NTNU
Det skapende universitet

Konstruksjon av digital heltallsaritmetikk

Multiplikativ divisjon

Karl Marius Stafto

Master i elektronikk

Oppgaven levert: Juni 2008

Hovedveileder: Kjetil Svarstad, IET

Biveileder(e): Simen Gimle Hansen, Kongsberg Defence & Aerospace

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

Oppgaven tar utgangspunkt i resultater fra flere hovedoppgaver innen digital aritmetikk utført ved KDA, og edt skal utvikles en høyhastighets pipelinet regnemodul for divisjon og restberegning av heltall i FPGA teknologi for 16 og 64 bits operander.

Det skal utføres et forstudium om digital aritmetikk med vekt på multiplikativ divisjon, og en komparativ analyse av metoder og effektivitet gjennomføres. Et velbegrunnet valg av algoritme og metode skal så gjøres basert på dette. Man skal så spesifisere en regnemodulen som bør ha en fleksibel konfigurerbar arkitektur.

Neste del er å implementere en eller et fåtall alternative metoder i VHDL for syntese mot FPGA for heltalls divisjon, og spesielt for 16 og 64 bits operander.

Løsningen skal sammenlignes med andre relevante subtraktive og multiplikative løsninger.

Oppgaven gitt: 17. januar 2008
Hovedveileder: Kjetil Svarstad, IET

Forord

Detter er min masteroppgave innen elektronikk ved Norges Teknisk-Naturvitenskapelige Universitet (NTNU). Oppgaven er gitt av Kongsberg Defence & Aerospace (KDA). Oppgaven har blitt utført i løpet av vårsemesteret 2008 ved instituttet for elektronikk og telekommunikasjon ved NTNU, hvor jeg gikk studieretningen krets- og system konstruksjon..

Oppgaven gikk ut på å studere digital heltallsaritmetikk for å utvikle en pipelinet regnemodul for multiplikativ divisjon med heltalls operander på 16 og 32 bit. Regne modulen skal implementeres i VHDL for syntese mot FPGA. Jeg har designet 16 og 32 bits løsninger hvor divisjonsprosessen er partisjonert mellom ulike blokker som til sammen utgjør en pipelinet divisjonsmodul som kan implementeres i VHDL. Jeg valgte oppgaven fordi jeg syntes det virket spennende å sette seg inn i teorien bak den grunnleggende men krevende regneoperasjonen divisjon.

Jeg vil gjerne takke min veileder Kjetil Svarstad for god oppfølging og støtte gjennom hele vårsemesteret i forbindelse med arbeide på oppgaven.

Trondheim 2008.06.19

_____

Karl Marius Stafto

Sammendrag

Denne oppgaven beskriver hvilke algoritmer og metoder som kan benyttes til å utføre regneoperasjonen multiplikativ divisjon i maskinvare. Videre beskrives arkitekturen til de mest egnete metodene for å beregne divisorens resiprokal. Dette resiprokalet multipliseres så med dividenden for å produsere en kvotient.

Av de grunnleggende aritmetiske operasjonene addisjon, subtraksjon og multiplikasjon, er divisjon den som er mest krevende å utføre. Kongsberg Defence & Aerospace har gidd denne oppgaven med å undersøke mulighetene for å realisere en divisjonsmodul på en FPGA. Divisjonsmodulen skal være pipelinet, operere med 16 og 32 bits operander og basert på algoritmer for multiplikativ divisjon.

Det ble valgt å benytte Newton-Raphson-algoritmen for å iterere over en approksimert verdi av divisorens resiprokal. Denne approksimasjonsverdien hentes fra en bipartit oppslagstabell som adresseres med divisoren. Resiprokalene som er lagret i oppslagstabellen har en nøyaktighet på 1 ULP og Newton-Raphson-algoritmen dobler antall riktige bit for hver iterasjon. Dermed er det kun nødvendig med en iterasjon for å beregne en korrekt verdi av resiprokalet. Selve den iterative regneoperasjonen består av to sekvensielle multiplikasjoner og en subtraksjon. Arkitekturmessig er arbeidet med divisjonsprosessen fordelt på ulike hovedblokker som er sekvensielt sammenkoblet og som hver utfører sin del av prosessen. For hver blokk i de spesifiserte løsningene i denne rapporten, kommer divisjonsoperasjonen et steg nærmere en kvotient og en rest.

Det ble ikke tid til å implementere de spesifiserte løsningene i VHDL så det er ikke utarbeidet noen synteserapport for løsningene. Det burde imidlertid være relativt ukomplisert å utføre implementeringen basert på arkitekturene som er spesifisert i denne rapporten.

Ut i fra teoristudiet med egnete algoritmer og metoder, ble de metodene som virket best med tanke på ytelse benyttet til å spesifisere løsninger for 16 og 32 bits operander. Løsningene er like med unntak av den bipartite oppslagstabellen som får plass i RAM på FPGA for 16 bits operander, men blir så stor at den må legges i ekstern RAM for 32 bits operander. Ytelsesmessig er det ingen forskjeller på disse to løsningene, men realiseringen av løsningen for 32 bits operander er litt mer komplekst.

1 Innledning

Med et økende behov for informasjonsbehandling på minst mulig areal til lavest mulig kostnad, kombinert med høy ytelse, øker behovet for å behandle informasjonen digitalt.
Med de overnevnte kravene lønner det seg å konvertere all analog informasjonen til digitale signaler for så å behandle og prosessere de i det digitale domenet lengst mulig.

Stadig økende krav til ytelse og økende antall integrerte løsninger på interne digitale kretser og brikker, presser utviklingen av høyhastighets moduler videre fremover. Det blir stadig vanligere med hele komplette systemer på en brikke, såkalt system on chip (SoC), der kravet til effektiv arealutnyttelse og ytelse står sentralt.

Integrerte løsninger som utfører de grunnleggende aritmetiske operasjonene addisjon, subtraksjon og multiplikasjon er standard. Integrerte høyhastighets løsninger for tyngre aritmetiske operasjoner som divisjon er som regel ikke standard. I enkelte tilfeller lar det seg gjøre å utføre slike operasjoner i programvare, men i andre sammenhenger blir ytelsen da for lav og behovet for en integrert løsning i maskinvare er sterkt til stede.

Denne oppgaven beskriver hvilke metoder som kan benyttes til å implementere den krevende divisjonsoperasjonen i maskinvare. De fleste digitale systemer har en egen enhet for multiplikasjon som er effektiv. I den forbindelse kan denne multiplikatoren utnyttes til å utføre divisjon ved metoder som omskriver divisjon til multiplikasjon, også kalt multiplikativ divisjon. Denne oppgaven tar for seg metoder som egner seg for implementering på Field Programmable Gate Array (FPGA).

1.2 Målsetting

Målet med denne oppgaven er å beskrive teori og analysere metoder for multiplikativ heltalls divisjon. Ut i fra denne teorien skal det beskrives et begrunnet valg av algoritmer og metoder som egner seg for implementasjon i VHDL for syntese mot FPGA med 16 og 32 bits operander. Opprinnelig var det bestemt 16 og 64 bits operander, men etter dialog med Kongsberg Defence & Aerospace (KDA) som har gitt denne oppgaven, ble dette gjort om til 16, 32 og eventuelt 64 bits operander.

Det som skal utføres i oppgaven er:

- Forstudium om digital heltallsaritmetikk med vekt på multiplikativ divisjon
- Komparativ analyse av algoritmer og metoder og ut i fra det begrunne valg av algoritme og metode
- Spesifisere en pipelinet regnemodul for 16 og 32 bits operander
- Implementere en eller et fåtall alternative metoder i VHDL for syntese mot FPGA for multiplikativ heltallsdivisjon med 16 og 32 bits operander.

1.3 Hypotese

- Antar at den største utfordringen med multiplikativ divisjon blir å invertere divisoren
- Regner med å benytte en del tid til å finne og tolke algoritmer som allerede er utarbeidet for invertering av tall. Spesielt master og hovedfagsoppgaver som er utført her i landet innen digital aritmetikk.
- Antar at invertering kan i alle fall gjøres ved flere iterative beregninger, men at det kan bli tidkrevende å nødvendig og utføre gjentatte beregninger for å oppnå tilfredsstilende nøyaktighet.

1.4 Arbeidet med oppgaven

Arbeidet med oppgaven har vært delt i 3 deler:

1. Studiefase – Algoritmer og metoder for multiplikativ divisjon og initialverdier for iterative metoder ble beskrevet.
2. Implementasjonsfase – Løsning for multiplikativ divisjon med 16 bits operander ble beskrevet. Arkitektur med blokkskjema og tilstandsdiagram for 16 bits løsning ble utredet. Løsning for 32 bits operander basert på løsning for 16 bits operander ble skissert.
3. dokumentasjonsfase – Arbeidet med oppgaven ble beskrevet og rapport ble ferdigstilt

Arbeidsfordelingen mellom de ulike fasene har grovt fortenet seg slik: 60%, 20% og 20%.

1.5 Disposisjon av rapporten

1 Innledning: Innledning til oppgaven.

2 Bakgrunn: Bakgrunnen til oppgaven og problemstilling samt tidligere utførtarbeid.

3 Divisjonsalgoritmer: Inneholder en beskrivelse av flere multiplikative divisjonsalgoritmer samt kort hvordan subtraktive divisjonsalgoritmer fungerer. Gjennomgår iterative metoder.

4 Initialverdier: Inneholder en beskrivelse av metoder for å beregne initialverdier til iterative metoder for multiplikativ divisjon.

5 Implementasjon: Løsning for 16 og 32 bits operander spesifiseres.

6 Diskusjon: Valg av metoder og de spesifiserte løsningene diskuteres samt mulige forbedringer av løsningene blir gjenomgått. Forslag til videre arbeid blir også beskrevet.

7 Konklusjon: Oppsummering av hva som er oppnådd med masteroppgaven.

1.6 Lesing av rapporten

Det forutsettes at leseren har god kjennskap til generell digitalteknikk og hvordan binær aritmetikk kan realiseres i maskinvare.

I tillegg trengs noe grunnleggende kunnskap innen mattematikk. Dette for å forstå ligningene og utledningene av disse samt slutningene som blir trukket av de. Utledningene i denne rapporten ligger til grunn for teorien bak hvordan divisjon kan omskrives til multiplikasjon.

Det er forsøkt å forklare ting så grundig som mulig slik at det skal bli enklere å raskt forstå teorien. I tillegg er det forsøkt å benytte norske ord og termer for flere engelske faguttrykk fra artikler og teori stoff som denne oppgaven baserer seg på.

2 Bakgrunn

Kongsberg Defence & Aerospace (KDA) som har gitt denne oppgaven, produserer utstyr til blant annet militære formål. Deriblant avlyttningssikkert digitalt radiosambandsutstyr. Dette utstyret utfører en god del digital signalbehandling, med høye krav til ytelse ved regneoperasjoner.

Divisjon er en grunnleggende og viktig del innen aritmetiske operasjoner. Det optimale er så hurtig divisjon som mulig på så lite areal som mulig. En divisjonsmodul er anvendelig i andre apparater og applikasjoner. Multiplikativ divisjon som denne oppgaven omhandler, benytter en multiplikator som allerede finnes i en del utstyr.

I denne oppgaven er 2 iterative algoritmer for multiplikativ divisjon gjennomgått. Goldschmidt- og Newton-Raphson-algoritmene. Newton-Raphson-algoritmen dobler antallet riktige bit for hver iterasjon og egner seg dermed bra til multiplikativdivisjon.

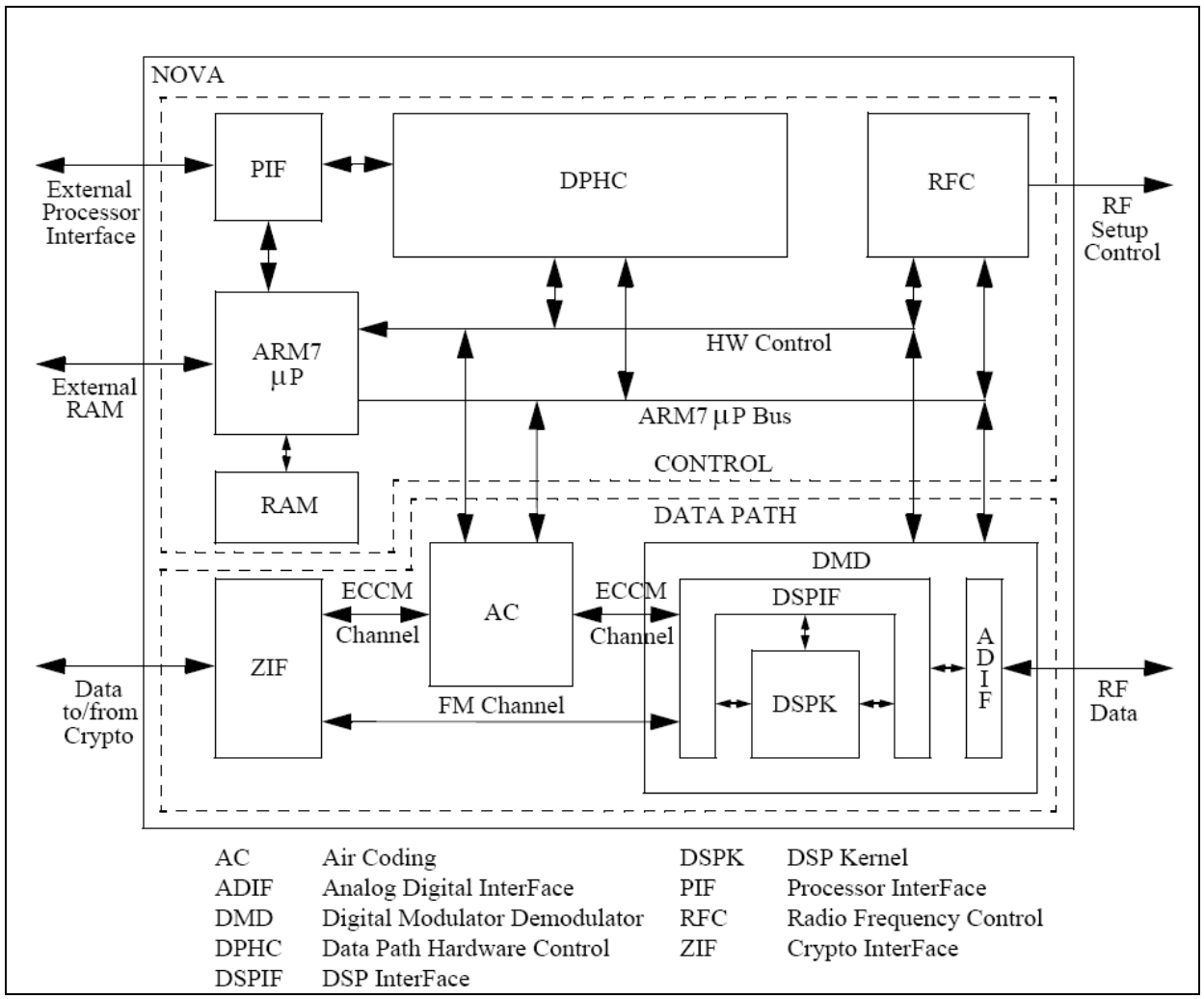
De iterative algoritmene trenger en nøyaktig approksimasjon for en initialverdi. For å redusere antallet iterasjoner som kreves må den initelle approksimasjonen være så nøyaktig som mulig. Dermed kreves det en metode for å skaffe en initialverdi så hurtig som mulig uten for mange operasjoner. Dette blir en utfordring når det stilles krav til initialverdiens nøyaktighet. Initialverdiens nøyaktighet er viktig for å redusere det påfølgende antallet iterasjoner som er nødvendig for at sluttverdien skal bli korrekt.

Tidligere arbeid som er utført på området er masteroppgavene [1][2][3] som også er gitt av KDA. Disse oppgavene har vært til støtte for arbeidet med denne oppgaven. De gir en innføring i divisjons- og kvadratrotalgoritmer for bruk i et prosjekt kalt Lett Felt Radio (LFR) ved KDA. Dette prosjektet er en ny type håndholdt radio som KDA har jobbet med for Det Norske Forsvaret. For prosjektet ble en egen Application-Spesific Integrated Circuit (ASIC) utviklet av KDA. Denne ble kalt NOVA, og innholdt moduler for digital modulasjon og demodulasjon, grensesnitt mot eksterne prosessorer og en mikroprosessor. Blokkskjema av NOVA kretsen vises i figur 1.

Den digitale signal prosessorkjernene (DSPK) i nova kretsen som vises på figur 1, benytter en radiks-2 SRT-divisjonsalgoritme for å utføre divisjon. Dette er en av flere subtraktive divisjonsalgoritmer[9][20]. SRT-algoritmen defineres etter hvilken *radiks r* som benyttes. Radiks betyr i denne sammenhengen antall bit som dannes av svaret på divisjonsregnestykket (kvotienten) per iterasjon. En radiks-r SRT algoritme genererer $\log_2 r$ bit av kvotienten per iterasjon. Dermed vil det ta totalt $\left(\frac{n}{\log_2 r}\right)$ antall iterasjoner for å beregne *n* antall kvotientbit. Radiks-2 SRT-algoritmen generer ett kvotientbit per iterasjon.

Da SRT-algoritmen er forholdsvis kostnadseffektiv, er den blitt benyttet i flere mikroprosessorer oppigjennom årene. Av kjente implementeringer kan Pentium 4(radiks-2 SRT) [4], Pentium[5], Pentium II og Pentium III (radiks-4 SRT), UltraSPARC [6] og UltraSPARC-II (radiks-8 SRT) nevnes.

For å tilfredsstille kravene til ytelse er 16 radiks-2 SRT-trinn koblet sammen i serie i DSPK. Hvert trinn beregner ett bit av kvotienten. Denne sammenkoblingen gir en svært dyp og kompleks kombinatorisk krets som krever et stort areal.



Figur 1: Blokkskjema for oppbygningen til NOVA-kretsen. Bildet er hentet fra[1].

2.1 Problemstilling

Da kapasiteten på FPGAer har økt de siste årene har det blitt interessant å undersøke mulighetene for å implementere systemet på en FPGA, som er billigere å utvikle enn ASIC. Deriblant trengs en divisjonsmodul slik som denne oppgaven går ut på å finne en egnet løsning for.

For å få økt ytelse ved divisjon, er det interessent å se på andre metoder og logaritmer enn den subtraktive SRT-algoritmen som er benytt frem til nå. KDA ønsker å utarbeide en løsning som benytter allerede eksiterende multiplikatorer implementert på FPGA til multiplikativ divisjon. En slik løsning bør være pipelinet og operere med operandstørrelser på 16 og 32 bit.

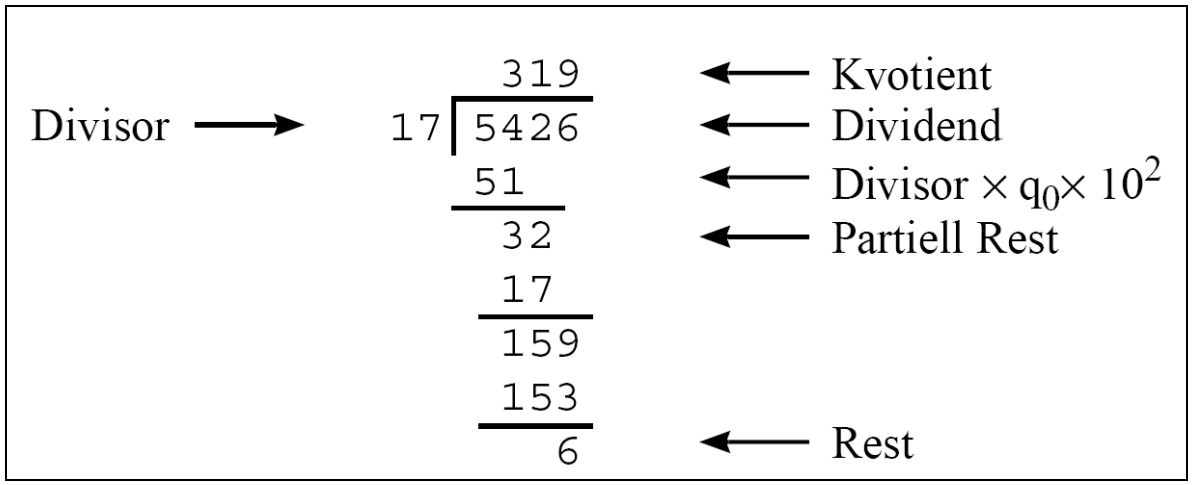
3 Divisjonsalgoritmer

Divisjonsalgoritmer kan deles inn i to hovedgrupper avhengig av hvilke operasjoner de er basert på. Subtraktive og multiplikative divisjonsalgoritmer hvor navnene indikerer hvilken type operasjoner beregningene er basert på. Den subtraktive metoden ligner mye på den såkalte penn-og-papir metoden som er basert på en serie subtraksjoner hvor et siffer av kvotienten eller svaret fremskaffes for hver subtraksjon. Kvotienten angir her hvor mange ganger divisoren er subtrahert fra dividende uten at resultatet ble negativt. Multiplikative algoritmer genererer kvotienten ved å multiplisere resiprokalet eller den inverterte til divisoren, med dividenden. Med multiplikative divisjonsalgoritmer blir utfordringen å fremskaffe resiprokalet til divisoren.

3.1 Subtraktive divisjonsalgoritmer

Denne formen for divisjon er basert på en serie subtraksjoner helt til kvotienten er bestemt og en eventuell rest er bestemt.

Subtraktiv divisjon ligner mye den tradisjonelle penn-og-papir metoden å utføre divisjon på. En partiell rest blir beregnet for hver gang divisoren trekkes fra dividenden, og denne partielle resten benyttes i neste trinn av divisjonsregnestykket dvs. divisoren subtraheres fra den partielle resten ved neste trinn. På dette viset blir et nytt siffer i kvotienten beregnet for hvert trinn. For hvert trinn må det imidlertid sjekkes om den partielle resten har blitt negativ. En slik divisjonsmetode vil kreve en subtraksjon og en sjekk av den partielle resten for hvert trinn eller gjennomkjøring av algoritmen. Denne fremgangsmåten vil kreve mange trinn eller iterasjoner før kvotienten er bestemt. Antallet nødvendige iterasjoner er lineært avhengig av antall siffer eller bit til dividenden, dette gjør at subtraktive divisjonsalgoritmer konvergerer langsomt. Dette gir en særdeles lang og omstendelig prosess for å få bestemt kvotienten ved regning med store dividender.



Figur 2: Illustrasjon av Newton-Raphsons metode med den ikke-lineære funksjonen f(z) og dens tangent i punktet z_i. Ser her at tangentens nullpunkt z_{i+1} er en bedre tilnærming etter i iterasjoner til f(z) sitt faktiske nullpunkt z.

Det finns en del varianter av subtraktive algoritmer som alle er basert på en serie subtraksjoner og kontrollering av fortegnet til kvotienten for hver subtraksjon. Disse algoritmene har forholdsvis enkle iterasjonsskjemaer, men antall nødvendige iterasjoner øker allikevel lineært med antall siffer i operandene. Enkelte subtraktive algoritmer er i stand til å generere opptil noen få bit av kvotienten per iterasjon, men disse er også lineært avhengig av sifrene i operandene og konvergere også langsomt ved store operander.
Restoring divisjon, Non-restoring divisjon, Performing divisjon, Non-performing divisjon, Array divisjon, SRT divisjon er kjente subtraktive algoritmer [9].

3.2 Multiplikative divisjonsalgoritmer

I likhet med subtraktiv divisjon, gjenspeiler navnet multiplikativ divisjon hvilken type operasjon som ligger til grunn for divisjonsmetoden. Multiplikative algoritmer tar utgangspunkt i at kvotienten Q er lik produktet av dividenden X og resiprokalet (den inverterte) av divisoren Y.

$$Q = \frac{X}{Y} = X \cdot \frac{1}{Y} \qquad (1)$$

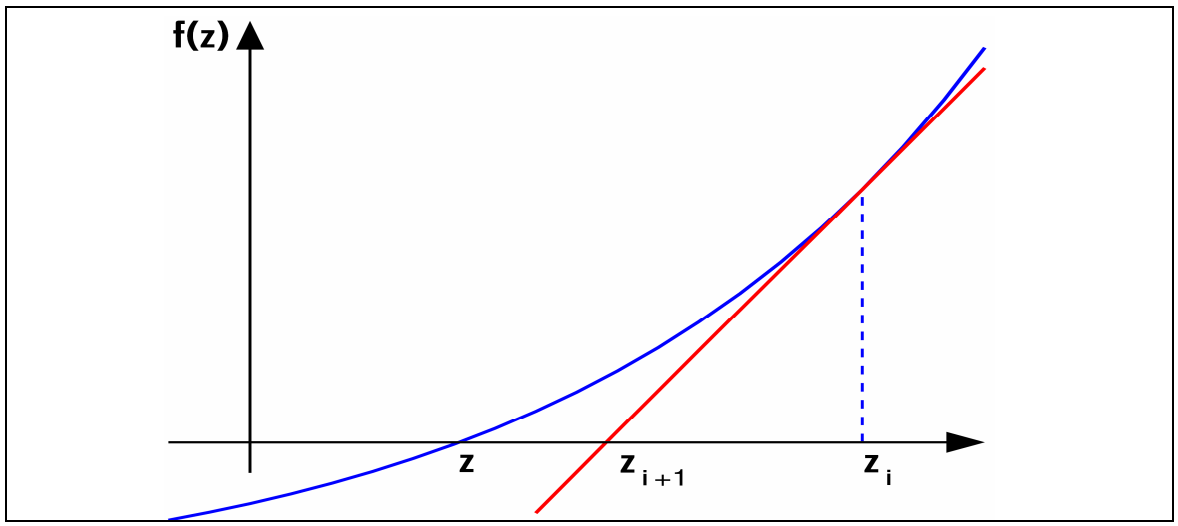
Med denne metoden blir utfordringen å beregne eller finne resiprokalet av Y, så hurtig som mulig. Ved denne fremgangsmåten er det som skal beregnes resiprokalet bare avhengig av en operand i motsetning til i subtraktiv divisjon. Newton-Raphsons metode og Goldschmidts metode er to vel kjente metoder for å beregne resiprokalet.

3.2.1 Newton-Raphson

Newton-Raphson er en iterativ tilnærmingsmetode for å finne en ikke-lineær funksjon sitt nullpunkt. Metoden ble først beskrevet av I. Newton og senere renskrevet og bearbeidet av Raphson. Denne metoden produserer en serie tilnærmede verdier til funksjonens nullpunkt som konvergerer eksponentielt med antall iterasjoner mot funksjonens faktiske nullpunkt.
Newton-Raphson metode har følgende iterative formel (se formel 2).

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

I formel 2 er $f(z)$ den ikke-lineære funksjonen og $f'(z)$ er funksjonens første deriverte og z_i er en tilnærming til funksjonen $f(z)$ sitt nullpunkt. Ser også at z_{i+1} er en mer presis tilnærming til nullpunktet z. Den iterative prosessen starter med en antatt passende startverdi, z_0. Dersom z_0 oppfyller konvergenskravene vil z_{i+1} konvergere mot det faktiske nullpunktet z, og gi en bedre approksimasjon for hver iterasjon. Prisnippet er skissert grafisk i figur 3.



Figur 3: Illustrasjon av Newton-Raphsons metode med den ikke-lineære funksjonen f(z) og dens tangent i punktet z_i. Ser her at tangentens nullpunkt z_{i+1} er en bedre tilnærming etter i iterasjoner til f(z) sitt faktiske nullpunkt z.

Utledning av Newton-Raphson- metode
Newton-Raphson metoden tar utgangspunkt i definisjonen på den deriverte:

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

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

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

Anvendelse av Newton-Raphsons-metoden til divisjon
Med utgangspunkt i ligning 3, kan Newton-Raphson metoden benyttes til å beregne resiprokalet $\frac{1}{Y}$ til divisoren *Y*. Velger funksjonen $f(z)$ lik:

$$f(z) = \frac{1}{z} - Y \qquad (4)$$

Funksjonens første deriverte blir da:

$$f'(z) = -\frac{1}{z^2} \qquad (5)$$

Funksjonens nullpunkt finnes når z er lik resiprokalet, $z = \frac{1}{Y}$

$$f(z) = 0$$
$$0 = \frac{1}{z} - Y$$
$$z = \frac{1}{Y} \tag{6}$$

Formel 4, 5 satt inn i formel 3:

$$z_{i+1} = z_i - \frac{\frac{1}{z_i} - Y}{-\frac{1}{z^2}}$$
$$z_{i+1} = z_i - \frac{\frac{z_i^2}{z_i} - Yz_i^2}{-\frac{z_i^2}{z_i^2}}$$
$$z_{i+1} = z_i + z_i - Yz_i^2$$
$$z_{i+1} = 2z_i - Yz_i^2$$
$$z_{i+1} = z_i(2 - Yz_i) \tag{7}$$

Beregning av resiprokalet som her er lik z_{i+1} i formel 7, består her av en subtraksjon og to multiplikasjoner for hver iterasjon. Etter et tilstrekkelig antall iterasjoner som sikrer tilstrekkelig nøyaktighet av resiprokalet, skal resiprokalet multipliseres med dividenden for å få utført divisjonen fra formel 1.

Resiprokalets feilrate ved approksimasjon

Med denne metoden får det beregnete resiprokalet z_{i+1} en feilrate E_{i+1} som er lik differansen mellom z_{i+1} og det faktiske resiprokalet $\frac{1}{Y}$ etter hver iterasjon.

$$E_{i+1} = \frac{1}{Y} - z_{i+1}$$

$$E_{i+1} = \frac{1}{Y} - z_i(2 - Yz_i)$$

$$E_{i+1} = \frac{1}{Y} - 2z_i + Yz_i^2$$

$$E_{i+1} = Y\left(\frac{1}{Y^2} - \frac{2Z_i}{Y} + z_i^2\right)$$

$$E_{i+1} = Y\left(\frac{1}{Y} - z_i\right)$$

$$E_{i+1} = YE_i^2 \qquad (8)$$

Ser fra ligning (8) at den absolutte feilen vil konvergere kvadratisk mot null for hver iterasjon dersom $|E_o| < 1$. Sagt på en annen måte vil E_{i+1} konvergere for verdier av startverdien z_0 som gir at $E_i \to 0$ når $i \to \infty$. Finner verdier for z_0 som oppfyller konvergenskravene:

$$E_{i+1} = YE_i^2$$

$$E_i = YE_{i-1}^2$$

$$E_{i-1} = YE_{i-2}^2$$

$$E_{i-1}^2 = Y^2E_{i-2}^4$$

$$\Rightarrow E_i = Y(Y^2(Y^4(...(Y^{2^{i-1}}E_0^{2^i}))...))) \qquad (9)$$

Eksponenten til Y følger en geometrisk rekke.

$$2^0 + 2^2 + 2^4 + ... + 2^{2^{i-1}} = \frac{1-2^i}{1-2} = 2^i - 1 \qquad (10)$$

Resultatet fra ligning (10) gir da at feilen etter i iterasjoner kan utrykkes slik:

$$E_i = Y^{2^i-1}E_0^{2^i}$$

$$E_i = Y^{2^i-1}\left(\frac{1}{Y} - z_0\right)^{2^i}$$

$$E_i = Y^{2^i-1}\frac{(1-Yz_0)^{2^i}}{Y^{2^i}}$$

$$E_i = \frac{Y^{2^i}}{Y}\frac{(1-Yz_0)^{2^i}}{Y^{2^i}}$$

$$E_i = \frac{(1-Yz_0)^{2^i}}{Y} \qquad (11)$$

For at $E_i \to 0$ når $i \to \infty$ slik at prosedyren konvergerer, må telleren i ligning (11) gå mot null, og det skjer når $\left|1 - Yz_0\right| < 1$. Det gir følgende området for z_0 som oppfyller konvergenskravene:

$$1 - Yz_0 = 1$$
$$Yz_0 = 0$$
$$z_0 = 0$$

$$1 - Yz_0 = -1$$
$$Yz_0 = 2$$
$$z_0 = \frac{2}{Y}$$

$$\Rightarrow 0 < z_o < \frac{2}{Y} \qquad (12)$$

Denne metoden gir dobbelt så mange riktige siffer for hver iterasjon. Antall iterasjoner som trengs for å oppnå ønsket nøyaktighet, har følgende funksjon som er avhengig av nøyaktigheten på startverdien z_o.

I = antall nødvendige iterasjoner
p = antall korrekte bit i startverdien
q = ønsket nøyaktighet

$$p \cdot 2^I = q$$
$$\log_2\left(p \cdot 2^I\right) = \log_2(q)$$
$$\log_2(p) + \log_2(2) \cdot I = \log_2(q)$$
$$I = \log_2(q) - \log_2(p)$$
$$I = \log_2\left(\frac{q}{p}\right) \qquad (13)$$

Oppsummering av Newton-Raphson-metoden
Newton-Raphson-metoden gjør multiplikativ divisjon mulig ved en effektiv iterativ approksimeringsmetode som gir en kvadratisk konvergering av resiprokalet til divisoren. Antall iterasjoner som benyttes bør ballanseres med tanke resiprokalets nøyaktighet, ytelse og kostnad. Startverdien som benyttes i første iterasjon, har stor betydning for hvor optimal konvergeringsprosessen blir.

3.2.2 Goldschmidt

Goldschmidts algoritme benytter Taylor-rekkeutvikling til å approksimere funksjonsverdier. Den generelle formelen for en funksjon $f(z)$ i punktet $z = p$ er gitt ved ligning 14, der antall ledd bestemmer nøyaktigheten.

$$f(z) = f(p) + f'(z)(z-p) + \frac{f''(z)}{2!}(z-p)^2 + ... + \frac{f^{(n)}(p)}{n!}(z-p)^n + ... \qquad (14)$$

Denne algoritmen kan benyttes til divisjon ved å finne funksjonen til resiprokalet, $f(z) = \frac{1}{Y}$

Dermed kan kvotienten til en divisjon beregnes slik:

$$Q = \frac{X}{Y} = X\frac{1}{Y} = X \cdot f(z)$$

Av begrensningsmessige grunner velges $f(z) = \frac{1}{1+z}$ og $p = 0$ til ligning 14. Videre antas operander i området fra og med 0,5 og opp til 1 for å gjøre fremstillingen lettere. Lar divisoren Y være lik $1+z$, $-0{,}5 \le z < 0$. Resiprokalet til Y kan da tilnærmes med Maclaurin-rekken for $\frac{1}{1+z}$:

$$f(z) = \frac{1}{1+z} = 1 - z + z^2 - z^3 + z^4 - ...$$

Høyresiden til utrykket over kan uttrykkes slik:

$$\frac{1}{1+z} = (1-z)(1+z^2)(1+z^4)..(1+z^{2i}) \qquad (15)$$

Dermed kan kvotienten uttrykkes slik:

$$Q = X \cdot f(z) = X \cdot ((1-z)(1+z^2)(1+z^4)..(1+z^{2i})) \qquad (16)$$

Hvert ledd i rekkeutviklingen $(1-z)(1+z^2)(1+z^4)..(1+z^{2i})$ kan betraktes som en sekvens med produkter $r_o \cdot r_1 \cdot ... \cdot r_n$ som sørger for at divisoren Y konvergerer mot 1, samtidig som dividenden X konvergerer mot kvotienten Q. Dette kan uttrykkes slik:

$$Q = \frac{X}{Y} = \frac{X \cdot r_o \cdot r_1 \cdot ... \cdot r_n}{Y \cdot r_o \cdot r_1 \cdot ... \cdot r_n} = \frac{X\prod_{i=0}^{n} r_i}{Y\prod_{i=0}^{n} r_i}$$

Hvor r_i er normaliseringsfaktoren som for en tilstrekkelig stor *n* sørger for at:

$$Y \cdot \prod_{i=0}^{n} r_i \to 1 \text{ og } X \prod_{i=0}^{n} r_i \to Q \text{ når } \prod_{i=0}^{n} r_i \to \frac{1}{Y}$$

Ut i fra dette kan det dannes en itererende produktrekke for divisoren:

$$Y_i = Y \cdot r_o \cdot r_1 \cdot \ldots \cdot r_i \text{ eller } Y_i = Y_{i-1} \cdot r_i$$

Der r_i representerer ledd nummer i, på høyre side av ligning 15. Dermed kan uttrykket skrives slik på generell form:

$$Y_i = Y_{i-1} \cdot r_i = \left(1 - z^{2^i}\right)\left(1 + z^{2^i}\right) = 1 - z^{2^{i+1}} \tag{17}$$

Fra ligning 17 kommer det fram at Y_i konvergerer kvadratisk mot 1 når i øker, fordi vi har antatt at $0{,}5 \le Y < 1$, og dermed vil $Y_{i-1} < Y_i$. Dermed vil algoritmen utvikle rekken ved å multiplisere inn et nytt ledd i *i*te iterasjon på formen $1 + z^{2^i}$.
Leddet r_i i ligning 17 kan dannes direkte ved å ta toer komplementet av Y_{i-1}:

$$r_i = 2 - Y_{i-1} = 2 - \left(1 - z^{2^i}\right) = 1 + z^{2^i}$$

Dermed kan leddene som skal multipliseres inn i rekken beregnes ved å utføre en toer-komplementering og en multiplikasjon.

Det generelle utrykket for kvotienten blir da:

$$X_i = X_{i-1} \cdot r_i$$

Hvor X_i er kvotienten etter *i* iterasjoner.

Ved hjelp av en initiell approksimasjon av resiprokalet til divisoren, kan antallet iterasjoner reduseres for å få utført divisjonen. Dette gjøres ved å multiplisere dividenden og divisoren med en initiell startverdi r_0 for resiprokalet. Denne initial verdien for r_0 gitt ved:

$$r_0 \approx \frac{1}{Y} \approx (1 - z)\left(1 + z^2\right)\left(1 + z^4\right)..$$

Dette vil tilsvare de første leddene på høyre side i ligning 15, slik at $Y_0 = Y \cdot r_0$ og $X_0 = X \cdot r_0$. Nøyaktigheten til den initielle startverdien for r_0 bestemmer hvor mange iterasjoner som trengs for å utføre divisjonen.

Oppsummering av Goldschmidts metode

Goldschmidts algoritme består av to multiplikasjoner og en toer-komplementering. Multiplikasjonene er ikke avhengig av hverandre og kan derfor utføres parallelt i motsetning til multiplikasjonene i Newton-Raphson algoritmen hvor multiplikasjonene må utføres sekvensielt. Feilanalysen til Goldschmidts algoritme er langt mer kompleks enn det som er tilfellet med Newton-Raphson algoritmen hvor en feil i en iterasjon blir rettet opp i neste iterasjon. Da Goldschmidts algoritme akkumulerer feil for hver iterasjon blir det nødvendig med en feilanalyse for hver iterasjon[7].

4 Initialverdier

Startverdien eller z_o som benyttes ved første iterasjon i ligning (7) har stor betydning for den approksimerte verdien av resiprokalet og prosessens ytelse. En startverdi med mange siffer lik det korrekte resiprokalet, vil redusere antallet iterasjoner som trengs for å oppnå ønsket nøyaktighet på det approksimerte resiprokalet.

For å unngå at resiprokalet får en særdeles liten verdi, blir *Y* normaliseres til $[1,2)$ slik at resiprokalet til divisoren $\frac{1}{Y}$ finnes i mengden $\left[\frac{1}{2},1\right)$. Dette følger også IEEE standarden 754 for binære flyttall.

De tre vanligste metodene for å bestemme initial verdien til resiprokalet ved multiplikativ divisjon er:

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

For å kunne sammenligne effektiviteten til disse metodene benyttes den relative feilen $e(Y)$ samt antall korrekte bit metodene gir. Den relative feilen $e(Y)$ er gitt av forholdet mellom feilen E_{i+1} delt på det eksakte resiprokalet $\frac{1}{Y}$.

$$e(Y) = \left| \frac{\frac{1}{Y} - z_0}{\frac{1}{Y}} \right| = |1 - Yz_0| \qquad (18)$$

I ligning 18 er *Y* divisoren og z_o er den initelle approksimasjonen til resiprokalet. Med en presisjon på *P* riktige bit, vil den initelle approksimasjonen til $\frac{1}{Y}$ ha en relativ feil $e(Y) < 2^{-P}$
Den relative feilen blir lik forholdet mellom feil og riktigsvar, e.

$$|e_{i+1}| = \left| \frac{\frac{1}{Y} - z_i}{\frac{1}{Y}} \right| = \left| \frac{E_{i+1}}{\frac{1}{Y}} \right| = |Y \cdot E_{i+1}| = |Y(Y \cdot E_i^2)| = |Y^2 E_i^2| = |YE_i|^2 = |e_i|^2$$

Antall riktige bit *p*, i en startverdi regnes ut ved hjelp av den relative feilraten på følgende måte:

$$e(Y) = 2^{-p}$$
$$\log_2(e) = -p \cdot \log_2(2)$$
$$p = -\frac{\log_2(e)}{\log_2(2)} \qquad (19)$$

4.1 Konstant startverdi

Den aller enkleste metoden er å benytte en konstant startverdi uansett hvilken divisor *Y* som benyttes, men dette gir også størst feil siden startverdien vil ha færre riktige bit. En initiell approksimasjon $z_0 = \frac{1}{2}$ vil gi en tilnærming som gir minimalt med logikk siden produktet mellom *Y* og z_0 kan realiseres ved å foreta ett høyreskift av *Y*. Med en divisor lik 1 blir den relative feilen $e(Y) = 1 - \frac{1}{2}$ og vil få en maksimal verdi på $\frac{1}{2}$ for $1 \le Y < 2$. Ved å benytte ligning 19 kan presisjonen på startverdien regnes ut (se ligning 20 hvor presisjonen blir regnet ut til å bli ett bit):

$$p = -\frac{\log(e)}{\log(2)} = -\frac{\log_2\left(\frac{1}{2}\right)}{\log_2(2)} = -\frac{\frac{1}{\ln(2)} \cdot \ln\left(\frac{1}{2}\right)}{\frac{1}{\ln(2)} \cdot \ln(2)} = 1 \qquad (20)$$

Ved å ta utgangspunkt i grenseverdiene til *Y* og si at den relative feilen skal være lik for å finne en verdi $Y = 1 \text{ og } Y = 2$, blir det mulig å finne en startverdi for z_0 som gir lavere relativ feilrate.

$$e(1) = e(2)$$

$$|1 - z_0| = |1 - 2z_0| \Rightarrow z_0 = \frac{2}{3} \vee 0$$

For at konvergenskravene skal være møtt må $z_0 = \frac{2}{3}$. Dette gir en relativ feilrate $e(Y)$ på $\frac{1}{3}$.

Med feilraten på $\frac{1}{3}$ innsatt i ligning (19) oppnås en initiell presisjon på 1,585 bit

4.2 Stegvis tilnærming

Ved å dele divisorens tallområde inn i delintervaller med ett resiprokal for hvert av delintervallene, vil den initielle verdien for resiprokalet få en langt større nøyaktighet. Denne metoden kalles stegvis tilnærming og er illustrert i figur 4. For å tilpasse denne metoden til det binære tallsystemet er det hensiktsmessig at antallet delintervaller er en potens med tallet to som grunntall. Ved å benytte de ledende bitene i divisoren til å bestemme hvilken konstant som skal benyttes for den gjeldende divisoren. Divisorens intervall $[1,2)$ deles inn i 2^k like store delintervaller inndelt på følgende måte:

$$\left[\frac{\left(2^k + j\right)}{2^k}, \frac{\left(2^k + j + 1\right)}{2^k}\right) \qquad , \; j \in \left\{0,1,...,2^k - 1\right\}$$

Konstantene for hvert delintervall bør legges til delintervallenes midtpunkt for å oppnå høyest mulig nøyaktighet på resiprokalene. Delintervallets midtpunkt blir da:

$$Y = \frac{\left(2^{k+1} + 2j + 1\right)}{2^{k+1}} \tag{21}$$

Dermed blir resiprokalets verdi:

$$\frac{1}{Y} = \frac{2^{k+1}}{\left(2^{k+1} + 2j + 1\right)} \tag{22}$$

Dersom divisoren er i enden av delintervallet, blir feilen aller størst. Dette kan sees på figur 4 hvor verdien til resiprokalet avviker mest i forhold til grafen dersom divisoren befinner seg i enden av et delintervall. Den relative feilen er gitt ved ligning (18) hvor z_0 er startverdien innen et gitt delintervall.
Dermed får den relative feilen for nedre og øvre grensepunktene for et delintervall følgende verdi:

$$\begin{aligned} e(Y)_{nedre} &= \left|1 - \frac{2^k + j}{2^k} \cdot \frac{2^{k+1}}{2^{k+1} + 2j + 1}\right| \\ e(Y)_{nedre} &= \left|1 - \frac{2^{2k+1} + j2^{k+1}}{2^{2k+1} + j2^{k+1} + 2^k}\right| \\ e(Y)_{nedre} &= \left|\frac{2^k}{2^{2k+1} + j2^{k+1} + 2^k}\right| \end{aligned} \tag{23}$$

$$\begin{aligned} e(Y)_{øvre} &= \left|1 - \frac{2^k + j + 1}{2^k} \cdot \frac{2^{k+1}}{2^{k+1} + 2j + 1}\right| \\ e(Y)_{øvre} &= \left|1 - \frac{2^{2k+1} + j2^{k+1} + 2^{k+1}}{2^{2k+1} + j2^{k+1} + 2^k}\right| \end{aligned}$$

$$e(Y)_{øvre} = \left| -\frac{2^k}{2^{2k+1} + j2^{k+1} + 2^k} \right| \qquad (24)$$

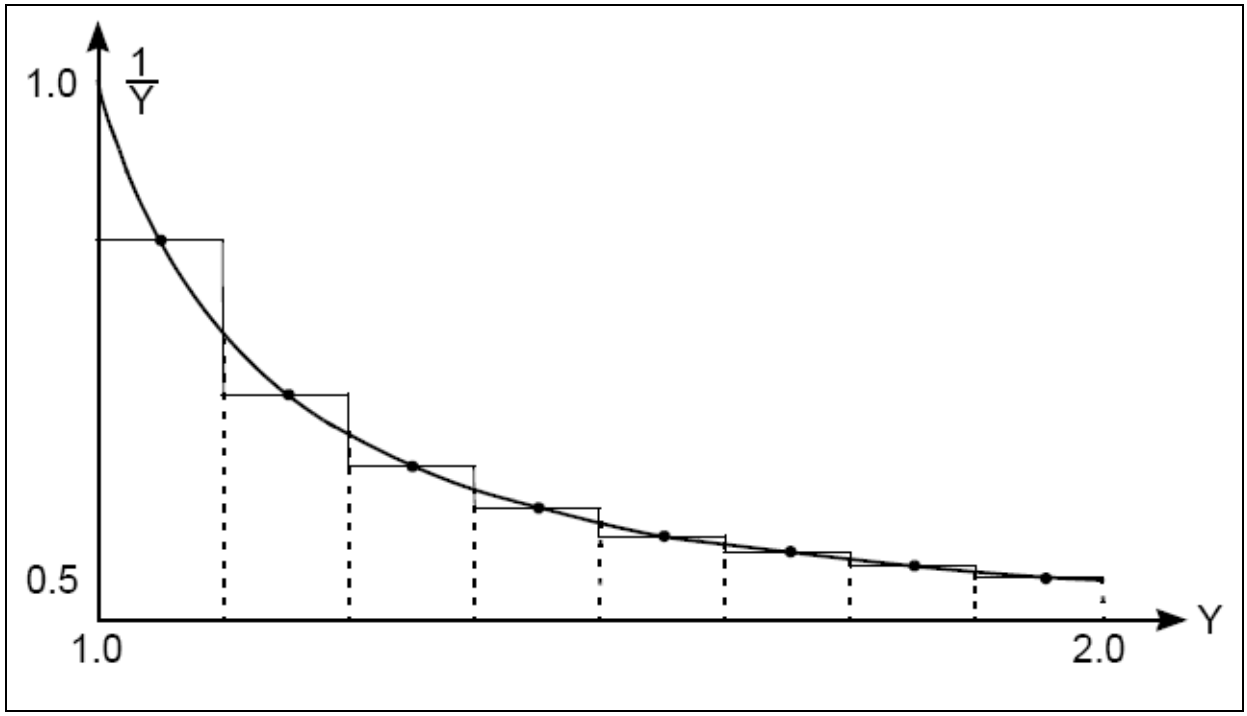
En ser fra ligning 23 og 24 at den relative feilen er lik ved både den øvre og nedre grensen for et delintervall. Ut fra nevneren i ligning 23 og 24 vises det også at den relative feilen blir minst ved stor *J*-verdi som gir en høy intervall indeks. Dersom divisoren tilhører det første delintervallet hvor $j = 0$, vil den relative feilen bli størst og få følgende relative feil:

$$e(Y)_{j=0} = \left| \frac{2^k}{2^{2k+1} + 2^k} \right| = \left| \frac{2^k}{2^k\left(2^{k+1} + 1\right)} \right| = \left| \frac{1}{2^{k+1} + 1} \right| \qquad (25)$$

Fra ligning (25) går det frem at den initielle presisjonen *p* blir en funksjon av antall delintervaller *k* + 1 implisitt bit, samt sammenhengen:

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

Ved å sette inn $2^{-(k+1)}$ i ligning 20, kommer det frem at den initielle presisjonen blir lik *k*, som kommer av hvor mange 2^k delintervaller som divisorens område er oppdelt i.



Figur 4: Stegvis tilnærming til resiprokalets funksjon.

4.3 Lineær interpolasjon

For å oppnå en mer nøyaktig tilnærming til funksjonen som gir resiprokalet ut fra divisoren, kan en polynomisk approksimasjon av *n*te grad benyttes (se ligning 26 hvor *Y* er divisoren og *c* er konstanter).

$$z(Y) = c_0 + c_1 Y + c_2 Y^2 + \ldots + c_n Y^n \tag{26}$$

For å tilfredsstille krav til ytelse og areal, forenkles approksimasjonen til en lineær funksjon på følgende form:

$$z_0(Y) = c_0 + c_1 Y \tag{27}$$

I ligning 27 er *Y* divisoren og c_0, c_1 er konstanter. Denne forenklingen gir færre korrekte bit enn en høyeregrads tilnærming. Ved å sette inn verdiene for intervallets endepunkter i ligning 237får vi følgende ligninger (se ligning 28 og 29):

$$\frac{1}{2} = c_0 + 2c_1 \tag{28}$$

$$1 = c_0 + c_1 \tag{29}$$

Verdier for c_0 og c_1 finnes da ved å løse ligning 28 og 29.

$$c_0 = \frac{3}{2}$$
$$c_1 = -\frac{1}{2}$$

Dermed kan formelen for initial verdien utrykkes slik:

$$z_0(Y) = c_0 + c_1 Y$$
$$z_0(Y) = \frac{3}{2} + -\frac{1}{2} Y$$
$$z_0(Y) = \frac{(3 - Y)}{2} \tag{30}$$

Ligning 30 kan realiseres ved hjelp av en subtraksjon og et høyreskift. Den relative feilen til $z_0(Y)$ blir da:

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

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

Bestemmer hvilken verdi for divisoren som gir størst relativ feilrate:

$$e'(Y) = Y - \frac{3}{2}$$

$$Y - \frac{3}{2} = 0$$

$$\Rightarrow \quad Y = \frac{3}{2}$$

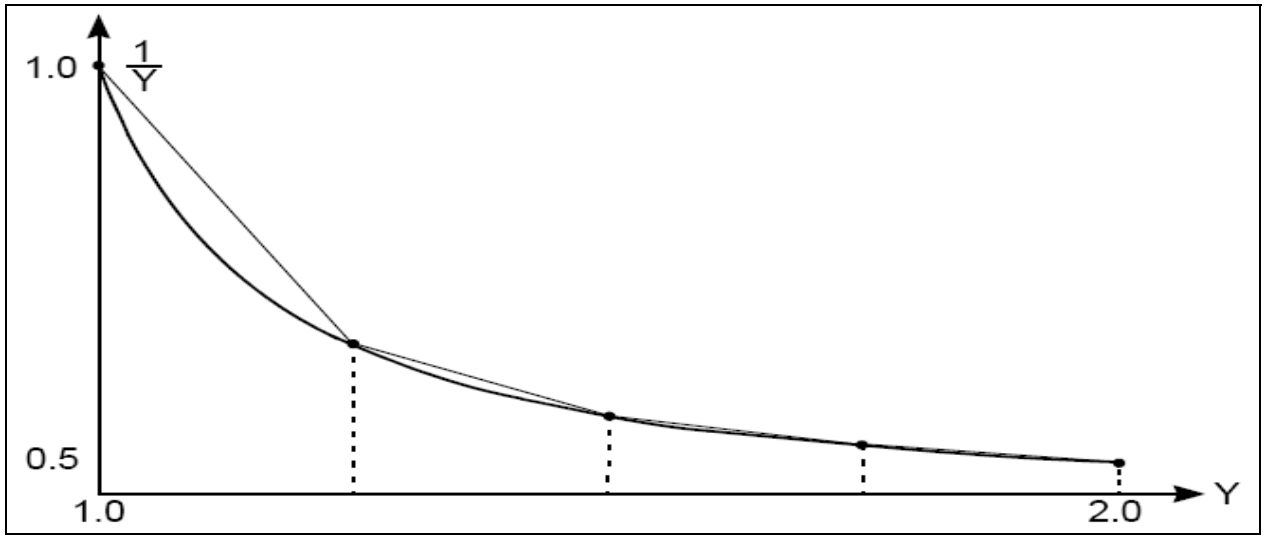
Finner maksimal feilrate for denne lineære approksimasjonsmetoden:

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

$$e\left(\frac{3}{2}\right) = -\frac{1}{8}$$

Med en absolutt maksimal feilrate på $\frac{1}{8}$ oppnås en initiell nøyaktighet på 3 bit.

Lineær interpolerasjon kan utnyttes ved stegvis tilnærming hvor divisorens tallområde deles opp i delintervaller og lineær interpolerasjon benyttes på hvert delintervall (se illustrasjon 5 for grafisk fremstilling).



Figur 5: lineær interpolasjon hvor tallområdet er delt inn i intervaler.

Divisorens tallområdet deles inn i 2^k like store delintervaller, på samme måte som med stegvis tilnærming:

$$\left[\frac{\left(2^k + j\right)}{2^k}, \frac{\left(2^k + j + 1\right)}{2^k}\right), \qquad j \in \left\{0,1,...,2^k - 1\right\}$$

Hvert delintervall har en øvre og en nedre grenseverdi $\left[Y_l, Y_h\right)$. Disse benyttes til å beregne konstantene c_0 og c_1 i formelen for lineær approksimasjon fra ligning 27:

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

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

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

$$\Rightarrow \quad \frac{\left(Y_h - Y_l\right)}{Y_l Y_h} = c_1\left(Y_l - Y_h\right)$$

$$\Rightarrow \quad c_1 = -\frac{1}{Y_l Y_h} \quad \Rightarrow \quad c_0 = \frac{1}{Y_l} + \frac{1}{Y_h}$$

Formelen for den initielle approksimasjonen blir da:

$$z_0\left(Y\right) = \left(\frac{1}{Y_l} + \frac{1}{Y_h}\right) - \frac{Y}{Y_l Y_h} \tag{31}$$

Med ligning 31 for den initielle approksimasjonen innsatt i ligning 18 oppnås følgende relative feilrate:

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

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

$$e\left(Y\right) = 1 - \left(\frac{Y_l Y + Y_h Y - Y^2}{Y_l Y_h}\right) \tag{32}$$

Fra figur 5 og ligning 23 og 24 kommer det frem at den relative feilen til de lineære approksimasjonene blir størst ved de første delintervallene. Aller størst blir den relative feilen på midtpunktet til det første delintervallet hvor *j=0*. Fra formel 21 blir det:

$$Y = \frac{\left(2^{k+1} + 1\right)}{2^{k+1}}$$

Delintervallets grenseverdier:

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

Innsatt i ligning 32 blir dermed den maksimale relative feilen følgende:

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

$$\left|e(Y)_{\max}\right| = 1 - \left(\frac{\frac{2^{2k+2} + 2^{k+2} + 1}{2^{2k+2}}}{\frac{2^k+1}{2^k}} \right)$$

$$\left|e(Y)_{\max}\right| = \frac{2^{3k+2} + 2^{2k+2}}{2^{3k+2} + 2^{2k+2}} - \frac{2^{3k+2} + 2^{2k+2} + 2^k}{2^{3k+2} + 2^{2k+2}}$$

$$\left|e(Y)_{\max}\right| = \frac{2^k}{2^{3k+2} + 2^{2k+2}}$$

$$\left|e(Y)_{\max}\right| = \frac{1}{2^{2k+2} + 2^{k+2}} < 2^{-(2k+2)} \qquad (33)$$

4.4 Bipartite tabeller

Bipartite (to delte) oppslagstabeller er en effektiv måte for å finne en initialverdi til resiprokalet for videre iterering med Newton-Raphson -metoden. Metoden går ut på å benytte en normalisert verdi av divisoren for å adressere en tabell som inneholder resiprokalverdier. Dermed oppnås et relativt nøyaktig initiellt resiprokal for iterering noe som resulterer i et minimum av tidkrevende regneoperasjoner, i bytte mot økt areal.

Divisoren, Y normaliseres til området $[1,2)$ for så å trunkeres eller begrenses til i bit til høyre for kommaplassen på følgende form:

$$Y = 1, b_1 b_2 b_3 b_4 ... b_i$$

Disse i-bitene benyttes så til å adressere tabellen som gir j-bit ut:

$$\frac{1}{Y} = 0,1 b_1' b_2' b_3' b_4' ... b_j'$$

Dermed blir dette en i-bit inn, j-bit ut resiprokal tabell med en størrelse på $j2^i$ bit. I [8] beskrives algoritmer for bipartite resiprokale tabeller som tilfredsstiller krav om en

nøyaktighet på 1 ULP (Unit in Last Place, posisjonsverdien til det minst signifikante bitet i et dataord). Disse algoritmene for bipartite tabeller har $j+g$ bit inn og j bit ut, hvor $g \geq 1$ og $j \geq 3$. De j+g-inn bitene blir inndelt i 3 partisjoner, høy (Y_h), middels (Y_m) og lav (Y_l), etter deres posisjonsverdi i den normaliserte og trunkerte inn-vektoren på følgende vis:

$$Y = 1, b_1 b_2 b_3 ... b_{j+g-1} b_{j+g} = 1, b_1 b_2 b_3 ... b_h b_{h-1} b_{h-2} .. b_{h-m} b_{h-m-1} b_{h-m-2} .. b_{h-m-l}$$

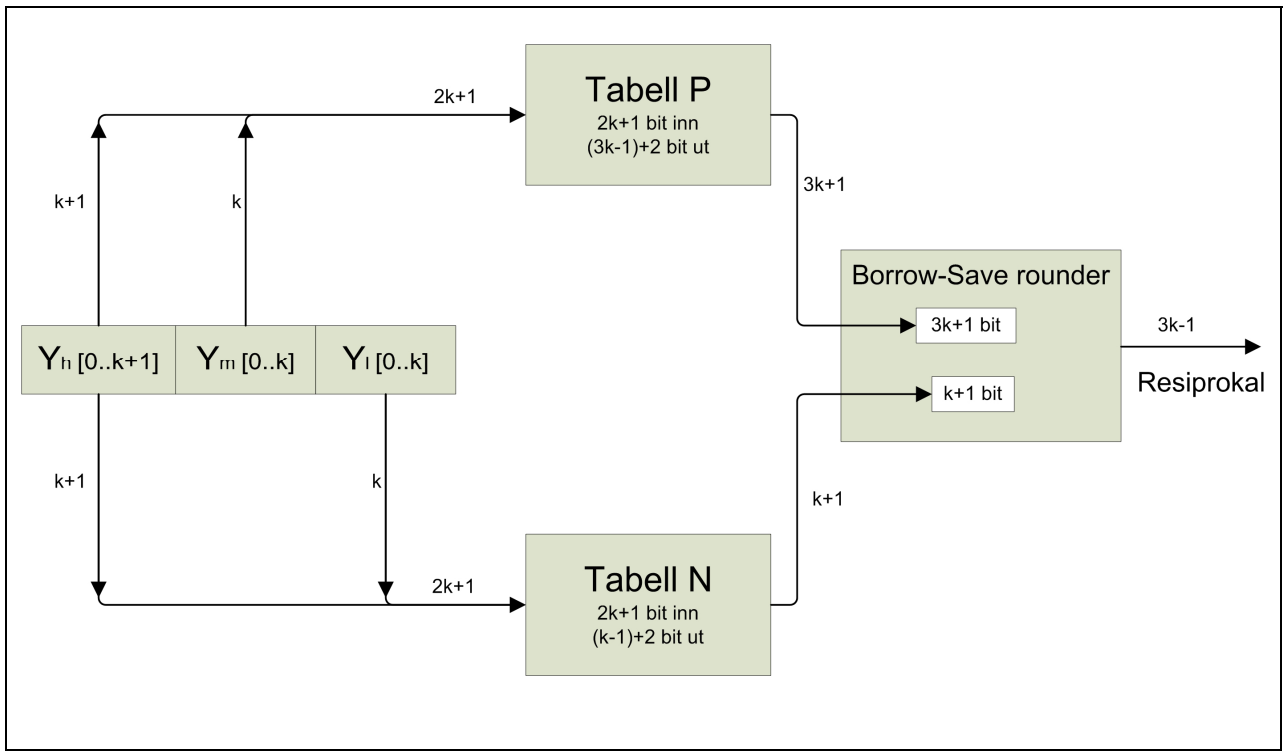
I [8] finnes et illustrerende eksempel som forklarer prinsippet med bipartite tabeller. I eksemplet som benyttes, sendes $j+g = 3k+1$ bit inn og et resiprokal på $3k-1$ bit oppnås ut fra tabellen, der $g = 2$. Oppslagsprosessen er delt opp i 4 steg (se figur 6 for blokkskjema for oppslag i den bipartite tabellen).

1. $j+2 = 3k+1$ bit sendes inn. Disse er partisjonert i 3 grupper, Y_h, Y_m og Y_l bestående av henholdsvis $k+1$, k og k bit.

2. De $2k+1$ bitene fra Y_h og Y_m partisjonene $\left(\sim \frac{2}{3} j\right)$ adresserer en positiv tabell P som gir ut $j = 3k+1$ bit.

3. De $2k+1$ bitene fra Y_h og Y_l $\left(\sim \frac{2}{3} j\right)$ adresserer en negativ tabell N som gir ut $k+1$ $\left(\sim \frac{1}{3} j\right)$ bit.

4. En Borrow-Save adderer som i tillegg runder av resultatet. Resultatet blir et resiprokal på $j = 3k-1$ bit som er korrekt til 1 ULP.



Figur 6: Blokkdiagram som viser strukturen til bipartite tabeller.

Intervallet til divisoren, $1 \le Y < 2$ kan deles opp "blokk", "segment" og "verdi" i samsvar med de 3 partisjonene av Y, Y_h, Y_m og Y_l som er inndelt etter posisjonsverdien de har i det $j+g$ bit lange ordet Y.

Uttrykkene for Y_h, Y_m og Y_l kan skrives slik:

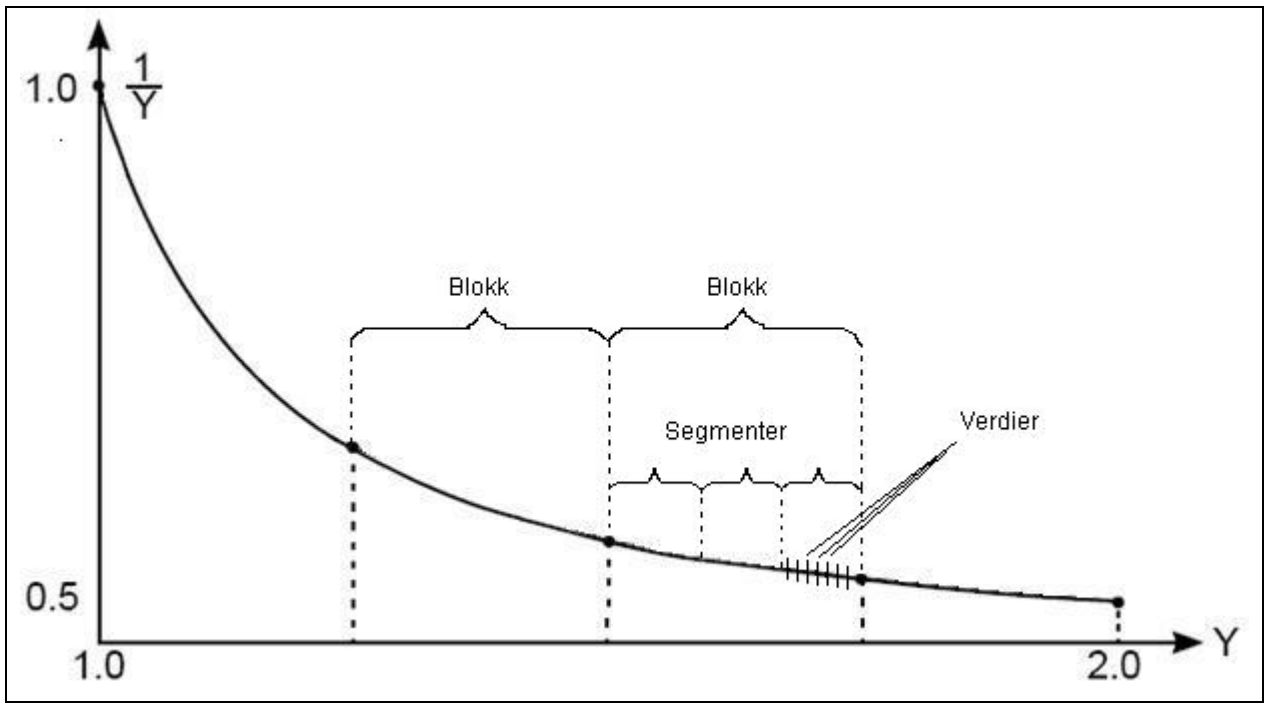
$$Y_h = 1, b_1 b_2 b_3 \ldots b_h \tag{34}$$

$$Y_m = 0, b_{h-1} b_{h-2} b_{h-3} \ldots b_m \cdot 2^h \tag{35}$$

$$Y_l = 0, b_{h-m-1} b_{h-m-2} b_{h-m-3} \ldots b_{h-m-l} \cdot 2^{h+m} \tag{36}$$

Det gjør det hensiktsmessig med følgende inndeling av intervallet $1 \le Y < 2$ basert på blokker, segmenter og verdier (se figur 7 for grafisk fremstilling av inndelingen):

Blokk: Intervallet $1 \le Y < 2$ deles inn i 2^h blokker.
Segment: Hver blokk deles opp i 2^m segmenter.
Verdi: Hvert segment deles opp i 2^l verdier.



Figur 7: Grafisk fremstilling av divisorens intervall inndelt i blokker, segmenter og verdier.

Med en inndeling av divisorens intervall på 2^h antall blokker vil hver blokk få en bredde på $\frac{1}{2^h}$. Dette gir at intervallet som en blokk dekker kan skrives på følgende måte:

$$\left[1+\frac{1}{2^h}a,1+\frac{1}{2^h}(a+1)\right)= \left[\frac{2^h+a}{2^h},\frac{2^h+a+1}{2^h}\right) \tag{37}$$

Variabelen a i ligning 37 representerer blokk nummeret. $a \in [0,2^h-1]$.

Hver blokk består av 2^m segmenter med en bredde på $\frac{1}{2^{h+m}}$, som dekker følgende intervall:

$$\left[\frac{2^h+a}{2^h}+\frac{b}{2^{h+m}},\frac{2^h+a}{2^h}+\frac{b+1}{2^{h+m}}\right)= \left[\frac{2^{h+m}+a}{2^h}+\frac{b}{2^{h+m}},\frac{2^h+a}{2^h}+\frac{b+1}{2^{h+m}}\right) \tag{38}$$

Variabelen b i ligning 38 representerer segmentnummeret. $b \in [0,2^m-1]$. Variabel a er den samme som i ligning 37.

Hvert segment består av 2^l verdier med en bredde på $\frac{1}{2^{h+m+l}}$, som dekker følgende intervall:

$$\left[\frac{2^{h+m}+2^{m}a+b}{2^{h+m}}+\frac{c}{2^{h+m+l}},\frac{2^{h+m}+2^{m}a+b}{2^{h+m}}+\frac{c+1}{2^{h+m+l}}\right)=$$
$$\left[\frac{2^{h+m+l}+2^{m+l}a+2^{l}b+c}{2^{h+m+l}},\frac{2^{h+m+l}+2^{m+l}a+2^{l}b+c+1}{2^{h+m+l}}\right) \quad (39)$$

Variabelen c i ligning 39 representerer verdinummeret. $c \in [0,2^{l}-1]$. Variablene a og b er henholdsvis de samme som i ligning 37 og 38.

Antall riktige ønskelige bit ut fra den bipartite resiprokal tabellen er som tidligere nevnt, $j + g$ bit eller $j = h + m + l - g$ bit. Utverdien beregnes ut i fra en Borrow-Save subtraksjon av de to verdiene fra tabell P og N. Den positive tabellen P har $h + m$ bit inn, og har $h + m + l$ bit lagret for utlesning til Borrow-Save subtraksjonen sammen med verdien fra den negative tabellen N. Tabell N har $h + l$ bit inn, og har $l-g+3$ bit lagret for utlesning til Borrow-Save subtraksjonen.

Med 2^{h+m} inn verdier til P tabellen og 2^{h+l} inn verdier til N tabellen vil den totale størrelsen for denne bipartite tabellmetoden bli følgende: $2^{h+m}(h+m+l)+2^{h+l}(l-g+3)$.

En *i*-bit inn, *j*-bit ut bipartit resiprokal-tabell sies å være optimal dersom den gir samme resultat som en optimal full-oppløsning $2^{i}j$ -bit resiprokal tabell. Kompresjonsfaktoren på en bipartit resiprokal-tabell i forhold til en full-oppløsning resiprokal-tabell blir $2^{i+1}j$ delt på størrelsen til en *i*-bit inn *j*-bit ut bipartit resiprokal-tabell. Grunnen til at det er $2^{i+1}j$ som benyttes er at dette er størrelsen på den minste full-oppløsning resiprokal-tabellen som sikrer en nøyaktighet på 1 ULP i følge [8].

I [8] ses det på en algoritme for en bipartit resiprokal-tabell som gir ut flere enn 10 bit. Med den algoritmen genereres en *(j+2)*-bit inn, *j*-bit ut bipartit resiprokal-tabell, som holder kravet om en unøyaktighet på mindre enn 1 ULP.

4.4.1 Algoritme for bipartite resiprokal-tabeller

I [8] beskrives oppbygningen av en algoritme for å konstruere en *j*+2 bit inn, *j* bit ut bipartit tabell. Denne algoritmen baserer seg på innbitene (*j*+*g*) heltallsdividert med 3, for å kunne innføre parameteren $k = \left(\frac{j+g}{3} \right)$, som benyttes til å dele inngangs bitene inn i 3 kategorier, $j \in [3k-2, 3k-1, 3k]$. Grunnen til denne inndelingen kommer av hensyn til en nøyaktighet på 1 ULP. Generelt sett kan inndelingen beskrives ved hjelp av variabelen *u*. $j \in [k+1, k+u, k]$. Variabelen *u* angis på følgende vis:

$u = -1$ hvis $(j+g) \bmod 3 = 0$
$u = 0$ hvis $(j+g) \bmod 3 = 1$
$u = 1$ hvis $(j+g) \bmod 3 = 2$

Med denne fordelingen, genererer algoritmen en optimal bipartit tabell hvis størrelse vokser med en faktor på 4 for hvert 3. ekstra bit den utvides med. Til sammenligning øker konvensjonelle optimale tabeller med full-oppløsning med en faktor på 8 for tilsvarende utvidelse. Størrelsen samt dimensjonering på den optimale bipartite tabellen for de ulike inndelingene nevnt over, vises i tabell 1.

Ut bit *j*	Inn bit Partisjon av (*j+2)*	Positiv tabelldel Dimensjonering	Negativ tabelldel Dimensjonering	Total tabellstørrelse i bit
3*k-2*	*k*+1, *k*-1, *k*	2*k* inn, 3*k* ut	2*k*+1 inn, *k*+1 ut	$2^{2k}(5k+2)$
3*k*-1	*k*+1, *k*, *k*	2*k*+1 inn, 3*k*+1 ut	2*k*+1 inn, *k*+1 ut	$2^{2k}(8k+4)$
3*k*	*k*+1, *k*+1, *k*	2*k*+2 inn, 3*k*+2 ut	2*k*+1 inn, *k*+1 ut	$2^{2k}(14k+10)$

Tabell 1: Tabellen viser total tabellstørrelse for optimal bipartit tabell generert med algoritme fra [8]. Verdiene i tabellen er også hentet fra [8].

For å få et perspektiv på hvor store datamengder som skal lagres, kan dette beregnes ut i fra formlene i kolonne 5 i tabell 1 for total tabellstørrelse. Beregner total tabellstørrelse for to utvalgte operandstørrelser, henholdsvis 16 og 32 bit.

j = 16:
For å kunne heltallsdividere på 3, må de 16 ut bitene følge formelen i rad 2, kolonne 1 i tabell 1.

$3k - 2 = 16$
$3k = 18$
$k = 6$

Benytter $k = 6$ i formel for total tabellstørrelse i rad 2, kolonne 5 i tabell 1.

$tot\_størrelse = 2^{2k}(5k+2)$
$tot\_størrelse = 2^{2 \cdot 6}(5 \cdot 6 + 2)$
$tot\_størrelse = 131072bit$
$tot\_størrelse = 16{,}4kB$

j = 32:

For å kunne heltallsdividere på 3, må de 32 ut bitene følge formelen i rad 2, kolonne 1 i tabell 1.

$$3k - 1 = 32$$
$$3k = 33$$
$$k = 11$$

Benytter $k = 11$ i formel for total tabellstørrelse i rad 3, kolonne 5 i tabell 1.

$$tot\_størrelse = 2^{2k}(8k + 4)$$
$$tot\_størrelse = 2^{2 \cdot 11}(8 \cdot 11 + 4)$$
$$tot\_størrelse = 385875968 bit$$
$$tot\_størrelse = 48{,}2 MB$$

Til sammenligning med ordinære optimale full-oppløsning tabellers størrelse gir bipartite tabeller en betraktelig mindre størrelse. Dette vises i tabell 2.

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

Tabell 2: Tabellen viser oversikt over total tabellstørrelse for optimal ROM tabell og bipartit tabell for ulike inn bit j. Tabellen er hentet fra [8].

For beregning av verdiene som skal lagres i den bipartite tabellen benyttes blant annet midtpunktet mellom to verdier innen et segment. Midtpunktsfunksjonen er den samme som ligning 21, som det er bevist gir minst relative feilrate. Her vil resiprokalet ha en verdi gitt av ligning 22. Her angir (k+1) hvilket antallet delintervall som resiprokalets området er inndelt i, og j det respektive delintervallet hvor verdien befinner seg.

$$\frac{1}{Y} = \frac{2^{k+1}}{2^{k+1} + 2j + 1} \tag{22}$$

Denne formelen kan skrives om slik at formelen blir basert på inn bitene i metoden for bipartite tabeller..

$$\frac{1}{Y} = \frac{2^{3k+u+2}}{2^{3k+u+2} + 2\left(y_h + y_m 2^{k+1} + y_l 2^{2k+u+1}\right) + 1} \tag{40}$$

$$\frac{1}{Y} = \frac{2^{h+m+l+1}}{2^{h+m+l+1} + 2\left(y_h + y_m 2^h + y_l 2^{h+m}\right)+1} \tag{41}$$

Denne formelen skrives om til den som benyttes i algoritmen for generering av bipartite tabeller i [8].

$$\frac{1}{Y} = \frac{2^{3k+u+1}}{2^{3k+u+1} + \left(y_h + y_m 2^{k+1} + y_l 2^{2k+u+1}\right)+\frac{1}{2}} \tag{42}$$

Ligning 42 blir heretter beskrevet som midtpunktsfunksjonen midRes $\left(y_h, y_m, y_l\right)$ slik som i algoritme 2 i [8].

Algoritme 2 (konstruksjon av bipartit resiprokal-tabell)

Stimuli: heltall *k* og *u*.

Respons: 2*k*+1+*u* bit inn, 3*k*+1+*u* bit ut tabell P og 2*k*+1 bit inn, *k*+1 bit ut tabell N

Steg 1 (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}-1, 0) - midRes(y_h, 2^{k+u}-1, 2^k-1)
- L3: averagespread(y_h) = $\frac{\text{firstspread}(y_h) + \text{lastspread}(y_h)}{2}$
- for y_m = 0 til 2^{k+u} -1
  - 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) = $\frac{\text{averagespread}(y_h) - \text{spread}(y_h, y_m)}{2}$
  - L6: table_P(y_h, y_m) = midRes(y_h, y_m, 0) + adjust(y_h, y_m)
  - L7: round down table_P(y_h, y_m) til 3k+u+1 bit
- End

End

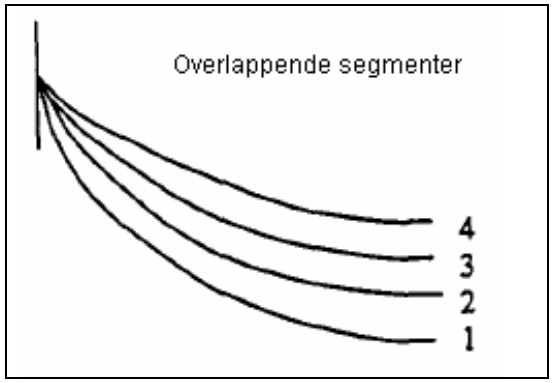
Steg 2 (generering av tabell N):

for y_h = 0 til 2^{k+1}-1
- for y_l = 0 til 2^{k+1}-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}-1, 0) - midRes(y_h, 2^{k+u}-1, y_l)
  - L10: table_N(y_h, y_l) = $\frac{\text{firstdiff}(y_h, y_l) + \text{lastdiff}(y_h, y_l)}{2}$
  - L11: rund av til nærmeste table_N(y_h, y_l) til k+1 bit
- end

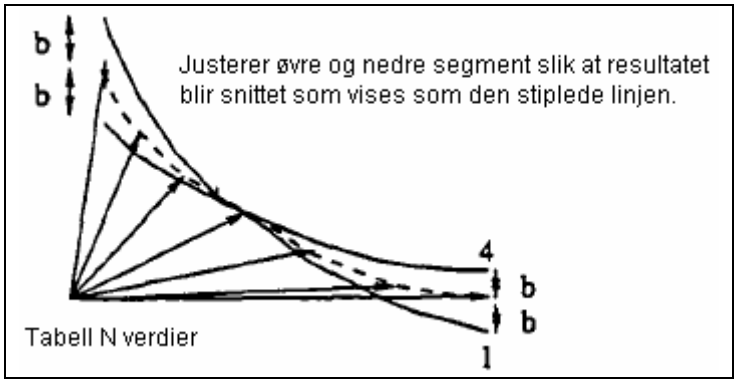
end

Gjennomgang av algoritme 2
Steg 1 (generering av tabell P)
Siden den relative feilen er størst i starten av funksjonen som gir resiprokalet, kan nøyaktighetene økes ved å fordele feilen utover hver blokk. Dette gjøres ved å finne den gjennomsnittlige spredningen mellom første og siste midtverdi i hvert segment i hver blokk (L1 og L2 i algoritme 2), før resultatet midles (L3 i algoritme 2). Deretter finnes spredningen i det segmentet som skal genereres (L4 i algoritme 2), før det midles over denne spredningen og gjennomsnittelig spredning (L5 algoritme 2) for å finne en justeringsverdi. Denne justeringsverdien blir så lagt til midtverdien for å fordele feilen, for så å bli lagret i tabell P som den nye midtverdien (L6 i algoritme 2). Til slutt blir den lagrete midtverdien avrundet ned til (3*k*+u-1) riktige antall bit (L7 i algoritme 2). Det skal også legges til en ener på minst signifikante bit på hver verdi, men denne blir ikke lagret da alle verdiene innholder denne.
Steg 2 (generering av tabell N)
Dette steget finner gjennomsnittelig avstand mellom den første verdien og alle de andre verdiene innen alle segmentene i en gitt blokk (dette blir utført av L8 og L9 i algoritme 2). Dette resultatet midles (L10 i algoritme 2) før det avrundes ned til (*k*+1) riktig antall bit (L11 i algoritme 2) og verdien lagres i tabell N.

En grafisk forklaring vises ved hjelp av figur 8 og 9. I figur 8 vises kurvene til overlappende segmenter i en blokk, som det midles over for å få en kurve som representerer et gjennomsnitt slik som i figur 9. Figur 9 viser snittet som den stiplede linjen hvor N verdiene hentes ut fra. Figur 9 illustrerer også prinsippet med hvordan averagespread blir til for P verdiene.



Figur 9: Overlappende segmenter innen en blokk. Figuren er hentet fra [8].



Figur 9: Segmenter justert med en verdi b, gir snittet vist som stiplet linje som N verdiene kan hentes ut i fra. Figuren er hentet fra [8].

4.4.2 Feilanalyse av algoritme

For å beregne antall riktige bit som tabell P har, kan beviset fra stegvis tilnærming benyttes. Under stegvis tilnærming bevises det at ved bruk av midtpunktsverdien vil antall riktige bit bli en mer enn antall innbit. Dermed oppnås 3*k*+u+2 riktige bit ut med et innsignal på 3*k*+u+1 med algoritme 2. Sammenlignet med antall bit som skal være riktige til slutt (3*k*+u), så vil P verdien oppnå en nøyaktighet på $\frac{1}{4}$ ULP.

Feilen til N verdien er avhengig av både feilen ved midtpunktsverdien og hvor mye feil som innføres når den justeres. Analyserer derfor feilen som innføres ved adjust()-beregningen med hensyn på første og siste segment da algoritme 2 gir størst feil ved enden av blokkene, og aller størst i for blokk 0. Undersøker dermed blokk 0, segment 0 for å finne maks feilrate.

$$adjust(y_h, y_m) = \frac{averagespread(y_h) - spread(y_h, y_y)}{2} \tag{43}$$

$$averagespr\ \ ead\ (y_h) = \frac{firstsprea\ \ d(y_h) + lastspread\ \ \ (y_h)}{2} \tag{44}$$

Ligning 40 innsatt i ligning 39:

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

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

$$adjust(y_h, y_m) = \frac{1}{4}\left(lastspread(y_h) - firstspread(y_h)\right) \tag{45}$$

Setter inn $y_h = 0$ i ligning 39 da det er størst feil i blokk 0:

$$adjust(0,0) = \frac{1}{4}\left(lastspread(0) - firstspread(0)\right)$$

Tar utgangspunkt i L10 i algoritme 2:

$$table\_N = \frac{\text{firstdiff}(y_h, y_l) + \text{lastdiff}(y_h, y_l)}{2} \tag{46}$$

Største feil blir i differansen mellom første og siste verdi i blokk 0. Beregner denne:

$$diff = \frac{\text{firstdiff}(0, 2^k - 1) + \text{lastdiff}(0, 2^k - 1)}{2} \tag{47}$$

Denne differansen er lik:

$$diff = \frac{1}{2}\left(firstspread(0) - lastspread(0)\right) \tag{48}$$

Legger sammen og får:

$$adjust(y_h, y_m) - diff = \frac{1}{4}\left(firstspread(y_h) - lastspread(y_h)\right) \tag{49}$$

Alle verdiene multipliseres med 2^{3k} for å få en ULP verdi som angir feilen som oppstår ved utregningen av N-verdiene.

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

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

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

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

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

$$\frac{1}{8}\frac{2^{3k+1}+2^{2k-1}}{2^{3k+1}+2^{2k-1}} < \frac{1}{8} \tag{50}$$

Midtpunktsverdien gir en maksimal feil på 1/4 ULP uten avrunding og justeringen gir en maksimal feil på 1/8 ULP uten avrunding. Den uavrundete verdien som hentes ut fra den bipartite tabellen får dermed en feil som er lik summen av disse, 3/8 ULP.

Videre er det nødvendig med avrunding og beregning av feilen som da blir introdusert, og det benyttes da et bevis fra [3] for å beregne avrundingsfeilen. Dette beviset er opprinnelig hentet fra [8].

Første steg er å definere en funksjon $RD_g(y)$ som runder av ned til g antall bitt etter ULP og som sørger for at alle feil blir positive og at maksimal feil blir 2^{-g}. Da feilen alltid er positiv kan den halveres ved å tillate at den kan være både positiv og negativ. Dette oppnås ved å legge til en ekstra ener til de g bitene i subtraksjonen mellom P og N verdien, slik at $RD_g(y)+2^{-(g+1)}$ får en maksimal feil på $2^{-(g+1)}$.

Deretter defineres $RN_g(y)$ som runder av ned til nærmeste g antall bit etter ULP. Ved å gjennomføre at alt større enn eller lik $2^{-(g+1)}$ rundes av oppover og ellers ned, oppnås en maksimal feil på $2^{-(g+1)}$.

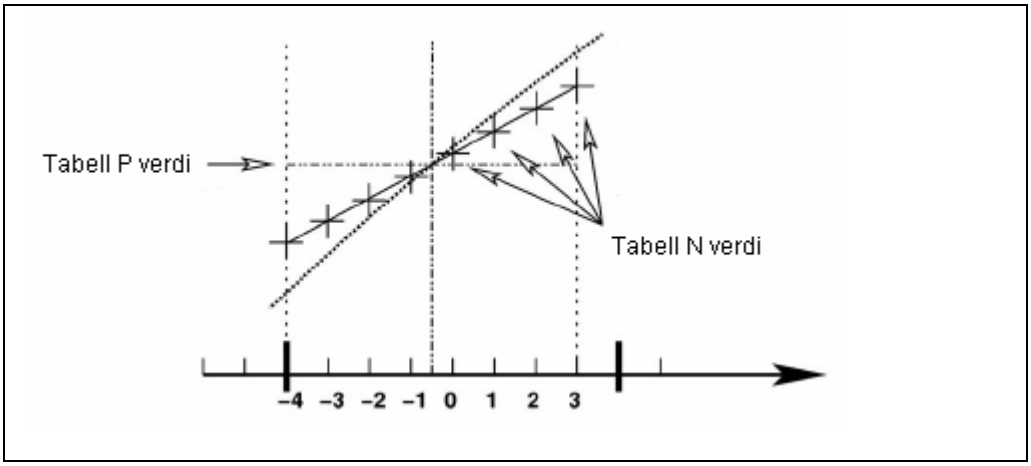
Definerer så en $z = RN_g(e) + RD_g(f) + 2^{-(g+1)}$, slik at differansen mellom z og en avrundet z blir lik:

$$RN_0\left(RN_g(e)+RD_g(f)+2^{-(g+1)}\right)=RN_0\left(RN_g(e)+RD_g(f)\right)$$
$$\left|z-RN_0(z)\right|\le\frac{1}{2}+2^{-(g+1)} \qquad (51)$$

I ligning 51 tilsvarer RN_0 avrunding til nærmeste uten ekstra bit, dermed oppnås en maksimal avrundingsfeil på $\frac{1}{2}+2^{-(g+1)}$ som i dette tilfellet gir en avrundings feil på $\frac{5}{8}$. Dermed kan den totale feilen for den bipartite tabellmetoden beregnes ved å summere feilen som oppstår hos verdiene i tabellen og avrundingsfeilen. $\frac{3}{8}+\frac{5}{8}=1\,\text{ULP}$. Dermed holder metoden med oppslag i en optimal bipartit tabell seg innenfor 1 ULP grensen.

4.4.3 Utnyttelse av symmetri

Det er mulig å oppnå reduserte tabellstørrelser ved å utnytte segmentenes symmetriske egenskaper. I stede for å ta utgangspunkt i resiprokalet i enden av et segmentet, kan resiprokalet legges til midten av segmentet i tabell P for så å enten legge til eller trekke fra for å bestemme resiprokalet. Se figur 10 for grafisk fremstilling av segmentsymmetri.



Figur 10: Grafisk fremstilling av P verdi midt i et segment med N verdier som enten trekkes fra eller legges til som er symmetrisk spredt rundt P verdien. Figuren er basert på figur 3 i [10].

Med denne inndelingen av P-verdiene vil det være tilnærmet like stor avstand til neste verdi over og under. På dette viset vil justeringen være tilnærmet lik for å finne nærmeste resiprokal både over og under. Denne metoden krever imidlertid noe ekstra areal da det er nødvendig med både en addisjonskrets og en subtraksjonskrets. Ved å benytte toer-komplementering av verdiene kan subtraksjonskretsen erstattes med XOR porter slik at forsinkningen minskes.

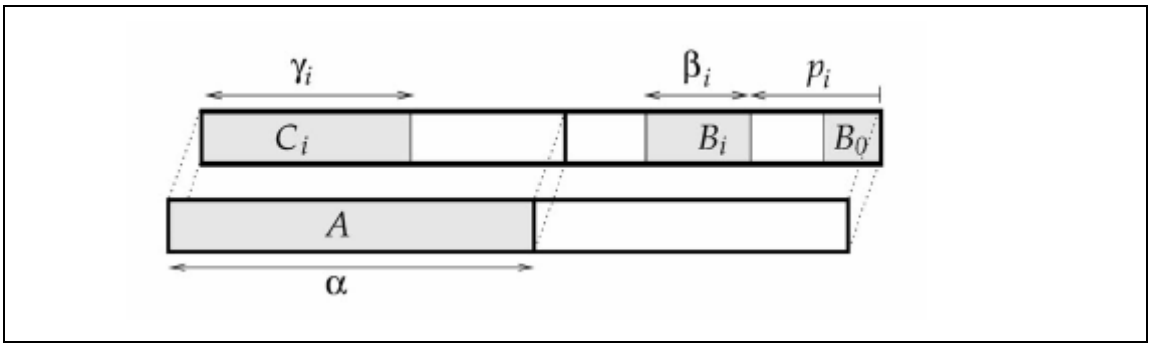
4.5 Multipartitetabeller

Oppslagstabellen for resiprokalets initialverdi kan deles opp i enda flere deler enn den tidligere nevnte bipartite tabelløsningen. Denne metoden kalles multipartite tabeller og baserer seg på samme prinsippet som bipartite tabeller bortsett fra at tabellen er fordelt over flere undertabeller. De mest signifikante bitene i oppslagsverdien benyttes til oppslag i en tabell P som inneholder en tilnærmet verdi for resiprokalet. Denne verdien blir så justert ved hjelp av lagrede verdier i flere underliggende tabeller. Se figur 11 for illustrasjon av oppslagsordets inndeling.

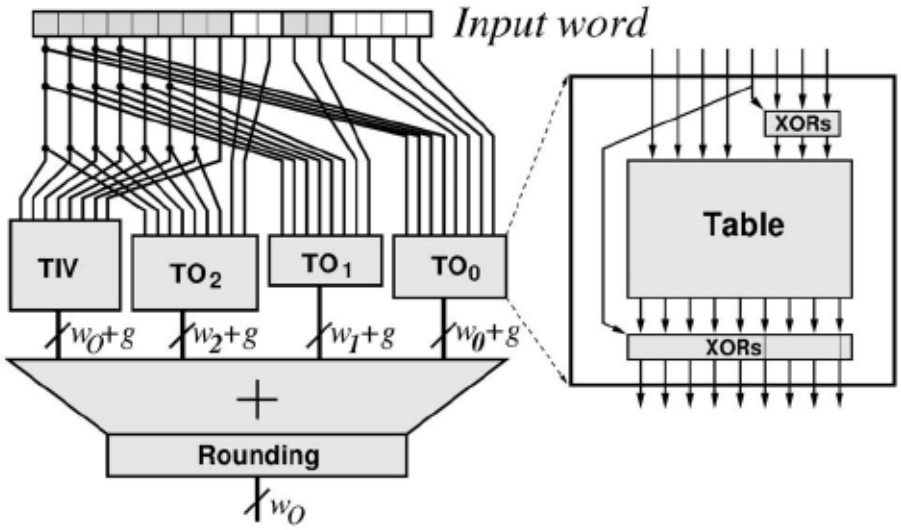
Med en slik flerdelt tabell kan den totale tabellstørrelsen bli redusert i forhold til en bipartit løsning. På grunn av oppdelingen av tabellområdet trengs det flere addisjoner, men reduksjonen i total tabellstørrelse vil kompensere for det økte arealet som skylles adderere. Dermed blir det en avveining mellom addisjonsareal og noe økt prosesseringstid som følge av flere addisjoner, og redusert tabellareal. I tillegg vil det innføres en feilrate ved inndelingen av de ulike deltabellene som vil summeres sammen til en større global feilrate. Det er imidlertid mulig å kompensere for denne inndelings-/kvantiseringsfeilen fra hver deltabell ved å innføre større nøyaktighet på utverdiene. Dette vil riktig nok føre til en økning i tabellstørrelsen. Dermed blir det en avveining mellom tabellstørrelse og nøyaktighet.

Strukturen til en multipartit tabell vises i figur 12. Denne strukturen er basert på å utnytte tabellsymmetri som nevnt i foregående kapitel, og inneholder dermed XOR porter som vist i utsnittet av TO-tabellen i figur 12.

I [11] benyttes en firedelt tabell for oppslag av resiprokalets initalverdi med en nøytaktighet på 1 ULP. Denne løsning oppnådde en total tabellstørrelse på ~27KB for 24 bits operander noe som er betydelig mindre enn hva en bipartit løsning ville ha gitt.



Figur 11: Illustrasjon av oppslagsordets oppdeling som består av A + B bit. De α mest signifikante bitene benyttes til å slå opp i en tabell som innholder en initiell approksimasjon av resiprokalet. Videre benyttes underinndelinger B, B_i til oppslag i andre deltabeller for justeringsverdier som til sammen gir et riktigere resiprokal. Figuren er hentet fra [10].



Figur 12: Blokkskjema viser den generelle strukturen for multipartite tabeller. TIV-tabellen (Table of Initial Values) inneholder det initiale resiprokalet mens TO-tabellene (Table of Offsets) inneholder justeringsverdier. Figuren er hentet fra [10].

5 Implementasjon

For å utføre divisjon ved hjelp av multiplikasjon må dividenden X multipliseres med resiprokalet til divisoren Y. Utfordringen blir da å beregne resiprokalet så nøyaktig som mulig og på kortest mulig tid.

$$Q = X \cdot \frac{1}{Y}$$

Det ble utredet en løsning for 16 bits operander og skissert en løsning for 32 bits operander, hvor begge er pipelinet. Dessverre ble det ikke tid til å utarbeide VHDL kode for løsningene som kan syntetiseres for FGPA. I stedet blir de gjennomgått og beskrevet i følgende kapitler med tilhørende blokkskjema og tilstandsdiagram.

5.1 Løsning for 16 bits operander

Ut fra teori beskrevet i kapitelene 3 og 4 ble noen metoder valgt som gir en divisjonsmodul for 16 bits operander. Løsningen fremskaffer resiprokalet på en effektiv måte og utfører divisjon med 16 bits operander som står i kø for å bli prosessert.

5.1.1 Oppslagstabell

Det ble valgt å benytte en oppslagstabell for å fremskaffe en nøyaktig approksimasjon av resiprokalet for å redusere antall iterasjoner som kreves for å finne et korrekt resiprokal. Det ble dessverre ikke tid til å se nærmere på multipartite oppslagstabeller. Den bipartite tabell-løsningen krever 131kb minne for å lagre initialverdiene til 16 bits operander. Denne informasjonsmengden er mulig å lagre i RAM-blokker til FPGAer i cyclone og virtex-seriene som har opptil 1Mb RAM [12][13]. De største FPGAene fra Actel har opptil 8Mb flashminne[15]. Dermed ble den bipartite oppslagstabellen som er beskrevet tidligere i kapittel 4.4, valgt til å lagre initialverdier av resiprokalet.

5.1.2 Iterasjons-algoritme

For iterasjonsprosessen av et initiellt resiprokal, kan både Goldschmidts-algoritme og Newton-Raphsons-algoritme benyttes. Begge disse algoritmene er tidligere beskrevet i kapittel 3.2. Goldschmidts-algoritme og Newton-Raphsons-algoritme krever begge 2 multiplikasjoner for hver iterasjon. I Goldschmidts algoritme er multiplikasjonene uavhengige slik at de kan utføres parallelt mens med Newton-Raphson-algoritmen må de utføres sekvensielt [1][15]. Feilanalysen til Goldschmidts algoritme er langt mer kompleks enn feilanalysen til Newton-Raphson-algoritmen. Med Goldschmidts-algoritme vil feilen bli akkumulert i motsetning til Newton-Raphson-algoritmen som er selvrettende og gir en kvadratisk konvergens. Dermed ble Newton-Raphson algoritmen valgt til en implementasjons løsning i denne oppgaven, da den gir enklere og sikrere feilanalyse samt tilstrekkelig grad av effektivitet.

Den bipartite oppslagstabellen gir et initiellt resiprokal som er nøyaktig til 1 ULP. Da en Newton-Raphson-iterasjon dobler antall korrekte bit, vil det kun være nødvendig med en iterasjon av det initelle resiprokalet fra tabellen for å få et korrekt resiprokal.

5.1.3 Normalisering og håndtering av spesielle operander

Før divisjonen kan starte må operandenes verdi sjekkes. Det må bestemmes om de er positive eller negative. Det må avgjøres om dividenden X er større enn null, og det må avgjøres om divisoren Y har verdien 1 eller 2 eller er større enn 2. For positive operander trengs det ikke gjøres noe, men negative operander må beskrives med toer-komplementering. Tabell 3 viser spesial tilfellene av operandene og hvordan de skal behandles.

Operander	Beskrivelse	Status	Håndtering
$Y = \|2\|$	Divisor lik 2	Gyldig operand	Returner dividenden skiftet ett bit mot høyre som kvotient, og det bitet som ble skiftet ut som rest.
$Y = \|1\|$	Divisor lik 1	Gyldig operand	Returnerer dividenden som kvotient og rest lik 0
$Y = 0$	Divisor lik 0	Ugyldig operand	Returnerer kvotient og rest lik 0
$X = 0$	Dividend lik 0	Gyldig operand	Returnerer kvotient og rest lik 0

Tabell 3: Tabellen viser spesial tilfeller av operandene som må behandles spesielt.

Som nevnt i teorikapittel 4 må divisoren normaliseres til tallområdet fra og med 1 opp til 2, før divisoren kan benyttes til å adressere den bipartite oppslagstabellen. Dette gjøres ved at divisoren skiftes mot høyre helt til mest signifikante ener kommer på minst signifikante plass. Antall plasser vektoren må skiftes er lik 16 minus plassen til mest signifikante ener i vektoren. Det vil si at kommaet plasseres til høyre for mest signifikante ener. Et eksempel med en vilkårlig 16 bits vektor hvor mest signifikante ener står på plass nr 11, er vist under:

0000111100001111 → 1,111000011110000

Den verdien som beskriver hvor mange ganger divisoren skal skiftes mot høyre beskriver egentlig bare plasseringen av kommaet og må lagres for senere bruk. For å finne plasseringen til mest signifikante ener med en gang, kan logikk benyttes. Dette lar seg gjøre på operander av størrelse 16 bit uten at logikken blir for kompleks og arealet blir for stort. Når divisoren er normalisert benyttes den til å adressere den bipartite tabellen for å skaffe et initiellt resiprokal for iterasjon. Deretter kan iterasjonen utføres ved hjelp av Newton-Raphson-algoritmen. Etter iterasjonen er resiprokalet korrekt og klart for å multipliseres med dividenden X for å finne kvotienten.

5.1.4 Beregning av Kvotienten

Multipliseringen av resiprokalet med dividenden X, kan beskrives som to 16 bits vektorer som skal multipliseres sammen før plasseringen av kommaet skiller kvotientvektoren fra restvektoren. Kommaplassen er den lagrede verdien fra normaliseringsprosessen som beskriver hvor mange ganger divisoren ble skiftet mot høyre. Multipliseringen kan beskrives slik:

$$resultat = M_1 \cdot M_2 \cdot 2^{-e_2}$$

Der M_1 er dividenden, M_2 er resiprokalet og e_2 er antall høyreskift som ble utført ved normaliseringen. Produktet fra multipliseringen blir en 32 bits vektor som deles opp i en 16 bits kvotientvektor og en 16 bits restvektor. Den 16 bit store kvotientvektoren består av bitene til venstre for komma plassen og den 16 bit store restvektoren består av bitene til høyre for kommaet. Kommaplassen bestemmes av verdien til e_2.

5.1.5 Oppsummering av divisjonsprosessen

Divisjonsprosessen kan punktvis oppsummeres slik:

1. Konverter eventuelle negative operander til positive operander. Dette gjøres ved toer-komplementering.

2. Sjekk at operandene er gyldige. Ugyldige og spesielle operander håndteres spesielt slik som beskrevet i tabell 3.

3. Normaliser divisoren Y slik som beskrevet tidligere i kapittelet 5.1.3.

4. Oppslag i tabell med den normaliserte divisoren for å finne en initiell approksimasjon z_0 til divisorens resiprokal, $\frac{1}{Y}$.

5. Newton-Raphson-iterasjon av det initielle resiprokalet fra tabellen.

   $$\frac{1}{Y} = 2 \cdot z_0 - Y \cdot z_0 \cdot z_0$$

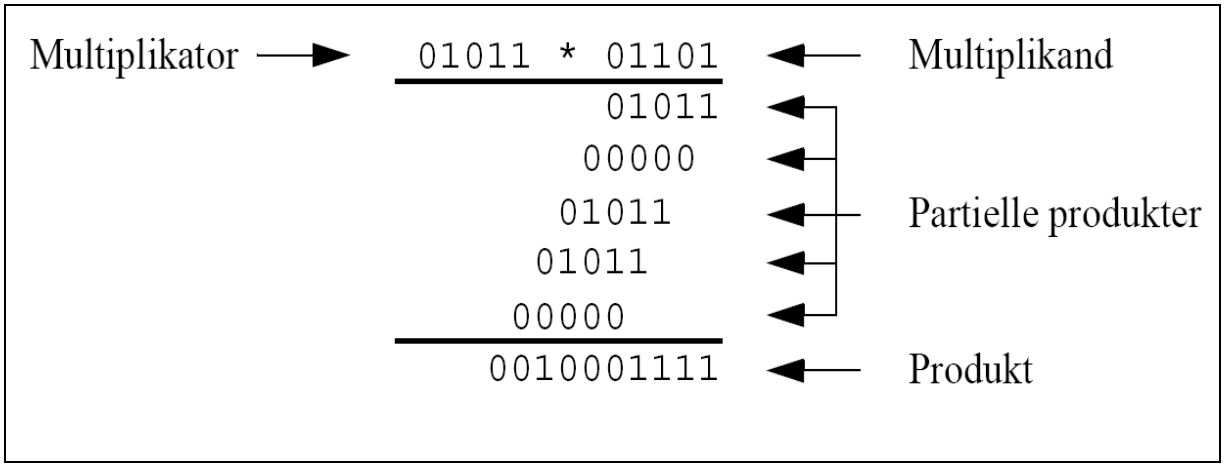
   Det er tilstrekkelig med en iterasjon da det initelle resiprokalet fra tabellen er riktig til 1 ULP og iterasjonen sørger for en dobling av antall riktige siffer.

6. Beregne kvotienten $Q = X \cdot \frac{1}{Y}$ ved å multiplisere dividenden med resiprokalet. Finner kommaplassen på produktet ved hjelp av verdien som anga antall høyreskift fra normaliseringen av divisoren. Deler så opp produktet i en kvotientvektor på 16 bit bestående av bitene til venstre for kommaplassen, og en restvektor på 16 bit bestående av bitene til høyre for kommaplassen.

5.1.6 Multiplikator

Multiplikasjon står sentralt i en multiplikativ divisjons-algoritme, og en hurtig multiplikasjonsprosess er dermed avgjørende for divisjonsmodulens ytelse. Både Newton-Raphson-algoritmen og beregningen av kvotienten krever multiplikasjon. Det kreves totalt 3 multiplikasjoner, 2 sekvensielle multiplikasjoner for en iterasjon og deretter en multiplikasjon for beregning av kvotienten.

En multiplikasjon består av beregning av partielle produkter og summering av disse. Figur 13 viser penn-og-papir metoden for å utføre multiplikasjon på. For å få utført multiplikasjon hurtig er det derfor viktig at operasjonene med å fremskaffe partielle produkter og å summere de går så hurtig som mulig. En hurtig multiplikator blir viktigere jo større ordbredde operandene har da det vil øke antallet partielle produkter som skal legges sammen. For å øke ytelsen kan antall partielle produkter som skal summeres reduseres, summeringen av de utføres hurtigere eller en kombinasjon av begge deler.



Figur 13: Multiplikasjon med binære tall på penn-og-papir metoden.

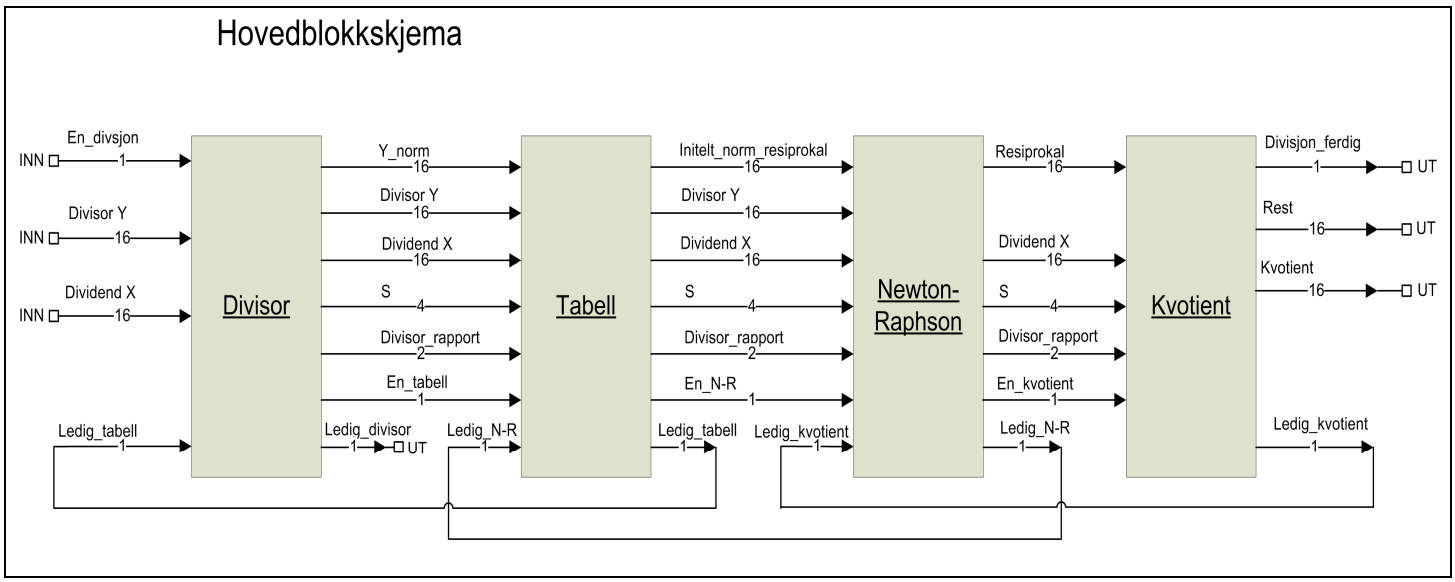
Multiplikatorer kan inndeles i 2 grupper, sekvensielle og parallelle. De sekvensielle genererer de partielle produktene en etter en, før de summeres en etter en. De parallelle genererer de partielle produktene samtidig og utfører deretter summeringen av de parallelt. Denne varianten krever større areal enn den sekvensielle varianten, men har til gjengjeld langt høyere ytelse.

I denne oppgaven er det ikke gått inn i detalj i teorien bak hurtige multiplikatorer men nevner allikevel noen utbredte metoder for parallelle multiplikatorer. For generering av de partielle produktene kan det benyttes en komprimeringsalgoritme eller et reduksjonstre. En kjent komprimeringsalgoritme som kan benyttes er Booth-coding. Et reduksjonstre er bygd opp av flere nivåer med kompressorer for å redusere antallet partielle produkter som må summeres. En utbredt kompressor er (4:2)-kompressoren som komprimerer 4 partielle produktbit til 2 partielle produktbit for hvert nivå i reduksjonstreet [16][17][18][19]. Summeringen av de partielle produktene blir utført parallelt ved hjelp av et summasjonstre eller et summasjonsarray. Ved benyttelse av summasjonstrær (eks. wallace-tree) er det vanlig å kutte ut carry-propageringen for at summeringene skal skje så hurtig som mulig. I stedet generer summasjonstreet to delsummer, hvor den ene indikerer carry-propageringen. Disse to verdiene adderes sammen til slutt.

5.1.7 Arkitektur

Divisjonsprosessen er fordelt på 4 hovedblokker som hver utfører sin del av prosessen. De 4 hovedblokkene Divisor, Tabell, Newton-Raphson og Kvotient er sekvensielt koblet sammen, og vises i figur 14. Hver blokk utfører og beregner verdier ut i fra sine respektive oppgaver av divisjonsprosessen. Verdiene som er resultatet fra oppgavene de har, sendes videre til neste blokk for videre beregninger så snart de er ferdig beregnet. Blokkene kommuniserer seg i mellom og sier i fra når de er ledige å klare til å ta i mot neste verdier. Når de har beregnet ferdig verdier som er klare for å sendes til neste blokk, venter de til neste blokk er ledig før verdiene sendes.

Alle verdier en blokk mottar blir lagret i registre uansett om verdiene inngår i blokkens beregningsoppgaver eller ikke. Når blokken er ferdig med sine respektive oppgaver sendes så de nylig beregnete verdien videre sammen med de verdiene som ble mottatt, men ikke endret, videre til neste blokk. Det vil si at hver divisjonsoppgave som kommer inn, går fra blokk til blokk, hvor hver blokk bidrar til å løse divisjonsoppgaven. For hver blokk kommer divisjonsoppgaven nærmere en kvotient og en rest. Sekvensen på blokkene som verdiene går i gjennom er: Divisorblokken, Tabellblokken, Newton-Raphson-blokken før Kvotientblokken avslutter divisjonsprosessen. Slik oppnås en pipelinet divisjonsmodul som kan motta flere nye divisjonsoppgaver før den første er ferdig beregnet.



Figur 14: Blokkdiagram som viser de 4 hovedblokkene i designet for en 16 bits løsning for multiplikativ divisjon.

Divisorblokken
Divisorblokken utfører punkt 1 til og med punkt 3 i den punktvise oppsummeringen av divisjonsprosessen (kapittel 5.1.5). Divisorblokken har som oppgave å kommunisere ut mot omverden og å ta inn operandene som det skal beregnes en kvotient og en rest av. Videre skal operandene kontrolleres og divisoren skal normaliseres dersom den ikke er 0, 1 eller 2. I tillegg lager den en rapport fra divisorkontrollen som sier noe om begge operandene er gyldige og ikke faller innen for noen spesial områder (se tabell 3). Denne rapporten leveres videre til neste blokk, tabellblokken sammen med dividenden X, divisoren Y, den normaliserte divisoren Y_norm og verdien som beskriver antallet plasser divisoren ble skiftet under normaliseringen. Disse verdiene sendes videre når tabellblokken sier i fra at den er ledig på ”ledig_tabell” porten (se blokkskjema figur 14).

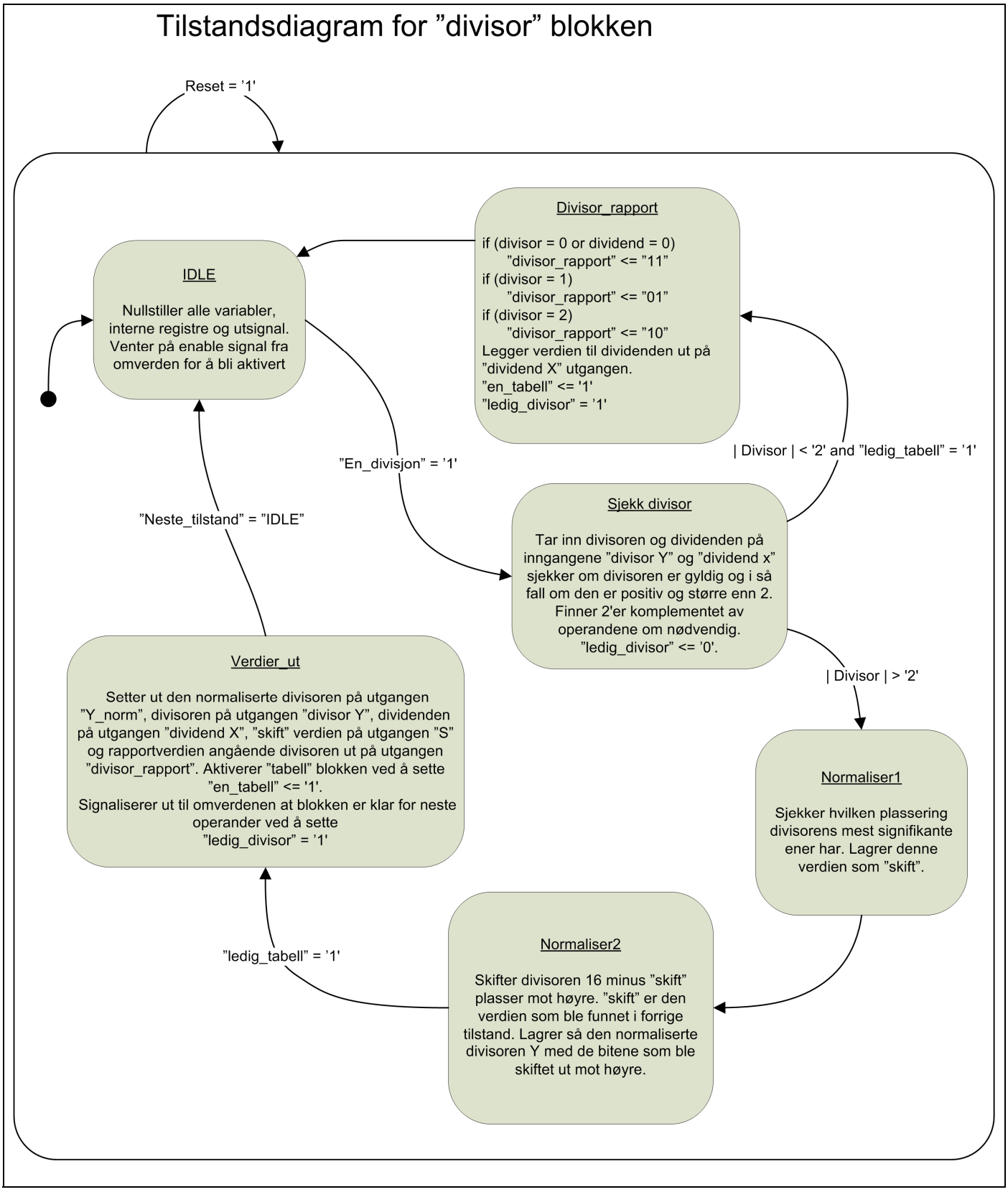
Divisorrapporten består av 2 bit og kan dermed signalisere 4 ulike tilstander. En oversikt over de ulike verdiene til divisorrapporten vises i tabell 4. Divisorrapporten som sendes med til neste blokk, avgjør hva neste blokk utfører. Dersom divisorrapporten er ”00” utfører alle de påfølgende blokkene sine normale oppgaver. Dersom divisorrapporten derimot er en annen, vil de påfølgende blokkene ikke utføre noe annet enn å bare sende dividenden X og divisorrapporten videre til neste blokk. Dette skjer helt til divisorrapporten har nådd kvotientblokken som da foretar en handling i samsvar med divisorrapportens verdi.

Så snart tabellblokken er ledig, sendes verdier over til den. Deretter signaliserer divisorblokken til omverdenen at den er klar for neste divisjonsstykke på utgangen ”ledig_divisor” (se figur 14).

Divisorblokkens oppførsel er beskrevet i tilstandsdiagrammet i figur 15.

Divisorrapport	**Beskrivelse**
”00”	Begge operandene (divisor og dividend) er gyldige, divisoren er større enn 2.
”01”	Begge operandene er gyldige. Divisoren har verdien 1. Kvotienten skal få verdien til dividenden. Resten er lik null.
”10”	Begge operandene er gyldige. Divisoren har verdien 2. kvotienten er lik dividenden skiftet en plass mot høyre. Resten er lik det bitet som ble skiftet ut til høyre.
”11”	En eller begge operandene er ugyldige. Kvotient og rest skal få verdien null.

Tabell 4: Tabellen viser de ulike verdiene divisorrapporten kan ha og betydningen av disse.



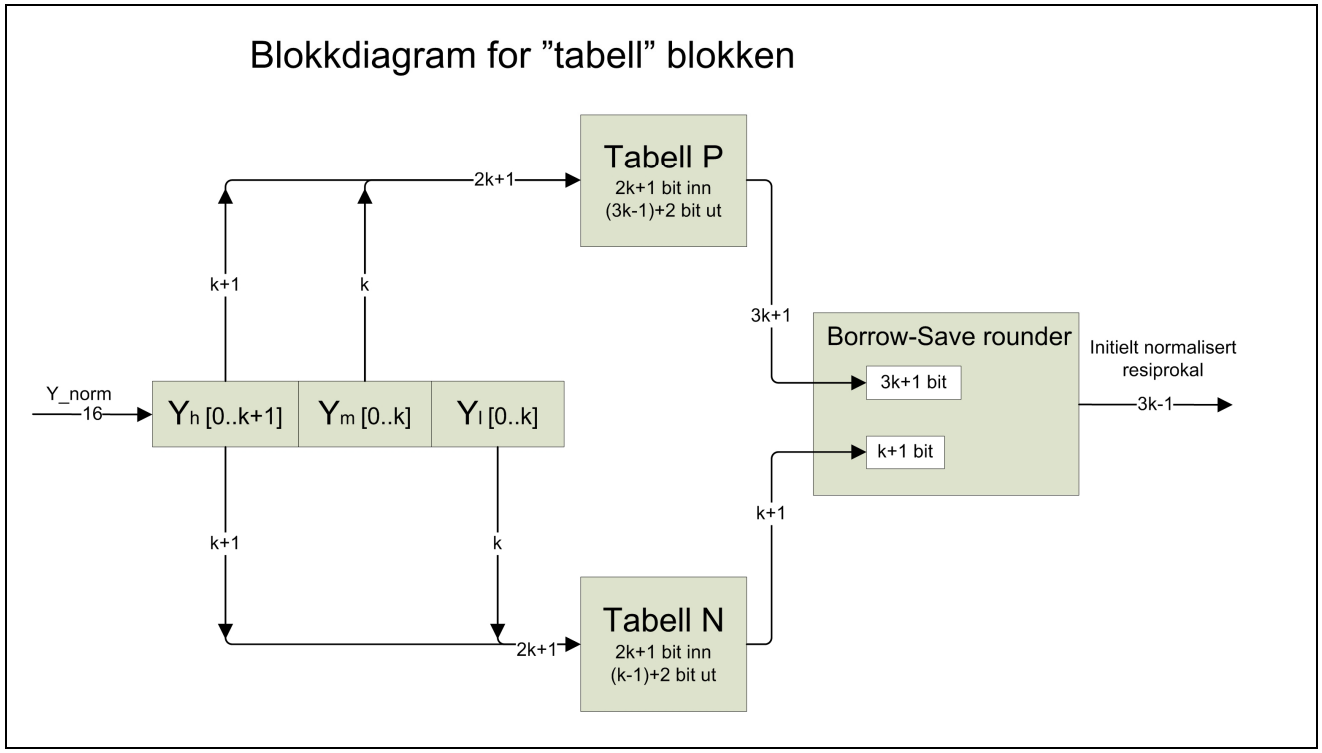
Figur 15: Tilstandsdiagram for ”divisor” blokken på figur 14.

Tabellblokken
Denne blokken benytter den normaliserte divisoren Y-norm som den fikk fra divisorblokken til å adressere en bipartit tabell som er beskrevet i kapittel 4.4. Fra tabellen finnes en initiell approksimasjon av resiprokalet. Tabellblokken utfører punkt 4 i den punktvise oppsummeringen av divisjonsprosessen (se kapittel 5.1.5). Hovedblokkene til denne tabellblokken vises i figur 16. I tillegg kommer registre for å lagre divisor, dividenden, skiftverdien fra normaliseringsprosessen, og divisorrapporten.

Addisjonen mellom P-verdien og N-verdien gjøres med en Borrow-Save adderer slik som i [8]. En Borrow-Save adderer legger sammen tallene ved hjelp av parallelle addisjonsblokker og ligner derfor på en Carry-Save adderer. I [19] beskrives oppbygningen og virkemåten til en Borrow-Save adderer, i figur 17 og 18 vises strukturen til en Borrow-Save adderer.

Dersom divisorrapporten ikke er ”00”, setter divisor blokken bare ut dividenden X og divisor rapporten ut til neste blokk, Neton-Raphson-blokken. Oppførselen til tabellblokken er beskrevet i tilstandsdiagrammet i figur 19.

Tabellen legges i RAM på FPGA fra eksternt flashminne ved oppstart, sammen med konfigurasjonen av FPGAen.

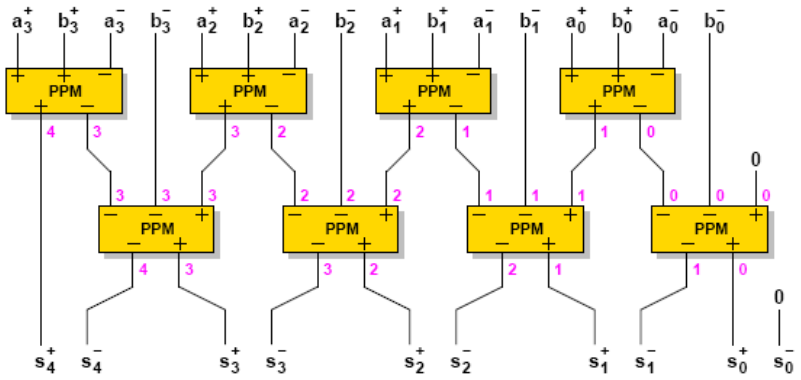


Figur 16: Blokkdiagram som viser hovedblokkene i ”tabell” blokken fra figur 14. Her vises innregisteret og inndelingen av det som benyttes til å adressere tabellene P og N for å få ut to verdier som legges sammen til et resiprokal. Blokkdiagrammet viser ikke logikken for inn og utlesning av de andre vektorene.

Borrow-Save Addition

In borrow-save, the number A is represented in radix 2 using digits $a_i \in \{-1, 0, 1\}$ coded by 2 bits such that $a_i = a_i^+ - a_i^-$ where $a_i^+ \in \{0, 1\}$ and $a_i^- \in \{0, 1\}$

$$A = \sum_{i=0}^{n-1} a_i 2^i = \sum_{i=0}^{n-1} (a_i^+ - a_i^-) 2^i$$



A borrow-save addition is performed with the delay of 2 PPM cells ($T = 0(1)$)

Figur 17: Beskrivelse av struktur å virkemåte til en Borrow-Save adderer. Bildet er hentet direkte fra [19].

PPM Cell

a^+	b^+	d^-	c^+	s^-
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

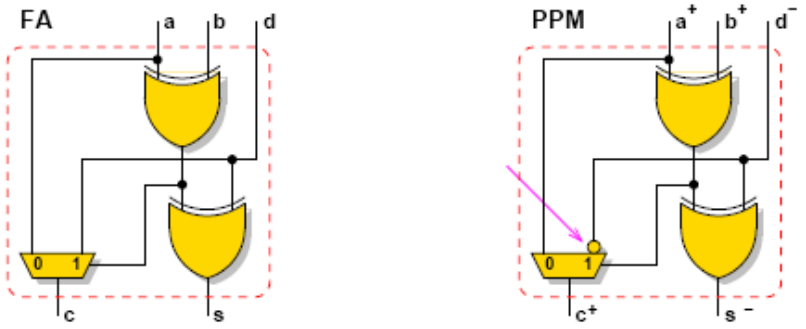
Arithmetic equation:

$$2c^+ - s^- = a^+ + b^+ - d^-$$

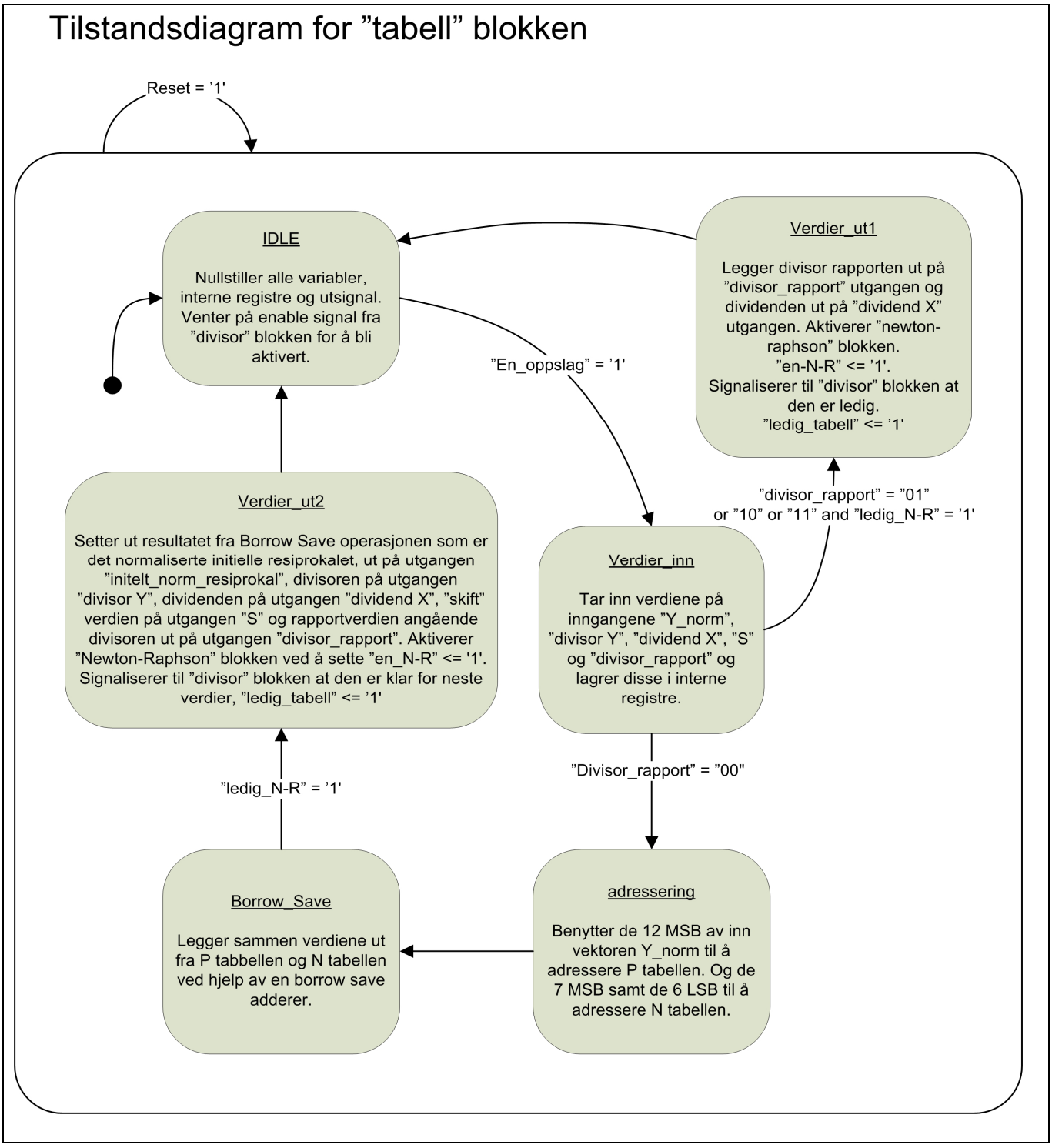
Logic equation:

$$s = a^+ \oplus b^+ \oplus d^-$$

$$c = a^+ b^+ + a^+ \overline{d^-} + b^+ \overline{d^-}$$



Figur 18: Beskrivelse av struktur å virkemåte til PPM blokken i en Borrow-Save adderer. Bildet er hentet direkte fra [19].



Figur 19: Tilstandsdiagram for ”tabell” blokken på figur 14.

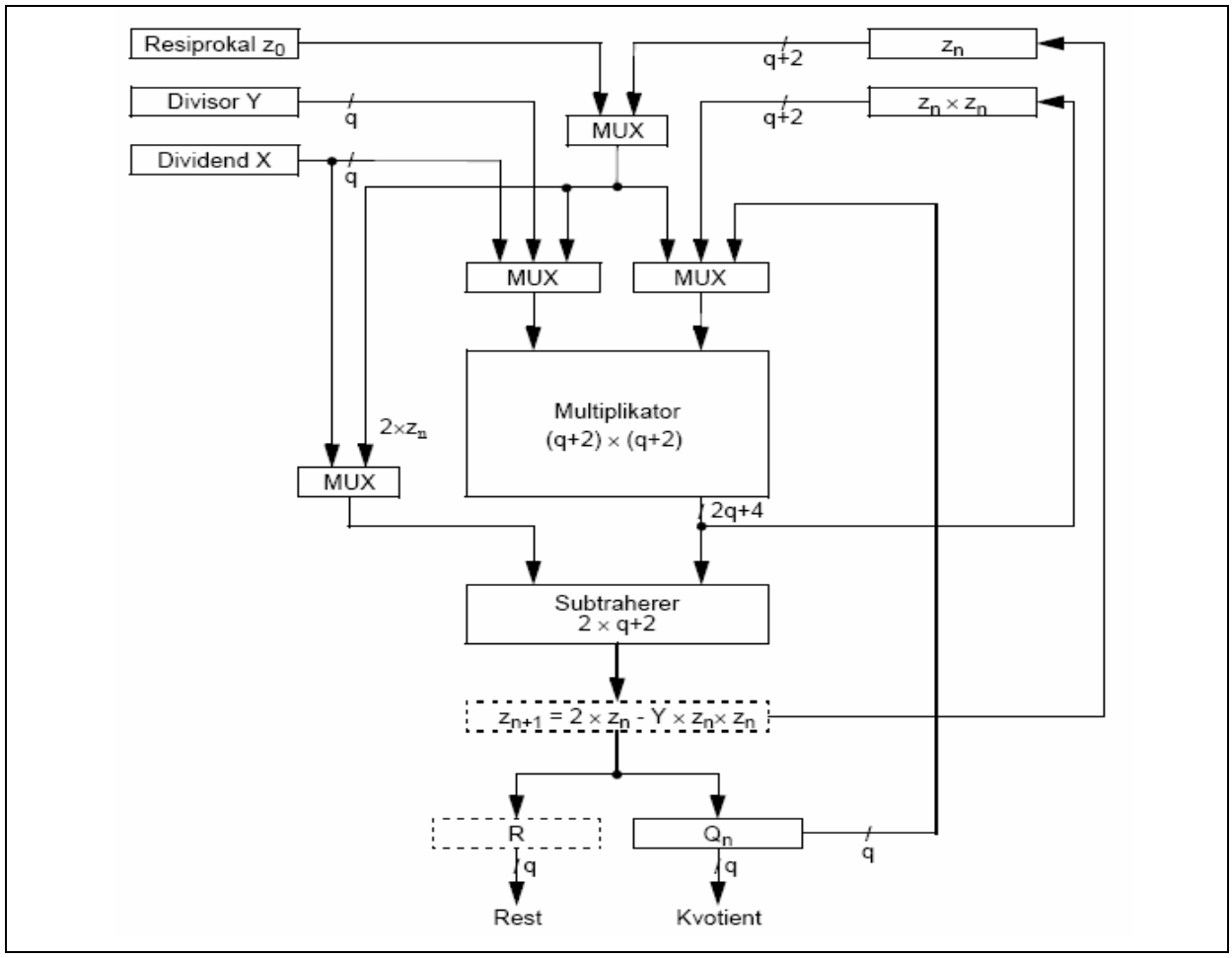
Newton-Raphson-blokken
Denne blokken utfører punkt 5 og 6 i den punktvise oppsummeringen av divisjonsprosessen i kapittel 5.1.5. Blokken tar inn den initielle approksimasjonen av resiprokalet, divisoren Y, dividenden X, skift-verdien fra normaliseringen som ble utført i divisorblokken og divisorrapporten og lagrer disse i interne registre.

En iterasjon utføres ved hjelp av Newton-Raphson-algoritmen som er beskrevet tidligere i kaptiell 3.2.1. Iterasjonen består av 2 multiplikasjoner som utføres sekvensielt slik at samme multiplikator kan benyttes, og en subtraksjon samt et venstreskift. Første produktet er det initelle resiprokalet ganget med seg selv, før dette produktet multipliseres med dividenden Y. Videre skal dette produktet trekkes fra 2 ganger det initelle resiprokalet. Resultatet vises i ligningen under.

$$\frac{1}{Y} = 2z_0 - z_0 z_0 Y$$

Her er z_0 det initelle resiprokalet som kommer fra tabellblokken og Y er divisoren. Dersom divisor rapporten ikke er ”00” sendes dividend X og divisorrapport vektorene videre til kvotientblokken uten å utføre punkt 5 og 6 i kapitel 5.1.5, så snart kvotient blokken er ledig. Dataflytdiagram for denne blokken vises i figur 20, størrelsen på vektorene som benyttes for iterasjonen som beregner resiprokalet, vises i tabell 5. Oppførselen til Newton-Raphson-blokken beskrives av tilstandsdiagrammet i figur 21.

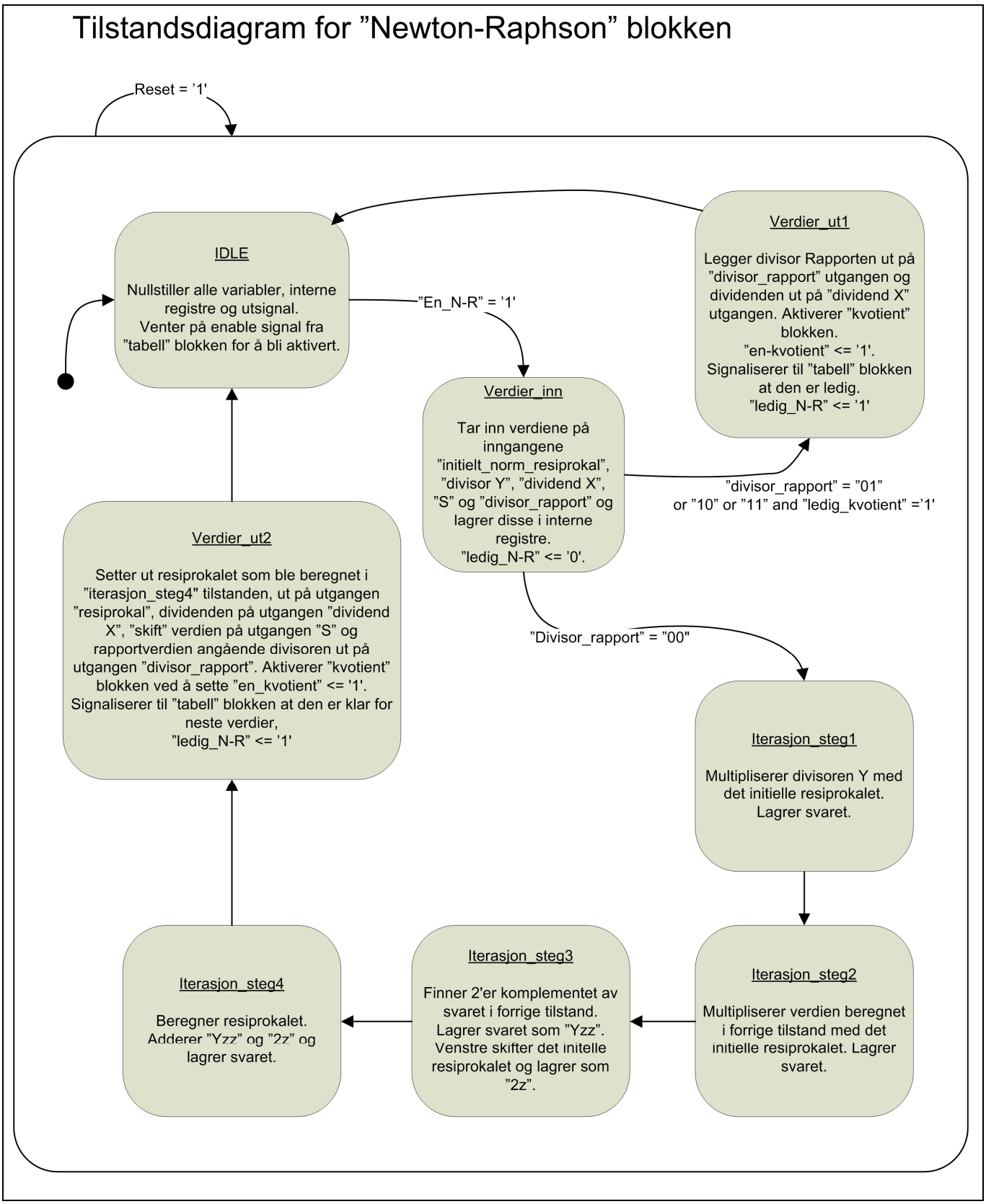
Multiplikatoren som benyttes i Newton-Raphson-blokken har stor betydning for ytelsen da blokken utfører 2 multiplikasjoner per iterasjon.



Figur 20: Dataflytdiagram for generell Newton-Raphson-algoritme for divisjon. Figuren viser ikke logikk for normalisering. Figuren er hentet fra [1].

Term	Størrelse	q = 16 bit
X	q	16
Y	q	16
Z_n	$q+2$	18
$Z_n \cdot Z_n$	$(q+2)(q+2) \rightarrow q+2$ (trunkert)	$18 \cdot 18 \rightarrow 18$
$Y \cdot (Z_n \cdot Z_n)$	$q(q+2) \rightarrow 2q+2$	$16 \cdot 18 \rightarrow 34$
$2 \cdot Z_n - (Y \cdot Z_n \cdot Z_n)$	$(q+2) - (2q+2) \rightarrow 2q+2$	34
$X \cdot Z_n$	$q(q+2) \rightarrow 2q+2$	$16 \cdot 18 \rightarrow 34$
$Q \cdot Y$	$q \cdot q \rightarrow 2q$	$16 \cdot 16 \rightarrow 32$
$X - (Q \cdot Y)$	$q - 2q \rightarrow 2q$	32

Tabell 5: Tabellen viser størrelsen på de ulike vektorene som benyttes i Newton-Raphso-algoritmen.



Figur 21: Tilstandsdiagram for ”Newton-Raphson” blokken på figur 14.

Kvotientblokken
Denne blokken utfører de siste beregningene i divisjonsprosessen, punkt 7 i den punktvise oppsummeringen av divisjonsprosessen i kapittel 5.1.5.

Blokken tar inn resiprokalet, dividenden X, skiftverdien, og divisorrapporten fra Nwton-Raphson-blokken. Divisor rapporten sjekkes for å avgjøre hva som skal utføres. Dersom divisorrapporten er ”00”, beregnes kvotientvektoren og restvektoren på normal måte. Dette gjøres ved å multiplisere resiprokalet med dividenden X. Resultatet blir en 32 bits vektor. Deretter benyttes ”skiftverdien” fra normaliseringsprosessen i divisorblokken til å bestemme hvor kommaet skal plasseres. Bitene til venstre for kommaplassen lagres som en 16 bits kvotientvektor, og bitene til høyre for kommaplassen lagres som en 16 bits restvektor. Deretter legges både kvotient- og restvektoren ut på sine respektive utganger til omverdenen for utlesing samtidig med at det varsles ut at divisjonsprosessen er ferdig. Dette gjøres på utgangen ”divisjon_ferdig” med en ener.

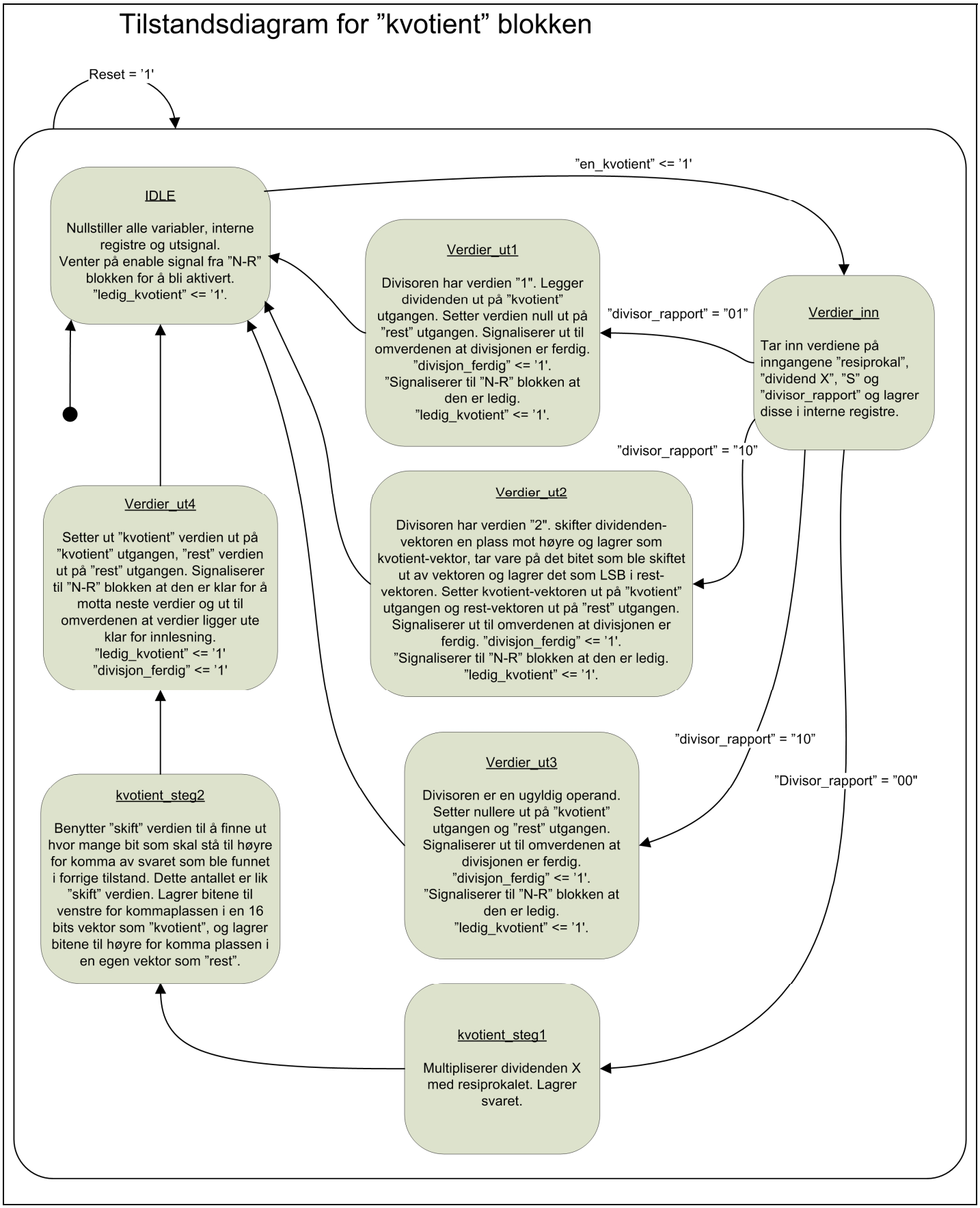
Dersom divisorrapporten er lik ”01”, betyr det at divisoren har verdien en. Da plasseres dividenden X ut på kvotientutgangen (se figur 14 for utgangene) og nullere settes på restvektoren, og divisjonen er ferdig.

Dersom divisjons rapporten er lik ”10” betyr det at divisoren har verdien to. Da skiftes dividenden en plass mot høyre og lagres som kvotienten. Bitet som skiftes ut plasseres i restvektoren på minst signifikante plass. Videre settes både kvotient- og restvektoren ut på sine respektive utganger (se figur 14 for utgangene).

Dersom divisjonsrapporten er lik ”11” betyr det at enten divisoren Y og/eller dividenden X er ugyldig eller at dividenden har verdien null. Da legges nullere ut på både kvotient- og restutgangen.

I denne blokken har multiplikatoren som benyttes stor betydning for blokkens ytelse ved normale operander, det vil si at divisorrapporten er likk ”00”.

Kvotientblokkens oppførsel beskrives av tilstandsdiagrammet i figur 22.



Figur 22: Tilstandsdiagram for ”kvotient” blokken på figur 14.

5.2 Løsning for 32 bits operander

For 32 bits operander er det blitt skissert en løsning som baserer seg på løsningen for 16 bits operander. Økningen fra 16 til 32 bit innebærer noen nye utfordringer ved utførselen av punkt 3, 4, 6 og 7 i den punktvise oppsummeringen av divisjonsprosessen som er beskrevet i kapittel 5.1.5. Noen av utfordringene krevde endringer.

Ved normaliseringen av divisoren Y må det avgjøres hvilken plass mest signifikante ener har for så å høyreskifte divisoren til den har en verdi under 2. Dersom denne posisjonen skal avgjøres vedhjelp av logikk slik som ved 16 bits operander, vil denne bli mer kompleks og arealkrevende. Det er mulig, og ytelsesmessig lønnsomt å gjøre det ved logikk for å få en hurtig avgjørelse. Hvis ikke måtte antall høyreskift blitt telt inntil mest signifikante ener står på minst signifikante plass i vektoren, noe som er tidkrevende.

I forbindelse med utføringen av Newton-Raphson-iterasjonen og beregningen av kvotienten, kreves det en hurtig 32 x 32 bit multiplikator.

5.2.1 Endringer fra 16 bits løsning

Oppslagstabellen for å skaffe det initelle 32 bits resiprokalet byr på så store utfordringer at det er nødvendig med en ny løsning.. Med den bipartite tabellen vil det kreves 390Mb eller 48MB for å lagre alle verdiene ved benyttelse av 32 bits operander. Disse verdiene er beregnet i kapittel 4.4.1. Denne datamengden lar seg ikke lagre i RAM på noen FPGA. Dermed må det benyttes eksternt minne som kan adresseres fra FGPAen for å skaffe det initelle resiprokalet. Flashminne har i dag en overføringshastighet på langt over 50MB eller 400Mb per sekund, og har lagrings kapasitet langt over det som kreves for å lagre en bipartit tabell for 32 bits operander. Eksternt RAM har også tilstrekkelig lagringskapasitet og kan kjøres på 100 MHz klokke. Med eksternt RAM med 100MHz klokke og en linje inn til FPGA per bit, vil det kunne ta 2 klokkeperioder å utføre ett oppslag som da vil ta 20ns. Dermed blir det mulig å utføre 50 millioner oppslag i det eksterne minnet per sekund.

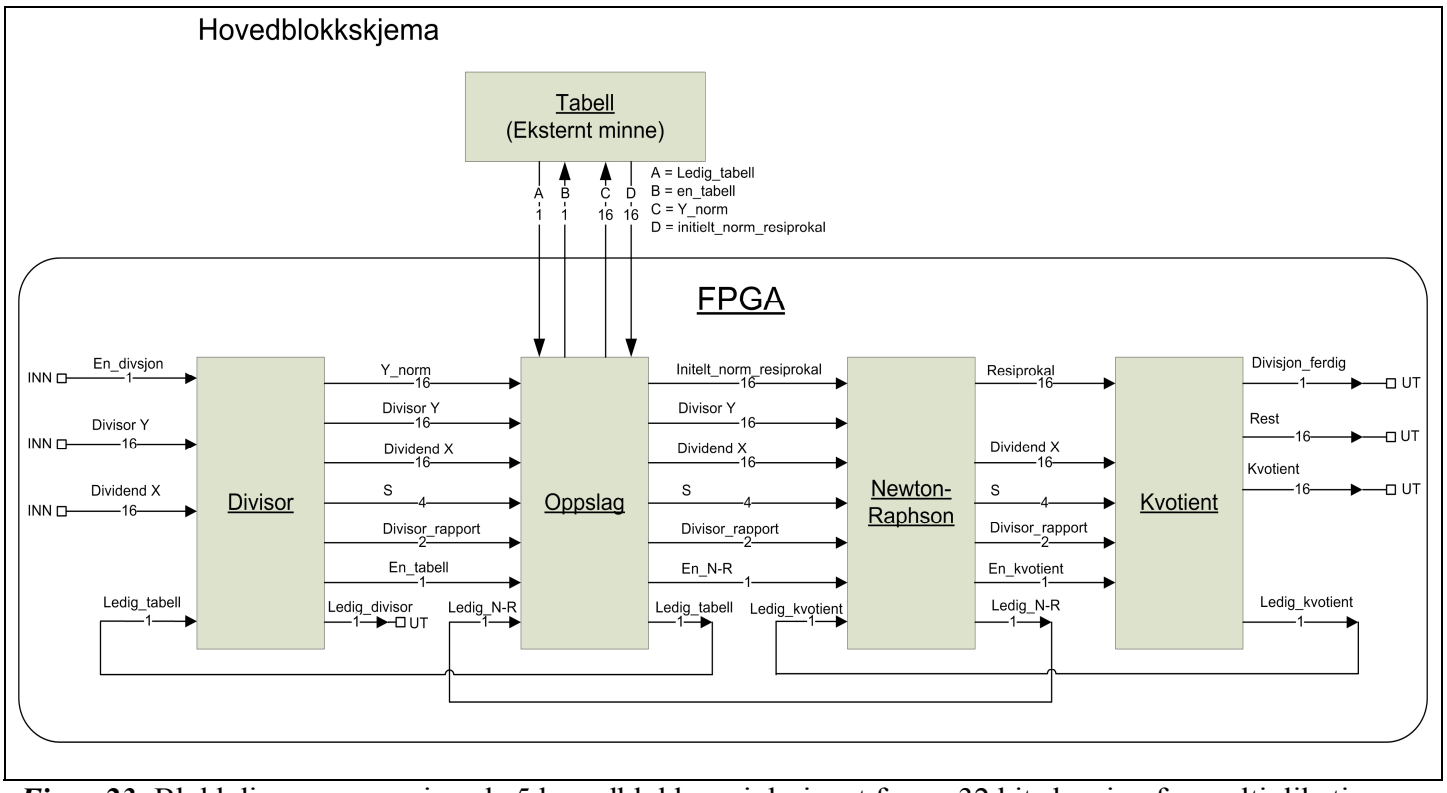
5.2.2 Arkitektur

Da løsningen for oppslagstabell må endres i forhold til løsningen for 16 bits operander, blir hovedblokkskjemaet også endret. Divisjonsprosessen er nå fordelt på 5 hovedblokker som hver utfører sin del av prosessen. 4 av hovedblokkene befinner seg på FPGAen, mens den 5. er det eksterne minnet. Hovedblokkskjemaet for 32 bits løsning vises i figur 23.

Strukturen på blokkene er svært lik den med intern tabell i RAM på FPGA bortsett fra at den interne tabellblokken er byttet ut med en oppslagsblokk. Denne oppslagsblokken kommuniserer ut mot det eksterne minnet og adresserer det med den normaliserte divisoren fra divisorblokken.

Akkurat som med løsningen for 16 bits operander, utfører hver blokk sin del av divisjonsprosessen. For hver blokk divisjonsregnestykket går i gjennom kommer det nærmere en kvotient og en rest. Blokkene kommuniserer seg i mellom og sier i fra når de er ledige å klare til å ta i mot neste verdier. Når de har beregnet ferdig verdier som er klare for å sendes til neste blokk, venter de til neste blokk er ledig før verdiene sendes.

Sekvensen på blokkene som verdiene går i gjennom er: Divisorblokken, Oppslagsblokken, Tabellblokken, Newton-Raphson-blokken før Kvotientblokken avslutter divisjonsprosessen.



Figur 23: Blokkdiagram som viser de 5 hovedblokkene i designet for en 32 bits løsning for multiplikativ divisjon.

6 Diskusjon

6.1 Valg av løsninger

Ut i fra teoristudiet i denne oppgaven ble noen metoder valgt for å beregne den inverterte verdien av divisoren. Disse metodene ble valgt med tanke på ytelse. Metodene består av Newton-Raphson-algoritmen for iterering og en tabelløsning for initialverdiene til resiprokalet som skal det skal itereres over.

Av de to iterative algoritmene Newton-Raphson og Goldschmidt som er beskrevet i denne rapporten, er det mulig å benytte begge til multiplikativ divisjon. Men det er kun Newton-Raphson-algoritmen som dobler antall korrekte bit for hver iterasjon, og som dermed egner seg best til multiplikativ divisjon. Feilanalysen til Goldschmidts algoritme er så kompleks at det ikke er sikkert om den gir høyere ytelse enn Newton-Raphson-algortimen selv om multiplikasjonene i Goldschmidts algoritme kan utføres i parallell.

For å skaffe initialverdien til resiprokalet ble det benyttet en bipartit oppslagstabell. Denne løsningen krever større lagringsplass enn en multipartit tabell. En hurtig gjennomgang av multipartite tabeller tilsa at det vil være en bedre løsning enn en bipartit tabell. Dessverre ble det ikke tid nok til å utrede og utvikle en multipartit oppslagstabell i detalj for å ta i bruk som oppslagstabell.

Oppslagstabellen for initialverdien til resiprokalet går fint inn i RAM på de aller fleste FPGAer ved 16 bits operander. Men med en nøyaktighet på 1 ULP, blir oppslagstabellen svært plasskrevende for 32 bits operander og må plasseres i eksternt minne. Ytelsen på løsningen for 32 bits operander med eksternt minne vil være fullt på høyde med den for 16 bits operander med intern RAM på FPGA, men løsningen er litt mer kompleks.

Siden initialverdiene fra oppslagstabellen har så stor nøyaktighet og Newton-Raphson-algoritmen dobler antall korrekte bit per iterasjon, blir det ikke nødvendig med en lang serie tidkrevende iterasjoner. Med de spesifiserte løsingene er det kun nødvendig å utføre én iterasjon, noe som er svært lavt i forhold til hva som ble antatt helt i starten av arbeidet med oppgaven.

6.2 Mulige forbedringer av løsningene

Multipartite oppslagstabeller ser ut til å ha en klar fordel i forhold til bipartite tabeller med tanke på total tabellstørrelse ved samme grad av nøyaktighet. En multipartit oppslagstabell for det initielle resiprokalet ville ha frigjort en del RAM for både 16 bits operander med intern RAM, og for løsningen med 32 bits operander med eksternt minne.

Newton-Raphson-blokken er den blokken med flest sekvensiell regneoperasjoner av de blokkene som er skissert i implementasjons kapittelet, og blir dermed den tregeste med 2 multiplikasjoner og en subtraksjon. Det hadde vært mulig å benytte 2 slike blokker i parallell for å øke den totale ytelsen til divisjonsmodulen. De fleste FPGAer har i dag flere titalls innebygde multiplikatorer, og de største har opp mot tusen. Disse kan benyttes til Newton-Raphson-blokken og kvotientblokken. Dermed er mulighetene til stede for å benytte flere Newton-Raphson-blokker og eventuelt også kvotientblokker i parallell, og fortsatt ha ressurser igjen til andre realiserte moduler på samme FPGA.

Da Newton-Raphson-algoritmen dobler antallet korrekte bit for hver iterasjon, hadde det vært mulig å benytte mindre oppslagstabeller med mindre nøyaktighet på initialverdiene, og heller øke antallet iterasjoner. Dette hadde gjort det mulig å benytte en bipartit oppslagstabell for en 32 bits løsning med internt minne. Da måtte tabellen inneholdt kun 16 korrekte bit og benyttet Newton-Raphson iterasjon til å doble dette for og få 32 korrekte bit. Denne metoden åpner også muligens for løsninger med 64 bits operander. Dette kunne blitt realisert med en oppslagstabell lagret i RAM på FPGA med kun 16 korrekte bit for hvert resiprokal, og så utført 2 iterasjoner med Newton-Raphson-blokken. Dermed hadde resiprokalet oppnådd 64 korrekte bit til slutt. Med flere iterasjoner per resiprokal ville divisjonsprosessen tatt lengre tid og Newton-Raphson-blokken hadde blitt prosessens flaskehals. Dette kunne ha blitt utbedret ved å benytte flere Newton-Raphson-blokker i parallell. I tillegg kunne det vært aktuelt og økt antallet kvotientblokker.

6.3 Videre arbeid

Dessverre ble det ikke tid til å implementere løsningene i VHDL og simulere og syntetisere de, så her er det fortsatt arbeid som gjenstår. Løsningene som er beskrevet i denne rapporten beskriver hvilken maskinvare og hvilke blokker som kreves for å realisere løsningene på en FPGA. I tillegg er det også utarbeidet tilstandsdiagram for hovedblokkene som vil lette implementasjonsarbeidet i VHDL. Implementasjon og syntetisering av løsningene som er beskrevet i denne rapporten vil gi konkrete tall på arealutnyttelse og ytelse som løsningene har ved realisering på FPGA. For generering av tabellverdiene er pythonkode beskrevet i [3]. Denne koden generer initialverdiene basert på en bipartit oppslagstabell som kan benyttes til implementasjon i VHDL, og egner seg for løsningene som er beskrevet i denne rapporten.

Da multipartite tabeller ser ut til å være en bedre løsning enn bipartite tabeller, er det naturlig å jobbe mer med multipartite oppslagstabeller for lagring av initialverdiene til resiprokalet. Her bør total tabellstørrelse for aktuelle operandstørrelser kartlegges og sammenlignes med tilsvarende verdier for bipartite tabeller, samt utarbeide en løsning som genererer verdiene som skal lagres i tabellen.

Det kunne vært interessant å se mer på feilanalysen til Goldschmidts algoritme da denne har muligheten til å utføre multiplikasjonene i parallell. Dersom det viser seg at feilen for hver iterasjon konvergerer, har algoritmen en mulig ytelses fordel foran Newton-Raphson-algoritmen hvor multiplikasjonene må utføres sekvensielt.

7 Konklusjon

Fra teoristudiet kom det fram at Newton-Raphson-algoritmen er egnet til å implementere en modul i VHDL som utføre iterasjoner. Til å skaffe resiprokalets initialverdi til iterasjonsprosessen er tabelloppslag i en bipartit tabell en egnet løsning, før resiprokalet multipliseres med dividenden. Mot slutten av teorifasen kom det fram at en multipartit tabell vil være en bedre løsning som oppslagstabell enn en bipartit tabell. Dette kommer av at en multipartit tabell gir lavere total tabellstørrelse ved lagring av samme informasjon. Det ble ikke tid til å gå tilstrekkelig i dybden på multipartite tabeller til at en løsning med multipartite tabeller ble benyttet.

Det ble ikke tid til å implementere arbeidet i VHDL for å få eksakte verdier på hvor mye areal som trengs og hvor mange klokkesykler divisjonsprosessen krever. Det ble i stedet fokusert på å analysere og beskrive algoritmer og metoder for multiplikativ divisjon samt og beskrive hvilken maskinvare og arkitektur som trengs for å realisere de spesifiserte løsningene. Implementeringen i VHDL burde være forholdsvis ukomplisert ut i fra arkitekturene som er spesifisert i denne rapporten.

Det ble spesifisert en løsning for 16 bits operander og en løsning for 32 bits operander, som er pipelinet og egner seg for realisering på FPGA. Ut i fra de utvalgte metodene ble oppgavene med divisjonsprosessen beskrevet i detalj og fordelt på hovedblokkene som divisjonsmodulen ble inndelt i. Metodene som ble beskrevet for de spesifiserte løsningene ser ut til å gi best ytelse av de metodene som ble gjenomgått for multiplikativ divisjon. Løsningene beregner divisorens resiprokal relativt hurtig med kun en iterasjon. Løsningen for 32 bits operander er basert på løsningen for 16 bits operander med unntak av oppslagstabellen for initialverdiene som må plasseres i eksternt minne.

Hastigheten på løsningen for 32 bits operander med eksternt minne vil være fullt på høyde med den for 16 bits operander med intern RAM på FPGA, men arkitekturen vil være litt mer kompleks. Løsningene har mulige ytelsesforbedringer ved blant annet å parallellisere Newton-Raphson iterasjonene slik som beskrevet i diskusjonskapittelet.

8 Referanseliste

[1] Simen Gimle Hansen, ”Multifunksjonell beregningsenhet for digital signalprosessering”, Universitetet i Oslo, 2005.

[2] Marius Gimle Hansen, ”Analyse og implementering av multiplikative divisjonsalgoritmer”, Universitetet i Oslo, 1997.

[3] Martin Rognerud, ”Konstruksjon av digital heltalsaritmetikk”, NTNU, 2007.

[4] Glenn Hinton et al, “The Microarchitecture of the Pentium® 4 Processor”, Intel Technology Journal Q1, 2001.

[5] H. P. Sharangpani, M. L. Barton, “Statistical Analysis of Floating Point Flaw in the Pentium™ Processor”, Intel Corporation, November 1994.

[6] J. b u n Prabhu and Gregory B. Zyner, ”167 MHz Radix-8 Divide and Square Root Using Overlapped Radix-2 Stages”, Proceedings 12th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1995, pp 155-162.

[7] Michael J. Flynn, Stuart Oberman, Steve Fu, ”The SNAP Project: Towards Sub-Nanosecund Arithmetic” Stanford University, Proceedings 12th Symposium on Computer Arithmetic , IEEE 1995.

[8] Debjit Das Sarma, “Faithful Bipartite ROM Reciprocal Tables”, Proceedings 12th Symposium on Computer Arithmetic , IEEE 1995.

[9] Behrooz Parhami, “Computer Arithmetic algorithms and hardware designs, Division”, UCSB, 2007

[10] Florent de Dinechin, Member, IEEE, and Arnaud Tisserand, Member, IEEE ”Multipartite Table Methods”, IEEE Transactions on Computers, vol. 54, no. 3, March 2005.

[11] Peter Kornerup, David W. Matula, ”Single Precision Reciprocals by Multipartite Table Lookup” Proceedings of the 17th IEEE Symposium on Computer Arithmetic, 2005 IEEE.

[12] Altera Product Guide: “http://www.altera.com/products/prd-index.html”.

[13] Xilinx Silicon Devices: ” http://www.xilinx.com/silicon_solutions/fpgas/index.htm”.

[14] Actel Product Guide: ” http://www.actel.com/products/default.aspx”.

[15] Guy Even, Peter-Michael Seidel, “A Parametric Error Analysis of Goldschmidt’s Division Algorithm”, Proceedings of the 16th IEEE Symposium on Computer Arithmetic, 2003 IEEE.

[16] R.V.K. Pillai, D. Al-Khalilitand A.J. Al-Khalili “Energy Delay Analysis of Partial Product Reduction Methods for Parallel Multiplier Implementation” ISLPED 1996 Monterey CA USA.

[17] Tso-Bing Juang, Shen-Fu Hsioao, “A Cell-Driven Multiplier Generator with Delay Optimization of Partial Products Compression and an Efficient Partion Technique for the Final Addition”, IEICE Trans. INF. & Syst. Vol.E88-D, no.7 july 2005.

[18] J.B. Kuang], T.C. Buchholtz', S.M. Dance', J.D. Wamock3, “The Design and Implementation of Double-Precision Multiplier in a First-Generation CELL processor”, 2005 IEEE International Conference on Integrated Circuit and Technology.

[19] Arnaud Tisserand Ar_enaire INRIA LIP, “Algorithms and Number Systems for Hardware Computer Arithmetic”, ISSAC 2005, Tutorial Beijing, China.

[20] Stuart F. Oberman, Student Member, IEEE, and Michael J. Flynn, Fellow, IEEE “Division Algorithms and Implementations“,IEEE TRANSACTIONS ON COMPUTERS, VOL. 46, NO. 8, AUGUST 1997.