UNIVERSITETET I OSLO
Institutt for Fysikk

Matriseinvertering på FPGA ved hjelp av QR-dekomponering

Masteroppgave

Eirik Kile

15. desember 2006

Sammendrag

Denne oppgaven presenterer en rask måte å invertere komplekse matriser i en FPGA (Field Programmable Gate Array). Motivasjonen for oppgaver har vært behov for rask matriseinvertering i forbindelse med kanalutjevning i Kongsberg Defence Communication (KDC) nye generasjon av taktiske radiolinje, RL532A. Denne skal benytte seg av frekvenshopp for å beskytte seg mot støy og jamming.

Den nye radiolinjen til KDC skal brukes i kupert terreng. Her vil det oppstå refleksjoner som fører til multipath-forsinkelser i det mottatte signalet. Mottakeren vil oppleve dette som støy eller Inter Symbol Interferens (ISI). For å fjerne ISI, benytter radiolinjen seg av en LMS-utjevner (Least-Mean Square). LMS'en finner det inverse kanalfilteret på bakgrunn av data den mottar over tid, men dette blir for tregt når frekvenshopping skal benyttes. LMS'en finner det inverse kanalfilteret mye rasker hvis den får en startverdi. Denne startverdien er det mulig å finne ved hjelp av matriseinvertering. Radiolinjen har derfor bruk for en rask måte å invertere matriser på.

Fixed point QR-dekomponering og tilbakesubstitusjon blir brukt i matriseinverteringen, som er dokumentert å gi numerisk stabile løsninger. Tre forskjellige metoder for å utføre QR-dekomponeringen blir sammenlignet og Givens rotasjon blir valgt som metode. Givens rotasjon blir optimalisert ved bruk av en spesialtilpasset kvadratrot enhet og blir implementert i VHDL med en 11 stegs pipeline. Oppgaven viser også hvordan resten av systemet kan pipelines. Et estimat viser at det er mulig å invertere en 25x25 matrise på under $100\mu s$ med denne implementeringen. Til slutt blir implementeringen sammenlignet mot en annen, som viser at min implementering er 3 ganger raskere.

Forord

Dette er min masteroppgave til graden Master of Science, gitt av KDC. Oppgaven har blitt gjennomført på noe kortere tid en vanlig. KDC ga meg eget kontor i utviklingsavdelingen deres, hvor jeg kunne jobbe med oppgaven i fred og ro. Her var det et inspirerende miljø og gode muligheter til å få hjelp med oppgaven når jeg trengte det. Dette gjorde det mulig å gjennomføre oppgaven på den tiden jeg hadde til rådighet.

Først og fremst vil jeg takke KDC for at jeg fikk tildelt en interessant masteroppgave. Til andre som vurderer å ta en ekstern oppgave, vil jeg anbefale KDC. Her blir man tatt vel imot og får god veiledning fra dag en. Jeg vil takke min eksterne veileder Roar Skogstrøm i KDC og min interne veileder Jim Tørresen ved Universitetet i Oslo. Begge har vært til god hjelp ved utformingen og gjennomlesning av oppgaven.

Jeg takke Asgeir Nysæter, Simen Gimle Hansen og Per-Kristian Remvik på KDC for gjennomlesning og for hjelp med oppgaven. Til slutt vil jeg takke Linda Langtangen ved FMC for gjennomlesning av oppgaven.

Innhold

Figurer

Kapittel 1

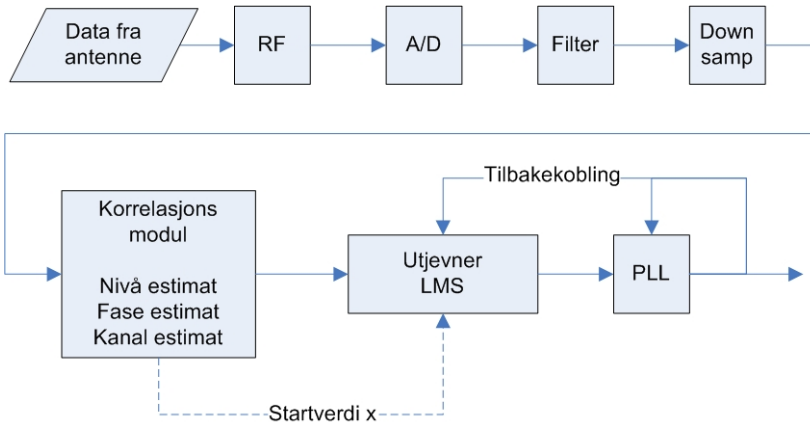
Innledning

Dagens FPGA'er inneholder mye mer aritmetikk og logikk enn tidligere. Dette er en utvikling som fortsetter og gjør det mulig å bruke FPGA'er til stadig mer krevende beregninger. KDC bruker derfor FPGA'er i sin nye taktiske radiolinje, RL532A. Ved frekvenshopping, som brukes for å sikre mot ISI og jamming, kan radioen sende og motta data i full duplex med hastighet opp til 2048 kilobit/s (kbps). Ved å bruke fast frekvens, kan hastigheten økes til 8448kbps.

1.1 Multipath radio

Den taktiske radiolinjen skal kunne brukes i kuperte terreng. Her vil signalene som sendes bli reflektert av fjell og andre større objekter. Siden det er spredning på signalene ut fra antennen, vil bølgene finne flere veier frem til mottakeren. Veiene vil som regel ikke være like lange, så bølgene bruker forskjellig tid på å komme frem til mottakeren. I tillegg vil noen av bølgene svekkes, mens andre forsterkes på grunn av refleksjonene. Dette kalles multipath og vil oppleves som støy for mottakeren.

Mottakeren mottar signalene, som er forvrengt av refleksjonene. Signalene mottakeren får kan uttrykkes ved å filtrere utsendt signal med et filter, et kanalfilter hvor både demping og multipath-forsinkelse er inkludert. Derfor må den ha en utjevner som opphever effekten av kanalfilteret. I radiolinjen bruker KDC en adaptiv utjevner av typen Least-Mean Square (LMS), se Figur 1.1. På bakgrunn av dataene som radioen mottar over tid, tilnærmer metoden seg gradvis en minste kvadraters versjon av det inverse kanalfilteret. Etter en viss tid, vil LMS'en låse seg til det inverse kanalfilteret. Når radioen har låst seg, vil den kunne demodulere dataene.



Figur 1.1: LMS utjevner for å låse seg til en kanal

For beskyttelse mot ISI og jamming, bruker radiolinjen frekvenshopping i kombinasjon med LMS utjevner. Når radioen bytter sendefrekvens, vil signalene reflektere annerledes i terrenget. Utjevneren må derfor innstilles på nytt. Ved mellom 100 og 1000 frekvenshopp i sekundet må LMS-utjevneren reinnstilles like mange ganger. Det betyr at det er behov for en rask reinnstilling.

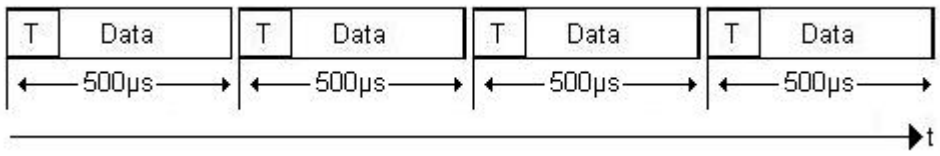
Nedkorting av tiden LMS bruker på å innstille seg gjøres ved å gi den en startverdi (den stiplede linja på Figur 1.1). Startverdien beregnes på bakgrunn av en treningssekvens med kjente data som ligger i starten av hver pakke (se Figur 1.2). Startverdien til LMS-utjevneren er den inverse av kanalfilteret og kalles i denne oppgaven for x. Denne beregnes ved å løse ligningssystemet i ligning 1.1.

$$Ax = b \tag{1.1}$$

I denne ligningen er A en matrise, x og b vektorer. A og b bygges opp av dataene i treningssekvensen, mens x er ukjent. For å finne x, må ligning 1.1 løses med hensyn på x. Dette gir ligning 1.2, som involverer en invertering av A matrisen.

$$x = A^{-1}b \tag{1.2}$$

Å invertere A matrisen er en regnekrevende oppgave. I tillegg inneholder dataene i treningssekvensen komplekse tall, som igjen fører til at tallene i A og b er komplekse. Dette gjør matriseinverteringen av A enda mer regnekrevende. Det blir da viktig å finne en rask og effektiv måte å invertere komplekse matriser. For å få en god startverdi, er det også viktig at matriseinverteringen er nøyaktig.



Figur 1.2: Rammer med treningssekvens og data.

Treningssekvensen ligger i starten av hver ramme og inneholder kjente data. Disse dataene inneholder komplekse tall og brukes til å bygge opp A og b. Ved 8448kbps er det 500μs mellom hver pakke.

1.2 Målsetning

Målsetningen med oppgaven var å finne en rask metode for å invertere en kompleks matrise i en FPGA. Et mål som ble satt ganske tidlig, var at det var ønskelig at metoden skulle kunne invertere en 25x25 matrise på 100μs. Matrise størrelsen skulle det være mulig å gå ned på, men beregningstiden skulle helst ikke overstiges. Metoden bør være presis og bruke minst mulig maskinvare. FPGA'en som radiolinjen bruker, kjører på en frekvens på 53,76MHz. Det er ønskelig at matriseinverteringen kan kjøre på samme frekvens.

Målet er å implementere matriseinverteringen i VHDL, med Xilinx Virtex 4 SX35 som målteknologi. Dette er en stor FPGA med mange multiplikatorer og LUT'er(Look Up Table). KDC har valgt denne FPGA'en til den taktiske radiolinken RL532A. Det er ingen prosessor i FPGA'en, derfor skal matriseinverteringen kun implementeres i VHDL.

1.3 Arbeid med oppgaven

Arbeidet har vært delt inn i 4 deler, med følgende prosentvis fordeling:

1. **20%** Studiefase, her satt jeg meg inn i hvordan QR-dekomponering kunne brukes for å invertere matriser på en FPGA. Det var flere metoder som kunne utføre QR-dekomponering. Jeg valgte ut 3 metoder som jeg ville studere nærmere. Disse var Givens rotasjon, fast-Givens rotasjon og CORDIC rotasjon.

2. **35%** Kreativ fase 1, her implementerte jeg matriseinverteringen i C++. Alle 3 metodene ble implementert i C++ for se hvem av de som var best.

3. **15%** Kreativ fase 2, her implementerte jeg metoden jeg mente var best i VHDL, og simulerte i Modelsim 6.0 SE. Algoritmen ble syntetisert med Mentor Graphics Precision og kjørt Place-and-Route verktøyet i ISE7.1 for å finne arealforbruk og hastighet.

4. **30%** Dokumentasjon av arbeidet.

1.4 Disposisjon av rapporten

Kapittel 1 : *Innledning* til oppgaven.

Kapittel 2 : *Bakgrunn* beskriver teknologien som ligger til grunn for oppgaven.

Kapittel 3 : *Rotasjonsmetoder* inneholder en beskrivelse av forskjellige metoder og hvilken av disse som ble valgt.

Kapittel 4 : *Implementasjon av rotasjonsenheten* er en beskrivelse av den valgte rotasjonsmetoden og hvordan den ble implementert i VHDL.

Kapittel 5 : *Matriseinvertering* En detaljert beskrivelse av hvordan resten av matriseinverteringen kan implementeres i VHDL.

Kapittel 6 : *Test* inneholder beskrivelse av hvordan systemet ble simulert og testet.

Kapittel 7 : *Konklusjon* inneholder konklusjon og forslag til forbedringer

Vedlegg : *CD* inneholder VHDL koden til Givens rotasjon, C++ koden til SoftQR og Matlab programmene. Den inneholder også noen Excel regneark som ble brukt til forskjellige beregninger. ”ReadMe.txt” gir en oversikt over innholdet på CD’en.

1.5 Om lesing av oppgaven

Det forventes at leseren har grunnleggende kunnskaper om matriseregning og regning med komplekse tall. Leseren bør også ha noe kunnskap om digital aritmetikk og FPGA’er. Kapitlene 1-5 bygger på hverandre, så det er viktig å lese disse kapitlene i kronologisk rekkefølge for å få best mulig forståelse av oppgaven.

Jeg har i størst mulig grad forsøkt å bruke norske ord og uttrykk. Men i noen tilfeller er det mer naturlig å bruke engelske fagtermer. Slik som for eksempel ordet *pipeline*.

Kapittel 2

Bakgrunn

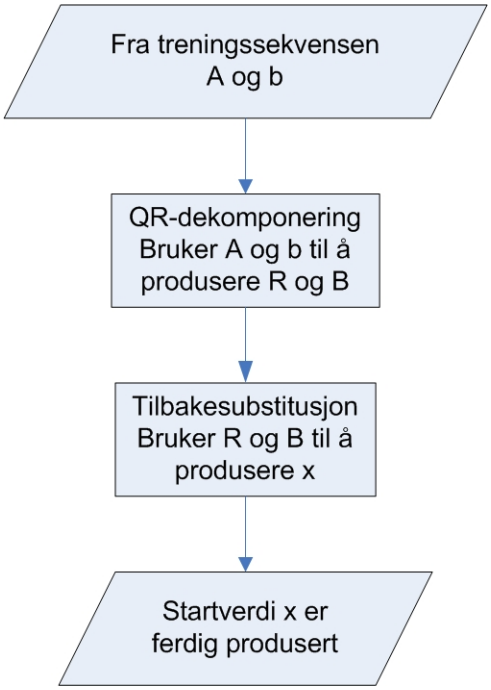
Dette kapitlet beskriver hvordan matriseinvertering kan utføres på en FPGA. Invertering av matriser kan deles inn i to hovedkategorier, direkte og indirekte. De direkte metodene har i noen tilfeller dårlig numerisk stabilitet, det vil si at de ikke alltid vil gi korrekt resultat. [10] viser hvordan en direkte metode utføres og [11] viser et tilfellet hvor den direkte metoden er unøyaktig. Indirekte metoder er mer numerisk stabile og gir mer korrekte resultater.

QR-dekomponering (QRD) og LU-dekomponering er begge indirekte metoder som kan brukes til matriseinvertering. LU-dekomponering er raskere enn QR-dekomponering, men den førstnevnte kan lett bli ustabil [5]. Av denne grunnen valgte jeg å ikke se nærmere på LU-dekomponering.

QR-dekomponering har mer avanserte algoritmer [5] og har god numerisk stabilitet [12]. Det fins flere metoder for å utføre QR-dekomponering [23], noen eksempler er Householders refleksjoner, Givens rotasjoner, Gram-Schmidt, fast-Givens rotasjoner [1] og CORDIC rotasjoner [2]. Dette kapitlet beskriver hvordan QR-dekomponering kan brukes til matriseinvertering og hvilke metoder som er aktuelle.

2.1 Matriseinvertering ved bruk av QR-dekomponering

Denne delen forklarer hvordan matriseinvertering ved hjelp av QR-dekomponering kan brukes til å finne startverdien x. Figur 2.1 viser hvilke deler som trengs for å utføre matriseinverteringen. Som vi ser kan matriseinverteringen deles inn i to hovedmoduler, QR-dekomponering og tilbakesubstitusjon.



Figur 2.1: Flytdiagram for matriseinverteringen

QR-dekomponering og tilbakesubstitusjon[1] brukes til å finne en løsning på ligning 1.1, uten at den finner A^{-1} direkte. Isteden representerer den A som et produkt av en Q og en R matrise (se ligning 2.4), derav navnet QR-dekomponering.

Aller først skal vi se litt på Q og R matrisene. Q er en ortogonal matrise, det vil si at den transponerte[2] av Q (Q^T) multiplisert med Q er lik enhetsmatrisen I, slik som ligning 2.1 viser. Det er verdt å merke seg at dette også gjelder for den inverse av Q (Q^{-1}), se ligning 2.2.

[1] Avsnitt 2.1.7 beskriver hvordan tilbakesubstitusjonen fungerer.
[2] Se avsnitt 2.1.3 for transponering av matriser.

$$Q^T Q = QQ^T = I \tag{2.1}$$
$$Q^{-1} Q = QQ^{-1} = I \tag{2.2}$$
$$Q^{-1} = Q^T \tag{2.3}$$

Dette gir ligningen 2.3, som gir en forenkling av matriseinverteringen. Avsnitt 2.1.3 gir en fyldigere beskrivelse av Q matrisen.

R matrisen er øvre triangulær, det vil si at alle elementene under hoveddiagonalen er lik 0. Dataene vil dermed danne en trekant, derav navnet øvre triangulær. Figur 2.2 viser et eksempel på hvordan R matrisen ser ut. Den øvre triangulære formen til R er en viktig del av matriseinvertering ved hjelp av QR-dekomponering (Se avsnitt 2.1.7).

$$R = \begin{bmatrix} 7 & 8 & 4 & 2 & 2 \\ 0 & 3 & 5 & 7 & 6 \\ 0 & 0 & 5 & 1 & 4 \\ 0 & 0 & 0 & 1 & 9 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

Figur 2.2: Eksempel på en øvre triangulær matrise.

Denne delen beskriver hvordan Q og R brukes i matriseinvertering ved hjelp av QR-dekomponering.

$$A = QR \tag{2.4}$$
$$Ax = b \tag{2.5}$$
$$x = A^{-1} b \tag{2.6}$$
$$x = [QR]^{-1} b \tag{2.7}$$

Ved å sette ligning 2.4 inn i 2.6, får vi ligning 2.7.

$$x = R^{-1} Q^{-1} b \tag{2.8}$$
$$x = R^{-1} Q^T b \tag{2.9}$$
$$x = R^{-1} B \tag{2.10}$$

Ligning 2.7 gir 2.8. Siden Q er en ortogonal matrise så er Q^{-1} det samme som Q^T, dette gir 2.9. I denne oppgaven blir resultatet av $Q^T b$ for ordens skyld kalt B, dermed får vi ligning 2.10. Nå har vi løst ligningssystemet med hensyn på x. Men det er fortsatt behov for å invertere R matrisen. Dette er det mulig å slippe ved å pre-multiplisere begge sider av ligningen med R. Dette gir den endelige ligningen 2.11.

$$Rx = B \tag{2.11}$$

Fordelen med å ha ligningssystemet på denne måten er tydeligere hvis vi ser hvordan matrisen og vektorene ser ut.

$$\begin{matrix} \text{R} & & \text{x} & & \text{B} \end{matrix}$$
$$\begin{bmatrix} 1 & 4 & 2 \\ 0 & 3 & \text{-1} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -5 \\ 2 \end{bmatrix}$$

Figur 2.3: Ligningssystemet

Figur 2.3 viser hvordan ligningssystemet i ligning 2.11 kan se ut. Siden R er øvre triangulær er det mulig å løse ligningssystemet ved hjelp av tilbakesubstitusjon (se avsnitt 2.1.7). Tilbakesubstitusjonen finner ett og ett element i x vektoren av gangen. Den starter med x_2 og fortsetter oppover helt til den har løst x_0. Når den er ferdig, har den funnet startverdien x til LMS-utjevneren. Matriseinverteringen er dermed ferdig.

Som vi ser er det ikke behov for å finne A^{-1} direkte. Men vi finner A^{-1} indirekte ved å finne $[QR]^{-1}$. Dette er grunnen til at matriseinvertering ved hjelp av QR-dekomponering er en indirekte metode. Denne metoden gir færre beregninger enn om vi skulle finne A^{-1} direkte.

2.1.1 Annihilering

Dette er et viktig begrep når det er snakk om QR-dekomponering og betyr det samme som å nulle ut. I matrise sammenheng er det vanlig å si at et element i matrisen blir annihilert. Med dette menes at et elementet på en gitt plass i matrisen blir nullet ut. Figur 2.4 viser et eksempel på dette.

$$\begin{bmatrix} 1 & 4 & 2 \\ 6 & 3 & \text{-1} \\ 3 & 4 & 1 \end{bmatrix} => \begin{bmatrix} 1 & 4 & 2 \\ 3 & 6 & 8 \\ 0 & 3 & \text{-1} \end{bmatrix}$$

Figur 2.4: Eksempel på annihilering av et element i en matrise.

I eksempelet i Figur 2.4 blir elementet nederst i venstre hjørne annihilert. Samtidig som dette elementet blir satt til null, endrer noen av de andre

elementene i matrisen verdi. Dette er for at matrisen fortsatt skal inneholde samme informasjon som tidligere, selv om et av elementene blir satt til null. QR-dekomponeringen bruker annihilering for å gjøre R matrisen øvre triangulær.

2.1.2 Metoder

Som tidligere nevnt fins det flere metoder for å utføre QR-dekomponering. [5] har sett på fordeler og ulemper mellom Givens, Householder og Gram-Schmidt. Her kommer det frem at Gram-Schmidt er raskere og enklere å implementere enn de to andre. Men den kan være ustabil, dette gjør at jeg ikke vil se nærmere på denne metoden. Householder og Givens har omtrent like god stabilitet. Householder er noe raskere enn Givens, men til gjengjeld er den også vanskeligere å implementere [5]. Av denne grunn vil jeg se nærmere på Givens rotasjon.

Det er to metoder til som er ganske like Givens rotasjon, fast Givens rotasjon og CORDIC rotasjon. Både fast Givens og CORDIC har blitt utviklet på bakgrunn av Givens rotasjon. Fast Givens rotasjon er en kvadratrotfri utgave av Givens, mens CORDIC rotasjon kan utføre Givens kun ved hjelp av operatorer som subtraksjon, addisjon og shift. Fast Givens og CORDIC er ikke omtalt i [5], men i [1, 2] blir de sett på som konkurrenter til Givens. Jeg valgte derfor å se nærmere på disse to metodene i tillegg.

Alle tre metodene bruker rotasjoner for å utføre QR-dekomponeringen og har gode muligheter for parallellitet. QR-dekomponeringen deles inn i flere etapper, hvor en rotasjon er en etappe. Alle tre metodene bruker like mange etapper for å utføre QR-dekomponeringen. Givens, fast Givens og CORDIC kalles for rotasjonsmetoder og blir beskrevet nærmere i Kapittel **3**.

2.1.3 Q og R matrisene

Q matrisen blir bygget opp av Givens, fast Givens eller CORDIC rotasjon. Hver rotasjon er en etappe i QR-dekomponeringen og produserer en Q_x. Ved å gange sammen alle Q_x får man tilslutt Q matrisen, se ligning 2.12.

$$Q = Q_n \cdots Q_4 \cdot Q_3 \cdot Q_2 \cdot Q_1 \tag{2.12}$$

For å finne R, tar vi utgangspunkt i ligning 2.4. Denne pre-multipliseres med Q^T på begge sider av ligningen og vi får da ligning 2.13. Av denne ligningen ser vi at oppgaven til Q^T matrisen er å transformere A matrisen til den øvre triangulære R matrisen.

$$Q^T A = R \tag{2.13}$$

Som vi ser så har vi behov for finne Q^T, som er den transponerte av Q. Transponering av matriser er enkelt, her bytter man rett og slett elementene i matrisen som har koordinat [x,y] med [y,x]. Eksemplet i Figur 2.5 under viser transponeringen av matrise Q til Q^T.

$$Q = \begin{bmatrix} \text{-0.8480} & \text{-0.3180} & \text{-0.4240} \\ 0.5223 & \text{-0.3655} & \text{-0.7705} \\ 0.0901 & \text{-0.8748} & 0.4760 \end{bmatrix} => Q^T = \begin{bmatrix} \text{-0.8480} & 0.5223 & 0.0901 \\ \text{-0.3180} & \text{-0.3655} & \text{-0.8748} \\ \text{-0.4240} & \text{-0.7705} & 0.4760 \end{bmatrix}$$

Figur 2.5: Transponering av Q matrisen

Det er mulig å bygge opp Q^T på samme måte som Q matrien. Først må Q_x transponeres til Q_x^T, så kan de multipliseres sammen (se ligning 2.14).

$$Q^T = Q_n^T \cdots Q_4^T \cdot Q_3^T \cdot Q_2^T \cdot Q_1^T \tag{2.14}$$

Når Q^T multipliseres med A, får vi den øvre triangulære matrisen R. Det vil si at alle elementene under hoveddiagonalen i A matrisen blir annihilert. Det vil igjen si at Q^T brukes til å annihilere elementene i A for å lage R.

Det er ikke nødvendig å først regne ut Q^T for å så multiplisere denne med R. Hvis vi setter inn ligning 2.14 inn i 2.13, så får vi ligning 2.15.

$$R = Q_n^T \cdots Q_4^T \cdot Q_3^T \cdot Q_2^T \cdot Q_1^T \cdot A \tag{2.15}$$

I ligning 2.15 kommer det frem at det ikke er nødvendig å beregne hele Q^T, kun alle Q_x^T. Disse kan multipliseres med A ettersom de blir produsert, noe som vil gi færre beregninger og øke hastigheten. Hver gang en Q_x^T multipliseres med A matrisen, blir ett element i A matrisen annihilert. Slik at når Q_n^T blir multiplisert med A, vil alle elementene under hoveddiagonalen være lik 0. R har dermed blitt en øvre triangulær matrise.

2.1.4 b vektoren

b vektoren blir produsert av samme treningssekvens som A matrisen. Likning 2.16 viser forholdet mellom A, b og startverdien x.

$$A \cdot x = b \tag{2.16}$$

Siden A matrisen multipliseres med Q^T for å danne R matrisen, så må b vektoren også multipliseres med Q^T.

$$Q^T \cdot Ax = Q^T \cdot b \tag{2.17}$$

$$Rx = B \tag{2.18}$$

I likhet med R så er det ikke nødvendig å lage Q^T, det er bedre å multiplisere med alle Q_x^T etterhvert som de blir produsert, se likning 2.19.

$$B = Q_n^T \cdots Q_4^T \cdot Q_3^T \cdot Q_2^T \cdot Q_1^T \cdot b \tag{2.19}$$

Når R og B er ferdig produsert, løses ligning 2.18 ved hjelp av tilbakesubstitusjon for å finne x vektoren. Utføringen av tilbakesubstitusjonen blir beskrevet i avsnitt **2.1.7**.

2.1.5 Eksempel på annihilering

Q_x^T brukes til å annihilere elementer i A og oppdatere elementer i A og b, for å lage R og B. Men hvordan ser Q matrisen egentlig ut? Q_x^T matrisen blir produsert av enten Givens, fast Givens eller CORDIC. Alle disse 3 har sin egen måte å produsere Q matrisen på, men virkningen av matrisen er lik. Jeg vil derfor gi et eksempel på hvordan Q_x^T matrisen ser ut når Givens rotasjon brukes.

La oss anta at vi fikk følgende A matrise og b vektor utifra trengingssekvensen. På bakgrunn av verdiene i A matrisen, lager Givens følgende Q_1^T. Verdiene og plasseringen til verdiene i Q_1^T, bestemmer hvilket element som skal annihileres. I dette eksempelet så er det elementet nederst til venstre som skal annihileres. Hvordan Givens lager Q_1^T matrisen blir beskrevet i avsnitt 3.2.

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix} \Rightarrow \quad Q_1^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.6 & 0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix}$$

Q_1^T blir multiplisert på begge sider av ligningen. Dette annihilerer et elementet i A matrisen og oppdaterer b vektoren.

$$Q_1^T \cdot A \cdot x = Q_1^T \cdot b$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.6 & 0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix} \cdot \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.6 & 0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix}$$

Nå har Q_1^T blitt multiplisert med A og b. Vi sitter da igjen med følgende verdier i A og b. Som vi ser så har elementet nederst til venstre i A matrisen blitt annihilert. Samtidig fikk de andre elementene i de to nederste

linjene i A og b nye verdier. Det som har skjedd er at alle elementene i de to nederste linjene i A og b har blitt rotert. I dette eksemplet ser vi at x vektoren ikke blir påvirket av annihileringen.

$$A \cdot x = b$$

$$\begin{bmatrix} 8 & 1 & 6 \\ 5 & 10.2 & 5.8 \\ 0 & 1.4 & -4.4 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 3.4 \\ -1.2 \end{bmatrix}$$

Dette er kun første steg for å produsere R og B utifra A og b. For å produsere R må alle elementene under hoveddiagonalen i A annihileres. Hvordan dette utføres i praksis beskrives i neste avsnitt.

2.1.6 Produsere R og B

R matrisen måtte som sagt være øvre triangulær før den kunne brukes i tilbakesubstitusjonen. Denne delen beskriver en metode for å lage R matrisen og B vektoren ved hjelp av Givens, fast Givens eller CORDIC rotasjon. Disse rotasjonsmetodene brukes til å rotere elementene i matrisen, slik at ett element i matrisen blir annihilert.

Hver rotasjon påvirker kun to linjer i matrisen, hvilke to linjer er avhengig av hvilket element som blir annihilert. Siden hver rotasjon kun påvirker to linjer i matrisen, er det mulig å kjøre flere rotasjoner i parallell. Det vil si at det er mulig å annihilere flere elementer samtidig, noe som kan utnyttes med parallelliteten til en FPGA.

Det er da viktig å velge riktige elementer i matrisen som skal annihileres samtidig. Elementene må velges slik at flest mulig elementer kan annihileres i parallell, men det er også viktig at elementene annihileres i rett rekkefølge. Hvis rekkefølgen er feil, kan rotasjonen føre til at elementer som allerede er annihilert får nye verdier. [15] beskriver en annihileringsrekkefølge som ikke påvirker elementer som tidligere har blitt annihilert.

Denne rekkefølgen kan brukes på alle matrisestørrelser. Figur 2.6 viser et eksempel på annihileringsrekkefølgen til en 6x3 matrise[3]. Tallene i A matrisen er ikke ment å være data, men sier hvilken rekkefølge elementene skal annihileres. Like tall kan annihileres samtidig. Strekene viser hvilke to linjer i A og b som blir påvirket av annihileringen. Elementene med verdi x og b er i dette eksemplet ment å være tilfeldige verdier. Disse verdiene vil endre seg under oppdateringen. x vektoren vil ikke endre verdi under

[3]Dette eksemplet bruker en 6x3 matrise for at eksempelet ikke skal ta for stor plass. Normal størrelse er en nxn matrise.

QR-dekomponeringen.

Figur 2.6 viser steg for steg hvilke elementer som blir annihilert. Her indikerer $\Rightarrow$ neste steg. Elementene annihileres etter hvilken plass de har. Plass 1 annihileres først og plass 7 sist. Når en plass har blitt annihilert, markeres plassen med en 0. I det første steget ser vi at plass 1 skal annihileres. Når vi kommer til steg 2, skal plass 2 annihileres. I tillegg er det innført en 0 på plass 1, som sier at denne er annihilert. I steg 3 er det to 3 tall. Begge disse plassene skal annihileres i dette steget. Slik fortsetter annihileringen frem til steg 7.

$$
\begin{array}{ccc} \text{A} & \text{x} & \text{b} \end{array}
$$

$$
\left[\begin{array}{ccc} x & x & x \\ 5 & x & x \\ 4 & 6 & x \\ 3 & 5 & 7 \\ \hline 2 & 4 & 6 \\ \mathbf{1} & 3 & 5 \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} b \\ b \\ b \\ b \\ \hline b \\ b \end{array}\right]
\Rightarrow
\left[\begin{array}{ccc} x & x & x \\ 5 & x & x \\ 4 & 6 & x \\ \hline 3 & 5 & 7 \\ \mathbf{2} & 4 & 6 \\ \hline 0 & 3 & 5 \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} b \\ b \\ b \\ \hline b \\ b \\ \hline b \end{array}\right]
\Rightarrow
$$

$$
\left[\begin{array}{ccc} x & x & x \\ 5 & x & x \\ \hline 4 & 6 & x \\ \mathbf{3} & 5 & 7 \\ \hline 0 & 4 & 6 \\ 0 & \mathbf{3} & 5 \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} b \\ b \\ b \\ b \\ \hline b \\ b \end{array}\right]
\Rightarrow
\left[\begin{array}{ccc} x & x & x \\ \hline 5 & x & x \\ \mathbf{4} & 6 & x \\ \hline 0 & 5 & 7 \\ 0 & \mathbf{4} & 6 \\ 0 & 0 & 5 \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} b \\ b \\ b \\ b \\ \hline b \\ b \end{array}\right]
\Rightarrow
$$

$$
\left[\begin{array}{ccc} x & x & x \\ \mathbf{5} & x & x \\ \hline 0 & 6 & x \\ 0 & \mathbf{5} & 7 \\ \hline 0 & 0 & 6 \\ 0 & 0 & \mathbf{5} \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} b \\ b \\ b \\ b \\ b \\ \hline b \end{array}\right]
\Rightarrow
\left[\begin{array}{ccc} x & x & x \\ 0 & x & x \\ 0 & \mathbf{6} & x \\ \hline 0 & 0 & 7 \\ 0 & 0 & \mathbf{6} \\ 0 & 0 & 0 \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} b \\ b \\ b \\ b \\ b \\ \hline b \end{array}\right]
\Rightarrow
$$

$$
\begin{array}{ccc} & \text{R} & \text{x} & \text{B} \end{array}
$$

$$
\left[\begin{array}{ccc} x & x & x \\ 0 & x & x \\ 0 & 0 & x \\ 0 & 0 & \mathbf{7} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} b \\ b \\ b \\ b \\ b \\ b \end{array}\right]
\Rightarrow
\left[\begin{array}{ccc} x & x & x \\ 0 & x & x \\ 0 & 0 & x \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]
\left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right]
=
\left[\begin{array}{c} B \\ B \\ B \\ B \\ B \\ B \end{array}\right]
$$

Figur 2.6: Fra A og b til R og B

Annihileringsrekkefølgen som er vist i Figur 2.6 er den som brukes i denne oppgaven. En annen rekkefølge blir beskrevet i [21] og bruker like mange etapper. Begge rekkefølgene er like gode, men jeg valgte den i Figur 2.6.

For hver annihilering produserer Givens, fast Givens eller CORDIC en Q_x^T. Denne multipliseres med A matrisen og b vektoren. Eksempel 1 og 2 beskriver nærmere hvordan annihileringen foregår. Eksempel 2 viser også hvordan flere elementer kan annihileres i parallell.

Eks 1: Annihilering av elementet på plass 1 og 2
La oss først se på plass 1. Her er det kun et ett tall. Det vil si at det kun er dette elementet som kan annihileres i denne omgangen. Det er kun elementet på plass 1 som blir annihilert, men annihileringen oppdaterer hele linje 5 og 6 med nye verdier. Her blir Q_1^T produsert og multiplisert med linje 5 og 6 i A matrisen og b vektoren, se Figur 2.7. Som følge av at A blir multiplisert med Q_1^T, blir plass 1 i A matrisen annihilert.

$$Q_1^T \cdot Ax = Q_1^T \cdot b \qquad \Rightarrow \qquad Ax = b$$

$$Q_1^T \cdot \left[\begin{array}{ccc} x & x & x \\ 5 & x & x \\ 4 & 6 & x \\ 3 & 5 & 7 \\ \hline 2 & 4 & 6 \\ \mathbf{1} & 3 & 5 \end{array}\right] \begin{bmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = Q_1^T \cdot \left[\begin{array}{c} b \\ b \\ b \\ b \\ \hline b \\ b \end{array}\right] \Rightarrow \left[\begin{array}{ccc} x & x & x \\ 5 & x & x \\ 4 & 6 & x \\ 3 & 5 & 7 \\ 2 & 4 & 6 \\ 0 & 3 & 5 \end{array}\right] \begin{bmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} b \\ b \\ b \\ b \\ b \\ b \end{bmatrix}$$

Figur 2.7: Annihilering av plass 1

Plass 2 produserer Q_2^T som påvirker linje 4 og 5. Det er ikke mulig å starte annihileringen av plass 2 før plass 1 er ferdig. Grunnen til dette er at begge annihileringene oppdaterer linje 5. Annihilering av plass 2 kan starte når plass 1 er ferdig med å oppdatere linje 5.

Eks 2: Annihilering av elementene på plass 5
Forhåndsbetingelsen til dette eksempelet er at plassene 1, 2, 3 og 4 er ferdig annihilert. Det fins tre 5 tall i matrisen, som vil si at alle disse tre kan annihileres samtidig uten at de påvirker hverandre. Det øverste 5 tallet påvirker linje 1 og 2, det neste påvirker linje 3 og 4 og det siste påvirker linje 5 og 6. Her kan Q_7^T, Q_8^T og Q_9^T regnes ut samtidig. Når de er ferdig utregnet blir de ganget med A matrisen og b vektoren samtidig, se Figur 2.8.

$$Q_9^T \cdot Q_8^T \cdot Q_7^T \cdot Ax = Q_9^T \cdot Q_8^T \cdot Q_7^T \cdot b \qquad \Rightarrow \qquad Ax = b$$

$$\begin{matrix} Q_7^T \cdot \\ Q_8^T \cdot \\ Q_9^T \cdot \end{matrix} \left[\begin{array}{ccc} x & x & x \\ \mathbf{5} & x & x \\ \hline 0 & 6 & x \\ 0 & \mathbf{5} & 7 \\ \hline 0 & 0 & 6 \\ 0 & 0 & \mathbf{5} \end{array}\right] \left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right] = \begin{matrix} Q_7^T \cdot \\ Q_8^T \cdot \\ Q_9^T \cdot \end{matrix} \left[\begin{array}{c} b \\ b \\ \hline b \\ b \\ \hline b \\ b \end{array}\right] \Rightarrow \left[\begin{array}{ccc} x & x & x \\ 0 & x & x \\ 0 & 6 & x \\ 0 & 0 & 7 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{array}\right] \left[\begin{array}{c} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{array}\right] = \left[\begin{array}{c} b \\ b \\ b \\ b \\ b \\ b \end{array}\right]$$

Figur 2.8: Annihilering av plass 5

2.1.7 Tilbakesubstitusjon

Som tidligere nevnt er det mulig å bruke tilbakesubstitusjon på R og B for å finne startverdien x. Tilbakesubstitusjon brukes til å løse ligningssystemet

$$Rx = B \tag{2.20}$$

hvor R og B er kjente og mens er x en ukjent vektor. Under er et eksempel på hvordan man løser ett slikt ligningssystem vha tilbakesubstitusjon.

$$\left[\begin{array}{ccc} 1 & 4 & 2 \\ 0 & 3 & \text{-1} \\ 0 & 0 & 1 \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array}\right] = \left[\begin{array}{c} 3 \\ -5 \\ 2 \end{array}\right]$$

Figur 2.9: Ligningssystemet

Siden R i Figur 2.9 er øvre rektangulær, så er det mulig å løse dette ligningssystemet. Dette gjøres ved å gange R matrisen med x vektoren og deretter sette dette lik B vektoren. Den første ligningen blir da slik.

$$\begin{aligned} 0x_1 + 0x_2 + 1x_3 &= 2 \\ x_3 &= 2 \end{aligned}$$

Siden R er øvre triangulær, er det bare x_3 som er ukjent i dette ligningssystemet. Neste ligning blir å løse x_2.

$$\begin{aligned} 0x_1 + 3x_2 - 1x_3 &= -5 \\ 3x_2 &= x_3 - 5 \\ 3x_2 &= 2 - 5 \\ x_2 &= -3/3 \\ x_2 &= -1 \end{aligned}$$

Ved å sette inn x_3 i ligningen, er det kun x_2 som er ukjent denne gangen.

$$\begin{aligned} 1x_1 + 4x_2 + 2x_3 &= 3 \\ x_1 &= 3 - 4x_2 - 2x_3 \\ x_1 &= 3 + 4 - 4 \\ x_1 &= 3 \end{aligned}$$

Nå er hele ligningssystemet løst ved hjelp av tilbakesubstitusjon.

$$x = \begin{bmatrix} 3 \\ -1 \\ 2 \end{bmatrix}$$

Dette er prinsippet for hvordan tilbakesubstitusjonen brukes for å finne startverdien x. For å gi en enkel presentasjon av tilbakesubstitusjonen ble det brukt reelle tall i dette eksemplet. Men denne metoden fungerer også for komplekse tall. Den eneste forskjellen er at beregningene blir tyngre, siden den da vil bruke komplekse multiplikasjoner og divisjoner (se avsnitt **2.2.1**).

2.2 Komplekse tall og matriser

A matrisen og b vektoren som skal brukes for å finne startverdien x er komplekse matriser/vektorer. Komplekse matriser er ikke annet enn at elementene i matrisene inneholder komplekse tall. Figur 2.10 viser et eksempel på en 2x2 kompleks matrise. Noen ganger er behov for å ta kvadratroten av et negativt tall, noe som ikke er mulig med det reelle tallsystemet. For eksempel fins det ikke noe reelt tall som kan representere x slik at $x^2 = -1$ [16]. Det ble da konstruert et tallsystem som tok høyde for dette. Tallsystemet bestod av en reell del og en imaginær del. Et komplekst tall kan skrives på denne måten $z = a + bi$, hvor $i^2 = -1$.

$$Z = \begin{bmatrix} 1 + 2i & 2 - i \\ -1 + i & 3 + 0.5i \end{bmatrix}$$

Figur 2.10: Kompleks matrise

2.2.1 Komplekse operasjoner

Denne utvidelsen av tallsystemet fører til at det er andre regneregler som gjelder. Regnereglene er ikke spesielt vanskelige, men de fører ofte til mye større regneoperasjoner. Eksemplene under viser hvor krevende komplekse operasjoner som multiplikasjon og divisjon kan være. Multiplikasjonene kan deles inn i to deler, halvkomplekse og komplekse. En halvkompleks er en multiplikasjon mellom et reelt tall og et komplekst tall, mens en kompleks er en multiplikasjon med to komplekse tall.

$$\begin{aligned} a &= 2 \\ b &= 2 + 2i \\ c &= 3 + 1i \end{aligned}$$

Figur 2.11: Reelle og komplekse tall

Variablene som brukes i eksemplene, vises i Figur 2.11. a er et reellt tall, mens b og c er komplekse.

$$\begin{aligned} d &= a \cdot c \\ &= 2 \cdot (3 + 1i) \\ &= (2 \cdot 3) + (2 \cdot 1i) \end{aligned}$$

Figur 2.12: Eksempel på en halvkompleks multiplikasjon

Først ser vi på den halvkomplekse multiplikasjonen mellom a og c, som vises i Figur 2.12. Her kan vi telle hvor mange operatorer som trengs for å utføre en operasjonen. Som vi ser trengs det 2 reelle multiplikasjoner, det er dobbelt så mye som en vanlig multiplikasjon.

$$\begin{aligned} f &= b \cdot c \\ &= (2 + 2i) \cdot (3 + 1i) \\ &= 2 \cdot 3 + 2 \cdot 1i + 2i \cdot 3 + 2i \cdot 1i \\ &= (2 \cdot 3 - 2 \cdot 1) + (2 \cdot 1i + 2i \cdot 3) \end{aligned}$$

Figur 2.13: Eksempel på en kompleks multiplikasjon

Den komplekse multiplikasjonen vises i Figur 2.13 og er enda mer krevende enn den halvkomplekse. Denne trenger 4 multiplikasjoner, en addisjon

og en subtraksjon, som er 6 ganger så mange operasjoner som en reell multiplikasjon.

$$\begin{aligned} h &= \frac{1}{c} \\ &= \frac{1}{(3+1i)} \\ &= \frac{1\cdot(3-1i)}{(3+1i)\cdot(3-1i)} \\ &= \frac{3-1i}{3\cdot 3-3\cdot 1i+1i\cdot 3-1i\cdot 1i} \\ &= \frac{3-1i}{3\cdot 3+1\cdot 1} \\ &= \frac{3}{3\cdot 3+1\cdot 1}-\frac{1i}{3\cdot 3+1\cdot 1} \end{aligned}$$

Figur 2.14: Eksempel på en halvkompleks divisjon

I Figur 2.14 ser vi at en halvkompleks divisjon av typen $1/x$. Siden flere av operasjonen er like, trenger denne 2 divisjoner, 2 multiplikasjoner og en addisjon.

$$\begin{aligned} g &= \frac{b}{c} \\ &= \frac{(2+2i)}{(3+1i)} \\ &= \frac{(2+2i)\cdot(3-1i)}{(3+1i)\cdot(3-1i)} \\ &= \frac{2\cdot 3-2\cdot 1i+2i\cdot 3-2i\cdot 1i}{3\cdot 3-3\cdot 1i+1i\cdot 3-1i\cdot 1i} \\ &= \frac{(2\cdot 3-2\cdot 1)+(2\cdot 1i+2i\cdot 3)}{3\cdot 3+1\cdot 1} \\ &= \frac{(2\cdot 3-2\cdot 1)}{3\cdot 3+1\cdot 1}+\frac{(2\cdot 1i+2i\cdot 3)}{3\cdot 3+1\cdot 1} \end{aligned}$$

Figur 2.15: Eksempel på en kompleks divisjon

Den mest krevende av operasjonene er kompleks divisjon og vises i Figur 2.15. I likhet med den halvkomplekse divisjonen er også her flere av

operasjonen like. Denne trenger derfor 6 multiplikasjoner, 2 divisjoner, 2 addisjoner og en subtraksjon.

Eksemplene over viser hvor krevende det er å utføre komplekse operasjoner i forhold til reelle. Dette fører til at det er mye tyngre å invertere en kompleks matrise enn en reell matrise. Den vil både kreve mer maskinvare og bruke lenger tid. Det er derfor være viktig å begrense bruken av de tyngste komplekse operatorene, slik som for eksempel kompleks divisjon.

2.3 Koordinatsystem

Til slutt vil jeg si litt om koordinatsystemet som blir brukt for å indeksere matrisene. Figur 2.16 viser hvordan koordinatsystemet er bygget opp. Indekseringen vil se ut som dette A[y][x], hvor x og y er koordinaten i matrisen. Elementet øverst til venstre er satt til å være origo, det vil si element a_{00} som vil indekseres med A[0][0]. Når x øker, flytter vi oss mot høyre og når y øker flytter vi oss nedover. Det vil si at element a_{21} indekseres med A[2][1]. Algoritmene som blir beskrevet senere i denne oppgaven vil indekserer matrisen på denne måten.

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \dots & a_{1n} \\ a_{20} & a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n0} & a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

Figur 2.16: Koordinatsystem for matrisene

2.4 Tallformat

I den nye taktiske radiolinjen til KDC brukes et *fixed-point* tallformat. Derfor bruker også jeg dette formatet i min oppgave. I *fixed-point* blir komma satt til en bestemt plass i bitvektoren. Jeg bruker *fixed-point* med *signed magnitude*, som vil si at første bit bestemmer om tallet er positivt eller negativt. Her angir 0 at tallet er positivt og 1 at tallet er negativt.

2.5 Problemstilling

KDC skal bruke en LSM utjevner for å låse seg til det inverse kanalfilteret. Hvis tilnærmingen av det inverse kanalfilteret gjøres på bakgrunn av signalene den får over tid, vil tilnærmingen ta lang tid. Radiolinjen skal bruke frekvenshopping for å beskytte seg mot støy og jamming. Hver gang radioen gjør et frekvenshopp eller dataene blir ødelagt av en støypuls (jammet), må LMS-utjevneren reinnstille seg. Dette gjør at tilnærming over tid ikke er noen gunstig løsning.

For at radiolinjen skal bli effektiv, må reinnstillingen være rask. Den raskeste måten å reinnstille LMS-utjevneren på, er å gi den en startverdi på bakgrunn av treningssekvensen i rammene. Denne finnes ved hjelp av matriseinvertering, som kan være en tung og tidkrevende prosess. KDC ønsket derfor en metode som kan invertere en 25x25 matrise på under $100\mu s$.

Kapittel 3

Rotasjonsmetoder

Dette kapitlet beskriver de 3 rotasjonsmetodene som var aktuelle for å utføre QR-dekomponeringen og hvilken av de som ble valgt. Metodene det står mellom er Givens, fast Givens og CORDIC. For å velge metode blir det sett på hvor mange operatorer de bruker og hvor store tall de genererer i matrisen.

For å kunne teste ut metodene på et tidlig stadium, laget jeg en oppførselsmodell i C++ som kunne utføre QR-dekomponeringen med alle 3 rotasjonsmetodene. I tillegg til at den kunne finne x vektoren ved hjelp av tilbakesubstitusjon. Denne modellen ble kalt SoftQR og er nærmere beskrevet i Kapittel **6.1**.

Givens, fast Givens og CORDIC utfører det samme. Av de tre så er det bare Givens som egentlig produserer Q^T matrisen. Fast Givens lager en lignende matrise som kalles F(fast) matrisen og CORDIC produserer ikke noen Q^T matrise i det hele tatt. Algoritmen til CORDIC gjør at den kan brukes til å lage R matrisen , selv om den ikke lager Q^T matrisen. Dette blir forklart nærmere i seksjonen som omtaler CORDIC.

3.1 Startverdier for rotasjonsmetodene

Alle tre rotasjonsmetodene henter startverdiene fra A matrisen. De henter de samme verdiene, men for å skille dem bruker jeg forskjellige navn. Startverdiene til Givens kalles f og g, fast Givens x_1 og x_2 og CORDIC X_0 og Y_0. Figur 3.1 viser hvor verdiene hentes ut når element på plass 1(se Figur 2.7) skal annihileres.

Givens

$$\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \\ f & x & x \\ g & x & x \end{bmatrix}$$

Fast Givens

$$\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \\ x_1 & x & x \\ x_2 & x & x \end{bmatrix}$$

CORDIC

$$\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \\ X_0 & x & x \\ Y_0 & x & x \end{bmatrix}$$

Figur 3.1: Startverdier ved annihilering av plass 1

Elementene g,x_2 og Y_0 settes til elementet som skal annihileres, mens f, x_1 og X_0 settes til elementet rett over. Ved annihilering av plass 5, ville startverdiene vært som i Figur 3.2

Givens

$$\begin{bmatrix} f_1 & x & x \\ g_1 & x & x \\ 0 & f_2 & x \\ 0 & g_2 & x \\ 0 & 0 & f_3 \\ 0 & 0 & g_3 \end{bmatrix}$$

Fast Givens

$$\begin{bmatrix} x_{1-1} & x & x \\ x_{2-1} & x & x \\ 0 & x_{1-2} & x \\ 0 & x_{2-2} & x \\ 0 & 0 & x_{1-3} \\ 0 & 0 & x_{2-3} \end{bmatrix}$$

CORDIC

$$\begin{bmatrix} X_{0-1} & x & x \\ Y_{0-1} & x & x \\ 0 & X_{0-2} & x \\ 0 & Y_{0-2} & x \\ 0 & 0 & X_{0-3} \\ 0 & 0 & Y_{0-3} \end{bmatrix}$$

Figur 3.2: Startverdier ved annihilering av plass 5

3.2 Givens rotasjon

Givens rotasjon ble utviklet av W. Givens i 1954 [8]. Rotasjonen brukers for å produserer en Q_x^T, som brukes videre for å rotere elementene i A matrisen slik at ett element annihileres. For å finne Q_x^T beregner Givens rotasjon c og s , hvor $c = cos(\theta)$ og $s = sin(\theta)$. θ er vinkelen som elementene i A matrisen må roteres med for å kunne annihilere et bestemt element. Givens rotasjon produserer i utgangspunktet en G matrise, som brukes videre til å lage Q_x^T. G matrisen er en 2x2 matrise, som har følgende form.

$$G = \begin{bmatrix} c & s \\ -conj(s) & c \end{bmatrix}$$

I [3] blir det presentert en måte å regne ut c og s, når startverdiene f og g inneholder komplekse tall. Denne metoden regner ut c og s direkte, uten at den først trenger å finne vinkelen θ. Ligning 3.1 og 3.2 viser denne metoden.

$$c = \frac{|f|}{\sqrt{|f|^2 + |g|^2}} \tag{3.1}$$

$$s = sign(f)\frac{conj(g)}{\sqrt{|f|^2 + |g|^2}} \tag{3.2}$$

Denne fungerer også på reelle tall. For å se at denne virker kan vi bruke den på A matrisen fra avsnitt 2.1.5. La oss anta at vi skal annihilere plass 1. Da får f og g følgende verdier.

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \Rightarrow f = 3, g = 4$$

Ved å sette f og g inn i ligning 3.1 og 3.2 får vi følgende verdier i c og s.

$$c = \frac{|3|}{\sqrt{|3|^2 + |4|^2}} = 0.6$$

$$s = sign(f)\frac{conj(4)}{\sqrt{|3|^2 + |4|^2}} = 0.8$$

Dette gir igjen følgende G. Denne matrisen blir satt inn i Q_x^T slik at c elementene ligger i hoveddiagonalen. Siden f og g ligger i linje 2 og 3, blir G satt inn i linje 2 og 3. De resterende elementene i hoveddiagonalen blir satt til 1.

$$G = \begin{bmatrix} 0.6 & 0.8 \\ -0.8 & 0.6 \end{bmatrix} \Rightarrow Q_1^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.6 & 0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix}$$

Hvis vi nå sammenligner Q_1^T i dette eksemplet med Q_1^T fra avsnitt 2.1.5, ser vi at de er helt like. I avsnitt 2.1.5 ble det vist at denne Q_1^T annihilerer et element i A matrisen. Dermed er det vist hvordan Givens rotasjon annihilerer et element i A matrien.

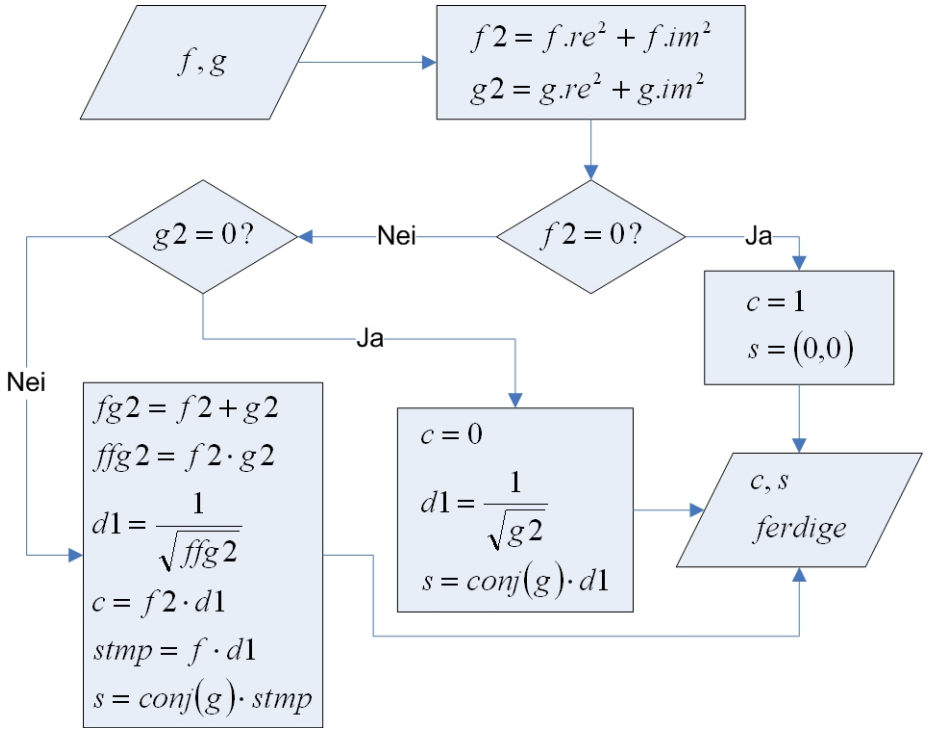
Givens kan deles inn i to deler, beregningsmodus og oppdateringsmodus. Oppgaven til beregningsmodusen er å lage Q_x^T, mens oppgaven til oppdateringsmodusen er å multiplisere Q_x^T med A og b.

3.2.1 Beregningsmodus

Under beregningsmodusen blir c og s produsert, som igjen brukes for å lage Q_x^T. [3] har sett på hvordan Givens kan beregnes effektivt og pålitelig med komplekse tall. Disse algoritmene er en videreutvikling av ligning 3.1 og 3.2. Målet med denne videreutviklingen er å øke presisjonen og hastigheten til Givens når f og g er komplekse tall. Hastigheten er ikke helt relevant i mitt tilfelle, fordi algoritmen er basert på å kjøre på en PC. Jeg vil ta utgangspunkt i algoritmene i denne artikkelen ved evaluering av Givens rotasjon.

Algoritmen i Figur 3.3 er forslaget til [3] for å beregne c og s når f og g er komplekse tall. f og g blir hentet ut av A matrisen og siden A er en kompleks matrise så vil f og g også være det. Men i beregningen av $f2$ og $g2$ blir den reelle og den imaginær delen av f og g multiplisert med seg selv. Dette fører til at den imaginære delen forsvinner og $f2$ og $g2$ blir da reelle.

Algoritmen inneholder 2 spesialtilfeller, det er når enten $f2$ eller $g2$ er lik null. Disse spesialtilfellene må være med for at algoritmen skal gi korrekt svar når enten $f2$ eller $g2$ er lik null. Men stort sett vil både $f2$ og $g2$ være forskjellige fra null, så vi konsentrerer oss om den delen av algoritmen.



Figur 3.3: Flytdiagram over algoritmen til Givens roasjon

Siden $f2$ og $g2$ er reelle tall, blir alle beregninger frem til og med beregningen av c reelle. Dette er positivt, fordi reelle operasjoner krever mindre maskinvare enn komplekse (se avsnitt 2.2.1). Men i beregningen av $stmp$ blir $d1$ multiplisert med f som er kompleks. Dette fører til at $stmp$ blir kompleks og dermed så blir også s kompleks. Av dette kom det frem at c er reell og s er kompleks.

I avsnitt 3.2 så vi at G inneholdt både c og s og at Q_x^T inneholdt G. Q_x^T blir så multiplisert med A og b. I disse multiplikasjonene er det egentlig c og s som blir multiplisert med A og b. Siden c er reell, blir multiplikasjonen mellom c og A eller b halv komplekse multiplikasjoner (se avsnitt 2.2.1). Dette fører til færre beregninger i oppdateringsmodusen til Givens rotasjon.

Ved å telle antall operasjoner i beregningsmodusen til Givens, kom jeg frem til resultatet i Tabell 3.1

Divisjoner	Kvadratrot	Multiplikasjoner	Addisjoner	Subtraksjoner
1	1	12	4	1

Tabell 3.1: Bruk av operasjoner i beregningsmodus til Givens

3.2.2 Oppdateringsmodus

I oppdateringsmodusen blir Q_x^T multiplisert med A og b. Dette fører til at et element i A matrisen blir annihilert.

$$Q_1^T \cdot A \cdot x = Q_1^T \cdot b$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.6 & 0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix} \cdot \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.6 & 0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix}$$

$$\Downarrow$$

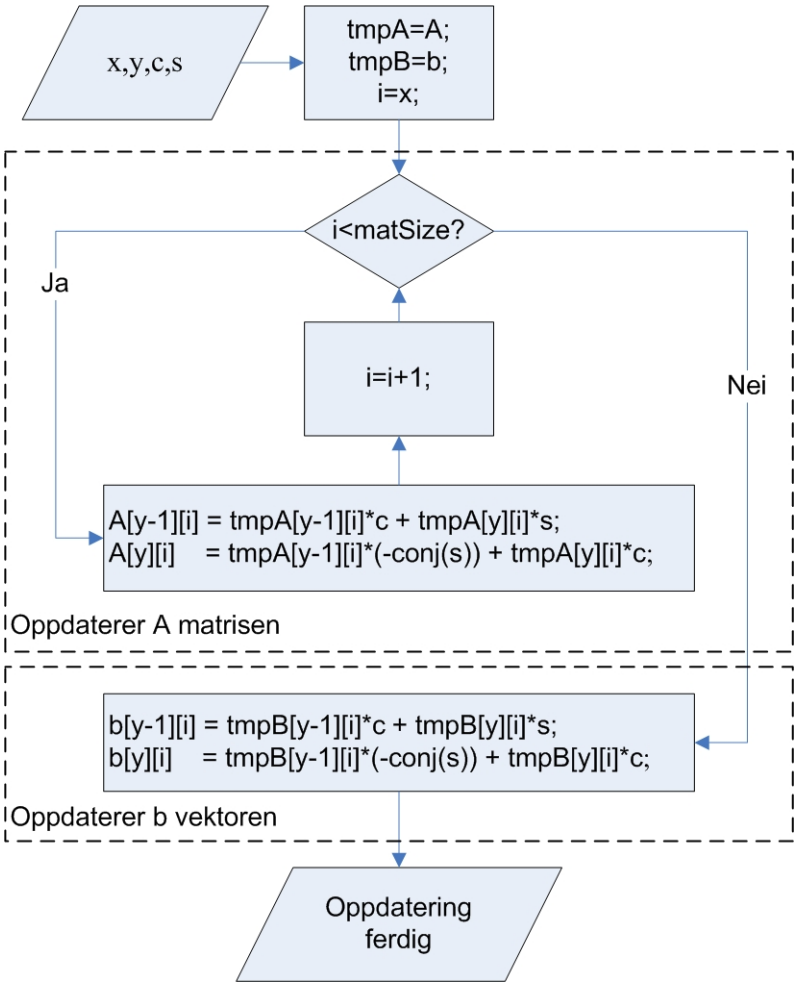
$$\begin{bmatrix} 8 & 1 & 6 \\ 5 & 10.2 & 5.8 \\ 0 & 1.4 & -4.4 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 3.4 \\ -1.2 \end{bmatrix}$$

Hvis vi ser nærmere på matrisemultiplikasjonene over, ser vi at det er kun to av linjene i A og b som ble påvirket av multiplikasjonen. Slik vil det være for alle matrisestørrelser. Siden de andre elementene i matrisen ikke blir påvirket av multiplikasjonen, slipper vi å utføre disse multiplikasjonene. Vi trenger bare å utføre multiplikasjonen som påvirker de to linjene. Hvilke to linjer, er avhengig av hvilket element som skal annihileres (se avsnitt 2.1.6).

Figur 3.4 viser et flytdiagram en algoritmen som multipliserer Q_x^T med A og b. Denne algoritmen utfører kun de multiplikasjonene som trengs. Det vil si at de multiplikasjonene som ikke påvirker elementene i A og b, ikke blir utført. Isteden for å multiplisere Q_x^T med A og b, utfører algoritmen multiplikasjoner direkte med c og s. Multiplikasjonene blir utført på en slik måte at det er ekvivalent med å multiplisere Q_x^T med A og b.

I flytdiagrammet til algoritmen, ser at den har 4 inngangsverdier x, y, c og s. Variablene c og s er resultatet fra beregningsmodusen. x og y er koordinaten til elementet som skal annihileres, altså g. Det første algoritmen gjør er å kopiere A og b til tmpA og tmpB, som brukes i multiplikasjonene. Algoritmen er delt inn i to deler, den første delen inneholder en løkke som utfører alle multiplikasjonene mellom c, s og A matrisen. Denne løkken starter med elementene A[y][x] og A[y-1][x] og fortsetter mot høyre. Den multipliserer to og to elementer med c og s til den når enden av A matrisen. Den andre delen av algoritmen multipliserer c og s med b vektoren. Når denne algoritmen er ferdig er g annihilert og resten av elementene i de to linjene har blitt oppdatert.

I beregningsmodusen kom det frem at c var reell og s kompleks. Alle ele-



Figur 3.4: Flytdiagram over algoritmen for å oppdatere A og b

mentene i A matrisen og b vektoren er komplekse tall og disse multipliseres med c og s. Siden c er reell, vil alle multiplikasjoner mellom elementer i A, b og c være halvkomplekse multiplikasjoner. Mens mult mellom elementer i A, b og s vil være komplekse, se avsnitt 2.2.1 angående halvkomplekse og komplekse mult. Dette fører til at halvparten av multiplikasjonene i oppdateringsmodusen til Givens er halvkomplekse, mens resten er komplekse. Ved å teller antall operasjoner , kom jeg frem til resultatet i Tabell 3.2.

Divisjoner	Kvadratrot	Multiplikasjoner	Addisjoner	Subtraksjoner
0	0	12	10	2

Tabell 3.2: Bruk av operasjoner i oppdateringsmodus til Givens

3.2.3 Fordeler og ulemper

Givens rotasjon har flere gode egenskaper, slik som mulighet for parallellitet og god presisjon. Men Givens har en stor ulempe, nemlig at den inneholder en kvadratrotberegning. Dette er en tung og tidkrevende operasjon som førte til at noen begynte å utvikle en ny rotasjonsmodul. Denne ble kalt fast Givens og kunne utføre Givens rotasjon uten å bruke kvadratrot.

I mitt tilfellet fant jeg ut at ulempen med kvadratrot ikke var så stor, dette fordi KDC tidligere hadde utviklet en god kvadratrot modul. Denne modulen beregner kvadratrøtter ved hjelp av multiplikativ metode. Modulen er liten, rask og effektiv. I tillegg har jeg også spesialtilpasset modulen for å brukes i Givens rotasjon. Dette gjorde den enda mindre og raskere. Dette beskrives nærmere i avsnitt 4.3.

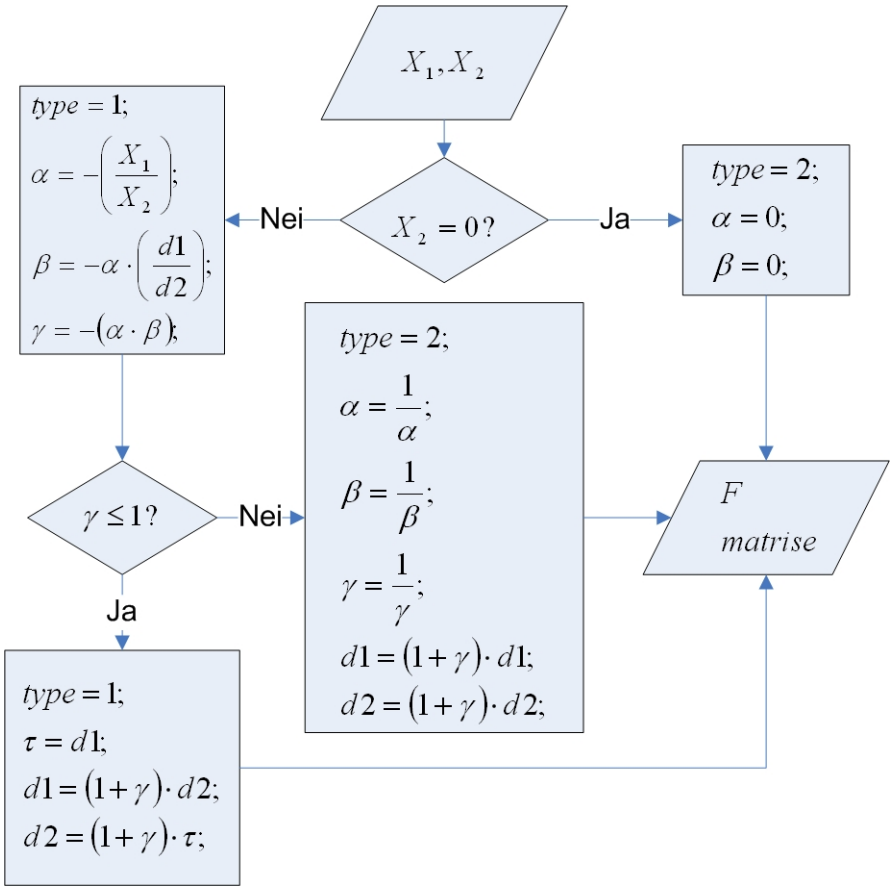
3.3 fast Givens rotasjon

fast Givens ble utviklet W. Morven Gentleman i 1973 [7] og videreutviklet av S.Hammarling i 1974 [13]. Målet med fast Givens er å utelukke kvadratrot beregningen i Givens. Siden fast Givens ble utviklet har det vært uenigheter om Givens eller fast Givens er den beste metoden. [6] stiller spørsmålet om fast Givens faktisk er "'fast"', etter en test i Matlab hvor Givens var raskere. Men siden denne testen baserer seg på at f og g inneholder reelle tall, kunne jeg ikke bruke dette som argument for å ikke bruke fast Givens. Jeg måtte undersøke hvordan fast Givens oppførte seg med komplekse tall og vurdere hvordan metoden egnet seg for implementasjon i FPGA.

3.3.1 Beregningsmodus

I likhet med Givens har også fast Givens både beregningsmodus og oppdateringsmodus. Disse modiene har samme funksjon som modiene til Givens. Det vil si at beregningsmodusen til fast Givens beregner F matrisen. Denne matrisen er ekvivalent med Q_x^T matrisen til Givens. F matrisen inneholder α og β isteden for c og s. Det vil da si at oppgaven til beregningsmodusen er å finne α og β.

Algoritmen i Flytdiagrammet i Figur 3.5 er hentet fra [9]. Denne kan benyttes til å finne α og β. Fast Givens kan lage store tall i A og b, derfor inneholder denne algoritmen variablene $d1$ og $d2$. Oppgaven til disse variablene dempe veksten av tallene i A og b. Startverdien til $d1$ og $d2$ kan velges, men en typisk verdi er 1 [9]. Variabelen $type$ får tildelt verdien 1 eller 2, som bestemmer om F matrisen skal være av type 1 eller 2. Typen bestemmer hvordan F matrisen bygges opp, dette forklares nærmere neste avsnitt. Variabelen γ avgjør hvilken verdi $type$ får.



Figur 3.5: Flytdiagram for fast Givens algoritme

Beregningsmodusen til Givens består av mange reelle operasjoner, selv om inngangsverdiene f og g er komplekse tall. Men i beregningsmodusen til fast Givens er alle operasjoner komplekse når x_1 og x_2 er komplekse. Dette fører til at α og β blir komplekse.

Den lengste veien gjennom algoritmen er når $x_2 \neq 0$ og $\gamma > 1$. Ved å gå denne veien gjennom algoritmen ser vi at den bruker 2 komplekse divisjoner, 3 divisjoner av typen $1/x$ og 4 komplekse multiplikasjoner. Tabell 3.3 viser hvor mange reelle operasjonen dette tilsvarer.

Divisjoner	Kvadratrot	Multiplikasjoner	Addisjoner	Subtraksjoner
10	0	34	16	6

Tabell 3.3: Bruk av operasjoner i beregningsmodus til fast Givens

3.3.2 Oppdateringsmodus

I oppdateringsmodus blir F matrisen multiplisert med A og b. Dette er ekvivalent med å multiplisere Q_x^T med A og b, se avsnitt 2.1.5. Oppbygningen til F matrisen har 2 formater, Type1 og Type2 (se Figur 3.6). Hvilken type som skal brukes er avhengig av verdien til γ, se Figur 3.5.

$$F_1 = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & \beta & \dots & 1 & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & 1 & \dots & \alpha & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \qquad F_2 = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & 1 & \dots & \alpha & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & \beta & \dots & 1 & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix}$$

Figur 3.6: Type1=F_1 og Type2=F_2

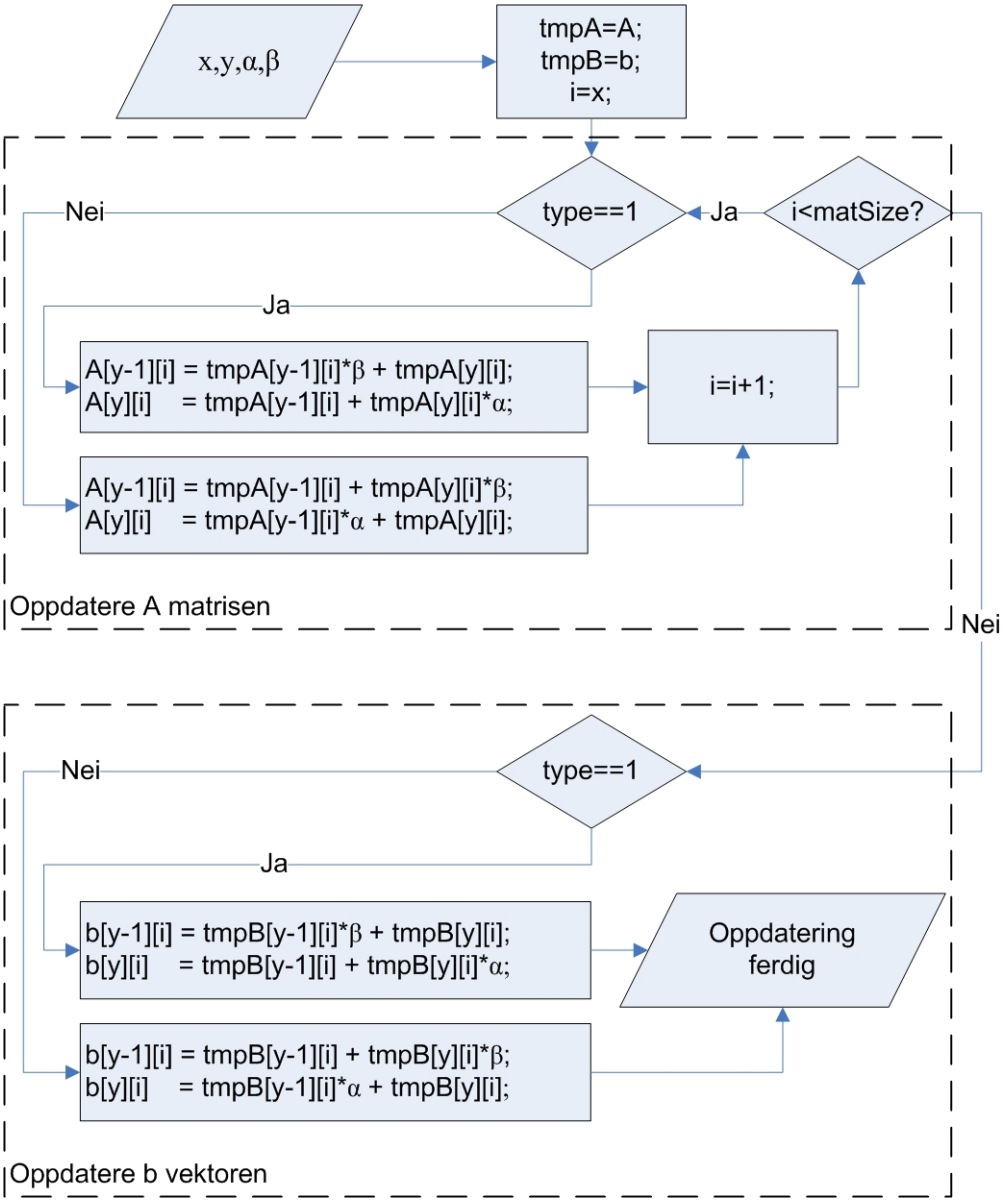
Ved å se på Figur 3.6 ser man at den inneholder færre elementer enn Q_x^T matrisen til Givens. Givens inneholder 2 c'er og 2 s'er, mens fast Givens inneholder 1 α og 1 β. Dette fører til færre multiplikasjoner når F multipliseres med A og b. For A matriser med reelle tall, vil fast Givens ha halvparten så mange multiplikasjoner som Givens i oppdateringsmodusen[1]. For komplekse tall er forskjellen noe mindre.

I likhet med Givens så er det kun 2 linjer i A og b som blir påvirket av multiplikasjonen med F. Det vil si at det heller ikke her er behov for en full matrise multiplikasjon, kun de multiplikasjonen som påvirker de to linjene. Flytdiagrammet i Figur 3.7 viser en algoritme for matriseoppdatering som utfører kun de nødvendige multiplikasjonene. $tmpA$ og $tmpB$ brukes på samme måte som i Givens. Algoritmen har 4 inngangsverdier: x, y, α og β. x og y har samme funksjon som i oppdateringsmodusen til Givens, de gir koordinaten til elementet som skal annihileres, altså x_2. α og β multipliseres med A og b.

Denne algoritmen oppdaterer to og to elementer om gangen. Dette gjør den på akkurat samme måte som Givens. De eneste forskjellene er at fast Givens bruker α og β og at den må sjekke hvilken type F matrise som skal brukes. Ellers er oppdateringsmodusen til fast Givens likt som Givens. Ved å teller antall operasjoner, kom jeg frem til resultatet i Tabell 3.4. Av denne tabellen ser vi at oppdateringsmodusen til fast Givens bruker færre operasjoner enn Givens.

Divisjoner	Kvadratrot	Multiplikasjoner	Addisjoner	Subtraksjoner
0	0	8	6	2

Tabell 3.4: Bruk av operasjoner i oppdateringsmodus til fast Givens



Figur 3.7: Flytdiagram for oppdatering av A og b

3.3.3 Fordeler og ulemper

fast Givens har flere fordeler, en av dem god mulighet for parallellitet. En annen fordel er at algoritmen ikke bruker kvadratrot, som gjør den mer attraktiv for systemer som ikke har mulighet til å utføre denne operasjonen effektivt. En tredje fordel er at den har færre multiplikasjoner i oppdateringsmodusen en Givens. Den sistnevnte fordelen er et godt brukt argument for å velge fast Givens [7, 13, 1, 9] fremfor Givens. Men denne fordelen er ikke like stor for komplekse tall som for reelle.

Fast Givens har også noen ulemper, som større mulighet for generere store tall i A og b enn Givens. Denne ulempen er nevnt i [9], men de kom også tydelig frem i oppførselsmodellen SoftQR (se avsnitt 3.5). På grunn av disse ulempene har [1] utviklet en selvskalerende metode for fast Givens. Men denne algoritmen er større enn den originale algoritmen, som fører til at beregningsdelen av metoden bruker mer maskinvare.

Den største ulempen med fast Givens er at den har behov for mange komplekse operasjoner. For å utføre disse operasjonen trenger fast Givens mange reelle operasjoner. Ved å sammenligne tabell 3.3 med 3.1, ser vi at fast Givens trenger mange flere operasjoner enn Givens. For komplekse tall så har fast Givens 10 ganger så mange divisjoner som Givens.

3.4 CORDIC rotasjon

CORDIC (COordinate Rotation DIgital Computing) ble utviklet av Jack E. Volder i 1959, selv om Henry Biggs i 1624 publiserte lignende teknikker [22]. CORDIC er en iterativ algoritme, som med små modifikasjoner kan få mange bruksområder. Den kan beregne trigonometriske funksjoner som cosinus, sinus, tanngens, polar til rektangulær transformasjon og rektangulær til polar [2]. I tillegg kan CORDIC brukes til å utføre det samme som Givens rotasjon. CORDIC algoritmene beregner disse funksjonene kun ved hjelp av addisjon, subtraksjon og skift operasjoner. Dette gjør at algoritmene bruker lite hardware og er nok grunnen til at CORDIC har blitt så mye brukt og videreutviklet.

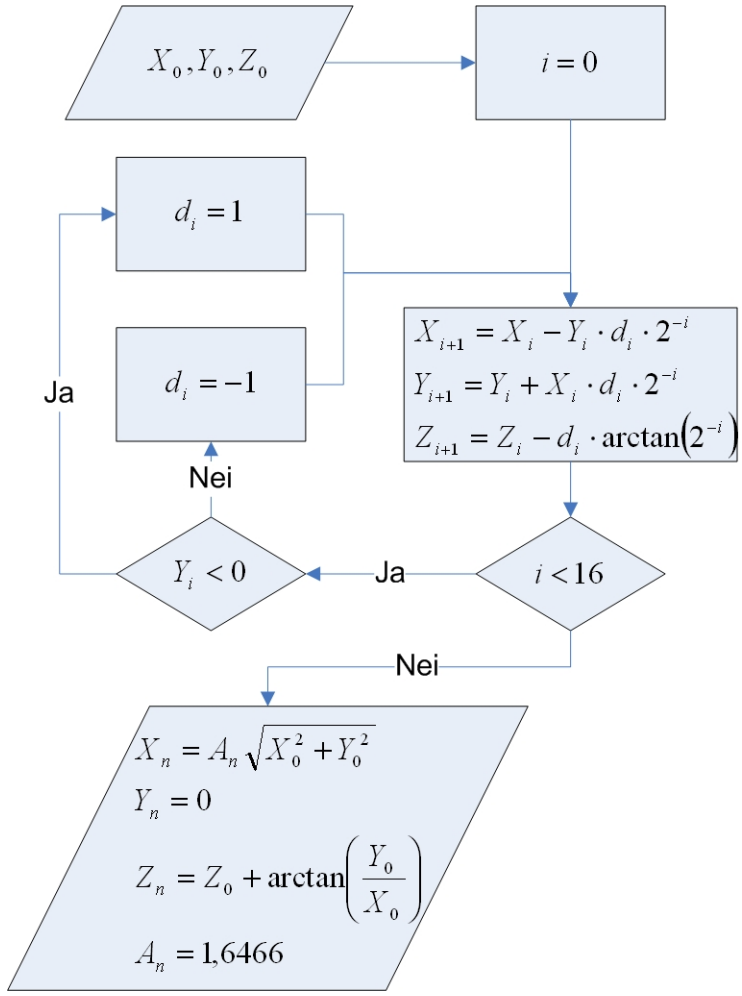
CORDIC algoritmene er iterative og presisjonen til algoritmene er avhengig av antall iterasjoner. Hver iterasjon bruker subtraksjon, addisjon og skift og gir 1 bit økning i presisjon. Det vil si at hvis man skal ha 16-bit presisjon, må man kjøre algoritmen 16 iterasjoner.

Ved bruk av CORDIC for å utføre Givens rotasjon, kan man dele inn CORDIC i to moduser (*vektormodus og rotasjonsmodus*). I vektormodus blir det

bestemte elementet annihilert, pluss at rotasjonsvinkelen θ blir regnet ut. Denne vinkelen brukes videre i rotasjonsmodusen, som oppdaterer de gjenstående elementene i de to linjene i A og b. Algoritmene for å bruke CORDIC til å utføre Givens rotasjon er hentet fra [2].

3.4.1 Vektormodus

Vektormodusen til CORDIC kan sammenlignes med beregningsmodusen til Givens og fast Givens. X_0 og Y_0 er elementer i A matrisen og Y_0 er elementet som skal annihileres. Z_0 er vinkelen θ får startverdien 0 i vektoriseringsmodus. Oppgaven til vektormodusen er å annihilere Y_0, samtidig som den regner ut vinkelen θ og roterer X_0.



Figur 3.8: CORDIC vectormodus

Figur 3.8 viser algoritmen for vektormodusen til CORDIC. Denne algoritmen trenger *arctangens* for å kunne regne ut vinkelen θ, som kunne vært en tung operasjon. Men siden *arctangens* funksjonen kun inneholder kjente verdier ($2^0 \rightarrow 2^{-n}$), kan verdiene regnes ut på forhånd og legges inn i en liten tabell. Antall elementer i tabellen er likt som antall iterasjoner.

I vektormodus er målet til algoritmen å annihilere Y_0. Siden algoritmen er iterativ, vil Y_0 tilnærme seg 0 for hver iterasjon. Denne tilnærmingen fungerer på samme måte som suksessiv approksimasjon. Algoritmen trekker fra en bit fra Y_0 og så sjekke om Y_0 er større eller mindre enn 0. Hvis den fortsatt er større, vil den trekke fra en bit som er halvparten så stor som den forrige biten og gjøre en ny sjekk. Hvis Y_0 blir mindre enn null vil den legge til en bit isteden. Slik fortsetter algoritmen å tilnærme Y_0 til null, samtidig blir X og Z også justert tilsvarende. I denne algoritmen er det variabelen d_i som bestemmer om det skal legges til eller trekkes fra.

Nå erX_n blitt rotert, Y_n er elementet som har blitt annihilert og Z_n er vinkelen θ som Y_0 måtte roteres med for at Y_n skulle bli lik null. Ved å se litt nærmere på resultatet, ser vi at A_n har blitt en del av svaret. A_n har ingenting med A matrisen å gjøre, det er kun en konstant. Denne konstanten er blitt en del av svaret fordi CORDIC algoritmen inneholder en forsterkning. Denne forsterkningen er avhengig av antall iterasjoner og etter ca 16 iterasjoner eller mer vil A_n tilnærme seg 1,6466 [2]. Hvis CIRDIC skal brukes som rotasjonsmodul, må det tas høyde for denne forsterkningen.

I motsetning til Givens og fast Givens så regner ikke CORDIC ut c og s eller α og β. Dette er grunnen til at CORDIC ikke lager Q^T eller F matrisen. Isteden regner den ut rotasjonsvinkelen θ, som den bruker videre i rotasjonsmodusen. Det som utføres i rotasjonsmodusen er ekvivalent med å multiplisere Q^T eller F med A og b.

For at algoritmen skal bli effektiv, må den pipelines. Det vil si å legge alle iterasjonene i serie. Det vil si at hvis vi skal ha 16 bits presisjon, trengs det 16 moduler i serie. Hver modul inneholder da 6 addisjoner og subtraksjoner. Tabell 3.5 viser antall operasjoner, ved bruk av 16 moduler i serie.

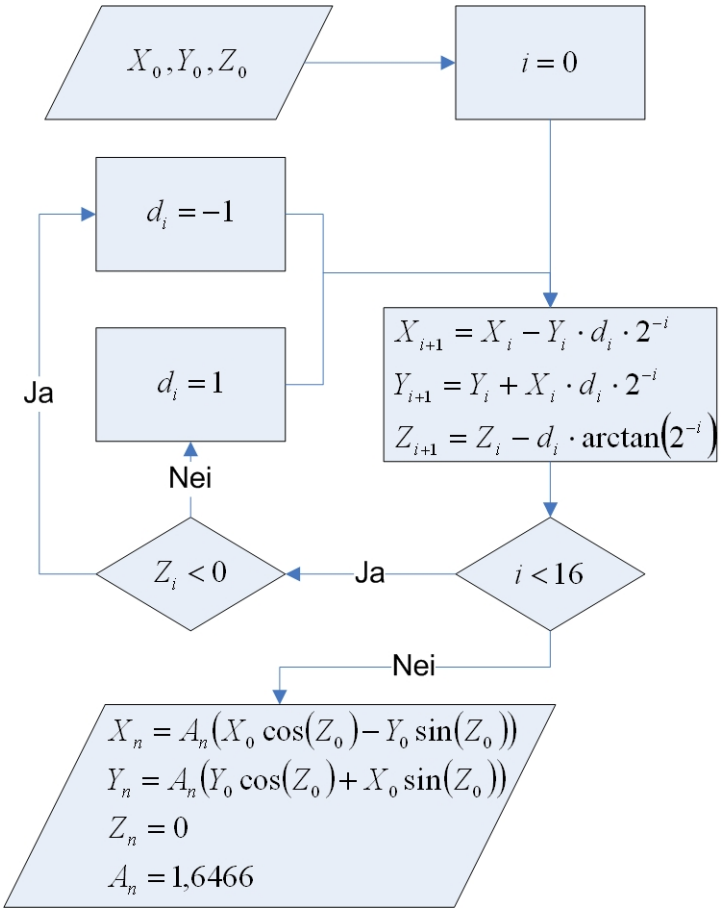
Divisjoner	Kvadratrot	Multiplikasjoner	Addisjoner og Subtraksjoner
0	0	0	96

Tabell 3.5: Bruk av operasjoner i vektormodusen til CORDIC

3.4.2 Rotasjonsmodus

Rotasjonsmodusen utfører det samme som oppdateringsmodusen til Givens og fast Givens, den oppdaterer de gjenstående elementene i de to linjene i A eller b. CORDIC gjør dette ved å hente ut X_0 og Y_0 fra de to linjene A eller b og så rotere dem med vinkelen θ. Z_0 blir i rotasjonsmodus satt til vinkelen θ som ble produsert i vektor modus.

Flytdiagrammet i Figur 3.9 viser algoritmen for rotasjonsmodusen til CORDIC. Denne algoritmen er ganske lik algoritmen for vektormodus. Forskjellen mellom de er at i denne algoritmen er det Z_i som bestemmer om d_i er positiv eller negataiv, mens i vektormodus så er det Y_i som bestemmer. Grunnen er at i denne algoritmen ønsker vi at Z_0 skal bli null, mens i vektormodus ønsker vi at Y_0 skal bli 0.



Figur 3.9: CORDIC roatasjonsmodus

Målet med denne algoritmen er å nulle ut vinkelen Z_0. Samtidig blir X_0 og Y_0 oppdatert. Algoritmen nuller ut Z_0 på samme måte som vektormodusen nuller ut Y_0. Samtidig oppdaterer algoritmen X_0 og Y_0 tilsvarende. Når X_0 og Y_0 er ferdig oppdatert, blir de satt inn i A eller b på samme plassen som de ble hentet ut.

Denne algoritmen oppdaterer 2 elementer i A og b samtidig og må kjøres på alle elementene i de to linjene i A og b. Først elementene til høyre for X_0 og Y_0, som ble brukt i vektormodusen og så elementene til høyre for disse igjen. Slik fortsetter den å oppdatere elementene helt til den kommer til de siste elementene i A matrisen. Helt til slutt oppdaterer den elementene i b vektoren.

I Figur 3.9 ser vi at også rotasjonsmodusen inneholder forsterkningen A_n. Både X_n og Y_n blir påvirket av denne forsterkningen. Denne forsterkningen fører til at tallene i A og b blir større. Elementene i de nederste linjene i A og b, må roteres flere ganger enn de øverste. Dette er fordi det er flere elementer i bunn av matrisen som skal annihileres enn på toppen. Siden elementene i de bunn roteres flere ganger enn de på toppen, blir elementene i bunn av A og b påvirket av forsterkningen A_n flere ganger enn de på toppen. Dette fører til at tallene i A og b blir små på toppen og større jo nærmere de er bunnen. Det er ikke gunstig at tallene i A og b blir store. Dette fører til at de opptar større lagringsplass og at de har bruk for større operatorer.

For at også denne algoritmen skal kunne bli effektiv, må den også pipelines. Dette gjøres på samme måte som i rotasjonsmodusen. Siden algoritmene er så like, vil de bruke like mange operasjoner. Dette vises i tabell 3.6.

Divisjoner	Kvadratrot	Multiplikasjoner	Addisjoner og Subtraksjoner
0	0	0	96

Tabell 3.6: Bruk av operasjoner i rotasjonsmodusen til CORDIC

3.4.3 Fordeler og ulemper

Fordelen til CORDIC er at den ikke bruker noen avanserte operatorer, slik som multiplikasjoner, divisjoner og kvadratrøtter. I tillegg kan den lages slik at den bruker veldig lite maskinvare, men da vil den også bruke lang tid. Grunnen til at CORDIC vil bruke lang tid er at den er iterativ, men dette er det mulig å unngå. La oss anta at vi kun ønsker at algoritmen skal bruke 16 iterasjoner. Isteden for å lage en for løkke som gjentar algoritmen 16

ganger, så er det mulig å koble 16 moduler i serie. Hvis vi i tillegg klokker utgangene til disse modulene, vil kjeden med modulene bli pipelinet. Dette vil igjen øke dataraten, men også bruke 16 ganger så mye maskinvare.

Ulempen med CORDIC er forsterkningen A_n. Denne ble jeg oppmerksom på etter at jeg hadde implementert CORDIC i SoftQR. I starten så det ut som om den ikke hadde stor effekt, men det var fordi jeg først testet den med små matriser (opp til 5x5 matriser). Men effekten av forsterkningen A_n kom tydelig frem når jeg testet CORDIC på 25x25 matriser. Jeg oppdaget da at tallene nederst i A og b var mye større enn de øverste tallene. Elementene som vil gå gjennom flest rotasjoner (for en 25x25 matrise) er $A_{23\ 23}$, $A_{23\ 24}$ og b_{23}. Disse elementene vil gå igjennom 47 rotasjoner hver, som vil si at de vil bli multiplisert med A_n 47 ganger. Dette gir en total forsterkning på $1,6466^{47} \approx 15.123 \cdot 10^9$. Det vil si at denne forsterkningen må håndteres hvis CORDIC skal brukes som rotasjonsmodul.

3.5 Analyse av tallstørrelser

Jeg har evaluert Givens, fast Givens og CORDIC rotasjon for å se hvem av dem som er best egnet til min oppgave. Til hjelp i evalueringen ble metodene testet i SoftQR. De første testene gikk ut på å verifisere at metodene regnet riktig. Måten dette ble testet på står beskrevet i avsnitt 6.1. Denne fasen viste at CORDIC ikke regnet korrekt. Grunnen var at jeg ikke tok høyde for at CORDIC trengte inngangsverdier mellom $-\pi/2$ og $\pi/2$ [2]. Jeg kunne jobbet videre med å få CORDIC rotasjon til å virke, men av andre grunner valgte jeg å ikke fortsette med CORDIC. Disse grunnene beskrives i avsnitt 3.6.

Neste test gikk ut på å sjekke hvor store tallene i algoritmen til Givens og fast Givens ble. Dette ble testet ved å utføre mange QR-dekomponeringer med enten Givens eller fast Givens som rotasjonsmodul. Samtidig ble verdiene i A og b overvåket, for å se hvor store tallene ble. Begge rotasjonsmodulene ble testet med 200.000 QR-dekomponeringer. Matrisene som QR-dekomponeringen ble kjørt på, ble generert i SoftQR. Tallene i A og b er tilfeldige komplekse tall i tallområdet -2 til 2. I denne testen var det Givens som kom best ut.

Input	maxFre	=	6.165087
	maxFim	=	5.783945
	maxGre	=	9.793876
	maxGim	=	9.848924
Mellomregninger	maxF2	=	38.598272
	maxG2	=	100.000000
	maxFG2	=	101.695168
	maxFFG2	=	1514.248091
	maxD1	=	7609.728633
Output	maxC	=	1.000000
	maxSre	=	1.000000
	maxSim	=	1.000000

Tabell 3.7: Resultat av 5.000.000 QRD med Givens

Givens rotasjon genererte akseptable tallstørrelser i testen med 200.000 QR-dekomponeringer. Derfor kjørte jeg en ekstra test hvor det ble utført 5.000.000 QR-dekomponeringer. Resultatet vises i Tabell 3.7. $maxF$ og $maxG$ viser de største tallene som ble generert i A og b. De andre variablene er direkte lenket til variablene i algoritmen til Givens. Som vi ser så er det $maxFFG2$ og $maxD1$ som inneholder de største tallene, men disse er det mulig å gjøre mindre ved f.eks. å skalere $f2$ og $g2$. Jeg var fornøyd med dette resultatet, spesielt det at utgangsverdiene $maxC$, $maxSre$ og $maxSim$ aldri blir større enn 1. Dette er noe av grunnen til at Givens ikke lager veldig store tall i A og b.

Input	maxF	=	332538867870226 12.000000
	maxG	=	11792502408899366.000000
Output type 1	maxAlpha1	=	18044422506997788.000000
	maxBeta1	=	135.420074
Mellomregning og skalering	maxGamma1	=	17996.270698
	maxTau	=	0.000000
	maxD11	=	723.000000
	maxD21	=	0.000000
Output type 2	maxAlpha2	=	0.999358
	maxBeta2	=	0.999358
Mellomregning og skalering	maxGamma2	=	0.998482
	maxD12	=	0.000000
	maxD22	=	5.325149

Tabell 3.8: Resultat av 200.000 QRD med fast-Givens

Resultatet fra 200.000 QR-dekomponeringer med fast Givens som rotasjonsmodul vises i Tabell 3.8. Alle variablene i fast Givens er komplekse, men for enkelhetsskyld vises kun den største delen (real delen eller den imaginære delen). I likhet med Givens er også her $maxF$ og $maxG$ største verdi i A og b og som vi ser kan disse verdiene bli veldig store. Alle variabler som ender på 1 tilhører Type 1 matrisen, altså F_1 og de som ender på 2 tilhører F_2. Som vi ser så er det beregningen av type 1 som er den kritiske i denne algoritmen. Mest sannsynlig så er det den store$maxAlpha1$ som har ført til de store verdiene i $maxF$ og $maxG$.

3.6 Valg av rotasjonsmetode

For å sammenligne algoritmene har jeg sett på antall operasjoner de forskjellige rotasjonsmodulene trenger. Jeg har implementert matriseinverteringen i C++. Denne modellen ble kalt SoftQR. Dette gjorde at jeg på et tidlig stadium måtte sett meg inn i hvordan algoritmene fungerte. SoftQR ble også brukt til å analysere tallstørrelsene i algoritmene. En siste ting som jeg tok hensyn til er at FPGA'en som brukes (Xilinx Virtex-4 SX35) inneholder 192 18x18 multiplikatorer. KDC har brukt opp ca 70-80 av disse multiplikatorene til andre deler av radiolinjen. Det vil si at jeg har rundt 110 multiplikatorer til rådighet.

Modus	Divisjon	Kvadratrot	Multiplikasjon	Addisjon og Subtraksjon
Beregning	1	1	12	5
Oppdatering	0	0	12	12
Totalt	**1**	**1**	**24**	**17**

Tabell 3.9: Oppsumering av operasjoner i Givens rotasjon

Modus	Divisjon	Kvadratrot	Multiplikasjon	Addisjon og Subtraksjon
Beregning	10	0	34	22
Oppdatering	0	0	8	8
Totalt	**10**	**0**	**42**	**30**

Tabell 3.10: Oppsumering av operasjoner i fast Givens rotasjon

Av tabell 3.9, 3.10 og 3.11 ser vi at Givens rotasjon bruker færrest operasjoner, mens CORDIC bruker flest. fast Givens kommer midt i mellom, men

Modus	Divisjon	Kvadratrot	Multiplikasjon	Addisjon og Subtraksjon
Vektor	0	0	0	96
Rotasjon	0	0	0	96
Totalt	**0**	**0**	**0**	**192**

Tabell 3.11: Oppsumering av operasjoner i CORDIC rotasjon

denne bruker mest divisjoner og multiplikasjoner. Hvis rotasjonsmodulen skulle implementeres i en ASIC (Application Spesific Integrated Circuit), ville sannsynligvis CORDIC gitt minst bruk av maskinvare. Dette fordi addisjoner og subtraksjoner kan lages mye mindre enn multiplikatorer. Men i en FPGA bygges addisjoner og subtraksjoner opp av LUT'er. Siden FPGA'en som brukes har mange ledige multiplikatorer, er det først og fremst ønskelig å utnytte disse. Dette er grunnen til at jeg valgte å ikke fortsette med CORDIC. Utifra ønsket om minst mulig maskinvare, er det mer fornuftig å velge Givens som rotasjonsmodul.

Av de tre rotasjonsmetodene var det Givens som kom best ut av testene i SoftQR. I tillegg hadde de andre metodene ulemper som at de kunne generere store tall i matrisen. Fast Givens har bruk for mange komplekse operatorer, som fører til større bruk av maskinvare og lengre beregningstid. Den eneste ulempen med Givens var at den hadde en kvadratrot operasjon. Men denne kan løses på en effektiv måte ved å bruke spesialtilpasset aritmetikk. Med alt dette tatt i betraktning, var det Givens som ble valgt som rotasjonsmetode.

Kapittel 4

Implementering av rotasjonsenheten

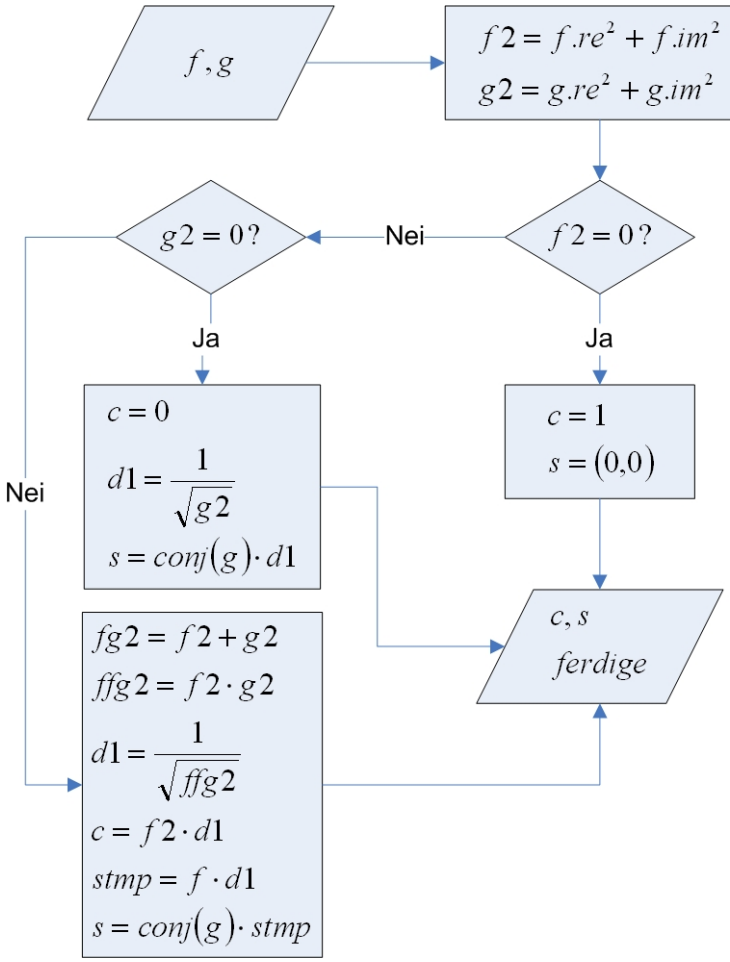
Dette kapitlet beskriver hvordan jeg har implementert beregningsmodusen til Givens i maskinvare. Først og fremst utføres det en grundigere analyse av algoritmen til Givens, for å finne årsaken til at operandene $maxFFG2$ og $maxD1$ ble så store som de ble (se Tabell 3.7). Resultatene i denne tabellen blir brukt sammen med analysen til å finne ut hvor mange bit (ordbredder) variablene i algoritmen trengte. Analysen og resultatene blir også brukt for å finne ut hvilke variabler i algoritmen som må skaleres. Skalering er innført som et tiltak for å begrense behovet for store ordbredder.

Det blir også sett på hvordan ordbreddene påvirker bruk av multiplikatorer i FPGA'en som brukes, Xilinx Virtex4 SX35. I tillegg vises det hvordan spesialoperatoren for kvadratrot minker beregningstiden og maskinvarebruk. Denne spesialaritmetikken tar utgangspunkt i en multiplikativ kvadratrotenhet som KDC tidligere har implementert. Til slutt blir det forklart hvordan parallelliteten til Givens er blitt utnyttet ved hjelp av pipelining.

Jeg har ikke funnet noen detaljert beskrivelse av hvordan Givens kan implementeres i maskinvare. Bortsett fra algoritmen til beregningsmodusen til Givens, er alt som presenteres i dette kapitlet eget arbeid.

4.1 Analyse av Givens rotasjon

Når Givens skulle implementeres i maskinvare, var det viktig å analysere rotasjons algoritmen litt nærmere. Først og fremst for å avdekke hvor store ordbredder algoritmen trenger, men også for å finne hvilke variabler som må skaleres. I [3] er det sett på hvordan Givens kan beregnes effektivt og presist. Flytdiagrammet i Figur 4.1 viser algoritmen som blir presentert i [3]. Dette er algoritmen jeg tok utgangspunkt i når Givens skulle implementeres.



Figur 4.1: Flytdiagram for Givens

Ved å se på flytdiagrammet i Figur 4.1, ser man at tiden det tar å beregne c og s er avhengig av $f2$ og $g2$. Hvis $f2$ eller $g2$ er lik null, vil tiden være kortere enn når $f2$ og $g2$ er forskjellige fra null. Denne algoritmen er i ut-

gangspunket beregnet på å kjøre effektivt på en datamaskin, hvor det bare er positivt at noen av stegene tar kortere tid. Men siden algoritmen skal implementeres i maskinvare, må algoritmen gjøres mer deterministisk. Det vil si at det må ta like lang tid å beregne c og s uavhengig av verdiene til $f2$ og $g2$. Det er to andre grunner for å gjøre Givens deterministisk. For det første gjør det kontrollogikken enklere og for det andre vil det være enklere å pipeline Givens.

Tilfellene hvor $f2$ eller $g2$ er lik null er spesialtilfeller, som ikke inntreffer veldig ofte. Det vil si at den normale veien gjennom algoritmen er når $f2$ og $g2$ er forskjellige fra null. Dette ble da valgt som veien gjennom pipelinen. For å spare maskinvare, er det ønskelig at spesialtilfellene ikke bruker andre operatorer enn de som pipelinen allerede bruker. Derfor må spesialtilfellene integreres inn i pipelinen.

4.1.1 Integrering av spesialtilfellene

Det enkleste tilfellet er når kun $f2$ er lik 0. Det eneste som må beregnes er $f2$ og $g2$. Dette er det mulig å legge inn i pipelinen, uten at det er nødvendig med nye operasjoner. I beregningen av c, blir c satt til 1 hvis $f2$ er lik null. Samme sjekken blir gjort i beregningen av s, hvor s blir satt til null hvis $f2$ er lik null.

Algoritmen for når $g2$ er lik null er noe mer avansert, men til gjengjeld så har den store likhetstrekk den normale veien. Ved å sammenligne den normale veien i algoritmen og denne, ser man at utregningen av $d1$ og s i algoritmene er veldig like. Her trengs det ikke noen nye operasjoner for å regne ut $d1$ og s. Det som er nødvendig å gjøre er å endre operandene, $ffg2$ byttes ut med $g2$ og $stmp$ byttes ut med $d1$. Det eneste som må tas hensyn til, er at i den vanlige algoritmen så er s en full kompleks multiplikasjon. Men i dette spesialtilfellet er s en halvt kompleks multiplikasjon. Dette er egentlig ikke noe problem, fordi $d1$ kan sees på som et komplekst tall hvor den imaginære delen er lik 0. Dermed kan det utføres en full kompleks multiplikasjon, hvor $d1$ blir gjort om til et komplekst tall med den imaginære delen lik 0.

4.1.2 Analyse av variabel størrelser

Ordbredden til variablene i algoritmen er avhengig av hvor store og små tallene kan bli. Store tall krever mange bit før komma og små tall krever mange bit etter komma. I testen av Givens rotasjon kom det frem hvor store tallene i variablene kunne bli, se resultatet i Tabell 3.7. Her kommer det frem at det er variablene $ffg2$ og $d1$ som inneholder de største tallene i algoritmen. For å finne ut hvorfor disse blir større enn de andre variablene, må algoritmen analyseres nærmere. Dette gjøres ved å gå gjennom algoritmen steg for steg. Vi starter da med $f2$ og $g2$.

Resultatet som det henvises til i denne analysen er resultatet i Tabell 3.7.

$$f2 = f.re^2 + f.im^2 \tag{4.1}$$

$$g2 = g.re^2 + g.im^2 \tag{4.2}$$

Resultatet viste at $maxF2 = 38.59$ og $maxG2 = 100$. Dette er ikke spesielt høye verdier, men når begge brukes videre i algoritmen kan de føre til større verdier.

$$fg2 = f2 + g2 \tag{4.3}$$

Resultatet viste også at $maxFG2 = 101.69$. Dette er heller ikke spesielt høy verdi, som kommer av at det kun utføres en addisjon.

$$ffg2 = f2 \cdot fg2 \tag{4.4}$$

Her viste resultatet at $maxFFG2 = 1514.24$. Denne verdien er ca 15 ganger større enn tidligere verdier. Hvis vi skriver om ligningen litt får vi at $ffg2 = f2 \cdot (f2 + g2)$. Her ser vi at $f2$ blir multiplisert med seg selv, pluss at den multipliseres med $g2$. Stor $f2$ vil føre til stor $ffg2$. På en annen side så kan $f2 < 1$ føre til at $ffg2$ blir veldig liten. Som vi ser er denne ligningen veldig avhengig av størrelsen til $f2$.

$$d1 = 1/\sqrt{ffg2} \tag{4.5}$$

Resultatet viste at $maxD1 = 7609.72$, som er den største verdien i algoritmen. Denne ligningen har $ffg2$ som eneste variabel. Det vil si at den er like avhengig av størrelsen til $f2$ som ligning 4.4. Men her er avhengigheten motsatt. Det vil si at en liten $f2$ vil gi en stor $d1$. Og en stor $f2$ fører til en liten $d1$. For å begrense denne størrelsen, kan det gjøres noe med størrelsen til $f2$.

$$c = f2 \cdot d1 \tag{4.6}$$

$$\tag{4.7}$$

I ligning 4.5 kom vi frem til at hvis $f2$ er liten så blir $d1$ stor, og omvendt. Det vil si at de på en måte utjevner hverandre i beregningen av c. Dette er nok grunnen til at resultatet viste at c aldri ble større enn 1.

$$stmp = f \cdot d1 \tag{4.8}$$
$$s = conj(g) \cdot stmp \tag{4.9}$$

Siden $f2$ er avhengig av f, gjelder nesten samme resonnement i ligning 4.8 som i ligning 4.7. Resultatet viste at s aldri var større enn 1. Dermed ingen problemer med store tall i ligning 4.9.

4.1.3 Håndtering av variabel størrelsene

De største variablene i algoritmen er $ffg2$ og $d1$. Ut i fra analysen av variabelstørrelsene kommer det fram at størrelsen til disse er mest avhengig av størrelsen til $f2$. Størrelsen til $g2$ spiller selvfølgelig også en rolle, men ikke så stor som $f2$. For å gjøre $ffg2$ og $d1$ mindre, valgte jeg å skalere $f2$. En stor $f2$ gir stor $ffg2$ og liten $d1$, og omvendt. Det vil si at de er motsatt avhengige av $f2$. Det er da viktig å skalere $f2$ til et område som gir lav verdi i både $ffg2$ og $d1$.

Men når $f2$ skaleres så må $g2$ også skaleres med samme faktor. Ellers vil $fg2$ få feil verdi i når $f2$ og $g2$ legges sammen i ligning 4.3. For å finne ut hvilket tallområde $f2$ og $g2$ gir minst verdier i $ffg2$ og $d1$, satt jeg opp en beregning i et Excel ark. Tabell 4.1 viser hvilke verdier $f2$ og $g2$ gir $ffg2$ og $d1$.

$f2$	$g2$	$fg2$	$ffg2$	$d1$
0,5	0,5	1	0,5	1,414213562
1	1	2	2	0,707106781
2	2	4	8	0,353553391
4	4	8	32	0,176776695
8	8	16	128	0,088388348
16	16	32	512	0,044194174

Tabell 4.1: Skaleringstabell for $ffg2$ og $d1$

Denne tabellen tar utgangspunkt i at $f2$ og $g2$ blir skalert ned til et gitt tallområde[1]. Ved å skalere til 0,5 eller 1, blir både $ffg2$ og $d1$ små. Men jeg fant

[1] $f2$ og $g2$ består av 27 bit og har 7 bit før og 20 bit etter komma (se avsnitt 4.1.3). Det er denne kommaplasseringen jeg bruker som referanse når jeg skalerer.
Eksempel: Hvis $f2$ og $g2$ skal skaleres til 8, blir det første bitet til den største av dem satt til bit 4 før komma.

ut at det var gunstigere å skalere til et noe høyere tallområde. Grunnen til dette er at tabellen antar at både $f2$ og $g2$ blir skalert til samme området. Men i de fleste tilfeller vil de bli skalert til forskjellige områder, siden de har forskjellig størrelse i utgangspunktet og må skaleres med samme faktor. Her vil den største av de bli skalert ned til tallområde og den minste må følge etter. Dette kan føre til at den minste av de mister presisjon, fordi den har blitt skalert for mye ned. Jeg valgte da å bruke 4 som tallområdet, som fører til at den minste av $f2$ og $g2$ får 3 bit mer. I tillegg blir ikke $ffg2$ for stor og $d1$ for liten.

De vanskeligste tilfellene er når $f2$ og $g2$ er langt fra hverandre i størrelse. Her er det den største av de som bestemmer hvor mye det skal skaleres. Det vil si, hvis $f2$ er størst vil denne bli skalert til 4. $g2$ vil bli skalert med samme faktor og kan miste litt presisjon. I motsattfall kan f2 miste presisjon hvis $g2$ er størst. Hvor mye presisjon som mistes er avhengig av hvor stor størrelsesforskjellen mellom $f2$ og $g2$ er.

Men dette er ikke det eneste tilfellet det er behov for skalering. Algoritmen som utfører kvadratrot beregningen i ligning 4.5, krever at operanden ligger i et tallområde fra $0,5$ til 2. Dette forklares nærmere avsnitt 4.3. Denne skaleringen påvirker beregningen av $stmp$ i ligning 4.8. Dette gjorde at det var behov for å skalere $stmp$ før den brukes videre i ligning 4.9.

Tallområdet som $stmp$ blir skalert til er ikke like kritisk som $f2$ og $g2$. Dette er fordi det kun er beregningen av s som er avhengig av $stmp$. Jeg valgte derfor å skalere $stmp$ til tallområdet 1.

Det brukes en variabel i algoritmen som holder rede på hvor mye de andre variablene har blitt skalert. Dette er viktig, fordi c og s må skaleres tilbake til utgangspunktet når de er ferdig beregnet.

Ordbredder

f og g er direkte lenket til verdiene i A og b. Av resultatet i Tabell 3.7 ser vi at disse to ligger under 10. Det vil si at de kun trenger 4 bit før komma. Elementene i A og b har en oppløsning på 16 bit når de blir laget av treningssekvensen. Og det var ønskelig å beholde denne presisjonen i startverdien x. Siden A og b må gjennom mange beregninger før vi endelig får startverdien x, vil det bli tap av presisjon. For å være på den sikre siden, valgte jeg å bruke 20 bit etter komma. I tillegg trengs det et bit for å bestemme fortegnet, da ender vi opp med å bruke 25 bit til f og g. Det vil da si 25 bit til den reelle delen og 25 bit til den imaginære delen av tallene. f og g har derfor av 50 bit.

Skaleringen i algoritmene førte til at ordbreddene til variablene ble mindre. Variablene endte da opp med følgende ordbredder etter implementasjonen i maskinvare.

$$\begin{aligned} f &= 25bit\ x2\ (Kompleks) \\ g &= 25bit\ x2\ (Kompleks) \\ f2 &= 27bit \\ g2 &= 27bit \\ fg2 &= 27bit \\ ffg2 &= 32bit \\ d1 &= 21bit \\ c &= 21bit \\ stmp &= 22bit\ x2\ (Kompleks) \\ s &= 22bit\ x2\ (Kompleks) \end{aligned}$$

Om skaleringen

For å kunne skalere tallene trengte jeg en metoden som telte hvor mange 0'ere det var før første 1'er, altså en Ledende Null Detektor (LND). Denne beskrives nærmere i [17, 18]. Den har en struktur og størrelse som minner om CLA (Carry Look Ahead) i addisjon og har tidligere blitt implementert i [19]. For å få best mulig hastighet på skaleringene, brukte jeg denne til å finne 0'ere.

4.2 Multiplikatorbruk

Under implementasjonen av beregningsmodus til Givens kom det frem at algoritmen kom til å bruke mange multiplikatorer. Grunnen til dette er at multiplikatorene i Xilinx Virtex4 SX35 i utgangspunktet er laget for 18x18 bit signed multiplikasjoner eller 17x17 bit unsigned. Ordbreddene til variablene i algoritmen er større enn 17 bit unsigned. Det er derfor viktig å forstå hvordan FPGA'en utfører multiplikasjoner med større ordbredder.

De fleste multiplikasjonene som utføres i algoritmen er unsigned multiplikasjoner. Følgende gjelder da for unsigned variabler. For ordbredder som er mindre eller lik 17x17 bit, blir det brukt kun en 18x18 multiplikator. Hvis en av operandene er større en 17 bit, blir det laget en 18x36 multiplikator som bruker 2 18x18 multiplikatorer. Hvis begge er større enn 17 bit, blir det laget en på 36x36 bit. Denne bruker hele 4 18x18 multiplikatorer. Jeg brukte Xilinx System Generator for å prøve å optimalisere multiplikatorene, men resultatet ble det samme.

Variablene består mer enn 18 bit og derfor brukes 36x36 multiplikatorer. Dermed kan alle variablene være på 36 bit. Men siden variablene må skaleres, er det ønskelig å prøve å holde ordbredden så liten som mulig. Dette fordi den LND'en bruker mer maskinvare og lenger beregningstid jo større variablene er.

Det ble kjørt Place-And-Route på VHDL koden i Xilinx ISE 7.1 og denne viste at implementasjonene bruker 47 18x18 multiplikatorer. KDC har valgt å bruke FPGA'en Xilinx Virtex-4 SX35, som inneholder 192 18x18 bit signed multiplikatorer. Dermed er det mulig for Givens å bruke mange multiplikatorer.

4.3 $1/\sqrt{x}$ algoritmen

Beregning av $d1$ i ligning 4.5 krever en operasjon av typen $1/\sqrt{x}$, altså en divisjon og en kvadratrot beregning. Det ikke vanlig at FPGA'er inneholder logikk for divisjon og kvadratrot. I en FPGA blir kvadratrot vanligvis beregnet ved hjelp av en CORDIC algoritme, som er iterativ og krever mange iterasjoner for å få god presisjon. Det er spesielt kvadratrot beregningen som blir brukt som argument for å ikke bruke Givens.

KDC har tidligere implementert raske og effektive divisjons og kvadratrot kretser i VHDL. Disse er implementert av Simen Gimle Hansen i KDC som en del av hans hovedfagsoppgave [14]. En divisjon med 18 bits oppløsning krever 3 klokkeperioder og en kvadratrot med 18 bits oppløsning krever 4. Det vil si at det vil ta 7 klokkeperioder å beregne $1/\sqrt{x}$.

Ved å studere kvadratrot kretsen nærmere, fant jeg ut at det er mulig å gjøre beregningen av $1/\sqrt{x}$ enda raskere. Grunnen til dette er at kvadratroten blir regnet ut ved hjelp av en multiplikative metode, basert på Newton-Raphson. Denne metoden beregner kvadratroten ved hjelp av en oppslagstabell, en iterativ algoritme og multiplikasjoner. Oppslagstabellen inneholder kvadratrotresiplokaler, det vil si $1/\sqrt{x}$ (se ligning 4.10). Presisjonen til disse kvadratrotresiprokalene er avhengig av størrelsen til oppslagstabellen. Fordelen med denne metoden er at presisjonen til kvadratrotresiprokalet blir fordoblet i hver iterasjon i den iterative algoritmen (se ligning 4.11). Til slutt blir kvadratrotresiprokalet multiplisert med x(se ligning 4.12), for å finne kvadratroten (se ligning 4.13).

$$Fra\ oppslagstabell: \quad \frac{1}{\sqrt{x}} \tag{4.10}$$

$$Etter\ en\ iterasjon: \quad \frac{1}{\sqrt{x}} \quad (Presisjon\ x2) \tag{4.11}$$

$$Multiplikasjon\ med\ x: \quad x \cdot \frac{1}{\sqrt{x}} \tag{4.12}$$

$$Resultat: \quad \sqrt{x} \tag{4.13}$$

Ved å kutte ut multiplikasjonen med x, får vi svaret $1/\sqrt{x}$ direkte. Det er derfor ikke lenger behov for å utføre divisjon. Her blir altså divisjonen kuttet ut og kvadratrot beregningen er redusert med en multiplikasjon. Operasjonen vil da bruke 3 klokkeperioder isteden for 7, som er en stor ytelsesforbedring. Ikke bare for denne operasjonen, men også for Givens algoritmen. Dette medfører mindre arealbruk og kortere pipeline.

Til Givens vil en 18 bits kvadratrotberegning være tilstrekkelig. [14] har tidligere implementert en oppførselsmodell av en 18 bits kvadratrotenhet i VHDL basert på Newton-Raphson. Denne har en oppslagstabell med 9 bits presisjon og trenger derfor bare en iterasjon for å få 18 bits presisjon. Jeg brukte denne som utgangspunkt når jeg implementerte $1/\sqrt{x}$ på rtl-nivå (Registry Transfer Level). VHDL koden ble implementert slik at $1/\sqrt{x}$ ble pipelinet.

4.4 Pipelining av Givens

For å oppnå ønsket ytelse måtte beregningsmodusen Givens pipelines. Jeg måtte da velge hvordan VHDL-koden måtte bygges opp for å få det til. Slik jeg så det hadde jeg to alternativer, enten implementere Givens i en stor VHDL fil eller dele den inn i flere filer. Jeg fant ut at det mest hensiktsmessige var å dele inn Givens i moduler fordelt på flere filer, for å så koble sammen alle modulene i en strukturell VHDL fil. Det var to åpenbare fordeler ved å gjøre det på denne måten, koden ble mer oversiktlig og hver modul kunne testes individuelt. Alle hadde klokkede utganger, slik at Givens automatisk fikk en pipelinet struktur når modulene ble koblet sammen. Noen av modulene trengte flere klokkeperioder for å bli ferdig, f.eks tok beregningen av d1 3 klokkeperioder. VHDL koden til denne modulen ble derfor bygget opp som en 3 stegs pipeline.

Algoritmen ble delt inn i moduler som hadde klart definerte oppgaver. Disse modulene bruker ett eller flere steg i pipelinen. Tabel 4.2 viser hvordan pipelinen er bygget opp. Den første linjen forteller hvilken modul det er[2]. Den andre linjen forteller hvilke operasjoner som blir utført i modulen. Den tredje forteller hvilket steg i pipelinen operasjonene blir utført. For oversikt over algoritmen til givens, se Figur 4.1.

[2]Modul 5 og 6 er slått sammen. Med dette menes at begge modulene starter sine beregninger i pipelinesteg 8. Disse påvirker ikke hverandre og kan derfor utføres i parallell.

Modul:	1		2	3	4			5 , 6		7	
Operasjon:	$f2$,$g2$		$fg2$	$ffg2$	$d1$			c og $stmp$		s	
Steg:	1	2	3	4	5	6	7	8	9	10	11

Tabell 4.2: Pipeline steg

Som vi ser så består pipelinen kun av 11 steg, dvs at det tar 11 klokkepulser å beregne c og s. Den mest tidkrevende operasjonen er å beregne $d1$, men denne operasjonen ville brukt 4 klokkepulser ekstra hvis det ikke var for spesialtilpasningen av kvadratrotoperasjonen. Uten spesialtilpasningen av denne operatoren ville Givens fått en 15 stegs pipeline. Dette reduserer antall pipelinesteg med 27%.

Modul 1: Steg 1 og 2
Oppgaven til denne blokken er å finne $f2$ og $g2$. I **steg 1** beregner modulen $f2$ og $g2$, mens **steg 2** har som oppgave å skalere $f2$ og $g2$.

Modul 2: Steg 3
Oppgaven til dette steget er å beregne $fg2$, som kun er en addisjon.

Modul 3: Steg 4
Oppgaven til denne er å beregne $ffg2$, som er en multiplikasjon av $f2$ og $fg2$.

Modul 4: Steg 5,6 og 7
Denne inneholder spesialoperatoren for kvadratrot som beregner $d1$. I **steg 5** normaliseres operanden til et tallområde [0.5,2) og kvadratrotresiprokalet blir hentet ut fra oppslagstabellen. I **steg 6 og 7** utføres multiplikasjoner til beregning av kvadratrotresiprokalet.

Modul 5: Steg 8
Her utføres en reell multiplikasjon som beregner c. I tillegg blir all skalering av c fjernet i dette steget. c er ferdig beregnet etter dette steget og må lagres i **Delay modulen** inntil s er ferdig.

Modul 6: Steg 8 og 9
Her utføres en halvtkompleks multiplikasjon som beregner $stmp$, som utføres i **steg 8**. I **steg 9** utføres en skalering av $stmp$ for å unngå store tall i beregningen av s.

Modul 7: Steg 10 og 11
Denne modulen utfører en kompleks multiplikasjon mellom conj(g) og $stmp$ for å beregne s. En kompleks multiplikasjon består av 4 multiplikasjoner, en

addisjon og en subtraksjon, se avsnitt 2.2.1. I **steg 10** blir de 4 multiplikasjonene utført og i **steg 11** utføres addisjonen og subtraksjonen. I likhet med modul 5 blir skaleringen av s fjernet. c og s er nå klare for å brukes til å oppdatere A og b.

Delay modul: Alle steg
Noen av variablene brukes flere ganger i algoritmen. Et eksempel er $f2$ som først brukes i ligning 4.3, så i ligning 4.4 og til slutt i ligning 4.7. Siden algoritmen blir pipelinet, vil $f2$ endre verdi hver klokkeperiode. Dette gjelder for mange andre variabler også og det er derfor nødvendig med en modul som kun har som formål å lagre midlertidige verdier. For $f2$ signalet får modulen 1 inngang og 2 utganger. Signalet blir lagret 1 klokkeperiode før det blir sendt ut på den første utgangen, dette kan nå brukes i ligning 4.4. Signalet blir lagret ytterligere 3 klokkepulser før det blir sendt ut på utgang nummer 2 og er da klart til å brukes i ligning 4.7. Samme typen oppbygning blir brukt på alle variablene som trenger det. Moduler tar altså inn signaler, lagrer de et gitt antall klokkepulser og sender de ut igjen når algoritmen trenger det.

4.4.1 Effektiviteten til pipelinen

Effektiviteten til pipelinen, har en direkte sammenheng med hvor mange rotasjoner som kan utføres samtidig. Men dette antallet varierer med hvilken plass som skal annihileres. Vi må da finne ut hvor mange rotasjoner som i gjennomsnitt utføres i parallell. For å regne ut dette, trenger vi antall rotasjoner og antall plasser. Dette fant jeg ved å sette opp SoftQR til å telle antall rotasjoner og antall plasser. En 25x25 matrise trenger 300 rotasjoner og har 47 plasser. Ved å dele antall rotasjoner på antall plasser er det mulig å finne ut gjennomsnittet. Ligning 4.14 viser denne utregningen.

$$\frac{300}{47} \approx 6,38 \tag{4.14}$$

Dette vil si at algoritmen i gjennomsnitt fyller opp 6 steg av 11 steg i pipelinen. Sammenlignet med en ikke pipelinet versjon av algoritmen, vil den pipelinede versjonen være tilnærmet 6 ganger raskere. Disse beregningene gjelder for 25x25 matriser.

4.5 Oppsummering

Givens rotasjon ble implementert i VHDL med flere tiltak for å forbedre presisjon og hastighet. Det første tiltaket var å lage en spesialoperator som kun bruker 3 kokkeperioder for å beregne operasjoner av typen $1/\sqrt{x}$. Det

andre var å utnytte parallelliteten til Givens ved å implementere den med en 11 stegs pipeline. Det ble også regnet ut at for en 25x25 matrise øker hastigheten med 6 ganger ved å pipeline algoritmen.

Tiltaket for å forbedre presisjonen, består stort sett av å skalere variablene. Et annet tiltak er bruk av større ordbredder, som fører til bruk av 36x36 multiplikatorer. Det ble kjørt syntese av Givens ved hjelp av Precision, som viste at Givens kunne kjøre på ca 67.5MHz. Men dette er kun syntese, så for å får et mer nøyaktig tall kjørte jeg Place-and-Route i ISE 7.1. Etter Place-and-Route gikk frekvensen ned til 58.3MHz. Dette er raskt nok ta at Givens kan brukes i den nye radiolinjen, hvor målfrekvensen er 53.76MHz.

Kapittel 5

Matriseinverteringen

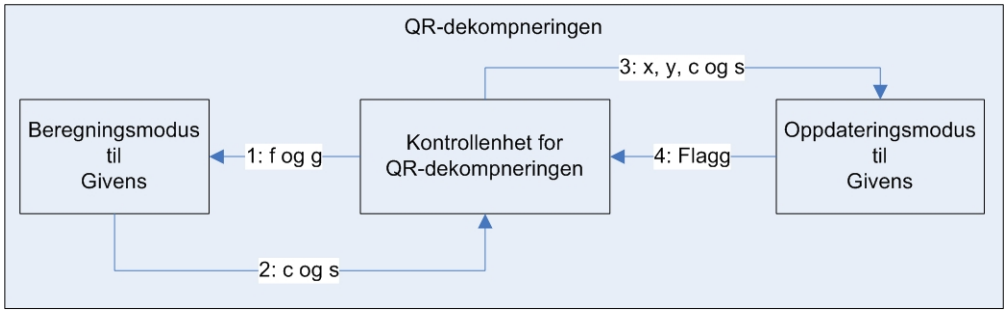
Dette kapitlet beskriver hvordan resten av systemet kan implementeres i maskinvare. Altså resten av QR-dekomponeringen og tilbakesubstitusjonen. Det gis forslag til hvordan disse modulene kan pipelines for å kunne forbedre ytelsen. Kapitlet gir også beskrivelse av hvordan matriseoppdateringen kan implementeres med flere oppdateringsmoduler i parallell, for å minke beregningstiden. Det vises også hvordan det er mulig minke maskinvarebruken til tilbakesubstitusjonen ved bruk av spesialaritmetikk.

I tillegg inneholder kapitlet en estimering av hvor lang tid en matriseinvertering vil bruke. Til slutt blir implementasjonen sammenlignet med andre implementasjoner.

5.1 Algoritme for QR-dekomponering

Algoritmen til QR-dekomponeringen inneholder beregningsmodusen, oppdateringsmodusen og kontrollogikk. Det vil si at den bestemmer hvilket element i matrisen som skal annihileres til enhver tid. Oppgaven til denne algoritmen blir da å velge ut f og g (inngangsverdi til beregningsmodusen) fra A matrisen og sende disse til Givens rotasjon. Givens vil da produsere c og s som igjen sendes til Matriseoppdateringen for å rotere resten av elementene i A og b.

Algoritmen kan deles inn i 3 deler. Beregningsmodusen til Givens, denne delen beregner c og s, se avsnitt 3.2.1. Oppdateringsmodusen til Givens, denne har som oppgave å utføre matriseoppdateringen, se avsnitt 3.2.2. Kontrollogikk for QR-dekomponeringen, denne styrer annihileringsrekkefølgen. Det vil si at denne bestemmer når beregningsmodusen og oppdateringsmodusen til Givens starter. Figur 5.1 viser en oversikt over algoritmen til QR-dekomponeringen. Tallene på pilene indikerer rekkefølgen de utføres på.



Figur 5.1: Oversikt over algoritmen til QR-dekomponeringen

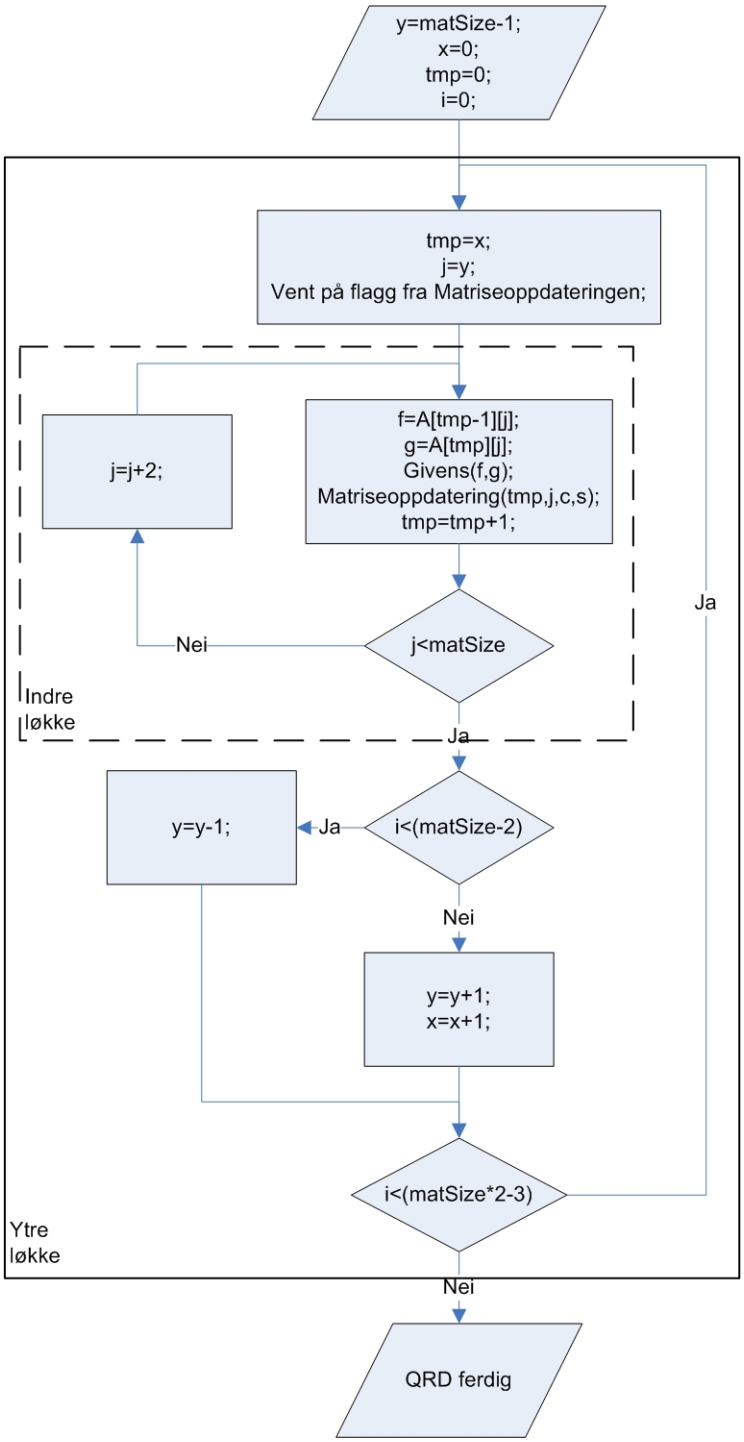
5.1.1 Kontrollogikken

Kontrollogikken bestemmer hvilket element som skal annihileres til enhver tid. Det vil si at denne algoritmen også kontrollerer hvilke elementer som skal annihileres i parallell. Annihileringsrekkefølgen som brukes ble valgt i avsnitt 2.1.6. På bakgrunn av Figur 2.6 lagde jeg en algoritme som styrte rekkefølgen. Figur 5.2 viser et flytdiagram over denne algoritmen

Algoritmen i Figur 5.2 er en generell algoritme for en nxn matrise. Denne algoritmen følger annihileringsrekkefølgen som er beskrevet i avsnitt 2.1.6. Variabelen *matSize* sier hvor stor matrisen er, hvis *matSize* er 25 så antar algoritmen at det er en 25x25 matrise. Variablene tmp (x) og j (y) er koordinaten til elementet i A som skal annihileres, altså g.

Algoritmen er bygget opp av to løkker. Den ytre løkken finner koordinaten til elementet som skal annihileres. Hvis det fins flere elementer som kan annihileres i parallell, for eksempel annihilering av plass 5(se Figur 2.8), så vil koordinaten være til det øverste elementet. For plass 5 vil dette da si A[1][0]. Oppgaven til den indre løkken er å finne resten av elementene som kan annihileres samtidig som A[1][0]. Denne løkken leter fra topp mot bunn. For plass 5 vil den da finne A[3][1] og så A[5][2].

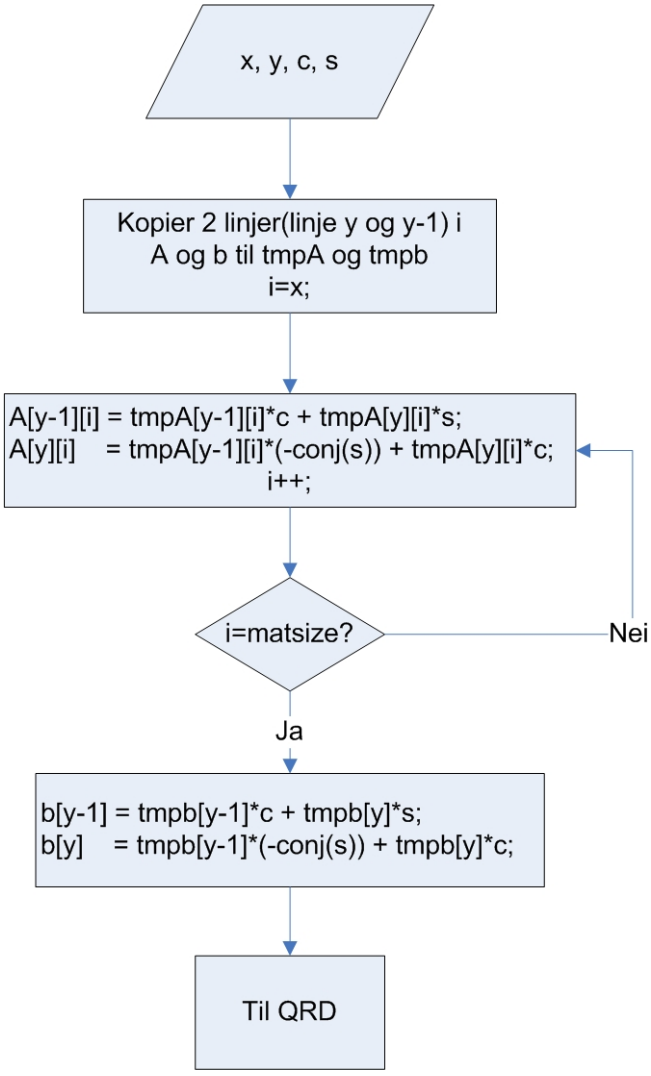
Algoritmen finner elementene som skal annihileres i tur og orden. Dette passer godt med at Givens er pipelinet. Når den indre løkken finner ett nytt element, starter den beregningsmodusen til Givens (se pil 1 i Figur 5.1). Beregningsmodusen produserer c og s i tur og orden og sender disse tilbake til kontrollogikken, se pil 2 i Figur 5.1. Kontrollogikken sender c og s videre til matriseoppdateringen sammen med koordinaten til g elementet, se pil nr 3. Når den indre løkken er ferdig, venter den ytre løkken på et flagg fra matriseoppdateringen (se pil 4 i Figur 5.1). Matriseoppdateringen bruker dette flagget for å si at det er klar til å utføre nye beregninger av Givens og påfølgende matriseoppdateringer.



Figur 5.2: Flytdiagram for kontrollogikken

5.1.2 Matriseoppdateringen

Denne delen beskriver hvordan matriseoppdateringen til Givens rotasjon kan implementeres i maskinvare. Matriseoppdateringen er del 2 av QR-dekomponeringen og har som oppgave å oppdatere A og b. Det vil si å multiplisere Q_x^T med A og b. Jeg laget en algoritme som utfører disse multiplikasjonene. Flytdiagrammet i Figur 5.3 viser denne algoritmen.



Figur 5.3: Flytdiagram for Matriseoppdateringen

Algoritmen har 4 inngangsverdier, x, y, c og s. Av disse er x og y koordinaten til elementet i A matrisen som skal annihileres, altså g. c og s er resultatet fra Givens rotasjon. $tmpA$ og $tmpb$ er kopier av A matrisen og b vektoren før oppdateringen starter. Det er nødvendig med kopiene $tmpA$ og $tmpb$ fordi resultatet av multiplikasjonene lagres direkte i A matrisen og b vektoren. Hvis de nye verdiene brukes i neste multiplikasjon vil dette gi feil resultat. Det er ikke nødvendig å kopiere hele A og b over i $tmpA$ og $tmpb$, kun de to linjene som skal oppdateres. Det vil si linje y og y-1. Se avsnitt 3.2.2 mer info om matriseoppdateringen til Givens.

Flytdiagrammet over matriseinverteringen i Figur 5.3 viser hvilke multiplikasjoner som skal til for å oppdatere A og b. Denne algoritmen utfører to og to multiplikasjoner om gangen. Den tar først alle elementene i de to linjene i A matrisen og så til slutt de to elementene i b vektoren.

Pipelining og hastighet

For at hastigheten til Matriseoppdateringen skal bli god nok, må den pipelines. For enkelhets skyld bør pipelinen lages slik at den tar to og to elementer samtidig, det vil si element A[y-1][i] og A[y][i]. Elementene kan da settes inn i pipelinen hver klokkepuls og de siste elementene som settes inn for hver oppdatering er b[y-1] og b[y] (se Figur 5.3). Denne pipelinen trenger kun de 3 følgende steg:

Step:1
- Utføre multiplikasjonene i de komplekse multiplikasjonene, det vil si multiplikasjoner med s.
- Utføre multiplikasjonene i de halvkomplekse multiplikasjonene, det vil si multiplikasjoner med c.

Step:2
- Resultatet fra de halvkomplekse multiplikasjonene er klare i dette steget.
- Utføre addisjon og subtraksjon i forbindelse med de komplekse multiplikasjonene.

Step:3
- Resultatet fra de komplekse multiplikasjonene er klare i dette steget.
- Utføre addisjonene mellom resultatene fra de halvkomplekse multiplikasjonene og de komplekse multiplikasjonene.

For å kunne utnytte parallelliteten til Givens fullt ut, må Matriseoppdateringen kjøre flere oppdateringer samtidig. Det vil si at den trenger flere pipelinede oppdateringsmoduler i parallell. Hvor mange moduler som kan kjøres i parallell er avhengig av matrisestørrelsen. Antallet er det mulig å

se ut i fra annihileringsrekkefølgen til matrisen. Metoden som brukes er å telle hvor mange elementer som kan annihileres samtidig. Figur 5.4 viser annihileringsrekkefølgen for en 7x7 matrise. De plassene som kan annihilere flest elementer om gangen er plass 6 og plass 7, disse kan ta 3 elementer. Plassene som kan ta minst er plassene 1,2,10 og 11, som kun tar et element. For en 7x7 matrise er mulig å oppdatere fra 1 til 3 elementer samtidig.

$$\begin{bmatrix} x & x & x & x & x & x & x \\ 6 & x & x & x & x & x & x \\ 5 & 7 & x & x & x & x & x \\ 4 & 6 & 8 & x & x & x & x \\ 3 & 5 & 7 & 9 & x & x & x \\ 2 & 4 & 6 & 8 & 10 & x & x \\ 1 & 3 & 5 & 7 & 9 & 11 & x \end{bmatrix}$$

Figur 5.4: Annihileringsrekkefølge for en 7x7 matrise

Jeg fant ut at det var et forhold mellom matrisestørrelsen og antall elementer som kan annihileres i parallell. For en nxn matrise gjelder følgende: Antall elementer= $n/2$, her ser vi bort ifra resten. Det vil si en 7x7 matrise gir: $7/2 = 3$. For en 25x25 matrise vil det da være mulig å utføre fra 1 til 12 oppdateringer i parallell. Hastighetsmessig vil det være best med 12 moduler i parallell, men dette er ikke særlig god utnyttelse av maskinvaren i de tilfellene det kun er mulig å kjøre 1 oppdatering. En 25x25 matrise trenger 300 oppdateringer for å lage R og b (Samme som antall rotasjoner, se avsnitt 4.4.1). Disse oppdateringene fordeles på 47 plasser. Med disse opplysningene er det mulig å regne ut hvor mange oppdateringer det er gjennomsnittlig per plass, som gir:

$$\frac{300}{47} \approx 6,38 \tag{5.1}$$

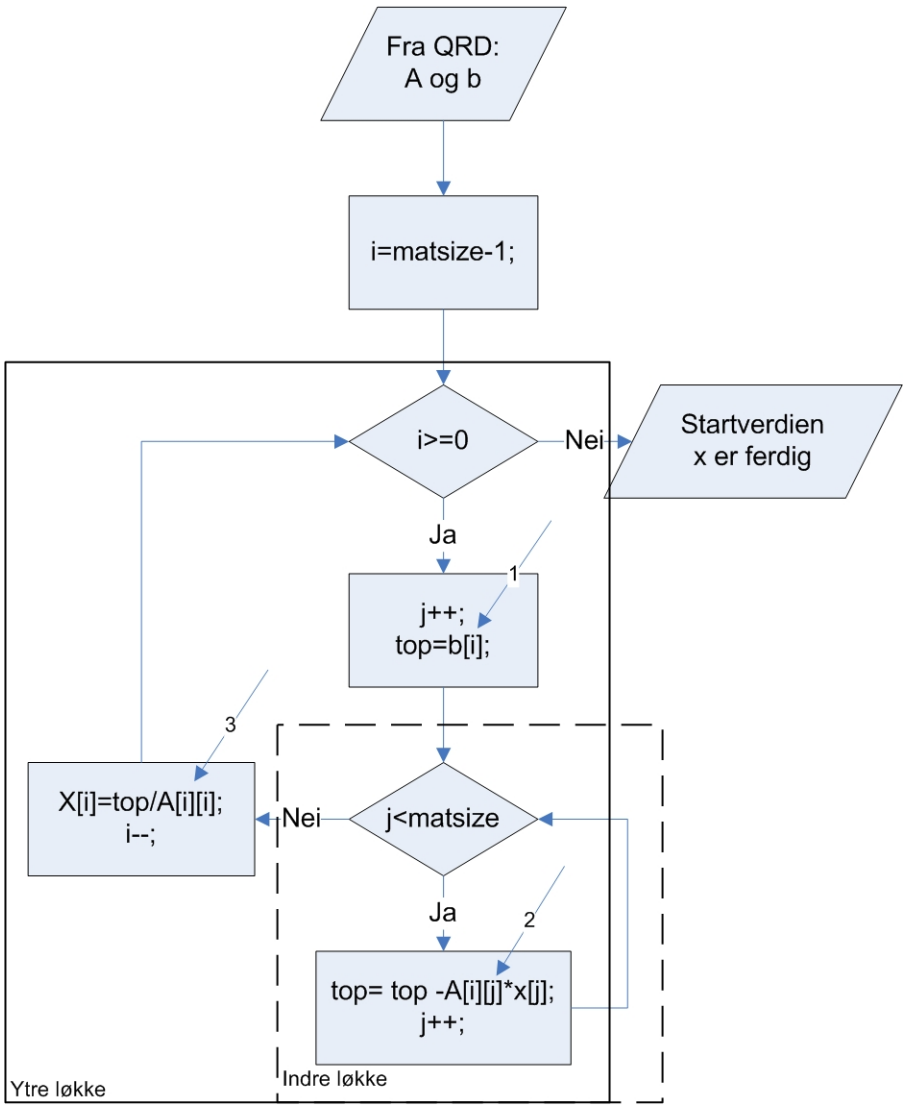
Det vil si at for en 25x25 matrise så vil 6 moduler gi et godt ytelse/maskinvare forhold. Men før man bestemmer seg for å bruke 6 moduler, er det viktig å vite hvor mye maskinvare 1 modul bruker. Hver modul bruker 2 komplekse multiplikasjoner, 2 halv komplekse mult og 2 komplekse addisjoner. Som totalt vil gi 12 multiplikasjoner, 5 addisjoner og en subtraksjon per modul (se avsnitt 2.2.1). 6 moduler vil da bruke 72 multiplikasjoner, 30 addisjoner og 6 subtraksjoner.

En annen ting som det må regnes på er om det er nødvendig med 6 moduler i parallell. Hvis det er mulig å slippe unna med færre moduler og fortsatt klare kravet på $100\mu s$ vil det være bedre. Med for eksempel 2 moduler i parallell, ender vi opp med å bruke 24 multiplikasjoner, 10 addi-

sjoner og 2 subtraksjoner. 2 moduler i parallell vil ikke utnytte potensialet helt, men det vil gi tilnærmet en fordobling i hastigheten til Matriseoppdateringen. I avsnitt 5.3 blir det utført en beregning av hvor mange moduler som trengs for å nå kravet på $100\mu s$.

5.2 Tilbakesubstitusjon

Hvordan tilbakesubstitusjon blir utført blir beskrevet i avsnitt **2.1.7**. Dette er beskrivelsen av hvordan den kan implementeres i maskinvare. Denne tilbakesubstitusjonen forutsetter at A matrisen er en *nxn* matrise, men utenom det er denne algoritmen uavhengig matrise størrelsen. Tilbakesubstitusjonen utføres på R og B, som ble produsert av QR-dekomponeringen.



Figur 5.5: Flytdiagram for tilbakesubstitusjonen

Algoritmen i Figur 5.5 har jeg utviklet på bakgrunn av beskrivelsen av tilbakesubstitusjon i avsnitt **2.1.7**. Denne algoritmen starter med å finne det nederste elementet, x[n]. Når den har funnet x[n], løser den ligningen for å finne x[n-1]. Neste element blir x[n-2] og slik fortsetter den til den løser element x[0]. Når den er har funnet x[0], har den funnet hele x vektoren.

For at denne algoritmen skal kunne utnytte maskinvaren effektivt, bør også denne pipelines. Pipelinen til tilbakesubstitusjonen deles inn i 2 deler. Den første pipelinen utfører operasjonene i den indre løkka. Den andre pipelinen utfører den operasjonen i den ytre løkka. Under er et forslag på hvordan den første pipelinen kan løses. Den andre delen av pipelinen beskrives i avsnitt 5.2.1

Step:1 Utføre multiplikasjonene i den komplekse multiplikasjonen $(A[i][j] \cdot x[j])$.

Step:2 Utføre addisjonene og subtraksjonene i den komplekse multiplikasjonen $(A[i][j] \cdot x[j])$.

Step:3 Utføre den komplekse subtraksjonen $(top - (A[i][j] \cdot x[j]))$.

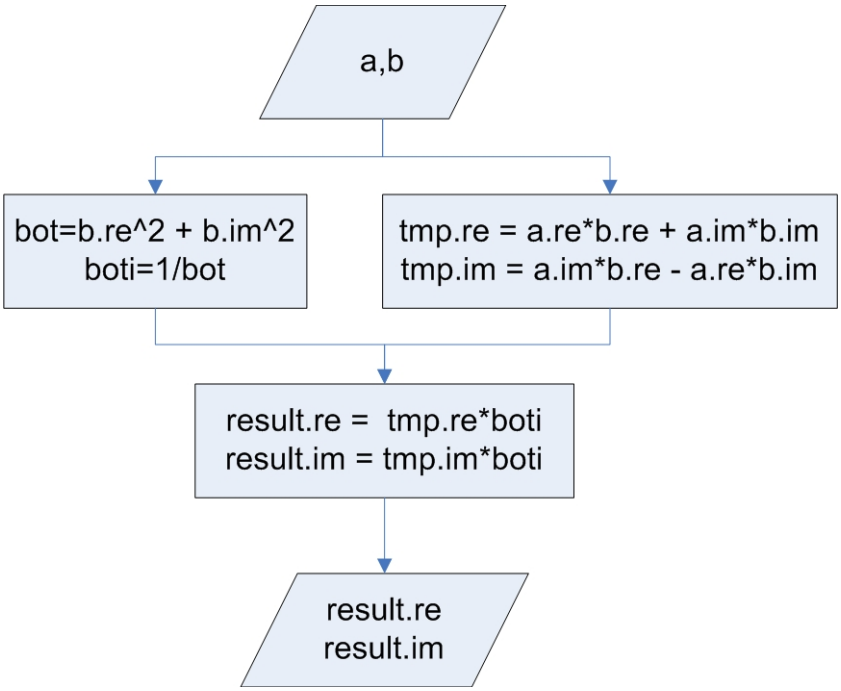
Men hva er sannsynligheten for at algoritmen produserer store tall? Variabelen top kan etter hvert bli stor. top er liten i beregningen av x[n], men den kan bli større og større jo nærmere vi kommer x[0]. Dette fordi beregningen av x[n-1] har en multiplikasjon og en subtraksjon mer enn x[n]. x[n-2] har 2 multiplikasjoner og 2 subtraksjoner mer enn x[n]. Slik fortsetter det helt til element x[0]. I en 25x25 matrise vil x[0] ha 24 multiplikasjoner og 24 subtraksjoner mer enn x[n].

For å unngå dette problemet kan man skalere ned begge sider av ligningen. Dette kan gjøres ved at man først skalerer ned top når den blir satt til $b[i]$ (se pil 1 i Figur 5.5). Deretter kan man skalere ned svaret fra $A[i][j] \cdot x[j]$ før man utfører subtraksjonen (se pil 2 i Figur 5.5). I tillegg må også $A[i][i]$ skaleres før man regner ut $top/A[i][i]$ (se pil 3 i Figur 5.5). Siden skaleringen blir utført på begge sider av ligningen med samme faktor, er det ikke nødvendig å holde rede på hvor mye den har blitt skalert.

Hastigheten til tilbakesubstitusjonen er ikke kritisk. Dette fordi den inneholder veldig få regneoperasjoner i forhold til QR-dekomponeringen. Tilbakesubstitusjonen bør derfor løses ved bruk av minst mulig maskinvare. Dette vil tilsvare 1 kompleks multiplikator, 1 kompleks subtraksjon og 1 kompleks divisjon pluss logikk for skalering.

5.2.1 Kompleks divisjon

Tilbakesubstitusjonen trenger komplekse divisjoner for å regner ut x vektoren. En kompleks divisjon er regnekrevende, se avsnitt 2.2.1. Noe som gjør at den fort kan bruke lang tid og bruke mye maskinvare. Figur 5.6 viser et flytdiagram over den komplekse divisjonen (a/b). Som vi ser deler flytdiagrammet seg i to i starten. Det vil si at disse to blokkene er uavhengige av hverandre og kan da kjøres i parallell.



Figur 5.6: Flytdiagram for kompleks divisjon

Variabelen a er dividenden og b er divisoren. Variabelen bot er den utregnede divisoren og tmp er den utregnede dividenden. Denne algoritmen deler ikke $tmp.re/bot$ og $tmp.im/bot$ direkte. Dette ville krevd 2 divisjoner. Isteden regner den først ut $1/bot$ og multipliserer denne med $tmp.re$ og $tmp.im$. Det vil si at vi bytter ut de to 2 divisjonene med en divisjon av typen $1/x$ og 2 multiplikasjoner. Dette sparer litt maskinvare

Siden tilbakesubstitusjonen skal pipelines, bør den komplekse divisjonen også pipelines. Det burde være mulig å løse hele den komplekse divisjonen med 4 pipeline steg. På neste side viser et forslag til hvordan den kan løses med i 4 steg.

Step:1
- Beregne bot.
- Utføre multiplikasjonene i beregningen av tmp.

Step:2
- Utføre addisjon og subtraksjon i beregningen av tmp. Denne beregningen er ferdig etter steg 2.
- Beregne $1/bot$. Denne beregningen krever 2 steg, den starter da i steg 2 og er ferdig etter steg 3.

Step:4
- Multiplisere $tmp.re$ og $tmp.im$ med $1/bot$.

Xilinx Virtex4 SX35 har mange multiplikatorer og addere, men ingen divisjonsenheter. Som nevnt i avsnitt 4.3 har KDC i tidligere prosjekter implementert raske og effektiv divisjons enheter. [14] implementerte 2 divisjons enheter basert på multiplikative algoritmer, nærmere bestemt Goldschmidt og Newton-Raphson. Det er matematisk bevist at Newton-Raphson regner korrekt (Disse bevisene er det referert til i [14]), mens Goldschmidt algoritmen er verifisert ved å teste alle mulige tallkombinasjoner innenfor et gitt tallområde (Dette ble utført i [14]). Goldschmidt algoritmen ble syntetisert, mens Newton-Raphson kun ble implementert på oppførsel nivå.

Newton-Raphson divisjon fungerer på nesten samme måte som Newton-Raphson kvadratrot, som blir beskrevet i avsnitt **4.3**. Først skalerer den divisoren x til tallområde [1,2). Deretter henter den ut resiprokalet $(1/x)$, utifra en oppslagstabell (se ligning 5.2). Presisjonen til dette resiprokalet er avhengig av størrelsen på oppslagstabellen. Dette resiprokalet går gjennom en iterativ algoritme som dobler presisjonen for hver iterasjon (se ligning 5.3). Til slutt multipliserer den resiprokalet med dividenden y (se ligning 5.4), for finne kvotienten (se ligning 5.5).

$$Fra\ oppslagstabell: \quad \frac{1}{x} \tag{5.2}$$

$$Etter\ en\ iterasjon: \quad \frac{1}{x} \quad (Presisjon\ x2) \tag{5.3}$$

$$Multiplikasjon\ med\ y: \quad y \cdot \frac{1}{x} \tag{5.4}$$

$$Resultat: \quad \frac{y}{x} \tag{5.5}$$

En multiplikativ Newton-Raphson divisjon med 18 bits presisjon, krever 3 klokkeperioder [14]. Siden variablene i tilbakesubstitusjonen blir skalert, er det nok med 18 bits presisjon. I denne oppgaven har vi ikke bruk for kvotienten (y/x), men resiprokalet $(1/x)$ når vi regner ut $1/bot$. Dermed kan vi kutte ut multiplikasjonen med y. Dette gjør at det kun trengs 2 klokkeperioder for å utføre divisjonen vi trenger. Divisjons enheten er veldig lik

Newton-Raphson kvadratrot. Pipelining av denne algoritmen blir dermed veldig lik pipelining av kvadratrot og får følgende 3 steg:

Step:1 Skaler x til tallområdet [1,2) og henter ut resiprokalet.

Step:2 Starter beregningene for å øke presisjonen til resiprokalet.

Step:3 Etter dette steget er beregningen resiprokalet ferdig beregnet.

5.3 Tidsforbruk

Kravet til tidsforbruk er at det skal være mulig å finne startverdien x på under 100μs. I tillegg er det ønskelig at A matrisen skal være på 25x25 elementer. Siden hele matriseinverteringen ikke ble implementert i maskinvare, er det ikke mulig å kjøre noen simulering for å teste dette kravet. Men det er viktig å vite om det vil være mulig å tilfredsstille dette kravet. Jeg vil derfor gi et estimat for tidsforbruket til matriseinverteringen.

Beregningene tar utgangspunkt i at A er en 25x25 matrise og at klokkefrekvensen er 50MHz. Siden beregningsmodusen til Givens kan kjøre på 58.3MHz (se avsnitt 4.5), antar jeg at matriseinverteringen også kan kjøre på denne frekvensen. Dette antar jeg fordi jeg anser Givens som den mest krevende modulen å implementere. For å transformere A og b til R og B, trengs det 300 rotasjoner og 5500 enkeltoppdateringer (Disse tallene ble telt med SoftQR og gjelder for 25x25 matriser). Videre blir det tatt hensyn til at Givens har en 11 stegs pipeline og at Matriseoppdateringen har en 3 stegs pipeline.

For å beregne tidsforbruket til Givens tok jeg utgangspunkt i ligning 4.14. Denne sier hvor mange rotasjoner som gjennomsnittlig kjøres i parallell, det er ca 6. Med denne informasjonen er det mulig å estimere hvor lang tid beregningsmodusen til Givens bruker.

$$300 \cdot 11 = 3300 \tag{5.6}$$

$$3300/6 = 550 \tag{5.7}$$

$$550 \cdot 20ns = 11\mu s \tag{5.8}$$

Først multipliseres antall rotasjoner med pipeline steg (ligning 5.6), for å finne totalt antall klokkepulser som brukes. Siden pipelinen til Givens ikke alltid er full, deles dette tallet på antall rotasjoner som gjennomsnittlig utføres i parallell (ligning 5.7). Til slutt multipliseres dette tallet med periodetiden (ligning 5.8). Denne estimeringen antyder at beregningsmodusen

til Givens bruker ca $11\mu s$. Det vil si at det fortsatt er $89\mu s$ for å utføre resten av matriseinverteringen.

Så skal det estimeres hvor lang tid Matriseoppdateringen bruker på 5500 enkeltoppdateringer. Pipelinen til denne består av 3 steg og er nesten alltid full. Siden pipelinen kun består av 3 steg, antar jeg i denne beregningen at det tar 0 tid å fylle opp pipelinen. Dette fører til at Matriseoppdateringen vil bruke ca 5500 klokkepulser. Dette multipliseres med klokkeperioden som er 20ns og gir at Matriseoppdateringen vil bruke ca $110\mu s$ på alle oppdateringene. Dette er en estimert tidsberegning når Matriseoppdateringen bruker 1 oppdateringsmodul. I avsnitt 5.1.2 blir det forklart hvordan denne tiden kan minkes ved bruk av flere oppdateringsmoduler. Ved å bruke 2 moduler vil tiden halveres. Dette gir da et nytt estimert tidsforbruk på ca $55\mu s$. Med 2 moduler er 66 av $100\mu s$ brukt opp på Givens og Matriseoppdateringen, som gir $34\mu s$ til tilbakesubstitusjonen.

Til slutt er det tiden til tilbakesubstitusjonen som skal estimeres. Her har jeg gjort beregninger i et Excel ark som estimerer hvor mange klokkepulser som brukes. (Dette Excel arket ligger på CD'en som følger med oppgaven og heter backsub.xls). Den bruker omtrent 475 klokkepulser. Med en frekvens på 50MHz vil den bruke omtrent $9.5\mu s$.

Dette gir et estimert totalt tidsforbruk for hele matriseinverteringen på $75.5\mu s$. Dette er omtrent 25% mindre enn kravet på $100\mu s$. Ifølge dette estimatet bør det være realistisk å gjennomføre matriseinverteringen på under $100\mu s$, ved bruk av 2 oppdateringsmoduler i parallell.

5.4 Andre implementeringer

For å se hvor god min implementasjon er, har jeg prøvd å sammenligne med andre matrise inverteringer. Det første jeg la merke til er at andre implementeringer ofte baserer seg på invertering av små matriser. Både [20] og [4] har implementert matriseinvertering på Xilinx FPGA'er for henholdsvis 3x3 og 4x4 matriser.

[20] baserer seg på AccelChip DSP Synthesis tool. Denne implementasjonen bruker AccelChip for å lage syntetiserbar VHDL kode utifra en modell av matriseinverteringen i Matlab. [20] konkluderer med at CORDIC gir best ytelse/maskinvare forhold. Denne artikkelen ser i utgangspunktet god ut, men den er det vanskelig å sammenligne med. Dette fordi de uttrykker hastigheten i MSPS (Mega Samples Per Second), men sier ikke hva en sample er. I tillegg sier den ikke hvor store matrisene er, men utifra Matlab koden ser det ut til å være en 3x3 matrise.

[4] har implementert matriseinverteringen ved bruk av Fast Givens eller "Squared Givens" som de kaller det i denne artikkelen. Denne implementeringen baserer seg på flyttall beregninger og kjører på en Xilinx Virtex-4 xc4vlx200 FPGA. Denne artikkelen inneholder flere fakta som det går å sammenligne med. Jeg regnet derfor ut hvor lang tid min implementasjon ville brukt på å invertere en 4x4 matrise.

Implementasjon	FPGA	Frekvens	Tid: 4x4	Slices	DSP48
Squared Givens [4]	4vlx200	115MHz	8.1μs	9117	22
Givens	4vsx35	58.3MHz	2.69μs	3255[1]	47[2]

Tabell 5.1: Sammenligning av implementasjon

Tabell 5.1 sammenligner implementasjonen til [4] mot min egen. For å regne ut hvor lang tid min implementasjon vil bruke på en 4x4 matrise, har jeg telt antall klokkepulser. Dette var mulig siden matrisen er såpass liten. Jeg endte da opp med at designet ville bruke 157 klokkepulser på å invertere en 4x4 matrise.

Beregningsmodusen til Givens er den tyngste delen i systemet mitt. Derfor antar jeg at en full implementasjon for 4x4 matriser, vil bruke færre slicer enn [4]. Av tabellen ser vi at min implementasjon bruker flere DSP48 (18x18 multiplikatorer) enn [4]. Men til gjengjeld er min implementasjon 3 ganger raskere.

[1] Dette antallet gjelder kun for beregningsmodusen til Givens.
[2] Dette antallet gjelder også kun for beregningsmodusen til Givens.

5.5 Oppsummering

Dette kapitlet har beskrevet hva som må tas hensyn til når resten av matriseinverteringen skal implementeres i maskinvare. Det er vist hvordan delen som styrer annihileringsrekkefølgen kan implementeres. Dette ble også gjort for Matriseoppdateringen og tilbakesubstitusjonen. Det ble gitt forslag til hvordan disse to kan pipelines. For å gi bedre oversikt ble det laget flytdiagrammer over alle algoritmene.

Det ble regnet ut at 6 oppdateringsmoduler i parallell i Matriseoppdateringen gir best ytelse/maskinvare forhold. Men det ble også vist at det ikke er nødvendig med mer enn 2 moduler i denne oppgaven.

Den komplekse divisjonen i tilbakesubstitusjonen ble optimalisert slik at det var mulig å bruke en divisjonsenhet basert på Newton-Raphson istedenfor 2.

Til slutt ble det utført en estimering av tidsforbruket til matriseinverteringen. Her ble det estimert at ved bruk av 2 oppdateringsmoduler i parallell, kan matriseinverteringen utføres på 75.5μs. Dette er omtrent 25% lavere enn kravet på 100μs. Dermed er det realistisk at matriseinverteringen kan gi LMS-utjeveren startverdien x på under 100μs. Dette vil gjøre det mulig for den nye taktiske radiolinjen å benytte seg av frekvenshopping.

Kapittel 6

Test

Dette kapitlet beskriver hvordan systemet ble testet og verifisert. Testene gikk ut på å verifisere at systemet regnet korrekt og sjekke hvor store tall de forskjellige rotasjons algoritmene kunne generere. Disse testene ble utført i softwaremodellen SoftQR, som er en komplett implementering av matriseinverteringen i C++. Her ble alle tre rotasjonsmodulene implementert og testet.

Testene ble utført på hele matrise inverteringen, slik at tilbakesubstitusjonen og QRD algoritmen ble verifisert samtidig med rotasjonsmodulene. Senere ble den valgte rotasjonsmetoden implementert på i syntetiserbar VHDL. Deretter ble VHDL koden simulert i ModelSim 6.0 SE og syntetisert i Precision. Til slutt ble det kjørt Place-and-Route i Xilinx ISE 7.1.

6.1 Softwaremodellen SoftQR

Jeg implementerte software modell av matriseinverteringen i C++ og den har følgende funksjoner:

1. Givens rotasjon
2. fast-Givens rotasjon
3. CORDIC rotasjon
4. QR-dekomponering
5. Tilbakesubstitusjon
6. Generere tilferdige matrise og vector
7. Lese inn matrise og vector fra fil
8. Skrive matrise og vector til fil

Givens, fast Givens og CORDIC rotasjon ble implementert på en slik måte at det er likegyldig for QR-rekkefølgen og Matriseoppdateringen hvem av metodene som blir brukt. Disse to er implementert etter beskrivelsen av QR-dekomponeringen i avsnitt 2.1.6. Tilbakesubstitusjonen er implementert etter beskrivelsen i avsnitt 2.1.7. Denne kan også brukes på alle 3 metodene. Dette gjør det enkelt å bytte mellom rotasjonsmodulene i SoftQR.

Metoden som genererer tilfeldige matriser og vektorer i SoftQR, genererer disse med komplekse tall. De komplekse tallene har en verdi på mellom -2 og 2. Denne metoden blir brukt for å raskt generere en matrise å teste på. Generatoren for tilfeldige tall i C++ er *pseudo-random*. Det vil si at den følger en fast, men tilsynelatende tilfeldig rekkefølge når den genererer tilfeldige tall. Dette fører til at SoftQR genererer de samme tilfeldige matrisene, hver gang den kjører. Dette vil igjen si at når de 3 rotasjonsmodulene blir testet med tilfeldige matriser, blir alle testet med de samme matrisene.

Metoden for å skrive til fil, lager matrisene og vektorene på et format som Matlab forstår. De blir lagret i filen på samme måte som man bruker når man skriver inn en matrise i Matlab for hånd. Figur 6.1 viser hvordan en kompleks 2x2 matrise skrives inn i Matlab og hvordan denne matrisen ser ut. Metoden for å lese inn matriser og vektorer, leser det samme formatet.

A=[1+i,2+2i;3-5i,-1-i]

$$\Downarrow$$

$$A = \begin{bmatrix} 1+i & 2+2i \\ 3-5i & -1-i \end{bmatrix}$$

Figur 6.1: Matlab format for matriser

6.1.1 Test av rotasjonsmodulene

Før jeg kunne implementere en metode i VHDL, måtte jeg sjekke at de regnet korrekt. Dette ble gjort på følgende måte:

1. SoftQR genererte en kompleks matrise A og vektor b
2. Matrisen og vektoren ble lagret i en fil
3. SoftQR utførte en QR-dekomponering med en gitt algoritme(Givens, fast-Givens eller CORDIC) for å lage R matrisen og B vektoren
4. SoftQR utførte tilbakesubstitusjon på R og B for å lage x vektoren
5. x vektoren ble lagret til fil
6. Filen inneholder nå A,b og x og leses inn i Matlab
7. Matrise invertering ble utført på A og b i Matlab og Matlab produserte en fasit for x vektoren
8. Matlab sammenlignet x med fasiten og rapporterte avviket

Denne testen ble kjørt på alle 3 rotasjonsmetoden for å verifisere at de regnet korrekt.

6.1.2 Verifisering av SoftQR

For å verifisere at SoftQR fungerte ble det testet mot et testprogram i Matlab, som ble laget av Asgeir Nysæter på KDC. Programmet lager kanaldata som skal være ganske like de dataene kretsen vil motta i virkeligheten. Ut ifra kanaldataene laget programmet A matrisen og b vektoren og med hjelp av innebygde metoder i Matlab laget det også x vektoren (startverdien). Disse dataene er alt som trengs for å få testet SoftQR.
Testsekvensen er som følger:

1. Matlab programmet genererer A,b og x.
2. Matlab starter SoftQR.
3. A og b blir lest inn i SoftQR.
4. SoftQR kjører qrd og tilbakesubstitusjon.
5. SoftQR lagrer x vektoren på et format som Matlab skjønner i en .m fil.
6. Matlab programmet leser inn x vektoren fra SoftQR og sammenligner med sin egen x vektor.

Matlab programmet har 2 parametre som kunne justeres på. Den første er signal-støy forholdet og den andre er forsinkelsesspredning. Forsinkelsesspredningen forteller hvor stor spredning det er på det mottatte multipath signalet i tid. Matlab programmet ble satt opp til å justere disse slik at alle mulige interessante kombinasjoner ble testet. Hver test gjennomførte testsekvensen ovenfor og la maksimum avvik mellom x vektorene inn i en tabell. Til slutt ble det største avviket i tabellen lest ut. Dette testprogrammet ble utført på SoftQR, etter at Givens hadde blitt valgt som rotasjonsmodul. Siden avviket mellom x vektorene var lite (avvik<10^{-5}), viste dette at QR-dekomponeringen, Givens rotasjon og tilbakesubstitusjonen fungerte i SoftQR.

6.2 Simulering av Givens

Det å teste Givens med alle mulige inngangsverdier ville ta utrolig lang tid. Siden f og g er komplekse, inneholder de som tidligere nevnt en reell og en imaginær del. Det vil si at det er 4 inngangsvariabler, hvor alle består av 25 bit. Alle mulig kombinasjoner av disse variablene ville føre til $2^{25} \cdot 2^{25} \cdot 2^{25} \cdot 2^{25} = 2^{100} \approx 1.268 \cdot 10^{30}$ forskjellige muligheter, noe som ikke er mulig å teste på en vanlig pc.

En annen mulighet ville vært å testet med tilfeldige tall, men dette er ikke noen fullgod test metode. Dette fordi man kan simulere i lang tid uten at de tilfeldige tallene gir en feilsituasjon. Denne metoden kan gi en indikasjon på at systemet fungerer, men det vil aldri kunne verifisere systemet godt nok.

I avsnitt **4.1** kom det frem at størrelsen til $f2$ og $g2$ bestemte om algoritmen kom til å lage store eller små verdier. Jeg valgte derfor å bruke dette som utgangspunkt når jeg skulle velge testverdier. For å teste et så bredt spekter som mulig, valgte jeg å gi f og g verdier som ville føre til stor variasjon i verdiene til $f2$ og $g2$. Med dette mener jeg verdier som testet:

$$
\begin{aligned}
f2 = &\quad null &&\quad mot\ g2\ lik\ null,\ liten,\ medium\ og\ stor.\\
f2 = &\quad liten &&\quad mot\ g2\ lik\ null,\ liten,\ medium\ og\ stor.\\
f2 = &\quad medium &&\quad mot\ g2\ lik\ null,\ liten,\ medium\ og\ stor.\\
f2 = &\quad stor &&\quad mot\ g2\ lik\ null,\ liten,\ medium\ og\ stor.
\end{aligned}
$$

Ved å teste med disse verdiene, fikk jeg sjekket at beregningsmodusen til Givens fungerte for et bredt spekter av inngangsverdier. Dette kunne gjøres uten bruk av en alt for omfattende testbenk. Det ble også testet at Givens regnet korrekt når f og g inneholdt negative verdier. Testbenk fila til Givens ligger i den vedlagte CD'en og heter "tb_givens.vhd".

Det ville ikke vært nødvendig å teste med alle mulige verdier av f og g. Dette fordi f og g genererer $f2$ og $g2$, som blir skalert med en gang. På grunn av denne skaleringen blir veldig mange tilfeller av størrelsene til $f2$ og $g2$ like, som fører til forskjellige inngangsverdier kan teste det samme. Testbenken ble så enkel at det ikke var nødvendig å lage en automatisk sjekk av verdiene, resultatet ble sjekket mot et Excel regneark som utførte Givens Rotasjon. Dette regnearket ble verifisert mot Givens i Matlab. Excel regnearket ligger i den vedlagte CD'en og heter "Givens-vhdl-test.xls" .

Kapittel 7

Konklusjon

I denne oppgaven har jeg sett på hvordan matriseinvertering av store matriser kan implementeres i en FPGA. Jeg har vurdert flere metoder for å utføre matriseinverteringen og fant ut at matriseinverteringen ved hjelp av QR-dekomponering var best egnet. QR-dekomponering ble valgt fordi den har en god numerisk stabilitet og at den har gode muligheter til parallellisering.

Videre har jeg sammenlignet 3 metoder for å utføre QR-dekomponeringen. Metodene som ble sammenlignet er Givens, fast Givens og CORDIC rotasjon. Disse metodene ble valgt fordi de så ut til å være best egnet til å utføre QR-dekomponeringen. Sammenligningen av metodene gikk ut på å analysere hvor mange operasjoner algoritmene brukte og hvor store tall de kunne generere.

Som hjelpemiddel til å velge metode, implementerte jeg matriseinverteringen i en softwaremodell i C++. Denne softwaremodellen kalte jeg SoftQR og den inneholdt alle 3 rotasjonsmodulene. SoftQR ble satt opp til å teste hvor store tall rotasjonsmetodene genererte. I denne testen kom Givens best ut. SoftQR ble også brukt til å verifisere at matriseinverteringen regnet korrekt. Dette ble gjort ved å sammenligne resultatet fra matriseinverteringen med et Matlab program.

Før beregningsmodusen ble implementert i maskinvare, ble det gjennomført en grundigere analyse av algoritmen. Ved å gjøre dette fant jeg ut hvilke variabler som måtte skaleres. Jeg har også gjort en avveining på hvor mye variablene burde skaleres. Her valgte jeg å skalere variablene til et tallområde som unngikk store og små tall i senere variabler, men som samtidig ikke gikk for mye utover presisjonen til variablene.

Beregningsmodusen til Givens ble pipelinet for å oppnå god nok ytelse.

Denne pipeliningen kostet lite ekstra maskinvare. Det ble regnet ut for en 25x25 matrise, at pipelinen gjorde beregningsmodusen til Givens 6 ganger mer effektiv enn en upipelinet versjon.

Jeg har spesialtilpasset en multiplikativ kvadratrotenhet basert på Newton-Raphson til bruk i beregningsmodusen. Denne spesialoperatoren bruker kun 3 klokkeperioder for å beregne operasjoner av typen $1/\sqrt{x}$. Dette medførte at pipelinen til beregningsmodusen gikk ned fra 15 til 11 steg. I tillegg eliminerte den behovet for bruk av divisjon i algoritmen. Dette ga mindre bruk av maskinvare og kortere beregningstid.

Beregningsmodusen til Givens rotasjon ble implementert i syntetiserbar VHDL. Jeg valgte å implementere denne delen av systemet fordi jeg anså den som den vanskeligste delen å implementere. Jeg laget også en testbenk som testet et bredt spekter av forskjellige inngangsverdier. Dette er en liten testbenk med nøye utvalgte testverdier. All simulering ble utført ved hjelp av Modelsim 6.0 SE og syntetisert ved hjelp av Mentor Graphics Precision. Etter syntese av beregningsmodusen til Givens ble det kjørt Place-and-Route (PAR) i Xilinx ISE 7.1. Tabell 7.1 viser dette resultatet fra PAR. Målfrekvensen til systemet er 53.76MHz. Denne implementasjonen oppnådde en maks klokkefrekvens på 58.3MHz og kan derfor kjøres på målfrekvensen.

Antall DSP48	47 av 192	24%
Antall Slices	3255 av 15360	21%
Maks klokkefrekvens	58.289MHz	

Tabell 7.1: Resultat fra PAR i ISE7.1

Jeg har også sett på hva som skal til for å implementere resten av systemet på en effektiv måte. Det er blitt gitt forslag til hvordan matriseoppdateringen og tilbakesubstitusjonen kan pipelines. Det ble sett på hvordan en multiplikativ divisjonsenhet basert på Newton-Raphson kan brukes for utføre operasjoner av typen $1/x$. Det ble også vist hvordan denne operasjonen kan utnyttes i den komplekse divisjonen i tilbakesubstitusjonen. Den komplekse divisjonen krever i utgangspunktet 2 divisjoner. Det ble vist hvordan disse kan byttes ut med en divisjon av typen $1/x$ og 2 multiplikasjoner. Dette fører til mindre bruk av maskinvare.

Det ble gjort et estimat over hvor lang tid hele matriseinverteringen vil bruke. Dette estimatet viser at det er mulig å invertere en 25x25 matrise på $75.5\mu s$. Dette er omtrent 25% lavere enn kravet på $100\mu s$. Dermed bør det være realistisk at matriseinverteringen kan gi LMS-utjevneren startverdien x på under $100\mu s$. Dette vil gjøre det mulig for KDC sin nye taktiske radiolinjen RL532A å benytte seg av frekvenshopping.

Til slutt sammenlignet jeg min implementasjon mot [4] sin implememementasjon fra 2005, som baserer seg på fast Givens. Sammenligningen baserer seg på invertering av en 4x4 matrise med komplekse verdier. Denne sammenligningen viser at min implementasjon antakeligvis bruker færre slicer, men flere DSP48. Til gjengjeld er den 3 ganger raskere.

7.1 Videre arbeid

Etter at implementeringen av beregningsmodusen til Givens var ferdig, la jeg merke til at det hadde vært mulig å gjøre pipelinen enda kortere. Beregningen av s er den siste beregningen som utføres og beregnes på følgende måte:

$$stmp = f \cdot d1 \tag{7.1}$$
$$s = (-conj(g)) \cdot stmp \tag{7.2}$$

Ved å omorganisere ligning 7.1 og 7.2 får vi følgende ligninger:

$$stmp = f \cdot (-conj(g)) \tag{7.3}$$
$$s = d1 \cdot stmp \tag{7.4}$$

Beregningen av stmp i ligning 7.3 er en kompleks multiplikasjon mellom inngangsverdiene f og g. Denne multiplikasjonen er uavhengig av alle andre operasjoner i algoritmen og kan derfor beregnes i parallell med de andre operasjonene. Dermed er det mulig å beregne denne slik at den er ferdig samtidig som $d1$. Multiplikasjonen i ligning 7.4 er en halvkompleks multiplikasjon. Denne er det mulig å utføre på 1 klokkeperiode.

I avsnitt 4.4 kommer det frem at $d1$ er ferdig beregnet i pipeline steg 7. Beregningen av s blir derfor ferdig i steg 8. Dette er samme steg som c er ferdig. Dermed trenger beregningsmodusen til Givens kun 8 steg i pipelinen. Dette gir en ytterligere reduksjon i antall pipelinesteg på 27%. Optimaliseringen av beregningene i pipelinen har dermed nesten halvert antall pipelinesteg, fra 15 til 8.

Det er viktig å presisere at dette kun er en omorganisering av beregningene og krever derfor ikke mer maskinvare. Derimot får man en raskere beregning av c og s. For KDC har dette liten betydning siden estimatet viser at den nåværende implementasjonen er rask nok.

Bibliografi

[1] Andrew A. Anda and Haesun Park. Self-scaling fast rotations for stiff least squares problems. *Linear Algebra and its Applications*, 234:137–161, 1996.

[2] Ray Andraka. A survey of cordic algorithms for fpga based computers. *ACM Press*, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays:191–200, 1998.

[3] David Bindel, James Demmel, William Kahan, and Osni Marques. On computing Givens rotations reliably and efficiently. *ACM Transactions on Mathematical Software*, 28(2):206–238, June 2002.

[4] J.R. Calvallaro Chris Dick, M. Karkooti. Fpga implementation of matrix inversion using qrd-rls algorithm. In *39th Asilomar Conference on Signals, System and Computers*, pages 1625–1629, 2005.

[5] Laszlo Erdos. Linear algebra for math2601: Numerical methods. http://www.math.gatech.edu/ bourbaki/math2601/Webnotes/6num.pdf, Nov 2006.

[6] Alan Gens, Zongli Lin, Charles Jones, Dali Luo, and Thorsein Rrenzel. Fast givens goes slow in matlab. *ACM Press*, 26:11–16, 1991.

[7] W. Morven Gentleman. Least squares computations by givens transformations without square roots. *IMA Journal of Applied Mathematics*, 13(3):329–336, 1973.

[8] Wallace J. Givens. Numerical computation of the characteristic values of a real symmetric matrix. Ornl-1574, Oak Ridge National Laboratory, 1954.

[9] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[10] Penn State Polymer Physics Group. Direct calculations: minors. http://www.plmsc.psu.edu/∼www/matsc597/vectors/inv_methods/node1.html, Nov 2006.

[11] Penn State Polymer Physics Group. Problems with the direct method. http://www.plmsc.psu.edu/∼www/matsc597/vectors/inv_methods/node2.html, Nov 2006.

[12] John Halleck. Least squares network adjustments via QR factorization. http://home.utah.edu/∼nahaj/survey/QRintro.html'#'Givens, Nov 2006.

[13] S. Hammarling. A note on modifications to the givens plane rotations. *IMA Journal of Applied Mathematics*, 13(2):215–218, 1974.

[14] Simen Gimle Hansen. Multifunksjonell beregningsenhet for digital signalprosessering. Master's thesis, Universitetet i Oslo, april 2005.

[15] Prof. Michael T. Heath. Parallel numerical algorithms 15 qr factorization. Department of Computer Sience. University of Illinois at Urbana-Champaign, 2004.

[16] Tom Lindstrøm Lisa Lorentzen, Arne Hole. *Kalkulus Med èn og flere variable*. Universitetsforlaget, 2003.

[17] V.G. Oklobdzija. An implementation algorithm and design of a novel leading zero detector circuit. In *26th Asilomar Conference on Signals, System and Computers*, 1992.

[18] V.G. Oklobdzija. An algorithmic and novel design of leading zero detector circuit: Comparison with logic synthesis. *IEEE Transaction on VLSI Systems*, 2(1):124–128, March 1994.

[19] J. Rolstad. Realisering av en multiplikator for komplekse flyttall uten avrunding av mellom resultater optimalisert for hastighet. Master's thesis, Institutt for Informatikk, Universitetet i Oslo, Feb 1998.

[20] Ramon Uribe and Tom Cesear. Implementing matrix inversions in fixed-point hardware. *DSP magazine*, 2005.

[21] Frank Westad, Klaus Diepold, and Harald Martens. Qr-plsr: Reduced-rank regression for high-speed hardware implementation. *Journal of Chemometrics*, 10:439–451, Des 1998.

[22] Wikipedia. CORDIC. http://en.wikipedia.org/wiki/CORDIC, Okt 2006.

[23] Wikipedia. Qr decomposition. http://en.wikipedia.org/wiki/QR_decomposition, Nov 2006.