP-en. Den består i hovudsak av ein programteller, instruksjonsdekodar, løkkekontroll og forgreiningskontroll.

Programtellaren er eit register som lagrar minneadressa til den neste instruksjonen som skal utførast. Etter denne instruksjonen er lest av inkrementerast programtellaren med '1', og adresserar dermed den neste instruksjonen som skal utførast. Når programtellaren kjem til ein "hopp"-instruksjon blir ei ny adresse lasta inn og det inkrementerast frå denne til neste gong den møter eit "hopp". Om prosessoren får instruksjon om å byrje på ei subrutine, eit "program-i-programmet", stoppast inkrementeringa og adressa i programtellaren blir lagra i "stacken." Når subrutina er ferdig, det vil seie at den får ein "return"-instruksjon, lastast den lagra adressa tilbake frå "stacken" og ein kan fortsette å utføre instruksjonane frå der ein avbraut dei [**26**].

Instruksjonsdekodaren les inn instruksjonar lagra i instruksjonsregisteret og gjer desse om til kontrollsignaler for resten av prosessoren, dvs. kontrollstien, datastien og minnet.

Forgreiningskontrollaren er ei eining som kontinuerleg evaluerer ein eigen verdi, gitt ved starten av ei forgreining, med ein verdi i eit av statusregistera. Om dei gitte føresetnadane inntreff, som at verdien i statusregisteret er lik den gitte verdien i forgreiningskontrollaren, vil eininga gje løkkekontrolleren beskjed om å utføre ei ny forgreining [**27**].

Løkkehandtering

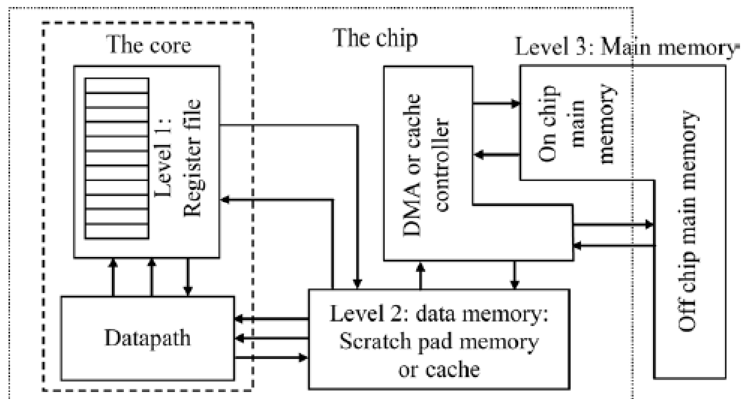
Ein spesiell eigenskap ved DSP-ar er at dei kan handtere løkker meir effektivt enn generelle prosessorar og mikrokontroller. Spesiell maskinvare gjer at ein kan køyre løkker utan nokon ekstra sykluskostnad, dette vil seie at ein kan gjenta ei rekke på 1-N instruksjonar utan å måtte reinitialisere ein løkketeller [**28**] [**25**].

Adressegenereringseining (AGU):

Som namnet antyder er dette ei eining som brukast for å generere adresser til minneområder i prosessoren. Grunnen til at det er naudsynt å ha med ei slik eining er at det er vanskeleg og upraktisk å adressere minne direkte i instruksjonar, og det forverrar gjennomstrøyminga i datastien til prosessoren betrakteleg om ein nyttar den til å rekne ut minneadresser. AGU-en kan operere i parallell med datastien og gjer difor at ein ikkje treng å bruke ekstra klokkesyklusar på å generere adresser [**30**].

2.4.3 Dataminne og dataadressering

Dataminnet er eit minneområde som inneheld den dataen som programma som køyrer på prosessoren skal bruke. Dataminnet kan vere delt opp i eit hierarki med ulike nivå som har ulik storleik og nærleik til datastien. Dei minnenivåa som ligg nærast datastien er dei minste i areal og dei krever minst tid for å aksessere. Det betyr at det minnet som ligg "nærast" datastien (Nivå 1) vil ha lågast latens og lågast energikostnad kopla til aksess. Dette nivået består av register, nokre av dei for generell bruk og nokre for spesielle formål.



Figur 5: Døme på minnehierarki

Scratchpad- eller Cache-basert minne

Over dette nivået finnes det eit høgare nivå som i tilfellet DSP oftast består av ein cache, også kalla "hurtigbuffer", eller ein sokalla "scratchpad", også kalla "arbeidsminne." Begge er relativt små og raske i forhold til eit hovudminne, men det er fleire ting som skiljer dei frå kvarandre:

	Scratchpad	**Cache**
VIRKEMÅTE	- Ein scratchpad består av ein liten fast del av minneadresserommet. - Resten av adresserommet består av eit hovedminne.	- Cache utnyttar det faktum at dataen som har temporær lokalitet ofte er samla i samme område på minnet. - Data frå eit område i det relativt store, effekt-forbrukande og trege hovedminnet blir lasta inn i ein mindre og meir effektiv cache.
DATA	- Data som ikkje finnes i hovedminne. - Ofte midlertidig data	- Koherent kopi av data i hovedminnet - Medfører ekstrakostnad i forhold til scratchpad for å flytte data til og frå hovedminnet.
PLANLEGGING	- Brukar/kompilator handterar lokalitet og kommunikasjon - Stiller høge krav til effektiv kompilator og programmerar - Låg kompleksitet - Deterministisk aksesstid - "Bommar" aldri på minneaskessar	- Ikkje synleg for brukar/kompilator - Kan "bomme" på cacheforespørslar - Dette medfører udeterministisk oppførsel
EFFEKT- og AREAL-FORBRUK	- Banakar et. al. [**29**] kan vise til ein gjennomsnittleg reduksjon i effektforbruk for eit sett applikasjonar på 40% i forhold til cache samt eit gjennomsnittleg arealreduksjon på 46%.	- Nyttar omlag 25%-45% av effekta i ein prosessor og opp til 50% av arealet.
BRUKSOMRÅDE	- Integrerte system - Kritisk areal- og effektforbruk	- Generelle prosessorar - Der ein treng forenkla programmering og bruk.

Tabell 1

Tabell samansett frå [**16**] [**29**].

Fokuset i presentasjonen over er på dataminne, men begge metodane er også aktuelle for programminne. For å kunne diskutere kva som er mest egna må ein først sjå på korleis applikasjonen oppfører seg både med tanke på dataflyt og programstorleik.

2.4.4 Datasti

Datastien er der dataen blir behandla. Den består typisk av ein eller fleire aritmetisk-logiske einingar (ALU), ein eller fleire multipliser-akkumuler-einingar (MAC) og ei registerfil. I ein DSP vil den typisk også innehalde ei rekke akselleratorar, det vil seie eininar som er spesialiserte for eit eller nokre få føremål og som kan gjennomføre oppgåvene knytta til desse på ein særskild effektiv måte.

ALU:

Som namnet antyder utfører ALU-en aritmetiske og logiske operasjonar. Den får data frå register i prosessoren, behandlar desse etter instruksjonar den får frå kontrolleininga og lagrar resultatet i eit utgangsregister. Dei fleste ALU-ar kan utføre integeroperasjonar, som til dømes addisjon og subtraksjon, logiske bit-operasjonar, som AND, NOT og XOR og skiftoperasjonar [**26**].

Registerfil:

Registerfila er ei samling av ei rekke register. Desse lagrar data, adresser og statusinformasjon som er nødvendig for at prosessoren kan køyre. Desse registera inkluderer vanlegvis: programteller, instruksjonsregister, statusregister, stakkpeikarar og nokre generelle register [**26**].

Instruksjonsregisteret held instruksjonane medan dei blir utførd, bufferar koplar saman prosessoren med minnesystemet, ved at dei held midlertidig data. Statusregisteret blir brukt til å vise informasjon om ALU-operasjonane som blir utførd. Typiske døme på slik informasjon er “carry”, som viser at ALU-operasjonen skal ta med seg ein verdi vidare til neste operasjon, “overflow”, “null” og “negativ [**26**].”

Stakken som er omtalt ovanfor blir nytta til å lagre data frå andre register når desse skal nyttast til andre formål [**26**].

MAC (“Multiply Accumulate”):

Ei lang rekke DSP-applikasjonar har sokalla MAC operasjonar i kjerna si. Dette strekker seg frå filtrering i tidsdomenet (IIR og FIR) til mellom anna konvulsjon og transformar (“FFT, DFT mfl.”). For å få ein effektiv DSP-arkitektur bør ein då inkludere eigne spesialiserte MAC-einingar. Ein MAC-operasjon, t.d. utfører “MAC A,B,C” operasjonen A = A + B*C. Som namnet tilseier multipliserer den to tal og akkumulerer resultatet mellom dei [**12**].

Talet MAC-einingar i kjerna må vere tilpassa dei applikasjonane som skal køyre på DSP-en. Det må vere mange nok til å kunne utnytte parallelliteten i dataen som prosesserast.

Ei MAC-eining kan til dømes bestå av ein 16-bits multiplikator og ein 40-bits akkumulator (Analog Devices Blackfin BF5xx). Akkumulatoren må vere større enn multiplikatoren for å sikre mot “overflow” [**28**].

2.4.5 I/U – DMA, avbruddsdriven I/U eller programert I/U

Denne applikasjonen krever som dei aller fleste sanntidsapplikasjonar at ADC og DAC må kunne skrive og lese frå minnet med jamne eller ujamne mellomrom. I dette tilfellet vil ein ADC skrive til minnet ein gong per sampleperiode.

Ein måte å gjere dette på er å bruke prosessorkjerna og la den gjennomføre dette som vanlige “load-store” operasjonar, noko ein kan kalle avbrotsdriven I/U (“interrupt -drive”) eller

programmert I/U alt etter som det henhaldsvis fører til at den inneverande operasjonen i prosessoren avbrytas eller ikkje [**30**].

DMA – Direkte minneaksess

Det andre alternativet er å nytte DMA eller , denne er beskriven tidlegare i avsnitt Dette er ein minnearkitektur som gjer at einingar i systemet kan få direkte tilgang til minne utan å gå via prosessoren. Dette gjer at prosessoren har færre arbeidsoppgåver, og det er svært vanleg å nytte i DSP [**12**].

I eit system utan DMA blir det brukt eit antal operasjonar per byte som skal flyttast. Antalet er avhengig av korleis instruksjonssettet ser ut, men det vil generelt sett krevje at instruksjonen lastast inn, dekodast og data flyttast. I prosessorar som ikkje kan lagre og hente ut informasjon vil dette måtte gjerast over to instruksjonar. For å unngå dette kan ein introdusere ei eining som gjer at ulike einingar i systemet kan flytte data mellom seg utan å nytte prosessorkjerna. Dette gjer at ein unngår mange unødvendige operasjonar og dermed kan overføre data meir effektivt. Flyttinga skjer anten ved at ein programmerar dei, typisk flytting frå ein minneregion til ein annan, eller ved at eksterne einingar "krever" den. Den skjer anten i faste blokker som nyttar ein "buss-aksess" for å å overføre ei blokk, eller i store bolkar der ein brukar ein "buss-aksess" for å flytte over ei rekke blokker etter kvarandre [**30**].

2.5 Akseleratorar

Akseleratorar er funksjonelle einingar som nyttast i prosessoren i tillegg til standardeiningane som må vere til stades for at prosessoren skal fungere. Akseleratorane gjer at ein kan auke ytinga til prosessoren når den utfører visse typar instruksjonar og program. Dei kan grovt sett delast opp i to ulike nivå, **operasjonsnivå** og **funksjonsnivå**.

Når ein akselerator er på operasjonsnivå er den laga for å akselerere ein grunnleggande operasjon som kan nyttast av mange ulike funksjonar. Eit typisk og veldig vanleg døme på dette er MAC, som er kort beskrive tidlegare i avsnitt 2.4.4.

Ei særs nyttig akselerator er CORDIC, som er eit akronym for "Coordinate Rotation Digital Computer". Sidan denne har mange andre bruksområder enn akkurat dette er den presentert under, medan dei andre approksimasjonane av arcus tangens er representert seinare

2.5.1 CORDIC

CORDIC er eit sett algoritmar som berre nyttar seg av skift- og addisjons-operasjonar og kan utføre ei rekke ulike trigonometriske, hyperbolske, lineære og logaritmiske funksjonar. For å gjere dette nyttar dei seg berre av skift- og addisjonsoperasjonar. Dei opererer i to forskjellige modi: vektormodus og rotasjonsmodus.

I vektormodus roterast inngangsvektoren til den ligger likt med x-aksen, den gjer dette ved å forsøke å minimere y-komponenten ved kvar iterasjon. Resultatet av operasjonen er ein verdi for vinkelen [**31**].

CORDIC baserar seg på den generelle rotasjonstransformasjonen, gitt ved:

$$x' = x\cos(\theta) - y\sin(\theta)$$

Formel 2

$$y' = x\cos(\theta) - y\cos(\theta)$$

Den kan skrivast om til:

$$x' = \cos\theta\,[x - y\tan(\theta)]$$

Formel 3

$$y' = \cos\theta\ \ [y + x\tan\theta]$$

Om ein begrensar tan(θ) til verdiane $\tan(\theta) = \pm 2^{-i}$, der i er rotasjonsiterasjonen får ein:

$$x_{i+1} = K_i[x_i - y_i d_i 2^{-i}]$$

Formel 4

$$y_{i+1} = K_i[y_i + x_i d_i 2^{-i}]$$

Der

$$K_i = \cos\left(\tan^{-1} 2^{-i}\right) = \frac{1}{\sqrt{1 + 2^{-2i}}}$$

Formel 5

og $d_i = \pm 1$.

Som ein kan sjå av

Formel 4 utgjer dette med unntak av K_i ein skift og ein addisjon per iterasjon. Som det er framgår av formlane over består prosessen i grunn ut på å gonge den noverande vinkelen med ein ny vinkel "i" gongar. Denne vinkelen blir stadig mindre, og dess fleire gongar ein gjer dette dess nærare kjem ein x-aksen. Den opprinnelege vinkelen er summen av alle desse vinkel-multiplikasjonane. For å finne vinkelen med denne metoden må ein anten bruke ein oppslagstabell eller ein akkumulator. Når ein nyttar oppslagstabell er vinkelen representert ved ein vektor som er samansett av sekvensen av retningane til rotasjonane og ein tar berre vektoren og slår opp på den tilsvarande vinkelen i ein tabell som er programmert på førehand [**32**].

Når akkumulator blir initialisert med verdien null vil den innehalde den traverserte vinkelen etter siste iterasjon. Akkumulasjonen ser slik ut når z representerer vinkelen:

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i})$$

Formel 6

Både vektormodus og rotasjonsmodus nyttar likningane i

Formel 4 og Formel 5, men dei skil seg på definisjonen av d_i. For vektormodus er d_i definert som [**32**]:

$$d_i = 1\ viss\ y_i < 0\ , ellers\ er\ d_i = -1$$

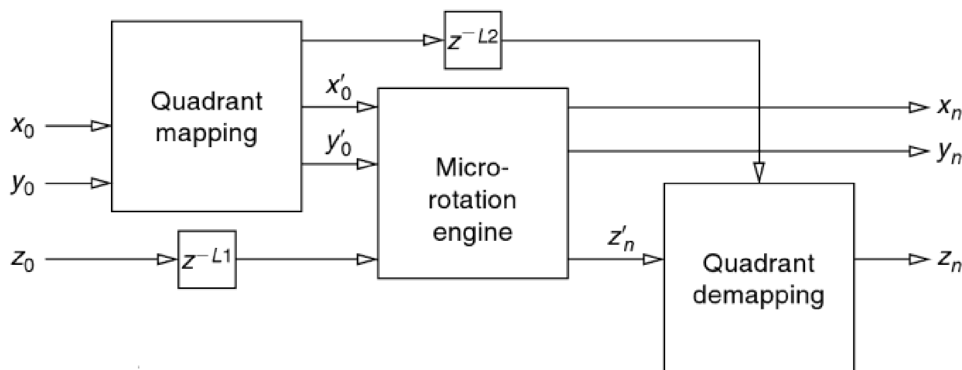
for rotasjonsmodus er :

$$d_i = -1\ viss\ z_i < 0\ , ellers\ er\ d_i = -1$$

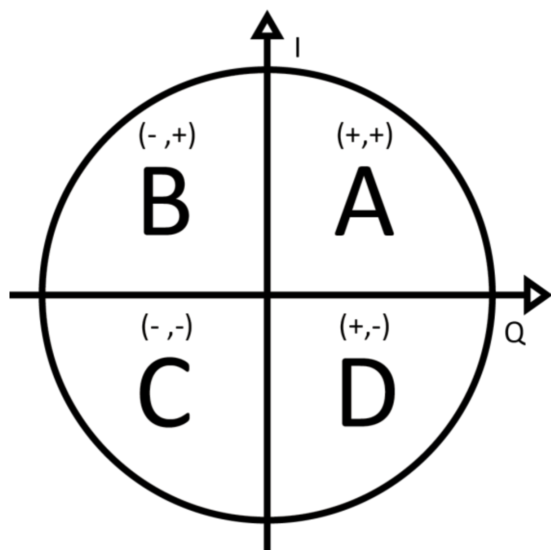
Eit problem er at dei likningane som brukast kun fungerar i området –π/2 til π/2. Dette er ikkje akseptabel og derfor er det første steget i ein CORDIC-operasjon er å undersøke kva kvadrant vektoren ligger i. Om den ligger i kvadrant B eller C, som vist i Figur 7, må ein flytte dei til A eller D for å kunne rekne ut vinkelen. Dette kan ein enkelt gjere ved å sjå på forteikna til I og Q og legge til ein vinkel etter ein har fullført rotasjonane, ±π avhengig om I er positiv eller negativ [**33**].

Denne flyttinga må kompenserast for når ein er ferdig med rotasjonane. Dette kan ein oppnå ved å legge til ei eining før og etter CORDIC-eininga som tek seg av å rotere vektoren til korrekt kvadrant og legge den til igjen når ein har gjort ferdig CORDIC-iterasjonane. På Figur 6 ser ein korleis ei kan løyse dette. Her er Z fasen og x og y er koordinatane til vektoren.

Det er mogleg å utføre CORDIC i ein standard ALU utan akselerator, men på grunn av mange forgreiningar ("branch") krever dette eit relativt stort antal klokkesyklusar [**27**]. Difor vil det vere naturleg å implementere denne som ein akselerator.



Figur 6: Kvadrantjustering



Figur 7: Kvadrantar for CORDIC-operasjonar

Kvart steg svarar til ein stadig mindre vinkel, denne er gitt av [**34**]:

$$\theta_n = \arctan(\frac{1}{2^n})$$

Formel 7

For dei første åtte iterasjonane gjev det verdiane:

STEG (n)	arctan()	Vinkel i grader	Vinkel i radianer	Maks. feil i rad.
1	arctan(2^0)	45,00	0,7854	0,7854
2	arctan(2^{-1})	26,5651	0,4636	0,4636
3	arctan(2^{-2})	14,0362	0,2450	0,2450
4	arctan(2^{-3})	7,1250	0,1244	0,1244
5	arctan(2^{-4})	3,5763	0,0624	0,0624
6	arctan(2^{-5})	1,7899	0,0312	0,0312
7	arctan(2^{-6})	0,8952	0,0156	0,0156
8	arctan(2^{-7})	0,4476	0,0078	0,0078

Tabell 2

Som ein ser byrjar ein med ein vinkel på 45 grader og så fortsetter ein med stadig mindre vinklar. Dermed ser ein at resultatet blir meir nøyaktig for kvar iterasjon. Ein oppnår eit bits presisjon per iterasjon [**35**], det vil seie at når ein roterar eit åtte bits ord er den maksimale nøyaktigheita oppnådd etter åtte iterasjonar. Som den siste kollonna viser er den maksimale feilen for ei faseutrekning etter n iterasjonar den same som den vinkelen ein dreiar med. Dette er intuitivt fordi den største feilen som er mogleg vil seie at ein har hamna akkurat på vinkelen 0 i den førre iterasjonen. Dette er vist i eit matlab-program i Vedlegg 1.6.

Dette avsnittet har for det meste tatt føre seg korleis CORDIC kan nyttast til å berekne arcus tangens, dette er fordi denne applikasjonen er viktig med tanke på dei demodulasjonsmetodane som blir presentert seinare. Utover dette kan CORDIC nyttast til å approksimere ei lang rekke ulike funksjonar. Desse er vist iTabell 3 [**32**] på neste side.

Funksjon	Beskriving	Formel
Sinus og cosinus	Ved å sette "y-inngangen" til 0 kan ein rekne ut sinus og cosinus samtidig. Ein kan notere seg at ved å sette x_0 til $1/A_n$ vil ein kunne få ein uskalert versjon av resultatet	$x_n = A_n x_o \cos z_0$ $y_n = A_n x_o \sin z_0$
Polar- til kartesisk-koordinat-transformasjon	Ved å bruke rotasjonsmodus og sette x_0 = polar lengde (el. "Magnitude") z_0 = polar fase og $y_0 = 0$, kan ein konvertere frå polare til kartesiske koordinatar	$x = r \cos \theta$ $y = r \sin \theta$
Kartesisk- til polar-koordinat-transformasjon	Ved hjelp av vektor-modus kan ein finne lengda og vinkelen til ein vektor. Dette utgjer polarkoordinaten	$r = \sqrt{x^2 + y^2}$ $\theta = \operatorname{atan}(y/x)$
Generell vektor-rotasjon	Rotasjonsmodus kan i si enklaste form brukast til å rotere vektorar med ein ønska vinkel. Dette kan vere svært nyttig i rørsle-styringssystemer	$\theta_{ut} = \theta_0 + \theta_1$
Utrekning av vektorlengde	Som nemnt i samband med beskrivinga av utrekning av arcus tangens vil ein av utgangsverdiane i denne prosessen vere storleiken, eller lengda til den vektoren ein finn vinkelen til. Etter rotasjon vil vektoren ligge langs x-aksen, og x-komponenten til denne vil difor tilsvare lengda til den opprinnelege vektoren. Systemet vil også ha ei forsterking A_n som ein må ta hand om i ein annan del av systemet	$x_n = A_n \sqrt{x_0^2 + y_0^2}$
Arcsin og arccos	På same måte som ein kan finne arctan kan ein også finne den inverse av sinus og cosinus ved hjelp av CORDIC. Metoden går ut på å starte med ein vektor som ligger langs x-aksen og rotere denne til den er lik ein vektor gitt av inngongen på CORDIC-eininga. Arcus sinus kan dermed estimerast ved å sjå på den totale rotasjonen som er gjort for å oppnå likheit mellom vektorane. Arc cos er noko meir komplisert, sjå [**32**] for meir informasjon.	$\theta_n = \theta_0 + \sin^{-1}\left(\frac{c}{A_n x_0}\right)$
Lineære funksjonar	I rotasjonsmodus kan CORDIC nyttast som ein multiplikator, men den er ikkje like effektiv som ein dedikert multiplikator. Kan vere nyttig i tilfelle der ein av andre grunnar har implementert CORDIC og ikkje ein dedikert multiplikator	$x_n = x_0$ $y_n = y_0 + x_0 z_0$ $z_n = 0$
	I vektormodus kan CORDIC nyttast for å samanlikne skilnaden i storleik, eller ratioen, mellom to inngangsverdiar.	$x_n = x_0$ $y_n = 0$ $z_n = z_0 - \frac{y_0}{x_0}$
Hyperbolske funksjonar	Dei hyperbolske utgåvene av dei tidlegare nemnte trigonometriske funksjonane kan approksimerast med CORDIC. I tillegg kan ein approksimere tan, tanh, exp og ln. Sjå [**32**] for meir informasjon.	

Tabell 3

2.6 **Parallell databehandling**

Signalprossesseringsapplikasjonar involverer ein stor grad av parallellitet i data. Det vil seie at det er mange ulike "datastraumar" som skal behandlast med dei same operasjonane [**12**]. Derfor er det typisk mange like einingar i DSP-en. Til dømes meir enn ein MAC, ALU, DMA osb. [**28**]. Prosessorar av denne typen kan kallast parallelle prosessorar.

Motsetnaden til dette er ein skalar prosessorarkitektur. I denne forma for arkitektur utstedar ein **ein** ny instruksjon i kvar klokkesyklus og utfører den på **ein** datastraum. Det finnes berre ei eining tilgjengeleg av kvar type. Dette er den minst komplekse av dei moglege arkitekturane, og det er dermed rimeleg å anta at den nyttar minst areal [**12**].

2.6.1 **Samlebandsarkitektur**

Samlebandsteknikkar ("pipelining") går ut på å designe systemet slik at ein kan utføre mange oppgåver samtidig på ulike einingar. For prosessoren som heilheit vil dette seie at ein kan utføre ulike operasjonar samtidig, til dømes kan ein hente ei adresse samtidig som ein utfører ein multiplikasjon. Inne i einingar vil ein til dømes kunne dele opp ein prosess i mange "steg" ved å introdusere forseinkingar mellom ulike delar av prosessen. Dette fører til at ein ikkje treng like høg klokkehastigheit, men ein kan framleis få ut eit resultat per klokkeperiode, desse blir berre forseinka med like mange klokkeperiodar som det er steg i samlebandet [**12**].

For RISC-prosessorar er det typisk å implementere samlebandsarkitektur som fire eller fem steg: Hent ("Fetch"), Dekod ("Decode"), Les("Read"), Utfør("Excecute") og Skriv ("Write Back") [**12**].

	t1	t2	t3	t4	t5	t6	t7
Inst. 1	Hent	Dekod	Les	Utfør	Skriv		
Inst. 2		Hent	Dekod	Les	Utfør	Skriv	
Inst. 3			Hent	Dekod	Les	Utfør	Skriv
Inst. 4				Hent	Dekod	Les	Utfør

Figur 8: Døme på samleband

Figuren over viser korleis eit slikt samleband kan vere. Denne utfører fire instruksjonar, kvar er representert ved ei linje i diagrammet, merka Inst. 1-4, kvar klokkesyklus er merka med t1-7. Den antar at desse instruksjonane er atomiske, det vil seie uavhengige av kvarandre. Om dette ikkje var tilfellet og til dømes instruksjon 2 var avhengig av instruksjon 1 måtte den ha venta til instruksjon 1 er fullstendig utført før den kan byrje med "les"-operasjonen fordi denne kan bli endra av instruksjon 1.

Fordelen med å nytte samleband blir tydeleg når ein ser på antal instruksjonar som blir utførd per klokkesyklus. Om ein ikkje nyttar samleband vil kvar instruksjon bruke fem

klokkesyklusar før ein ny kan settast i gong. Det vil seie at ein instruksjon blir ferdig kvar femte klokkesyklus. Med samleband vil ein kunne ha ein instruksjon ferdig per klokkesyklus. Med andre ord vil ein med samleband kunne fullføre fem gongar så mange instruksjonar per klokkesyklus som utan [**12**].

Utover denne bruken av samleband kan ein også nytte den internt i akseleratorar. Dette gjer ein fordi ein vil redusere klokkehastigheita som er naudsynt for å få ønska gjennomstrøyming (“troughput”). Ein kan til dømes tenke seg ein akselerator som utfører fire multiplikasjonar og to addisjonar serielt.. Dette gjer at det blir lågare krav til klokkefrekvens, men det introduserer ein forseinking, eller latens, på eit antal klokkesyklusar som tilsvarar antal steg i samlebandet [**12**].

Ulempa med samleband er at det medfører større kompleksitet, fordi det krever mellom anna at ulike instruksjonar ikkje nyttar ulike resursar samtidig, som til dømes minne og aritmetiske einingar. Som tidlegare nemnt vil også avhengigheit mellom ulike instruksjonar komplisere implementasjon av samleband. Dette kan til dømes vere tilfellet om to instruksjonar behandlar same variabel. I tillegg til denne typen problem, sokalla “data hazards”, kan det også oppstå problemer og konflikter med kontrollflyt (“control hazards”) og strukturelle problemer (“structural hazards”), som er når arkitekturen ikkje klarar å støtte alle moglege kombinasjonar av instruksjonar [**12**].

Samlebandet frå Figur 8 er relativt minimalt, og det er mogleg å utvide det til mange fleire steg. Potensielt kan ein auke gjennomstrøyminga til ein prosessor ved å auke antal steg i samlebandet, men det finnes ein del begrensingar på lengda. Forseinking er naturlegvis eit problem sidan kvart nytt steg vil forseinke bitstraumen med ein klokkeperiode. I tillegg er “overhead” eit problem, det er på grunn av at register mellom stega må ha tid til å stabilisere seg og på grunn av sokalla “clock skew [**12**].”

Det er også mogleg å nytte variabel lengde i samlebandet, på denne måten kan ulike instruksjonar ha ulikt antal steg, noko som kan gi betre effektivitet om det er stor skilnad på dei ulike instruksjonane. Ulempa med dette er at det aukar kompleksiteten til prosessoren.

2.6.2 Ulike parallelle prosessortypar

SIMD

Enkel instruksjon fleire data (“Single Instruction Multiple Data” SIMD) vil seie at ein utstedar ein instruksjon per klokkesyklus, men utfører den på fleire datastraumar parallelt. Dette er praktisk for digital signalbehandling fordi dataen oftast er parallell i natur, og det gjev også moglegheit for relativt liten kodestorleik fordi ein berre treng ein instruksjon for å utføre mange operasjonar [**12**].

MIMD

“Multiple Instruction Multiple Data” baserar seg på ein arkitektur som i prinsippet består av to eller fleire uavhengige prosessorkjerner eller prosessorar. Desse får då sendt ut ulike instruksjonar som kan vere frå same program. Denne typen prosessor er typisk for moderne PC-ar, og er nok meir velegna for generelle system enn for digitale signalprosessorar [**12**].

VLIW

Veldig lange instruksjonsord (“Very Long Instruction Word” - VLIW) utnyttar parallellisme på instruksjonsnivå for å kunne køyre mange instruksjonar parallelt på ei enkel kjerne. Metoden går ut på at kompilatoren kodar mange instruksjonar inn i eit langt

instruksjonsord. Dette gjer at ein kan utføre fleire instruksjonar per instruksjonssyklus, og dermed auke effektiviteten til prosessoren. Det flyttar også ansvaret for oppgåveplanlegging frå prosessoren til kompilatoren. Det gjer det enklare å utnytte parallellitet i dataen som skal behandlast [**12**].

Det er vist at DSP-ar med høg parallelitet, til dømes fleire MAC-einingar, har lågare effektforbruk i forhold til DSP-ar utan moglegheit for parallelitet [**25**], men med samme gjennomstrøyming. På grunn av dette er det naturleg å vurdere VLIW. Ulempa og begrensinga til denne metoden er at på grunn av den store lengda på instruksjonane blir energikostnaden til kvar minneaksess svært høg. I tillegg vil det føre til auka kompleksitet og arealbruk og dermed også meir effektlekasje i forhold til prosessorar som sender ut ein instruksjon per klokkesyklus. Om ein skal få ein positiv effekt av denne typen parallellisering er ein prisgitt at ein kan utnytte den ekstra kapasiteten det medfører.

Superskalar prosessor

Som VLIW nyttar superskalare prosessorar seg av ILP for å kunne køyre fleire instruksjonar på ei enkel kjerne samtidig. For å redusere den relativt store kodestorleiken til VLIW-prosessorar kan ein nytte ein superskalar arkitektur. I denne hentar ein fleire sekvensielle instruksjonar per klokkesyklus. Desse blir så fordelt utover i systemet etter at ein har løyst problem med eventuelle avhengigheiter. Data-avhengigheiter behandlast dynamisk av prosessoren, det vil seie at kompilatoren ikkje treng å ta hensyn til dei. Dette betyr at ein i forhold til VLIW flyttar kompleksitet frå kompilatoren til maskinvaren [**12**].

2.7 Løkkehandtering

Digitale signalprosessering ber i stor grad preg av løkker. Løkker er seriar av instruksjonar som gjentakast fleire enn ein gong med lita eller inga forandring. For at ein DSP skal fungere så effektivt som mogleg bør ein ta spesielle hensyn til denne eigenskapen i DSP-applikasjonar og implementere effektiviserte måtar å ta seg av desse løkkene. Effektivisering vil i dette tilfellet seie at ein prøvar å oppnå løkker som ikkje har noko “overhead”, det vil seie at systemet skal kunne handtere løkker, og kanskje også løkker inne i andre løkker (“nested loops”), utan at dette påverkar korleis dei andre arbeidsoppgåvene til prosessoren køyrer og utan å bruke ekstra klokkesyklusar i datastien. Handtering av løkker består av å avgjere om ein skal fortsette å inkrementere programtellaren, det vil seie at ein går vidare med neste instruksjon i minnet, eller om ein skal hoppe til ein annan plass (“branch”) [**36**] [**37**].

2.7.1 Løkkehandtering i programvare

Når løkker blir handtert i programvare, utan nokon form for støtte vil løkker bli handtert med vanlege branch-instruksjonar i programmet. Dette betyr at for kvart steg i ei løkke må ein inkludere instruksjonar som inkrementerar ei løkketellar som ligger i eit register samt instruksjonar som sjekkar om denne løkketellaren har nådd ein verdi som gjer at ein er nødd til å utføre ein “branch.” Som dette tydeleg viser vil løkker medføre eit stort antal instruksjonar som tek opp plass og klokkesyklusar i samlebandet. Når ein i tillegg tek med at samlebandet må tømmast ved ein kvar “branch”-operasjon blir det tydeleg kvifor det er ønskeleg å implementere løkkehandtering i maskinvare [**37**].

2.7.2 Løkkehandtering i maskinvare

Dedikert maskinvare gjer at ein kan utføre løkketellerinkrementering, sjekking av teller og forgreiningar parallelt med at datastien utfører andre instruksjonar. Dette gjer at datastien ikkje treng å bruke ekstra klokkesyklusar på å handtere desse oppgåvene. Dette har naturleg nok store fordelar med tanke på effektforbruk og gjennomstrøyming [**36**]. Handtering av løkker i maskinvare kan gjerast på ein del ulike måtar, dei vanlegaste er enkeltinstruksjonsløkker, instruksjonsblokk-løkker og løkke-i-løkke-handtering (“nested looping”) [**37**].

Enkeltinstruksjonsløkke-handtering

Mange, om ikkje dei fleste, signalprosesseringsprogram inneheld løkker der ein instruksjon blir gjentatt ei rekke gongar og utfører den samme oppgåva på ulike data. Dette gjerast typisk ved at ein opprettar ein eigen tellar for ein instruksjon og dekrementerar denne kvar gong instruksjonen blir køyrd. Medan dette skjer stoppar ein programtellaren og denne blir ikkje starta igjen før den dedikerte instruksjonstellaren når null. Effekten av dette er at instruksjonen som skal gjentakast blir “halde” av programtellaren til den byrjar å inkrementere instruksjonar igjen. Informasjon om antalet iterasjonar som skal gjennomførast finnes i operanden på den instruksjonen som skal utførast [**37**].

Ein implementasjon av denne typen vil typisk bestå av ein tellar, som teller anten opp eller ned, eit register som inneheld antal iterasjonar som skal utførast og ein samanliknar som avgjer om løkka skal avsluttast [**27**]. Dette tilseier at denne typen løkkehandtering ikkje vil vere særskild plasskrevande, men det storleiken vil avhenge av kor mange iterasjonar ein må ha støtte for.

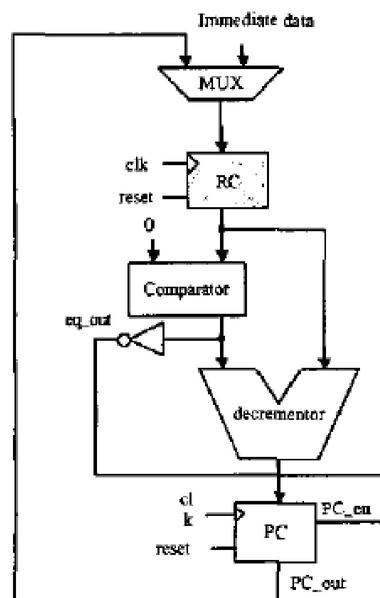


Fig. 2. The hardware looping block diagram for a single instruction.

Figur 9: Løkkehandtering for enkeltinstruksjonar

Instruksjonsblokkløkkehandtering

Om viktige løkker består av meir enn ein instruksjon per iterasjon vil ikkje eit system som det presentert i førre avsnitt kunne by på noko særleg betring i yting. Ein må då implementere ein løkkehandterar som kan repetere større delar av koden. Dette gjerast gjerne ved at ein lagrar start og sluttadressa samt ein teller for antall iterasjonar som skal gjennomførast. Etter kvar iterasjon dekrementerar løkkehandteraren tellaren og undersøker om denne har nådd null. Om den har det forsett ein som vanleg ved å inkrementere programtellaren. Om den ikkje har det hoppar ein tilbake til den første instruksjonen ved å laste inn den lagra startadressa i programtellaren [**37**].

Om denne løkka er kort nok kan det lønne seg å implementere ein buffer som held instruksjonane som løkka består av. Dette gjer at ein slepp å gjere kostbare kall til det relativt store programminnet, og dette vil spare effektforbruk og potensielt øke ytelsen til systemet, fordi ein får mindre latens. Ulempa er at løysinga er meir kompleks, fordi ein må ha logikk som kan laste data inn og ut av bufferen, samt ha ekstra minne for bufferen. Dette kan potensielt gje høgare statisk effekt, men det vil truleg bli vege opp av dei fordelane det gjev i forhold til å gjere mange aksessar på det store programminnet [**37**].

Løkke-i-løkke-handtering

Om programmet ein skal køyre inneheld løkker inne i andre løkker kan det lønne seg å ta høgde for at også desse bør handterast av løkkehandteraren. Metoden for dette er lik den beskrive over for instruksjonsblokker, men den utvidar antalet start- og stopp-adresser samt løkketellarar ein kan ha. Desse blir lagra på samme måte som på blokker, men i form av ein LIFO-stakk som gjer at ein behandlar ferdig den “indre” løkka før ein fortsetter på dei “utanfor.” Kor mange løkkenivå løkkehandteraren kan ta seg av er begrensa av kor djup denne stakken er, ein vil trenge ein lagringsplass for kvart nivå [**37**].

3. Instruksjonssett

3.1 Kva er eit instruksjonssett?

Eit instruksjonssett, eller instruksjonssettarkitektur, er den delen av prosessoren som er "synleg" og tilgjengeleg for programmeraren og den som lagar kompilatoren. Dette inkluderar informasjon om : minneadressering, adresseringsmodi, type og storleik for operander, tilgjengelege operasjonar (databehandling og kontrollflyt) og kodingstype [**12**].

3.2 Minneadressering

Dette avsnittet tek for seg dei vanlegaste adresseringsmodusane i ein DSP. Adresseringsmodusane til ein prosessor er dei ulike måtane den har tilgang til data.

Registeradressering
I denne forma for adressering nyttar ein registere direkte i instruksjonen.

Direkte adressering
I denne forma for adressering oppgjev ein direkte adresser i dataminnet i instruksjonen.

Indirekte adressering
Her blir adressa til dataen ein skal operere henta frå eit register, til dømes i AGU. I instruksjonen spesifiserer ein dette registeret i staden for adressa til minneområdet.

Umiddelbar adressering
I denne forma for adressering blir ein del eller all dataen spesifisert i sjølve instruksjonen. Til dømes kan det vere ein integer.

Døme SUB 5 R2 R1, som tilsvarar: $5 - data\ i\ R2\ \rightarrow R1$

Bit-reversert adressering
Denne forma for adressering er særskild nyttig når ein skal utføre FFT. Dette er ikkje særskild aktuelt for denne applikasjonen, men kan diskuterast i forhold til auka fleksibilitet.

Moduloadressering
Dette er ei form for adressering som forenklar implementasjonen av sirkulære bufferar. Ein sirkulær buffer, også kjend som ringbuffer, er eit register der ein har "kopla saman" endane. Det vil i praksis seie at når ein når eit vist punkt eller adresse i registeret vil ein automatisk byrje på nytt i den andre enden. Dette er også nyttig når ein skal utføre konvulsjon, noko som er ein vanleg operasjon for DSP-ar.

3.3 Operasjonar i instruksjonssettet

Tabell 4 gjev eit overblikk over instruksjonar som typisk finnes i alle prosessorar [**12**]:

TYPE	DØME
Aritmetiske og logiske operasjonar	Adder, subtraher, 'and', 'or', multipliser, del og samanlikning
Dataoverføring	Hent og lagre (load-store), flytt
Kontroll	"Branch", "Jump", prosedyre-kall, returner

Tabell 4

I tillegg vil dei fleste instruksjonssett innehalde ein del instruksjonar som er særskilde for den applikasjonen dei skal nyttast til. Det kan til dømes vere operasjonar for flytande-punkt-aritmetikk, for styring av operativsystem, for behandling av "stringar" eller grafikk [**12**].

Med tanke på å forenkle koding av instruksjonssettet, beskrive i avsnitt 3.4, bør instruksjonane grupperast etter dei eigenskapane dei har felles. Til dømes vill "ADD" og "SUB" vere naturleg å sette i samme gruppe fordi dei støttar dei samme operandane. Vidare kan ein gruppere "NEG" og "NOT" fordi desse berre nyttar ein operand [**38**]. Grunnen til at ein gjer dette er at det forenklar implementasjonen av dekoderen, dette blir utdjupa i avsnitt 3.4.

3.3.1 Aritmetiske og logiske operasjonar

Tabellen under viser ein del typiske aritmetiske instruksjonar:

ARITMETISKE OPERASJONAR	
Addisjon	ADD
Subtraksjon	SUB
Multiplikasjon	MUL
Divisjon	DIV
LOGISKE OPERASJONAR	
AND	AND
OR	OR
NOT	NOT
XOR	XOR
SKIFTOPERASJONAR	
Aritmetisk skift høgre	ASR
Aritmetisk skift venstre	ALS
Logisk skift høgre	LSR
Logisk skift venstre	LSL

Tabell 5

Det er verdt å merke seg at multiplikasjon og divisjon kan implementerast med dei andre instruksjonane og difor ikkje treng å vere med i eit minimalt instruksjonssett, men det er vanleg å inkludere multiplikasjon.

I tillegg kan det vere aktuelt med mellom anna større enn/mindre enn –instruksjonar.

3.3.2 Kontrollflytoperasjonar

Kontrollflytoperasjonane består av:

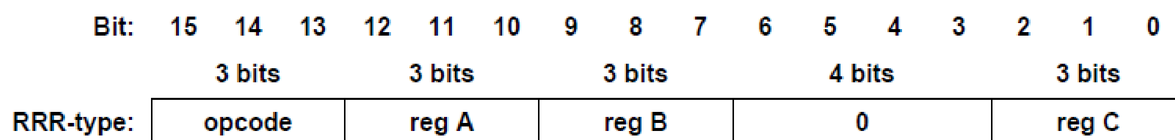
1. Forgreiningar
2. Hopp
3. Prosedyrekall
4. Prosedyre-retur

Forgreiningar og hopp har er beskrive kort i samband med beskriving av kontrolleininga i avsnitt 2.4.2. Desse to operasjonane fører begge til at ein byrjar å lese instruksjonar frå ein annan del av minnet. Det som skil dei er at forgreiningar er betinga at ei på førehand bestemt hending inntreff, som til dømes at ein teller når slutten, medan hopp skjer ubetinga andre hendingar [**12**].

Prosedyrekall og returar frå desse fører også til at ein byrjar å lese instruksjonar frå ein annan del av minnet. Det som skil den frå dei andre operasjonane er at den involverer ei overføring av kontroll til resursane til prosessoren og ei eller anna form lagring av tilstanden til prosessoren i det kallet blir gjort. Dette er nyttig fordi det betyr at prosessoren kan avbryte ei arbeidsoppgåve, byrje på ei ny, fullføre denne og så gå tilbake til den opprinnelege arbeidsoppgåva. I praksis betyr dette at ein lagrar verdiane frå dei registera som skal nyttast til andre register, til dømes i ein stakk [**12**].

3.4 Koding av instruksjonar

Kva operasjonar som skal utførast og kva minneadresser desse operasjonane nyttar blir formidla til prosessoren gjennom instruksjonar koda som binære sekvensar. Desse er delt opp i ulike felt på eit visst antal bit, det finnes typisk eit felt som viser kva slags struktur resten av instruksjonen har, eit felt for operasjonskoden og eit eller fleire felt for adresser. I tillegg kan ein nytte eit eige felt som fortel kva adresseringsmodus som skal nyttast. Dette kan vere nyttig om prosessoren støttar mange adresseringsmodi, men i tilfeller der den støttar relativt få kan desse kodast som ein del av operasjonskoden [**12**].



Figur 10: Døme på instruksjon [40]

Figuren over viser eit døme på ein enkel instruksjon beståande av ein operasjonskode på tre bit, tre registeradresser på tre bit samt eit felt på fire bit, i dette tilfellet satt til verdien null for å vise at det ikkje er i bruk. Om operasjonskoden til dømes er "add" vil denne instruksjonen føre til at prosessoren legger saman innhaldet i register B og register C og lagre resultatet i register A.

Lengda på ein instruksjon kan anten vere variabel, det vil seie ulik for ulike instruksjonar, eller fast. Instruksjonssett der ein nyttar variabel lengde gjev mindre kodestorleik, men dei er meir kompliserte å dekode, fordi ein først må dekode lengda til instruksjonen [**38**]. Fast instruksjonslengde brukas gjerne i tilfelle der ein har få adresseringsmodi. Lengda på instruksjonane er vanlegvis i storleiksorden på byte, dvs. 8 bit. Det går også an å velge ei

blanding av desse, ein sokalla hybrid. Dette går ut på at ein har eit visst antal variable lengder [**12**].

Naturlegvis må ein operasjonskode bestå av nok bit til at alle operasjonar har sin eigen unike kode. Sidan den er koda binært må difor ein operasjonskode vere minst $\log_2(m)$ lang, der "m" er antal operasjonar prosessoren skal kunne utføre. Utover at kvar operasjon skal ha sin eigen unike kode er det nokre andre hensyn ein bør ta når ein vel operasjonskodar:

1. Operasjonskoden kan ikkje vere kortare enn det minste objektet prosessoren kan hente ut av minnet, til dømes eit byte.
2. For å dekode ein operasjonskode på til dømes 8 bit trengs det ein 8-linjers til 256-linjers dekoder. Ved å velge smarte operasjonskodar og utnytte mønster og likheiter i dei ulike kodane kan denne delast opp i mindre einingar.

Her kan til dømes dei to første bit-a bety at operasjonen tilhøyrer ei gruppe som har to operandar. Denne inneheld ei rekke operasjonar, som "ADD" , "MUL" og "SUB". Dei tre neste bit-a avgjer kva operasjon det er snakk om, til dømes "MUL". Dei tre siste bit-a kan avgjere kva type operandar som skal nyttast, til dømes eit register og ein konstant.

4. Bruksområdet til prosessoren

For at prosessoren skal kunne oppnå låg effekt må ein ofre noko av fleksibiliteten. Sjølv om det er ønskeleg for ein SDR-prosessor å kunne køyre "alle" radioprotokollar er dette ikkje foreinleg med lågt effektforbruk fordi det impliserer noko der store delar av prosessoren er redundant. Med andre ord vil ein prosessor med absolutt fleksibilitet innehalde funksjonalitet som sjeldan eller aldri blir tatt i bruk. Dette kan vere akseptabelt, anten om ein faktisk har behov for å benytte seg av veldig mange ulike protokollar eller om prosessoren skal nyttast i eit høve der effektforbruk ikkje er kritisk. Dette er ikkje tilfellet med den applikasjonen som denne arkitekturen skal nyttast til, difor er det naudsynt å avgrense bruksområdet til den aktuelle arkitekturen.

Dette kapitellet er todelt, avsnitt 4.1 og 4.2 er ein presentasjon av bruksområdet til prosessoren, resten av kapitellet består av ei analyse av kva krav denne typen applikasjon vil stille til ein digital signalprossesor.

Denne arkitekturen skal vere tilpassa trådlause einingar med relativt kort rekkevidde og relativt låge overføringshastigheiter. Dømer på slike protokollar er Bluetooth og ANT.

4.1 Bluetooth

Bluetooth (IEEE 802.15.1) er eit radiosystem med kort rekkevidde, opp til hundre meter i industrielle applikasjonar og typisk 1-10 meter for brukarapplikasjonar. Bluetooth nyttar eit frekvensband rundt 2.4 GHz, kjend som ISM-bandet [**39**]. Den er meint som ei erstatning for korte kablar, særskild i mobile brukarapplikasjonar. Det kjenneteiknast ved låg kompleksitet, lågt effektforbruk og låg kostnad [**40**]. Den normale overføringsraten til Bluetooth ("basic rate" - BR) er opp til 1 Mb/s og nyttar GFSK-modulasjon. Nyare versjonar av Bluetooth (v2.0) oppnår rater på 2Mb/s og 3Mb/s med henhaldsvis π/4-DQPSK- og 8DPSK-modulasjon, desse er også kjende som "enhanced data rate" eller EDR [**41**].

Mange andre teknologiar nyttar ISM-bandet fordi ein ikkje treng lisens for å nytte det. For å unngå å påverke andre system på same band hoppar Bluetooth-einingar mellom ulike sende-frekvensar, dei hoppar mellom 79 ulike frekvensar 1600 gongar i sekundet [**42**].

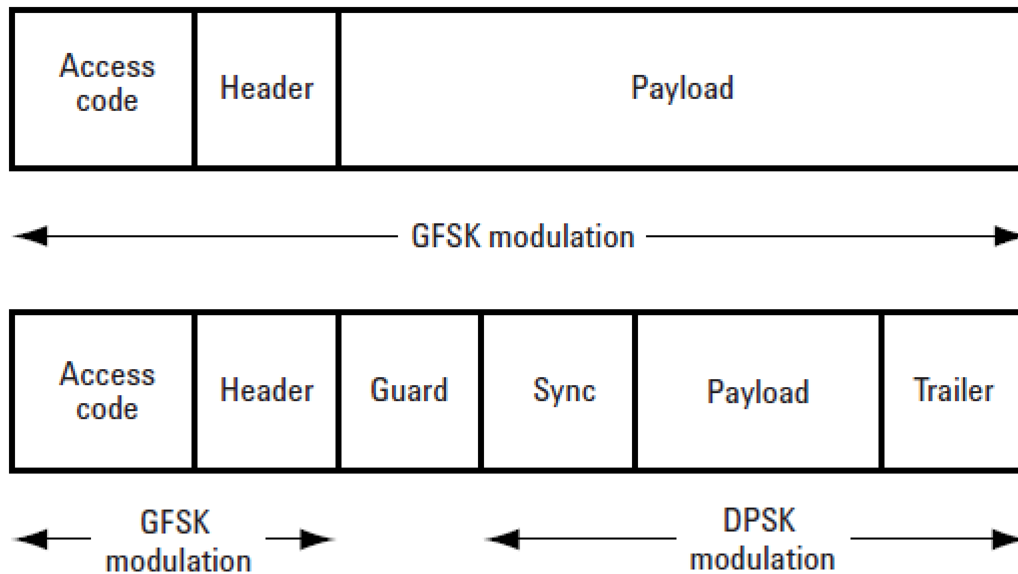
Når to einingar skal kommunisere opprettar dei eit lite lokalt nett, eit "piconett". Ein basestasjon sender ut eit kall til ei rekke "adresser" som ligger innanfor eit visst område som på førehand er satt av til den typen einingar som basestasjonen vil nå. Produsenten av eininga har på førehand koda inn ei adresse i dette området, og når denne får førespurnad av basestasjonen svarar dei og opprettar dermed eit nett. Dette nettet kan bestå av opp til åtte einingar [**42**] [**40**].

Bluetooth sender informasjon i form av pakkar. Desse pakkane inneheld den dataen som skal sendast, men den inneheld også mellom anna ein sokalla aksesskode ("access code") og ein "header." Pakkane i BR består berre av desse to i tillegg til dataen ("payload"), medan pakkar i EDR har ein del fleire bestanddelar. Desse består av "guard" og "sync" før dataen og "trailer" etter. Figur 11 viser to slike pakkar, den øvste er ein BR-pakke og den nedste er ein EDR-pakke.

Som ein kan sjå frå figuren nyttast GFSK-modulasjon for aksesskoden og "header"-en i begge dataratene. Aksesskoden består av ein "preamble", synkroniseringsord og "trailer" på til saman 72 bit. Den nyttast til taktgjenvinning, identifikasjon og kompensasjon av "DC-

offset." "Preamble"-et nyttast til taktgjenvinning, synkroniseringsordet består av ein unik kode for sendaren som gjer at ein berre kommuniserer med dei einingane et var meininga å nå.

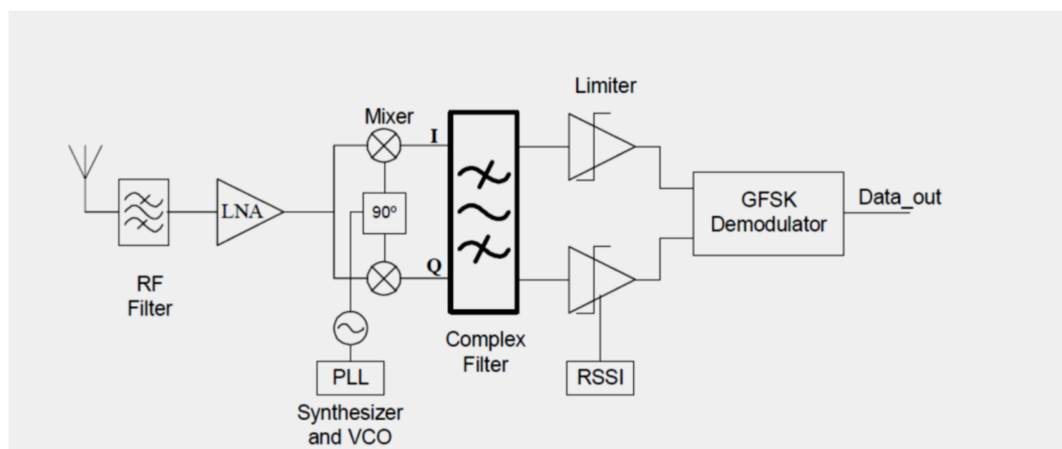
"Header"-en er på 54 bit og består av viktig kontrollinformasjon, mellom anna kan den informere mottakaren om det er ønska automatisk svar og feilkorrigerande kode. Dataen kan variere i lengde mellom 0 og 2745 bit [**43**].



Figur 11: Bluetooth-pakkar [41]

Låg-IF-mottakar eller mottakar med direkte konvertering :

Dei to vanlegaste metodane for å implementere ein mottakar i programvareradio er Låg-IF ("low intermediate frequency") og direktekonvertering.



Figur 12: Låg-IF Bluetoothmottakar (basic rate) [44]

Låg-IF går ut på at ein konverterar ned frekvensen til inngongssignalet ned til ein midlertidig frekvens som er lågare enn sendefrekvensen før. Dette er i motsetnad til ein direktekonvertering der ein miksar inngangssignalet med ein frekvens som er lik bærefrekvensen utan å konvertere til ein lågare frekvens først.

Ved å konvertere direkte kan ein forenkle mottakaren i forhold til ein låg-IF-mottakar der ein har fleire konverteringssteg for å få ned frekvensen til signalet. I tillegg til dette unngår ein problematikk tilknytta billdeavvising ("image rejection"), det vil seie at andre enn den frekvensen ein vil demodulere produserer den same IF som det ønska signalet. Denne typen mottakar har også ein del problem, det største av desse er at det kan lekke energi frå miksaren til antenna slik at denne kjem tilbake til miksaren og dermed skaper eit potensielt kraftig "DC-offset." Dette kan løysast med andre komponentar, men dette aukar kompleksiteten [**45**].

Å velje mellom låg-IF og direktekonvertering er ei vanskeleg problemstilling, men det ligger noko utanfor omfanget til denne oppgåva. Begge deler blir brukt av kommersielle aktørar. Formålet med oppgåva er å presentere eit forslag til ein lågeffekt DSP til bruk i digital kommunikasjon, fleksibilitet er eit viktig aspekt ved eit slikt design og det er ikkje usannsynleg at det er mogleg å implementere ulike mottakararkitekturar ved å bruke den som ei digital signalbehandlingskjerne.

RSSI

"Recieved Signal Strength Indicator" er ei måling som gjev ein indikasjon på kor stor effekten i eit mottatt radiosignal er.

Begrensar

Ein begrensar, eller "limiter", er som namnet antydar ein komponent som begrensar amplituden til dei delane av eit signal som er over ein gjeve effektverdi. Samtidig som den gjer dette let den dei delane av signalet som har lågare amplitude enn dette nivået passere uforandra. I utgangspunktet blir ikkje informasjonen i frekvensmodulerte signaler påverka av endringar i amplitude fordi all informasjonen er representert i berre frekvensbandet, men stor variasjon og for høge verdiar kan føre til støy og uønskte effektar i demodulatoren [**46**].

For denne applikasjonen kan bruken av ein begrensar gje ein stor fordel. Ved å bruke denne kan ein anta at amplituden til det signalet som kjem inn i demodulatoren har ein konstant amplitude. Dermed kan ein unngå den høge kostnaden som er knytt til å implementere ei divisjonseining i eininga som approksimerar den deriverte av arcus tangens. Formelen for denne utrekninga (Formel 10) viser at ein deler på amplituden av signalet (I(t)^2 + Q(t)^2). Kva følger det får å introdusere ein limiter blir diskutert saman med andre alternative implementasjonar av mottakaren i kapittel 5.

Taktgjenvinning

Kvar gong ein skal sende ein pakke i Bluetooth er det først naudsynt å synkronisere mottakar med den innkomande pakken. Dette er kalla taktgjenvinning. I byrjinga av ein pakke blir det sendt ein serie av null og einarar. Dette såkalla "pre-amble"-et kan nyttast for å synkronisere mottakar med pakken [**47**].

4.2 Algoritmar som nyttast i desse protokollane

4.2.1 GFSK

"Gaussian Frequency Shift Keying" er ein metode for frekvensmodulasjon der ein nyttar eit gaussisk filter for å forbetre sendeeigenskapane til systemet. Frekvensmodulasjon går igjen ut på at ein gjer om dei ulike symbola ein skal sende til ulike frekvensar. Om ein har to ulike symbol nyttar ein to ulike frekvensar for å representere dei, som til dømes:

$$x_o(t) = Acos(2\pi f_0 t + \theta_0) \qquad kT_s \leq t < (k+1)T_S \quad er \;\; binær\; 0$$ Formel 8

$$x_o(t) = Acos(2\pi f_1 t + \theta_1) \qquad kT_s \leq t < (k+1)T_S \quad er \;\; binær\; 1$$ Formel 9

Der T_S = symbolperioden, θ_n = fasen , A = amplituden til signalet og f_n =frekvensane dei ulike symbola blir satt til [**4**].

Det finnes mange ulike måtar å implementere GFSK på, dei har ulike eigenskapar. I prosjektoppgåva [**4**] knytt til denne masteroppgåva blir nokre av desse presentert, og ein metode blir vald ut og analysert med tanke på bruk i ein DSP. Metoden som er vald er ein sokalla "arctan-diferrentiert digital demodulator", som er ein FM-diskriminator etterfølgd av ein "integrer-og-dump". Denne metoden nyttar det at ei endring i frekvens også gjev utslag i ei endring i fase. For å demodulere GFSK kan ein derfor nytte seg av den deriverte av invers tangens.

Dette kan skrivast som:

$$\frac{d}{dt}\arctan\left(\frac{Q(t)}{I(t)}\right) = \frac{I(t)\cdot Q'(t) - Q(t)I'(t)}{I^2(t)+Q^2(t)}$$ Formel 10

Dette er den "konvensjonelle" typen GFSK-demodulator og tilsvarar ei utrekning av fasen ved hjelp av kvadraturdemodulasjon og utrekning av arcus tangens etterfølgd av ein "diskriminator", men det finnes ein del andre alternativer. Nokre av dei er meir egna for låg-effekt-implementasjon enn andre, mellom anna: korrelasjonsdemodulator, digital kryss-differentsiert multiplikasjons-demodulator (DCDM), ST-DFT, som er basert på ei kort-tids diskrete Fouriertransformalgoritme og nullkrysningsdetektor [**48**].

Arctan-demodulatoren er vald som døme på grunn av at den er så vanleg i bruk, noko som forenklar analyse av den, og fordi den mellom anna opnar for at ein kan nytte den sokalla CORDIC-algoritma, som er presentert tidlegare i avsnitt 2.5.1. Det ligger noko utanfor omfanget til denne oppgåva å diskutere kva som er den mest velegna demodulatoren, men det kan vere interessant å ha litt innsikt i dette.

Korrelasjonsdemodulator er ein "optimal demodulator" for GFSK og har høgast yting av dei føreslåtte demodulatorane, med dette meinar ein at den gjev minst bit-feil ved ein gitt SNR, ulempa med den er at den har høg kompleksitet på grunn av multiplikatorar og den blir difor sjeldan brukt i FSK-implementasjonar [**48**].

Ei løysing som er robust samtidig som den gjev lågt effektforbruk er kort-tids DFT saman med ein null-IF-arkitektur [**49**]. Den kan utan problem avvise statisk "offset" med eit minimum av ekstra maskinvare og er i tillegg lite følsom for låg SNR. Sjå Lopelli et. al. [**48**] for ein nærare presentasjon av desse alternative demodulatorane.

4.2.2 DPSK

DPSK ("Differential Phase Shift Keying") er ei form for modulasjon der ein endrar fasen til eit signal for å overføre symboler, amplituden og frekvensen til signalet held seg konstant [**50**].

For å sjå korleis DPSK er modulert må ein sjå på signalet over to bitperiodar fordi det er endring mellom desse som representerar sendte symboler. Om ein til dømes definerar ein binær 1 som ingen endring i fase og ein binær 0 som 180 graders endring i fase vil det sendte signalet sjå slik ut [**51**]:

$$s(t) = \begin{cases} \sqrt{\frac{E_b}{T_b}}\cos(2\pi f_c t)\,, & 0 \leq t \leq T_b \\ \sqrt{\frac{E_b}{T_b}}\cos(2\pi f_c t\ +\ \pi), & T_b \leq t \leq 2T_b \end{cases}$$

Formel 11

Formelen over viser korleis ein binær 0 kan representerast med eit faseskift på π. Om dette er differensiell koding vil det vere umogleg å seie kva s(t) representerer i perioden 0- T_b . F_c er bærefrekvensen, T_b er ein bitperiode.

Sidan det i denne applikasjonens tilfelle er snakk om differensiell demodulasjon finn ein den demodulerte signalet ved å sjå kor mykje fasen i ein gitt bitperiode skil seg frå fasen til signalet i den førre bitperioden. Ein måte å finne denne fasen på er å nytte ein kvadraturdelmodulator og rekne ut arcus tangens av I/Q:

$$\theta_n = \tan^{-1}\frac{Q_n}{I_n}$$

Formel 12

Bluetooth EDR nyttar to former for DPSK til demodulasjon, π/4 –DQPSK for ei datarate (2Mbps) og 8-DPSK for ei datarate (3Mbps) [**41**]. Desse nyttar faseskift med ulik storleik for å representere verdiar. Den førstnemde kan representer fire ulike symboler (00 01 10 11) ved hjelp av fire ulike faseskift som er jamnt fordelt utover enheitssirkelen, 8-DPSK kan tilsvarande måte representere åtte symboler.

4.3 Analyse og profilering

I prosjektoppgåva [**4**] som er knytt til denne masteroppgåva blir ein enkel pseudokode for GFSK- og DPSK-demodulasjon presentert og analysert med tanke på kva slags effektforbruk den ville ha om den køyrde på ein gitt kommersiell tilgjengeleg generell DSP. Ved hjelp av denne koden kan ein identifisere finne ei rekke viktige opplysningar som er viktig for å kunne lage eit effektivt instruksjonssett og ein god arkitektur:

- Identifisere nødvendige operasjonar
- Identifisere kjerner
- Identifisere moglegheiter for parallellitet

Det er særskild viktig å finne kjerner i koden, når ein har funne desse kan ein vurdere om dei skal implementerast som akseleratorar.

Ideelt sett burde ein ha kode, i til dømes C, som er testa og klar for å køyre for å kunne skape eit heilt presist bilete av korleis applikasjonen oppfører seg. Å skrive og teste slik kode ligger noko utanfor omfanget til oppgåva og det er derfor ikkje brukt noko tid på dette. Sidan hensikta er å designe ein prosessor og ikkje ein applikasjon kan ein anta at pseudokode vil tene formålet godt nok til å kunne profilere krava til koden. Pseudokode for dei ulike delane av mottakaren er inkludert som vedlegg 1.

For både GFSK og DPSK demodulasjon vil den første delen av pseudokoden bestå av miksing med ein lokal oscillator og filtrering av eit lågpassfilter. Det er eit opent spørsmål om desse skal implementerast i maskinvare. Intuitivt er det usannsynlig at miksing skal gjerast i det digitale domenet, dette blir diskutert i avsnitt 5.1

4.3.1 Analyse av GFSK

Sjå Vedlegg 1.1 : Pseudokode for GFSK for koden som blir nytta til denne analysa.

Ved å sjå på koden kan ein identifisere kjerna til GFSK-demodulatoren, det er ei utrekning av den deriverte av invers tangens av I(t)/Q(t) [**4**]. Den deriverte av arctan(I/Q) er gitt i (Formel 10).

Den deriverte av I(t) og Q(t) kan ein approksimere ved å ta differansen mellom to etterfølgande samples. Gitt at ein har I(t) og Q(t) tilgjengeleg samt den deriverte av desse vil dette medføre at ein til dømes kan utføre denne funksjonen som:

1. Multiplikasjon: $a = I(t) \cdot Q'(t)$
2. Multiplikasjon: $b = I'(t) \cdot Q(t)$
3. Subtraksjon: $c = a - b$
4. Multiplikasjon: $a = I(t)^2$
5. Multiplikasjon: $b = Q(t)^2$
6. Addisjon: $d = a + b$
7. Divisjon: $utgang = \frac{c}{d}$

Om dette skal gjennomførast i prosessoren utan ein eigen aksellerator eller moglegheit for paralellitet kjem det til å ta 7 steg og alle desse stega består av fleire operasjonar som kvar treng fleire klokkesyklusar for å bli ferdig. Multiplikasjon og lagring kan potensielt kreve ein klokkesyklus for å laste inn verdiane, ein for å multiplisere og ein for å lagre verdiane. Om ein ikkje har ein har ein divisjonsoperasjon tilgjengeleg må dette gjerast gjennom ei subrutine og kan ta ange fleire klokkesyklusar.

Sidan dette er ei uttrekning av den deriverte av invers tangens er det også mogleg å implementere den ved hjelp av CORDIC eller andre former for approksimasjon av arcus tangens, desse blir presentert i neste avsnitt i samband med analyse av DPSK. Ein kan då til approksimere den deriverte av arcus tangens ved å subtrahere to etterfølgande resultat frå uttrekning av arcus tangens.

4.3.2 Analyse av DPSK

Sjå Vedlegg 1.2 : Pseudokode for DPSK for koden dette avsnittet tek for seg.

Ved å sjå på denne koden finn ein fort at hovudoppgåva og kjerna til ein DPSK-demodulator er ei kompleks utrekning av fasen til inngangssignalet over ein bit-periode [**4**], og samanlikning av denne verdien med fasen til den førre bit-perioden [**50**]. Kjerna i denne forma for demodulasjon er utrekninga av invers tangens av I/Q [**4**]. Invers tangens, eller arcus tangens, av komplekse tal (x+ jy) nyttast i mange former for digital signalbehandling [**50**]. Dette er ei krevande oppgåve, og for å gjere den effektiv gjer ein ein approksimasjon av resultatet [**52**].

Invers tangens gjennomføres her på I(t)/Q(t) og vi må derfor ha med en divisjon. Denne divisjonen kan anten utføres som ei subrutine eller implementerast som ei eiga eining. Ein måte å implementere dette på er:

1. Divisjon: $a = \frac{I}{Q}$
2. Arctan: $b = \arctan(a)$
3. Subtraksjon: c = b-d
4. Samanlikning: (c>1) -> utgang = 1 (c<1) -> utgang = 0
5. Lagring: d = b

Punkt 2 viser til ein approksimasjon av arcus tangens. Dette er i seg sjølv ei krevande oppgåve som kan utførast på ei rekke ulike måtar. Dette temaet er diskutert vidare i avsnitt 5.2.

4.4 Minimumskrav til nøyaktigheit i approksimasjon av fase

For å kunne samanlikne dei ulike implementasjonane av approksimasjon av arcus tangens, som er presentert i neste kapittel, er det først viktig å fastslå omlag kor stor nøyaktigheit ein treng i denne approksimasjonen. Dette er eit komplisert spørsmål, og det ligg utanfor omfanget av denne oppgåva å diskutere dette fullt ut, sjølv om det er ein nyttig diskusjon med tanke på val av akseleratorar.

Nøyaktigheita må vere stor nok til at ein kan avgjere om den målte fasen ligg innanfor desisjonsgrensene for dei ulike symbola som er motteke. Dette betyr i utgangspunktet at ein treng ei nøyaktigheit som tilsvarar antal bit i symbolet. For Bluetooth vil det seie:

Modulasjon	Bits nøyaktigheit
GFSK	1
π/4-DQPSK	2
8-DPSK	3

Tabell 6

Eit anna spørsmål her er om systemet berre skal støtte hard desisjon eller også opne for mjuk desisjon. Harde desisjonsgrenser vil seie at ein berre nyttar akkurat det antalet bit ein treng, "soft"-desisjon vil seie at ein nyttar fleire bit for å gjere systemet meir robust mot støy [**53**]. Det kan då vere snakk om å nytte 2-4 fleire bit for å oppnå dette.

5. Implementasjon av akselleratorar

5.1 Plassering av ADC

Ei avgjersle av kva komponentar av det fysiske laget som skal implementerast som maskinvare og kva som skal implementerast som programvare er kritisk til kva krav som stilles til ein prosessor. Den ideelle SDR er i introduksjonen beskrive som ein DSP som er kopla rett på antenna. Dette er ikkje særskild realistisk, fordi det vil sette så høge krav til prosessoren at det vil føre til eit for høgt effektforbruk. Ein modell for RF-delen er presentert i antakingane, men det kan vere verdt ein kort diskusjon å sjå på kva konsekvensar det vil ha å "flytte" ADC-en nærare antenna.

Om ein inkluderer lågpassfiltera og miksaren i den digitale delen av prosessoren vil dette antyde at ein treng ein mykje større grad av parallellitet, fordi spesielt miksaren vil dominere ytingskrava overfor demodulatoren, spesielt med tanke på kor mange parallelle multiplikatorar ein treng. Dette er vist i prosjektoppgåva [**4**] knytt til denne oppgåva. Dette vil truleg auke kompleksiteten, arealet og effektforbruket til prosessoren betrakteleg, fordi ein kan anta at det er meir energieffektivt å implementere desse analogt.

5.1.1 Begrensar

Begrensar er introdusert i avsnitt 4.1, som nemnt der vil ein med ein begrensar kunne redusere kompleksiteten og effektforbruket til mellom anna GFSK-demodulatoren, fordi den fører til at ein kan anta konstant amplitude på inngongssignalet. Spørsmålet er om denne reduksjonen i kompleksitet og effektforbruk veg opp for kostnaden ved å introdusere eininga i systemet.

Å introdusere ein begrensar til systemet vil ha ein del påverking på dei ulike løysingane som er diskutert i dette kapittelet, og dette er belyst når det er tilfellet. Ein begrensar er vil uansett ligge utanfor den foreslåtte DSP-en, så det vil ikkje bli gjeve ein detaljert beskrivelse av implementeasjon av denne. Ein begrensar vil ligge mellom RF-"front end"-en og DSP-en og påverkar også korleis ein implementerar ADC-ane.

5.2 Approksimasjon av arcus tangens

Som vist tidlegare er approksimasjon av invers tangens kritisk for dei to presenterte demodulasjonsmetodane. Å rekne ut invers tangens er ei krevjande oppgåve og ein kan nytte mange ulike approksimasjonar som har ulike eigenskapar, til dømes: CORDIC som er beskrive tidlegare i avsnitt 2.5.1, og oppslagstabellar("look-up table") [**52**] og høg-ordens algebraiske polynom, samt nokre andre meir spesialiserte metodar, desse blir gjennomgått i dette avsnittet og samanlikna med CORDIC.

For å finne arcus tangens ved hjelp av **CORDIC** nyttar ein vektormodus og initialiserar akkumulatoren med verdien null. Dette vil direkte gje vinkelen mellom x og y. Sidan ein berre treng verdien frå akkumulatoren kan ein oversjå forsterkinga som CORDIC gjev [**32**]. Ein stor fordel ein får ved å bruke CORDIC er at den gjev fleksibilitet i forhold til alternativa. Det er som tidligare sagt mogleg å bruke den til utrekning av ei lang rekke funksjonar, som er presentert i Tabell 3.

Metoden med ein rein **oppslagstabell** er i prinsippet intuitiv og enkel å implementere, I og Q gjev adresser i eit minneområde og desse inneheld approksimasjonar av fasen som er rekna ut på førehand. [**52**].

I praksis finnes det ein del variasjonar når det gjeld å implementere oppslagstabellen. Det finnes standard oppslag, oppslag med interpolasjon og invers oppslag med interpolasjon. Det første ein kan merke seg, er at ein berre treng å ha oppslagsverdiar for ein åttandedel av "sirkelen," eller ein oktant. Dette er fordi ein kan flytte ein vektor inn i første kvadrant anten ved å rotere den med 90 eller 180 grader. Vidare kan ein enkelt avgjere i kva region av denne kvadranten denne vektoren befinner seg og dermed kan ein begrense oppslagstabellen til ein åttandedel av "sirkelen". Forteiknet til x og y beskriver kva kvadrant den opprinnelege vektoren befinn seg i, som vist i Figur 7 [**53**] [**54**].

Tabellen må dermed innehalde verdiar for:

$$0 \leq \frac{y}{x} \leq 1\,, 0 \leq \theta \leq \frac{\pi}{4}$$

For å forbetre nøyaktigheita til tabellen kan ein gjere ein interpolasjon mellom to verdiar som ligger nære den ein vil ha [**54**].

Høg-ordens algebraiske polynomer er eit alternativ, i form av anten Kjebysjev-polynomer eller Taylor-rekker, men arcus tangens er svært ulineær og derfor vanskelig å approksimere med eit fornuftig antal ledd. [**52**].

Taylor-rekka til invers tangens er [**55**]:

$$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \ldots\ldots, \quad der\ |x| \leq 1$$

Formel 13

Som vist held dette berre for |x|≤ 1, men dette vil alltid halde for første kvadrant, og som beskrive tidlegare er det mogleg å flytte vektoren dit.

Utover det finnes det nokre meir uvanlege approksimasjonar, som den presentert av **Lyons** [**52**], som verken nyttar oppslagstabell eller høg-ordens polynomer. Den er særskild nyttig dersom ein allereie har rekna ut Q^2 og I^2 i forbindelse med andre operasjonar:

$$\tan^{-1}(Q/I) \approx \theta' = \frac{Q/I}{1 + 0.28125(Q/I)^2} \text{ rad}$$

Formel 14

Denne kan nyttast i området:

Formel 15

$$-1 < \frac{Q}{I} < 1, eller -\frac{\pi}{4} < \theta < \frac{\pi}{4}$$

Denne kan også skrivast som:

$$\theta' = \frac{IQ}{I^2 + 0.28125Q^2}$$ Formel 16

Dermed eliminerar ein to av divisjonane og erstattar dei med multiplikasjonar, noko som kan tenkast å vere nyttig om ein har fleire multiplikatorar tilgjengeleg. 0,28125Q^2 kan og skrivast som (1/4 + 1/32)Q^2 og dermed kan ein implementere denne delen som Q^2 skifta med to bit pluss Q^2 skifta med fem bit. Med denne metoden oppnår ein ein feil på maksimum 0,0049 radianar (0,28°) [**52**].

Rajan et.al. [**56**] foreslår ein approksimasjonsmetode der dei nyttar andre- og tredje-ordens polynomer og enkle rasjonelle funksjoner. Dei nyttar Lagrange-interpolasjon og "minimax"-kriterier for å finne dei optimale koeffisientane. Sjølve metodikken fell utanfor denne oppgåva, men resultatet er ei rekke approksimasjonar med ulik yting og kompleksitet. Mellom anna kjem dei fram til omlag den same approksimasjonen som liknar Lyons, men nyttar to multiplikasjonar, ein divisjon og ein addisjon, i staden for to addisjonar, ein divisjon og ein multiplikasjon.

Sidan denne metoden liknar på den presentert over, og er litt meir komplisert å implementere [**56**], er det kanskje mest interessant å sjå på dei approksimasjonane som ikkje nyttar divisjon, det finnes fleire alternativ:

$arctan(x) \approx$	Maksimal feil	Addisjonar	Multiplikasjonar
$\frac{\pi}{4}x$	0,07 rad	0	1
$\frac{\pi}{4}x + 0,273x(1 - \|x\|)$	0,0038 rad	1	2
$\frac{\pi}{4}x - x(\|x\| - 1) \cdot (0,2447 + 0,0663\|x\|)$	0,0015 rad	2	3

Tabell 7

Kjelde for tabell: [**56**]

METODE	FORDELAR	ULEMPER
CORDIC	- **Fleksibilitet:** Den kan nyttast til ei rekke ulike funksjonar. - **Enkel:** Den nyttar berre addisjon og skift-operasjonar	- **Treg:** Den er sekvensiell av natur og vil derfor kreve eit antal steg for å oppnå eit resultat med like stor nøyaktigheit som dei andre alternativa på lik tid. Dette kan implementerast anten ved å auke klokkefrekvensen eller ved å dele opp eininga i eit samleband med fleire steg.
Oppslagstabell	- **Rask:** Dette er den raskast av dei moglege metodane	- **Arealkrevande:** For å lagre eit stort antal verdiar må ein ha store ROM-einingar i arkitekturen. Minne har generellt sett eit høgt effektforbruk. - **Ufleksibel:** Låst til ein metode
Taylor-rekke	- **Direkte:** Går rett ut frå defininsjonen av funksjonen	- **Ineffektiv:** Konvergerer for sakte for verdiar nære 1. Den må difor ha høg kompleksitet for å oppnå god nok nøyaktigheit. M.a.o. må ein ha mange ledd i rekka før ein oppnår eit godt resultat. - **Ufleksibel:** Av samme grunn som over.
“Lyons-metode”	- **Rask:** Denne er meir spesialisert enn t.d. CORDIC og har større moglegheit for parallellitet	- **Kompleks:** Divisjon er kompleks å implementere - **Ufleksibel.**
“Rajan-metodar”	- **Låg kompleksitet:** I forhold til Lyons metode slepp ein divisjon.	- **Ufleksibel.**

Tabell 8

Tabell samansett frå: [**56**] [**52**]

Desse vurderingane gjer at ein kan eliminere nokre av løysingane på eit kvalitativt grunnlag. Sidan den vurderte applikasjonen er ein lågeffekt programvaredefinert radio vil det vere naturleg å gje parameterane fleksibilitet og effektforbruk størst vekt i ei vurdering av dei ulike metodane. Taylor-rekker verkar umiddelbart uaktuelt fordi det vil føre til eit høgt effektforbruk samanlikna med løysingar som kan gje den same fleksibiliteten. Oppslagstabell vil også vere uaktuell på grunn av stort areal, noko som fører til høgt statisk effektforbruk, og låg fleksibilitet.

Dermed gjenstår CORDIC og dei ulike løysingane til Lyons [**52**] og Rajan et. al. [**56**]. CORDIC tilbyr ein veldig høg fleksibilitet, og det er eit opent spørsmål om ei dedikert CORDIC-eining bør inkluderast uansett for å gjere prosessoren meir generell. Om ein ser bort frå dette er det interessant å sjå på kva av desse metodane som nyttar det lågaste areal, klokkefrekvens og effektforbruk for å oppnå den same nøyaktigheita.

Nøyaktigheit kan her målast som den maksimale feilen i radianar eller grader, som vist for nokre utvalde blant Lyons og Rajan et. al. sine metodar i Tabell 7.

CORDIC kan gje ulik nøyaktigheit ut frå antal iterasjonar, dette kan anten vere gjort gjennom å ha ei CORDIC-eining per steg eller ei eining som går gjennom alle iterasjonane. Implementasjonen av dette er beskrive vidare i avsnitt 5.4. For å kunne samanlikne CORDIC-løysinga med dei andre alernativa må ein difor finne ut kor mange iterasjonar CORDIC treng for å oppnå den samme nøyaktigheita.

Om ein samanliknar Tabell 7 og Tabell 2 kan ein sjå når dei ulike løysingane har tilsvarande grad av nøyaktigheit. Den svært enkle approksimasjonen,

$$\boldsymbol{arctan}(\boldsymbol{I}/\boldsymbol{Q}) \approx \frac{\pi}{4}\left(\frac{I}{Q}\right) \qquad \text{Formel 17}$$

oppnår ein maksimal feil på omlag 0,07 radianar i området -1<(I/Q)<1. Denne krever ein multiplikasjon og naturlegvis også ein divisjon for å bestemme forholdet mellom I og Q. For å få eit meir nøyaktig resultat enn dette, nærmare bestemt ein maksimal feil på 0,0624 radianar, må CORDIC nytte fem iterasjonar.

5.3 Samanlikning av arealbruk

Arealet komponentane brukar gjev ein indikasjon på den statiske effektforbruket, men utan å implementere komponentane er det vanskeleg å seie noko bestemt om kor stort arealforbruk dei enkelte komponentane. Det er desverre for krevjande å byrja å implementere alle delar av prosessoren, men om ein kan seie noko om det typiske forholdet mellom ulike komponentar kan dette gje eit godt grunnlag for seie noko om arealkostnaden til ulike implementasjonar av komponentar i systemet.

Ved hjelp av arealestimat frå ein implementasjon av eit liknande system [**27**], kan ein grovt vise korleis forholdet er mellom ulike standardkomponentar. Denne oppgåva har brukt standard-cellebiblioteker for å oppnå desse estimata.

Ein kan byrje med å definere ei eining til å ha storleik 1 og så definere arealforbruket relativt til denne. Valet er arbitrært, men ein adderer er vald, spesifikt er dette ein ripple-carry adderer.

Den relative storleiken av dei andre er tatt ved å dele det estimerte arealet til dei andre einingane på det estimerte arealet til ein adderer.

Eining	Bit	Arealforbruk (relativt)
Adderer	16	1
Subtraherer	16	1
Multiplikator	16	6
Dividerar[1]	16	70
MUX 2-1	16	0,3
MUX 3-1	16	0,6
MUX 4-1	16	0,625
ROM – asynkron	1024 (1 kB)	290
RAM 2-port -asynkron	1024 (1 kB)	1800
Register	16	0,575
Hardkoda konstant	16	0,15
Skifter (1-bit)[2]	16	4,8

Tabell 9

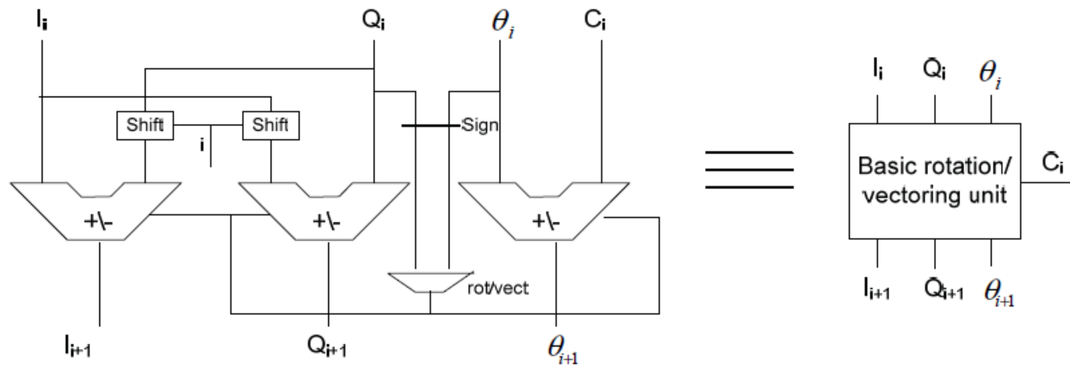
Notater for tabellen over

1. Dividereren er basert på ein 16-bits standard fast-punkts divisjonseining [**58**] som består av 16 adderarar, 16 subtraherarar, 16 samanlikninskretsar,16 skifterar samt 16x32bits register.
2. Ein skifter (barrel-skifter) består av like mange 2-1 MUX-ar som det er bit som skal skiftast. Ved skift på meir enn eit bit må ein nytte like mange muxar som det antalet det er mogleg å skifte [**57**].

5.4 Implementasjon av CORDIC

Det er mogleg å nytte CORDIC i demodulasjon av både GFSK og DPSK sidan algoritmen er ein effektiv måte å finne fasen til eit kompleks signal gjennom å approksimere arcus tangens. Difor kan det lønne seg med tanke på effektforbruk og areal å implementere denne algoritma som ein akselerator. Den store fordelen med å gjere dette vil vere at denne eininga kan nyttast til begge formene for demodulasjon, i tillegg til ei rekke andre funksjonar. Dette bidrar til å auke fleksibiliteten til systemet. Dette avsnittet tek for seg nokre ulike måtar å implementere CORDIC på. Dette vil gje eit grunnlag for å kunne samanlikne denne løysinga med andre implementasjonar av demodulasjon.

Om ein definerar ei grunnleggande eining for vektorrotasjon kan ein implementere CORDIC anten på ein rekursiv måte med ei eining eller "bretta ut" ("unfolded") med fleire einingar. Ei slik eining kan sjå slik ut:



Figur 13: Vektorrotasjonseining [27] [35]

Den rekursive metoden, som vist i er den som krever minst areal, sidan den berre nyttar ei eining. Eininga vil då iterere med dei same verdiane i ei løkke eit visst antal gongar før den er klar for nye verdiar. Om ein brettar ut denne løkka må ein ha like mange einingar som det finnes steg i løkka. Ulempa med dette er at den utbretta løkka vil bruke større areal, fordelen er at ein kan senke frekvensen og driftsspenninga og dermed sannsynlegvis oppnå lågare effektforbruk. Ein annan fordel av å velge ein løysing med ei eining er at ein i større grad får moglegheita til å velje eit fritt antal iterasjonar. Ein kan oppnå noko liknande med ei utbretta løysing også ved å gjere det mogleg å "hente ut" verdiar mellom dei ulike stega, men dette vil nok komplisere eininga ein del.

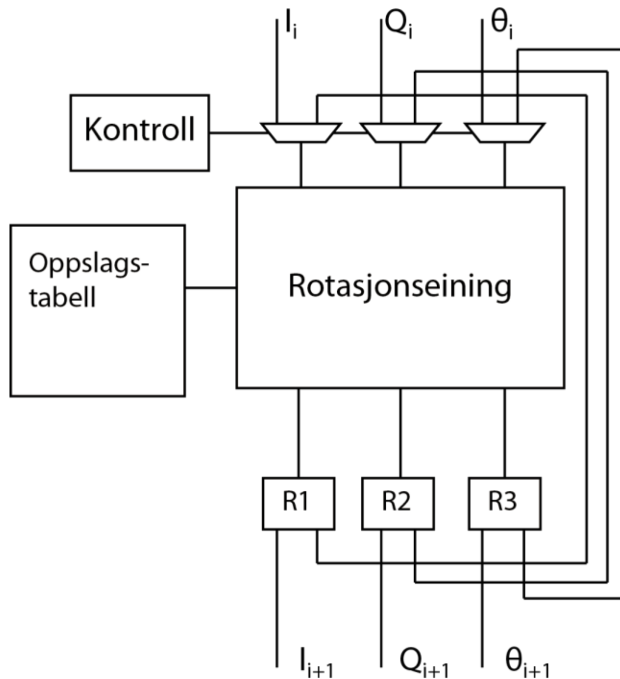
Ved hjelp av Tabell 9 kan ein no berekne det relative arealforbruket til ei slik eining, dette skal nyttast seinare i ei samanlikning mellom CORDIC og andre måtar å implemetere GFSK og DPSK. Ei slik eining består av :

Komponent	Areal	Antal	Totalt areal
Add/Sub 16-bit	2	3	6
Skiftere	4,8	2	9,6
MUX 2-1 (1-bit) "Forteikns-MUX"	0,03	1	0,03
Hardkoda konstant 16-bit	0,15	1	0,15
TOTALT	-	-	15,78

Tabell 10

I tillegg til dette må ein ha register for å mellomlagre resultat etter kvar iterasjon. Dette artar seg litt ulikt utifrå om korleis ein vel å implementere CORDIC. Om ein vel å iterere med ei eining vil ein trenge færre register for mellomlagring, berre eit for kvar av verdiane, men ein er i gjengjeld nøydd til å ha eit minne i form av ein oppslagstabell som inneheld alle koeffisientane som brukast for å summere den traverserte vinkelen. Om ein rullar det ut vil ein trenge ei slik eining for kvart steg, ein slepp oppslagstabellen, men ein må ha eit register for kvart steg for å mellomlagre resultat. Dette blir diskutert nedanfor etter at dei forskjellige løysingane er presentert.

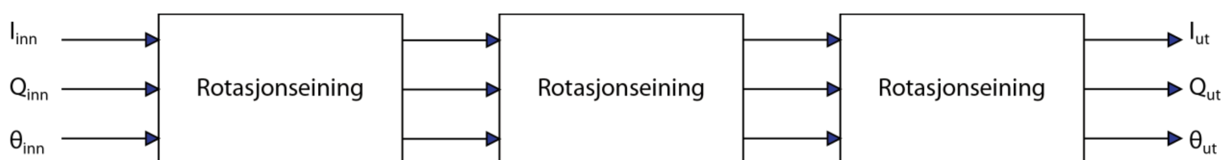
Ein variant der ein itererar med same eining kan implementerast slik [**35**] [**27**]:



Figur 14: Seriell CORDIC

Figuren ovenfor viser korleis ein kan nytte ei av dei tidlegare beskrivne vektorrotasjonseiningane til å implementere ei rekursiv eining. Den tar inn verdiar og gjer eit antal iterasjonar, desse er kontrollert av "kontroll"-eininga. Når ein har gjennomført eit ønskeleg antal er posisjonen til vektoren (I, Q) samt den vinkelen ein har traversert (θ) tilgjengeleg på tre register merka R1, R2 og R3. Kontrolleren styrer om ein skal muxe inn nye verdiar eller fortsette å behandle dei som ligger i registera. Ein oppslagstabell nyttast for å hente fram verdien for dei vinklane som skiftast. Det antalet verdiar som finnes i denne oppslagstabellen vil begrense antal moglege iterasjonar [**35**]. Fordelen med denne er at den brukar lite areal samanlikna med ei utbretta løysing.

Det andre alternativet er å implementere den med ei vektorrotasjonseining per iterasjon.



Figur 15: Parallell CORDIC

Figuren over viser ei løysing med tre einingar som utfører ein operasjon tilsvarande det tre iterasjonar i den serielle eininga i systemet. Fordelen med å gjere dette er at ein kan implementere den som eit samleband. Det betyr at ein kan ha eit høgare antal "output" per klokkesyklus, noko som gjer at ein eventuelt kan senke klokkefrekvensen og oppnå samme yting som med den serielle løysinga. Ulempa er at den vil ta opp meir areal. Intuitivt vil det krevje **n** gongar så mykje areal for å brette ut ei seriell løysing med **n** steg. På grunn av optimalisering stemmer ikkje dette. I [**27**] kjem ein fram til at ein treng om lag tre gongar så mykje areal for å implementere ei parallell løysing på åtte steg. Grunnen til dette er mellom anna at ein kan hardkode vinkelskift-koeffisientane fordi kvar eining brukar ein fast koeffisient.

Kva blir då arealbruken til dei ulike implementasjonane? (x representerer antal iterasjonar)

TYPE IMPLEMENTASJON	Antal CORDIC-einingar	Areal	Antal registere (16-bit)	Areal	Totalt arealforbruk
Iterasjon med samme eining	1	15.63 + LU-tabell(x)	6	3,45	19,08 + 0,6x
Samleband	x	15,78x	6x	3,45x	20,23x
Fleire samleband	x^2	15,78x^2	6x^2	3,45x^2	20,23x^2

Tabell 11

Som tabellen over viser kan ein ofre areal for å oppnå høgare gjennomstrøyming utan å auke klokkefrekvensen. For å implementere som eit samleband treng ein x einingar samt register for å mellomlagre resultat. Dette gjev høgare x gongar større gjennomstrøyming på same klokkefrekvens, men introduserer også ein latens på x klokkesyklusar.

5.5 Implementasjon med oppslagstabell

Som tidlegare beskrive er det mogleg å implementere approksimasjonen av arcus tangens som ein oppslagstabell. Det er ein enkel metode der verdiar av I og Q vil tilsvare plassar i ein tabell som inneheld den tilsvarande faseverdien, som igjen kan nyttast til å avgjere om frekvensen har endra seg. I utgangspunktet må ein ha eit oppslag for alle moglege verdiar for I og Q, men som tidlegare diskutert er det mogleg å redusere dette til ein fjerdedel av verdiane på grunn av symmetrien den viser. Måten å gjere dette på er den same som er beskrive for CORDIC, og vist i Figur 6. Det vil seie:

1. Sjekk verdien på forteiknet på I og Q
2. Slå opp i ein tabell som tilsvarar 1 kvadrant.
3. Legg til ein verdi som tilsvarar den opprinnelege kvadranten.

Dermed gjenstår vil ein akselerator basert på denne metoden bestå av ei samanlikning av forteikn, eit oppslag i ein tabell som tilsvara verdiane i første kvadrant og ein addisjon med ein verdi frå ein tabell med fire verdiar.

Sidan I og Q er 16 bits vektorar med det første bitet som forteikn vil tabellen kunne slå opp i $15^2 \cdot 15^2 = 50\ 625$ verdiar. Sidan denne tabellen må kunne støtte alle symbolbreiddene for GFSK, π/4 –PSK og 8-DPSK må den innehalde stor nok nøyaktigheit til å kunne dekode den høgaste symbolbreidda, 8-DPSK, det vil seie tre bit per symbol. Det er naturlegvis også mogleg å bruke ein tabell for kvar av oppgåvene, men dette vil truleg berre gje marginale fordelar når det gjeld hurtigheit i dekodaren og det vil utvilsomt ta opp eit mykje større areal og dermed auke det statiske effektforbruket. Ein slik oppslagstabell vil krevje:

$$50\ 625\ \cdot 3\ bit = 151\ 875\ bit \approx 125\ kB$$

Formel 18

Det tilsvarar eit relativt areal på:

$$125 \cdot 290 = 36250$$ **Formel 19**

Dette er eit svært høgt tal og det vil vere urealistisk å implementere ei slik løysing. Skulle ein implementere ei slik løysing må ein kanskje definere eit mindre område gyldige (I,Q)-koordinatar eller på eit slags vis avrunde dei før ein sjekkar tabellen. Her kan til dømes eit system som nyttar ein begrensar garantere at I og Q ligger innanfor eit gitt område, noko som kan redusere storleiken på tabellen betrakteleg.

5.6 Implementasjon av GFSK

I avsnitt 4.3.1 er det presentert ei analyse av GFSK-demodulatoren. Denne konkluderte med at kjerna i demodulatoren er ei utrekning av den deriverte av invers tangens. Dei instruksjonane som må utførast for å gjennomføre denne demodulasjonen blir også presentert.

Denne funksjonaliteten kan implementerast på ei rekke ulike måtar. I dette avsnittet blir nokre av dei mest aktuelle løysingane diskutert og vurdert opp mot kvarandre. På øvste nivå er det aktuelt å anten implementere denne funksjonaliteten som eit program som nyttar dei tilgjengelege grunnleggande operasjonane til prosessoren, eller implementere den ved hjelp av applikasjonsspesifikke instruksjonar, som ein akselerator.

Umiddelbart kan ein sjå at multiplikasjonane er uavhengig av kvarandre, og om ein hadde **to multiplikatorar** tilgjengeleg vil ein kunne utføre desse over i berre **to steg,** om ein har **fire multiplikatorar** tilgjengeleg kan ein utføre det i **eit steg**. Subtraksjonen og addisjonen er uavhengig av kvarandre, men avhengig av multiplikasjonane, så desse vil kreve eit eller to steg avhengig av kva slags einingar ein har tilgjengeleg. Divisjonen er avhengig av addisjonen og subtraksjonen og må difor utførast til slutt. Dette gjev nokre ulike moglegheiter for implementasjon.

Nokre av desse moglegheitene blir utdjupa i dei neste avsnitta.

5.6.1 Implementasjon utan akselerasjon

Den første utfordringa ved å implementere denne funksjonaliteten ved hjelp av instruksjonar er at det ikkje er optimalt å gjere dette før ein har bestemt korleis arkitekturen og instruksjonssettet ser ut. Sidan prosessoren som skal nyttast skal vere særskild tilpassa denne applikasjonen er ein nøydd til å ta hensyn til korleis applikasjonen oppfører seg før ein tar endelege avgjersler. Dermed blir det viktig å diskutere korleis ein kan endre arkitekturen og instruksjonssettet for å optimalisere køyring av applikasjonen samtidig som ein ser på korleis applikasjonen oppfører seg. Metoden vald for dette er å ta utgangspunkt i ein enkel arkitektur og så sjå på korleis applikasjonen vil oppføre seg på denne.

Arkitekturen som blir tatt utgangspunkt i desse karakteristikkane:

- RISC-instruksjonssett (inkludert divisjon og MAC)
- Moglegheit for å hente to verdiar samtidig frå minne (dvs. to AGU-ar)
- Samleband tilsvarande det presentert i Figur 8 (fem steg)
- Behandlar/”sender ut” ein instruksjon per klokkesyklus
- “Load-store”-minne

- Når ein instruksjon "stallar" for å vente vil dette stoppe alle dei etterfølgjande instruksjonane uavhengig av avhengigheit. Dei føregåande instruksjonane vil fullførast, utan dette er det umogeleg å komme ut av "stall [**12**]."

Utover dette er det ikkje tatt noko avgjersle på oppbygnad av minne, I/O antall og storleik på register, dette blir diskutert seinare.

Nokre hensyn er allereie tatt til applikasjonen. Analyse av både denne forma for demodulasjon og DPSK viser at divisjon er viktig i begge applikasjonane. Difor er det antatt at denne bør implementerast i maskinvare og ikkje i programvare.

Ved hjelp av denne modellen kan ein sjå på korleis applikasjonen vil oppføre seg og kva krav den vil stille til prosessoren. Dette gjev naturlegvis ikkje eit heilt nøyaktig bilete av ytinga til applikasjonen på den gitte prosessoren, fordi metoden ikkje tek hensyn til andre prosessar som køyrer samtidig. Det den derimot kan gjere er å gje eit samanlikningsgrunnlag mellom ulike implementasjonar av GFSK-demodulasjon. Det vil også gje ein peikepinn på kva krav dei ulike implementasjonane stiller til arkitektur og instruksjonssett.

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15	t16	t17
1	H	D	L	Mul	S												
2		H	D	L	Mul	S											
3			H	D	-	-	L	Sub	S								
4				H	-	-	D	L	Mul	S							
5							H	D	L	Mul	S						
6								H	D	-	-	L	Add	S			
7									H	-	-	D	-	-	L	Div	S

Figur 16: Tidsplan for GFSK

Figuren over viser korleis GFSK-demodulatoren vil oppføre seg på den hypotetiske prosessoren, instruksjonane er nummerert frå ein til sju, slik som i avsnitt 4.3.1, og klokkesyklusane er nummerert t1-17. Den antek (feilaktig) at det berre er GFSK-demodulasjon som køyrer, om ein tek med dei andre prosessane som til ei kvar tid er i gang på prosessoren vil det nok vere fleire resurskonfliktar og det vil resultere i fleire stopp i instruksjonane.

Som ein kan sjå frå denne figuren vil det ta 17 klokkesyklusar mellom kvar gong ein har demodulert ein "sample" i bitstraumen. Med ein samplingsfrekvens på 16 MHz vil det seie at ein må ha ein klokkefrekvens på minst:

$$F_{s,min} = 16MHz \cdot 17 = 272\ MHz \qquad \textbf{Formel 20}$$

Her kan ein sjå at ytinga kan forbetrast ved å parallellisere nokre av einingane. Som tidlegare nemd er multiplikasjonsoperasjonane mogleg å parallellisere. Med to einingar vil det vere mogleg å utføre den same funksjonaliteten på 15 klokkesyklusar. Med fire kan ein utføre det på 14 steg. Om ein i tillegg har to ALU-ar tilgjengeleg blir det mogleg å utføre funksjonen på 9 klokkesyklusar. Noko som vil tilsvare ein minimum klokkefrekvens på :

$$F_{s,min} = 16MHz \cdot 9 = 144\,MHz$$ **Formel 21**

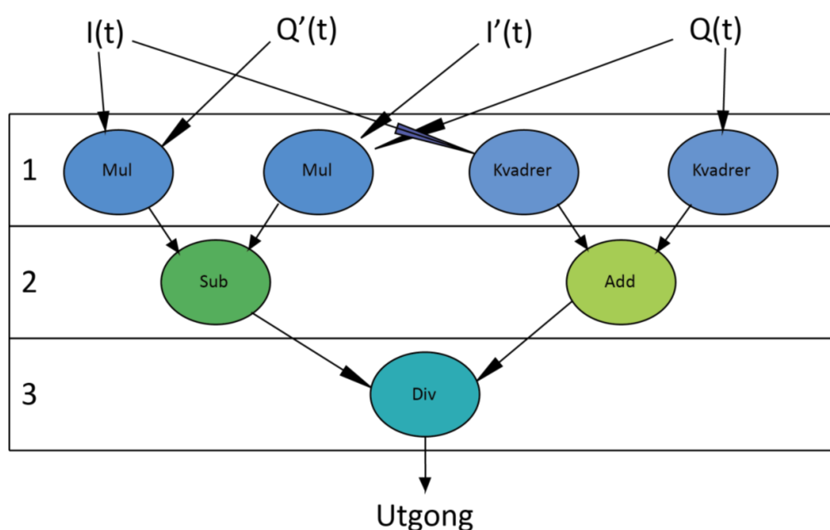
Som ein ser kan dette dramatisk senke den naudsynte klokkefrekvensen og fordi dette betyr at ein kan senke driftsspenninga kan ein då også senke effektforbruket dramatisk. Konsekvensen av å dette er at ein må gjere visse endringar på den DSP-en som er brukt som modell.

Ein konsekvens som er tydeleg er at ein no må ha moglegheita til å utføre meir enn ein instruksjon i gongen, det betyr at ein må gå vekk frå RISC og over til **VLIW**. Dette betyr at ein må ha fleire AGU-ar, større bandbreidde til minne, fleire register.

Sidan det maksimale antalet parallelle instruksjonar her er fire kan det tenkas at ein lang instruksjon bør bestå av minst fire enkelt-instruksjonar. Det er mogleg at det vil gje fordelar å ha enda fleire enn dette sidan prosessoren køyrer andre funksjonar samtidig som den køyrer sjølve demodulasjonen, ein må til dømes i tillegg gjere ei dekoding, men denne er relativt enkel då den med harddekoding berre omfattar å gjere ei samanlikning med ein gitt verdi og å sette ein verdi i minnet (utgongen av demodulatoren) til den dekoda symbolverdien.

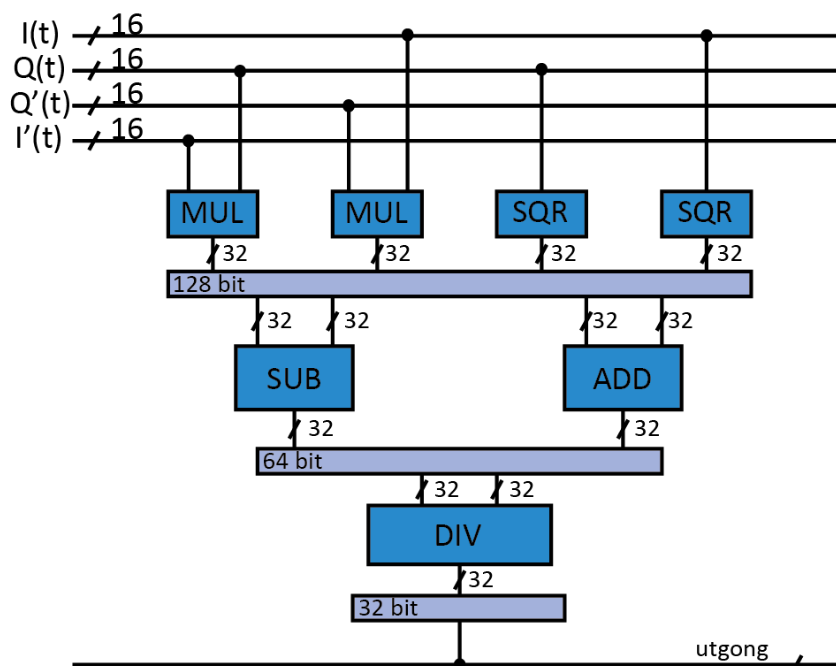
5.6.2 Implementasjon med akselerasjon

Figuren under (Figur 17) viser korleis ein kan implementere dette. Kvar av “boblene” representerer ein operasjon eller ei funksjonell eining, kvar av linjene merka 1-3 representerer “steg” i operasjonen. Det vil seie at dei ulike stega er avhengig av at operasjonane i steget over er ferdige før dei kan gjere noko. Desse representerer også ei moglegheit for oppdeling i eit eventuelt samleband. Det er naturlegvis mange måtar å implementere dette på, alt etter kva arealforbruk og yting ein ønskjer. Eininga vist i figuren nyttar to multiplikatorar og to kvadrerarar i første “steg.” Til dømes: om ein vil bruke mindre areal kan ein dele opp “steg 1” i fleire steg ved å gjenbruke ein multiplikator. Dette fører til mindre arealforbruk, men også lågare yting, anten som lågare gjennomstrøyming eller, i tilfellet den er implementert som eit samleband, som høgare latens på grunn av fleire steg i samlebandet.



Figur 17: GFSK

Dette kan implementerast slik:



Figur 18: Implementasjon av den deriverte av arctan

Implementasjonen vist i figuren over er satt opp som eit samleband på tre steg. Det første steget består av to multiplikatorar og to kvadrerarar, resultatet frå desse lagrast i eit register på 128 bit (eller 4x32 bit). I neste steg utføres subtraksjon og addisjon parallelt og resultatet frå desse lagrast i eit register på 64 bit (2x32bit). Det siste steget i samlebandet er ein divisjonsoperasjon mellom to 32 bits verdiar, resultatet frå denne lagrast og er tilgjengeleg på utgongen.

Det er og mogleg å implementere den utan samleband, men dette fører til at ein må ha ein lågare maksimal klokkefrekvens, fordi den kritiske stien gjennom kretsen vil bli lenger. Det som avgjer den maksimale klokkefrekvensen for eit slikt samleband er det mest krevjande steget. I dette tilfellet er dette divideraren. Det er verdt å merke seg at ved å implementere ein limiter, som beskrive tidlegare, er det mogleg å anta at ein ikkje vil behøve denne divisjonen, fordi ein då kan anta at amplituden til signalet er konstant. Dermed treng ein heller ikkje å implementere kvadreringsoperasjonane og den tilhøyrande adderaren.

Det relative arealet til desse løysinga er:

	MUL	**SUB**	**SQR**	**ADD**	**DIV**	**Reg**	**Areal**
Utan limiter	2x 6=12	1x1=1	2x6=12	1x1=1	1x70	224/16 x 0,575= 8,05	104,5
Med limiter	12	1				112/16 x 0,575 = 4,025	17

Tabell 12

Tabellen over viser at ein limiter vil ha stor påverking på arealet på kjerna, og det er truleg at dette er ei god løysing for systemet.

5.6.3 Implementasjon med CORDIC

Sidan CORDIC kan brukast til å finne arcus tangens kan ein også bruke den til å finne den deriverte av arcus tangens. Den deriverte kan hentast ut ved at ein samanliknar den approksimerte arcus tangens til to etterfølgjande samplar.

Dette vil krevje nokre klokkesyklusar ekstra fordi ein vil ha behov for å lagre resultatet og samanlikne det med føregåande samplar.

5.7 Implementasjon av DPSK

5.7.1 Implementasjon utan akselerasjon

Dette avsnittet tek dei same føresetnadane som er beskrive i diskusjonen rundt GFSK i avsnitt 5.5. I tillegg til dette må ein velje ein av dei foreslåtte implementasjonane av arcus tangens. Approksimasjonen som er beskrive i Formel 17 har ein maksimal feil på 0,07 radianar, noko som er tilstrekkeleg lite, og difor er denne eit godt val for samanlikning. Approksimasjonen består av ein divisjon og ein multiplikasjon.

Dette medfører at denne delen av programmet ser slik ut:

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11
1	H	D	L	Div	S						
2		H	D	L	Mul	S					
3			H	D	-	-	L	Sub	S		

Figur 19: DPSK i samleband

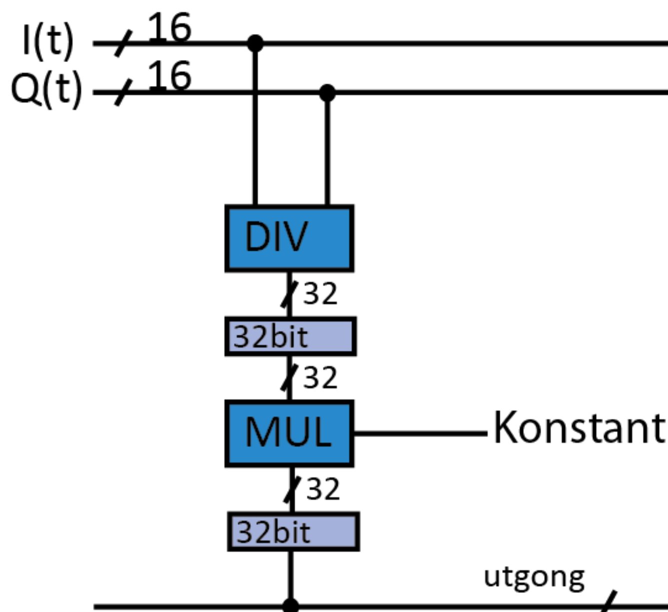
Som ein kan sjå er dette programmet langt mindre enn det som er vist for GFSK, men det er verdt å merke seg at det programmet også tar høgde for å dele på amplituden av signalet. Det gjer ikkje dette. Det er også verdt å merke seg at begge programma også vil krevje desisjon.

Den mest krevjande operasjonen her er divisjonen, og om ein skulle velje denne typen implementasjon vil det vere naturleg å implementere ei divisjonseining i datastien, fordi det er svært tidkrevjande å implementere divisjon ved hjelp av andre instruksjonar. Samtidig er ei divisjonseining er svært arealkrevjande å implementere, i tillegg er det ein krevjande operasjon som sannsynlegvis vil vere bestemmande for den maksimale frekvensen til systemet ved å senke denne betrakteleg.

Som ein ser vil denne kodebiten bruke 9 klokkesyklusar, ikkje inkludert dekoding.

5.7.2 Implementasjon med akselerator

Den foreslåtte akseleratoren for denne eininga er basert på den same approksimasjonen som er brukt i avsnittet over. Det vil seie den beskrive i Formel 17 på side 47. Denne involverer ein divisjonsoperasjon og ein multiplikasjon. Som beskrive for akselerasjon av GFSK er eininga implementert som eit samleband, her med to steg representert i figuren under som register som mellomlagrar resultatet. Ein implementasjon av dette kan sjå slik ut:



Figur 20: Aksellerasjon av DPSK

Det relative arealet for denne løysinga blir 70 + 12 + 2,3 = 84,3. Her utgjer divideraren den største delen med 70, multiplikatoren 12 og registera 2,3.

5.7.3 Implementasjon med CORDIC

Måten arcus tangens kan approksimerast ved hjelp av CORDIC er grundig beskrive tidlegare i avsnitt 2.5.1. Vidare er det i avsnitt 4.4 konkludert med at ein treng tre bits nøyaktigheit for å kunne gjere gode nok approksimasjonar for den gjeldande applikasjonen. Det vil seie at ein treng ein CORDIC med tre steg.

Dette kan anten implementerast ved at ein brukar ein brukar tre CORDIC-einingar på samleband eller ei som itererer fleire gongar. For at denne ikkje skal bruke meir enn ein klokkesyklus, noko som vil sakke ned samlebandet, kan det vere aktuelt å tredoble klokkehastigheita til denne eininga i forhold til resten av systemet.

Det relative arealet til desse to løysingane blir då 20,88 for den iterative løysinga og 60,69 for samlebandsløysinga.

5.8 Samanlikning og diskusjon

Fordi dei to applikasjonane skal kunne operere på den same prosessoren er det naudsynt å vurdere korleis dei ulike løysingane samla påverkar arkitekturen til prosessoren. For å kunne seie noko om kor stort effektforbruk dei forskjellige løysingane vil ha relativt tek ein utgangspunkt i formelen for dynamisk effektforbruk (Formel 1) og antar ein grunnleggande frekvens f_1 som ei av løysingane nyttar. Det vil også vere trygt å anta at dei ulike løysingane vil ha den same aktiviteten, fordi dei utfører den same oppgåva. Statisk effektforbruk er ikkje tatt med, men relativt areal vil også påverke det statiske effektforbruket betydeleg. C er her antatt å vere proporsjonal med arealet til eininga. Det er verdt å merke seg at det er urealistisk at akseleratoren som involverer ei divisjonseining og ei multiplikasjonseining kan oppnå like høg maksimal frekvens som CORDIC som er mykje mindre kompleks og som truleg vil ha ein kortare kritisk sti, det er også verdt å merke seg at samlebandsløysinga til CORDIC vil by på større moglegheit for å senke Dvd fordi denne potensielt har ein høgare

maksimal frekvens. Tabellen nedanfor gjev derfor ikkje eit særskild realistisk bilete av effektforbruket, men den indikerer korleis dei ulike løysingane skil seg frå kvarandre.

Løysing	Relativt Areal	Relativ frekvens	Relativt effektforbruk
Oppslagstabell	36250	f_1	$36250 \cdot f_1 \cdot V_{dd}^2$
To akseleratorar med samleband	84,83 + 104= 188,83	f_1	$188 \cdot f_1 \cdot V_{dd}^2$
CORDIC utan samleband	20,88	$3f_1$	$20{,}88 \cdot 3f_1 \cdot V_{dd}^2$
CORDIC med samleband	60,88	f_1	$60{,}88 \cdot f_1.V_{dd}^2$

Tabell 13

Som ein kan sjå nyttar løysinga med spesialiserte akseleratorar om lag tre gongar så mykje areal som CORDIC-løysinga, og når ein veit at dei kan oppnå om lag same gjennomstrøyming som CORDIC , samtidig som dei gjev mindre moglegheiter for å skalere ned spenninga og samtidig ikkje gjev den same fleksibiliteten som CORDIC er det tydeleg at valet bør falle på CORDIC.

Om ein skal velje mellom å auke frekvensen i ei eining eller nytte eit samleband er avhengig av kva krav ein stiller til arealbruk, men det er truleg at samlebandet kan gje eit lågare dynamisk effektforbruk fordi det er mogleg å nedskalere driftsspenninga. Ulempa med samlebandet er at det vil gje høgare statisk effektforbruk på grunn av arealet, så kva som vil lønne seg av dei to med tanke på effektforbruk er også avhengig av kor ofte eininga blir brukt.

Konklusjonen er at det er hensiktsmessig å inkludere ei CORDIC-eining for å akselerere DPSK og GFSK.

5.9 Programstorleik

For å kunne berekne kor stort programminnet og programcachen bør vere må ein sjå på kor store dei ulike programma og kjernane i programma vil vere gitt at ein:

1. Nyttar 16-bits instruksjonar
2. Antar at ein brukar ei CORDIC-eining for å akselerere GFSK og DPSK. Dvs. at ein har ein CORDIC-instruksjon tilgjengeleg
3. Elles har ein standardinstruksjonar tilgjengeleg
4. Antek at ein har maskinvare for løkkehandtering
5. Antek at ein har moglegheit til å laste inn to verdiar frå dataminnet samtidig

Kjerna til begge programma består som vist av ei løkke som demodulerar og dekodar “samples” frå ADC til binære symbol. Sidan programma er relativt like er det DPSK, med to symbol vald som døme, eit program med fleire verdiar vil måtte ha fleire forgreiningsoperasjonar. For å få storleiken på programmet kan ein sjå på dei operasjonane som blir utførte og analysere kva slag instruksjonar som må til for å få til dette.

Operasjon:	Instruksjon	Beskriving	Storleik
Les inn (I,Q)	LOAD(I, a,Q, b)	Lastar inn I i register a og Q i register b	16 bit
Arcus tangens (I/Q)	COR (a,b,0) -> c	Utfører CORDIC med inngangsverdiane I, Q og 0. Setter e til den resulterande vinkelverdien	16 bit
Subtraksjon: - c-d	SUB(c,d) -> e	Subtraherer den føregåande verdien for å sjekke om den har endra seg	16 bit
Samanlikning: z - 1	BLO(e,1)	Eit betinga hopp i programmet om e er mindre enn 1	16 bit
Sett utgangsverdi:	STORE(0, UT) STORE(1, UT)	Dette er ein betinga instruksjon der den dekoda verdien 1 eller 0 blir lagra i hovudmannen.	32 bit
Mellomlagre førre sample	SET(d,c)	Setter registeret d til verdien i c	16 bit
TOTAL	-	-	112 bit

Tabell 14

For akkurat dette programmet vil det halde med ein instruksjonscache på 112 bit. Om ein skal støtte tre bits symbol må programmet innehalde to fleire forgreiningsinstruksjonar som leiar til åtte lagringsinstruksjonar. Dette vil medføre at det største programmet av denne typen blir på 288 bit.

6. Forslag til arkitektur

6.1 Instruksjonssett

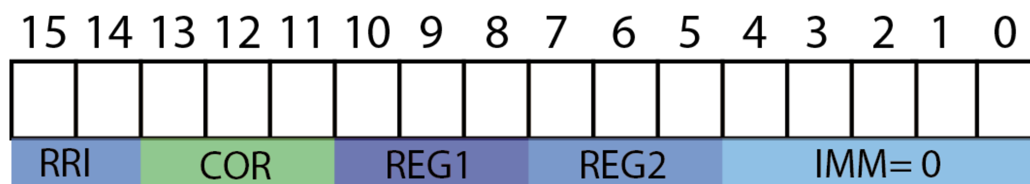
Dette avsnittet tek for seg eit forslag til eit instruksjonssett, det er ikkje beskrive i like stor detalj som akseleratorane og arkitekturen til prosessoren, men det gjev ein oversikt over korleis strukturen til instruksjonane bør vere samt dei "uvanlege" instruksjonane som applikasjonen antyder. Med dette meiner ein dei instruksjonane som må til for å kunne utnytte akselerasjon av applikasjonen. Det er vald å ikkje presentere dei meir grunnleggande instruksjonane, desse er kortfatta presentert i kapittel 3.

Det bør nok vere moglegheit for minst fire instruksjonstypar, branch, løkke og to typar datasti-strukturar. Det tilseier at ein treng **to bit** for å kode desse. Vidare kan ein anta at det bør halde med **tre bit** for å kode dei ulike operasjonane. Det gjev moglegheita til å ha åtte operasjonar innanfor kvar av instruksjonstypane. Dei to formata for datasti-instruksjonar kan til dømes vere RRI (Register –Register-Immediate) og RRR (Register Register Register), som viser til kva variablar som blir operert på. Registera blir vist til ved hjelp av registeradresser medan umiddelbare tal blir vist til ved å sette dei direkte inn i instruksjonen.

Den einaste dedikerte akseleratoren i systemet er ein CORDIC og for å belyse korleis ein instruksjon i prosessoren kan sjå ut er denne vald for å demonstrere korleis instruksjonssettet kan byggast opp.

CORDIC-eininga tar inn tre variablar: x, y og z. Om ein skal berekne arcus tangens så initialiserast eininga med x = I, y = Q og z = 0. Dette betyr at instruksjonen må ta inn to registerverdiar og ein umiddelbar verdi.

Det betyr at strukturen for ein CORDIC-instruksjon som blir brukt til å approksimere arcus tangens vil sjå slik ut:



Figur 21: Døme på CORDIC-instruksjon

Der RRI står for **R**egister **R**egister **I**mmediate og viser til strukturen til instruksjonen, COR er CORDIC-operasjonen, REG1-2 er dei to registera som inneheld I og Q og IMM=0 viser til den umiddelbare verdien null som initialiserar vinkelakkumulatoren. Det er verdt å merke seg at instruksjonen også kunne vore på forma RRR, då hadde dei siste bita vist til eit register som inneheld verdien som initialiserar vinkelakkumulatoren.

Analyse av applikasjonen har avslørt eit behov for **registeradressering, indirekte adressering og umiddelbar data**. **Moduloadressering** kan også tenkast å vere nyttig med tanke på at ein har ein har ein programcache med ei programkjerne som sjeldan må lastast på nytt frå hovudmannen. Moduloadressering kan effektivisere dette. Direkte adressering verkar i så måte ikkje aktuelt, fordi ein ikkje treng direkte tilgang til minnet og bit-reversert adressering verkar også unødvendig med tanke på at applikasjonen ikkje skal utføre noko FFT.

6.2 Arkitektur

6.2.1 Parallellitet

Analyse av applikasjonen har ikkje avslørt særskilde ytingskrav som antyder at ein må ha mange parallelle funksjonseiningar, med tanke på krav til gjennomstrøyming, men det er vanskeleg å seie noko bestemt om dette utan å gjere simulasjonar av systemet som heilheit. For at prosessoren skal vere så liten og enkel som mogleg kan er det mogleg å implementere prosessoren skalart, det vil seie at den utstedar ein instruksjon per klokkesyklus.

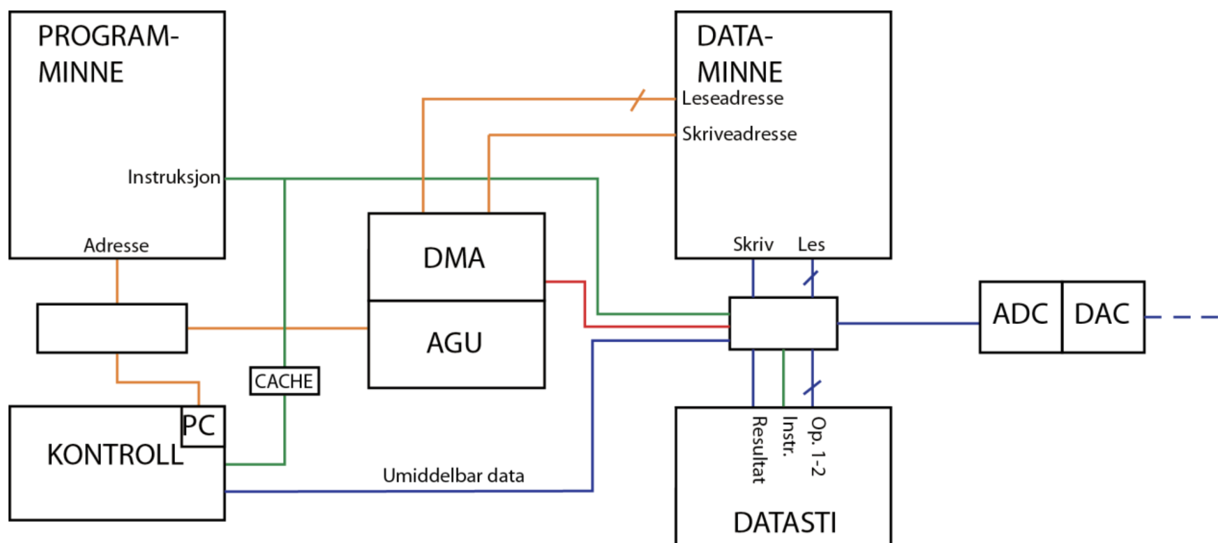
Med dette sagt så er det mogleg at det vil lønne seg å implementere ein viss grad av VLIW eller SIMD. Dette vil truleg gjere at ein kan senke frekvensen i systemet samt minke hyppigheita på spørjingar til programminnet ved at ein anten lastar mange instruksjonar samtidig (VLIW) eller nyttar ein instruksjon til å behandle mykje liknande data. Det siste alternativet verkar det ikkje som om det er særskild stort behov for, då det ikkje førekjem så mykje av denne typen databehandling i applikasjonen. Dermed er konklusjonen at det kan vere fordelar ved i å implementere ei form for VLIW, men dette er må undersøkast vidare, og ein må gjere ei vurdering om auket i yting vil vege opp for den auka kompleksiteten og arealet det medfører seg.

Samleband

For å få låg kompleksitet nyttar den foreslåtte prosessoren eit typisk RISC-samleband med fem steg, som presentert i avsnitt 2.6.1. Det vil seie at ein har stega “Hent”, “Dekod”, “Utfør”, “Minne-aksess” og “Skriv”.

6.2.2 Minne

Dette avsnittet tek for seg strukturen til minnet. Denne blir basert på ein Harvard-arkitektur utvida med cache, også kjend som SHARC (“Super Harvard Architecture”) [**15**].



Figur 22: Minnearkitektur

Figuren over viser minnearkitekturen til den foreslåtte prosessoren. Her viser blå linjer data, gule linjer adresser og grøne linjer instruksjonar. Den raude linja er ei kontrollinje som er tatt med for å indikere rolla til DMA i minnearkitekturen, dette er utdjupa i Figur 23.

Storleik og oppbygnad til dataminnet

Minne har eit særs høgt statisk effektforbruk, i tillegg er det knytt høg effektkostnad til minneaksessar. Som tidlegare nemnt vil ein difor gjerne minimere både storleiken på minnet og kor mange minneaksessar ein har. Fordi fleksibilitet også er eit mål for dette systemet må minnet vere stort nok til å kunne innehalde moglege applikasjonar. Som vist i Figur 22 vil ein dele opp dataminnet og program i to ulike områder, dette gjer at ein kan velje eit ulikt design på dei to delane. Desse to delane kan ta form av eit scratchpad-minne, eller eit større hovudminne pluss ein cache. Dei ulike eigenskapane knytt til desse to løysingane er presentert i avsnitt 2.4.3.

Som tidlegare presentert er dataen til ein programvareradio prega av det ein kan kalle ein datastraum med sanntidsdata. Det vil seie at data kjem inn i systemet frå ei ekstern kjelde med jamne mellomrom og må behandlast og sendast ut igjen innanfor eit gitt tidsrom.

Dataen består av sampler som kjem frå ADC eller til DAC, samt midlertidig data som skal behandlast av datastien og data som kjem frå eller til den applikasjonen som sendaren nyttast i (“payload”). Det tilseier at ei god løysing kan vere å dele opp dataminnet i eit lite hovudminne som til ei kvar tid held den data som ikkje umiddelbart er under behandling i datastien, og eit minne som er lokalt for datastien og berre held midlertidig data.

På grunn av den relativt lille datamengda vil det neppe vere hensiktsmessig med eit stort dataminne. Dette medfører vidare at det heller ikkje er naudsynt med ein cache fordi den vil vere relativt lik hovudmannen i storleik, dermed vil det ikkje gje noko særleg betring i yting og **“overheaden”** som er knytt til å laste ting ut og inn frå hovudmannen, samt halde dataen i cachen koherent vil auke effektforbruket heller enn å senke det. I tillegg vil er det problematisk at minneaksessane blir **udeterministiske** fordi applikasjonen er eit sanntidssystem. Dette gjer det vanskeleg å vere sikker på at systemet fungerer som det skal i alle situasjonar.

Som vist i Tabell 1 er "scratchpad"-basert minne meir energieffektivt og meir areal-effektivt enn cache-baserte løysingar. Vidare vil scratchpad-basert minne gje den ønska deterministiske latensen for minneaksessar som er ønskeleg for eit sanntids kommunikasjonssystem. Valet av denne typen minne forvanskar programmering, men dette er eit akseptabelt kompromiss sidan det er snakk om eit integrert system som ikkje er opent for anna enn svært spesialiserte applikasjonar.

Dermed kan ein konkludere med at dataminnet bør bestå av eit mindre scratchpadminne samt eit noko større hovudminne. Det er vanskeleg å seie noko om kor stort dette minnet bør vere på grunn av at det er vanskeleg å seie noko om dette utan å ha grundig analysert minnebruken til ulike aktuelle applikasjonar. Ein kan seie at det ikkje treng å vere særskild stort på grunn av at digitale kommunikasjonsapplikasjonar bærer preg av ein datastraum der det ikkje er særskilde behov for å lagre mykje data over lengre periodar, men ein liknande prosessor [**28**] antyder at dataminnet kan vere omlag **1kB.** Ein viss del av dette, til dømes 20 %, vil då bli brukt som scratchpad. Truleg er 1kB meir enn det som krevast for denne spesifikke applikasjonen, men det vil ramme fleksibiliteten å ha for lite dataminne, fordi andre applikasjonar kan nytte meir.

Storleik og oppbygnad til programminnet

Programminnet skil seg frå dataminnet i at det kan innehalde ein del data som ikkje blir brukt i lengre periodar. Spørsmålet er om ein ønskjer seg konfigurerbarheit og fleksibilitet ved

fabrikasjonstid eller køyretid. Om ein berre ønskjer seg konfigurerbarheit ved fabrikasjonen kan ein laste inn berre akkurat det programmet som skal køyrast av prosessoren og dermed treng ein berre stor nok plass til programma tilhøyrande akkurat denne applikasjonen.

Programdømet i avsnitt 5.9 har ein anslått storleik på 288 bit, eller 36 byte. Dette kan gje ein indikasjon på kor stort programminnet bør vere. Dette utgjer kjerna til programmet som demodulerar DPSK. I tillegg til dette kan ein anslå at kjerna til GFSK-demodulatoren vil vere om lag like stor. Utover dette vil programminnet må innehalde program for taktgjenvinning. Ein kan anslå at dette programmet kanskje vil vere to-tre gongar større enn kjerna. Dermed ender ein opp på ein programstorleik totalt på om lag 200 byte. For å få fleksibiliteten ein ønskjer seg i ein SDR bør det vere plass til meir enn eit program av denne storleiken. Dermed kan ein seie at programminnet bør vere minst **1 kB**. Om ein samanliknar dette med minnestorleiken i ein meir generell lågeffekt DSP er dette eit lågt tal. Texas Instruments sin TMS320VC5401 har til dømes 16kB RAM og 4kB ROM [**61**], eit tal som rett nok inkluderer programminnet, men det demonstrerer at minnet er relativt lite. Sidan minne er så kritisk for effektforbruk kan det vere aktuelt å redusere det ned til ein absolutt minimal storleik om prosessoren ikkje skulle oppfylle krav sette til effektforbruk.

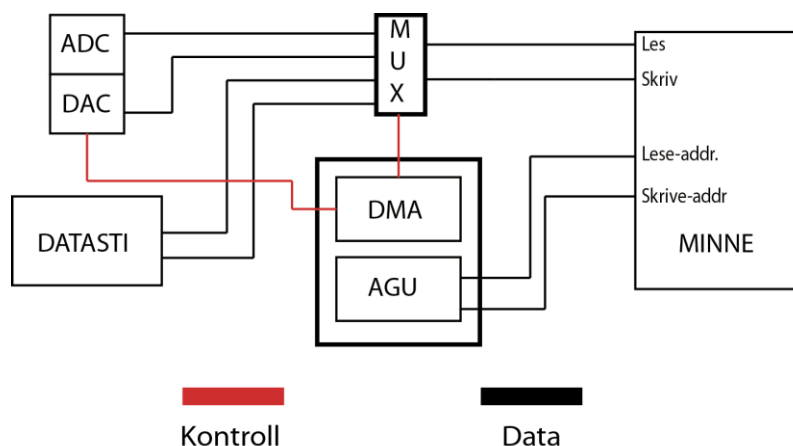
Storleik og funksjon til instruksjonscache

For å sleppe å gjere mange aksessar til det relativt store programminnet vil det vere naudsynt med ein instruksjonscache mellom programminnet og datastien/kontrolleininga. Denne bør vere stor nok til å innehalde kjerna til det programmet som skal køyrast, eventuelt heile programmet om dette er mogleg.

Instruksjonscachen bør vere stor nok til å innehalde minst kjerna i eit typisk program. I dette tilfellet er dette omlag 36 byte. For å ta høgde for program som er noko større kan ein auke dette til **40 byte**.

I/U-handtering

Avbrot ("interrupts") vil auke kompleksiteten og minske gjennomstrøyminga i systemet, dette vil truleg føre til eit høgare effektforbruk, fordi ein vil måtte drive systemet på ein høgare klokkefrekvens. Med tanke på at det er snakk om eit reelt sanntidssystem vil det også gjere systemet meir uforutsigbart, men dette vil truleg ikkje vere så problematisk som det ville vore i eit større system, fordi applikasjonen er relativt enkel og forutsigbar og kan planleggast statisk. Likevel vil det på grunn av "overhead" knytt til avbrots-driven I/U truleg vere betre å implementere ein enkel DMA-kontroller som let ADC og DAC få direkte tilgang til data-bussen og minnet. Eit forslag til eit slik løysing er vist i figuren under.



Figur 23: DMA i minnestrukturen

Som figuren viser styrer DMA saman med AGU tilgangen til databussen. Dette skjer ved at ADC eller DAC signaliserer at denne har data dei vil skrive til minnet. Her kan ein merke seg at sidan desse aldri vil operere samtidig, vil berre ein av dei ha data som skal skrivast eller lesast.

Fordi ADC og DAC vil ha lågare operasjonsfrekvens enn kjerna vil det ikkje vere kritisk at desse får prioritet, men dette kan planleggast statisk sidan samples skal tas med jamne mellomrom.

Adressegenereringseining (AGU):

Sidan det er ønskeleg å kunne laste data frå to adresser samtidig vil det vere nødvendig å ha to AGU-ar tilgjengeleg for å generere adresser til dataminnet. I tillegg må ein ha ein AGU for å generere adresser til programminnet, denne kan integrerast med programtellaren. For at samlebandet skal fungere som ønska og ikkje krevje ekstra klokkesyklusar for adressegenerering må desse einingane fungere parallelt med datastien. Utover dette er

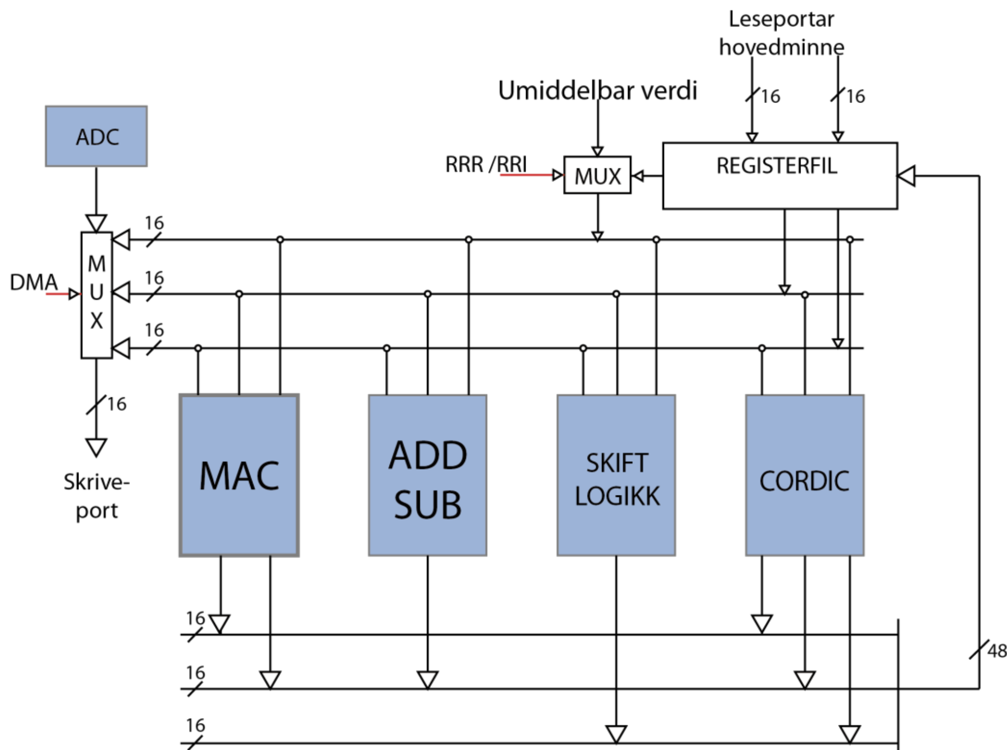
Registerfil (datadel av denne):

Formatet gitt for instruksjonane kodar to register med tre bit kvar, det vil seie at desse kan nyttast til å adressere 2^3= 8 register. Vidare kan dei siste fire bita nyttast til å adressere 2^4= 16 bit kvar. Det sistnemnde gjeld dessverre berre om desse refererer til eit eige sett register som ikkje kan brukast “om kvarandre” med dei førstnemnde koda med tre bit. Dette er noko upraktisk og bergrensande, og difor vil det kanskje vere meir hensiktsmessig å berre nytte tre av desse bita, slik at ein kan vise til det same settet register som dei første to registerfelta i instruksjonen. Dermed kan ein konkludere med at det maksimalt kan vere **åtte register** i registerfila, gitt at ein ikkje implementerer spesialiserte register for enkelte instruksjonar eller meir kompleks koding.

6.2.3 Datasti

Figuren under viser datastien i prosessoren, og er ein illustrasjon på korleis data blir behandla i systemet. I tillegg er det tatt med to kontrollsignal for å klargjere kva som styrer dei to muxane som er vist i figuren, desse er merka med raude linjer. Dei ulike einingane kan ha ein til tre inngangsverdiar der anten alle er register eller ein av dei er ein umiddelbar verdi som er spesifisert i instruksjonen. Kva av desse som blir brukt er avgjort av ein mux som er

styrt av instruksjonstypen spesifisert i instruksjonen, anten RRR eller RRI. Grunnen til at ein treng tre linjer på 16 bit inn er at CORDIC-eininga må initialiserast med tre verdiar.



Figur 24: Datastien

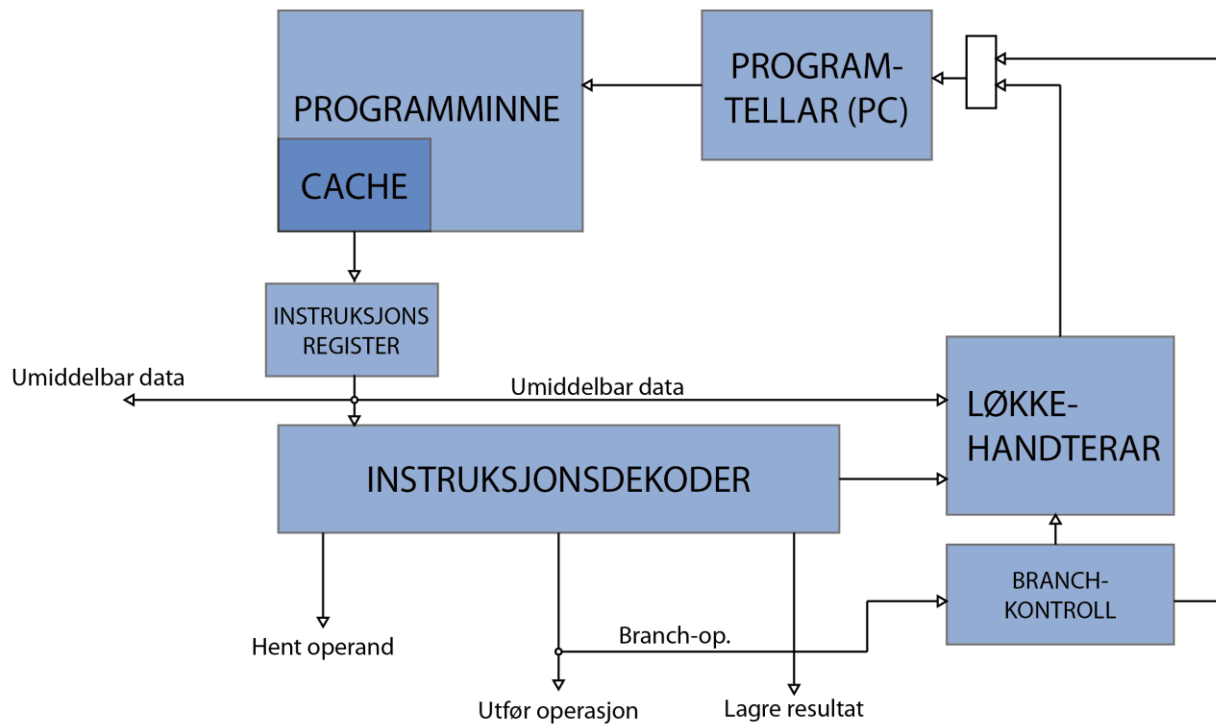
Figuren viser også at MAC-eininga har to utgangsportar, dette er for å ta høgde for verdiar opp til 32 bit lange. Det same kan implementerast for ADD/SUB-eininga om det er naudsynt. CORDIC-eininga har tre utgangsverdiar og treng derfor tre linjer på 16 bit kvar for å skrive desse til register.

DMA kontrollerer om ADC eller ALU-en skal ha tilgang til å skrive til hovudmannen, gjennom ein 4-1 mux. Hovudminnet har som tidlegare bestemt ein lagringsport på 16 bit, så om større verdiar enn dette skal lagrast krevjast det meir enn ein klokkesyklus. Systemet har ein load/store arkitektur og alle operasjonar føregår difor på register eller umiddelbare verdiar. ADD/SUB-, skift/logikk. og MAC-einingane treng i utgangspunktet ikkje å vere tilkopla tre inngangar kvar og desse kan reduserast til to portar, men ein av desse må vere kopla slik at den kan påtrykkas umiddelbare verdiar.

6.2.4 Kontrollsti

Dette avsnittet tek for seg kontrollstien i den foreslåtte prosessoren. Dette er gjort på ein forenkla måte og vil ikkje ta for seg alle aspekter ved kontroll i prosessoren, intensjonen er heller å belyse nokre viktige punkter i kontrollstien, som rollene løkkehandteraren og "branch"-kontrolleren spelar i systemet.

Figur 25 gjev eit forenkla overblikk over korleis dei ulike kontrolleiningane fungerer saman. Den utelet mellom anna logikken for korleis timingen i samlebandet vil fungere. Dette er berre markert med portar ut frå instruksjonsdekodaren, men i realiteten vil det krevje register som held dei dekoda operasjonane til kvart av stega i samlebandet slik at desse kan bli påtrykt samlebandet til rett tid. Virkemåten til instruksjonsdekodaren er ikkje diskutert her, men dei andre einingane som inngår i kontrollstien er behandla.



Figur 25: Kontrollstien

Programtellaren

Denne er implementert med eit register og ei inkrementasjonseining. Registeret inneheld adressa til ein instruksjon i programminnet, i dette tilfellet cachen som skal innehalde det aktuelle programmet som skal køyrast. Dette registeret endrar seg ein gong kvar klokkesyklus, anten blir det inkrementert med ein eller så blir det satt til den adressa løkkehandteraren eller "branch"-kontrollen påtrykker det.

Løkkehandteraren

Ein løkkehandterar er implementert i maskinvaren for å forbetre handtering av løkker. Denne står i kviletilstand til instruksjonsdekodaren tek imot ein løkke-instruksjon. Den vil deretter lagre den gjeldande instruksjonsadressa samt ein umiddelbar verdi som initialiserar løkketellaren, samt ein verdi som fortel kor mange instruksjonar som finnes i løkka, denne vil initalisere instruksjonstellaren. Desse må vere spesifisert i som umiddelbar data i instruksjonen, og dei vil ha ei øvre grense avgjort av kor stort dette feltet i instruksjonen er. Den nedre grensa er på ein instruksjon.

Løkkehandteraren vil gå gjennom det fastslåtte antalet instruksjonar ved å la instruksjonstellaren telle ned medan PC inkrementerer instruksjonsadressene som vanleg. Når instruksjonstellaren har nådd 0 påtrykker løkkehandteraren programtellaren adressa lagra i det interne instruksjonsregisteret og dekrementerar løkketellaren. Når løkketellaren når 0 er løkka avsluttast løkka, løkkehandteraren går i kvilemodus, og programtellaren fortsetter å inkrementere instruksjonar som vanleg.

Analysen av applikasjonen har ikkje avslørt noko behov for å behandle "løkker-i-løkker". Det vil seie at det ikkje er noko behov for å implementere ein stakk for løkker.

Umiddelbar data
Umiddelbar data er data som er skrive direkte inn i instruksjonane. Som vist på figuren er desse kopla til løkkehandteraren, slik at denne kan initalisere løkke- og instruksjonstellarane, samt datastien slik at denne kan påtrykkas data koda inn i instruksjonane.

Branch-kontroll
Når instruksjonsdekodaren får inn "hopp"- eller forgreiningsinstruksjonar vil branch-kontrollen stoppe programtellaren og i staden påtrykke ei ny instruksjonsadresse i programtellaren sitt interne register. Om hoppet skjer medan ein er inne i ei løkke vil det også få konsekvensar for løkka, fordi det er mogleg at instruksjonstellaren i løkkehandteraren må oppdaterast med ein ny verdi. Det kan til dømes vere eit hopp vil medføre at ein skal utføre ein mindre eller ein meir instruksjon i løkka enn det instruksjonstellaren var initialisert med opprinneleg, då må instruksjonstellaren anten inkrementerast eller dekrementerast med ein.

7. Konklusjon og vidare arbeid

7.1 Konklusjon

Det har blitt presentert ein ASIP DSP-arkitektur med låg kompleksitet spesielt tilpassa basebandsprosessering av lågeffekt SDR-applikasjonar. Fokuset har vore på gjere ei avveging mellom lågt effektforbruk og høg fleksibilitet. Grunnen til fokuset på lågt effektforbruk er at den aktuelle applikasjonen er retta mot ein handhalden mobil radio med kort rekkevidde og krav til lang batterilevetid.

Framgangsmåten har vore ei analyse av ein typisk applikasjon, basert på Bluetooth, med spesielt fokus på demodulasjon, som blei ansett som den mest krevjande oppgåva og dermed også drivande for krava applikasjonen stiller til prosessoren.

Denne analysa førte til konklusjonen at utrekning av arcus tangens er ei viktig kjerne i applikasjonen og det vil derfor vere hensiktsmessig å akselerere ein approksimasjon av denne funksjonen. Det blei foreslått ei rekke ulike akselerasjonsmetodar og ein kom fram til at ei løysing basert på CORDIC truleg ville vere beste av løysinga med tanke på kriteria fleksibilitet og effektforbruk.

Analyse av applikasjonen avslørte også at ein dedikert løkkehandterar kan forbetre ytinga til prosessoren, denne skal ha moglegheita til å handtere løkker som består av blokker av instruksjonar, men den behøver ikkje å implementerast med kapasitet for meir enn eit nivå.

Det vart også konkludert med at ein instruksjonscache kan betre ytinga og effektforbruket til prosessoren ved å redusere hyppigheita til spørjingar til programminnet. Den naudsynte storleiken til denne cachen blei berekna til om lag 40 byte. Det maksimale naudsynte minnet blei berekna til omlag 1 kB kvar for programminne og dataminne, men dette kan reduserast betrakteleg om ein ofrar noko av fleksibiliteten.

7.2 Vidare arbeid

Denne oppgåva kan sjåast på som første steg på veg mot å implementere ein DSP tilpassa lågeffekt SDR. For det første er det mogleg å gjere ei mykje breiare analyse av applikasjonen, både visse delar av den presentert i denne oppgåva, særskild taktgjenvinning som er utelatt her, og andre aktuelle applikasjonar. For det andre gjenstår det å beskrive korleis virkemåten til dei ulike einingane i datastien og kontrollstien skal implementerast, dette er utelatt med unntak av CORDIC, men dette kan vere relativt trivielt fordi dei kan baserast på velkjende løysingar som allereie er i bruk i liknande DSP-ar.

Ein må også definere eit komplett instruksjonssett og bestemme korleis dette skal dekodast.

Vidare vil målet vere å lage ein modell av prosessoren slik at denne kan verifiserast. Dette medfører også at ein kan gjere relativt nøyaktige estimat på arealforbruk, effektforbruk og yting. Til slutt vil ein kunne realisere prosessoren i maskinvare, først som FPGA og om mogleg som ASIC.

8. Referansar

[1] Wireless Innovation Forum -. (2011, Mai) Software Defined Radio - Defined. [internett]. http://www.wirelessinnovation.org/page/Introduction_to_SDR

[2] SDR Forum, SDRF Cognitive Radio Definitions, 2007.

[3] Tony J. Rouphael, *RF and Digital Signal Processing For Software-Defined Radio*, første utg. Burlington, USA: Newnes, 2009.

[4] Martin Maråk, Lågeffekt Programvareradio, 2010, Fordjupningsprosjekt knytta til Masteroppgåva.

[5] Wayne Wolf, "Building the Software Radio," *IEEE - Embedded Computing*, Mars 2005.

[6] Stephen M. Blust, Perspective on Software Defined Radio Focusing on Reconfiguration and (Radio) Software Download, 2003.

[7] Behzad Mohebbi, Eliseu Chavez Filho, Rafael Maestre, Mark Davis og Fadi J. Kurdahi, "A Case Study of Mapping a Software-Defined Radio (SDR) Application on a Reconfigurable DSP Core".

[8] Infineon Technologies AG, Product Brief: X-GOLD SDR 20 Programmable Baseband Processor for Multi-Standard Cell Phones, 2009.

[9] Peter Kenington, *RF and Baseband Techniques for Software Defined Radio.*: Artech House, Inc., 2005.

[10] Luca Benini og Giovanni Di Micheli, "System-Level Power Optimization: Techniques and Tools," *ACM Transactions on Design Automation of Electronic Systems*, vol. V, nr. 2, April 2000.

[11] Hallvard Næss, "A programmable DSP for low-power, low-complexity baseband processing," NTNU, Trondheim, Masteroppgåve 2006.

[12] Roger M. Koteng, "Evaluation of SDR-implementation of an IEEE 802.15.4," NTNU, Masteroppgåve 2006.

[13] Yuan Lin et al., "SODA: A Low-power Architecture For Software Radio," i *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[14] Michael Schulte et al., "A Low-Power Multithreaded Processor for Software Defined Radio," *Journal of VLSI Signal Processing Systems*, vol. 43 , nr. 2-3, ss. 143-159, Juni 2006.

[15] John. G Proakis og Dimitris G. Manolakis, *Digital Signal Processing - Principles,*

Algortihms and Applications, Fjerde utg.: Pearson Prentice Hall, 2007.

[16] John L. Hennessey og David A. Patterson, *Computer Architecture: A Quantative Approach*, Fjerde utg.: Elsevier, Inc., 2007.

[17] Dake Liu, *Application Specific Instruction Set Processors*. Linköping, Sverige: Morgan Kauffman, 2007.

[18] Lars Wanhammar, *DSP Integrated Circuits*.: Academic Press, 1999.

[19] BORES Signal Processing. (2010) Introduction to DSP - DSP processors. [internett]. http://www.bores.com/courses/intro/chips/6_basics.htm

[20] John P. Uyemura, *Introduction to VLSI Circuits and Systems*.: John Wiley and Sons, Inc. , 2002.

[21] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan og Peter Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," i *Proceedings of the tenth international symposium on Hardware/software codesign*, Estes Park, Colorado, USA, 2002, ss. 73-38.

[22] P. R. Panda et al., "Data and memory optimization techniques for embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, nr. 2, ss. 149-206, April 2001.

[23] Oliver Schliebusch, Heinrich Meyr og Rainer Leupers, *Optimized ASIP Synthesis from Architecture Description Language Models*.: Springer, Inc., 2007.

[24] Lennart Yseboodt et al. (2007, August) EE|Times Design : Ultra-low-power DSP design. [internett]. http://www.eetimes.com/design/signal-processing-dsp/4017529/Ultra-low-power-DSP-design

[25] Heinrich Meyr, "System-on-Chip for Communications: The Dawn of ASIPs and the Dusk of ASICs.," i *Workshop on: Signal Processing Systems*, August 2003, ss. 4-5.

[26] Manoj Kumar Jain, M. Balakrishnan og Kumar Anshul, "ASIP design methodologies: survey and issues," i *Fourteenth International Conference on VLSI Design*, Bangalore, India, 2001, ss. 76-81.

[27] Kingshuk Karuri et al., "Fine-grained application source code profiling for ASIP design," i *Proceedings of the 42nd annual Design Automation Conference*, New York, 2005, ss. 329-334.

[28] Manoj Kumar Jain, *Exploring Register File and Memory Organization in ASIP Synthesis*. Dehli, India: Indian Institute of Technology Dehli, 2004.

[29] T.V.K Gupta, Purvesh Sharma, M Balakrishnan og Malik Sharad, "Processor Evaluation in an Embedded Systems Design Environment," i *Thirteenth International Conference on VLSI Design*, Calcutta, India, 2000, ss. 98-103.

[30] Vojin G. Oklobdzija og Ram K. Krishnamurthy, *High-performance energy-efficient microprocessor design.*: Springer, 2006.

[31] Anil Kumar Maini, *Digital electronics: principles, devices and applications*. India: John Wiley & Sons, 2007.

[32] Hazarathaiah Malepati, *Digital Media Processing - DSP Algorithms Using C*, Første utg.: Elsevier Inc., 2010.

[33] Dimitrios Hristu-Varsakelis og W. S. Levine, *Handbook of networked and embedded control systems.*: Springer, 2005.

[34] Rajeshwari Banakar, Stefan Steinke, Boo-Sik Lee og Peter Marwedel, "Comparison of Cache- and Scratchpad- based Memory Systems with respect to Performance, Area and Energy Consumption," University of Dortmund, Dortmund, Technical Report 2001.

[35] John Catsoulis, *Designing Embedded Hardware*, Andre utg., Andy Oram, Red.: O'Reilly Media, Inc., 2005.

[36] Götz Kappen, Sofian el Bahri, Oliver Priebe og Tobias G. Noll, "Implementation of a CORDIC based double precision floating point unit used in an ASIP based GNSS receiver," Electrical Engineering and Computer Systems, Aachen Universitet,.

[37] Ray Andraka, "A Survey of CORDIC Algorithms for FPGA Based Computers," i *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, Monterey, 1998, ss. 191-200.

[38] Scott Hauck og André DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation.*: Morgan Kaufmann, 2008.

[39] Young Bok Kim, Yong-Bin Kim og J.T. Doyle, "A Low Power CMOS CORDIC Processor Design for Wireless Telecommunication," i *Midwest Symposium on Circuits and Systems*, Montreal, Kanada, 2007, ss. 1336-1339.

[40] Alfonso Luís Troya Chinchilla, "Synchronization and Channel Estimation in OFDM: Algorithms for Efficient Implementation of WLAN Systems," Brandenburgischen Technischen Universität, Cottbus, Doktorgradsavhandling 2004.

[41] N. Kavvadias og K. Masselos, "Efficient Hardware Looping Units for FPGAs," i *IEEE Computer Society Annual Symposium on VLSI*, 2010, ss. 35-40.

[42] Ya-Lan Tsao, Wei-Hao Chen, Wen-Sheng Cheng, Maw-Ching Lin og Shyh-Jye Jou, "Hardware nested looping of parameterized and embedded DSP core," i *SOC Conference Proceedings*, 2003, ss. 49-52.

[43] Randall Hyde, *The Art of Assembly Language*, andre utg. San Fransisco, USA: No Starch Press, 2003.

[44] Bruce Jacob, "The RiSC-16 Instruction-Set Architecture," University of Michigan, 2000.

[45] Bluetooth SIG. (2011, Mars) Bluetooth Basics. [internett]. http://www.bluetooth.com/Pages/Basics.aspx

[46] "IEEE Std 802.15.1-2002," IEEE, 2002.

[47] Agilent Technologies, "Bluetooth® Enhanced Data Rate (EDR): The Wireless Evolution - Application Note," 2006.

[48] Curt Franklin og Julia Layton. (2000, Juni) How Stuff Works - Bluetooth. [internett]. http://electronics.howstuffworks.com/bluetooth.htm

[49] Jaap Haartsen, "BLUETOOTH—The universal radio interface for ad hoc, wireless connectivity," *Ericsson Review*, nr. 3, ss. 110-117, 1998.

[50] Ahmed Ahmed Eladawy Emira, "Bluetooth/WLAN Receiver Design Methodology and IC Implementations," Texas A&M University, Doktorgradsavhandling 2003.

[51] Vladimir Dvorkin, James Wong og Min Zou. (2004, Februar) Microwaves and RF - Quad Demodulators Arm Direct-Conversion Receivers. [internett]. http://www.mwrf.com/Article/ArticleID/7470/7470.html

[52] Adel S. Sedra og Kenneth C. Smith, *Microelectronic Circuits*. Oxford, England: Oxford University Press, 2004.

[53] H. Labiod, H. Afifi og C.D. Santis, *Wi-Fi, Bluetooth, Zigbee and WiMAX*.: Springer, 2007.

[54] Emanuele Lopelli, J.D. Van Der Tang og A.H.M. Van Roermund, "FSK Demodulator Topologies for Ultra-low Power Wireless Transcievers," ss. 241-246.

[55] A. Wannsarnmaytha, S. Hara og N. Morinaga, "A novel FSK demodulation method using the short-time DFT analysis for LEO satellite communications systems," i *Global Telecommunications Conference*, 1995, ss. 549-553.

[56] John G. Proakis og Masoud Salehi, *Digital Communications*, 4th utg.: McGraw-Hill, 2001.

[57] Simon S. Haykin, *Digital Communications*.: Wiley, 1988.

[58] Richard Lyons, "Another Contender in the Arctangent Race," *IEEE SIGNAL PROCESSING MAGAZINE*, ss. 109-110, Januar 2004.

[59] N. Phamdo og F. Alajaji, "Soft-decision demodulation design for COVQ over white, colored, and ISI Gaussian channels," *IEEE Transactions on Communications*, vol. 48, nr. 9, s. 1499, September 2000.

[60] Dave Van Ess, Algortihm - Arctan as Fast as you Can, Januar 2006.

[61] Jasper Vijn. (2009, Februar) Off on an Arctanget. [internett]. http://www.coranac.com/documents/arctangent/

[62] Karl Rottmann, *Matematisk Formelsamling.*: Spektrum Forlag, 2006.

[63] Sreeraman Rajan, Sichun Wang, Robert Inkol og Alain Joyal, "Efficient Approximations for the Arctangent Function," *IEEE Signal Processing Magazine*, ss. 108-111, Mai 2006.

[64] Sherif Galal og Dung Pham, Division Algorithms and Hardware Implementations, Forelesning ved UCLA.

[65] Volnei A. Pedroni, *Circuit Design with VHDL.*: Massachussetts Institute of Technology, 2004.

[66] Texas Instruments. (2011, Juni) TMS320VC5401 - Fixed-Point Digital Signal Processor. [internett]. http://focus.ti.com/docs/prod/folders/print/tms320vc5401.html

[67] Umakanta Nanda og Kamalakanta Mahapatra, "Design of an Application Specific Instruction Set Processor Using LISA," i *International Confrence on Advanced Computing and Communication*, Kanjirapally, India, 2010, ss. 206-210.

9. Vedlegg

9.1 Vedlegg 1: Kode og pseudokode

Vedlegg 1.1 : Pseudokode for GFSK

```
1)   T = bitperiode
2)   T_s = sampleperiode
3)
4)   While(1){
5)   for i=0:T_s:T-1{                        // Summerar det gonga signalet over ein
     bitperiode
6)
7)   I_n   = I_n + mottatt*oscillator;       // Gongar signalet med ein lokal oscillator
8)   Q_n = Q_n + mottatt*oscillator_90;}
9)
10) I_n = lavpassfilter(I_n);
11) Q_n = lavpassfilter(Q_n);
12)
13) I_der = I_n – I_n_temp;                  // Finn den deriverte av I_n
14) Q_der = Q_n – Q_n_temp;
15)
16) y= ( I_n *Q_der – Q_n*I_der)/ ( I_n^2 + Q_n^2);   // Approksimasjon av invers tangens (11)
17)
18) I_temp = I_n;                            // Lagrar I(n-1) og Q(n-1)  til neste bitperiode
19) Q_temp = Q_n;
20)
21) If( y>0)                                 // avgjer verdien av utgangssignalet
22)         utgang = 1;
23) else
24)          utgang = 0;
25) end if;
26) }
```

Vedlegg 1.2 : Pseudokode for DPSK

```
1)    Tb = periode for eit bit;
2)    Fs = samplefrekvens;
3)
4)    While(1) {
5)    iSignal = mottatt · lokal_0_oscillator;   // Her gongast innsingalet (mottatt) med ein lokal oscillator
6)    qSignal = mottatt ·lokal_90_oscillator;
7)
8)    iSignal = lavpassfilter(iSignal);
9)    qSignal = lavpassfilter(qSignal);
10)
11)   in_0   =   integrer(iSignal, Tb);   // Det sampla signalet blir integrert over ein bitperiode
12)   qn_0 =   integrer(qSignal, Tb);
13)
14)   x = in_0/qn_0;                      // Deling for å finne forholdet mellom I og Q
15)
16)   y_1 =  arctan(x)                      // Finner arcus tangens
17)
18)   z = y_1 + y_0;
19)
20)   if( y > 0 )
21)            utgang = 1;               //  Den demodulerte verdien  er 1
22)   else
23)            utgang = 0;               // Den demodulerte verdien  er  0
24)   end if;
25)
26)   y_0 = y_1;                     // Lagrar verdi for samanlikning med neste gong
27)    }
```

Vedlegg 1.3: CORDIC C-kode

Kjelde: Sverre Wichlund

```
1)  double cordic(int x, int y, double *lookup, int iter)
2)  {
3)    char i,d;
4)    double phaseaccu;
5)    int xn,yn,tx,ty;
6)
7)    /* first, check if we are in the left half plane */
8)    if(x<0)
9)              {
10)                   phaseaccu=-pi;
11)                    xn=-x;
12)                    yn=-y;
13)               }
14)             else
15)              {
16)              phaseaccu=0;
17)              xn=x;
18)              yn=y;
19)       }
20)
21)   for(i=0;i<iter;i++)
22)   {
23)     /* first, determine which way to rotate */
24)     if(yn>=0)
25)                   d=-1;
26)     else
27)                   d=1;
28)
29)         /* then rotate in the correct direction and update xn,yn and accumulator */
30)                        /*tx=xn-yn*d/pow(2,i);*/          /* done as simple shift in HW */
31)                       /*ty=yn+xn*d/pow(2,i);*/          /* done as simple shift in HW */
32)      tx=xn-((yn*d)>>i);
33)     ty=yn+((xn*d)>>i);
34)     xn=tx;
35)     yn=ty;
36)     phaseaccu-=d*lookup[i];
37)   }
38)
39)   return(phaseaccu);
40) } /* end cordic */
```

Vedlegg 1.4: Matlab-kode for utrekning av maksimal feil for CORDIC-algoritma

```
%% Reknar ut den maksimale feilen for CORDIC i grader

vinklar = 0:0.001:90;
cordic = [45 26.5651 14.0362 7.1250 3.5763 1.7899 0.8952 0.4476];

maksfeil = zeros(1,8);

for i=1:length(cordic)
    for k=1:length(vinklar)

    temp = vinklar(k);

        for a=1:i
         if(temp>0)
             temp = temp - cordic(a);
          else
           temp = temp + cordic(a);
         end
        end

    resultat = temp;

    if(abs(resultat)>maksfeil(i))
        maksfeil(i)= abs(resultat);
    end;

    end
end
```

9.2 Vedlegg 2: ADL-basert ASIP-design:

Dette vedlegget er tatt med fordi den omtalar ein interessant metode for ASIP-design, som kan vere aktuell i framtida eller for store prosjekter. Desverre er metoden relativt “umoden” og i tillegg krever den dyre komersielle verktøy, noko som gjer den uaktuell for denne oppgåva.

Arkitekturbeskrivande språk (ADL – “architecture descrition language”) gjev ein formell metode/språk for å beskrive maskinvarearkitekur. Som mellom anna Liu [**13**] poengterar (i 2007) er diverre ikkje denne metodologien moden for bruk, og i dei fleste tilfeller vil det ikkje vere mogleg å nytte den på grunn av det løysingsrommet for arkitektur er for stort til at ein kan generere god nok maskinvare utifrå eit instruksjonsett. Dette avsnittet er likevel tatt med for å belyse ein alternativ og mogleg framtidig metode for ASIP-design.

ADL kan grovt sett delast opp i tre kategoriar [**18**]:

1. *Instruksjonssett-sentrert metode:*
   Denne metoden setter fokus på å beskrive arkitekturen frå perspektivet til ein programmerar som skal nytte ASIP-en. Dei brukast i hovedsak til å beskrive korleis instruksjonane skal kodast og oppførselen og syntaksen til “assembly”..omformuler...
2. *Arkiktektur-sentrert metode*:
   Fokuserar på å beskrive strukturen til systemet. Oppførselen til arkitekturen delast opp i blokker etter funksjon, og prosessoren beskrivast ved samansetninga av desse og koplingane mellom dei.
3. *Blanda metode (arkitektur og arkitektur):*
   Denne metoden kombinerar dei to første metodane. Nokre døme på slike språk er LISA og EXPRESSION.

I **EXPRESSION** beskrives arkitekturen ved hjelp av ulike komponentar og kommunikasjonen mellom desse. Komponentane delast opp i fire ulike klasser: einingar (t.d. aritmetiske einingar ALU), minneelementer(t.d. register), portar og til slutt samankoplingar (t.d. bussar).

Instruksjonssettet kan beskrivast ved ei rekke operasjonar. Desse består av operasjonskode, operandar, oppførsel og formatdefinisjon for instruksjonssettet. Vidare har språket ei rekke pre-definerte funksjonar som skal beskrive oppførselen til instruksjonsettet. Funksjonar som ikkje finnes blandt desse må settast saman av dei pre-definerte funksjonane. Desse blir delt opp i grov-vevde (“coarse-grain”) operasjonar (t.d. ein minneaksess) og fin-vevde(“fine-grain”) operasjonar. Dette gjer det mogleg å generere maskinvare utifrå RTL-blokker knytt til dei ulike funksjonane.

Problemet med EXPRESSION er at den “mappar” ein gitt arkitektur direkte til maskinvare, utan optimaliseringar. Sjølv om RTL-blokkene er optimalisert fører dette til at ein får maskinvare som ikkje er så effektiv som mogleg. Vidare støttar ikkje EXPRESSION per i dag (2007) effekt-sparande modi og debugging [**18**].

Desse manglane gjer at denne metoden ikkje er særskild velegna for lågeffektdesign. Ein annan metode/språk er **LISA** (“Language for Instruction Set Architecture”) [**58**]. Slik som EXPRESSION [**18**] gjer LISA eit forsøk på å fylle rommet mellom tradisjonell prosessordesign (t.d. v.h.a. VHDL eller Verilog) og prosessordesign basert på instrusksjonssettsfokuserte språk [**58**].

Ein prosessormodell basert på LISA består på øverste nivå av to delar: resurssdel og operasjonsdel. Instruksjonsresurssen er vanlegvis eit register som held instruksjonar, men det kan også vere ein minnelokasjon, ein inngong eller ei rekke minneelementer. Operasjonsdelen inneheld alle dei funksjonane som prosessoren skal kunne utføre. Under dette ligger beskrivingar av oppførsel, syntaks og koding. Oppførsel beskriv ..., koding beskriver korleis instruksjonane er bygd opp (binært) og syntaksen viser korleis funksjonane ser ut i eit “assembly”-program [**58**].

For å implementere arkitektur utifrå ein beskrivelse i LISA kan ein nytte spesialisert programvare: “LISA Processor Design Platform(LPDP)” frå CoWare.Ved hjelp av ein LISA-modell av ein arkitektur i denne programvaren kan ein automatisk generere programvareverktøy som til dømes C-kompilator, “assembler”, simulator og “profiler” [**58**].