Forord:

Master studiet ved Norges Tekniske og Naturvitenskapelige Universitet (NTNU) avsluttes med innleveringen av et prosjektarbeid som tar hele siste semester. Denne rapporten omhandler mitt prosjektarbeid i denne perioden. Oppgaven er gitt av Kongsberg Defence Comunictation (KDC), men jeg har vært tilknyttet Institutt for elektronikk og telekommunikasjon (IET) under retningen Krets og Systemkonstruksjon(KS), arbeidet er utført ved NTNU. Hovedveilederen har vært Professor Kjetil Svarstad og ekstern veileder er Simen Gimle Hansen fra KDC.

Oppgaven gikk ut på å konstruere digitalheltalls grammatikk, med fokus på multiplikative divisjons algoritmer. KDC benytter allerede i dag en slik algoritme, så med utgangspunkt i det rettet jeg fokus mot generering av initialverdier for denne algoritmen for å gjøre den mer effektiv. Grunnen til at jeg valgte denne oppgaven var at den virket meget interessant både teoretisk og praktisk, ettersom den omhandler noe så grunnleggende som divisjon, da det kan brukes i veldig mange digitale applikasjoner. Divisjon er også en resurskrevende opperasjon i den digitale verden, så det er stor gevinst i å gjøre denne bedre. Prosjektet har vært meget spennende og lærerikt både det rent designmessige med at det skal passe sammen med andres kode og utfordringer rundt problemstillingen samt de mange utfordringer i å verifisere å teste systemet.

Jeg vil gjerne takke Mine veiledere Kjetil Svarstad og Simen Gimle Hansen for den gode oppfølgingen jeg har fått gjennom hele dette prosjektarbeidet og deres behjelpelighet med å finne stoff og svare på spørsmål rundt oppgaven. Jeg vil også takke Roar Skogstrøm som har vært meddelelig på mange av vår prosjekt møter. Sist men ikke minst en stor takk for at jeg fikk besøke dere på KDC.

Trondheim 12.06.07

Martin Rognerud.

Sammendrag

Jeg har i denne oppgaven jobbet med digital heltallsaritmetikk, og da sett spesielt på feltet deling. Deling er en meget ressurskrevende opperasjon i digitalteknikk, det er derfor mye tid og areal å spare på å forbedre delingsopperasjonen. Jeg ser her på hvordan det er mulig å benytte multiplikative algoritmer til å utføre delingen, og da med hovedfokus på Newton-Raphson metoden. Newton-Rapson metoden er en gammel algoritme som har vært gjenstand for en mengde analyser rettet mot flere bruksområder, det finnes altså gode bevis på nøyaktighet, beregningsmengde og ressursbruk ved implementering i digitalekretser. Det er derfor ikke blitt utviklet noe nye algoritme basert på Newton-Raphson, men benyttet en godt dokumentert algoritme, da denne ser ut til å være den best egnede for implementasjon. Som andre iterative algoritmer trenger Newton-Raphson metoden et startpunkt for å kunne finne frem til det riktige svaret. Som vist i oppgaven har Newton-Raphson metoden den egenskapen at den fordobler antallet korrekte siffer per iterasjon og derfor er det viktig med et bra startpunkt hvis algoritmen skal komme fort fram til det antall korrekte bit som er ønsket i svaret. Problemet med å startverdier er hvis man skal ha stort antall korrekte bit trenger man mange verdier lagret og dette vil ta stor plass, eller man trenger en egen utregnings krets noe som ville ta ekstra tid. Oppgaven her viser først den simpleste formen for startverdi der algoritmen benytter samme startverdi til alle innverdier. Bare en innverdi vil gi lite areal kostnad men det vil føre til at man trenger flere iterasjoner for å finne det korrekte svaret og dermed vil det ta lenger tid. Oppgaven viser vider mer kompliserte oppslagstabeller og lineære interpolasjons metoder for å kunne øke antall korrekte bit i startverdien til minst mulig arealkostnad. Noen typer av oppslagstabellene er også blitt simuler for å vise at de faktisk gir tilstrekkelig korrekt svar. Til slutt i oppgaven er det hvis hvordan Newton-Raphson metoden og startverditabellene kan implementeres i FPGA.

Mitt arbeid i denne oppgaven bestod av fire deler.

Første del var et Literatur studie for å kunne bedømme hvilke algoritmer som ville passe best for denne oppgaven. Her landet jeg på Newton-Raphson metoden da den var godt bevist og ga gode resultater innen resurs og tidsbruk. Som beskrevet over er startverdien til Newton-Raphson metoden viktig for å gjøre metoden rask. Literatur studiet forstsatte dermed med en gjennomgang av forskjellige måter å fremskaffe startverdien. Startverdien er hovedfokuset i denne oppgaven, og hovedvekten legges på bipartitet og multipartite oppslagstabeller. Det for disse kan fremskaffe høyere antall korrekte bit en konvensjonelle oppslagstabeller uten den enorme veksten i areal.

Andre leg gikk ut på å implementer algoritmen med VHDL, her fikk jeg tilgang til kode fra KDC med en implementert Newto-Raphson algoritme, denne passet med den måten jeg hadde valgt å benytte Newton-Raphson metoden så jeg benyttet denne. Jeg konstruerte her et python skript som genererte VHDL koden til de forskjellige tabellene.

Tredje del omhandlet Testing av de implementerte modulene. Her tok jeg koden jeg hadde fått fra KDC og omskrev den slik at den ville passe sammen med oppslagstabellene jeg hadde laget. Testingen ble gjort i modelsim, med testbenker som selv sjekket om svaret er korrekt og skriver eventuelle feilmeldinger ut i en log fil. Hoveddelen av testingen ble gjort på en oppførsels modell av Newton-Raphson metoden, men også den fult syntetiserbare koden ble testet.

Fjerde del bestod av dokumentasjon av teori og implementasjonen av Newton-Raphson metoden og oppslagstabellene. Det ble også her fremstilt resultater fra syntes og simuleringer gjort under oppgaven.

Innholdsliste.

Vedleggsliste:

Kapittel 1.

Innledning.

I dagens samfunn er elektronisk utstyr blitt integrert i alle aspekter av hverdagen, skulle det være jobb, fritid, matlagning eller barnepass. Etter som elektronikken smelter seg inn i bakgrunnen av tilværelsen blir kravet om kvaliteten på disse tjenesten bare høyere og høyere. Dette fører til stor økning i beregnings og kommunikasjons hastighet. Prisen på tjenestene vil også bli presset ned etter som det blir en selvfølge for folk, og ikke noe spesielt som de er villig til å betale litt mer for. Dette presser produsentene av tjenestene til å finne så rimelige løsninger som mulig. Den billigste løsningen vil i de fleste tilfeller si å omgjøre signalene fra omverden til digitale signaler så fort som mulig og holde dem digitale så lenge som mulig. Problemet med dette er det at verden rundt er analog (vi ser her litt opp fra kvantefysikken), så disse signalene må omformes, noe som innfører feil i signalet. Er det samtidig benyttet billige sensorer mot omverden vil dette innføre enda mer feil. For at denne feilen ikke skal ødelegge signalene helt må signalet fra sensorene transformeres for å kompensere for feilen. Etter som kravet om økende kvalitet fortsetter trengs det høyere oppløsnings fra sensorene, noe som igjen fører til større utregninger i transformene.

Nå skal det sies at denne oppgaven omhandler mer profesjonelt utstyr enn mye av det som er nevnt over, men det er akkurat det samme som skjer i det profesjonelle liv. Forskjellen er at i profesjonelt utstyr er det enda høyere krav enn i det som benyttes privat, derfor koster det også mer. Digitale signalbehandlings transformer benyttes til mange forskjellige ting, skulle det være filtrering av støy, gjenkjenning av signaler, modulasjon/demodulasjon av signaler eller kryptering. Det er ofte flere transformer som skal behandle signalet etter hverandre, og dette fører til forsinkelse og reduksjon i overføringshastighets til systemet.

Digitale transformer inne holder forskjellige matematiske operatorer, deriblant pluss, minus, gange og dele, samt mer kompliserte operatorer som sinus og cosinus. Subtraksjon, addisjon og multiplikasjon finnes det mange kretser som er tildels effektive, selv om også disse inneholder forsinkelser. Divisjon derimot er en meget resurskrevende opperand, men som benyttes i flere typer transformer. Det er derfor ønskelig og kunne omskrive divisjonen slik at man kan benytte subtraksjon/addisjon og multiplikasjon. Denne oppgaven tar for seg hvordan dette er mulig å gjøre på en effektiv måte. Metodene benyttet i denne oppgaven kan også benyttes til andre operatorer som krever høy beregnings kraft for å regne ut.

1.1 Målsetting

Målet med oppgaven er å erverve seg kunnskap om hvordan divisjon foregår i digitale kretser med spesiell tanke på Newton-Rapshon metoden. Deretter sette seg inn i hvordan man kan gerere initialverdier for Newton-Rapshon metoden å da med spesiell tanke på bipartitetabeller.

Det som ønskes svar på er:

- Hvilke alternativer av initialverdier er det.
- Hvordan genereres bipartitetabeller
- Hvilke ytelses/areal forbedring kan bipartitetabeller gi
- Hvilke bakdeler er det med bipartitetabeller
- Hvilke mulige ytelse/areal forbedringer kan man få av multipartitetabeller eller interpolasjon
- Verifisere at tabellene gir riktig resultat.

Denne målsetningen differensierer fra den som er gitt i oppgave teksten. Opprinnelige oppgaveteksten er gitt under.

Studiefase hvor studenten skal sette seg inn i:
Digital kretskonstruksjon i VHDL. Digital aritmetikk med vekt på multiplikativ divisjon og hvis tiden tillater kvadratrot beregning. Metoder for å finne initiale startverdier ved divisjon og eventuelt kvadratrot. Syntese av VHDL til FPGA teknologi. Bruk av hardware konstruksjonsverktøy (Modelsim, Precision RTL og Xilinx ISE).

Kreativ fase hvor studenten skal utføre:
Konstruksjon, syntese og funksjonell verifikasjon av en regnemodul for heltalls divisjon og eventuelt kvadratrot i VHDL. Sammenligne løsningen med andre subtraktive og multiplikative løsninger. Regnemodulen bør ha en fleksibel konfigurerbar arkitektur (stort areal/rask eller lite areal/langsom).

Grunnen til at oppgaven ikke ble eksakt lik det som først ble gitt i oppgaveteksten som vist over er at det fort viste seg at metodene for divisjon og kvadratrot som var benyttet i tidligere master oppgaver for KDC virket å være den beste så den beste måten å forbedre den på var å forbedre den initiale start verdien. KDC uttrykte også et ønske om nærmere studier rundt dette.

1.2 Arbeidet med oppgaven

Mitt arbeid.

Prosjektet bestod av 4 deler.

1. Litteraturstudie.
2. Implementasjon.
3. Testing.
4. dokumentasjon.

Litteraturstudiet igjen var delt opp i en del hvor jeg skulle lese meg opp på de forskjellige måtene å utføre divisjon på, en del hvor jeg skulle fordype meg i den metoden som virket best. Første delen ble en overflatisk gjennomgang av forskjellige typer delings algoritmer, med hovedvekt på multiplikative algoritmer. Overfladisk både på grunn av at jeg oppfattet det som KDCs ønske å se spesielt videre på Newton-Raphson metoden å på grunn av at den så ut til å skille seg ut som det beste alternativet. Hovedkonkurrenten er Goldschmidts algoritme, men grunnet at denne ikke kan bevises i sin helhet var ikke denne et alternativ. Algoritmen beskrevet i [30] er også et alternativ. Det er ikke ønskelig og få en gjentagelse av Intels divisjons feil, der en liten feil gjorde at 1 av ca 9 milliarder forsøk ga feil resultat, selv om dette ikke er mye kan feilen komme på kritiske tidspunkt. Merk her at Intel feilen ikke var ved bruk av Goldschmidts algoritme, men en feil i oppslagstabellene i SRT divisjonsalgoritmen. En fin oppsummering av de mest kjente divisjonsalgoritmene er gitt i [6].

Andre del av Litteraturstudiet gikk da ut på å sette seg nærmere inn i Newton-Raphson metoden. Som beskrevet i tidligere arbeid under bakgrunn kapittelet er det blitt gjort omfattende studier rundt denne metoden innen feilen ved beregning og dens brukbarhet i elektronikk med hensyn på hvilke komponenter som trengs for å kunne utføre ittereringen. det var derfor ingen vits å forsøke og lage nye varianter av denne, men finne fram til den beste for akkurat det formålet den skal benyttes til her. Teorien rundt Newton-Raphson metoden finnes i kapittel 3 til denne oppgaven, [1][2][3][4][5][6] samt flere av de andre. Etter gjennomgangen av implementasjonen av Newton-Raphson metoden i [5] og [6] viste det seg at disse var meg et gode så jeg valgte å følge disse. En annen grunn til å velge akkurat denne implementeringen er at den allerede var benyttet i KDCs utstyr og jeg hadde fått tilgang på vhdl filene til KDCs implementasjon slik at jeg kunne benytte dem i simuleringene. Disse filene er ikke publisert med denne oppgaven.

1.3 Disposisjon

Kapittel 1: Innlednign til oppgaven.

Kapittel 2: Bakgrunnen til oppgaven og tidligere arbei utført på ommrådet.

Kapittel 3: Teori rundt Newton-Raphson algoritmen, funksjons og feilanalyse.

Kapittel 4: Teori rundt forskjellige typer initialeverdier. Spesielt med hensyn på Bipartite, multipartitetabeller og lineær interpolasjon med delte tabeller.

Kapittel 5: Om implementeringen av Newton-Raphson metoden og initialeverdier.

Kapittel 6: Diskusjon rundt oppgaven, oppnådde resultater og konklusjon

1.4 Lesing av rapporten

Det forutsettes i denne rapporten at leseren har litt kjenskap til digital teknikk og signalbehandling, det meste er likevel prøvd forklart grundig slik at sammenhengen går direkte fram av oppgaven. Det er tilstrebet i oppgaven å ikke gjenbruke variabler, men det er i visse tilfeller gjort på grunn av at det skal være likest mulig de artiklene det refereres til. Grunnen til gjenbruken av variabler er for å gjøre det enklere å sette det i sammenheng med liknende arbeider, dette er satt merknader der dette er tilfelle. Det skal ellers ikke være nødvendig å ha kjenskap til bipartitetabeller eller Newton-Raphson metoden før man leser denne oppgaven. Antallet korrekte bit som oppgis i denne rapporten er ett mindre enn den virkelige mengden hvis ikke annet er oppgitt. Grunnet til dette er at normaliseringen av nevneren i gir resiprokalet en verdimengde lik [1,0.5), altså vil verdimengden bestå av $0.1b_1b_2b_3\ldots$ i alle tilfeller med unntak av 1. Denne oppgaven inneholder mange utledninger dette er for å bevise de forskjellige teoriene har analytiske metoder å finne verste tilfelle eller tilnærmet verste tilfelle.

Kapittel 2.

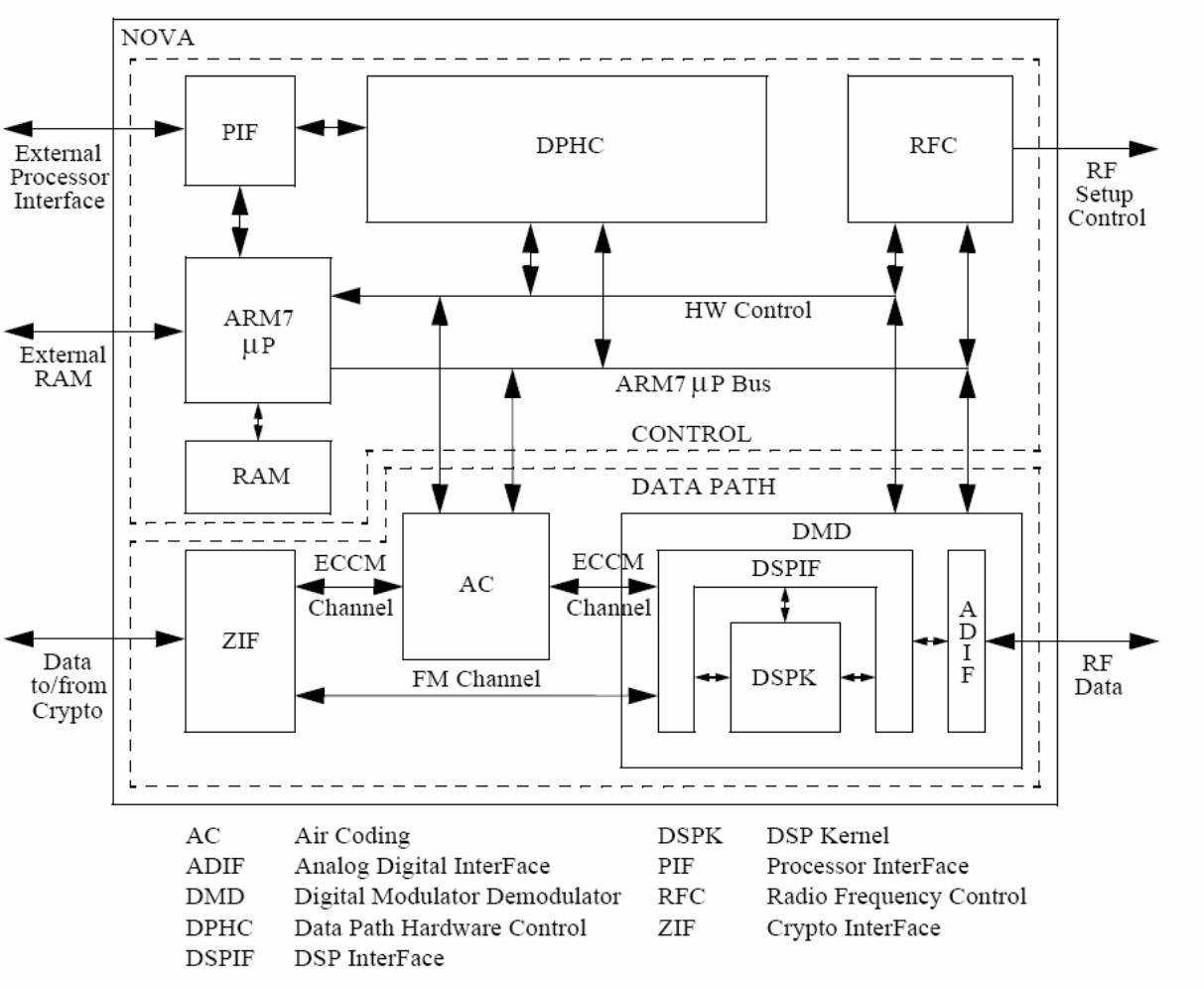
Bakgrunn.

KDC lager kommunikasjonsutstyr for militære formål, i slikt materiell er det store krav til at signalene er enten utilgjengelige for fienden (negativ S/N) eller at de er kryptert slik at det ikke er mulig for fienden å lese av informasjonen i sendingen. Det er også store krav om at utstyret fungerer selv om det er mye støy i omgivelsene. For og få til det som er beskrevet ovenfor er det nødvendig å transformere signalene over til spesielle format. Slik signal behandling har også blitt en viktig del i veldig mange andre applikasjoner skulle det være for en enkel termostat, avanserte medisinske apparater eller for kommunikasjons utstyr som beskrevet over.

Denne signal behandlingen skjer i all hovedsak digitalt og mange digitale transformasjoner inne holder divisjon. Etter som dette er transformer som vil gå kontinuerlig er det ønskelig at divisjonen går så fort som mulig, som i all konstruksjon av digitale komponenter er viktig at det tar lite plass, etter som prisen per areal er svert høy, eller som i denne oppgaven at man benytter FPGAer er det begrenset med resurser. Disse to punktene er ofte umulig og tilfredsstille samtidig derfor kan det være ønskelig å konstruere deler av kretsene slik at den kan benyttes til andre formål også. Det er her multiplikative divisjons algoritmer kommer inn, da de bruker multiplikatorer til å utføre divisjonen, disse kan da også benyttes til andre formål hvis det er ledig kapasitet, noe som ofte er tilfellet etter som det er mange transformer som ikke kan parallelliseres.

Algoritmen som benyttes i denne oppgaven er newton-raphson metoden som gir dobbelt antall riktige bit for hver itterasjon. Dette fører til at initialverdien som benyttes er veldig viktig for hvor kjapp algoritmen blir, det er derfor ønskelig å generere en mest mulig korrekt initialverdi. Problemet med initial verdien er at for å få den raskt må den ikke inneholde for mange kalkulasjoner for å finne den og for å få den presis nok trengs tabeller som øker eksponentielt etter som flere rette siffer ønskes. Det er her hovedfokuset for denne oppgaven kommer inn da den tar for seg bipartite tabeller som er en måte å dele opp disse tabellene slik at de ikke øker like mye som konvensjonelle tabeller og man trenger bare enkel koding av innholdet i disse tabellene for å finne initialverdien som skal brukes i newton-raphson algoritmen

Denne oppgaven tar utgangspunkt i tidligere master oppgaver [5][6] gitt av KDC, disse har omhandlet innføringen av divisjons og kvadratrot algoritmer i LFR som er et system KDC utvikler for militær kommunikasjon. For LFK utviklet KDC en egen ASIC kalt NOVA som inneholdt moduler for digital modulasjon, demodulasjon, grensesnitt mot omliggende kretser og en mikroprosessor. En modell av NOVA kretsen er vist i Figur 1.



Figur 1: modell av NOVA kretsen til KDC
(denne figuren er hentet fra[5])

1.1 Problemstilling.

Det blir stadig dyrere og produsere ASICer samtidig som kapasiteten til FPGAer øker kontinuerlig, KDC har dermed gått mer over på FPGA. Utviklingen fører til at det er nødvendig med mer og mer beregnings kraft, så KDC ønsker å øke overføringshastigheten og dermed bit bredden benyttet i algoritmene, dette vil igjen føre til at divisjons algoritmen blir mye tregere. En måte å hindre at algoritmen blir tregere er å starte med et mer presist start punkt. Denne oppgaven ser på muligheten med å benytte bipartitet [13] eller multipartite [15][17] oppslagstabeller for å generere initialverdiene.

1.2 Tidligere arbeid.

Deling har vært benyttet siden lenge før elektronikken ble oppfunnet, også det binære tallsystemet har eksistert mye lenger enn elektronikken, så matematikere har jobbet med problematikk rundt dette emnet siden lenge før jeg ble påtenkt. Et av matematikkens store problem er at ikke alle funksjoner har en nøyaktig løsning, nevner for eksempel femtegradslikninger og oppåver som Nils Henrik Abel i 1824 bevist ikke kan løses ved rot utdragning. På grunn av at det ikke alltid er mulig å finne eksakte svar har matematikere gjennom tiden prøvd å finne måter for å tilnærme disse funksjonene på en best mulig måte. I denne oppgaven er det hovedsakelig tatt utgangspunkt i Newton-Raphson metoden. Newton-Raphson metoden er lik Newtons metoden som først ble publisert i 1685 i boken ”A Treatise of Algebra both Historical and Practical” av John Wallis. Joseph Raphson forenklet den og viste at den kunne brukes til suksessiv tilnærming. Det finnes mange bøker og artikler som omhandler dette temaet, noen av disse er [1][2][3][4]. Etter som dette er en gammel metode er den blitt grundig analyser både med hensyn til feil, optimalisering og kompatibilitet til bruk innen digitalesystemer. Det er mulig å benytte newtons metode på mange forskjellige måter for å oppnå divisjon, disse har alle sine styrker og svakheter, den jeg har valgt å benytte er beskrevet mange steder men den er enkelt beskrevet i [5] og [6]. Implementeringen av Newton-Raphson metoden er også meget bra beskrevet i [5] og [6]. Generelt innen implementering av digitale moduler er det gjort mye arbeid, blant annet [7][8][9]. Innen implementasjon av divisjons algoritmer er det mye informasjon i [10][11][12].

Newton-Rapson metoden trenger en startverdi for at den skal kunne finne fram til riktig verdi, denne startverdien innehar stor viktighet på hvor mange iterasjonen som skal til for å nå svaret. Det har derfor vært en god del interesse rundt dette spørsmålet de senere årene da Newton-Raphson metoden kan benyttes på mange typer funksjoner, samt oppslagstabeller kan benyttes som kalkulerings enhet hvis ikke alt for mange korrekte bit er påkrevd. Her i oppgaven kommer det til å bli beskrevet tabeller opp mot 32 bits korrekthet [13]. [5] og [6] beskriver flere enkle metoder for å lage oppslagstabeller. Disse krever kjapt stor plass hvis ønsket antall korrekte bitt er stort. Da et bit gir en dobling av tabellstørrelse eller flere utregnings ledd nøyaktigheten til slike tabeller vises i [14]. En ny måte å tenke oppslagstabeller blir derimot innført i [13], denne måten reduserer økningen på tabellene per ønsket korrekt bit ut, slik at det er mulig å få større antall korrekte bitt i startverdien og dermed spare tid ved at vi kan redusere antall iterasjonen. Etter dette har det kommet flere artikler om dette temaet [15][16][17] og det er hovedtemaet også i denne rapporten. Andre artikler relaterte til bipartite og multipartitetabeller er [18][19][20][21][22][23][24][25][26].

Kapittel 3.

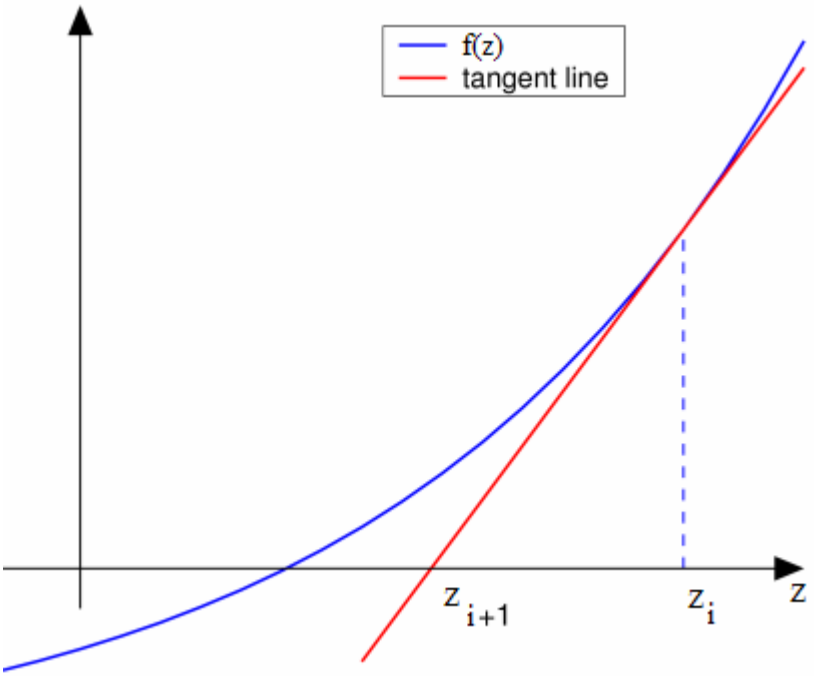
Newton-Raphson itterasjon:

Her er Newton-Raphson metoden beskrevet og hvordan denne kan benyttes til å beregne divisjon. Mesteparten av materialet til denne beskrivelsen er hentet direkte fra [5], men det er også hentet litt fra [31].

Newton-Raphson metoden også kjent som Newtons metode, er en mye brukt metode for beregning av problemer som ikke har eksakte løsninger (i det minste ikke kjent). Metoden går ut på å finne skjæringspunktet mellom en akse og en funksjon f. Metoden har en iterativ formel som vist under:

$$z_{i+1} = z_i - \frac{f(z_i)}{f'(z_i)} \qquad \mathbf{1}$$

Denne formelen fås ut fra definisjonen til derivasjon. Figuren under viser hva som menes med de forsjellige bokstavene i formelen.



Figur 1:Newton-Raphson metode for å finne nullpunkt
(bildet er hentet fra [31])

1.1 Utrednig av Newton-Raphson metoden

Starter med definisjonen på derivasjon:

$$f'(z) = \frac{f(z_{i+1}) - f(z_i)}{z_{i+1} - z_i} \qquad \mathbf{2}$$

Definisjonen til derivasjon inneholder også betingelsen at avstanden mellom z_i og z_{i+1} går mot null. Denne er ikke tatt med her men den vil bli oppfylt etter som z_{i+1} kommer nærmere å nærere det faktiske nullpunktet.

Flytter så z_{i+1} på venstre side for seg selv, med den forutsetning at $f(z_{i+1}) = 0$ som det er i dette tilfellet:

$$f'(z) = \frac{0 - f(z_i)}{z_{i+1} - z_i}$$
$$z_{i+1} - z_i = -\frac{f(z_i)}{f'(z_i)} \qquad \mathbf{3}$$
$$z_{i+1} = z_i - \frac{f(z_i)}{f'(z_i)}$$

Etter som Newton-Raphson metoden gir en enkel iterativ struktur er den godt egnet til bruk it maskinvare.

1.2 Divisjon med Newton-Raphson.

Her vises hvordan Newton-Raphson kan benyttes til divisjon. Det finnes mange flere måter å gjøre det på enn det som er vist i dette underkapitelet men den som er representert her er den beste med tanke på implementasjon i digitalteknikk.

1.2.1 Funksjonsanalyse.

Ved divisjon i Newton-Raphson metoden tar man utgangspunkt i at det er mulig å omskrive divisjonen slik:

$$Q = \frac{X}{Y} = X \cdot \frac{1}{Y} \qquad \mathbf{4}$$

Hvor Q er kvotienten, X er dividenden og Y er divisoren. Newton-Raphson blir da benyttet til å beregne resiprokalet til divisoren.

Velger på grunnlag av dette f(z) til å være:

$$f(z) = \frac{1}{z} - Y \qquad \textbf{5}$$

Denne vil i nullpungtet (f(z)=0) være lik resiprokalet.

$$\begin{aligned} f(z) &= 0 \\ 0 &= \frac{1}{z} - Y \\ z &= \frac{1}{Y} \end{aligned} \qquad \textbf{6}$$

Her kunne det blitt valg alle mulige funksjoner som er vil ha nullpunkt i resiprokalet eller at de har nullpunkter som kan benyttes til videre beregning av divisjonen, slik som X/Y, 1-1/Y, 1-X/Y og $1/Y^2$. Hvilke formel som velges har innvirkning på inplementasjonen, hvor fort svar formelen vil konverger og om den inneholder divisjon selv eller ikke, men den valgte er den best egnede (som forfatteren kjenner til).

Den deriverte av f(z) blir da:

$$f'(z) = -\frac{1}{z^2} \qquad \textbf{7}$$

Settes dette inn i Newton-Raphsom (ligning 1) metoden får vi:

$$\begin{aligned} z_{i+1} &= z_i - \frac{\frac{1}{z_i} - Y}{-\frac{1}{z_i^2}} \\ z_{i+1} &= z_i - \left(\frac{1}{z_i} - Y\right)\left(-z_i^2\right) \\ z_{i+1} &= z_i - \left(-z_i + Yz_i^2\right) \\ z_{i+1} &= 2z_i - Yz_i^2 \\ z_{i+1} &= z_i\left(2 - Yz_i\right) \end{aligned} \qquad 8$$

Substraksjonen er ekvivalent med en toer-komplemet opperasjon.Ulempen med ligning 8 er at multiplikasjonene er avhengig av hverandre slik at de ikke må gjøres sekvensielt.

1.2.2 Feilanalyse.

Den absolutte feilen for hver iterasjon med denne metoden er resultatet etter iterasjonen trukket fra den riktige verdien. Dette gir:

$$E_{i+1} = \frac{1}{Y} - z_{i+1}$$
$$E_{i+1} = \frac{1}{Y} - \left(2z_i - Yz_i^2\right)$$
$$E_{i+1} = \frac{1}{Y} - 2z_i + Yz_i^2$$
$$E_{i+1} = Y\left(\frac{1}{Y^2} - \frac{2z_i}{Y} + z_i^2\right) \quad \textbf{9}$$
$$E_{i+1} = Y\left(\frac{1}{Y} - z_i\right)^2$$
$$E_{i+1} = YE_i^2$$

Vi regner ut hva z_0 må være for at svaret skal konvergere ved å bruke kravet om at E_i -> 0 når i -> ∞. dette gjøres på følgende måte:

$$E_{i+1} = YE_i^2$$
$$E_i = YE_{i-1}^2$$
$$E_i = YE_{i-1}^2$$
$$E_{i-1} = YE_{i-2}^2$$
$$E_{i-1}^2 = Y^2E_{i-2}^4$$

Vi ser da at :

$$E_i = Y\left(Y^2\left(Y^4\left(...\left(Y^{2^{i-2}}\left(Y^{2^{i-1}}E_0^{2^i}\right)\right)..\right)\right)\right)$$

Potensene til Y ser vi at blir en geometrisk rekke så vi får da :

$$2^0 + 2^1 + ... + 2^{2^{i-1}} = \frac{1-2^i}{1-2} = 2^i - 1$$

feilen etter i iterasjoner blir da :

$$E_i = Y^{2^i-1}E_0^{2^i}$$
$$E_i = Y^{2^i-1}\left(\frac{1}{Y} - z_0\right)^{2^i}$$
$$E_i = Y^{2^i-1}\frac{\left(1-Yz_0\right)^{2^i}}{Y^{2^i}} \quad \textbf{10}$$
$$E_i = \frac{\left(1-Yz_0\right)^{2^i}}{Y}$$

Vi bruker så grensebetingelsen vist over:

$$\lim_{i \to \infty} E_i \to 0$$

$$\lim_{i \to \infty} \frac{(1 - Yz_0)^{2^i}}{Y} \to 0$$

Dette betyr at : **11**

$$\lim_{i \to \infty} (1 - Yz_0)^{2^i} \to 0$$

Dette er oppfylt hvis :

$$|1 - Yz_0| < 1$$

Grense verdiene blir da:

$$1 - Yz_0 = 1$$

$$z_0 = 0$$

$$1 - Yz_0 = -1$$

$$z_0 = \frac{2}{Y} \quad \textbf{12}$$

z_0 blir da liggende i intervallet :

$$0 < z_0 < \frac{2}{Y}$$

Vi ser her at hvis z_0 oppfyller dette kravet vil E_i bli mindre å mindre for hver iterasjon for $(1 - Yz_0)^{2^i}$ vil bli mindre å mindre for hver iterasjon.

Antallet riktige siffer finnes ut fra:

feilen vil vere en andel *e* av svarert, vi kan da skrive resultatet vårt på følgende måte :

$$\frac{1}{Y} - e \cdot \frac{1}{Y} \quad \textbf{13}$$

e må dermed være :

$$e = \frac{feil}{s\,\text{var}} = \frac{\frac{1}{Y} - z_i}{\frac{1}{Y}} \quad \textbf{14}$$

Dette tilsvarer den relative feilen i forhold til svaret. Det minste riktige bittet vil da bli det som ligger i sifferet over høyeste bitet i $|e|$. Etter som vi jobber i det binære tallsystemet tilsier dette at.

$$|e| < 2^{-p} \qquad \textbf{15}$$

Hvor p er antall riktige bit.

Hvis z_0 er oppfylt etter det som er beskrevet over, kan vi se ut fra ligning 9 at antallet riktige bit blir fordobla for hver iterasjon. (E_{i+1} er mindre enn E_i på grunn av at z_0 er i riktig intervall). Først regner vi ut den relative feilen til E_{i+1}:

$$|e_{i+1}| = \left| \frac{E_{i+1}}{\frac{1}{Y}} \right| = |YE_{i+1}| = \left|Y\left(YE_i^2\right)\right| = \left|Y^2E_i^2\right| = |YE_i|^2 = |e_i|^2 \qquad \textbf{16}$$

Vi antar så at den relative feilen fra E_i var mindre enn 2^{-p}. vi kan da regne ut antall riktige siffer:

$$\begin{aligned} |e_i| &< 2^{-p} \\ |e_i|^2 &< 2^{-2p} \end{aligned} \qquad \textbf{17}$$

Det blir altså doblet så mange riktige siffer etter en iterasjon.

På grunn av dette vil antallet nødvendige iterasjoner for å oppnå ønsket nøyaktighet bli:

I = nødvendige itterasjoner.
p = antallet korekte bit i startverdien.
q = ønsket nøyaktighet i bit.

For hver Iterasjon får vi fordobling av riktige bit så vi får :

$$\begin{aligned} & p \cdot 2^I = q \\ & \log(p \cdot 2^I) = \log(q) \\ & \log(p) + \log(2) \cdot I = \log(q) \\ & I = \log(q) - \log(p) \\ & I = \log\left(\frac{q}{p}\right) \end{aligned} \qquad \textbf{18}$$

1.3 Oppsumering.

Dette kappitelet tar for seg hvordan Newton-Raphson metoden kan benyttes til divisjon. Kapitelet fokuserer bare på en av de mange måtene det er mulig å benytte Newton-Raphson metoden til divisjon. Måten som er valgt er den som etter forfatterens kjennskap er den beste med hensyn på hvilke resurser den trenger for å implementeres og hastighet. Kapittelet inneholder også en komplett feilanalyse av Newton-Raphson metoden for å kunne bestemme bevise den analytiske metoden for å finne frem til hvor korrekt svare er.

Metoden kapittelet kommer fram til for å finne resiprokalet ved iterative forbedringer er $z_{i+1} = z_i(2 - Yz_i)$, denne vil gi en fordobling av antall korrekte bit per iterasjon.

Kapittel 4.

Initiale startverdi.

Her omtales forskjellige metoder for å finne initiale startverdier. Dette dokumentet er hentet fra [5][6][13][14][15][17]. Her innføres y som er nevneren i resiprokalet, denne y er normalisert til intervallet [1,2), grunnen til dette er at dette gir resultatmengde lik $[1,\frac{1}{2})$. Dette gir flere fordeler blant annet at med unntak av 1 vil hekle resultat mengden ha en ener i bit nummer -1, og det fører til multiplikasjoner av verdier i resultat mengden vil aldri gi resultat over en.

Som vist i kapittelet om Newton-Raphson metoden (ligning 18) er antallet iterasjoner som trengs for å utføre divisjonen meget avhengig av nøyaktigheten på startverdien. Antallet iterasjonen som trengs kan regnes ut med $I = \log_2\left(\frac{q}{p}\right)$, altså er startverdien veldig viktig i hvor fort algoritmen kommer fram til den ønskede nøyaktigheten.

1.1 Enkle oppslagstabeller

Her blir tre typer metoder for valg av initiale startverdier kjapt beskrevet, dette er:

1. Konstant startverdi.
2. Stegvis tilnærming.
3. Lineær interpolasjon.

For sammenligning mellom de forskjellige metodene brukes antallet riktige bit og den relative feilen som er gitt ved (som ligning 16):

$$e = \left| \frac{\frac{1}{Y} - z_0}{\frac{1}{y}} \right| = \left|1 - Yz_0\right| \qquad \mathbf{1}$$

Antall bit som er riktige kan da regnes ut etter følgende formel:

$$\begin{aligned} & e = 2^{-p} \\ & \log(e) = -p \cdot \log(2) \\ & p = -\frac{\log(e)}{\log(2)} \end{aligned} \qquad \mathbf{2}$$

Dette vil selvsagt gi akkurat det som gir feilen så etter som p skal være mindre må det være et tall som er mindre enn det denne formelen gir. Som nevnt i innledningen til dette kapittelet er y normalisert til [1,2). Dette realiseres ved å skifte y mot venstre helt til den første eneren står som MSB. Grunnen til at denne metoden for normalisering ble valgt er at dette representerer det samme som å gangen y med et antall toere og dermed vil det være en enkel sak å bare dele tallet på dette antallet toere for å få tilbake riktig resultat. Divisjon med en toer heltalls potens er det samme som å høyreskifte resultatet like mange ganger som potensen.

1.1.1 Konstant startverdi:

Konstant startverdi er som navnet tilsier at startverdien er den samme uansett hvilke Y som benyttes. Dette er den enkleste måten men fører til større feil og da mindre riktige bit i start-verdien. Hvis startverdien velges til 0.5 vil dette gi minst logikk, dette vil gi 1 riktig bit. Etter ligning 18 vil dette gi:

$$I = \log_2\left(\frac{16}{1}\right) = 4$$

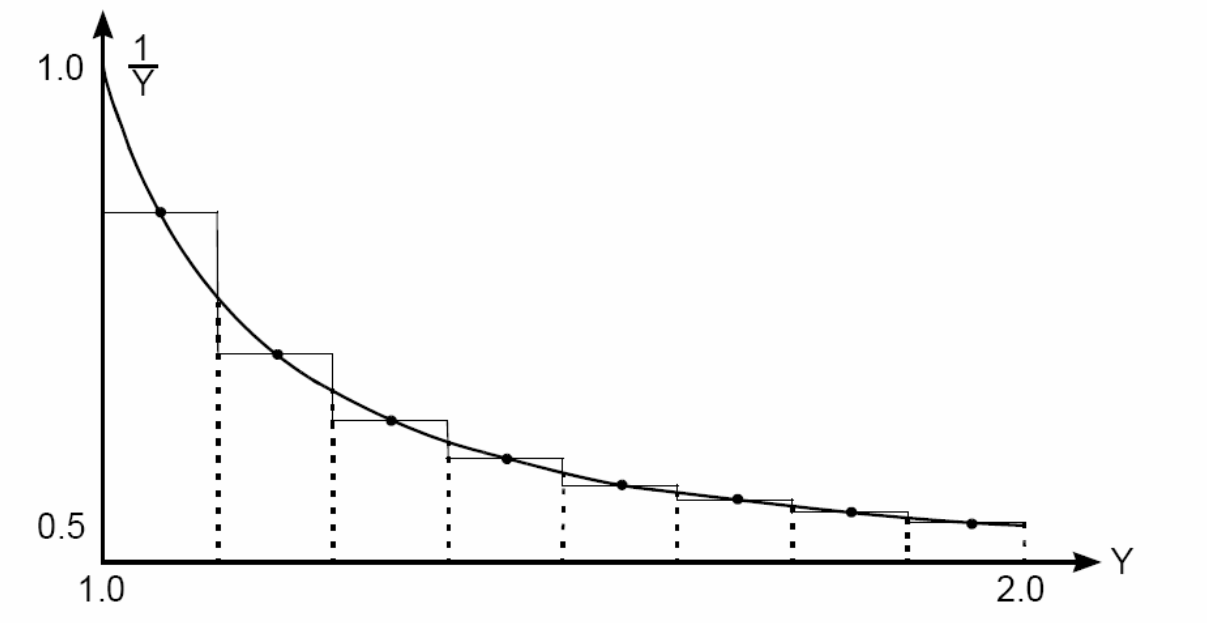
Det er derimot mulig å få en bedre tilpassning med å minimere den relative feilen. Dette gjøres ved at vi finner punktet der den relative feilen er lik i begge grenseområdene.

$$\begin{aligned} & Y = 1, Y = 2 \\ & |1 - z_0| = |1 - 2z_0| \\ & z_0 = 0 \vee \frac{2}{3} \\ & e = \frac{1}{3} \end{aligned} \qquad \mathbf{3}$$

Etter som z_0 ikke kan være 0 hvis den skal tilfredsstille konvergeringskravene må den bli 2/3. Dette vil gi 1,584 riktige bit (regnet ut med ligning 20). Grafisk blir dette det samme som å ha en rett linje.

1.1.2 Stegvis tilnærming.

Stegvis tilnærming er konstante startverdier bare delt opp utover verdimengden. Dette er illustrert i figuren under:



Figur 1: Grafisk visning av stegvis tilnærming
(hentet fra [5])

Det er ønskelig å ha et antall steg som går opp i en toerpotens etter som vi her benytter det binære tallsystemet. Deler derfor opp i 2^k intervaller, disse intervallene er like store. Intervallene er da inndelt slik:

$$\left[\frac{\left(2^k + j\right)}{2^k}, \frac{\left(2^k + j + 1\right)}{2^k}\right)$$
$$j = \left\{0,1,\ldots,2^k - 1\right\}$$

Det er vist i [13] at det som gir mest riktige siffer vil være å velge midtpunktet i hvert intervall. Dette kan også lett ses ut fra eksempelet gitt under konstant startverdi (ligning 21), der det ble løst at beste valg av startverdi var 2/3 som blir midt i "intervallet". Midtpunktene er da gitt ved:

$$\frac{2^{k+1} + 2j + 1}{2^{k+1}} \qquad \textbf{4}$$

Dette gir resiprokalene til å være:

$$\frac{2^{k+1}}{2^{k+1} + 2j + 1} \qquad \textbf{5}$$

For å kunne finne ut hvor mange riktige bit dette vil gi må vi finne hvor det er størst relativ feil:

Relative feilen er gitt ved (ligning 19) $e = \left|1 - Yz_0\right|$, z_0 er da startverdien innen det intervallet som ses på.

$$e_{nedre} = \left| 1 - \frac{2^k + j}{2^k} \cdot \frac{2^{k+1}}{2^{k+1} + 2j + 1} \right|$$

$$e_{nedre} = \left| 1 - \frac{2^{2k+1} + j2^{k+1}}{2^{2k+1} + j2^{k+1} + 2^k} \right|$$

$$e_{nedre} = \left| \frac{2^k}{2^{2k+1} + j2^{k+1} + 2^k} \right| \qquad \textbf{6}$$

$$e_{øvre} = \left| 1 - \frac{2^k + j + 1}{2^k} \cdot \frac{2^{k+1}}{2^{k+1} + 2j + 1} \right|$$

$$e_{øvre} = \left| 1 - \frac{2^{2k+1} + j2^{k+1} + 2^{k+1}}{2^{2k+1} + j2^{k+1} + 2^k} \right|$$

$$e_{øvre} = \left| - \frac{2^k}{2^{2k+1} + j2^{k+1} + 2^k} \right| = \left| \frac{2^k}{2^{2k+1} + j2^{k+1} + 2^k} \right|$$

Av utregningen over (ligning 24) ser vi det som tidligere er påstått at den relative feilen er lik i hver side av intervallet, men den viser også at den relative feilen minsker med økende j. Det vil si at det er størst feil ved j = 0, altså vil dette være verst mulig tilfelle, og da det tilfelle som må benyttes for å regne ut hvor mange riktige bit vi har.

$$e_{j=0} = \left| \frac{2^k}{2^{2k+1} + 2^k} \right|$$

$$e_{j=0} = \left| \frac{2^k}{2^k \left(2^{k+1} + 1\right)} \right| = \left| \frac{1}{2^{k+1} + 1} \right| \qquad \textbf{7}$$

$$\left| \frac{1}{2^{k+1} + 1} \right| < 2^{-(k+1)}$$

Denne gir altså k riktige bit (+1 implisitt bit som vist tidligere).

1.1.3 Lineære interpolasjon.

Dette er en mer korrekt tilnærming hvor det benyttes polynomer for å tilnærme seg funksjonen. Et polynom av nte grad vil da se slik ut:

$$z(Y) = c_0 + c_1 Y + c_2 Y^2 + \ldots + c_n Y^n \qquad \textbf{8}$$

C_i er konstanter.

Bare en lineær tilnærming vil her tilfredsstille kravene om hastighet å lite areal, men denne vil ha mindre korrekte bit en høyeregradstilnærminger. Dette gir tilnærmingen lik:

$$z(Y) = c_0 + c_1 Y \qquad \textbf{9}$$

Hvor c_0 og c_1 er konstanter. Hvis vi setter inn grenseverdiene til svar og verdimengden for vi følgende tilnærmingsformel:

$$\frac{1}{2} = c_0 + c_1 \cdot 2$$
$$1 = c_0 + c_1$$
$$c_0 = \frac{3}{2}$$
$$c_1 = -\frac{1}{2}$$
$$z(Y) = \frac{3}{2} - \frac{1}{2}Y$$
$$z(Y) = \frac{(3-Y)}{2}$$

10

Alt som trengs for å regne ut denne er en subtraksjon og et venstreskift.

Den relative feilen blir da.

$$e = \frac{\frac{1}{Y} - \frac{(3-Y)}{2}}{\frac{1}{Y}}$$
$$e = \frac{1}{2}Y^2 - \frac{3}{2}Y + 1$$

11

For å finne ut hvor den største feilen er må vi sjekke endepunktene og om det er noe maks/minimum innen intervallet (egentlig trengs det her ikke sjekke endepunktene etter som disse er benyttet til å lage tilnærmingen er feilen her lik 0):

$$e_{Y=1} = \frac{1}{2} \cdot 1^2 - \frac{3}{2} \cdot 1 + 1 = 0$$
$$e_{Y=2} = \frac{1}{2} \cdot 2^2 - \frac{3}{2} \cdot 2 + 1 = 0$$
$$e' = Y - \frac{3}{2}$$
$$e' = 0 \Rightarrow Y = \frac{3}{2}$$

12

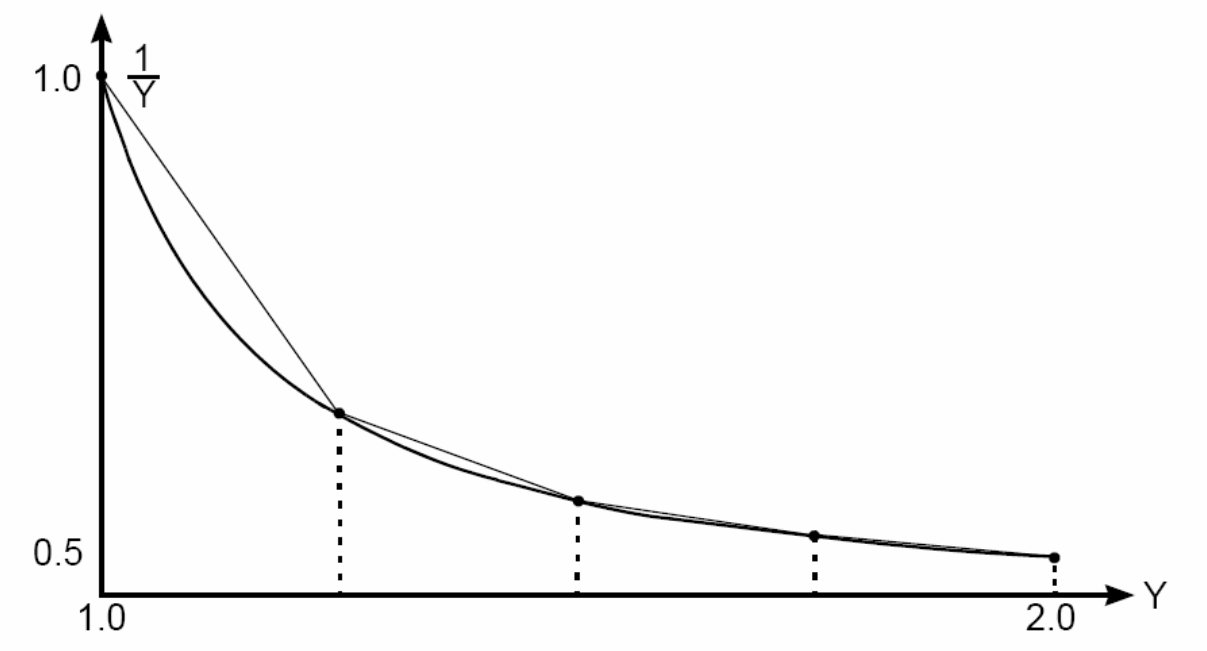
Feilen vil altså være størst med en Y på 1,5 og dette vil ha en feil på:

$$e_{Y=\frac{3}{2}} = \frac{1}{2} \cdot \left(\frac{3}{2}\right)^2 - \frac{3}{2} \cdot \frac{3}{2} + 1 = -\frac{1}{8}$$

13

Dette vil føre til at det er tilnærma 3 riktige bit (2^{-3}=1/8)

Det er mulig å gjøre denne metoden bedre med å dele den opp i flere intervaller. Dette vil grafisk se slik ut:



Figur 2: grafisk representasjon av Lineær interpolasjon
(denne figuren er hentet fra [5])

På samme måte som på stegvis tilnærming deler vi inn intervallet i et antall like store deler som er en potens av 2. Dette gir de samme intervallene som for stegvis tilnærming:

$$\left[\frac{\left(2^k + j\right)}{2^k}, \frac{\left(2^k + j + 1\right)}{2^k}\right)$$
$$j = \left\{0,1,\ldots,2^k - 1\right\}$$

Hvis vi kaller disse ende verdiene for Y_l og Y_h kan vi finne c_0 og c_1 forhvert intervall på samme måte som i ligning 28.

$$\mathrm{I}: \frac{1}{Y_l} = c_0 + c_1 Y_l$$

$$\mathrm{II}: \frac{1}{Y_h} = c_0 + c_1 Y_h$$

$$\mathrm{I} - \mathrm{II}: \frac{1}{Y_l} - \frac{1}{Y_h} = c_0 - c_0 + c_1 Y_l - c_1 Y_h$$

$$\frac{\left(Y_h - Y_l\right)}{Y_l Y_h} = c_1\left(Y_l - Y_h\right) = -c_1\left(Y_h - Y_l\right) \qquad \textbf{14}$$

$$c_1 = -\frac{1}{Y_l Y_h}$$

$$\frac{1}{Y_l} = c_0 - \frac{1}{Y_l Y_h} Y_l$$

$$c_0 = \frac{1}{Y_l} + \frac{1}{Y_h}$$

Tilnærmingsfunksjonen blir da.

$$z_0(Y) = \left(\frac{1}{Y_l} + \frac{1}{Y_h}\right) - \frac{1}{Y_l Y_h} Y \qquad \textbf{15}$$

Dette gir en relativ feil på (ut fra ligning 19).

$$e = \frac{\frac{1}{Y} - \left(\frac{1}{Y_l} + \frac{1}{Y_h} - \frac{1}{Y_l Y_h} Y\right)}{\frac{1}{Y}}$$

$$e = \left(\frac{1}{Y} - \left(\frac{Y_l + Y_h}{Y_l Y_h} - \frac{1}{Y_l Y_h} Y\right)\right) Y \qquad \textbf{16}$$

$$e = 1 - \left(\frac{Y_l Y + Y_h Y - Y^2}{Y_l Y_h}\right)$$

Utvider vi uttrykket til e med å sette inn grensene for intervallet Y_l og Y_h på samme måte som i ligning 24, ser vi at j vil ha større påvirkning i nevner enn i teller slik at den relative feilen blir mindre etter som j blir større. Feilen er altså størst i det første intervallet og den største feilen blir liggende midt i intervallet. Dette betyr at e_{maks} blir gitt av.

$$Y_l = 1, \quad Y_h = \frac{2^k + 1}{2^k}, \quad Y = \frac{2^{k+1} + 1}{2^{k+1}}$$

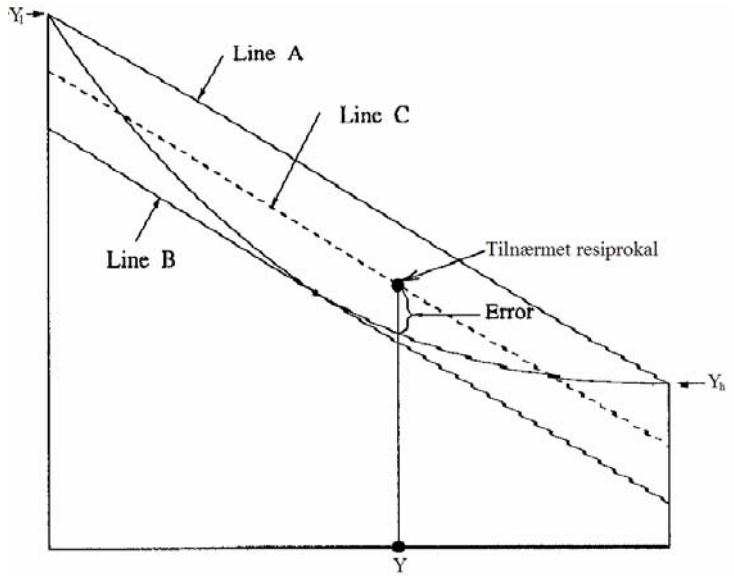
$$e = 1 - \left(\frac{1 \cdot \frac{2^{k+1}+1}{2^{k+1}} + \frac{2^k+1}{2^k} \cdot \frac{2^{k+1}+1}{2^{k+1}} - \left(\frac{2^{k+1}+1}{2^{k+1}} \right)^2}{1 \cdot \frac{2^k+1}{2^k}} \right)$$

$$e = 1 - \left(\frac{\frac{2^{2k+2} + 2^{k+2} + 1}{2^{2k+2}(2^k+1)}}{\frac{2^k+1}{2^k}} \right) \qquad \textbf{17}$$

$$e = \frac{2^{3k+2} + 2^{2k+2}}{2^{3k+2} + 2^{2k+2}} - \frac{2^{3k+2} + 2^{2k+2} + 2^k}{2^{3k+2} + 2^{2k+2}}$$

$$e = \frac{2^k}{2^{3k+2} + 2^{2k+2}} = \frac{1}{2^{2k+2} + 2^{k+2}} < 2^{-(2k+2)}$$

Det er dermed garantert 2k+1(+1 implisitt bit) antall korrekte bit ut fra tabell med lineær interpolarisasjon. Det er derimot lett å se ut fra Figur 3 at det her kan redusere feilen vis man tillater e å være både positiv og negativ. Vi ser dette i grafen under der feilen til den prikkete (linje C) linjen i midten er mindre enn linjen som tar utgangspunkt i endepunktene eller midt-punktet(linje A eller B).



Figur 3: Lineære tilnærming
(denne figuren er hentet fra [13])

Linje C er vertikale midtpunktet mellom linje A og linje B det er vist i [27] at dette gir den minimale feilen. Tilnærmingsfunksjonen blir da gitt av.

$$z_0(Y) = \frac{8(Y_l + Y_h) - 8Y}{Y_l^2 + 6Y_l Y_h + Y_h^2} \qquad \textbf{18}$$

c_0 og c_1 er da gitt ved.

$$c_0 = \frac{8(Y_l + Y_h)}{Y_l^2 + 6Y_lY_h + Y_h^2}$$
$$c_1 = \frac{8}{Y_l^2 + 6Y_lY_h + Y_h^2} \quad \textbf{19}$$

Den største feilen vil fortsatt ligge i midtpunktet på samme måte som når man tok utgangspunkt i endepunktene som linje a i Figur 3. Så ved samme framgangsmåte som over finnes maks feil til å være.

$$e_{\max} = \frac{1}{2^{2k+3} + 2^{k+3} + 1} < 2^{-(2k+3)} \quad \textbf{20}$$

Den gir altså et bit ekstra i presisjon.

1.2 Bipartitetabeller:

Starter med et eksempel på hvordan tabellen fungerer for å gi litt forståelse for hvorfor jeg bringer opp de forskjellige temaene som kommer senere i dette underkapitelet:

Et eksempel her er: tatt fra [13]
Vi har j+2=3k+1 bit inn i tabellen og 3k-1 bit ut av tabellen (g er da lik 2 i dette tilfellet.)

Det som skjer da er:

1. innbittene (3k+1) blir partisjonert inn i tre deler, høy(y_h), middels(y_m) og lav(y_l) med følgende størrelser k+1,k,k
2. de første 2k+1 bittene (y_h og y_m) indekserer den posetive tabellen P. Denne tabellen gir ut et svar på 3k+1 bit.
3. De 2k+1 bittene av høy og lav del (y_h og y_l) indekserer den negative tabellen N. Denne tabellen gir et utgangs signsal på k+1 bit.
4. Det er til slutt en borrow save funksjon som benytter resultatene fra de to tabellene til å lage et resultat på 3k-1 bit.

1.2.1 Forklaring av algoritme

Bipartite tabeller tar utgangspungt i at nevneren deles opp i tre deler som så ommskrives og forenkles. Under vises dette for et resiprokal (1/y) etter som det er dette oppgaven videre benytter.

$$\frac{1}{y} = \frac{1}{y_h + y_m + y_l} = \frac{1}{y_h + y_m} - \frac{y_l}{(y_h + y_m)(y_h + y_m + y_l)} \approx \frac{1}{y_h + y_m} - \frac{y_l}{y_h} \quad \textbf{21}$$

Benytter så y_h og y_m til oppslag i tabell P og y_h og y_l til oppslag i tabell N

I de bipartitetabeller som er beskrevet videre i kapitelet antas det at resiprokalets nevner (y) er i normalisert i området $1 \leq y < 2$ og deretter trunkert til j + g bits lengde. Der j er antall bit som er riktige[1] ut av tabellen. Når y er normalisert på denne måten og lengden av de forskjellige delene er gitt av h, m og l antall bitt, kan vi skrive y på følgende måte.

$$y = 1.b_1 b_2 b_3 \ldots b_h b_{h-1} \ldots b_{h-m} b_{h-m-1} \ldots b_{h-m-l} \quad \textbf{22}$$

Der det siste bittet i y vil som nevnt over også være lik:

$$b_{h-m-l} = b_{j+g} \quad \textbf{23}$$

Utrykkene for y_h, y_m og y_l blir da gitt ved:

$$\begin{aligned} y_h &= 1.b_1 b_2 \ldots b_h \\ y_m &= 0.b_{h-1} b_{h-2} \ldots b_{h-m} \cdot 2^h \\ y_h &= 0.b_{h-m-1} b_{h-m-2} \ldots b_{h-m-l} \cdot 2^{h+m} \end{aligned} \quad \textbf{24}$$

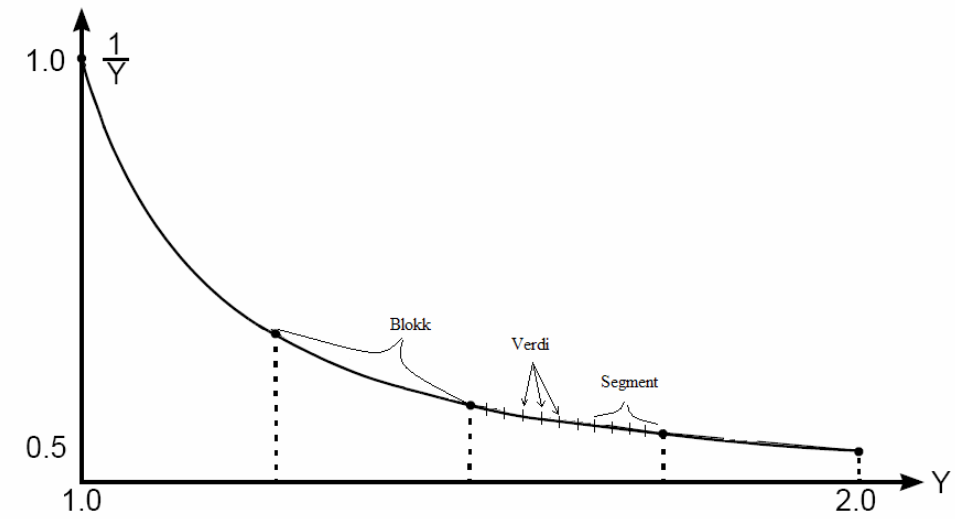
På grunn av at vi har j+g antall bit inn er det logisk å dele opp tabellen i 2^{j+g} verdier, disse verdiene deler vi så opp i grupper for å kunne systematisere tabellene og algoritmen for konstruksjonen av tabellene. h, m og l er allerede definert så derfor brukes disse videre til å definere delene av tabellen. Tabellen deles opp i blokker, segmenter og verdier

BLOKK: Intervallet til y ([1,2)) deles opp i 2^h antall blokker.

SEGMENTER: hver blokk er bygget opp av 2^m antall segmenter.

VERDIER: hvert segment inneholder 2^l antall verdier, en verdi er også et intervall med underverdier.

Grafisk vil dette se slik ut:



Figur 4: oppdeling av resiprokalfunksjonen

[1] Ritig betyr at den relative feilen er mindre enn det bitets verdi

Intervallene til de forskjllige definisjonene over kan vises matematisk slik:

Bredden til en blokk er $1/2^h$ etter som vi har 2^h blokker

$$\left[1+\frac{a}{2^h},1+\frac{a+1}{2^h}\right)= \left[\frac{2^h+a}{2^h},\frac{2^h+a+1}{2^h}\right) \quad \textbf{25}$$

Hvor "a" er nummeret på blokken, hvor blokk 0 starter i 1 og "a" er i intervallet $0 \rightarrow 2^h\text{-}1$

Bredden på et segment $1/2^{h+m}$, da det er 2^m segmenter for hver blokk.

$$\left[\frac{2^h+a}{2^h}+\frac{b}{2^{h+m}},\frac{2^h+a}{2^h}+\frac{b+1}{2^{h+m}}\right)= \left[\frac{2^{h+m}+2^m a+b}{2^{h+m}},\frac{2^{h+m}+2^m a+b+1}{2^{h+m}}\right) \quad \textbf{26}$$

"b" er segmentnummeret innenfor blokken og ligger i intervallet $0 \rightarrow 2^m\text{-}1$

Tilsvarende er bredden på mellom hver verdi $1/2^{h+m+l}$.

$$\left[\frac{2^{h+m}+2^m a+b}{2^{h+m}}+\frac{c}{2^{h+m+l}},\frac{2^{h+m}+2^m a+b}{2^{h+m}}+\frac{c+1}{2^{h+m+l}}\right)= \left[\frac{2^{h+m+l}+2^{m+l} a+2^l b+c}{2^{h+m+l}},\frac{2^{h+m+l}+2^{m+l} a+2^l b+c+1}{2^{h+m+l}}\right) \quad \textbf{27}$$

"c" ser verdinummeret innen for segmentet og ligger i intervallet $0 \rightarrow 2^l\text{-}1$

1.2.2 Reduksjon av tabellstørrelse.

Ut fra disse intervallene er det mulig å finne ut hvor mange bit tabell N må være på for at den ønskelige nøyaktigheten skal oppnås. I punktet om stegvis tilnærming er det vist at hvis vi benytter midtpunktet til å beregne resiprokalet vil vi få antall riktige bit ut lik det antall vi hadde inn pluss en. Antall bit inn i tabell P er h+m, det er derfor h + m + 1 riktige bit i tabell P. Antall ønskelige riktige bit ut fra den bipartitetabellen er j = h+m+l-g bit, men på grunn av definisjonen til antall riktige bit må vi legge til et bit for å få det korrekt matematisk, altså h+m+l-g+1. På grunn av at blokkene er delt opp slik at de er like store vil den største feilen ligge i den første blokken[2] (blokk 0).

[2] Det selvsagt er mulig å lage algoritmer for å lage tabellene slik at dette ikke stemmer men i alle tilfellene som blir tatt opp i denne rapporten og alle de tilfellene som jeg har kommet over i dette studiet er dette riktig.

Største avstand som må justeres med tabell N er da i blokk 0 mellom første å siste verdi i segment 0[3]:

$$1-\frac{2^{h+m+l}}{2^{h+m+l}+2^{m+l}\cdot 0+2^{l}\cdot 0+\left(2^{l}-1\right)}=$$
$$1-\frac{2^{h+m+l}}{2^{h+m+l}+2^{l}}=$$
$$\frac{2^{h+m+l}+2^{l}-2^{h+m+l}}{2^{h+m+l}+2^{l}}=$$
$$\frac{2^{l}}{2^{l}\left(2^{h+m}+1\right)}=\frac{1}{\left(2^{h+m}+1\right)}<2^{-(h+m)} \qquad \textbf{28}$$

Minste avstanden som skal justeres med tabell N er avstanden mellom den siste verdien og en halv[4]. Siste riktige verdi kan skrives som: $1/(2-(1/2^{h+m+l-g+1}))$

$$\frac{1}{2-\frac{1}{2^{h+m+l-g+1}}}-\frac{1}{2}=$$
$$\frac{2^{h+m+l-g+1}}{2^{h+m+l-g+2}-1}-\frac{1}{2}= \qquad \textbf{29}$$
$$\frac{1}{2^{h+m+l-g+3}-2}>2^{-(h+m+l-g+3)}$$

Altså bittene som skal være med ligger mellom h+m+l-g+3 og k+m, dette tilsvarer:

$$h+m+l-g+3-\left(h+m\right)=l-g+3 \qquad \textbf{30}$$

Tabell N må dermed være l-g+3 dyp for å tilfredsstille kravene om antall riktige bit.

Som det blir nevnt i fotnotene er det redundans i disse beregningene slik at resultatet svaret er litt for høyt men etter som det ikke er en hel bit for høyt er det tatt med for det viser på en enkel måte at bit dybden benyttet i denne oppgaven stemmer. En mer presis måte ville da være å beregne den største avstanden, mellom midtpunktet til første å andre verdi og midtpunktet til andre å tredje verdi og minste avstanden mellom midtpunktet til nest siste verdi og sisteverdi og sisteverdi og 0.5. Dette ville likevel ikke gi et eksakt bilde på hvordan stegene vil være i praksis grunnet algoritmen som benyttes til å lage tabellene, men det vil bli nærmere beskrevet ved gjennomgang av denne algoritmen.

Hva er så den optimale lengde fordelingen mellom h, m og l. For å kunne undersøke dette trenges først å sette opp hvordan tabellene er bygget opp. Tabell P er en h+m bit inn j+g bit ut tabell. Denne tabellen inneholder ikke den redundante bitten b_1 så denne må bli lagt til senere, vi skal også se at det blir lagt til et ekstra bit på slutten av denne verdien før den går blir trukket fra verdien i tabell N. Dette er for å opprettholde tilstrekkelig presisjon etter avrundinger.

[3] Dette er ikke den eksakte største feilen men den er større en den største slik at den vil gi et litt redundant svar.
[4] Også dette svaret inneholder redundans.

Tabell N er en h+l bit inn l-g+3 bit ut (ligning 51) tabell. Tabell N trenger bare ekstra nuller lagt til før subtraksjonen med Tabell P. Den totale lengden av de to tabellene er gitt ved

$$totsize = 2^{h+m}(h+m+l) + 2^{h+l}(l-g+3) \quad \textbf{31}$$

Ut fra ligning 52 er det lett å se at begge sider av pluss tegnet vil ha en kvadratisk økning. For det første uttrykket er dette helt korrekt ettersom h+m+l er en konstant, mens det andre uttrykket vil derimot også ha en ekstra komponent ved at l kan forandres. Det er derimot 2^{h+l} som vil styre utviklingen etter som l-g+3 er liten i forhold. Ut fra dette må de to uttrykkene være like stor for å oppnå minst mulig tabellstørrelse. Etter det resonnementet er det jo mulig å ha h stor å l og m små men dette vil føre til at l-g+3 blir veldig mye mindre enn h+m+l og dette vil trekke opp tabellstørrelsen. Jeg har i denne oppgaven valgt å benytte samme fordelig som i [13], fordelingen der går ut på å heltallsdele antallet inngangs bit på 3 slik at vi for en k[5] men etter som tre forskjellige antall inngangsbit vil gi samme k legges det til eller trekkes fra ett bit på m. Fordelingen blir da.

$$k+1, k+u, k$$
$$k = \text{int}((j+g)/3)$$
$$u = -1 \; hvis \; (j+g) \bmod 3 = 0$$
$$u = 0 \; hvis \; (j+g) \bmod 3 = 1$$
$$u = 1 \; hvis \; (j+g) \bmod 3 = 2$$

Int() betyr at tallet avrundes til nærmeste heltall.

De tre fordelingene blir da: (k+1,k-1,k), (k+1,k,k), (k+1,k+1,k). Det er derimot ganske enkelt å vise til at en fordeling på (k,k,k), (k+1,k,k), (k+1,k,k+1) vil gi mindre tabeller, dette er henvist til i [15]. Dessverre ble det ikke tid til å implementere disse å prøve dem ut. Fordelingen over vil føre til en gjennomsnittelig økning på 1,5 ulp per fordobling av tabellstørrelsen, det vil si at bipartitettabeller oppnår 3 flere riktige bit på en fire dobling mens enkelt tabeller øker med 8 ganger for en økning på 3 riktige bit. Dette vil si at for hver 3 bit vi øker j med vil kompresjonsfaktoren til den bipartitetabellen øke med 2.

Hvor mange guard bit som er ønskelig å ha med kommer an på hva man ønsker, men videre i denne oppgaven vil g hovedsakelig være lik 2. Dette fordi det gir resultater nært den optimale oppslagstabellen uten å påvirke størrelsen for mye. Under vises dette med en tabell hentet fra[13]

[5] Denne må ikke forveksles med k benyttet tidligere i raporten for å beskrive antall bit.

j	$j+1$-bits-in,j-bits-out Optimal ROM table			$j+2$-bits-in,j-bits- out Optimal ROM table			$j+2$-bits-in,j-bits-out Bipartite table		
	Table size (Kbytes)	Percent not RN	Max error (ulps)	Table size (Kbytes)	Percent not RN	Max error (ulps)	Table size (Kbytes)	Percent not RN	Max error (ulps)
10	2.5	12.453	0.999	5	6.259	0.722	0.6875	8.628	0.826
11	5.5	12.710	≈ 1	11	6.126	0.736	1.125	8.514	0.857
12	12	12.694	≈ 1	24	6.103	0.743	2.0625	8.438	0.853
13	26	12.511	≈ 1	52	6.217	0.746	3.375	8.638	0.865
14	56	12.501	≈ 1	112	6.248	0.748	5.5	8.616	0.901
15	120	12.455	≈ 1	240	6.228	0.747	10	8.578	0.904
16	256	12.522	≈ 1	512	6.259	0.748	16	8.677	0.919

Figur 5: oversiktstabell over bipartite tabeller mot optimale oppslagstabeller

Figur 5 viser at j+2 in, j ut bipartite tabeller ligger i nærheten av j+2 in, j ut optimal tabell (enkeltabell).

1.2.3 D. Das Sarma's algoritme for konstruksjon av bipartitettabeller.

Denne algoritmen er hentet fra [13].

Før algoritmen kan forklares defineres en midtpunktsfunksjon som vil bli benyttet i denne algoritmen. Midtpunktsfunksjonen finner midtpunktet mellom to verdier, for som det tidligere i denne rapporten er bevist er dette det resiprokalet som gir minst relativ feil og dermed flest antall riktige bit. Som vist under stegvis tilnærming er dette resiprokalet lik (ligning 23):

$$\frac{2^{k+1}}{2^{k+1}+2j+1} \qquad \textbf{32}$$

Merk: k og j her er antall bit inn og hvilke verdi som midtpunktet beregnes til ikke antall bit inn del på tre og antall riktige bit ut som benyttet ovenfor. Vi kan omskrive denne k'en slik at den passer med k'en benyttet rett ovenfor eller med h,m og l. Det samme kan vi gjøre med J'en slik at formelen blir seende slik ut:

$$\frac{2^{3k+u+2}}{2^{3k+u+2}+2\left(y_h+y_m 2^{k+1}+y_l 2^{2k+u+1}\right)+1} = \frac{2^{h+m+l+1}}{2^{h+m+l+1}+2\left(y_h+y_m 2^{h}+y_l 2^{h+m}\right)+1} \qquad \textbf{33}$$

Dette kan også omskrives slik at det blir likt det som er benyttet i [13].

$$\frac{2^{3k+u+1}}{2^{3k+u+1}+\left(y_h+y_m 2^{k+1}+y_l 2^{2k+u+1}\right)+\frac{1}{2}} \qquad \textbf{34}$$

Algoritme 1. [midtpunktsberegning]

Stimuli:

Midtpunktfunksjonen tar altså inn enten (y_h, y_m, y_l, k, u) eller (y_h, y_m, y_l, h, m, l). Etter som vi vil benytte hovedsakelig k i resten av rapporten velges det første alternativet, men k og u vil ikke bli skrevet opp i algoritme forklaringen er at disse alltid er det samme.

Respons:

Funksjonen returnerer svaret at ligning 54.

Heretter vil midtpunktsfunksjonen bli kalt midRes(y_h, y_m, y_l).

Selve algoritmen for generering av bipartitetabeller er beskrevet under.

Algoritme 2. [bipartittabell generering]

Stimuli:

Heltallene k og u.

Respons:

De bipartitetabellene P og N.

Steg1: [generering av Tabell P]

for $y_h = 0$ til $2^{k+1}-1$

L1: firstspread(y_h) = midRes(y_h, 0, 0) - midRes(y_h, 0, 2^k-1)

L2: lastspread(y_h) = midRes(y_h, 2^{k+u}, 0) - midRes(y_h, 2^{k+u}, 2^k-1)

L3: averagespread(y_h) = $\frac{firstspread(y_h) + lastspread(y_h)}{2}$

for $y_m = 0$ til 2^{k+u}

L4: spread(y_h, y_m) = midRes(y_h, y_m, 0) - midRes(y_h, y_m, 2^k-1)

L5: adjust(y_h, y_m) = $\frac{averagespread(y_h) - spread(y_h, y_m)}{2}$

L6: TableP(y_h, y_m) = midRes(y_h, y_m, 0) + adjust(y_h, y_m)

L7: Rund av ned TabelP(y_h, y_m) til 3k+u+1 bit

end

end

Steg 2 [generering av Tabell N]

for $y_h = 0$ til $2^{k+1}-1$

for $y_l = 0$ til 2^k-1

L8: firstdiff(y_h, y_l) = midRes(y_h, 0, 0) - midRes(y_h, 0, y_l)

L9: lastdiff(y_h, y_l) = midRes(y_h, 2^{k+u}, 0) - midRes(y_h, 2^{k+u}, y_l)

L10: TableN(y_h, y_l) = $\frac{firstdiff(y_h, y_l) + lastdiff(y_h, y_l)}{2}$

L11: Rund av til nærmeste TableN(y_h, y_l) til k+1 bit

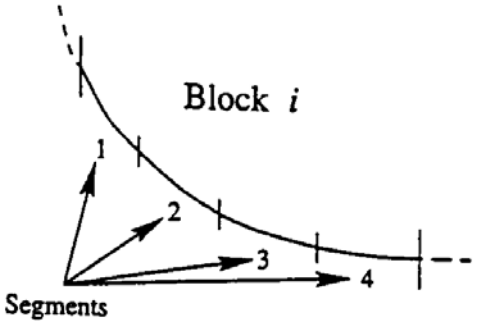
end

end

Forklaring på steg 1.

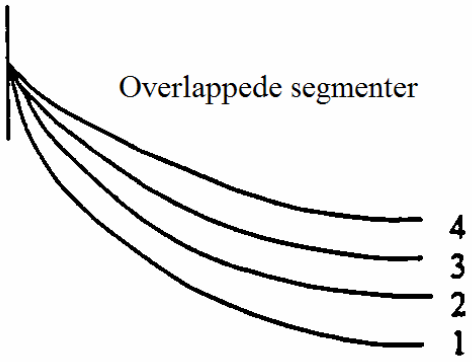
Etter som feilen er størst på begynnelsen av resiprokalfunksjonen kan man øke nøyaktigheten ved at men fordeler feilen utover hele blokka. Dette gjøres ved at man finner den gjennomsnittlige spredningen mellom første og siste midtverdi i hvert segment pr blokk. Dette igjen gjøres ved at man finner spredningen i første (L1) og siste(L2) segment i en blokk og deretter midler denne(L3). Så finner man sprednningen i det segmentet tabellen genereres for der å da(L4), deretter midles dette med den gjennomsnittelige spredningen(L5). Denne middelverdien blir så lagr midtverdiene(L6) slik at feilen smøres utover.
Tilslutt avrundes tabellen ned til det riktige antall bit(L7)(3k+u-1). Det skal også legges til en ekstra ener bakerst, men etter som denne er i alle verdiene i tabellen, lagres den ikke.

Forklaring steg 2.
Slik denne algoritmen er lagtopp benyttes den samme delen av tabell N for ei hel blokk. Dette steget finner derfor den gjennomsnittelige avstanden mellom den første verdien og alle de andre verdiene innen alle segmentene i en blokk. Dette gjøres ved at avtanden fra første midtverdi beregnes til alle andre verdier innen første segment(L8) og siste segment(L9). Dette midles(L10) så slik at den gjennomsnittelige verdien finnes, denne middelverdien er Nverdien som skal benyttes. Tilslutt avrundes tabell N til riktig antall bit, det vil si k+1 her. En grafisk forklaring på hvordan algoritmen fungerer er vist under.



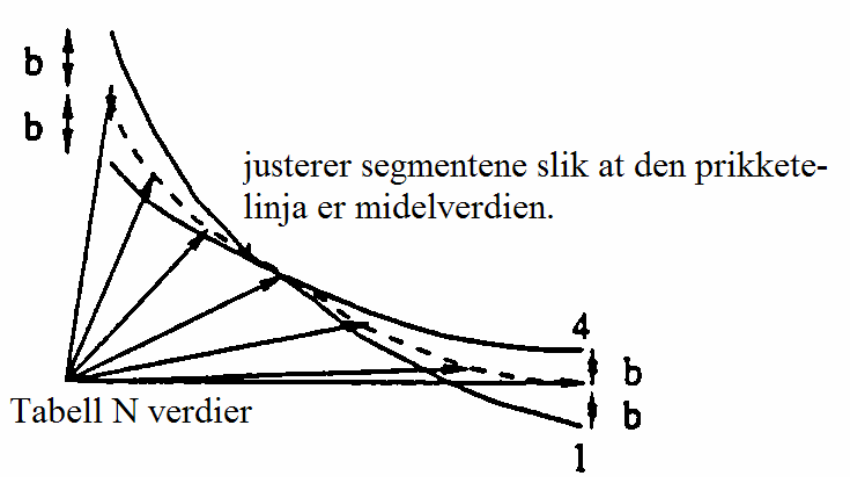
Figur 6:Blokk inndeling
(dette bildet er hetet fra [13])

Figur 6 illustrerer hvordan hver blokk er delt opp av segmenter.



Figur 7: overlappede segmenter
(dette bildet er hetet fra [13])

Figur 5 illustrerer at alle segmentene i en blokk er blitt plassert over hverandre.



Figur 8: Midling av segmentene
(dette bildet er hetet fra [13])

Figur 8 illustrerer hvordan segmentene midles og vi får N verdier langs middelverdien. Selv om dette ikke er direkte det samme som skjer med P verdiene er det representativt for hvordan averagespred blir til. Så det viser hvordan man fordeler feilen utover hele blokken.

1.2.3.1 Feilanalyse

For å bevise at denne algoritmen gir et tilstrekkelig antall riktige bit, kan vi starte med å bevise hvor mange riktige bit den uavrunnet tabellen gir. Dette beviset er hentet fra [13].
Dette beviset tar først for seg hvor mange riktige bit tabell P har. Dette kan lett bevises med å se på beviset under stegvis tilnærming, hvor det bevises at ved bruk av midtpunkt verdien vil antall riktige bit bli lik en mer enn antall bit i innsignalet, innsignalet her er 3k+u+1 altså får vi 3k+u+2 riktige bit ut. Antallet bit som skal være riktige til slutt er 3k+u altså vil P verdien være minimum 1/4ulp riktig. Neste steg i beviset er å vise hvor mye feil ekstra som blir innført ved at midtverdien og N verdier justeres, da disse virker sammen på justeringen. For å gjøre dette må vi analysere adjusten og midle feilen innfør i N verdine, dette med hensyn på første eller siste segment, etter som algoritme 2 er konstruert slik at maks feil blir liggende i endene av blokkene, det er blokk 0 som vil ha den største feilen så det er denne som må undersøkes. Her er det valgt å bruke førstesegment. Først finnes feilen ved adjust

$$adjust(y_h, y_m) = \frac{averagespread(y_h) - spread(y_h, y_m)}{2}$$

$$averagespread(y_h) = \frac{firstspread(y_h) + lastspread(y_h)}{2}$$

$$adjust(y_h, y_m) = \frac{\frac{firstspread(y_h) + lastspread(y_h)}{2} - firstspread(y_h)}{2} = \quad \textbf{35}$$

$$\frac{firstspread(y_h) + lastspread(y_h) - 2\,firstspread(y_h)}{4} =$$

$$\frac{1}{4}(lastspread(y_h) - firstspread(y_h))$$

Maks feilen blir som sakt i blokk 0, så den største feilen oppstår i

$$\frac{1}{4}(lastspread(0) - firstspread(0)) \quad \textbf{36}$$

For å finne ut hvor mye feil som oppstår ved justeringen som foretas på N verdiene tas utgangspunkt i steg 2 i algoritme 2.

$$TabellN = \frac{firstdiff(y_h, y_l) + lastdiff(y_h, y_l)}{2} \quad \textbf{37}$$

Den største feilen vil her oppstå i forskjellen mellom første verdi og siste verdi i blokk 0, forskjellen vil da bli.

$$diff = \frac{firstdiff(0, 2^k - 1) - lastdiff(0, 2^k - 1)}{2} \quad \textbf{38}$$

Som igjen er identisk lik.

$$diff = \frac{1}{2}(firstspread(0) - lastspread(0)) \quad \textbf{39}$$

Legger vi dette sammen får vi

$$adjust(y_h, y_m) - diff = \frac{1}{4}(fistspread(y_h) - lastspread(y_h)) \quad \textbf{40}$$

Setter så inn riktige verdier for de forskjellige delene. Grunnen til at alle verdiene ganges med 2^{3k} er at ulp vil da stå i nulte posisjonen altså fås direkte ut antallet ulp som er feil.

$$\frac{1}{4}\left(\left(\frac{2^{3k+1}}{2^{3k+1}+\frac{1}{2}}2^{3k}-\frac{2^{3k+1}}{2^{3k+1}+2^{k}-\frac{1}{2}}2^{3k}\right)-\left(\frac{2^{3k+1}}{2^{3k+1}+2^{2k}-2^{k}+\frac{1}{2}}2^{3k}-\frac{2^{3k+1}}{2^{3k+1}+2^{2k}-\frac{1}{2}}2^{3k}\right)\right)=$$

$$\frac{1}{4}\left(\frac{\left(2^{k}-1\right)2^{3k+1}}{\left(2^{3k+1}+\frac{1}{2}\right)\left(2^{3k+1}+2^{k}-\frac{1}{2}\right)}2^{3k}-\frac{\left(2^{k}-1\right)2^{3k+1}}{\left(2^{3k+1}+2^{2k}-2^{k}+\frac{1}{2}\right)\left(2^{3k+1}+2^{2k}-\frac{1}{2}\right)}2^{3k}\right)=$$

$$\frac{1}{4}\frac{2^{k}\left(2^{k}-1\right)\left(2^{k}-1\right)\left(2^{3k+2}+2^{3k}\right)2^{3k+1}}{\left(2^{3k+1}+\frac{1}{2}\right)\left(2^{3k+1}+2^{k}-\frac{1}{2}\right)\left(2^{3k+1}+2^{2k}-2^{k}+\frac{1}{2}\right)\left(2^{3k+1}+2^{2k}-\frac{1}{2}\right)}2^{3k}< \qquad \textbf{41}$$

$$\frac{1}{4}\frac{2^{3k}\left(2^{3k+2}+2^{3k}\right)2^{3k+1}}{2^{3k+1}\cdot 2^{3k+1}\left(2^{3k+1}+2^{2k}-2^{k}\right)2^{3k+1}}2^{3k}<$$

$$\frac{1}{16}\frac{2^{3k+2}+2^{2k}}{2^{3k+2}+2^{2k}-2^{k}}<\frac{1}{8}\frac{2^{3k+1}+2^{2k-1}}{2^{3k+1}+2^{2k}-2^{k}}<$$

$$\frac{1}{8}\frac{2^{3k+1}+2^{2k-1}}{2^{3k+1}+2^{2k-1}}<\frac{1}{8}$$

Vi får altså at maksimum feil uten av runding er 1/4ulp fra midtpunktsverdien og justeringen fører til 1/8ulp feil, maks uavrundet feil for svaret ut av den bipartitetabellen blir da summen av disse som er 3/8ulp.

Det er som kjent ikke mulig å ha uendelig mange siffer med når man skal gjøre beregninger elektronisk noe som fører til at man må gjøre avrundinger. Det er derfor nødvendig å finne maks avrundingsfeil slik at den totale feilen kan bestemmes.

Størrelsen på avrundingsfeilen beregnes under. Disse bevisene er hentet fra [13].

Definerer først $RD_g(y)$ som rund ned til g antall bit etter ulp. Dette vil forårsake at all feil vil være positiv og maks 2^{-g}. På grunn av at feilen alltid er positiv vil feilen kunne halveres men da blir den både negativ og positiv. Dette gjøres ved å legge til en ekstra ener etter de g bittene. Dette gir dermed at $RD_g(y)+2^{-(g+1)}$ har en maks feil på $2^{-(g+1)}$. Dette er det bitet som legges til i subtraksjonen mellom P og N verdiene for å oppnå denne effekten.

Definerer så $RN_g(y)$ som rund av til nærmeste g antall bit etter ulp. Når man her benytter seg av definisjonen at alt som større eller lik $2^{-(g+1)}$ skal rundes opp å ellers ned er det gitt at denne vil ha en maksimal feil på $2^{-(g+1)}$.

Definerer til slutt en $z = RN_g(e) + RD_g(f) + 2^{-(g+1)}$, forskjellen mellom z og avrundet z blir da.

$$RN_0\left(RN_g(e)+RD_g(f)+2^{-(g+1)}\right)=RN_0\left(RN_g(e)+RD_g(f)\right)$$
$$\left|z-RN_0(z)\right|\le \tfrac{1}{2}+2^{-(g+1)} \qquad \textbf{42}$$

RN_0 er da avrunding til nærmeste uten ekstra bitt. Dette fører til at avrundingsfeilen er på maks $\frac{1}{2}+2^{-(g+1)}$ som i dette tilfellet tilsvarer $\frac{5}{8}$. Den totale feilen blir da summen av den uavrundede feilen og avrundningsfeilen som er $\frac{3}{8}+\frac{5}{8}=\frac{8}{8}=1\,\text{ulp}$. Dette beviser at den bipartitetabellen holder seg innen en ulps kriteriet.

1.2.4 Flere typer bipartitetabeller.

Ut fra underkapitelet om Newton-Raphson metoden ser vi at prosessen som skal itereres inne holder en subtraksjon og multiplikasjoner. Hvis man skal bygge opp kretsen i en ASIC å kretsen bare skal benyttes til denne oppgaven eller at det er andre opprasjoner som også inneholder subtraksjon og multiplikasjon, kan det være ønskelig å benytte seg av ”borrow save multiplikator”[6] da vi kan tjene tid i utførelsen av subtraksjonene. Dette vil samtidig føre til at subtraksjonen i den bipartite tabellen også kan gjøres på ”borrow save” slik at den nesten ikke vil ta noe tid. Dette er grunnen til at akkurat den typen bipartittabell som er benyttet over ble valgt i denne oppgaven. Som det blir vist i ligning 63 er det mulig å konstruere tabellen slik at man oppnår en addisjon istedenfor subtraksjon. Dette kan være ønskelig hvis kommende ittrasjoner skal inneholde addering i stede for subtraksjon, for da kan ”carry save multiplikatorer” benyttes. Det er ikke tatt med like utfyllende bevis på at dette vil gi likt nummer av korrekte bit, men ved å følge samme framgangs måte som over vil det kunne vises at også disse metodene vil opfylle kravene.

Carry save tilnærming.(dette er hentet fra [15])

$$\frac{1}{y} = \frac{1}{((y_h + y_m)+1)} + \frac{1 - y_l}{(y_h + y_m + y_l)((y_h + y_m)+1)} \qquad \textbf{43}$$

At dette stemmer kan enkelt vises ved å.

$$\begin{aligned} &\frac{1}{((y_h + y_m)+1)} \cdot \frac{(y_h + y_m + y_l)}{(y_h + y_m + y_l)} + \frac{1 - y_l}{(y_h + y_m + y_l)((y_h + y_m)+1)} = \\ &\frac{(y_h + y_m + y_l)+1 - y_l}{(y_h + y_m + y_l)((y_h + y_m)+1)} = \frac{(y_h + y_m)+1}{(y_h + y_m + y_l)((y_h + y_m)+1)} = \\ &\frac{1}{(y_h + y_m + y_l)} = \frac{1}{y} \end{aligned} \qquad \textbf{44}$$

Ledd nummer 2 (tabell N, selv om den her ikke er negativ) kan forenkles på samme måte som er gjort i starten på dette underkapitelet (ligning 42) slik at tabell N kan velges av y_h og y_l. ledd en avhenger også her bare av y_h og y_m så disse vil styre Tabell P.

Som tidligere vist kan man oppnå høyere nøyaktighet med å ta midtpunktsverdien å dette kan oppnås ved å legge til en ener bak $y_h + y_m$. Dette vil kunne uttrykke matematisk på denne måten.

$$\frac{1}{y} = \frac{1}{((y_h + y_m) + 2^{-(h+m+1)})} + \frac{2^{-(h+m+1)} - y_l}{(y_h + y_m + y_l)((y_h + y_m) + 2^{-(h+m+1)})} \qquad \textbf{45}$$

At dette stemmer kan bevises på akkurat samme måte som over. Samme fremgangsmåte som over vil også vise at tabell P (ledd 1) vil kunne styres av y_h, y_m og tabell N vil kunne styres av y_h og y_l.

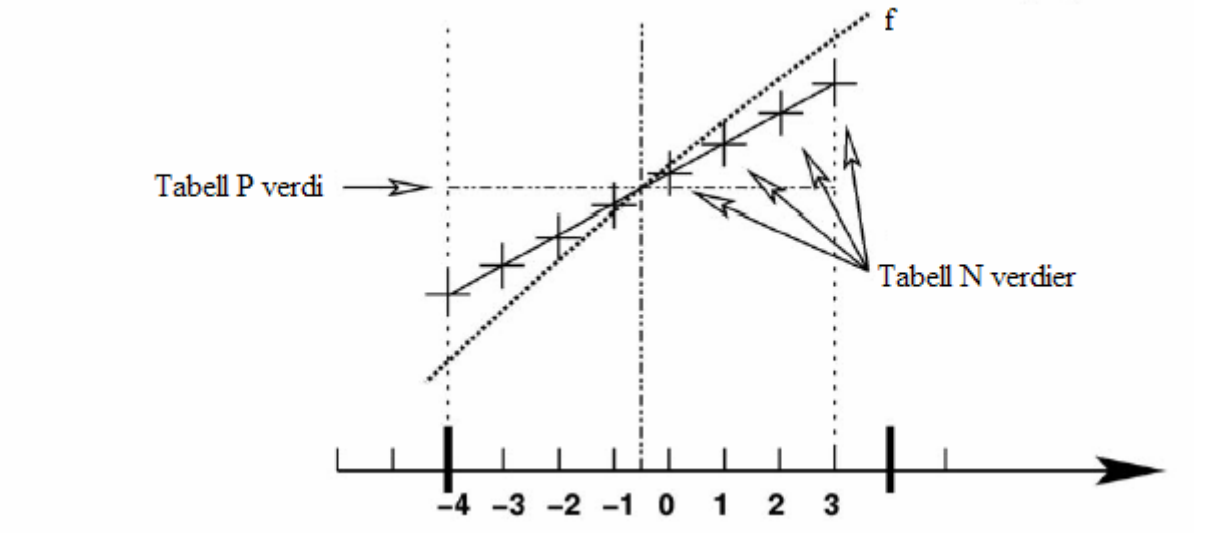
[6] Med borrow save og carry save multiplikatorer menes her multiplikatorer som tar imot innsignaler på borrow/carry save form

Formelen over (ligning 65) kan videre forbedres ved at også ledd 2 blir midtpunkts justert. Dette gjøres ved at vi trekker fra $\left(\frac{1}{2} - 2^{-(l-h-m)}\right)$ fra ledd to og legge det til ledd en. Dette gir følgende formel.

$$\frac{1}{y} = \frac{1}{\left((y_h + y_m) + 2^{-(h+m+1)} + 2^{-l}\right)} + \frac{2^{-(h+m+1)} - y_l + 2^{-l}}{(y_h + y_m + y_l)\left((y_h + y_m) + 2^{-(h+m+1)} + 2^{-l}\right)} \qquad \textbf{46}$$

1.2.5 Symetrisk bipartitetabeller.

Det er mulig å utnytte symmetrien i de bipartitetabellene ved at man i motsetning av gjøre som vist tidligere, med ta utgangspunkt i resiprokalet i den ene enden av segmentet, og ut fra denne enten legger til eller trekker fra for å finne det ønskede resiprokalet, kan man legge resiprokalet til midt i segmentet inn i tabell P og da legge til eller trekke fra for å finne det riktige resiprokalet. Dette kan representeres grafisk som i Figur 9



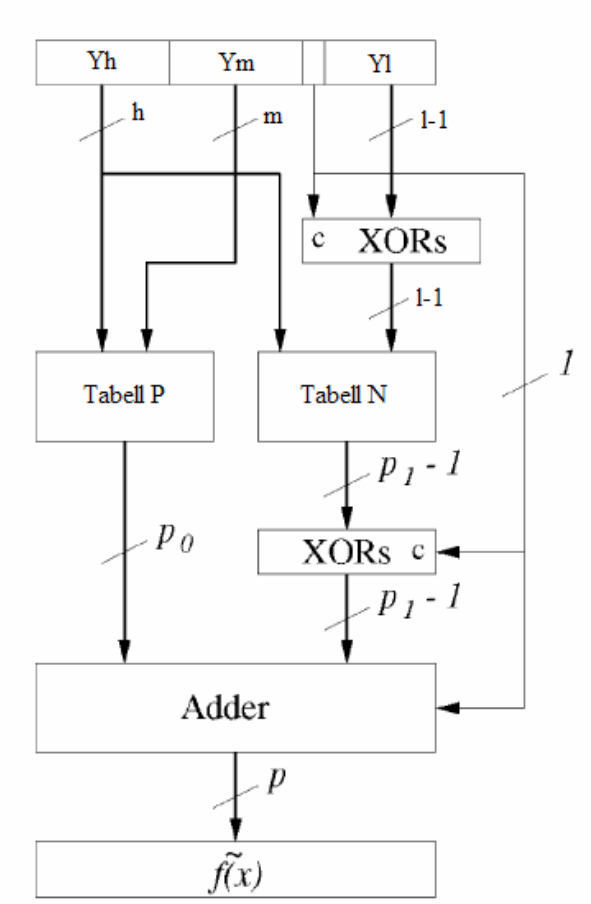
Figur 9: symetriske N verdier
(denne er hentet fra [17])

Grunnen til at dette kan benyttes til å komprimere den bipartitetabellen enda mer er at symmetrien innen segmentet, her forutsettes det at segmentene ikke er for stort da $\frac{1}{y}$ ikke lenger blir tilnærmet symmetriske ved store utsnitt. Denne symmetrien fører til at avstanden tabell P verdien skal justeres for å finne resiprokalet over vil være tilnærmet lik det den må justeres for å finne resiprokalet under. Alle resiprokalene vil ha et resiprokal som ligger like langt unna, bare i motsatt retning fra midtpunktet. Det åpenbare problemet med denne metoden er at man trenger både en summasjonskrets og en subtraksjonskrets, men dette problemet er mulig å unngå ved og benytte egenskapene ved 2s komplement. Omgjøringen av tabell N fra pluss til minus tall i 2s komplement kan gjøres ved bare en dybde med xor porter slik at den vil innføre en minimal ekstra forsinkelse. Dette kan gjøres etter som2s komplement er alle bittene i tallet invertert og lagt til 1. Xor porter er inverterende hvis den ene inngangen legges til konstant høy, dette ser vi ut fra sannhetstabellen Tabell 1.

a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

Tabell 1: Xor sannhetstabell

Som sanhetstabellen også viser påvirker ikke xor kretsen signalet hvis den ene inngangen er låst til 0. Ettallet som skal legges til for at det skal bli 2s komplement gjøres ved at man utnytter at det er en summasjons krets der allerede og legger til en på carry inngangen på den. På grunn av at tabell N kan halveres kan man benytte det høyeste sifferet i y_l for å bestemme om det skal adderes eller subtraheres. Denne vil da også kunne føres direkte til den ene inngangen på xor portene og direkte til carry inngangen på addereren. Problemet som da gjennstår er det at den første verdien i tabell N skal være lik den siste, men ved bare å fjerne øverste bittet i y_l vil den første verdien også benyttet som et hakk over halvparten mens den mitterste verdien også vil bli benyttet som den siste, altså må verdiene snu rekkefølge når det øverste bittet i y_l er 1. Dette kan også gjøres med xor porter da invertering av en tall relle vil snu denne. Blokkskjema over symmetriske bipartitetabeller er vis på i Figur 10



Figur 10: blokkskjema symmetrisk bipartitetabeller
(denne er hentet fra [16])

1.3 Multipartitetabeller

Det er mulig å gjøre tabellene enda mindre med å dele den opp i flere tabeller, dette kalles multipartitetabeller. Multipartitetabeller fungerer i hovedsak på samme måte som bipartitetabeller der du har en Tabell P som inneholder et tilnærmet resiprokal og flere underliggende tabeller som justerer denne verdien nærmere den korrekte verdien. Bakdelen ved denne metoden er at ved flere tabeller vil det trenges flere addisjons ledd. I [4] foreslåes en metode å finne den beste tabell fordelingen til et prosjekt. Denne metoden baserer seg på symmetriske prinsippet som ble beskrevet over i Figur 9. Metoden blir nå beskrevet.

1.3.1 Forklaring av multipartitetabell oppbygning.

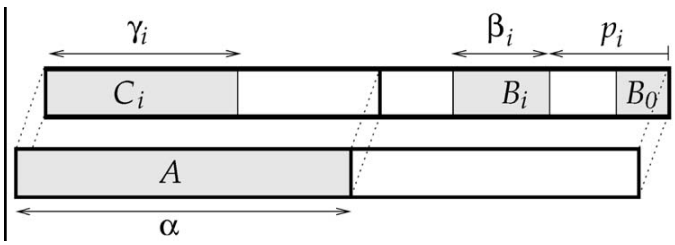
Algoritme 3. [valg av riktig oppdeling av innsignalet]

Stimuli:
Heltallene α , β, β_i, γ_i, p_i.
Respons:
Beste partisjonering av Y ut fra gitte tilfelle.

- Første deler man opp innsignalet i to deler som vi her omtaler som A og B, lengden av A er α og B er β, $\alpha + \beta = \omega_I$. A adresserer tabell P.
- Del så videre B opp i m antall deler, $B_0 \ldots B_{m-1}$. hvor den minst signifikante er B_0.
  B_i består av β_i bit.
  $p_0 = 0$ og $p_{i+1} = p_i + \beta_i$.
  B_i og en del av A kalt C_i, C_i er av lengde γ_i.
- Hver partisjon blir kaldt D, som består av følgende variabler
  α, β, m, $(\gamma_i, p_i, \beta_i)_{i=0\ldots m-1}$.

L1: Velg antall tabeller m som skal benyttes.
L2: Konstruer partisjoner D med forskjellige variabel størrelser.
L3: Beregna antall gard bits som trengs og da feilen per tabell, legg disse sammen, behold bare de partisjonene som oppfyller kravet om feil mindre enn en ulp.
L4: syntetiser de beste kandidatene

For å gjøre notasjonen her litt enklere å forstå er den vist grafisk i Figur 11



Figur 11:grafisk forklaring av notasjonene i algoritme 3
(denne figuren er hentet fra[17]. S6 (323))

Ved å benytte en løkke i L1 vil denne algoritmen også kunne evaluere versjoner med forskjellige antall tabeller. Hvis man setter m = 1, $\alpha = 2\ \omega_I$, $\gamma = \omega_I/3$, $\beta = \omega_I/3$ blir dette det samme som de bipartitetabellene forklart tidligere i oppgaven. Denne algoritmen vil øke eksponenti-

elt i kjøretid hvis vi skal prøve ut flere partisjoner, men kjøretiden til algoritmen kan reduseres på grunn av at feilen i vær av tabellene bare avhenger av typen funksjon som evalueres (her 1/y), dermed kan denne lagres i en tredimensjonal array ($e_N[p][\beta][\gamma]$) på forhånd å dermed slippes mye utregning. γ_i har også en minste verdi for at kravene skal være opprettholdt, som kan beregnes på forhånd slik at algoritmen ikke vil teste unødvendig mange partisjoner.

Beregning av antall gard bit, denne beregningsmåten er hentet fra [17].

$$g > -\omega_o - 1 + \log_2\left(\frac{((d-c)m)}{(d-c)2^{-\omega_o-1} - e^D_{ti\,\ln\,ærmet}}\right) \quad \textbf{47}$$

Så lenge $e^D_{tilnærmet}$ er mindre enn $(d\text{-}c)2^{-\omega o\text{-}1}$ vil dette gi riktig antall gard bit, så denne formelen kan benyttes direkte til å regne ut g. Er $e^D_{tilnærmet}$ større enn $(d\text{-}c)2^{-\omega o\text{-}1}$ vil det derimot ikke noe finnes noe antall gard bit som tilfredsstiller kravet om antall riktige bit. d og c er her verdi grensene til funksjonene som skal tilnærmes, som tilsvarer 1 og $\frac{1}{2}$ i resiprokal funksjonen som er hovedsakelig beskrevet i denne oppgaven, ω_o er antall bit i det tilnærmede signalet fra tabellene.

1.3.2 Sette værdiene i tabellene.

Her må det merkes at denne framgangsmåten bare gjelder for monotont synkende eller stigende funksjoner eller avgrensninger av funksjoner slik at det innen avgrensningene er monotone.

Verdien i tabell P finnes ved at funksjonen deles opp i intervaller hvor hvert intervall er av lengden $(b\text{-}a)2^{-\alpha}$. Hvor a og b er avgrensningene til funksjonen som tilnærmes (1,2 i resiprokalfunksjonen som er beskrevet tidligere i oppgaven). Endepunktene i intervallene blir kaldt x_l og x_r. Disse tabellene benytter seg av symmetrien i funksjonen å dermed vil tabell P (uavrundet) verdien bli.

$$P_{ua}(A) = \frac{f(x_l) + f(x_r)}{2} \quad \textbf{48}$$

Verdiene i N tabellene blir da sett etter prinsippet at verdiene ligger på en rett linje, denne rette linja har en heldning som er lik heldningen til funksjonen i intervallets midtpunkt. Dette kan skrives på følgende måte.

$$N(CB) = s(C)\cdot B \quad \textbf{49}$$

Hvor s(C) er heldningen i midtpunktet som intervallet C representerer, ved å dele opp B i flere deler, å med det dele opp i flere tabeller, ligning 70 viser at dette er mulig.
$B = B_0 + 2^{\beta 0}B_1 + \ldots + 2^{\beta - \beta m-1}B_{m-1}$.

$$
\begin{aligned}
N_{ua}(CB) &= s(C)\cdot\sum_{i=0}^{m-1} 2^{p_i} B_i \\
&= \sum_{i=0}^{m-1} s(C)\cdot 2^{p_i} B_i \\
&= \sum_{i=0}^{m-1} 2^{p_i} N_i(CB_i)
\end{aligned}
\qquad \textbf{50}
$$

Uavrundet N verdiene blir da.

$$N_{uai}(C_iB_i) = s(C_i)\cdot 2^{-\omega_I+p_i}(b-a)\left(B_i+\tfrac{1}{2}\right) \qquad \textbf{51}$$

Hvor $2^{-\omega I+pi}(b-a)$ er lengden mellom hver verdi i intervallet B_i, og grunnen til at vi legger på en halv er faktumet at P verdien ligger midt i mellom de midterste verdiene, altså minimum en halv lengde unna.

Avrundingen skjer her på samme måte som i den bipartitetabellen, ved enten å runde av til nærmeste eller runde ned og legge til en halv. P tabellen rundes av til nærmeste hvis m er oddetal mens hvis m er partall rundes P ned og legges til en halv. N tabellene rundes ned til nærmeste før man legger til en halv. Følgende formler gjeller da for N og P tabellene.

$$
\begin{aligned}
N_i(C_iB_i) &= RD\left(\frac{2^{\omega_0+g}}{d-c}N_{uai}(C_iB_i)\right) \\
P(A) &= RN\left(\frac{2^{\omega_0+g}}{d-c}(P_{ua}-c)+\frac{m+1}{2}+2^{g+1}\right),\ \mathrm{m}=\text{oddetall} \\
P(A) &= RD\left(\frac{2^{\omega_0+g}}{d-c}(P_{ua}-c)+\frac{m}{2}+2^{g+1}\right),\ \mathrm{m}=\text{partall}
\end{aligned}
\qquad \textbf{52}
$$

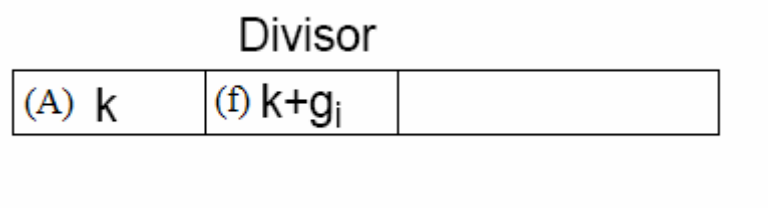
1.4 Interpolering.

Det blir også presentert en måte å benytte bipartitetabeller i [13], starter først med en vanelig lineær interpolasjon, men i motsetning til den vist tidligere vises nå en metode som kan det her halvere tabellplassende.

1.4.1 Forklaring av oppbygning

I stede for å finne c_0 og c_1 ved den linjen som vil gå gjennom endepunktene til hvert intervall, normaliseres hvert intervall slik at c_0 er lik øverste verdien i hvert intervall. I stede for å benytte hele divisoren benyttes da bare subdelen (f) som ligger under hver hoveddel (A). k skifter her igjen mening, den er nå en konstant som bestemmes ut fra hvor mange korekte bitt som er ønsket ut. k er delt opp som vist i

Initiale startverdi.



Figur 12: oppdeling av divisoren

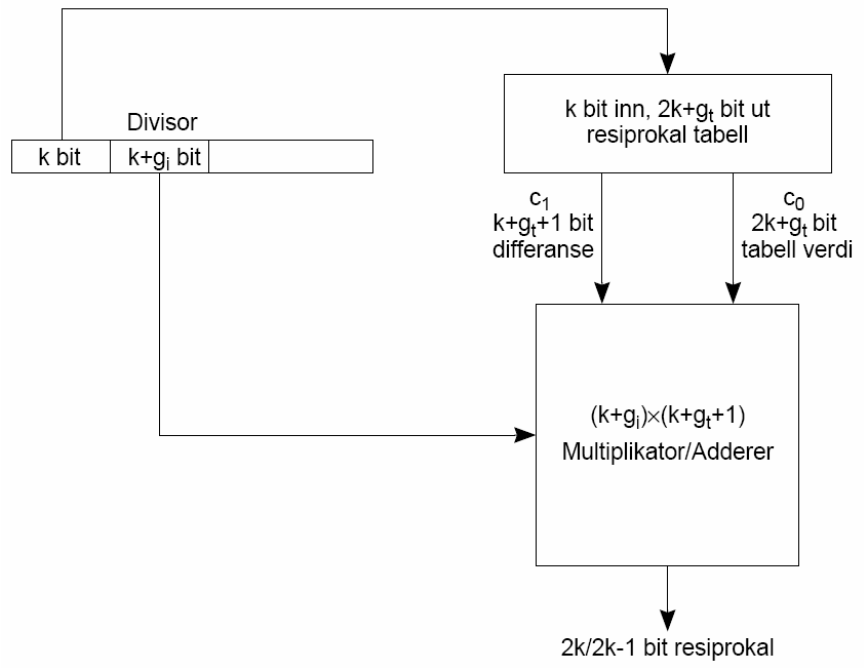
C_1 er gitt ved.

$$c_1'(i) = \frac{c_0(i) - c_0(i+1)}{\frac{1}{2^k}}$$
$$c_1'(i) = (c_0(i) - c_0(i+1))2^k \qquad \textbf{53}$$
$$z = c_0 - c_1' f'$$

For å gjøre det implementasjonen enklere og med det mer effektiv kan man benytte f og c1 som et heltall dette gir.

$$z = c_0 - \frac{c_1' \cdot 2^k}{2^{2k+g}} \cdot \frac{f}{2^{k+g}}$$
$$z = c_0 - \frac{c_1' f}{2^{2(k+g)}} \qquad \textbf{54}$$

Justeringen med 2(k+g) kan gjøres når delene trekkes fra hverandre med et tilegg av nuller. Hvis multiplikatoren her støtter borrow-save kan c_1 beregnes nesten uten å skape ekstra forsinkelse i kretsen. Flyt skjemaet for denne kretsen blir som vist under dette flytskjemaet er hentet fra [6].



Figur 13: Lineærtilnærming med halvering av tabellen
(denne er hentet fra [6])

Den lineære tilnærmingen skjer etter algoritme vist under, denne er gitt i [13].

Algoritme 4[genereing av lineær tilnærmings tabeller]

Stimuli: heltall $k \geq 1$ og $g \geq 1$.

Respons: k inn, $2k+g_t$ ut resiprokaltabell c_0 og $k+ g_t +1$ ut c_1.

for i = 0 til 2^k

L1: $c_0 = RU\left(\frac{2^k}{2^k + i}\right)$, runner opp til $2k+ g_t + 1$ bit

L2: $c_1 = c_0(i) - c_0(i+1)$

End

Algoritme 5[utregning av lineær interpolasjon]

Stimuli:

k, g_i, g_t. k inn, $2k+g_t$ ut resiprokaltabell og $2k+ g_i$ divisor y.

Respons:

2k bits resiprokal tilnærming.

L1: Y deles opp: $y = x + 2^{-k}f$. Hvor $y = \frac{j}{2^{2k+g_i}}$, $x = \frac{i}{2^k}$ og $f = \frac{l}{2^{2k+g_i}}$

L2: tilnærmet $\frac{1}{y} =$ RD($c_0(i) - c_1(i)f$), runder ned til 2k bit. $c_1(i)$ blir beregnet, den lagres ikke.

Beviset på at denne algoritmen gir riktig antall korrekte bit ut er ikke tatt med her men det finnes i kapittel 5 i [13].

1.4.2 Lineær interpolasjon med oppdelt tabell.

[13] legger fram to mulige måter å bygge opp disse tabellene som de kaller det like og odde tilfelle. Disse to tilfellende gir følgende tabellstørrelser.

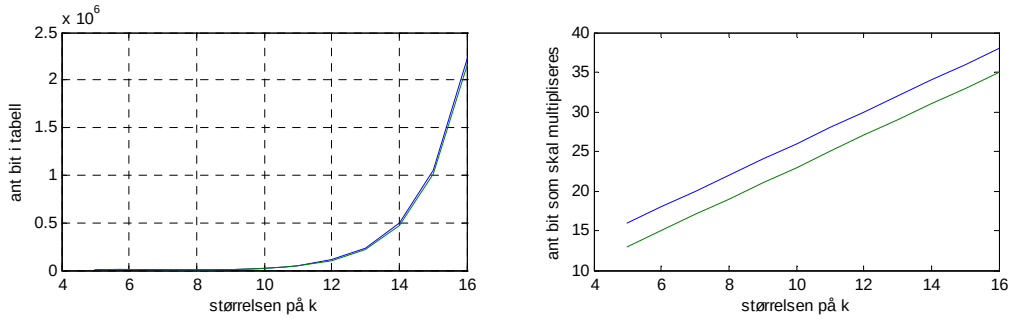
Det like tilfellet:

For å beregne et tilnærmet resiprokal på 2k korrekte bit fra en 2k+3 bit lang opperand, trenger man en k inn 2k+2 ut resiprokaltabell og en (k+3) × (k+3) multiplikator.

Det odde tilfellet:

For å beregne et tilnærmet resiprokal på 2k – 1 korrekte bit fra en 2k+1 bit lang opperand, trenger man k bit in, 2k+1 bit ut og en (k+2) × (k+1) multiplikator.

Størrelsene av tabellene vil utvikle seg eksponentielt, mens antall bit som skal multipliseres økes linjert som vist under.



Figur 14: tabell og multiplikatorstørrelse med variasjoner av k

Det blir ikke helt riktig å sammenligne disse to grafene etter som det er et bit forskjell i antall korrekte bit ut. Grafen til venstre som er tabellstørrelsen er tilnærmet lik når man ser på så store k som 16. grunnen til at jeg benyttet 16 som øverste k er å se på muligheten til å benytte dette til å lage en initialstartverdi på 32 bit slik at det er mulig med bare en iterasjon på Newton-Rapson metoden for å oppnå 64 korrekte bit. ved å benytte lik metode blir det mulig med dobbelpresisjon etter en iterasjon, mens singel presisjon kan oppnås med det odde tilfelle. Det like tilfelle gir en tabellstørrelse på 2228224bit og en 19×19 multiplikator og det odde tilfellet gir en tabellstørrelse på 2162688 bit og en 18×17 multiplikator. Andre grafen i Figur 14 er det totaleantallet bit som skal multipliseres, den like metoden ligger her altid over men den har en mer korrekt bit enn det den odde metoden har.

En av egenskapene til resiprokal funksjonen er at når den deles opp i mindre deler som gjordt her ligger den største feilen lengst til venstre, dette blir utnyttet i den lineære interpolasjonen vist i [13]. Det viser seg at ved å dele opp tabellen i to like deler vil den tabellen som tar for seg delen lengst til høyre kunne være halve størrelsen og fortsatt gi samme presisjon som den tabellen som tar for seg delen til venstre. Med denne måten vil tabellstørelsen minskes med 25 prosent, men det fører til at det er nødvendig med en litt større multiplikator og det er også nødvendig med litt ekstra logikk for å kunne veksle mellom de forskjellige tabellene. Det odde tilfellet kan minske tabellstørrelsen ytterligere etter som den skal ha en mindre korrekt bitt. Delingspunktet d er da gitt [6] som $d = RD(\log_2(k))+1$ når $2 \leq k \leq 15$ og $d = RD(\log_2(k))$ eller $RD(\log_2(k))+1$, ut fra hva k er. De to tabellene regnes dermed ut på følgende måte (beskrevet i [13]).

Algoritme 6[beregning av de oppdelte tabellene i interpolasjon]

Stimuli: heltallene k og d.

Respons: k inn, 2k+1 ut resiprokaltabell og k-1 2k+d ut resiprokaltabell.

for i = 0 til 2^{j-d}

$$c_{0topp}(i) = RU\left(\frac{2^k}{2^k + i}\right)$$, runder av til 2k+1 bit

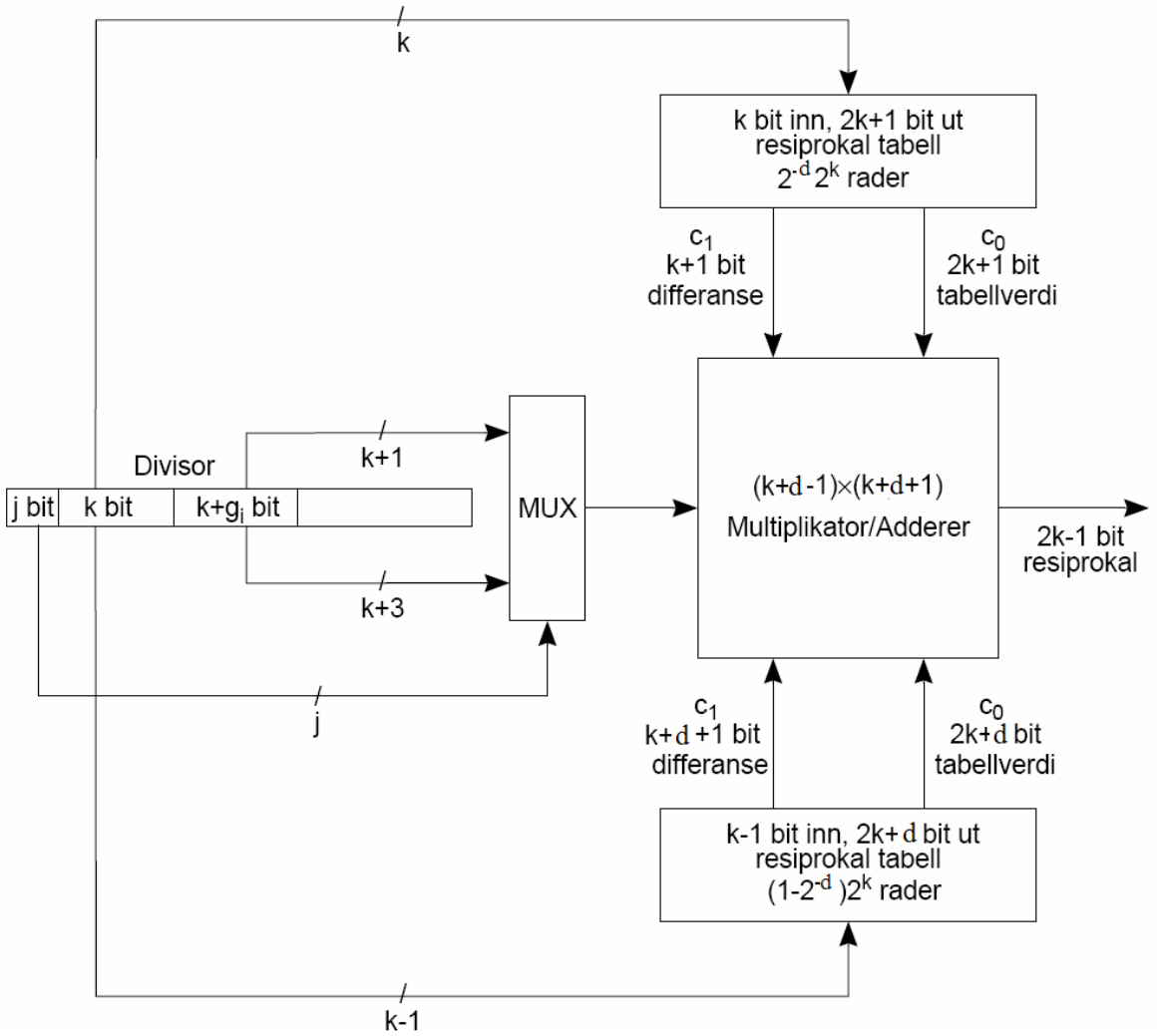
end

for j = 0 til $2^{k-1}-2^{k-j-1}$

$$c_{0bunn}(i) = RU\left(\frac{2^{k-1}}{2^{k-1} + 2^{k-j-1} + i}\right)$$, runder av til 2k+j bit

end

topp tabellen blir benyttet hvis de j øverste bittene i y er null mens bunn benytter hvis en eller flere av disse j bittene er 1. blokkskjemaet til denne kretsen er vist under i Figur 15.



Figur 15: todelt tabell for lineær tilnærming
(denne er hentet fra [6])

Ønskes det 32 bit ut fra denne må k være lik 16 dette gir da en d = $RD(\log_2(16)) = 4$ som gir videre viser å gi en større tabell enn det man hadde med bare en tabell. Det like tilfellet vil derimot gi en tabell med på 75% av den vist over så den vil da ha en størrelse på 1671168bit.

1.5 Oppsummering.

Dette kapittelet tar for seg teorien bak initialverdier til iterative forbedrings algoritmer. Kapitelet starter med den enkleste initialverdien, nemlig at alle innkommende verdiene har samme initialverdi. Kapittelet ender opp på en oversikt over bipartite og multipartitetabeller. De fleste algoritmene som er forklart i dette kapittelet er det også fullført en fullstendig funksjon og feilanalyse på for å vise at de vil gi de resultater som påstått om dem, men noen av analysene er såpass omfattende at de er utelatt fra denne oppgaven. Det er derimot alltid henvist til hvor det er mulig å finne disse analysene.

Det er her vist vanelige oppslags tabeller som passer best for bruk opp til 8 bit, bipartite tabeller tar der over opp til 16 bit og derfra tar multipartite tabeller over opp til 24bit. ønsker man høyere presisjon enn 24bit må man benytte lineær interpolasjon med delte tabeller men selv dette vil gi tabellstørrelser på over en megabyte hvis ønsket presisjon er 32 bit.

Kapittel 5.

Implementering.

1.1 Newton-Raphson metoden.

Algoritmen for Newton-Raphson metoden med q korekte bit blir er som følger (denne er hentet tilnærmet ordrett fra [5] s.40,41).

1. Hvis dividenden X eller divisoren Y er negativ, må disse først konverteres til positive operander. Dette gjøres med en toer-komplementering av eventuelle negative operander.

2. Sjekk at operandene er gyldige. Eventuelle ugyldige operander og andre spesielle operander må håndteres utenfor iterasjonsdelen av algoritmen.
   angir hvilke
   operander dette gjelder og hvordan disse skal håndteres.

3. Normaliser operandene slik at tallets høyeste siffer ligger i MSB.

4. Finn en initiell approksimasjon z0 til 1/Y. Anta at z0 har en presisjon på k bit, det vil si at z0 har en relativ feil på mindre enn 2-k.
5. Iterasjon: Beregn en mer presis tilnærming til resiprokalet utfra følgende rekursive formel:

   $$z_{n+1} = 2 \times z_n - Y \times z_n \times z_n$$

   Gjenta beregningen totalt RU(log2(q/k)) ganger. Hvor RU er rund opp til nærmeste hele tall. Presisjonen til zn dobles for hver iterasjon.

6. Beregn kvotienten: $Q = X \times z_n$.

7. Hvis dividenden X ble skiftet ett bit for lite i forhold til divisoren Y under normaliseringen, må dette kompenseres for før resten beregnes ved å skifte kvotienten Q ett bit mot venstre.

8. Beregn den tilhørende resten: $R = X - Q \times Y$

9. Hvis resten R er negativ må kvotienten Q minskes med én ulp og resten R økes med Y. Er derimot resten R større eller lik divisoren Y, må kvotienten Q økes med én ulp

og resten R minskes med Y.

10. Sjekk at dividenden X og resten R har samme fortegn. Hvis fortegnene er forskjellige, må resten R toer-komplementeres slik at fortegnene blir like.

11. Hvis én av operandene er negative skal kvotienten Q også være negativ. Dersom dette er tilfellet, skal kvotienten Q toer-komplementeres slik at denne også blir negativ.

12. For eventuelle ugyldige operander og andre spesielle operander må kvotienten Q og resten R tilordnes korrekt verdi i henhold til
.

13. Den endelige kvotienten på q bit ligger nå i Q, og den eksakte resten på q bit ligger i R.

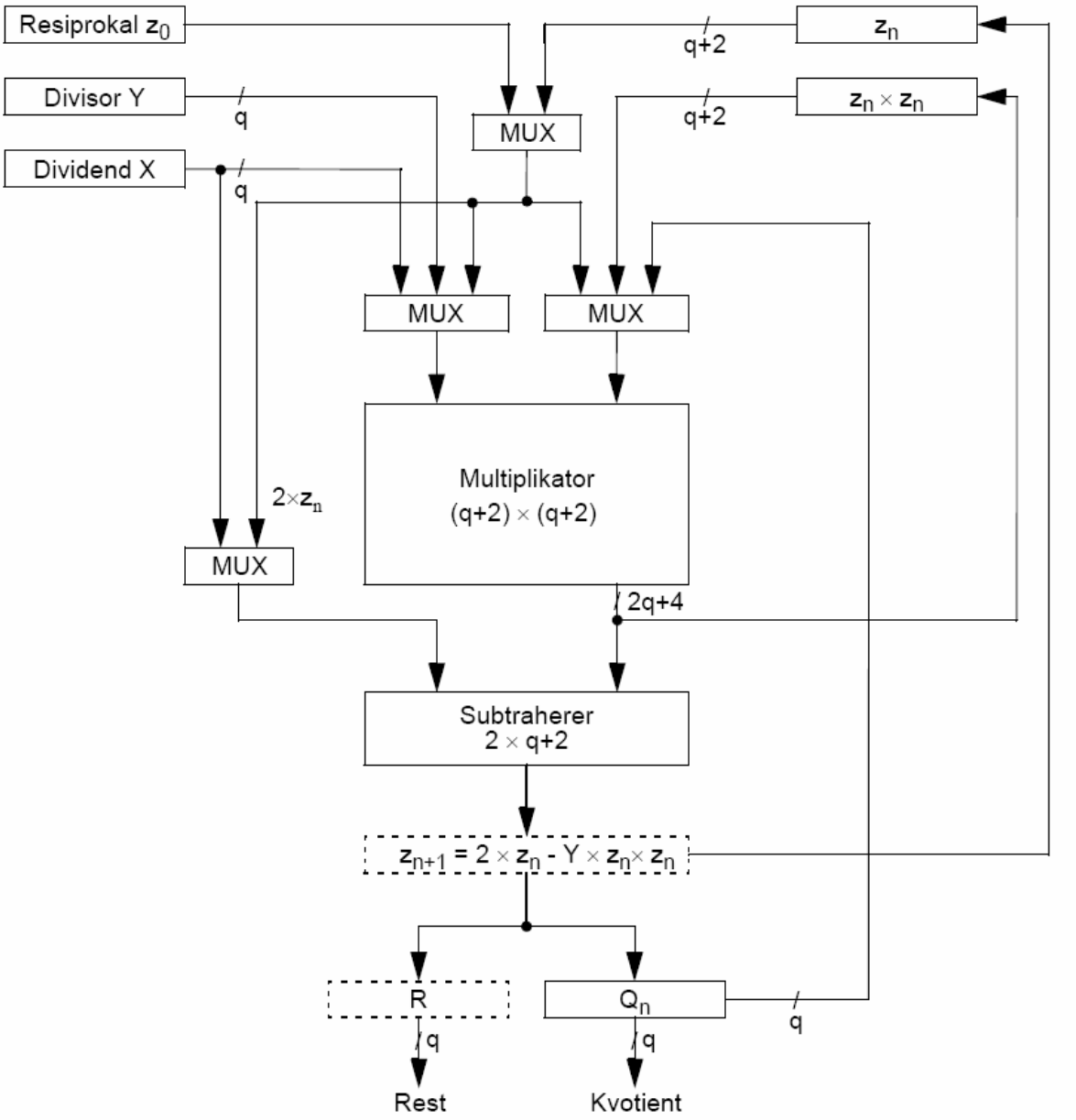
Grunnen til punkt 9 er at den tilnærmede kvotienten kan være for høy slik at produktet av Q og Y blir større enn X. kvotienten vil i dette tilfelle være en for høy så den må trekkes fra en og R vil da være Y feil å måtte justeres for dette. Hvis det enten X eller Y er negative kan di fortsatt behandles som positive tall i utregningen for punkt 10 og 11 vil rette dette opp (som vist i punkt 1).

Det er visse opperander som må behandles spesielt for de er ikke med i verdimengden til algoritmen benyttet til beregningen, de blir behandlet i punkt 2 og 12. Disse behandles ut fra følgende tabell (denne tabellen er hentet fra [5]).

Opperand	Beskrivelse	Status	håndtering
|X|>|2×Y|	Overflyt	Gyldig operand	Hvis fortegnene på operandene er forskjellige, returner kvotient lik -2 Hvis fortegnene er like returner kvotient lik 2-1 ulp Ved begge tilfellene returneres rest lik 0
Y = -2	Divisor = -2	Gyldig operand	Returner dividenden skiftet ett bit mot høyre som kvotient og 0 som rest
X = 0	Divisor lik null	Gyldig operand	Returner kvotient og rest lik 0
Y = 0	Dividend lik null	Ugyldig operand	Returner kvotient og rest lik 0

Tabell 1: Behandling av spesielle opperander
(hentet fra [5])

Ønsket implementert data flyt blir da som vist under (denne er hentet fra [5] s.39).



Figur 1: dataflyt for Newton-Raphson divisjon med avrunning og normalisering og justering av kvotient og rest
(hentet fra [5] s.43)

Dette gir følgende signal bredder ve 16bits nøyaktighet (hentet fra [5] s. 43).

Term	Størrelse (bit)	q = 16 (bit)
X	Q	16
Y	Q	16
Zn	q+2	18
zn × zn	(q+2) × (q+2) → q+2 (trunkert)	18 × 18 → 18
Y × (zn × zn)	q × (q+2) → 2q+2	16 × 18 → 34
2 × zn - (Y × zn × zn)	(q+2) - (2q+2) → 2q+2	34
X × zn	q × (q+2) → 2q+2	16 × 18 → 34
Q × Y	q× q → 2q	16 × 16 → 32
X - (Q × Y)	q - 2q → 2q	32

Tabell 2: bit bredder i dataflyten ved 16bits korrekt svar.
(hentet fra [5] s. 43)

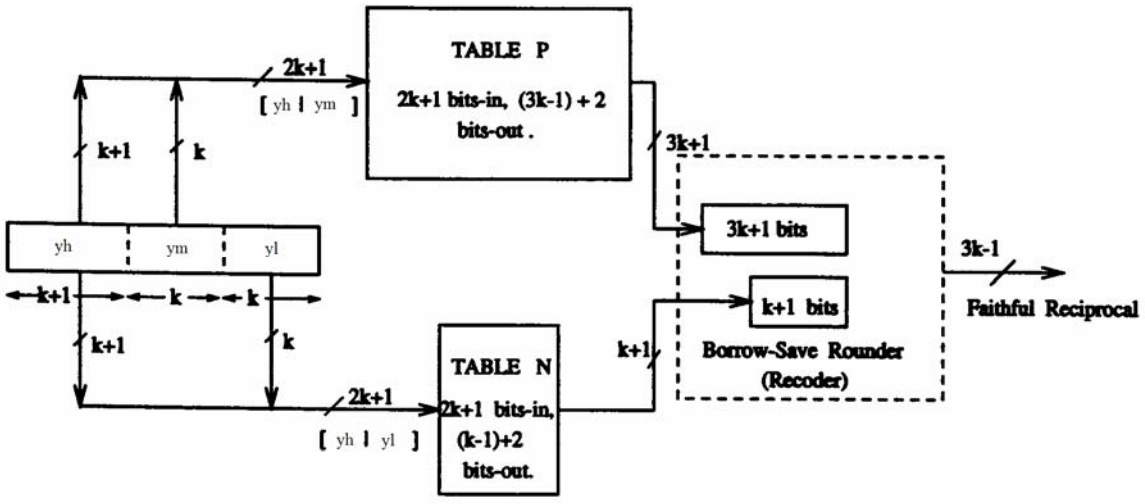
Et meget viktig poeng ved implementasjonen er hvor store mellomlagringene må være for at presisjonen ikke skal bli redusert. Det viser seg kjapt at det ikke er mulig å benytte det fulle resultatet da en iterasjon inneholder to multiplikasjoner (z_n x z_n og Y x (z_n x z_n)) og disse vil begge doble bit lengden som dermed vil med 16 bits operander si 16 x 4 = 64bit. Hvis dette skal gå to iterasjoner vil dette gi 64 x 4 = 256bit samt en ekstra multiplikasjon for å danne kvotienten. Dette sier seg selv ikke er praktisk mulig å gjennomføre. Det er derfor nødvendig å analysere hvor mange bit som er nødvendig for å opprettholde feilen mindre enn det som skal rundes av til slutt. Grunnet avrundingsmetoden som benyttes her må resultatet fra veregningene ha et minimum av q+1 riktige bit. Feilen per trunkering er mindre enn $2^{-(q+l)}$ med l ekstra bit. Feilen per iterasjon er da.

$$E_{trunk}^{n+1} = 2^{-(g+l)} + E_{trunk}^{n\ 2} \qquad \mathbf{1}$$

Hvis l settes lik 2 vil dette aldri kunne overstige $2^{-(q+l)}$ det er derfor tilstrekkelig å benytte 2 ekstrabit til mellomregningene. Dette er benyttet i Tabell 2.

1.2 Bipartitetabeller.

Antall iterasjoener er som vist i Newton-Raphson kapottelet (kap. 3) i likning 18 under initiale startverdier. Gitt ved 2s logaritmen av ønsket nøyaktighet del på initial nøyaktighet. Startverdien er derfor viktig på grunn av at den bestemmer antall iterasjoner som er nødvendig før resultatet har den ønskede presisjonen. Problemet med startverdier er for å få tilstrekkelig høy korrekthet må det enten inn flere beregnings opprasjoner som vil dra ned hastigheten eller det må til store arealer for å ha oppslagstabeller med høy korrekt het. I dagens teknologi er det meget viktig at ting går fort å dette går som regel ut over arealbruken. I [13] ble det foreslått en måte å dele opp disse tabellene i to deler kalt bipartitetabeller, og på denne måten kunne redusere tabellstørrelsen uten å redusere antallet korrekte bit. bipartitetabeller dette er vist grafisk i Figur 2:bipartittabellFigur 2.



Figur 2:bipartittabell
(dette bildet er hentet fra [13])

1.2.1 Skriptet.

Etter litraturstudiet gikk jeg først over på å lage et skript for å generere tabeller, dette skriptet tar utgangspunkt i algoritme 2. Jeg valgte her å benytte python for å programmere dette skriptet. Grunnen til at jeg valgte dette programmeringsspråket var at jeg ville lære meg det, på grunn av at det er enkelt å bygge opp prototyper av programmer. Det viste seg senere at python er såpass tregt at det ville vært bedre å benytte C++ til den delen av skriptet som skulle gå gjennom tabellen å sjekke at teorien holdt å den ga riktig antall korrekte bit ut. Under vises koden som genererer tabell P, den er bygget på algoritme 2, steg 1.

```
def TabelP(k,u,j):
    i = 1
    for xh in range(2**(k+1)):       #går gjennom alle blokkene
        fs = firstspread(xh,k,u)
        ls = lastspread(xh,k,u)
        As = averagespread(fs,ls)

        for xm in range(2**(k+u)):   #går gjennom alle segmenten
            sp = spread(xh,xm,k,u)
            ad = adjust(As,sp)
            p[xh][xm] = Pvalue(xh,xm,ad,k,u)
            pt = p[xh][xm]*2**(j+1)
            rd = rounddouwn(p[xh][xm],k,u)   #runner p ned til 3k+u+1 bit
            r[xh][xm] = float(rd)/float(2**(3*k+u+2))
            fi.write('      when "%s" =>' #%dec2bin(xh,k+1) + dec2bin(xm,k+u)
                     ' return "%s"; --%s\n' #%dec2bin(rd, j+2),xh*2**(k+u)+xm
                    %(str(dec2bin(xh, k+1)) + str(dec2bin(xm, k+u)),dec2bin(rd - 2**(3*k+u+1),
                    j+2),xh*2**(k+u)+xm))
    return None
```

firstspread(…): returnerer avstanden mellom første og siste midtverdien i første segment i blokk xh. (L1 algoritme 2)

lastspread(…): returnerer avstanden mellom første og siste midtverdien i siste segment i blokk xh. (L2 algoritme 2)

averagespread(…): returnerer middelverdien av firstspreead(…) og lastspread(…) (L3 algoritme2)

spread(…): returnerer avstanden mellom første og siste midtverdi i det segmentet som beregnes på. (L4 algoritme 2)

adjust(…): returnerer den midlede forskjellen mellom averagespread(…) og spread(…) (L5 algoritme 2)

pvalue(…): returnerer første midtverdien i den aktuelle segmentet (xh) justerer den med adjust(…). (L6 algoritme 2)

rounddouwn(…): runder ned til 3k+u+1 bit. (L7 algoritme 2)

fi.write(…): bygger opp VHDL fila med tabell P.

Koden som bygger opp tabell N er vist her, den er bygget på algoritme 2, steg 2.

```
def TableN(k,u,j):
   i = 1
   for xh in range(2**(k+1)):
      for xl in range(2**k):
         fd = firstdiff(xh,xl,k,u)
         ld = lastdiff(xh,xl,k,u)
         n[xh][xl] = Nvalue(fd,ld)
         nt = n[xh][xl]*2**(j+1)
         rn = raundtonerest(n[xh][xl],k,u)  #runner av til nermeste (k+1bit)
         rnn[xh][xl] = float(rn)/float(2**(3*k+u+2))
         fi.write('      when "%s" =>' #%dec2bin(xh,k+1) + dec2bin(xl,k)
                  ' return "%s"; --%s\n' #%dec2bin(rn, j+2),xh*2**(k)+xl
                  %(str(dec2bin(xh, k+1)) + str(dec2bin(xl, k)),dec2bin(rn, k+1),xh*2**(k)+xl))
   return None
```

firstdiff(…): returnerer differansen mellom første midtverdi i blokk xh og midtverdien til den aktuelle verdien (xl) i første segment i blokk xh. (L8 algoritme 2)

lastdiff(…): returnerer differansen mellom første midtverdii siste segment i blokk xh og midtverdien til den aktuelle verdien (xl) i sistesegment i blokk xh. (L9 algoritme 2)

Nvalue(…): returnerer middelverdien av firstdiff(…) og lastdiff(…) (L10 algoritme 2)

raundtonerest(…): runder av til nærmeste værdi med lengde k+1 bit (L11 algoritme 2)

fi.write(…) bygger opp VHDL fila med N.

En meget viktig funksjon som inngår i veldig mange av de overstående funksjonene, f. eks. firstspread(), lastspread(), firstdiff() og lastdiff() er beregningen av midtpunktet. Til en gitt verdi. Denne formelen er blitt utredet i ligning 54. I python skriptet mit er den løst på følgene måte.

```
def midResip(xh,xm,xl,k,u):
   number = xh*2**(2*k+u)+xm*2**k+xl
   return 2**(3*k+u+1)/(2**(3*k+u+1)+number+0.5)
```

Der number er nummeret til verdien det skal beregnes midtpunktet av. Bit avrundingsrutinene som er benyttet i skriptet, ganger opp tallet som skal avrundes med to opphøyd i antall bit som skal benyttes å så runder det ned eller opp etter som om det skal rundes ned eller om det skal rundes av til nærmeste. Hvis det skal rundes av til nærmeste benyttes regelen at alt som er ,5 eller større blir rundet opp.

VHDL fila som skriptet bygger opp trenger også en start og en avslutning da funksjonene TabellP og TabellN bare bygger opp en case struktur. Starten og avslutningen til dette blir gjort av funksjonene writehedderrtl() og writePtable(…). Jeg har valgt å legge begge tabellene i samme fil så writeNtable(…) skriver inn N tabellen. Til slutt trengs også subtraksjonen av tabellene for å oppnå riktig resultat. Dette blir gjort av writesubtraction(…). KDC skiller entity deklerasjonen ut i en egen fil, dette valgte jeg derfor også å gjøre. Denne fila blir opprettet av funksjonen writeentetyfile(…). Entety fila ligger i vedlegg 4.

oppførselsbeskrivelsen kommer så i en egen fil som vil ha formatet som er vist under (dette eksempelet er også tatt med 8 korrekte bit for ikke ha for lange ordbredder).

```
-- File name :  newton_int_divider_bipartit_reci_rom_rtl.vhd
-- Title     :  initiell reciprocal value
-- Module    :  initvalue
--
-- Purpose   :  create initial reciprocal value

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_signed.all;

architecture rtl of newton_int_divider_bipartit_reci_rom is
 function look_up_rom_Ptable(address : std_logic_vector(6 downto 0)) return std_logic_vector is
 begin
  case address is
   when "0000000" => return "1111111110"; --0
   when "0000001" => return "1111101110"; --1
   when "0000010" => return "1111011111"; --2
   when "0000011" => return "1111010000"; --3
                     ●
                     ●
                     ●
   when "1111110" => return "0000000111"; --126
   when "1111111" => return "0000000011"; --127
   when others => return "000000000";
  end case;
 end look_up_rom_Ptable;

 function look_up_rom_Ntable(address : std_logic_vector(6 downto 0)) return std_logic_vector is
 begin
  case address is
   when "0000000" => return "0000"; --0
   when "0000001" => return "0010"; --1
   when "0000010" => return "0100"; --2
   when "0000011" => return "0110"; --3
                     ●
                     ●
                     ●
    when "1111110" => return "0011"; --126
   when "1111111" => return "0100"; --127
   when others => return  "0000";
  end case;
 end look_up_rom_Ntable;

signal temp_out : std_logic_vector(12 downto 0);
begin
 temp_out <= ("01" & look_up_rom_Ptable(in1(9 downto 3)) & '1') - ( "00000000" &
look_up_rom_Ntable(in1(9 downto 6) & in1(2 downto 0)) & '0');
 out1 <= temp_out(12 downto 3);
end rtl;
```

Grunnen til at jeg benyttet meg av en case struktur er både fordi Zilinx benytter denne strukturen i mange av sine gratis filer som ligger ute på nettet samt at KDC benyttet denne strukturen i oppslagstabellene som er benyttet per dags dato, en struktur med ramblokker blir beskrevet senere i oppgaven. Hvis det skulle være ønskelig å benytte addisjon istedenfor subtraksjon har jeg bygget inn denne muligheten ved å legge til en funksjon som heter toskomplement(…), denne gjør som navnet tilsier, den omgjør tallet i N tabellen til tos kompliment og gjør det

dermed mulig og benytte addisjon. Man må da også bytte ut funksjonen writesubtraction(…) med funksjonen writeaddition(…).

1.2.2 Implementasjons data.

Denne implementeringen gir følgende forbruk av resurser i FPGAen med j = 8 → 16. disse tallene er hentet syntetisert med ISE 8.2.03i for virtex 4 XC4VFX60.

Selle type	J = 8	J = 9	J = 10	J = 11	J = 12	J = 13	J = 14	J = 15	J = 16
Buf	-	-	-	-	-	-	6	8	19
INV	5	3	3	1	1	2	5	2	4
LUT1	3	7	12	2	-	1	5	5	6
LUT2	5	33	85	24	46	57	76	104	130
LUT3	10	131	205	152	332	515	847	1451	2340
LUT4	78	11	12	319	483	779	1080	1382	2101
MUXCY	10	65	94	13	24	15	16	17	18
MUXF5	42	31	42	135	200	349	528	878	1378
MUXF6	22	15	20	61	82	140	206	354	534
MUXF7	11	5	8	27	37	60	84	140	201
MUXF8	-	-	-	11	15	21	26	48	73
XORCY	8	9	10	11	12	13	14	15	16

Tabell 3:resursbruk ved forskjllige antall riktige bit

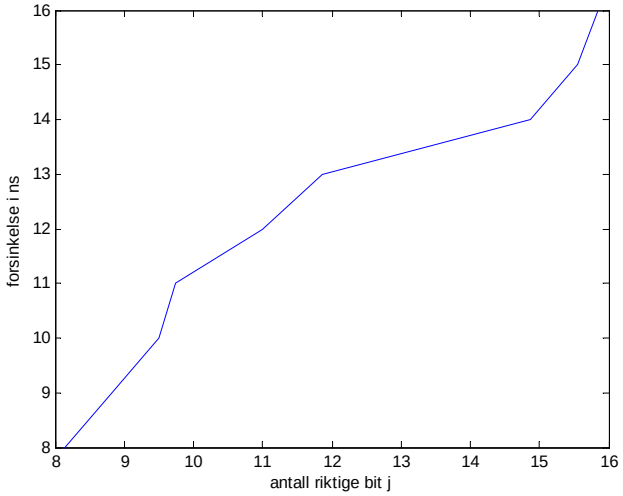
Med j = 16 benyttes 9% av slicene og 9% av lutene. Andelen slicer som benyttes går over en prosent ved 11 korekte bit. Andelen lut's går over en prosent ved 12 korekte bit.

Maks kombinatorisk forsinkelse:

J =	8	9	10	11	12	13	14	15	16
Forsinkelse ns	8.141	8.825	9.501	9.745	10.996	11.855	14.861	15.554	15.835

Tabell 4: forsinkelse i tabellen ved forskjellige antall riktige bit

Ser vi på dette grafisk ser vi at det er tilnærmet en rettlinjet økning i forsinkelsen over dette intervallet.



Figur 3: forsinkelse ved forskjellig j

Det viser seg her at med j = 16 kan man ikke benytte mer enn 63,15MHz, noe som tilsier at det er stor dybde i den kombinatoriske logikken i den syntetiserte koden. Med andre ord benytter syntetiseringen ikke minnet som allerede finnes ferdig i FPGAen, selv om den detekterer at dette kunne benyttes. Syntesen skjønner at det er snakk om minnebruk for det står oppført som ROM i synteserapporten.

1.2.3 Implementasjon med RAM

De fleste FPGAer har derimot innebygdt ROM og RAM blokker. Her sees det nærmere på hvordan man kan benytte RAM blokkene i virtex 4 FPGAer fra Xilinx. Informasjonen under er funnet i [28][29]. Virtex 4 familien inneholder fra 36 til 552 RAM blokker som kan initialiseres med verdier slik at de kan benyttes til statiske oppslagstabeller. Ram blokkene i virtex 4 familien er på 16kb samt at den har 2kb ekstra til paritetsbit. RAM blokkene er programmerbare til 6 forskjellige bit dybder, dette er 1, 2, 4, 8, 16 og 32. Paritets bittene kommer i tilegg til dette så i virkeligheten er det 1, 2, 4, 9, 18 og 36 bit dybde. Dette er vist i Tabell 5 fra [29].

Port Data Width	Port Address Width	Depth	ADDR Bus	DI Bus / DO Bus	DIP Bus / DOP Bus
1	14	16,384	<13:0>	<0>	NA
2	13	8,192	<13:1>	<1:0>	NA
4	12	4,096	<13:2>	<3:0>	NA
9	11	2,048	<13:3>	<7:0>	<0>
18	10	1,024	<13:4>	<15:0>	<1:0>
36	9	512	<13:5>	<31:0>	<3:0>

Tabell 5: Ram blokk oppdeling virtex 4
(denne er hentet fra [29])

Med j fra 8 til 16 passer denne inndelingen meget bra ut fra det faktum at tabell P vil ligge fra 10 til 18 i bit dybde, alle passer altså inn under 18 bits ord bredde. Lengden på tabell p er som kjent fra tidligere vist til å være 2^{2k+u+1}, dette vil føre til at den ligger fra $2^7 = 128$ til $2^{12} = 4096$. ut fra Tabell 5 ser vi hvor mange ord som kan lagres i RAM blokken med en ordbredde på 18, det er 1024. Tabell 6 viser da til hvor mange RAM blokker som må benyttes for å lage forskjellige tabellene.

J =	8	9	10	11	12	13	14	15	16
Antall RAM blokker, tabell P	1	1	1	1	1	1	2	4	4
Antall RAM blokker, tabell N	1	1	1	1	1	1	1	1	4
Totalt antall RAM blokker	2	2	2	2	2	2	3	5	8

Tabell 6: antall RAM blokker nødvendig bipartittabell (k+1,k+u,k)

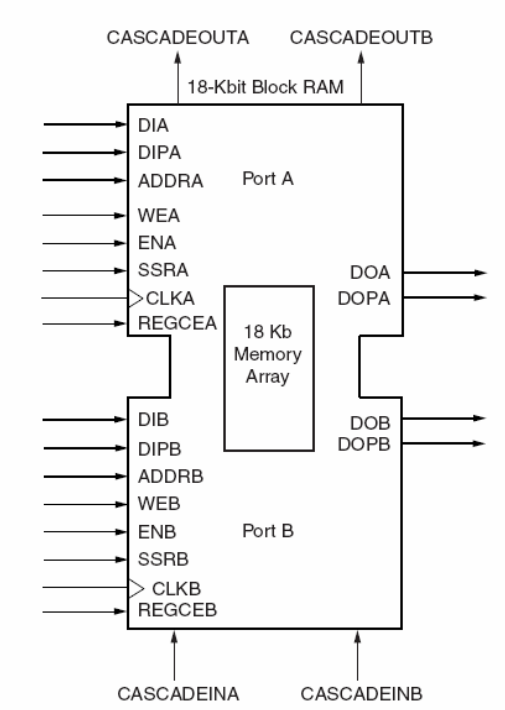
Bit dybden som benyttes til og lagre tabell N kan variere for j = 8 og 9 gir muligheten til å benytte ordbredde på 4 bit mens j fra 10 til 16 må ha en ordbredde på 9 bit. Her er det benyttet fordelingene som benyttet i algoritme 2, nemlig k+1,k+u,k. ved å benytte metoden som kan halvere tabell N vil bare hjelpe på j = 16 med antall RAM blokker. Dette er selvsagt en en sanhet med modifikasjoner da de RAM plassene som blir frigjort kan benyttes til andre systemer hvis det er mangel på RAM. Med en oppdeling av h, m og l = k,k,k | k+1,k,k | k+1,k,k+1. Får vi følgende tabell

J =	8	9	10	11	12	13	14	15	16
Antall RAM blokker, tabell P	1	1	1	1	1	1	2	2	4
Antall RAM blokker, tabell N	1	1	1	1	1	1	1	2	2
Totalt antall RAM blokker	2	2	2	2	2	2	3	4	6

Tabell 7: antall RAM blokker nødvendig i bipartitetabeller (k,k,k | k+1,k,k | k+1,k,k+1)

I denne tabellen viser det seg at med denne oppdelingsformen vil halvering av Tabell N føre til at det ikke trengs mer enn 1 RAM blokk for å lagre tabell N med j fra 8 til 16.

RAM blokkene i virtex 4 har følgende grensesnitt.



Tabell 8: Virtex4 RAM blokk
(Denne figuren er hentet fra [29])

DI[A|B] :	Skrive inngangen til RAM blokken.
DIP[A|B]:	Skrive inngangen til Paritets delen av RAMen.
ADDR[A|B]:	Adresse bussen til RAM blokken.
WE[A|B]:	Write Enable. Setter RAMen i skrive modus.
EN[A|B]:	Enabler RAM brikken.
SSR[A|B]:	Set/Reset.
CLK[A|B]:	Klokke inngang.
REGCE[A|B]:	Enabler utgangs register.
CASCADEIN[A|B]:	til kaskade kobling av RAM blokkene.
CASCADEOUT[A|B]:	til kaskade kobling av RAM blokkene.
DO[A|B]:	skrive utgangen fra RAM blokken.
DOP[A|B]:	skrive utgangen ril paritets delen av RAMen.

Tabell 9: beskrivelsene av innganger og utganger til RAM blokkene
(dette er hentet fra [29])

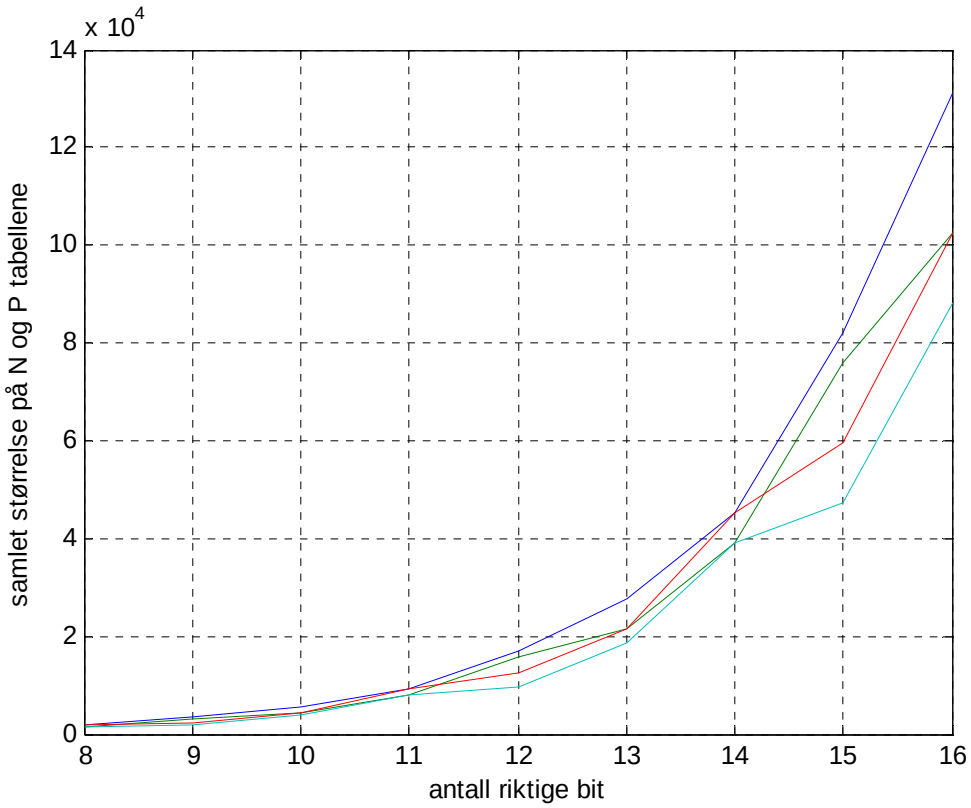
Kapittel 4 i [29] inneholder en fulstendig forklaring på hvordan man initialisrer denne rammen. Hvis man låser set reset og skrive inngangene vil dette fungere som en ROM og derfor være vell egnet til bruk i oppslagstabeller.

Beregnede verdier gir følgende størrelse for ulike j, ved forandring på fordelingen av y_h, y_m og y_l. tabellen som er merket (3k+u+1) er den som er benyttet i denne oppgaven å hentet fra [13]. Den som er merket (k,k,k | k+1,k,k | k+1,k,k+1) er hentet fra [15]. Når det står halv bak, betyr dette at algoritmen med halvering av N tabellen er benyttet.

J =	8	9	10	11	12
(3k+u+1)	1792	3328	5632	9216	16896
(3k+u+1)halv	1536	3072	4352	7936	15616
(k,k,k | k+1,k,k | k+1,k,k+1)	1792	2432	4352	9216	12288
(k,k,k | k+1,k,k | k+1,k,k+1)halv	1536	1920	3712	7936	9728

J =	13	14	15	16
(3k+u+1)	27648	45056	81920	131072
(3k+u+1)halv	21504	38912	75776	102400
(k,k,k | k+1,k,k | k+1,k,k+1)	21504	45056	59392	102400
(k,k,k | k+1,k,k | k+1,k,k+1)halv	18432	38912	47104	88064

Dette vist grafisk gir følgende graf.



Figur 4: Tabellstørrelse med forskjellig opperand partisjonering.

Den blå representerer (3k+u+1), den grønne (3k+u+1)halv, den røde(k,k,k | k+1,k,k | k+1,k,k+1) og den cyane representerer (k,k,k | k+1,k,k | k+1,k,k+1)halv.

VHDL er fleksibelt på den måten at det godtar kode som ikke kan syntetiseres, men kan simuleres og dette kan benyttets til å lage enkle oppførsels modeller for å simulere at kretsen vil virke før den er fullstendig ferdig. Dette faktum har KDC benyttet seg av og derfor laget en egen oppførsels modell til Newton-Rapson divisjonskretsen. Denne fikk jeg tilgang til så jeg leget også et tabelloppslag til denne. Etter som denne ikke skal syntetiseres er VHDL benyttet mer som programmeringsspråk kjent fra dataverden enn et redskap for å beskrive kretser entydig. Tabellen til oppførselsmodellen er derfor ikke noe tabell men en beskrivelse av hvordan tabellen skal skapes. Denne beskrivelsen ble lagt i en package, som igjen er delt opp i to filer, en som inneholder package headderen og en som inneholder package bodyen. Package headderen skapt av funksjonen writepck(…). Det er vedlagt et eksempel på package headderen i vedlegg 2.

Fila inneholder alle variablene som er nødvendig får og beregne tabellene samt deklarasjon av tabellen. Dette eksempelet er tatt med 8 korrekte bit.

Bodyen blir skapt av funksjonen writepckbdy(…) og inneholder esensielt algoritem 1 og 2.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.divlib.all;

package body newton_reci_table is

procedure init_look_up_table(tableP : inout look_up_tableP; tableN : inout look_up_tableN) is
   variable firstspread    : real;
   variable lastspread     : real;
   variable averagespread  : real;
   variable spread         : real;
   variable adjust         : real;
   variable p              : real;
   variable p_vec          : std_logic_vector(k+2 downto 0);
   variable firstdiff      : real;
   variable lastdiff       : real;
   variable n              : real;
   variable n_vec          : std_logic_vector(m downto 0);

   function resipmid(xh : integer; xm : integer; xl : integer) return real is
     variable temp : real;
   begin
     temp := real(xh)*real(2**(2*m+u))+real(xm)*real(2**m)+real(xl);
     return real(2**(3*m+u+1))/(real(2**(3*m+u+1))+temp+0.5);
   end resipmid;

 begin
   for ixh in 0 to 2**(m+1)-1 loop
     firstspread := resipmid(ixh, 0, 0)-resipmid(ixh, 0, 2**(m)-1);
     lastspread := resipmid(ixh, 2**(m+u)-1, 0)-resipmid(ixh, 2**(m+u)-1, 2**m-1);
     averagespread := (firstspread + lastspread)/2.0;

     for ixm in 0 to 2**(m+u)-1 loop
       spread := resipmid(ixh, ixm,0)-resipmid(ixh,ixm,2**m-1);
       adjust := (averagespread - spread)/2.0;
       p := (resipmid(ixh,ixm,0)+adjust)*real(2**(3*m+u+2))-0.5;
       p_vec := std_logic_vector(to_unsigned(integer(p),3*m+u+2));
       tableP(ixh*2**(m+u)+ixm+2**(2*m+u+1)) := p_vec(3*m+u downto 0);
```

```
    end loop;
  end loop;

  for jxh in 0 to 2**(m+1)-1 loop
    for jxl in 0 to 2**(m)-1 loop
      firstdiff := resipmid(jxh,0,0)-resipmid(jxh,0,jxl);
      lastdiff := resipmid(jxh,2**(m+u)-1,0) - resipmid(jxh,2**(m+u)-1,jxl);
      n := ((firstdiff+lastdiff)/2.0)*real(2**(3*m+u+2));
      n_vec := std_logic_vector(to_unsigned((integer(n)), m+1));
      tableN(jxh*2**(m)+jxl+2**(2*m+1)) := n_vec;
    end loop;
  end loop;
 end init_look_up_table;
end newton_reci_table;
```

Funksjonen resipmid() som blir deklarert etter variabel deklarasjonene er lik algoritme 1, beregning av midtpunktet til hver av verdiene. Fra den første for løkken starter algoritme 2 steg 1 og de tre etterfølgende linjene er L1, L2 og L3 i algoritmen. Etter andre for løkke kommer L1 og L5, L6 og L7 blir dekket av de 3 neste linjene. Ved den tredje for løkken starter algoritme 2 steg 2. Etter den fjerde for løkka følger L9 og L9, mens L10 og L11 fra algoritme 2

1.3 Oppsumering.

Dette kapittelet tar for seg implementeringen av Newton-Raphson metoden og problemstillinger rundt denne. Hvordan det er mulig å unngå problematikken med økende operander etter multiplikasjoner og hvordan kvotienten kan justeres for å gi riktig resultat i alle tilfeller. Anbefaler likevel å se nærmere på kapittel 4 i [5] hvis målet er og implementer en slik algoritme, for der er det gått dypere inn på problemstillingene.

Dette kapittelet tar også for seg implementeringen av bipartitetabeller, med hensyn på forskjellige parametere og strukturer. Diskuterer rundt hardware kostnadene ved implementasjon av slike kretser både i ferdig ram og i kombinatoriske kretser. Kapittelet legger størst vekt på implementering i FPGAer.

Kapittel 6.

Konklusjon.

1.1 Diskusjon

Begrepet kostnad er benyttet flere ganger i teksten under, her er det ikke ment kostnad som i rene penger, men en sammensatt kostnad av resursbruk, areal, utviklings tid/resurser og direkte penge utgifter med ekstra komponenter. Med merutgift menes da det at man benytter nye resurser, må utvikle noe helt nytt eller legge til nye komponenter.

Først er det kanskje logisk å spørre om hvor mange bit er det ønskelig å ha i start presisjon, dette kommer selvsykt helt an på hvor mange bit som skal være korrekte i svaret. Etter som kvalitetskravene hele tiden økes, skulle det være innen kryptering eller farger per piksel på et videobilde, ser det ut som det vil bli nødvendig med 64 bits presisjon i mange typer utstyr. Kravene blir også strengere når det gjelder overførings hastigheter og forsinkelse i systemet. Disse kravene gjør at det er ønskelig med så få itterasjoner som mulig i Newton-Raphson metoden, etter som Newton-Raphson algoritmen dobler antall korrekte bit per iterasjon er det da nødvendig med et minimum av halve presisjonen i initial verdien i forhold til slutt verdien, vis det skal være singel presisjon. Problemet med dette er at vi da ved å gå fra 32 til 64 korrekte bitt i svaret må øke presisjonen i initialverdien fra 16 til 32 bit. Med vanelig oppslagstabeller vil en økning fra 16 til 32 bit føre til en tabell økning på 131072 ganger pluss den ekstra logikken som trengs for å styre den økende tabellen. Det sier seg selv at dette ikke er noe man kan leve med grunnet arealprisen som er på silisium. Bipartite og multipartite tabeller som er tatt for seg i denne oppgaven vil gjøre dette en del bedre men det viser seg i praksis at bipartitettabeller gjør seg best opp til maksimalt 18 bit og multipartitetabeller opp til 24 bit. Det siste som er beskrevet her i oppgaven er interpolasjon med delte tabeller, der er det vist at 32 bits presisjon vil gi en tabell størrelse på ca 1,59Mb. Med interpolasjonen er det i tillegg to subtraksjoner og en multiplikator på 19×19 bit. I ASIC design kan subtraksjonsenhetene lages til borrow-save, hvis multiplikatoren samtidig lages til å takle dette formatet, dette vil i så fall redusere forsinkelsene i disse kretsene betraktelig. I FPGA derimot vil det å lage borrow-save logikk bare benytte mange logiske enheter som kan trenges til andre deler av systemet, det er derfor her bedre å bare benytte de innebygde subtraktorene og multiplikatorene, er det derimot ledig kapasitet er det smart å se på alternativet. I FPGAer er det innebygde RAM blokker så hvis systemet som skal designes ikke ellers benytter seg av denne RAMen er den der om den bruke eller ikke. Det er altså ikke ekstra kostnad å øke tabellstørrelsen hvis RAMen ikke benyttes, dette er ikke helt sant etter som det ikke er noe innebygd logikk for å benytte mer enn 2 RAM blokker samtidig. Det er ingen problemer å benytte Flere RAM blokker samtidig da de har enable innganger som kan benyttes til å kjøre flere samtidig. Problemet ligger derimot i at RAM blokkene ikke har tri-state utganger så det må igjen benyt-

tes ekstra logikk for å koble dette sammen. Derfor må det gjøres en analyse for hvert system som lages om det er ønskelig med den store areal forøkelsen som trengs for å doble presisjonen på initial verdi tabellen opp mot at man sparer bare en itterasjon i Newton-Raphson metoden. To punkter som er fornuftig å se på ved denne analysen er, som beskrevet før er det ubrukte RAM blokker, for da øker kostnadene ved økende tabeller lite og er det mulig å utføre divisjonen parallelt med andre opperasjoner og dermed forårsake at divisjonen blir mindre tidskritisk. Som en oppsummering kan man si at har man tilgjengelige resurser er det mulig å lage 32 initialtabeller og ha en iterasjon, men 16 bits tabeller og 2 iterasjoner vil være det beste så lenge det ikke over stiger tidskrav.

Er dette noe å satse på?

På det spørsmålet må jeg nok svare et klart ja, deling er fortsatt en meget krevende beregningsform selv med denne metoden i at den tar plass og krever mange klokke pulser. Det er derfor fortsatt ønskelig å unngå deling så mye som mulig, men dette er ikke i aller tilfeller mulig eller i verste fall vil omgjøringen av beregningen fører til at den tar lenger tid enn den ville gjort med divisjon. Newton-Raphson metoden er bevist at vil gi riktig resultat i alle tilfeller og den er gjort nøye feilanalyser slik at det er bevist at den selv uten uendelige nøyaktige mellomregninger vil kunne oppnå et tilnærmet korrekt svar, det er også her bevist hvor stor den største feilen vil være. Newton-Raphson metoden kan også benyttes til mange andre tilnærminger enn bare divisjon, det er derfor mulig å gjenbruke mange komponenter hvis ikke divisjonen benytter kapasiteten fult ut. Newton-Raphson metoden kan også realiseres i generelle prosessorer, da med noe større kostnader grunnet flere register flyttninger enn det som er mulig å oppnå i en spesiallaget krets.

Bipartite og multipartite kretser er også i høyeste grad noe å satse på hvis man trenger en initialverdi til en eller annen iterativalgoritme. Og hvis det ikke er nødvendig med mer enn 24 korrekte bit er dette et alternativ å se på direkte, altså ikke i sammenheng med noe forbedrings algoritme. Hva som er best vil variere etter f. eks. areal og hastighetskrav. Strøm bruk og varme utvikling er også mulige kriterier som vil kunne være avgjørende, men dette vil være meget avhengig av hvordan tabellene blir implementert. Bipartite og multipartite tabeller vil som Newton-Raphson metoden kunne benyttes på mange forskjellige fungsjons tilnærminger så det er ikke bare divisjon de kan være aktuelle for. Det skal derimot sies at de artikkelene jeg har lest bare tar for seg monotone funksjoner og beviset på den multipartitetabellen forutsetter at funksjonen som blir tilnærmet er monoton. Skal man derimot opp i 32bit må man benytte lineærinterpolasjon, disse vil fortsatt generere tabeller på over en megabite. Som nevnt tidligere er det mulig å implementere så store tabeller hvis det f. eks. er ubenyttete RAM blokker slik at logikk kostnaden ikke blir så stor.

RAM blokkene i en FPGA eller i et eget ASIC design vil også kunne benyttes til flere ting hvis det er sjelden divisjonen benyttes og disse er forutsigbare er det mulig å ha en harware kontroller som passer på å bytte ut innholdet i RAM blokkene før divisjonsenheten skal benytte den. Dette vil forårsake noe ekstra logikk til å kunne utføre utbyttingen og muligheten til å lese av minnet flere steder og det krever en nøye skeduling, men det kan i visse tilfeller frigi verdifullt minne. Det trengs jo også her et eksternt minne som kan lagre tabellene når de ikke ligger i FPGAen. Eksternt minne trengs nesten alltid for å lagre oppsettet til FPGAen likevel så det behøver ikke forårsake en stor kostnad. Med mer komplekse kontrollere kan det også i noen tilfeller være mulig å finne hvilket intervall tallet i nevneren er slik at det er tilstrekkelig å bare legge inn deler av tabbellen. Høyere kompleksitet på kontrolleren vil derimot forårsake

større kostnader, men det er ofte softcore prosessorer med i FPGA design i dag så hvis denne har ledig resurser kan dette dermed føre til bare en liten merkostnad.

Går man videre på tankegangen over er det mulig å benytte systemet i dynamisk rekonfigurerbare systemer, enten ved at man benytter hele som en pakke eller dele den opp i underdeler som for eksempel, normaliseringsenheten, multiplikatoren, subtraktoren og tabellene. Dette for at de kan benyttes i flere forskjellige typer og for at det skal være mulig å optimalisere. Det er dessverre ikke noe FPGAer som støtter så fin oppdeling enda så i dagens system ville det nok være mer fornuftig å lege det som en enhet som kan plassers inn i en komunikasjons struktur på FPGAen og heller ha flere tabellestørrelser ut fra nøyaktighetskrav og ledig plass.

Et viktig spørsmål som ikke er nevnt tidligere i denne oppgaven er om tabellene er monotont synkende eller ikke. Dette er viktig da det kan i visse transformer kunne oppstå oscileringer hvis kretsen ikke er monotont synkende. Med monotonitet synkende menes her at verdi(i) vil alltid være større eller lik verdi(i+1). Tabellene som blir konstruert av algoritmen 2 som er implementert i denne oppgaven vil gi monotont synkende tabeller, de andre algoritmene som også er nevnt i oppgaven kan gjøre så de blir garantert monotone. Generelle bevis og bevis for at de multipartitetabellene i denne oppgaven er monotone er beskrevet i [15].

1.2 Forslag til videre arbeid.

Et forslag til videre arbeid vil være og benytte algoritemen beskrevet i multi1partite tabeller delen av oppgaven til å gjennomføre en mer inngående undersøkelse av hvilke tabellstørrelser som vil være de optimale for forskjellige opperand lengder og antall korrekte bit ut. Dette for å kunne lage en enkel oversikt over hvilke partisjonering av opperanden som skal benyttes i de forskjellige tilfellene.

Det er som sagt under diskusjonsdelen over ønskelig å oppnå en bit presisjon på 32 eller 32,75 ut fra om man ønsker enkel eller dobbel presisjon. Det er derfor ønskelig å prøve å implementere den lineære interpolasjonsmetoden i FPGAer slik at man kunne få presise data på forsinkelsen som vil bli ut fra beregningen av c_1 når det ikke benyttes borrow-save metoden.

Det er som beskrevet i en tidligere master oppgaven fra KDC [5], ønskelig å se nærmere på Divisjons algoritmen gitt i [30], denne burde vært undersøkt i denne oppgaven men jeg klarte på ukjent vis å overse den til jeg hadde kommet for langt ut i oppgaven til å sette meg inn i den og implementere den.

1.3 Gjennomføring av oppgaven.

Prosjektet bestod av 4 deler.

1. Litteraturstudie.
2. Implementasjon.
3. Testing.
4. dokumentasjon.

De første månene gikk med til litteraturstudiet da det finnes mye materiale innen Newton-Raphson metoden og hvordan den kan implementeres, noe av dette materialet er nevnt i tidligere arbeid delen av oppgaven. Denne tiden gikk reimelig problemfritt med unntak av at underskrevede og D. Das Sarma er uenig på noen punkter hvordan ting skal legges frem for at det skal være lett forståelig for lesere, men etter flere gjennomlesninger og en sår hode bunn av all kløingen skjønte jeg til slutt hvordan det hang sammen.

Etter dette satte jeg meg inn i koden til divisjonsenheten fra KDC. Takket være meget oversiktelig og strukturert kode bydde dette ikke på noen store problemer. Det var noe av kodestilen som var uvant for meg, men dette var bare veldig lærerikt da jeg fikk se andre måter å løse problemer på enn det jeg normalt gjor. Jeg lærte også en del om hvordan man setter opp koden for at den skal være oversiktelig og enkel for andre å sette seg inn i.

For og implementere de forskjellige startverdi tabellene benyttet jeg meg av et python skript som genererte vhdl koden til forskjellige tabellstørrelsen. Jeg valgte python av den grunn at jeg ønsket å lære meg det, jeg er fortsatt ingen ekspert på python, men det er et enkelt språk å sette seg inn i så det tok ikke lang tid å få ting opp. Python er derimot litt tregt så ved beregninger for å finne ut hvor mange korrekte bit som det presist blir og hvor mange prosent som ikke runder av mot riktig side tok lang tid, så her burde jeg ha benyttet c/c++.

Til simuleringen benyttet jeg modelsim, jeg hadde litt problemer med dette i starten da skolen ikke lenger hadde alle nødvendige lisenser slik at jeg ikke fikk simulert, men dette ordnet seg etter hvert. Når jeg først fikk simulert fikk jeg feil divisjons resultat på en god del tall som ikke virket og henge i sammen så jeg satte meg ned med penn og papir å regnet ut de forskjellige tallene slik de skulle med utgangspunkt i initialverdiene jeg hadde i tabellen min. alle så da ut til å gi det riktige resultatet, men jeg fant ut hva alle tallene hadde til felles, de måtte alle justeres i justerings trinnet til Newton-Raphson metoden. Jeg skjønt ikke helt hvordan dette kunne ha seg da KDC jo hadde testet denne kretsen nøye så jeg satte meg ned å så på koden, etter en del feilsøking fant jeg ut at det var ”>” operatoren som ikke returnerte riktig resultat. Grunnen til dette fant jeg etter litt mer feilsøking ut av var for jeg hadde inkludert enda en bibliotek fil, std_logic_unsigned for å kunne benytte operatoren ”+” direkte på std_logic_vector. Problemet ligger at std_logic_unsigned også definere ”>”, dette klusset opp definisjonen til operatoren og derfor ble den aldri TRUE. Problemet løste seg derfor når jeg gikk tilbake til og bare benytte numeric_std.

Dokumentasjons fasen som jeg er i nå har gått bra med unntak av det ekstremt fine juni været som har vært den siste tiden.

1.4 Konklusjon.

Målet i denne oppgaven var å studere multiplikative divisjons algoritmer for implementering i digitale elektroniske systemer. Etter et oppklarings møte med oppgavegiverne KDC, ble det klart at de ønsket å se nærmere på startverdier til divisjonsalgoritmene. Jeg valgte å benytte Newton-Raphson metoden som divisjons algoritme, denne hadde KDC allerede benyttet og hadde VHDL kode som jeg kunne benytte til simuleringer. Burde sett nærmere også på algoritmen beskrevet i [30] før jeg valgte og også gjerne implementert begge for å kunne se de opp mot hverandre. Men begge algoritmene baserer seg på gode startverdier så denne oppgaven vil fortsatt være aktuell i samband med algoritmen gitt i [30].

Jeg har analysert Newton-Raphson metoden benyttet til divisjon, både i form av funksjons analyse og feilanalyse. Jeg har studert tideligere implementeringer av denne algoritmen som med et ekstra justerings trinn slik at den oppnår korrekt resultat i alle tilfeller.

Jeg gjort funksjons å feilanalyse på bipartite og Multipartite tabeller og studert forskjellige algoritmer for genereringa av disse tabellene.

Jeg har implementert flere forskjellige størrelser av bipartite tabeller og gjort beregninger rundt multipartite tabeller.

Jeg har simulert de bipartitetabellene som startverdi for Newton-Raphson metoden og sett at den fungerer på den måten den skal. Jeg har også simulert de bipartitetabellene direkte uten ekstra iterative forbedringer.

Jeg har benyttet implementeringen til å finne resursbruken etter syntes, opp mot virtex 4 familien fra Xilinx.

Slutt konklusjonen er at denne teknikken er meget interessant innen mange forskjellige funksjons tilnærminger. Det ser ut til at beregningsbehovet bare vil øke fremmover og da vil det være ønskelig å kunne tilnærme mange typer funksjoner som kan benytte bipartite eller multipartitetabeller enten som direkte giver av verdien eller som oppslagstabell for iterative algoritmer.

Referanseliste:

[1] P. Deuflhard, Newton Methods for Nonlinear Problems. Affine Invariance and Adaptive Algorithms. Springer Series in Computational Mathematics, Vol. 35. Springer, Berlin, 2004. ISBN 3-540-21099-7.

[2] C. T. Kelley, Solving Nonlinear Equations with Newton's Method, no 1 in Fundamentals of Algorithms, SIAM, 2003. ISBN 0-89871-546-6.

[3] J. M. Ortega, W. C. Rheinboldt, Iterative Solution of Nonlinear Equations in Several Variables. Classics in Applied Mathematics, SIAM, 2000. ISBN 0-89871-461-3.

[4] Endre Süli and David Mayers, An Introduction to Numerical Analysis, Cambridge University Press, 2003. ISBN 0-521-00794-1.

[5] Multifunksjonell beregningsenhet for digital signal prosessering av Simen Gimle Hansen.

[6] Analyse og implementering av multiplikative divisjonsalgoritmer av Marius Gimle Hansen.

[7] S. Waser, M. J. Flynn, “Introduction to Arithmetic for Digital Systems Designers”, Holt, Rinehart and Winston, 1982.

[8] A. R. Omondi, “Computer Arithmetic Systems: Algorithms, Architecture and Implementations”, Prentice Hall, 1994.

[9] J. J. F. Cavanagh, “Digital Computer Arithmetic: Design and Implementations”, McGraw-Hill, 1984.

[10] D. Wong, M. J. Flynn, “Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations”, IEEE Transactions on Computers, Vol 41, No 8, August 1992, pp 981-995.

[11] M. G. Hansen, “Analyse og implementering av multiplikative divisjonsalgoritmer”, Hovedfagsoppgave, Institutt for informatikk, Universitetet i Oslo, 14. februar 1997.

[12] R. C. Agarwal, F. G. Gustavson, M. S. Schmookler, “Series Approximation Methods For Divide and Square Root in the Power3™ Processor”, Proceedings 14th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1999, pp 116-123.

[13] D. Das Sarma, “Highly Accurate Initial Reciprocal Approximations for High Preformance Division Algorithms”, Ph.D Dissertation, School of Engineering and Applied Science, Southern Methodist University, 1995.

[14] D. Das Sarma, D. W. Matula, “Measuring the Accuracy of ROM Reciprocal Tabels”, Proceedings 11th Symposium on Computer Arithmetic, 1993, pp 95-102.

[15] Single Precision Reciprocals by Multipartite Lookup av Peter Kornerup og David W. Matula

[16] M. Schulte and J. Stine, “Approximating Elementary Functions with Symmetric Bipartite Tables,” IEEE Trans. Computers, vol. 48, no. 8, pp. 842-847, Aug. 1999.

[17] Multipartite Table Methods av Florent de Dinechin and Arnaud Tisserand

[18] H. Hassler and N. Takagi, “Function Evaluation by Table Look-Up and Addition,” in *Proc. 12th IEEE Symposium on Computer Arithmetic*. IEEE, 1995, pp. 10–16.

[19] M. Schulte and J. Stine, “Approximating Elementary Functions with Symmetric Bipartite Tables,” *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 842–847, 1999.

[20] J.-M. Muller, “A Few Results on Table-Based Methods,”*Reliable Computing*, vol. 5, no. 3, pp. 279–288, 1999.

[21] C. Iordache and D. Matula, “Analysis of Reciprocal and Square Root Reciprocal Instructions in the AMD K6-2 Implementation of 3DNow,” *Electronic Notes in Theoretical Computer Science*, vol. 24, 1999.

[22] F. de Dinechin and A. Tisserand, “Multipartite Table Methods,” in *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, 2005.

[23] P.-M. Seidel,“High-Speed Redundant Reciprocal Approximation,“ in *INTEGRATION, the VLSI Journal*, vol. 28, pp. 1–12, 1999.

[24] W. Wong and E. Goto, “Fast Evaluation of the Elementary Functions in Single Precision,” *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 453–457, 1995.

[25] J. Pineiro, J. Bruguera, and J.-M. Muller, “Faithful Powering Computation using Table Look-Up and a Fused Multiplication Tree,” in *Proc. 15th IEEE Symposium on Computer Arithmetic*. IEEE, 2001, pp. 40–47.

[26] F. de Dinechin and J. Detrey, “Multipartite Tables in JBits for the Evaluation of Functions on FPGA’s,” in *IEEE Reconfigurable Architecture Workshop, International Parallel and Distributed Symposium, Fort Lauderdale, Florida.* IEEE, April 2002.

[27] M.J. Schnlte, J. Omar and E.E. Swartzlander, “Optimal Approximations for the Newton-Raphson Division Algorithm,” presentgd at SCAN-93, thc *Conference* on *$cieniific Computing, Compajer* Ara‘J*metic, and NTumeric Validation* in Vienna,

Sept. 1993.

[28] Virtex-4 Family Overview, DS112 (v2.0) January 23, 2007, xilinx.com.

[29] Virtex-4 User Guide, UG070 (v2.2) April 10, 2007, xilinx.com

[30] R. C. Agarwal, F. G. Gustavson, M. S. Schmookler, “Series Approximation Methods for Divide and Square Root in the Power3TM Processor”, Proceedings 14th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1999, pp 116-123.

[31] Newton's method wikipedia.org.

Vedlegg 1.

```
# -*- coding: cp1252 -*-
#####################################################################
#   Bipartite resiprokaltabeller                                    #
#   Dette programmet konstruerer bipartite resiprokaltabeller       #
#   med j bit ut og j+2 bit inn                                     #
#                                                                   #
#   format på inn data er:                                          #
#   1,b(1)b(2)b(3)...b(j+2)                                         #
#                                                                   #
#   formatet op ut data er:                                         #
#   0,B(1)B(2)B(3)...B(j)                                           #
#   B(1) er altid lik 1                                             #
#   Det skal også vere med en ekstra ener på slutten i videre       #
#   beregninger                                                     #
#                                                                   #
#   denne algoritmen benyttes for innbit større enn 6               #
#####################################################################

#importerer eksterne funksjoner.
#--------------------------------------------------------------------
#import os
#import sys
import math

#lager P tabellen.
#--------------------------------------------------------------------
def TabelP(k,u,j):
    i = 1
    for xh in range(2**(k+1)):      #går gjennom alle blokkene
        fs = firstspread(xh,k,u)
        ls = lastspread(xh,k,u)
        As = averagespread(fs,ls)

        for xm in range(2**(k+u)):  #går gjennom alle segmenten
            sp = spread(xh,xm,k,u)
            ad = adjust(As,sp)
            p[xh][xm] = Pvalue(xh,xm,ad,k,u)
            pt = p[xh][xm]*2**(j+1)
            rd = rounddouwn(p[xh][xm],k,u)   #runner p ned til 3k+u+1 bit
            r[xh][xm] = float(rd)/float(2**(3*k+u+2))
            fi.write('      when "%s" =>' #%dec2bin(xh,k+1) + dec2bin(xm,k+u)
                    ' return "%s"; --%s\n' #%dec2bin(rd, j+2),xh*2**(k+u)+xm
                    %(str(dec2bin(xh, k+1)) + str(dec2bin(xm, k+u)),dec2bin(rd - 2**(3*k+u+1), j+2),xh*2**(k+u)+xm))
    return None

#lager N tabellen
#--------------------------------------------------------------------
def TableN(k,u,j):
    i = 1
    for xh in range(2**(k+1)):
        for xl in range(2**k):
            fd = firstdiff(xh,xl,k,u)
            ld = lastdiff(xh,xl,k,u)
            n[xh][xl] = Nvalue(fd,ld)
            nt = n[xh][xl]*2**(j+1)
            rn = raundtonerest(n[xh][xl],k,u)  #runner av til nermeste (k+1bit)
            rnn[xh][xl] = float(rn)/float(2**(3*k+u+2))
            fi.write('      when "%s" =>' #%dec2bin(xh,k+1) + dec2bin(xl,k)
                    ' return "%s"; --%s\n' #%dec2bin(rn, j+2),xh*2**(k)+xl
```

```
            %(str(dec2bin(xh, k+1)) + str(dec2bin(xl, k)),dec2bin(rn, k+1),xh*2**(k)+xl))
        #for å benytte 2's komplement. husk å bytte writesubtraction(...) til writeaddition(...)
        #fi.write('      when "%s" =>' #%dec2bin(xh,k+1) + dec2bin(xl,k)
        #         ' return "%s";\n' #%dec2bin(toskomplement(rn), j+2)
        #         %(str(dec2bin(xh, k+1)) + str(dec2bin(xl, k)),dec2bin(toskomplement(rn), k+1))
    return None

#finner spredningen i første segment for blokken xh
#----------------------------------------------------------------
#mid Resiprokalet av første verdi i første segment i blokk xh
#minus mid resiprokalet til den siste verdien i samme segment og
#blokk
def firstspread(xh,k,u):
    return midResip(xh,0,0,k,u) - midResip(xh,0,2**k-1,k,u)

#finner spredningen i siste segment i blokk xh
#----------------------------------------------------------------
#mid Resiprokalet av første verdi i siste segment i blokk xh
#minus mid resiprokalet til den siste verdien i samme segment og
#blokk
def laststspread(xh,k,u):
    return midResip(xh,2**(k+u)-1,0,k,u) - midResip(xh,2**(k+u)-1,2**k-1,k,u)

#finner gjennomsnittelig spredningen i blokk xh
#----------------------------------------------------------------
#spredningen i første segment pluss spredningen i siste segment i
#blokk xh del på 2
def averagespread(fs,ls):
    return (fs+ls)/2

#finner spredningen i segment xm i blokk xh
#----------------------------------------------------------------
#mid Resiprokalet av første verdi i segment xm i blokk xh
#minus mid resiprokalet til den siste verdien i samme segment og
#blokk
def spread(xh,xm,k,u):
    return midResip(xh,xm,0,k,u) - midResip(xh,xm,2**k-1,k,u)

#finner gjennomsnittelig spredningen i blokk xh
#----------------------------------------------------------------
#spredningen i første segment pluss spredningen i siste segment i
#blokk xh del på 2
def adjust(As,sp):
    return (As-sp)/2

#finner P verdien til blokk xh segment xm.
#----------------------------------------------------------------
#regner ut mid resiprokalet til første verdi i segment xm i blokk
#xh og justerer dette i forhold til gjennomsnittelig og egen
#spredning. (dette er "infinit riktige")
def Pvalue(xh,xm,ad,k,u):
    return midResip(xh,xm,0,k,u) + ad

#runder P ned og trunkerer den til 3k+u+1 bit
#----------------------------------------------------------------
def rounddouwn(p,k,u):
    return int(p*2**(3*k+u+2))

#Beregner resiprokalet til midt pungtet i intervallet.
#----------------------------------------------------------------
def midResip(xh,xm,xl,k,u):
    number = xh*2**(2*k+u)+xm*2**k+xl
    return 2**(3*k+u+1)/(2**(3*k+u+1)+number+0.5)

#Beregner differangsen til det første mid resiprokalet i første segment
#----------------------------------------------------------------
#mid resiprokalen først første segment i blokk xh minus mid
#resiprokalet til verdi xl i samme blokk og segment
def firstdiff(xh,xl,k,u):
    return midResip(xh,0,0,k,u)-midResip(xh,0,xl,k,u)

#Beregner differangsen til det først mid resiprokalet i siste segmentet
#----------------------------------------------------------------
#mid resiprokalen først i siste segment i blokk xh minus mid
#resiprokalet til verdi xl i samme blokk og segment
```

```
def lastdiff(xh,xl,k,u):
   return midResip(xh,2**(k+u)-1,0,k,u)-midResip(xh,2**(k+u)-1,xl,k,u)

#beregne den eksakte N verdien.
#-----------------------------------------------------------------
#beregner gjennomsnittet av forsjellen til det første mid
#resiprokalet i første og siste segment. med andre ord
#(fistdiff(xh,xl)-lastdiff(xh,xl))/2
def Nvalue(fd,ld):
   return (fd+ld)/2

#runner av til nærmeste tall.
#-----------------------------------------------------------------
def raundtonerest(n,k,u):
   n = n*2**(3*k+u+2)
   if(n-int(n) > 0.5):
      return int(n+0.5)
   else: return int(n)

#ommgjør fra desimaltall til binære tall
#-----------------------------------------------------------------
def dec2bin(d, antbit):
   bin = ''
   if d == 0: bin = '0'
   else:
      while(d > 0):
         bin = str(d%2) + bin #+ som string opperator ikke matematisk
         d = d >> 1
   for i in range(antbit - len(bin)):
      bin = '0' + bin
   return bin

#ommgjør til 2's komplement
#-----------------------------------------------------------------
def toskomplement(orginal, k):
   return 2**(k+1)-orginal

#skriver til entety vhdl fila.
#-----------------------------------------------------------------
def writeentetyfile(j):
   fe.write('-- File name :  newton_int_divider_bipartit_reci_rom_rtl.vhd\n'
         '-- Title     :  initiell reciprocal value\n'
         '-- Module    :  initvalue\n'
         '-- \n'
         '-- Purpose   :  create initial reciprocal value\n\n')

   fe.write('LIBRARY IEEE;\n'
         'USE IEEE.std_logic_1164.all;\n'
         'USE IEEE.std_logic_signed.all;\n\n')

   fe.write('entity newton_int_divider_bipartit_reci_rom is\n'
         '  port(\n'
         '       in1   :  in std_logic_vector(%s downto 0);\n' #%(j+1)
         '       out1  :  out std_logic_vector(%s downto 0)\n' #%(j+1)
         '        );\n'
         'end newton_int_divider_bipartit_reci_rom;\n\n' %(j+1,j+1))
   return None

#skriver hedderen til vhdl rtl fila.
#-----------------------------------------------------------------
def writehedderrtl():
   fi.write('-- File name :  newton_int_divider_bipartit_reci_rom_rtl.vhd\n'
         '-- Title     :  initiell reciprocal value\n'
         '-- Module    :  initvalue\n'
         '-- \n'
         '-- Purpose   :  create initial reciprocal value\n\n')

   fi.write('LIBRARY IEEE;\n'
         'USE IEEE.std_logic_1164.all;\n'
         'USE IEEE.std_logic_signed.all;\n\n')

   fi.write('architecture rtl of newton_int_divider_bipartit_reci_rom is\n')
   return None

#skriver P tabellen
```

```
#-------------------------------------------------------------------
def writePtable(j,k,u):
   fi.write('  function look_up_rom_Ptable(address : std_logic_vector(%s downto 0)) return std_logic_vector is\n' #%2*k+u
         '  begin\n'
         '    case address is\n' %(2*k+u))
   TabelP(k,u,j)
   tilegg = '\"' + '0'*(j+1) + '\"' #lager riktig antall nuller for "when others"
   fi.write('      when others => return %s;\n'
         '    end case;\n'
         '  end look_up_rom_Ptable;\n\n' %tilegg)
   return None

#skriver N tabellen
#-------------------------------------------------------------------
def writeNtable(k,u,j):
   fi.write('  function look_up_rom_Ntable(address : std_logic_vector(%s downto 0)) return std_logic_vector is\n' #%2*k
         '  begin\n'
         '    case address is\n' %(2*k))
   TableN(k,u,j)
   tilegg = '\"' + '0'*(k+1) + '\"' #lager riktig antall nuller for "when others"
   fi.write('      when others => return  %s;\n'
         '    end case;\n'
         '  end look_up_rom_Ntable;\n\n' %tilegg)
   return None

#lager sammenlegningen av tallene
#-------------------------------------------------------------------
def writesubtraction(j,k,u):
   tillegg = '0'*(2*k+u+2) #legger til riktig antall nuller
   fi.write('  signal temp_out : std_logic_vector(%s downto 0);\n' #%j+2+2
         'begin\n'
         #'    out1 <= ("01" & look_up_rom_Ptable(in1(%s downto %s))) - ( "%s" & look_up_rom_Ntable(in1(%s downto %s) &
in1(%s downto 0)));\n'
         '  temp_out <= ("01" & look_up_rom_Ptable(in1(%s downto %s)) & \'1\') - ( "%s" & look_up_rom_Ntable(in1(%s
downto %s) & in1(%s downto 0)) & \'0\');\n'
         '  out1 <= temp_out(%s downto 3);\n' #%j+2+2
         'end rtl;\n\n' %(j+4, j+1, k, tillegg, j+1, 2*k+u, k-1, j+4))
   return None

#lager sammenlegningen av tallene med 2's komplement
#-------------------------------------------------------------------
#def writeaddition(j,k,u):
#    tillegg = '1'*(2*k+u+2) #legger til riktig antall nuller
#    fi.write('begin\n'
#             '    out1 <= ("01" & look_up_rom_Ptable(in1(%s downto %s))) + ( "%s" & look_up_rom_Ntable(in1(%s downto %s) &
in1(%s downto 0)));\n'
#            'end rtl;\n\n' %(j+1, k, tillegg, j+1, 2*k+u, k-1))
#    return None

#normaliserer dividenden
#-------------------------------------------------------------------
def normelize(y):
   a = math.log(y,2)
   return y/2**(int(a))

#finner det korrekte resiprokalet(så korekt som den blir i python)
#-------------------------------------------------------------------
def correctdiv(y, numbit):
   return 1/normelize(float(y))

#trekker ut xh av divisoren
#-------------------------------------------------------------------
def findxhxmxl(y,numbit,k,u):
   y = normelize(float(y))
   print y
   temp = (y-1.0)*float(2**(k+1))
   #print temp
   xh = int(temp)
   print xh
   temp = (temp - float(xh))*float(2**(k+u))
   xm = int(temp)
   print xm
   temp = (temp - float(xm))*float(2**(k))
   xl = int(temp)
   print xl
```

```
    return xh, xm, xl

#finner resiprokalet ut fra de uavrunnede tabellene
#---------------------------------------------------------------------
def appreci(y,k,u,numbit):
    xh, xm, xl = findxhxmxl(y,numbit,k,u)
    return p[xh][xm] - n[xh][xl]

#finner resiprokalet ut fra de avrunnede tabellene
#---------------------------------------------------------------------
def apprecirouned(y,k,u,numbit):
    xh, xm, xl = findxhxmxl(y,numbit,k,u)
    return r[xh][xm] - rnn[xh][xl]

#beregner antall riktige bit
#---------------------------------------------------------------------
def numcorrbit(diff):
    if(diff < 0):
        diff *= (-1.0)
    return math.log(diff,2)

#lager tabell pakken til oppførsels modellen
#---------------------------------------------------------------------
def writepck(numbit, j, k, u):
    #headder
    fp.write('library IEEE;\n'
            'use IEEE.std_logic_1164.all;\n'
            'use IEEE.numeric_std.all;\n'
            'use work.divlib.all;\n\n')

    #konstanter
    fp.write('package newton_reci_table is\n\n'
            '  constant q : natural := %s;\n' #antall bit totalt (numbit)
            '  constant k : natural := %s;\n' #antall riktige bit (j)
            '  constant g : natural := %s;\n' #antall garderingsbitt
            '  constant u : natural := %s;\n' #differangsen mellom xm og xl (u)
            '  constant m : natural := %s;\n\n' #samme som k her i skriptet
            %(numbit, j, 2, u, k))

    #deklarere typer og prosedyren
    fp.write('  subtype depthP is std_logic_vector(k+1 downto 0);\n' #j+2
            '  subtype depthN is std_logic_vector(m downto 0);\n\n' #k+1
            '  type look_up_tableP is array(2**(2*m+u+1) to 2**(2*m+u+2)-1) of depthP;\n' #2*k+u,2*k+u
            '  type look_up_tableN is array(2**(2*m+u+1) to 2**(2*m+u+2)-1) of depthN;\n\n' #2*k+u,2*k+u
            '  procedure init_look_up_table(tableP : inout look_up_tableP; tableN : inout look_up_tableN);\n'
            )#%(j+2, k+1, 2*k+u, 2*k+u, 2*k+u, 2*k+u))

    #avslutter pakken
    fp.write('end newton_reci_table;')
    return None

#lager tabell pakke bodyen til oppførsels modellen
#---------------------------------------------------------------------
def writepckbdy(j, k, u):
    #hedder
    fb.write('library IEEE;\n'
            'use IEEE.std_logic_1164.all;\n'
            'use IEEE.numeric_std.all;\n'
            'use work.divlib.all;\n\n')

    fb.write('package body newton_reci_table is\n\n')

    fb.write('  procedure init_look_up_table(tableP : inout look_up_tableP; tableN : inout look_up_tableN) is\n'
            '    variable firstspread    : real;\n'
            '    variable lastspread     : real;\n'
            '    variable averagespread  : real;\n'
            '    variable spread         : real;\n'
            '    variable adjust         : real;\n'
            '    variable p              : real;\n'
            '    variable p_vec          : std_logic_vector(k+2 downto 0);\n' #j+2
            '    variable firstdiff      : real;\n'
            '    variable lastdiff       : real;\n'
            '    variable n              : real;\n'
            '    variable n_vec          : std_logic_vector(m downto 0);\n\n' # ommgjordt til full lengde. gammel =k+1
            )#%(j+2, k+1))
```

```
    fb.write('    function resipmid(xh : integer; xm : integer; xl : integer) return real is\n'
            '      variable temp : real;\n'
            '    begin\n'
            '      temp := real(xh)*real(2**(2*m+u))+real(xm)*real(2**m)+real(xl);\n'
            '      return real(2**(3*m+u+1))/(real(2**(3*m+u+1))+temp+0.5);\n'
            '    end resipmid;\n\n')

    fb.write('  begin\n'
            '    for ixh in 0 to 2**(m+1)-1 loop\n' #k
            '      firstspread := resipmid(ixh, 0, 0)-resipmid(ixh, 0, 2**(m)-1);\n' #k
            '      lastspread := resipmid(ixh, 2**(m+u)-1, 0)-resipmid(ixh, 2**(m+u)-1, 2**m-1);\n' #k,u,k,u,k
            '      averagespread := (firstspread + lastspread)/2.0;\n'
            '      for ixm in 0 to 2**(m+u)-1 loop\n' #k,u
            '        spread := resipmid(ixh, ixm,0)-resipmid(ixh,ixm,2**m-1);\n' #k
            '        adjust := (averagespread - spread)/2.0;\n'
            '        p := (resipmid(ixh,ixm,0)+adjust)*real(2**(3*m+u+2))-0.5;\n' #k,u
            '        p_vec := std_logic_vector(to_unsigned(integer(p),3*m+u+2));\n' #k,u
            '        tableP(ixh*2**(m+u)+ixm+2**(2*m+u+1)) := p_vec(3*m+u downto 0);\n' #k,u,k,u
            '      end loop;\n'
            '    end loop;\n\n'
            "
            '    for jxh in 0 to 2**(m+1)-1 loop\n' #k
            '      for jxl in 0 to 2**(m)-1 loop\n' #k
            '        firstdiff := resipmid(jxh,0,0)-resipmid(jxh,0,jxl);\n'
            '        lastdiff := resipmid(jxh,2**(m+u)-1,0) - resipmid(jxh,2**(m+u)-1,jxl);\n' #k,u,k,u
            '        n := ((firstdiff+lastdiff)/2.0)*real(2**(3*m+u+2));\n'
            '        n_vec := std_logic_vector(to_unsigned((integer(n)), m+1));\n' #k
            '        tableN(jxh*2**(m)+jxl+2**(2*m+1)) := n_vec;\n' #k,k
            '      end loop;\n'
            '    end loop;\n'
           )#ikke lenger riktig:%(k, k, k, u, k, u, k, k, u, k, k, u, k, u, k, u, k, k, k, u, k, u, k, k))

    fb.write('  end init_look_up_table;\n'
           'end newton_reci_table;')
    return None

#lager tabell pakken for oppførsels modellen ved kvaderat beregninger
#-----------------------------------------------------------------
def writepcksqr(numbit, j, k, u):
    #headder
    fsrp.write('library IEEE;\n'
           'use IEEE.std_logic_1164.all;\n'
           'use IEEE.numeric_std.all;\n'
           'use ieee.math_real.all;\n'
           'use work.divlib.all;\n\n')

    #konstanter
    fsrp.write('package newton_reci_table_sr is\n\n'
           '  constant q : natural := %s;\n' #antall bit totalt (numbit)
           '  constant k : natural := %s;\n' #antall riktige bit (j)
           '  constant g : natural := %s;\n' #antall garderingsbitt
           '  constant u : natural := %s;\n' #differangsen mellom xm og xl (u)
           '  constant m : natural := %s;\n\n' #samme som k her i skriptet
           %(numbit, j, 2, u, k))

    #deklarere typer og prosedyren
    fsrp.write('  subtype depthP is std_logic_vector(k+1 downto 0);\n' #j+2
           '  subtype depthN is std_logic_vector(m downto 0);\n\n' #k+1
           '  type look_up_tableP is array(2**(2*m+u+1) to 2**(2*m+u+2)-1) of depthP;\n' #2*k+u,2*k+u
           '  type look_up_tableN is array(2**(2*m+u+1) to 2**(2*m+u+2)-1) of depthN;\n\n' #2*k+u,2*k+u
           '  procedure init_look_up_table(tableP : inout look_up_tableP; tableN : inout look_up_tableN);\n'
           )#%(j+2, k+1, 2*k+u, 2*k+u, 2*k+u, 2*k+u))

    #avslutter pakken
    fsrp.write('end newton_reci_table_sr;')
    return None

#lager tabell pakke bodyen til oppførsels modellen
#-----------------------------------------------------------------
def writepckbdysqr(j, k, u):
    #hedder
    fsrb.write('library IEEE;\n'
           'use IEEE.std_logic_1164.all;\n'
           'use IEEE.numeric_std.all;\n'
```

```
        'use ieee.math_real.all;\n'
        'use work.divlib.all;\n\n')

    fsrb.write('package body newton_reci_table_sr is\n\n')

    fsrb.write('  procedure init_look_up_table(tableP : inout look_up_tableP; tableN : inout look_up_tableN) is\n'
        '    variable firstspread    : real;\n'
        '    variable lastspread     : real;\n'
        '    variable averagespread  : real;\n'
        '    variable spread         : real;\n'
        '    variable adjust         : real;\n'
        '    variable p              : real;\n'
        '    variable p_vec          : std_logic_vector(k+2 downto 0);\n' #j+2
        '    variable firstdiff      : real;\n'
        '    variable lastdiff       : real;\n'
        '    variable n              : real;\n'
        '    variable n_vec          : std_logic_vector(m downto 0);\n\n' # ommgjordt til full lengde. gammel =k+1
        )#%(j+2, k+1))

    fsrb.write('    function resipmid(xh : integer; xm : integer; xl : integer) return real is\n'
        '      variable temp1, temp2, temp3 : real;\n'
        '    begin\n'
        '      temp1 := real(xh)*real(2**(2*m+u))+real(xm)*real(2**m)+real(xl);\n'
        '      temp2 := sqrt(real(2**(3*m+u+1))+temp1);\n'
        '      temp3 := sqrt(real(2**(3*m+u+1))+temp1+1.0);\n'
        '      return 2.0*sqrt(real(2**(3*m+u+1)))/(temp2+temp3);\n'
        '    end resipmid;\n\n')

    fsrb.write('  begin\n'
        '    for ixh in 0 to 2**(m+1)-1 loop\n' #k
        '      firstspread := resipmid(ixh, 0, 0)-resipmid(ixh, 0, 2**(m)-1);\n' #k
        '      lastspread := resipmid(ixh, 2**(m+u)-1, 0)-resipmid(ixh, 2**(m+u)-1, 2**m-1);\n' #k,u,k,u,k
        '      averagespread := (firstspread + lastspread)/2.0;\n'
        '      for ixm in 0 to 2**(m+u)-1 loop\n' #k,u
        '        spread := resipmid(ixh, ixm,0)-resipmid(ixh,ixm,2**m-1);\n' #k
        '        adjust := (averagespread - spread)/2.0;\n'
        '        p := (resipmid(ixh,ixm,0)+adjust)*real(2**(3*m+u+2))-0.5;\n' #k,u
        '        p_vec := std_logic_vector(to_unsigned(integer(p),3*m+u+2));\n' #k,u
        '        tableP(ixh*2**(m+u)+ixm+2**(2*m+u+1)) := p_vec(3*m+u downto 0);\n' #k,u,k,u
        '      end loop;\n'
        '    end loop;\n\n'
        "
        '    for jxh in 0 to 2**(m+1)-1 loop\n' #k
        '      for jxl in 0 to 2**(m)-1 loop\n' #k
        '        firstdiff := resipmid(jxh,0,0)-resipmid(jxh,0,jxl);\n'
        '        lastdiff := resipmid(jxh,2**(m+u)-1,0) - resipmid(jxh,2**(m+u)-1,jxl);\n' #k,u,k,u
        '        n := ((firstdiff+lastdiff)/2.0)*real(2**(3*m+u+2));\n'
        '        n_vec := std_logic_vector(to_unsigned((integer(n)), m+1));\n' #k
        '        tableN(jxh*2**(m)+jxl+2**(2*m+1)) := n_vec;\n' #k,k
        '      end loop;\n'
        '    end loop;\n'
        )

    fsrb.write('  end init_look_up_table;\n'
        'end newton_reci_table_sr;')
    return None

#kjører newton raphson metoden
#---------------------------------------------------------------------
def newtonrap(y, z, itt, antbit):
    for i in range(itt):
        print '%f\n%f\n%f\n' %(y, z, itt)
        z = z*(2-y*z)
        z = float(int(z*2**(antbit+2)))/2**(antbit+2)
    return z


#Main
#---------------------------------------------------------------------
#hoved delen av programmet
print "starter"
bit = 16
j = 9
k = int((j+2)/3)
if((j+2)%3 == 0):
```

```
    u = -1
elif((j+2)%3 == 1):
    u = 0
else: u = 1

#lager variabler som skal brukes globalt
p = [float]*2**(k+1)
for l in range(2**(k+1)):
    p[l] = [float]*2**(k+u)
r = [float]*2**(k+1)
for l in range(2**(k+1)):
    r[l] = [float]*2**(k+u)
n = [float]*2**(k+1)
for l in range(2**(k+1)):
    n[l] = [float]*2**k
rnn = [float]*2**(k+1)
for l in range(2**(k+1)):
    rnn[l] = [float]*2**k

#åpner fil for skriving
try:
    #bytt w med a for å legge til isteden for overskrivning
    fi = open('newton_int_divider_bipartit_reci_rom_rtl.vhd', 'w') #ferdig tabell
    fe = open('newton_int_divider_bipartit_reci_rom_ent.vhd', 'w') #grensesnitt til ferdig tabell
    fs = open('newton_int_divider_bipartit_reci_rom_statistikk.txt', 'w') #statistikk
    fp = open('newton_reci_table_pck.vhd','w') #pakke til oppførses modell
    fb = open('newton_reci_table_bdy.vhd','w') #bodyen til pakken
    fsrp = open('newton_reci_table_sqr_pck.vhd','w') #pakka til kvaderatrot pakka
    fsrb = open('newton_reci_table_sqr_bdy.vhd','w') #bodyen til kvaderatrot pakka
except IOError:
    print 'Can\'t open file for writing.'
    sys.exit(0)

#lager vhdl filene
writeentetyfile(j)
writehedderrtl()
writePtable(j,k,u)   #her skjer også kalkuleringen av ptabellen.
writeNtable(k,u,j)   #N tabellen kalkuleres her.
writesubtraction(j,k,u)
writepck(bit, j, k, u)
writepckbdy(j, k, u)
writepcksqr(bit, j, k, u)
writepckbdysqr(j, k, u)


fe.close()
fi.close()
fs.close()
fp.close()
fb.close()
fsrp.close()
fsrb.close()

print "slutter\n"
```

Vedlegg 2.

Hedder fila til opførsels modellen av de bipartite tabellene. Eksempelet her er med j = 8.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.divlib.all;

package newton_reci_table is

  constant q : natural := 16;
  constant k : natural := 8;
  constant g : natural := 2;
  constant u : natural := 0;
  constant m : natural := 3;

  subtype depthP is std_logic_vector(k+1 downto 0);
  subtype depthN is std_logic_vector(m downto 0);

  type look_up_tableP is array(2**(2*m+u+1) to 2**(2*m+u+2)-1) of depthP;
  type look_up_tableN is array(2**(2*m+u+1) to 2**(2*m+u+2)-1) of depthN;

  procedure init_look_up_table(tableP : inout look_up_tableP; tableN : inout
look_up_tableN);
end newton_reci_table;
```

Vedlegg 3.

Bodyen til oppførsels moellen.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.divlib.all;

package body newton_reci_table is

  procedure init_look_up_table(tableP : inout look_up_tableP; tableN : inout look_up_tableN) is
    variable firstspread    : real;
    variable lastspread     : real;
    variable averagespread  : real;
    variable spread         : real;
    variable adjust         : real;
    variable p              : real;
    variable p_vec          : std_logic_vector(k+2 downto 0);
    variable firstdiff      : real;
    variable lastdiff       : real;
    variable n              : real;
    variable n_vec          : std_logic_vector(m downto 0);

    function resipmid(xh : integer; xm : integer; xl : integer) return real is
      variable temp : real;
    begin
      temp := real(xh)*real(2**(2*m+u))+real(xm)*real(2**m)+real(xl);
      return real(2**(3*m+u+1))/(real(2**(3*m+u+1))+temp+0.5);
    end resipmid;

  begin
    for ixh in 0 to 2**(m+1)-1 loop
      firstspread := resipmid(ixh, 0, 0)-resipmid(ixh, 0, 2**(m)-1);
      lastspread := resipmid(ixh, 2**(m+u)-1, 0)-resipmid(ixh, 2**(m+u)-1, 2**m-1);
      averagespread := (firstspread + lastspread)/2.0;
      for ixm in 0 to 2**(m+u)-1 loop
        spread := resipmid(ixh, ixm,0)-resipmid(ixh,ixm,2**m-1);
        adjust := (averagespread - spread)/2.0;
        p := (resipmid(ixh,ixm,0)+adjust)*real(2**(3*m+u+2))-0.5;
        p_vec := std_logic_vector(to_unsigned(integer(p),3*m+u+2));
        tableP(ixh*2**(m+u)+ixm+2**(2*m+u+1)) := p_vec(3*m+u downto 0);
      end loop;
    end loop;

    for jxh in 0 to 2**(m+1)-1 loop
      for jxl in 0 to 2**(m)-1 loop
        firstdiff := resipmid(jxh,0,0)-resipmid(jxh,0,jxl);
        lastdiff := resipmid(jxh,2**(m+u)-1,0) - resipmid(jxh,2**(m+u)-1,jxl);
        n := ((firstdiff+lastdiff)/2.0)*real(2**(3*m+u+2));
        n_vec := std_logic_vector(to_unsigned((integer(n)), m+1));
        tableN(jxh*2**(m)+jxl+2**(2*m+1)) := n_vec;
      end loop;
    end loop;
  end init_look_up_table;
end newton_reci_table;
```

Vedlegg 4.

Entytien til fila som inne holder den fulle bipartite tabellen. Eksempelet er med j = 8.

```
-- File name :  newton_int_divider_bipartit_reci_rom_rtl.vhd
-- Title     :  initiell reciprocal value
-- Module    :  initvalue
--
-- Purpose   :  create initial reciprocal value

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_signed.all;

entity newton_int_divider_bipartit_reci_rom is
  port(
      in1   :  in std_logic_vector(9 downto 0);
      out1  :  out std_logic_vector(9 downto 0)
            );
end newton_int_divider_bipartit_reci_rom;
```

Vedlegg 5.

VHDL fila som inne holder de bipartite tabellene. Eksempeler er med j = 8.

```
-- File name :  newton_int_divider_bipartit_reci_rom_rtl.vhd
-- Title     :  initiell reciprocal value
-- Module    :  initvalue
--
-- Purpose   :  create initial reciprocal value

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_signed.all;

architecture rtl of newton_int_divider_bipartit_reci_rom is
  function look_up_rom_Ptable(address : std_logic_vector(6 downto 0)) return std_logic_vector is
  begin
    case address is
      when "0000000" => return "1111111110"; --0
      when "0000001" => return "1111101110"; --1
      when "0000010" => return "1111011111"; --2
      when "0000011" => return "1111010000"; --3
      when "0000100" => return "1111000001"; --4
      when "0000101" => return "1110110010"; --5
      when "0000110" => return "1110100011"; --6
      when "0000111" => return "1110010101"; --7
      when "0001000" => return "1110000110"; --8
      when "0001001" => return "1101111000"; --9
      when "0001010" => return "1101101010"; --10
      when "0001011" => return "1101011101"; --11
      when "0001100" => return "1101001111"; --12
      when "0001101" => return "1101000010"; --13
      when "0001110" => return "1100110101"; --14
      when "0001111" => return "1100101000"; --15
      when "0010000" => return "1100011011"; --16
      when "0010001" => return "1100001110"; --17
      when "0010010" => return "1100000010"; --18
      when "0010011" => return "1011110110"; --19
      when "0010100" => return "1011101010"; --20
      when "0010101" => return "1011011110"; --21
      when "0010110" => return "1011010011"; --22
      when "0010111" => return "1011000111"; --23
      when "0011000" => return "1010111011"; --24
      when "0011001" => return "1010110000"; --25
      when "0011010" => return "1010100101"; --26
      when "0011011" => return "1010011010"; --27
      when "0011100" => return "1010001111"; --28
      when "0011101" => return "1010000101"; --29
      when "0011110" => return "1001111010"; --30
      when "0011111" => return "1001110000"; --31
      when "0100000" => return "1001100101"; --32
      when "0100001" => return "1001011011"; --33
      when "0100010" => return "1001010001"; --34
      when "0100011" => return "1001000111"; --35
      when "0100100" => return "1000111101"; --36
      when "0100101" => return "1000110100"; --37
      when "0100110" => return "1000101010"; --38
      when "0100111" => return "1000100001"; --39
      when "0101000" => return "1000010111"; --40
      when "0101001" => return "1000001110"; --41
```

```
when "0101010" => return "1000000101"; --42
when "0101011" => return "0111111100"; --43
when "0101100" => return "0111110011"; --44
when "0101101" => return "0111101010"; --45
when "0101110" => return "0111100010"; --46
when "0101111" => return "0111011001"; --47
when "0110000" => return "0111010000"; --48
when "0110001" => return "0111001000"; --49
when "0110010" => return "0111000000"; --50
when "0110011" => return "0110110111"; --51
when "0110100" => return "0110101111"; --52
when "0110101" => return "0110100111"; --53
when "0110110" => return "0110011111"; --54
when "0110111" => return "0110011000"; --55
when "0111000" => return "0110010000"; --56
when "0111001" => return "0110001000"; --57
when "0111010" => return "0110000000"; --58
when "0111011" => return "0101111001"; --59
when "0111100" => return "0101110001"; --60
when "0111101" => return "0101101010"; --61
when "0111110" => return "0101100011"; --62
when "0111111" => return "0101011100"; --63
when "1000000" => return "0101010100"; --64
when "1000001" => return "0101001101"; --65
when "1000010" => return "0101000110"; --66
when "1000011" => return "0100111111"; --67
when "1000100" => return "0100111001"; --68
when "1000101" => return "0100110010"; --69
when "1000110" => return "0100101011"; --70
when "1000111" => return "0100100100"; --71
when "1001000" => return "0100011110"; --72
when "1001001" => return "0100010111"; --73
when "1001010" => return "0100010001"; --74
when "1001011" => return "0100001010"; --75
when "1001100" => return "0100000100"; --76
when "1001101" => return "0011111110"; --77
when "1001110" => return "0011111000"; --78
when "1001111" => return "0011110010"; --79
when "1010000" => return "0011101011"; --80
when "1010001" => return "0011100101"; --81
when "1010010" => return "0011011111"; --82
when "1010011" => return "0011011010"; --83
when "1010100" => return "0011010100"; --84
when "1010101" => return "0011001110"; --85
when "1010110" => return "0011001000"; --86
when "1010111" => return "0011000011"; --87
when "1011000" => return "0010111101"; --88
when "1011001" => return "0010110111"; --89
when "1011010" => return "0010110010"; --90
when "1011011" => return "0010101100"; --91
when "1011100" => return "0010100111"; --92
when "1011101" => return "0010100001"; --93
when "1011110" => return "0010011100"; --94
when "1011111" => return "0010010111"; --95
when "1100000" => return "0010010001"; --96
when "1100001" => return "0010001100"; --97
when "1100010" => return "0010000111"; --98
when "1100011" => return "0010000010"; --99
when "1100100" => return "0001111101"; --100
when "1100101" => return "0001111000"; --101
when "1100110" => return "0001110011"; --102
when "1100111" => return "0001101110"; --103
when "1101000" => return "0001101001"; --104
when "1101001" => return "0001100100"; --105
when "1101010" => return "0001011111"; --106
when "1101011" => return "0001011011"; --107
when "1101100" => return "0001010110"; --108
when "1101101" => return "0001010001"; --109
when "1101110" => return "0001001101"; --110
when "1101111" => return "0001001000"; --111
when "1110000" => return "0001000011"; --112
when "1110001" => return "0000111111"; --113
when "1110010" => return "0000111010"; --114
when "1110011" => return "0000110110"; --115
when "1110100" => return "0000110010"; --116
```

```
    when "1110101" => return "0000101101"; --117
    when "1110110" => return "0000101001"; --118
    when "1110111" => return "0000100101"; --119
    when "1111000" => return "0000100000"; --120
    when "1111001" => return "0000011100"; --121
    when "1111010" => return "0000011000"; --122
    when "1111011" => return "0000010100"; --123
    when "1111100" => return "0000010000"; --124
    when "1111101" => return "0000001011"; --125
    when "1111110" => return "0000000111"; --126
    when "1111111" => return "0000000011"; --127
    when others => return "000000000";
  end case;
end look_up_rom_Ptable;

function look_up_rom_Ntable(address : std_logic_vector(6 downto 0)) return std_logic_vector is
begin
  case address is
    when "0000000" => return "0000"; --0
    when "0000001" => return "0010"; --1
    when "0000010" => return "0100"; --2
    when "0000011" => return "0110"; --3
    when "0000100" => return "1000"; --4
    when "0000101" => return "1001"; --5
    when "0000110" => return "1011"; --6
    when "0000111" => return "1101"; --7
    when "0001000" => return "0000"; --8
    when "0001001" => return "0010"; --9
    when "0001010" => return "0011"; --10
    when "0001011" => return "0101"; --11
    when "0001100" => return "0111"; --12
    when "0001101" => return "1000"; --13
    when "0001110" => return "1010"; --14
    when "0001111" => return "1100"; --15
    when "0010000" => return "0000"; --16
    when "0010001" => return "0010"; --17
    when "0010010" => return "0011"; --18
    when "0010011" => return "0101"; --19
    when "0010100" => return "0110"; --20
    when "0010101" => return "1000"; --21
    when "0010110" => return "1001"; --22
    when "0010111" => return "1010"; --23
    when "0011000" => return "0000"; --24
    when "0011001" => return "0001"; --25
    when "0011010" => return "0011"; --26
    when "0011011" => return "0100"; --27
    when "0011100" => return "0101"; --28
    when "0011101" => return "0111"; --29
    when "0011110" => return "1000"; --30
    when "0011111" => return "1001"; --31
    when "0100000" => return "0000"; --32
    when "0100001" => return "0001"; --33
    when "0100010" => return "0010"; --34
    when "0100011" => return "0100"; --35
    when "0100100" => return "0101"; --36
    when "0100101" => return "0110"; --37
    when "0100110" => return "0111"; --38
    when "0100111" => return "1001"; --39
    when "0101000" => return "0000"; --40
    when "0101001" => return "0001"; --41
    when "0101010" => return "0010"; --42
    when "0101011" => return "0011"; --43
    when "0101100" => return "0100"; --44
    when "0101101" => return "0110"; --45
    when "0101110" => return "0111"; --46
    when "0101111" => return "1000"; --47
    when "0110000" => return "0000"; --48
    when "0110001" => return "0001"; --49
    when "0110010" => return "0010"; --50
    when "0110011" => return "0011"; --51
    when "0110100" => return "0100"; --52
    when "0110101" => return "0101"; --53
    when "0110110" => return "0110"; --54
    when "0110111" => return "0111"; --55
    when "0111000" => return "0000"; --56
```

```
    when "0111001" => return "0001"; --57
    when "0111010" => return "0010"; --58
    when "0111011" => return "0011"; --59
    when "0111100" => return "0100"; --60
    when "0111101" => return "0101"; --61
    when "0111110" => return "0110"; --62
    when "0111111" => return "0110"; --63
    when "1000000" => return "0000"; --64
    when "1000001" => return "0001"; --65
    when "1000010" => return "0010"; --66
    when "1000011" => return "0011"; --67
    when "1000100" => return "0011"; --68
    when "1000101" => return "0100"; --69
    when "1000110" => return "0101"; --70
    when "1000111" => return "0110"; --71
    when "1001000" => return "0000"; --72
    when "1001001" => return "0001"; --73
    when "1001010" => return "0010"; --74
    when "1001011" => return "0010"; --75
    when "1001100" => return "0011"; --76
    when "1001101" => return "0100"; --77
    when "1001110" => return "0101"; --78
    when "1001111" => return "0110"; --79
    when "1010000" => return "0000"; --80
    when "1010001" => return "0001"; --81
    when "1010010" => return "0001"; --82
    when "1010011" => return "0010"; --83
    when "1010100" => return "0011"; --84
    when "1010101" => return "0100"; --85
    when "1010110" => return "0100"; --86
    when "1010111" => return "0101"; --87
    when "1011000" => return "0000"; --88
    when "1011001" => return "0001"; --89
    when "1011010" => return "0001"; --90
    when "1011011" => return "0010"; --91
    when "1011100" => return "0011"; --92
    when "1011101" => return "0011"; --93
    when "1011110" => return "0100"; --94
    when "1011111" => return "0101"; --95
    when "1100000" => return "0000"; --96
    when "1100001" => return "0001"; --97
    when "1100010" => return "0001"; --98
    when "1100011" => return "0010"; --99
    when "1100100" => return "0011"; --100
    when "1100101" => return "0011"; --101
    when "1100110" => return "0100"; --102
    when "1100111" => return "0100"; --103
    when "1101000" => return "0000"; --104
    when "1101001" => return "0001"; --105
    when "1101010" => return "0001"; --106
    when "1101011" => return "0010"; --107
    when "1101100" => return "0010"; --108
    when "1101101" => return "0011"; --109
    when "1101110" => return "0100"; --110
    when "1101111" => return "0100"; --111
    when "1110000" => return "0000"; --112
    when "1110001" => return "0001"; --113
    when "1110010" => return "0001"; --114
    when "1110011" => return "0010"; --115
    when "1110100" => return "0010"; --116
    when "1110101" => return "0011"; --117
    when "1110110" => return "0011"; --118
    when "1110111" => return "0100"; --119
    when "1111000" => return "0000"; --120
    when "1111001" => return "0001"; --121
    when "1111010" => return "0001"; --122
    when "1111011" => return "0010"; --123
    when "1111100" => return "0010"; --124
    when "1111101" => return "0011"; --125
    when "1111110" => return "0011"; --126
    when "1111111" => return "0100"; --127
    when others => return  "0000";
  end case;
 end look_up_rom_Ntable;
```

```
  signal temp_out : std_logic_vector(12 downto 0);
begin
  temp_out <= ("01" & look_up_rom_Ptable(in1(9 downto 3)) & '1') - ( "00000000" & look_up_rom_Ntable(in1(9 downto 6) &
in1(2 downto 0)) & '0');
  out1 <= temp_out(12 downto 3);
end rtl;
```