NTNU
Det skapende universitet

FPGA-basert styresystem for kybernetiske proteser

Marius Andre Mossum

Master i elektronikk
Oppgaven levert: Juli 2006
Hovedveileder: Bjørn B. Larsen, IET
Biveileder(e): Øyvind Stavdahl, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

I sammenheng med elektronisk styrte protesekomponenter er liten størrelse, lav vekt og høy ytelse ofte de viktigste faktorene for suksess. Miniatyrisering av styringsfunksjoner og bruk av moderne børsteløse motorer er derfor svært aktuelt.
I denne oppgaven skal resultatene fra et tidligere prosjektarbeid videreutvikles. Med utgangspunkt i foreliggende spesifikasjon skal en vurdere egnetheten av FPGA-teknologi som hardwareplattform, for å holde størrelsen nede og samtidig oppnå tilstrekkelig hastighet/ responstid på kritiske deler av styresystemet, for håndleddsprotesen ”NRWD” og eventuelt senere proteseprosjekter. Det skal tas utgangspunkt i Atmel’s FPSLIC, en brikke hvor både FPGA og mikrokontroller er tilgjengelig.
1. Konstruer en prototyp i VHDL, som inneholder de av systemets funksjoner som ansees hensiktsmessig å implementere i hardware. Det er ikke noe krav at funksjoner valgt lagt til software blir implementert, men valgene må beskrives og begrunnes.

2. Utred mulighetene for å styre flere uavhengige motorer fra en og samme FPGA-krets, sett i lys av tilgjengelige ressurser på den valgte brikken. Det er ønskelig at systemet konstrueres modulært/skalerbart med tanke på eventuell håndtering av flere motorer.

3. Utred mulighetene for å implementere CAN-kontroller i FPGA. Diskuter hvilke konsekvenser dette eventuelt vil kunne få for totalsystemet, slikt som utviklingstid, komponentkostnad og fysisk størrelse.
4. Utfør simuleringer av systemets hardwaremoduler på funksjonelt nivå, under stasjonære forhold. Dersom tiden strekker til utføres fysisk test av systemet via utviklingskort og nødvendig tilleggshardware. Foreta en systematisk evaluering av prototypen, der det legges spesielt vekt på tidskritiske funksjoner. Diskuter resultatene i lys av foreliggende mikrokontrollerbaserte løsning [Skjelten, 2005].

Oppgaven gitt: 23. januar 2006
Hovedveileder: Bjørn B. Larsen, IET

Sammendrag:

I denne oppgaven ser man nærmere på hva muligheter som ligger i å kombinere FPGA-teknologi og mikrokontroller-teknologi for bruk i moderne kybernetiske proteser. Mer spesifikt tar arbeidet utgangspunkt i brikken Atmel AT94K (FPSLIC), en brikke som i essens er en tradisjonell AVR mikrokontroller med integrert FPGA. I første omgang er arbeidet rettet mot protesen NRWD (NTNU Revolute Wrist Device), som er et pågående utviklingsprosjekt på ITK. Samtidig har arbeidet det perspektivet at teknologien bør kunne komme fremtidige protese-prosjekter til gode.

Det man søker er å komme frem til en plattform som er generell nok til at den med et minimum av tilpassninger kan benyttes i en vilkårlig kybernetisk protese. Mer spesifikt et system som ikke krever annet enn tilpasning av software ved integrering i en ny protese. I den grad tilpasninger av hardware er påkrevd, så bør ikke dette dreie seg om annet enn enkel skalering. Det prioriteres høyt at endringer av fysisk utlegg ikke skal være nødvendig. Dette innebærer at mulighet for samtidig styring av flere motorer må implementeres i FPGA, og at all eventuell skalering kan utføres her. Samtidig bør flest mulig av de tyngre funksjonene utføres i hardware, for å frigjøre mest mulig CPU-tid til spesialtilpasninger.

Utgangspunktet for oppgaven er i stor grad prosjektarbeidet utført av Marius Mossum (høsten 2005). Her fremmes en del generelle forslag til hvordan systemet kan designes; samt at en struktur for kjernefunksjonaliteten blir foreslått. Basert på dette skal de nødvendige vurderingene gjøres, før implementering i VHDL finner sted. Utredningen har til en viss grad totalsystemet som perspektiv, mens implementeringen kun dreier seg om funksjonalitet valgt lagt til hardware. Videre skal systemet vurderes opp imot en rekke kriterier, og avslutningsvis sammenlignes med foreliggende løsning basert på Håkon Skjeltens arbeid (Vår 2005).

Det første arbeidet tar for seg av rent praktisk art er en videre analyse av kravene til systemet, i utstrekning av den som ble utført i [Mossum 2005]. Her innbefattes både en detaljering av allerede utredete krav, og en klargjøring av de funksjonelle krav som har tilkommet i kjølevannet av samme prosjekt. Dette omfatter primært hvordan systemet bør organiseres, hva grunnleggende krav som stilles til motorstyringen, og noe om hva man ser for seg utført i software. I den forbindelse diskuteres også noen av kravene som stilles til softwaren, av den grunn at det har en viss relevans for de senere utredningene. Men det viktigste for denne delen av arbeidet er i alle tilfeller utredningen av hva krav som stilles til motorstyringen, for best mulig utnyttelse av den benyttede motoren.

Arbeidet har ført frem til et i grove trekk verifisert design for uavhengig kommutering og styring av fem uavhengige trefase børsteløse likestrømsmotorer (BLDC). Størst mulig skalerbarhet er tilstrebet, og en modulær designstruktur er benyttet. Dette for å lette arbeidet ved eventuelle senere utvidelser og tilpasninger. Et perspektiv har også vært areal- og effektoptimalisering ved å begrense bredden på komplekse beregninger, og å unngå redundant funksjonalitet. I dette ligger blant annet deling av en divisjonskrets mellom de fem motorkontrollerne, og multipleksing av ett enkelt pwm-signal fremfor bruk av tre individuelle generatorer per motor. I utstrekning av kommuteringen er en konfigurerbar kommuteringsforsinkelse implementert. Dette for å verne drivtrinnene fra kortslutning ved for hurtige kommuteringsskift. Hver motorkontroller har også fått integrert kjøretidsberegning av hastighet, med behovsprøvd generering av interrupt til mikrokontroller. Interrupt er et tiltak for ikke å laste mikrokontrolleren med periodisk polling av hastighetsmålingen. Videre har arbeidet ledet frem til en rekke potensielle utvidelser av funksjonaliteten, samt enkelte mer eller mindre nødvendige forbedringer av gjeldende design.

En takk rettes til:

En stor takk rettes til faglærer og veileder Bjørn B Larsen for en enormt hjelpsom innstilling under veis i det praktiske arbeidet. Den tid og oppmerksomhet jeg ble vist når problemene meldte seg har vært over all forventning. Han skal også ha kreditt for å ha stilt seg disponibel på kort varsel ved akutte behov.

Likeledes vil jeg takke prosjektledere og Øyvind Stavdahl for den måten han alltid har klart å fokusere på de positive tingene under veis. Denne mannen har en sjeldent god evne til å alltid trekke frem momenter i arbeidet som både stimulerer kreativiteten og gir ny motivasjon. Han skal også ha en stor takk for å ha vist stor åpenhet for mine ideer og innspill.

Innholdsfortegnelse:

1. Innledning / bakgrunn 3
1.1 Tidligere arbeid 3
1.2 Arbeidets målsetning 3
1.3 Arbeidets fokus, og endring av fokus 4
1.4 Rapportens struktur 5
2. Plattform og applikasjon 7
2.1 FPSLIC – Om brikken 7
2.1.1 Generelt om oppbygning og muligheter 7
2.1.2 Egnethet til anvendelse i NRWD og lignende proteser 9
2.2 Atmel System Designer – Kommentarer og erfaringer: 11
3. Virkemåte og styring av BLDC-motorer 13
3.1 Motorens oppbygning og virkemåte 13
3.2 Prinsipper for styring av BLDC-motorer 15
3.3 Ulike kommuteringsmetoder for BLDC-motorer 17
3.3.1 Avansert kommutering ved hjelp av estimert rotorvinkel 18
3.3.2 Pådragsstyring av BLDC-motor 18
4. Funksjonelle systemkrav og overordnede strategier 21
4.1 Relevante krav til totalsystemet 21
4.2 Funksjonelle krav til hardwaren (FPGA) 23
5. Systemdesign 27
5.1 Strukturelt design – Funksjonell partisjonering 27
5.1.1 Moduler i hardware 28
5.1.2 Software: 30
5.1.3 Hardware / software - Partisjonering: 31
5.2 Implementasjon – Systemets virkemåte 32
5.2.1 Kommunikasjonsmodulen 32
5.2.2 Motorkontroller – Overordnet grensesnitt: 36
5.2.3 Arbiter og divisjonsmodul: 38
5.2.4 Motorkontroller – Indre struktur: 40
5.3 Innhentede moduler 47
5.3.1 PWM: 47
5.3.2 Divisjon: 48
5.4 Systemets skalerbarhet 50
6. Systemtest / simulering / evaluering 51
6.1 Testmetode og testbenker 51
6.2 Simuleringsresultater 52
6.2.1 Motor_controller – Undermoduler: 52
6.2.2 Motor_controller – Sammensatt modul: 58
6.2.3 Communicator: 60
Communicator – Fortsettelse: 62
6.2.4 Division_unit: 64
6.2.5 Komplett system: 66
6.3 Synteseresultater 68
7. Diskusjon og videre arbeid 69
7.1 Mulige utvidelser og forbedringer av designet 69
7.1.1 Mulige forbedring av designet 69
7.1.2 Tiltak for øket funksjonalitet 74
7.1.3 Areal- og strømbesparende tiltak 76
7.1.4 Design for øket skalerbarhet 77

7.2 Implementering av CAN-bus ... 77
7.3 Sett i lys av Skjeltens løsning ... 80
7.4 Anbefalinger for det videre arbeidet. ... 81
8. Konklusjon ... 83
Referanser ... 84

1. Innledning / bakgrunn

1.1 Tidligere arbeid

Opp gjennom årene har det blitt gjennomført en rekke prosjekter og hovedoppgaver relatert til digital motorstyring og styringselektronikk for proteser. Mange av disse under veiledning av Øyvind Stavdahl, og med NRWD som felles mål. De kanskje viktigste arbeidene for å komme frem til en prototyp av NRWD har vært Terje Wessel Pettersens ”Digital Motorstyring” fra 1997, og Håkon Skjeltens ”Innvevd maskinvare for kybernetisk håndledd” fra våren 2005. Terje Wessel Pettersen gjorde en uvurderlig innsats med å analysere motorens oppbygning og virkemåte, som har vært brukt i flere prosjekter i ettertid. Håkon Skjelten var den første som bygget en fullt funksjonell krets spesifikt for NRWD. Dette designet ble brukt som utgangspunkt for en miniatyrisert prototyp, bygget i samarbeid mellom ITK og Sintef.

Andre relevante arbeider for NRWD er: ”Mikrokontrollerbasert motorstyring for børsteløs DC-motor” av Pål Ronny Kastnes i 97, ”Konstruksjon av et kontrollerkort for styring av underarmsprotese” av Trude Forsbak i 97, og ”Lavnivåregulering av protesemotor” av Anders Veberg i 98. Felles for alle disse er at de baserer seg på bruk av mikrokontrollere i styringskretsen. En annen retning, som kanskje var tidligere ute totalt sett, var bruk av ren digital hardware for å realisere styringsfunksjonene. Først ute her var antakelig Jan Gunnar Dyrset og Per Øivind Eger, med sitt prosjekt ”Digital krets for motorstyring realisert med programmerbare komponenter”, utført våren 1995. Her benyttet de en slags PAL-krets, som forøvrig har en del til felles med FPGA-teknologi. Men denne strategien ble altså lagt til side etter hvert som mikrokontrollerne gjorde seg mer gjeldende innenfor tilpassende datasystemer.

Det var først høsten 2005 at Marius Mossum (undertegnede) tok opp igjen dette arbeidet med et prosjekt ved tittelen ”FPGA-basert motorstyring for protesehåndledd”. Hensikten med dette prosjektet var å utrede hva moderne FPGA-teknologi kunne bidra med til anvendelsen, og å foreslå strukturen for et mulig design. Ulike strategier ble evaluert; og et design basert på en hybrid-brikke fra Atmel ble foreslått. Brikken bærer navnet FPSLIC (AT94K), og har den noe spesielle egenskapen at både en AVR mikrokontroller og en AT40K FPGA er å finne om bord. Man håpet med denne brikken å kunne oppnå enda mer fleksible løsninger, ved å ta det beste fra to verdener, integrert i en og samme brikke. Dette prosjektet, og den foreslåtte løsningen, dannet altså grunnlaget for denne hovedoppgaven.

1.2 Arbeidets målsetning

Arbeidets målsetning har vært å utrede nærmere de mulighetene som ligger i en eventuell ny hardware-plattform for kybernetiske proteser, samt å utvikle en testbar beskrivelse av motorstyringen i VHDL. Det er her tatt utgangspunkt i [Mossum 2005], og det strukturelle designet som ble foreslått der.

En del av arbeidet har gått på å vurdere egnetheten av FPSLIC til anvendelsen, samt å utrede noen av de tekniske utfordringene det vil innebære. En annen del av arbeidet har gått på å implementere motorstyringen i VHDL. Målet her har vært å komme frem til en fungerende løsning, verifisert ved simulering under stasjonære forhold. Altså uten en matematisk modell av motoren. Designet er i første omgang tenkt brukt som test og evalueringsgrunnlag sammen med et utviklingskort for FPSLIC. Men først må programvare for mikrokontrolleren skrives. På lengre sikt er håpet at løsningen skal kunne benyttes ved integrering i et komplett system.

En del av målsetningen har også vært å komme med en anbefaling for det videre arbeidet. Både når det gjelder hvorvidt man bør satse på denne teknologien, og eventuelt på hvilken måte. Med andre ord en anbefaling vedrørende egnetheten av FPSLIC spesielt, og hybrid-strategien generelt. Avslutningsvis vil løsningen bli diskutert i forhold til Skjeltens løsning.

1.3 Arbeidets fokus, og endring av fokus.

I begynnelsen av arbeidet fikk FPSLIC, og tanker rundt implementering av en fullverdig prototyp basert på denne, uhensiktsmessig mye oppmerksomhet. En del tid gikk med til fordypning i ting som egentlig ikke ville være relevant for implementering av selve motorstyringen. Bakgrunnen for dette var at man i denne tidlige fasen hadde et ønske om å bygge en fysisk krets for testing, selv om dette ikke er et uttalt krav i oppgaven. Dette viste seg imidlertid raskt å ville innebære en rekke premature beslutninger, samt at det ville sette en del av utredningene i skyggen. I samarbeid med veileder kom man derfor frem til at det antakelig ville tjene prosjektgruppa mer å få en bredere anbefaling, enn å bli presentert én enkelt løsning. Derfor valgte man altså å nøye seg med simulering av systemet.

I denne tidlige fasen ble det da også brukt mye tid på å søke etter opplæringsmateriell i Atmel SystemDesigner. Dette viste seg imidlertid å være en mer utfordrende oppgave enn antatt. Brorparten av de treffene man fikk på nett linket til ulike forum. De fleste av trådene var dessverre bare beklagelser over manglende dokumentasjon og høy brukerterskel. En del tid gikk derfor med til å klikke rundt i verktøyene på måfå, uten å sitte igjen med nevneverdig forståelse. Sannsynligvis også en del på grunn av begrenset erfaring med tilsvarende verktøy fra tidligere. Det gjorde ikke saken enklere å være den første på instituttet som benyttet denne pakken. Etter hvert som en overordnet forståelse begynte å melde seg, fant man at programpakken først tjente sin fulle hensikt ved integrering av et komplett system. For simulering latet det til at andre verktøy kunne være vel så greie å benytte.

Etter rådføring med faglærer kom man frem til at en bedre løsning var å gå over til bruk av FPGA Advantage. Dette er en programpakke som er tilgjengelig på instituttet, i tillegg til at flere her har noe erfaring med verktøyene. Denne viste seg da også å ha mulighet for syntese mot FPSLIC. Imidlertid ville man sannsynligvis måtte benytte Atmel SystemDesigner for integrering med programvare for mikrokontrolleren.

Utover i arbeidet fant man det også hensiktsmessig å frigjøre seg litt fra fokus på FPSLIC og NRWD, av årsaker som vil bli utdypet senere i rapporten. Dette ga samtidig mulighet for å rette et noe mer kritisk blikk mot eventuelle svakheter ved brikken, samtidig som man i større grad tillot seg å vurdere andre alternativer. Derfor har man også tilstrebet å gjøre implementasjonen så generell at en vilkårlig FPGA kunne vært benyttet, med et minimum av omskriving. Disse perspektivene vil være å se igjen i flere av betraktningene utover i rapporten; spesielt i utredningene.

1.4 Rapportens struktur

Nedenfor følger en summarisk beskrivelse av sentrale tema for hvert av rapportens kapitler. Dette for å raskere gi leseren et overblikk over rapporten; samtidig som det kan gjøre det lettere å finne tilbake til noe man eventuelt ønsker å se nøyere på. Beskrivelsen forholder seg kun til hovedkapitler, og overlater underkapitler til innholdsfortegnelsen.

Kapittel 2 – Plattform og applikasjon:
Her vil først brikkens egenskaper bli summarisk gjennomgått. I den sammenheng vil også de spesielle mulighetene som ligger i brikken bli kommentert. Videre tar man for seg i hvilken grad, og på hvilken måte man anser brikken som egnet til anvendelsen. Avslutningsvis vil en kort kommentar til DAK-verktøyet SystemDesigner være å finne.

Kapittel 3 – BLDC-motoren – Virkemåte og styring:
I dette kapittelet blir den nødvendige teorien og forståelsen av motoren presentert summarisk. Fremstillingen bærer preg av et anvendt perspektiv heller enn en matematiske beskrivelser av motoren. *Dette kapittelet er forøvrig et direkte utdrag av kapittel 1 fra [Mossum 2005].* For nærmere informasjon om motorens matematiske beskrivelse henvises det til [Pettersen 1997].

Kapittel 4 – Funksjonelle systemkrav og overordnede strategier:
Dette kapittelet baserer seg dels på [Mossum 2005], og dels på krav fremkommet av samtaler med prosjektleder og veileder Øyvind Stavdahl. Kapittelet er sånn sett både en utdyping av kjente krav, og en oversikt over nye krav som har fremkommet i løpet av arbeidet. Enkelte momenter vil også bli diskutert av hensyn til deres relevans for vurderinger og anbefalinger til det videre arbeidet.

Kapittel 5 – Systemdesign:
Dette kapittelet tar for seg alt som relaterer seg til design og implementasjon. Kapittelet er det desidert lengste i rapporten; og består derfor også av flere deler:

Første del tar for seg det strukturelle designet og den funksjonelle partisjoneringen. Dette innebærer også grunnleggende utredning av funksjoner som er tenkt plassert i software, da dette både har relevans for kommunikasjonsbehovet og for enkelte av utredningene utover i rapporten. I samme moment presiseres det ytterligere hva av totalsystemet som er arbeidets fokus; og hva som ansees som krav til andre deler av systemet.

Andre del tar systemets implementasjon og virkemåte mer detaljert for seg. Designet er her forsøkt fremstilt på en systematisk måte ved å begynne med de overordnede strukturene; for så å ta for seg den enkelte modulens virkemåte. Hele veien igjennom er forklaringene lagt på et funksjonelt nivå, uten å trekke inn kildekode. Dette for at det skal være mulig å sette seg inn i arbeidet uten bakgrunn i VHDL; noe som sannsynligvis vil være situasjonen for "kunden". For alle de viktigste modulene er den tekstlige forklaringen støttet av illustrerende figurer. Man har også tilstrebet å bruke autentiske signalnavn, slik at rapporten om ønskelig skal kunne leses parallelt med kildekoden. I den grad matematiske beregninger har vært nødvendig, så er også disse tatt med etter hvert. Moduler som er innhentet fra eksterne kilder er ikke kommentert inngående i denne delen.

Tredje del av kapittelet beskriver de innhentede modulene i systemet summarisk. Men siden disse modulene ikke har blitt studert i detalj, så fokuserer teksten hovedsakelig på hvordan de

interfaces og brukes i systemet. I den grad spesifikke tilpasninger og optimaliseringer er gjort med tanke på disse, så taes dette med her.

Fjerde og siste del kommenterer systemets skalerbarhet kort. Her oppsummeres de viktigste tingene som er gjort for å få systemet mest mulig skalerbart. Samtidig oppsummeres de viktigste tingene som eventuelt må gjøres, dersom skalering skal utføres.

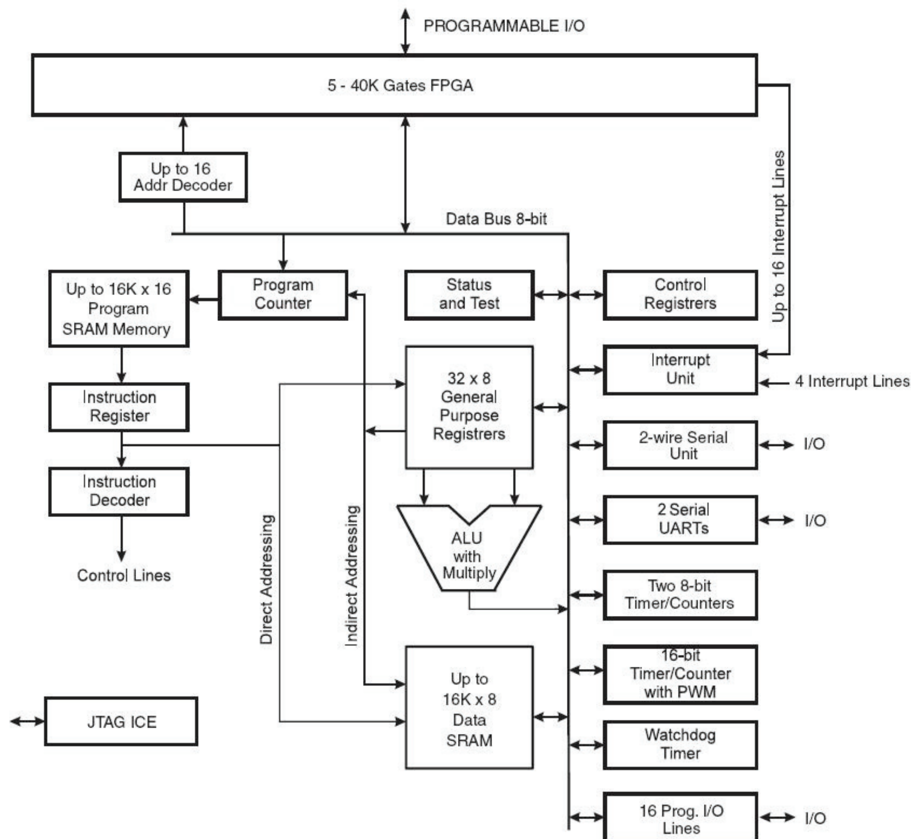
Kapittel 6 – Systemtest / simulering / evaluering:
Dette kapittelet er hovedsakelig myntet på beskrivelse av de simuleringene som er gjort, og hva man har tolket ut av disse. Simuleringer fra de enkelte modulene presenteres med figurer, resultatene kommenteres. Avslutningsvis fremlegges synteseresultatene for systemet.

Kapittel 7 – Diskusjon og videre arbeid:
Dette kapittelet er både en diskusjon av det utførte arbeidet, og en stor del anbefalinger til momenter og ideer som kan taes med i det videre arbeidet. Kapittelets siste del er også ment som en sammenfatning av de tanker en har gjort seg under veis, og hvordan disse står i forhold til de forhåpninger og forventninger man hadde til prosjektet innledningsvis. Kapittelet leder sånn sett frem til en konklusjon i kapittel 8.

2. Plattform og applikasjon

2.1 FPSLIC – Om brikken

Da første versjon av FPSLIC ble sluppet på markedet i 2001 ble den omtalt som verdens første ”System On a Chip”. I dette lå det at et komplett tilpasset datasystem nå kunne realiseres på en og samme brikke. Og det til en pris som gjorde brikken aktuell for rimelige tilpassede datasystemer og forbrukerelektronikk. Selv den dag i dag er FPGA og hardkodet mikrokontroller i samme brikke stort sett forbeholdt vesentlig større og dyrere brikker. Andre leverandører ser ut til å heller ha satset på større FPGAer, med mikrokontrollere levert som soft cores (IP-moduler). I figur 1 nedenfor er brikkens struktur illustrert, uten at denne vil bli kommentert nærmere her.

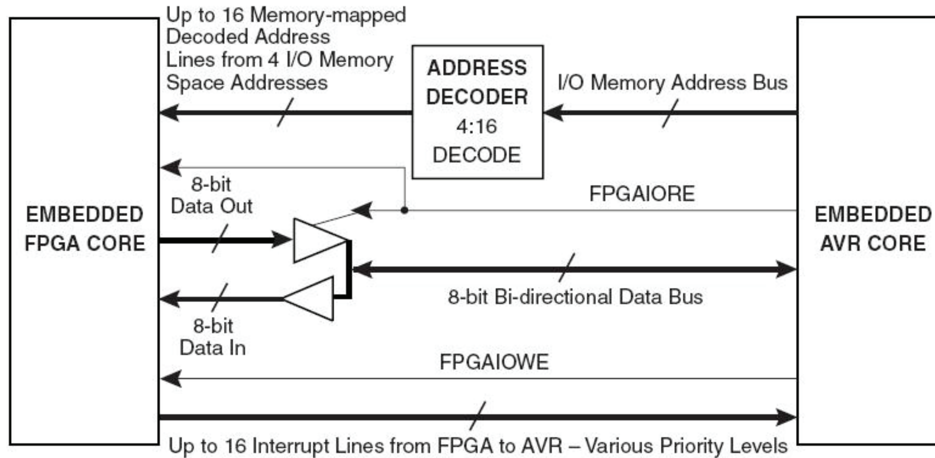


Figur 1 - AT94K Indre struktur

2.1.1 Generelt om oppbygning og muligheter

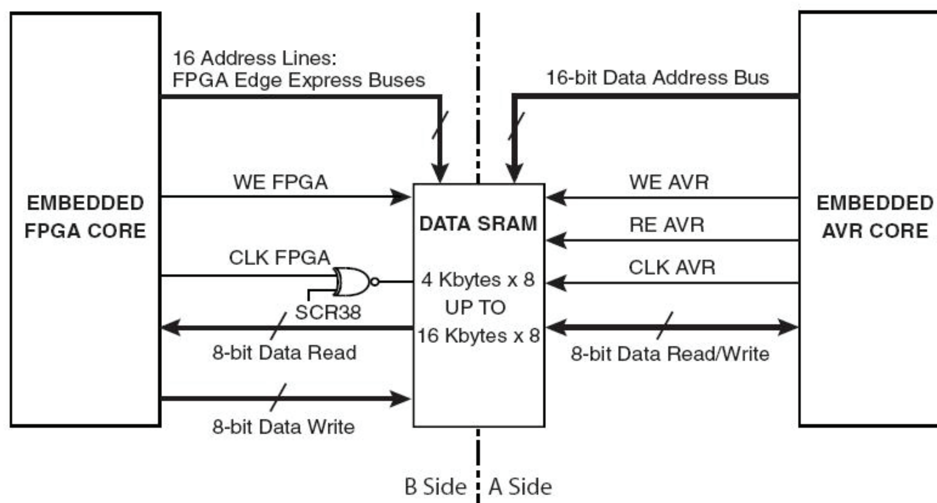
Kort fortalt er Atmel AT94K (FPSLIC) en Atmega103 mikrokontroller, en AT40K FPGA og delt tosidig SRAM i en og samme brikke. Mikrokontrolleren kjører med en klokkefrekvens på 25 MHz, og yter med det 22 MIPS. Dette er tilnærmet det samme som for Atmels enkeltstående mikrokontrollere. En ting som er spesielt med mikrokontrolleren i FPSLIC er at den i tillegg til de vanlige instruksjonene også har inkludert en hardware multiplikator. Videre leveres brikken i versjoner med fra 5k til 40k tilgjengelige logiske gates i FPGA. I tillegg til den delt SRAM, finnes også en toveis databuss mellom AVR og FPGA. Tilgang til bussen gies av mikrokontrolleren ved at 1 av 16 select-pinner settes, i kombinasjon med et signal for lesing eller skriving. FPGA har altså ikke mulighet til å gjøre krav på bussen direkte, men har til gjengjeld 16 interrupts tilgjengelig. Disse kan da benyttes til å be om mikrokontrollerens oppmerksomhet, og eventuelt tilgang til bussen. De mindre utgavene av brikken har imidlertid

et mindre antall av både select-pinner og interrupts. Uavhengig av størrelse, finnes brikken finnes videre i to utgaver: Den vanlige krever eksternt minne for lagring av konfigurasjonsdata til FPGA, mens den andre har dette innebygd i brikken. Sistnevnte omtales som FPSLIC Secure. Tanken med denne er å gjøre det vanskeligere å lese ut designet for uvedkommende. I tillegg øker det integrasjonsmulighetene å ha alt i samme brikke.



Figur 2 - Databus interface

I figur 2 er grensesnittet til den omtalte toveis databussen illustrert. Det er her verdt å legge merke til særlig to ting: Det ene er at adressedekoderen allerede er mappet fra 4 til 16 bit. Det er altså ikke mulig å feilkode på mikrokontrollersiden, slik at flere adresser eventuelt ble kalt samtidig. Det betyr også at eventuelle utvidelser av adresserommet må utføres på i protokollen, ved for eksempel å legge adresseinformasjon i headeren på datapakkene. Den andre tingen som er verdt å merke seg er at FPGA som figuren viser er fysisk avskåret fra å skrive til bussen, så lenge ikke mikrokontrolleren ikke har signalisert at den er klar for å motta data. Dette betyr i praksis at det ikke er mulig for FPGA å forårsake kollisjoner på bussen.



Figur 3 - Shared SRAM interface

Figur 3 viser det tilsvarende grensesnittet til den delte SRAMen. Bruk av denne krever imidlertid konfigurering fra mikrokontrolleren. Siden software ikke har vært et tema for oppgaven, og siden SRAM ikke er benyttet i implementasjonen, så vil ikke denne bli kommentert nærmere. For ytterligere detaljer henvises det til brikkens datablad.

En interessant egenskap ved denne brikken er at den støtter Cash-logic. Det betyr at flere ulike konfigurasjoner kan ligge lagret i FLASH, og at brikken kan rekonfigurere seg selv i kjøretid, til og med uten tap av data. Imidlertid er rekonfigurering en prosess som tar litt tid. Det er derfor nødvendig å vurdere om man for anvendelsen kan akseptere at brikkens funksjoner er utilgjengelige et øyeblikk. En tenkt anvendelse her kan være at man har ulike operasjonsmodi lagret, slik at protesen for eksempel kunne endre modi i forhold til hva type protese som ble koblet til. Samtidig kunne det gi mulighet for å tilby større funksjonalitet enn kretsen i seg selv rommer, såfremt funksjonene forventes å være godt spredt i tid. Dette kunne være ulike modi for håndtering av små finmotoriske bevegelser og store hurtige bevegelser. Eksempelvis skrelling av poteter versus støvsuging. På denne måten kan det altså være mulig å gi protesen vesentlig større funksjonalitet enn den benyttede brikkens størrelse skulle tilsi. Dette vil naturligvis også spare både plass og penger.

Med utgangspunkt i hvordan brikken er bygget opp, med mikrokontrolleren som master for både kommunikasjon og oppsett av brikken, så er det naturlig å anse brikken som en FPGA som en utstrekning av mikrokontrolleren heller enn omvendt. Man kan derfor se på FPSLIC som en AVR med mulighet for hardkoding av spesialtilpasset periferi. Dette kan være støttefunksjoner til mikrokontrollerens indre operasjoner så vel som utvidet I/O-funksjonalitet. Av den grunn er det rimelig å anta at brikken har potensial til å kunne håndtere atskillig tyngre og mer krevende oppgaver enn en tilsvarende ren mikrokontroller. Såfremt ikke produksjonsvolumet er til hinder, så skulle dette kunne gi vesentlig bedre fokusert regnekraft for pengene enn FPGA eller mikrokontroller hver for seg. I denne sammenhengen er det naturlig å dra en parallell til PC-arkitekturen, som bygger på en tredelt hierarkisk strukturering av minne med ulik hurtighet. På den måten har man altså klart å optimalisere ytelse i forhold pris på en meget god måte. Kanskje kunne slik hybrid-teknologi i fremtiden gjøre noe av det samme for tilpassede datasystemer?

2.1.2 Egnethet til anvendelse i NRWD og lignende proteser

Som [Mossum 2005] tar for seg, så er det flere fordeler ved å benytte FPSLIC som hardware-plattform fremfor flere separate mikrokontrollere, slik som i Skjeltens løsning. Den største fordelen for proteser generelt er at brikken ideelt sett skulle kunne redusere antallet aktive komponenter i kretsen. Dessverre har dette, grunnet enkelte funksjonelle krav, vist seg å ikke være tilfelle for NRWD. Her vil antakelig FPSLIC kreve tre aktive brikker istedenfor to. Imidlertid vil dette antallet heller ikke øke, selv om antallet motorer i systemet øker. Dermed er det først ved slike anvendelser at tilnærmingen kommer til sin rett, og de største vinningene kan forventes. Et annet argument for å benytte FPSLIC som plattform er at den i stor grad vil være kompatibel med programvare utviklet for prototypen av NRWD.

Foruten færre antall aktive brikker som mål i seg selv, håpet man også at den økte integreringen skulle resultere i forenklet ruting av kretskortene; og dermed mer kompakt design. Av overnevnte årsaker vil ikke dette være tilfelle for NRWD. Derimot ville man for anvendelser med flere enn to motorer oppnå færre brikker, en for en tilsvarende krets basert på mikrokontrollere. Dermed ville naturligvis også rutingen fortone seg enklere. Det er også rimelig å anta at den økte integreringen kan gi et mer robust design med tanke på støy, siden færre essensielle signallinjer vil bli åpent eksponert på kretskortet. Om støyfølsomhet er en relevant problemstilling for slike proteser er imidlertid uvisst.

Det er liten tvil om at FPSLIC var en banebrytende brikke da den kom i 2001. Imidlertid virker det som lite har skjedd med brikken siden den gang. Spesifikasjonene Atmel oppgir på

brikken nå i 2006 ser nemlig ut til å være de samme som ulike tekniske tidsskrifter presenterte i 2001. Når dokumentasjonen fremdeles er mangelfull, fem år etter lansering, så er det lite som tyder på at brikken er prioritert innen for Atmel sitt produktsortiment. Det man kanskje mest skulle kunne forvente var at brikken i løpet av disse årene ville bli gjort tilgjengelig i mer kompakte pakkeutførelser. Her blir den nemlig kraftig utkonkurrert av størrelsen til Atmels egne mikrokontrollere. Også Xilinx sin Sparten III, som riktig nok er en ren FPGA, slår FPSLIC ned i støvlene når det kommer til fysisk størrelse versus antall gates. Så brikken er nødt for å vise helt andre styrker for å være berettiget en plass i et fremtidig styresystem. At brikken nærmest vil være direkte kompatibel med programvare som skrives for den foreliggende prototypen er definitivt et sterkt argument. At ITK har bred erfaring med bruk av Atmels mikrokontrollere er heller ikke en uvesentlig faktor.

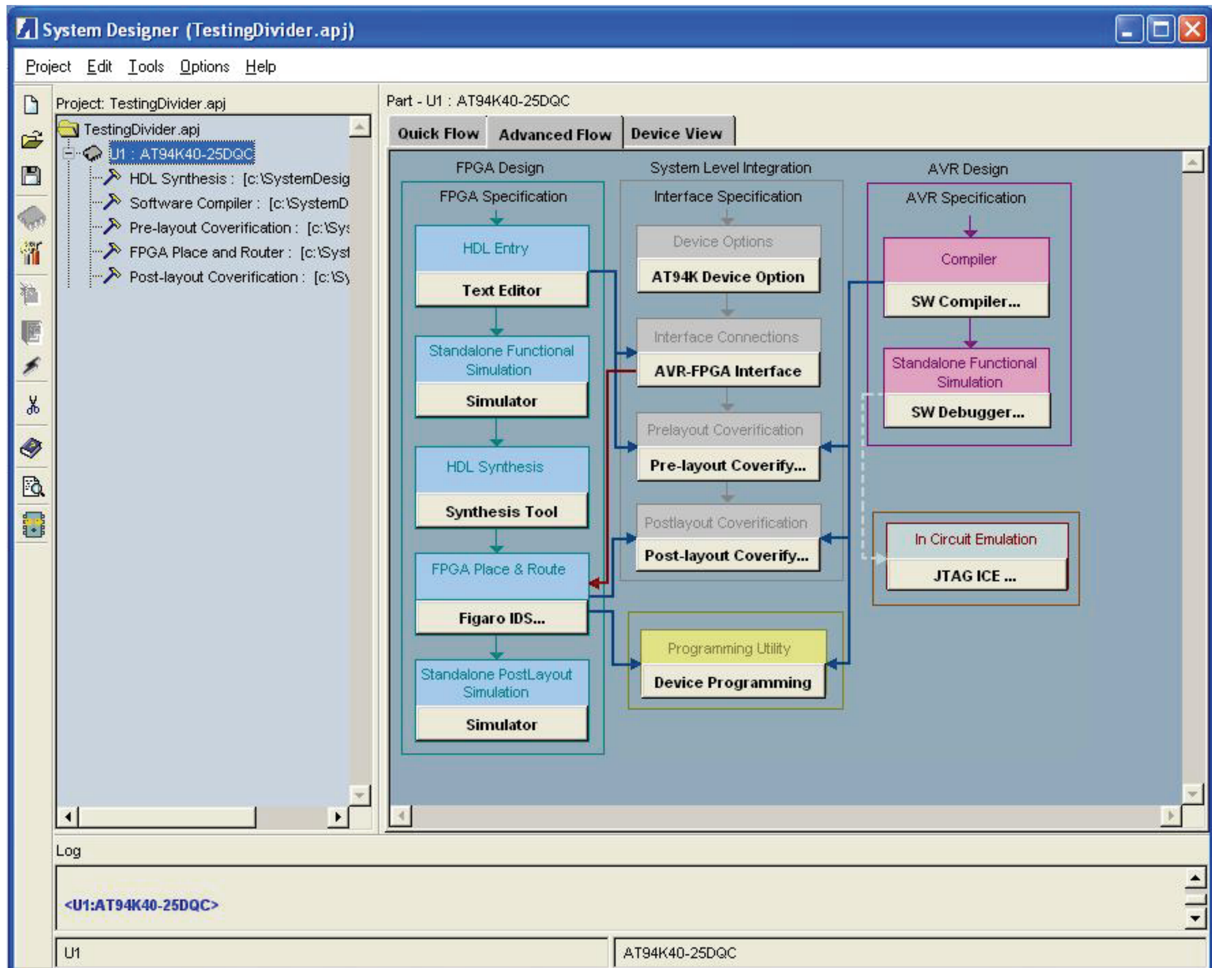
Konseptet med FPSLIC må sies å være genialt. Kombinasjonen av en billig mikrokontroller med en liten FPGA om bord burde kunne ha et enormt potensial innen tilpassede datasystemer. Men dessverre ser det ut som FPSLIC har lidd kraftig under nettopp det at Atmel ikke har valgt å satse så stort på denne teknologien. Det kan særlig se ut til at forskning på miniatyrisering her har uteblitt. Sammenlignet med hva antall gates per areal for eksempel Xilinx kan levere, så er det ingen grunn til at FPSLIC i 40k-utgave skal måle 20x20mm. Et annet problem her i Europa har vært tilgjengelighet. Brikken har vært kjent i mange år; skjønt det ikke har vist seg mulig å få tak i denne før høsten 2005.

Dersom applikasjonen ikke vil lide under kretsens fysiske størrelse, og man har kommet over kneika grunnet mangelfull dokumentasjon, så skal det sies at FPSLIC gir en hel rekke muligheter den er nokså alene om å kunne tilby. Men dette til tross så vil nok hensyn som tilgjengelighet, pris, support og disponibel logikk gjøre at andre alternativ kan vise seg sterkere. Imidlertid har det lyktes å få bekreftet fra Atmel at nye planer for brikken vil bli offentliggjort fjerde kvartal 2006. Kanskje kan ting komme frem da som setter brikken i et nytt lys også for NRWD?

Grunnen til at disse tingene nevnes såpass tidlig i rapporten er fordi det viste seg tidlig i arbeidet at brikken ikke var en så selvsagt kandidat til oppgaven som først antatt. Det vil under arbeidet bli tatt utgangspunkt i at designet skal kjøres på FPSLIC. Men det vil samtidig tilstrebes å gjøre en eventuell overgang til annen brikke så liten som mulig. Altså nok en grunn til å gjøre designet modulært, slik at minst mulig eventuelt må redesignes.

2.2 Atmel System Designer – Kommentarer og erfaringer:

Atmel System Designer er som navnet impliserer en pakke av alle de nødvendige programmer for å utvikle et system basert på AT94K, eller FPSLIC som brikken populært kalles. Pakken tilbyr også et overordnet perspektiv på designprosessen. Videre er pakken en blanding av bidrag fra Atmel, og lisensiert tredjeparts programmer. Deriblant ModelSim og Leonardo Spectrum. Pakken tilbyr imidlertid ingen fullverdig lisens på C-kompilator. Kun demo-versjoner medfølger, utenom den åpent tilgjengelige GCC-kompilatoren. Det er Forøvrig mulig å benytte en vilkårlig C-kompilator sammen med System Designer.



Figur 4 - System Designer - Hovedvindu

Det at pakken har sammenfattet alle nødvendige verktøy under et og samme grensesnitt er en bra ting. Det at de har valgt å basere deler av systemet på kjente verktøy er også en stor fordel. Disse verktøyene er da levert i begrensede versjoner, som kun har støtte for AT94K. Dette kan ansees både som en fordel og ulempe. Fordel fordi det reduserer den totale prisen på DAK-verktøyene. Ulempe fordi verktøyet da kun har nytteverdi for denne ene brikken. Man hadde også misstanke til at det kunne by på problemer å ha fullversjonene av ModelSim og Leonardo installert; noe som dessverre viste seg å være tilfelle.

Måten de rimelige universitetslisensene håndteres på er dessverre ikke den mest fleksible. Om det er mulig å få løsninger basert på en lisensserver er uvisst; men betviles siden pakken tilbyr tredjeparts programmer. I tilfelle ville systemet etter all sannsynlighet ikke være å betrakte som rimelig lengre. Måten lisensen på universitetspakken er håndtert er i alle tilfeller at en

lisensfil genereres av Atmel via et webgrensesnitt. Denne lisensfila er da knyttet til den spesifikke maskinen; noe som gjør samarbeid i grupper og tidsdeling av lisensen vanskelig. Hvis det skulle oppstå behov for bytte av maskinvare kan det også by på problemer.

Omtrent midtveis i arbeidet var undertegnede nødt for å skifte hovedkort på arbeidsmaskinen; noe som førte til at lisensen sluttet å fungere. Det lyktes da heller ikke å få noen bistand fra Atmel til å løse dette problemet. Imidlertid klarte man i samarbeid med den norske utstyrsleverandøren å finne ut at lisensen var knyttet til nettverkskortets MAC-adressen, noe som i dette tilfellet var integrert på hovedkortet. Med andre ord kunne lisensen se ut til å være tapt. Imidlertid klarte man ved hjelp av uoffisielle metoder å skifte maskinens MAC-adresse tilbake til den som lisensen var registrert for. Problemet lot seg altså løse; uten at dette i på noen måte var Atmels fortjeneste, og uten at det er å forvente av den jevne bruker.

Et annet problem som dukket opp var altså at lisensen på System Designer kom i konflikt med fullversjonene av ModelSim og Leonardo, som instituttet har nettlisenser på. Ved installering så det ut til å fungere. Men etter en oppdatering av FPGA Advantage var dette ugjenkallelig slutt. Heller ikke avinstallering og nyinstallering av de aktuelle programmene så ut til å ha noen effekt. Man endte opp med at en maskin nummer to ble satt opp med FPGA Advantage, slik at arbeidet kunne fortsette. System Designer ble dermed offisielt forkastet for dette prosjektet sin del.

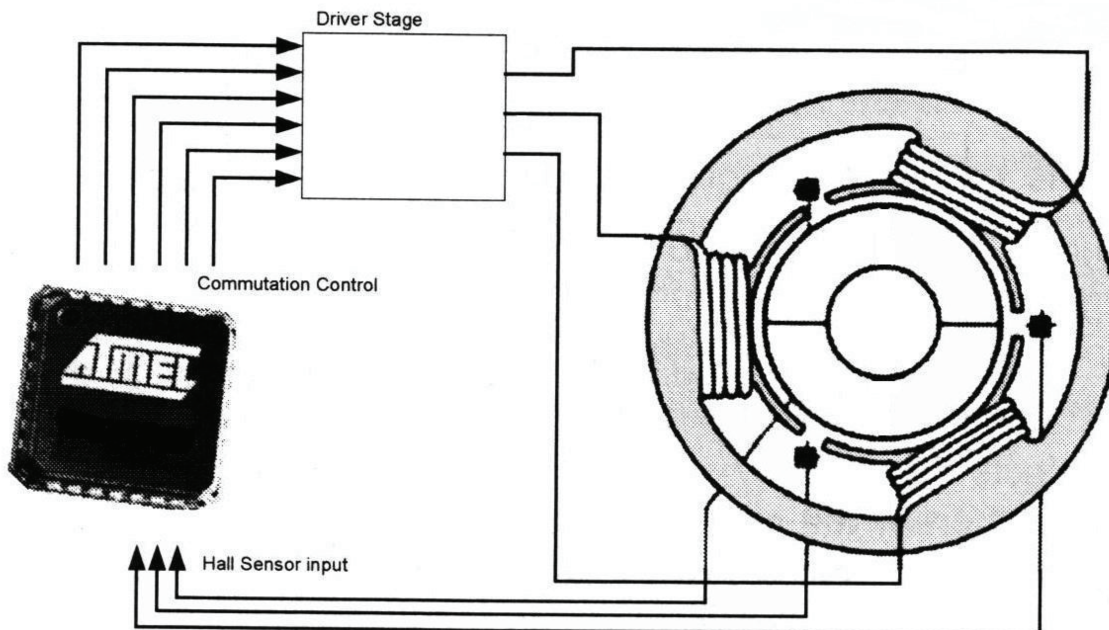
Til tross for en del tekniske problemer med verktøyene, er det utvilsomt dokumentasjonen som er programpakkens svakeste punkt. Det finnes svært få gode tutorials, hvis noen i det hele tatt. Dette er noe mange har beklaget seg over på ulike forum. Den som følger med programmet er så lineær at man i beste fall husker hvilke knapper man trykket på. Du blir guidet igjennom verktøyene for gjennomføring av en fullstendig integreringsprosess på en systematisk og grei måte. Men forklaring av de enkelte programmenes funksjon og virkemåte uteblir. Man benytter stort sett eksempelfiler til alle trinnene i prosessen; og blir ikke på noe tidspunkt fortalt hvordan disse lages. Så med mindre man har utstrakt erfaring med tilsvarende verktøy fra før, vil brukerterskelen her nok oppleves relativt høy for de fleste. Flere av innleggene som var å spore i de ulike forumene man kom over igjennom arbeidet ytret ting som at verktøypakken på mange måter var et godt hjelpemiddel når du først hadde lært deg den; men at brukerterskelen er relativt høy, og læringskurven slak.

For implementering og simulering vil man ikke ha det helt store behovet for System Designer. Dette kan man gjøre like bra i FPGA Advantage, som også har støtte for syntese mot AT94K. Imidlertid har man ikke mulighet for integrering mot mikrokontrolleren, eller oppsett og konfigurering av brikkens ressurser. Til dette er man avhengig av System Designer. Her finnes det også verktøy for verifisering og test av integrerte design; noe man dessverre ikke har fått gjort seg kjent med av overnevnte årsaker. I alle tilfeller hadde man på det nåværende tidspunktet ikke det helt store behovet for slik funksjonalitet, i og med at programvare ikke foreligger, og full integrering ikke er innenfor oppgavens søkelys.

3. Virkemåte og styring av BLDC-motorer

3.1 Motorens oppbygning og virkemåte

Den trefasede børsteløse likestrømsmotoren (BLDC-motor) som benyttes i protesen har to grunnleggende forskjeller fra tradisjonelle DC-motorer. For det første har den tre spolesett som kommuteres individuelt, mot det ene spolesettet en tradisjonell DC-motor har. For det andre har elektromagnetene og fastmagnetene byttet plass i forhold til i DC-motoren. Som følge av dette har man ikke mekaniske børster i slike motorer. Det er disse børstene som i tradisjonelle DC-motorer sørger for kommuteringsskift og overføring av energi til elektromagnetene i motoren.

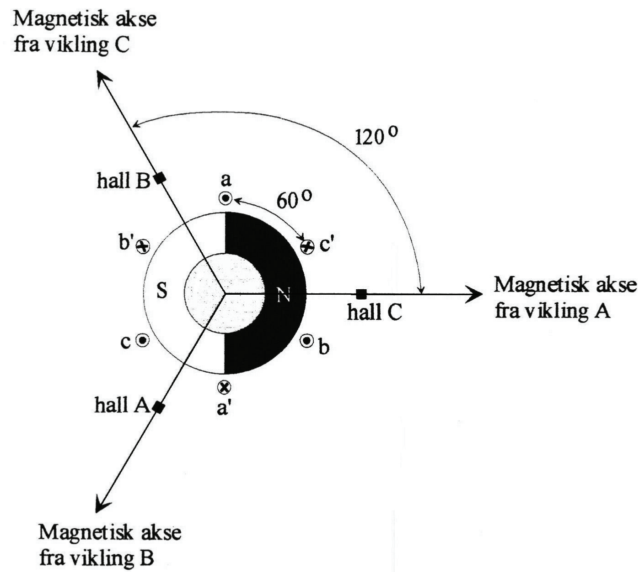


Figur 5 - 3-fase BLDC-motor

Det at BLDC-motoren ikke har de mekaniske kommuteringsbørstene gjør at det eneste fysiske kontaktpunktet i motoren er rotors opplagring og aksling. En fordel med dette er at slike motorer utvikler mindre varme, har mindre mekanisk slitasje, og derfor lengre levetid. Det at motoren kommuteres over tre faser gjør også at slike motorer blir relativt kraftige for størrelsen. Ulempen er at motoren må kommuteres elektronisk. Derfor blir drivkretsen til slike motorer mer komplekse og kostbare enn for tradisjonelle DC-motorer.

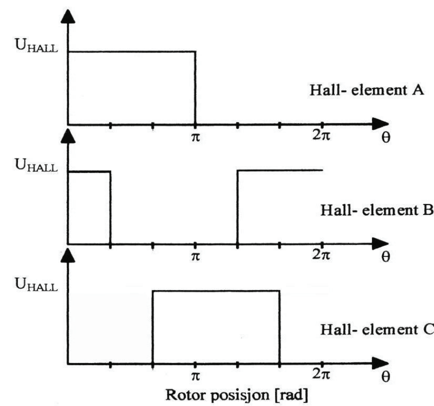
Det finnes flere måter å styre slike motorer på. Det vanligste er at motoren er utstyrt med sensorer som måler rotorens vinkel, slik at kommuteringen kan skje på grunnlag av denne informasjonen. Alternativt kunne den elektromotoriske spenningen fra den passive spolen måles, og brukes til å når rotormagneten passerer denne. Dette er imidlertid bedre egnet for anvendelsesområder uten veldig hurtige hastighetsendringer og uten behov for stabil regulering på lave turtall. Posisjonsregulering er også en svakhet med denne metoden.

Motoren som benyttes i protesen er utstyrt med hall-sensorer, som fungerer på den måten at de måler tilstedeværelse av rotormagnetens nordpol. Tre slike sensorer er plassert med 120° mellomrom. Rotormagneten er sirkulær, som vist i figur 6. Det betyr at én sensor alltid vil være dekket, og at aldri mer enn to sensorer kan være dekket på samme tid.



Figur 6 - Hallsensorer - Prinsippskisse

Når disse sensorene leses av vil informasjonen foreligge som en tre bits vektor, hvor hvert av bitene representerer én sensor. Siden én sensor alltid må være dekket, og aldri mer enn to sensorer kan være dekket, så vil ikke ordene "000" eller "111" forekomme. Signalet ut ifra de enkelte sensorene vil fortone seg som vist på figur 7.



Figur 7 - Signal fra hallsensorene

Av figur 7 ser vi også at kun en sensor av gangen kan endre status. Med utgangspunkt i disse signalene kan rotors vinkel bestemmes med en nøyaktighet på 60°. Tabell 1 viser en oversikt over de seks vinkelsonene, og deres tilhørende vektorer.

Tabell 1 - Hallvektorer

Rotorposisjon [°]	Status 1			Status 2		
	C	B	A	C	B	A
0	0	1	0	0	1	1
60	0	1	1	0	0	1
120	0	0	1	1	0	1
180	1	0	1	1	0	0
240	1	0	0	1	1	0
300	1	1	0	0	1	0

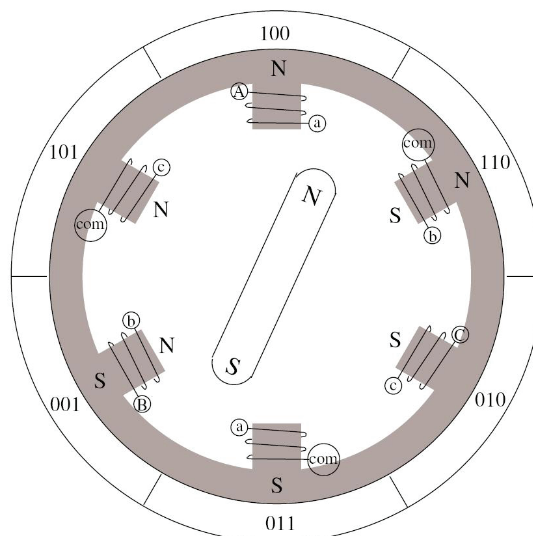
Ved kontinuerlig drift vil transisjonstidspunktet fra en sone til en annen, og dermed rotorvinkelen i øyeblikket, kunne bestemmes med god nøyaktighet. – Dersom det skulle være av interesse å bestemme rotors vinkel med større nøyaktighet enn hva hallsensorene gir informasjon om, kan dette oppnås ved å måle tiden fra en transisjon til neste. Såfremt hastigheten kan ansees som konstant over et visst antall transisjoner, så kan denne tiden brukes til å estimere vinkelen mellom to sensorer. *Henviser til Terje Wesselberg – Digital motorstyring på dette feltet. [1]*

3.2 Prinsipper for styring av BLDC-motorer

For BLDC-motorer uten posisjonssensorer, kan passeringstidspunktet for rotor registreres ved å måle den elektromotoriske spenningen i den passive spolen i motoren under drift. Dette gjøres i praksis ved å trigge på nullgjennomgangen, eller punktet der den elektromotoriske spenningen skifter fortegn. På grunnlag av dette kan kommutering utføres. Men siden dette kun fungerer under drift må andre teknikker benyttes for å finne motorens initialtilstand. For eksempel kan motoren tvinges til lås i en kjent tilstand ved å holde konstant spenning over et spolesett, til motoren antas å være låst til en fase. – Imidlertid er ikke denne teknikken like fleksibel som bruk av sensorer. Man står ikke like fritt til valg av kommuteringsteknikk, og har ikke mulighet til den samme presise styringen ved lave turtall. Men motorene blir rimeligere og mindre uten innebygde sensorer, noe som kan være viktigere i en del applikasjoner.

Det vanligste og enkleste er allikevel at motorene har innebygde sensorer for kontinuerlig avlesning av rotorvinkelen, og at disse benyttes til å simulere mekanisk kommutering. Altså at kommuteringsskiftene skjer på faste tidspunkt i forhold til de observerte transisjonene. Dette gjør at motorens initialtilstand er kjent, og at motoren derfor kan startes kontrollert.

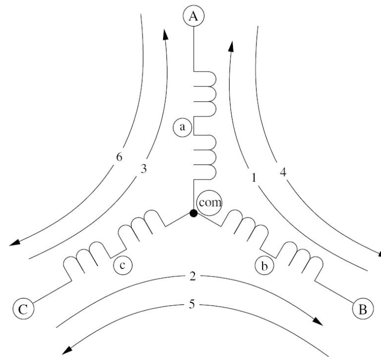
Motorens eksakte oppbygning er dessverre ikke kjent da produsenten ikke har vært villig til å utgi detaljert informasjon om den. Men på grunnlag av studiene utført av Terje Wesselberg [1] antas det at hver spole har en romlig utbredelse på 180°, og at sentrene for de tre spolene har en innbyrdes romlig vinkel på 120°. For logikkens del er det imidlertid tilstrekkelig å forholde seg til motorens ekvivalent, med hver av spolene konsentrert i et punkt. Se figur 8.



Figur 8 - Motorens ekvivalent

Som figuren viser består hver av de tre spolene av to seriekoblede spoler, hver av dem stilt opp på motstående side av stator. Den enden av hver spole som vender mot senter er koblet

sammen med spolen på motsatt side, slik at ekvivalenten av de distribuerte viklingene blir tre spoler. For hvert spolesett er den ene enden tilgjengelig, mens den andre enden er koblet sammen i et felles punkt kalt Common. Se figur 9.



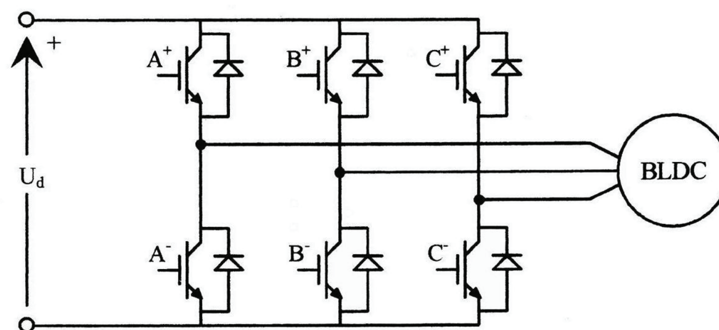
Figur 9 - Motorens interne oppkobling

Kommutering utføres da ved å sende strøm igjennom to av de tre magnetene om gangen. Man kobler da effektivt den ene av spolene til spenning, og den andre til jord. Av figur 9 ovenfor kan man da se hvordan strømmen vil bre seg ved de ulike kommuteringene. Dette gir seks mulige kombinasjoner av spolestrømmer, som vist i figur 10 nedenfor.

Figur 10

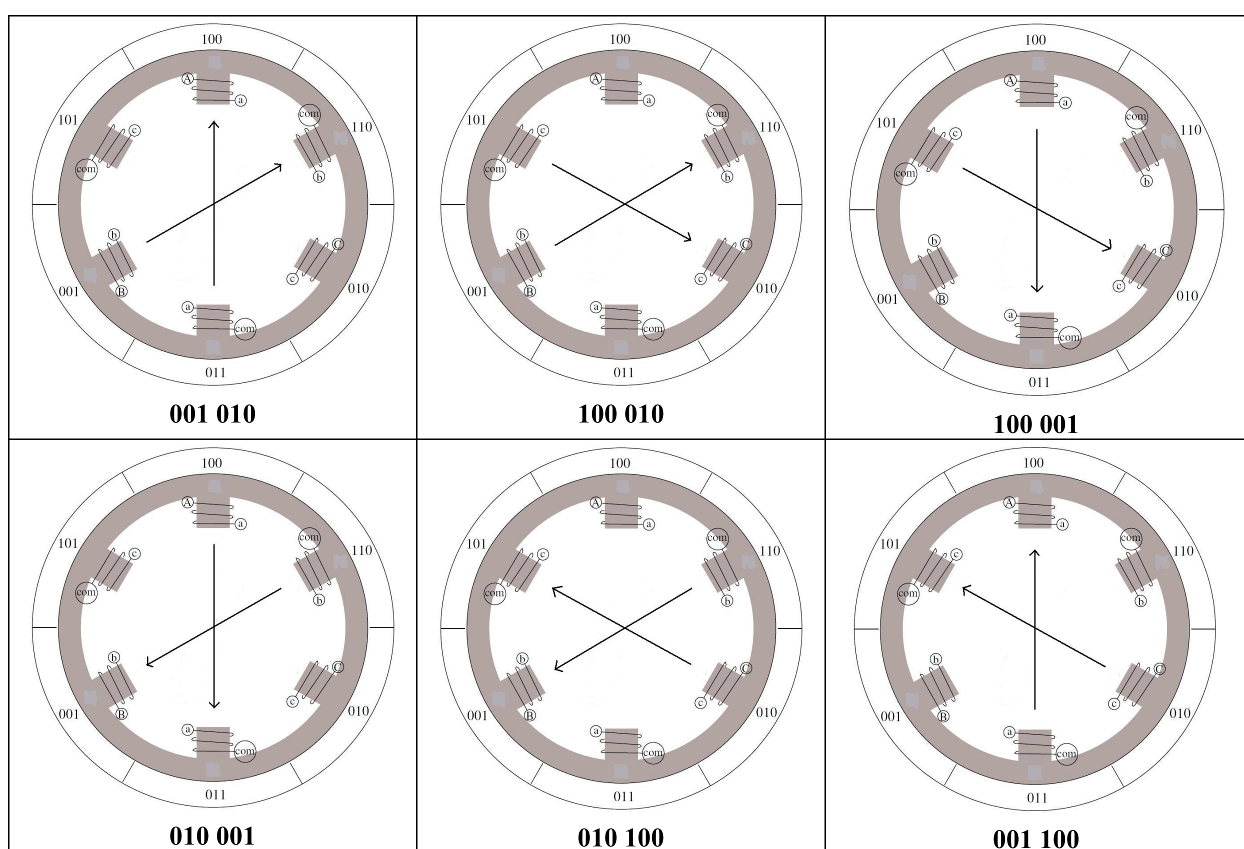
Transistorstrukturen i figur 11 kalles en H-bro, og er vanlig å benytte til styring av motorer. Hver fase er da styrt av to transistorer, som er koblet til henholdsvis spenning og jord. Hver transistor i figuren korresponderer til et bit i kommuteringsvektorene, som er illustrert i fig 8.

Figur 11 - H-broens virkemåte



Kobling av transistorene for å oppnå full 4- kvadrant styring

I figur 12 er de seks ulike feltkonfigurasjonene illustrert. Pilene beskriver kraftvektorene som hver fase vil virke på rotormagneten med. Altså peker vektorene fra N til S. Under hvert bilde står de tilhørende kommuteringsvektorene oppført. De tre laveste bittene her angir hvilke fase som kobles til jord, mens de tre høyeste bittene angir spenning. Altså hører følgende bit sammen til hver sin fase: 1 og 4, 2 og 5 samt 3 og 6.

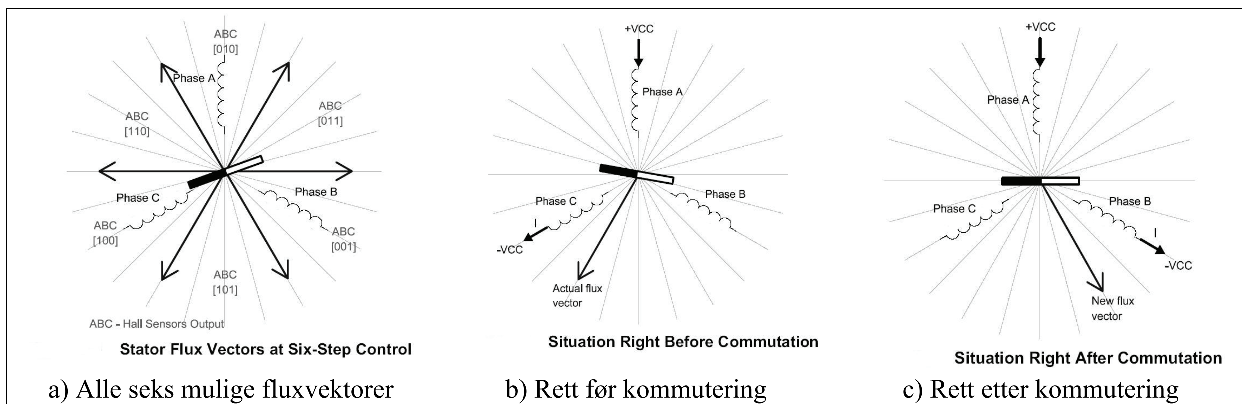


Figur 12 - Mulige feltkonfigurasjoner

3.3 Ulike kommuteringsmetoder for BLDC-motorer

Måten motoren er koblet opp på gir oss som beskrevet i figur 12 seks mulige feltkonfigurasjoner. Motorstyringen består da i å velge hvilke av disse som skal benyttes, og timingen for når kommuteringen skal utføres.

Den enkleste og mest vanlige fremgangsmåten er å simulere mekanisk kommutering. Altså å utføre kommutering på et definert tidspunkt relativt til transisjonstidspunktet. Men også her finnes det flere alternative fremgangsmåter. I hovedsak går det på hvor langt foran rotormagneten man til en hver tid velger å la det magnetiske feltet ligge. Vinkelen mellom magnetens nordpol og feltets sentrum kalles da kommuteringsvinkelen. Med motoren som benyttes i protesen kan denne styres i 60° intervaller. Det vanligste, og sannsynligvis mest hensiktsmessige er å la kommuteringsvinkelen være 90°. Altså hver gang en ny vinkelsone entres, så aktiveres sonene som ligger 60° og 120° foran (effektivt 90°), mens sonen rotor befinner seg i holdes passiv. I figur 13a) er de seks mulige fluxvektorene illustrert. Figur 13b) og 13c) illustrerer situasjonen rett før og rett etter kommutering ved 90° kommuteringsvinkel, og rotasjon mot klokka.



a) Alle seks mulige fluxvektorer b) Rett før kommutering c) Rett etter kommutering

Figur 13 - Kommutering

Et alternativ er å aktivere sonene som ligger henholdsvis 120° og 180° foran entringspunktet. Altså en kommuteringsvinkel på 150°. Vinningen med en slik tilnærming ville sannsynligvis være høyere maksimalt turtall, men på bekostning av motorens dreiemoment. Økt motindusert spenning er også en konsekvens av øket turtall.

I teorien skulle det også være mulig å benytte 30° kommuteringsvinkel. Motoren ville da bli sterkere; men ville antakelig ikke kunne oppnå stabil drift ved høyere turtall. Så fort hastigheten blir høy nok til at spoleforsinkelsen i størrelse blir sammenlignbar med passeringstiden så vil dette snarere bremse motoren enn å akselerere den. Derfor er det antakelig ikke hensiktsmessig å benytte 30° kommuteringsvinkel.

3.3.1 Avansert kommutering ved hjelp av estimert rotorvinkel

Dersom man implementerer beregning av høyere vinkeloppløsning for rotor enn de 60° hallsensorene direkte gir, så er det også mulig å justere kommuteringsvinkelen fritt. Mest aktuelt er det i den sammenheng å la regulatoren bruke kommuteringsvinkelen som et sekundert pådrag. På den måten kan man oppnå en slags trinnløs elektronisk styrt girkasse.

Det som vil begrense hvor høyt det er hensiktsmessig å justerer kommuteringsvinkelen er motindusert spenning i spolene. Det vil komme til et punkt der turtallet på grunn av dette ikke øker nevneverdig med videre økning av kommuteringsvinkelen. Alt over dette vil kun føre til høyere strømtrekk og større varmeutvikling.

En annen teknikk som med fordel kan benyttes ved alle disse kommuteringsteknikkene er å tilpasse kommuteringstidspunktet til transistor- og spoleforsinkelsene. Altså å utføre kommuteringen litt før rotor faktisk har skiftet sone, slik at momentet i motoren utnyttes enda bedre. Dette krever imidlertid nøyaktig estimering av rotors romvinkel, eller registrering av rotors passeringstid over hver sone.

3.3.2 Pådragsstyring av BLDC-motor

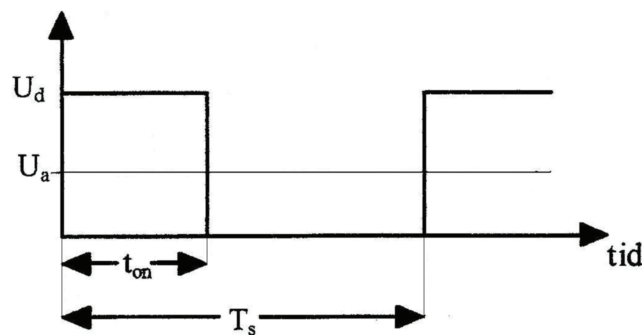
Elektronisk kommutering og eventuelt styring av kommuteringsvinkel er det som skiller en BLDC-motor i bruk fra alminnelige DC-motorer. Foruten påtrykt spenning er kommuteringsvinkelen som bestemmer hva moment og toppfart man oppnår.

Et av prinsippene i en elektromotor er at magnetspolene ved drift vil sette opp en motindusert spenning i viklingene, som øker proporsjonalt med motorens hastighet. Sammen med motorens mekaniske motstand er det i praksis dette som setter øvre grense for turtall. For en

BLDC-motor vil det ved en gitt kommuteringsvinkel og drivspenning si at turtallet øker inntil en av de følgende grensene nådd:

- Den motinduserte spenningen i spoleviklingene vil overgå spenningen fra drivtrinnet.
- Elektronikken er ikke i stand til å utføre kommuteringsskiftene raskere.
- Mekanisk stabilitet eller last begrenser videre økning av turtallet.

Ergo vil motoren med en gitt drivspenning og kommuteringsvinkel søke å stabilisere seg på ét turtall, med likevekt mellom drivspenning og motindusert spenning. I alle tilfeller er det drivspenningen som er den langt viktigste faktoren for motorens ytelse i forhold til kommuteringsvinkel. Derfor er det naturlig å benytte drivspenningen som det primære pådragsorganet. Den vanligste og enkleste måten å gjøre dette på er å benytte puls bredde modulasjon (PWM). Enkelt forklart betyr det at spenningen vekselvis slåes av og på med en frekvens vesentlig høyere enn motorens mekaniske båndbredde. I figur 14 er prinsippet enkelt illustrert.



Figur 14 – PWM

Her er Ud spenning inn, og Ua ekvivalentspenning ut. Ts er her periodetiden for pulstoget, mens ton er tiden spenningen holdes høy. Denne tiden omtales ofte som dutycycle, og er da omregnet til prosent av periodetiden. Ekvivalentspenningen er da gitt av følgende uttrykk:

(0.1)
$$U_a = \frac{t_{on}}{T_s} U_d$$

4. Funksjonelle systemkrav og overordnede strategier

I dette kapittelet vil de viktigste kravene til systemet bli presentert og diskutert. Det som omtales som systemet i denne sammenhengen er de delene av kontrollerkretsen som er tenkt realisert i hardware (FPGA). Fokus for arbeidet har i hovedsak vært å komme frem til en prototyp av disse funksjonene i VHDL, samt å vurdere egnetheten av brikken for bruk i proteser. Funksjonalitet som er tenkt realisert i software, samt nødvendige eksterne komponenter, vil således ikke være noe sentralt tema i denne rapporten. Men siden enkelte av punktene i kravspesifikasjonen kan være litt vanskelig å tilfredsstille direkte med bruk av FPSLIC, vil noen rammede funksjoner knyttet til periferi nevnes kort for sammenhengens skyld. Dette for å gi et litt mer helhetlig bilde av bakgrunnen for de anbefalingene til det videre arbeidet. For videre detaljer om kravene henvises det til kravspesifikasjonen, [Mossum 2005] og [Skjelten 2005].

Første del av kapittelet vil kort ta for seg de krav til totalsystemet som vurderes relevant for arbeidet og / eller vurderingene som skal foretaes. Andre del av kapittelet vil ta for seg de kravene som stilles til kommuteringslogikken direkte. I den sammenheng vil også grunnleggende funksjoner valgt lagt til software bli diskutert kort. Dette fordi disse funksjonene er å anse som en del av grensesnittet kommuteringslogikken har å forholde seg til. Disse punktene kan således sees på som uformelle krav til logikken.

4.1 Relevante krav til totalsystemet

Perspektivet på hva totalsystemet innebærer for dette prosjektet kan oppsummeres i tre hovedpunkt: Motorstyring, sensoravlesning og kommunikasjon. Motorstyring er som plasseringen impliserer det mest sentrale for det praktiske arbeidet. I tillegg nevnes enkelte aspekt innenfor de to punktene sensoravlesning og kommunikasjon, da dette har hatt en viss relevans for evalueringsdelen, samt valg av tekniske strategier videre.

Partisjonering - Motorstyring og regulering:
Det mest grunnleggende kravet til systemet er at det skal kunne styre en børsteløs motor (helst flere). Funksjoner relatert til styringen skal deles mest mulig hensiktsmessig mellom AVR og FPGA. I [Mossum 2005] foreslåes en partisjonering der hele motorstyringen og overvåkningen foregår i FPGA, mens regulatoren kjøres i mikrokontrolleren. Dette innebærer at mikrokontrolleren sender et pådrag til motorkontrolleren i FPGA, og får inn motorens hastighet som tilbakekobling. Det er dermed et krav at motorenes hastighet skal beregnes og sendes mikrokontrolleren i kjøretid, slik at de kan inngå i reguleringssløyfen. Nødvendig presisjon på målingene vil bli diskutert i forbindelse med implementeringen.

Måling / beregning av motorhastighet er altså en funksjon under sensoravlesning som skal utføres i FPGA. Dette er også den eneste målingen som foreløpig er tenkt utført her. Teoretisk sett kunne også den tiltenkte målingen av global vinkel i AVR vært benyttet til estimering av hastighet. Men man har altså valgt å opprettholde dette som krav til logikken allikevel. Andre målinger og beregninger som vurderes hensiktsmessig å implementere i FPGA vil bli diskutert i kapittel om mulige utvidelser.

Måling av global vinkel / posisjon:
Totalsystemet skal altså kunne måle protesens absolutte vinkel. Det er ønskelig at dette skal kunne utføres uten noen form for kalibrering, og at posisjonen skal være kjent så fort strømmen slåes på. Dette blant annet fordi det er ønskelig å kunne spare strøm ved å slå av mest mulig av kretsen når motorene står stille. I den eksisterende prototypen er en sensor med pwm-utgang og serielt grensesnitt tatt benyttet. Denne leses her av via pwm-utgangen og en interrupt-pinne på mikrokontrolleren. Såfremt en slik pinne er ledig, så er dette en relativt lite ressurskrevende prosess for mikrokontrolleren. Derfor er det heller ikke hensiktsmessig å implementere avlesning av denne i FPGA, så lenge reguleringen utføres i software. Men dersom regulatoren blir overflyttet til FPGA, så vil saken stille seg annerledes på dette punktet. Særlig hvis implementering av posisjonsregulering blir aktuelt. I følge veileder er også dette den formen for regulering som først vil bli aktuell å flytte over, siden denne er relativt krevende å utføre i mikrokontrolleren sammenlignet med hastighetsregulering.

Analog sensoravlesning: Måling av motorstrøm:
I kravspesifikasjonen heter det at systemet skal kunne måle og begrense motorstrømmen. Som [Mossum 2005] påpeker ble dette forsøkt i Skjeltens design, uten at tilfredsstillende resultater ble oppnådd. Skjelten konkluderer med at målemetoden sannsynligvis er uegnet for bruk i reguleringssløyfen, og at andre metoder eventuelt burde vurderes. Dette, i kombinasjon med begrenset plass, gjorde at strømmåling ble utelatt fra prototypen av NRWD. Om denne typen måling fremdeles er å anse som et stående krav oppfattes derfor som uklart. I alle tilfeller antas det mest hensiktsmessig at denne målingen utføres i software, så lenge det er her regulatoren ligger. Det vil derfor ikke bli tatt høyde for noen eventuell strømmåling ved implementering av kommuteringslogikken.

Et annet krav til sensoravlesning er analoge innganger for EMG-sensorer. – Til orientering er dette sensorer som måler utstrålt felt fra muskelaktivitet. Slike sensorer monteres utenpå huden til protesebrukeren, i kontakt med friske muskler, slik at brukeren kan bevege protesen. – Dette krever ADC, noe foreliggende utgave av FPSLIC ikke tilbyr. Derfor kreves ekstern ADC. Dette er egentlig utenfor oppgavens søkelys, men nevnes siden dette vil bli tatt med i anbefalingene for det videre arbeidet.

Kommunikasjon:
Kravspesifikasjonen for NRWD sier at "kontrolleren skal kommunisere med eksterne enheter via et passende digital grensesnitt". Det er klart at CAN-bus er grensenittet som vil bli brukt her; både fordi det er meget robust, og fordi det virker sannsynlig at dette vil bli standard for kybernetiske proteser i fremtiden. Som kjent fra [Mossum 2005] har FPSLIC ikke CAN-bus tilgjengelig. En mulig løsning her er å benytte en ekstern CAN-kontroller. En annen mulighet er å implementere en slik kontroller i FPGA. Sistnevnte alternativ vil bli diskutert senere i denne rapporten. Ellers nevnes også UART og TWI / I2C, samt analoge utganger (pwm) for styring av industristandard proteser. De to førstnevnte standardene er støttet av AVR-delen i FPSLIC. Analoge utganger kan realiseres ved hjelp av tilgjengelige pwm-utganger; også disse på AVR-siden av FPSLIC. Dersom det er behov for SPI, kan dette implementeres i FPGA. Sannsynligvis vil det være mulig å finne IP-moduler for dette på nett.

I alle tilfeller vil man ta sikte på at all høynivå kommunikasjon med eksterne enheter håndteres i mikrokontrolleren. Også dersom CAN-kontroller eller SPI-kontroller vil bli implementert i FPGA. Tanken er at alt som har med egenspesifiserte protokoller å gjøre er mest fleksibelt å implementere i software. Så vil i tilfelle FPGA kun ta seg av de laveste og standardiserte nivåene av kommunikasjonsprotokollene, dersom CAN eller SPI blir implementert i VHDL. Sånn sett vil mikrokontrolleren, så snart en driver er skrevet, ikke se forskjell på de innebygde standardene og eventuelle standarder implementert i FPGA. Hvis integrering av disse blir gjennomført, så er disse tenkt plassert separat fra motorstyringen. Det eneste de nødvendigvis må dele er kommunikasjonsmodulen, og databussen. – Med disse perspektivene som utgangspunkt ansees høynivå kommunikasjon for å være utenfor oppgavens søkelys.

4.2 Funksjonelle krav til hardwaren (FPGA)

Dette kapittelet begynner å ta for seg litt mer av de detaljene som angår prosjektets praktiske arbeid. Det er her tatt utgangspunkt i [Mossum 2005]. Av den grunn er det naturlig å starte med de grunnleggende funksjonene, og bygge på med de høyere nivåene etter hvert. Noen av kravene som ble utredet i denne rapporten er utenfor søkelyset til denne rapporten. Men de som er av interesse vil bli belyst nærmere her. Videre vil noen momenter vedrørende FPSLIC bli nevnt, siden arbeidet tar utgangspunkt i denne.

Krav til kommuteringen:
I [Mossum 2005] kapittel 2.1.1 blir de grunnleggende kravene til kommuteringshastighet diskutert. Her blir det også påpekt at eksakte krav er vanskelig å sette opp, uten mer detaljert kunnskap om motorens egenskaper. Et annet relevant moment i den sammenheng er at det for undertegnede ikke er kjent hva slags drivtrinn som er benyttet i foreliggende prototyp, og om dette vil bli brukt for fremtiden. I alle tilfeller vil kjennskap til drivtrinnets maksimale svitsjefrekvens være nyttig.

Man vet ut ifra motorens datablad at maksimal rotasjonshastighet er ca 100.000 RPM. Med seks kommuteringssoner gir dette følgende:

$$\frac{100.000[r/m]}{60[s/m]}\times 6[com/r] = 10.000[com/s] \qquad (0.2)$$

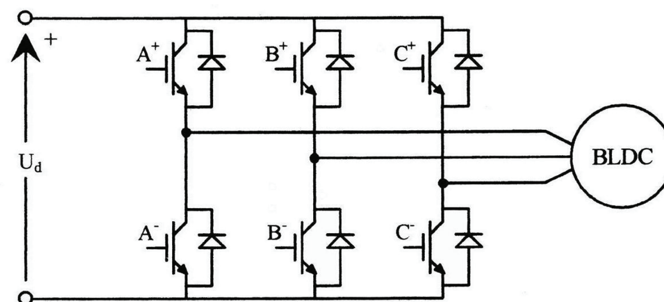
Hvor mye høyere kommuteringsfrekvens som er nødvendig er vanskelig å si noe om. Derfor har en isteden formulert et løst krav i samarbeid med veileder om at logikken må være i stand til å utføre kommuteringene på minst 10 ganger høyere frekvens enn sensorskiftene vil forekomme. Altså med en frekvens på minst 100 kHz. Ved 25 MHz klokkefrekvens, som er et relativt lite ambisiøst anslag, gir dette en faktor på 250 i forskjell. En kommuteringsprosess kan altså ikke ta lengre tid å utføre enn 250 klokkesykler, såfremt klokkefrekvensen for kommuteringen er 25 MHz. Ut over dette kravet så er høyest mulig frekvens ønskelig, for å utnytte motorens moment best mulig. 250 klokkesykler ansees som et romslig krav, og vil derfor ikke bli utredet nærmere her. Kravspesifikasjonen for NRWD fremsetter forøvrig ikke noe strengere krav enn 10.800 RPM (1,4 rad/s ved 1:800 utveksling). De utvidete kravene er satt fordi en mener å ha stor nok sleggefaktor til at heller ikke dette vil være noe problem å oppnå. Antakelig vil det i ettertid bli mer relevant å skru klokkefrekvensen ned igjen, for å begrense strømtrekket i kretsen. Men dette overlates til eventuelt videre arbeid.

En mer relevant problemstilling er for hvilke hastighetsområde det er ønskelig å kunne hastighetsregulere motoren. Her har en i samarbeid med prosjektleder kommet frem til at området 1 til 30 RPM ved håndleddet er det største området en har bruk for å regulere over. Dette gir 800 – 24.000 RPM i motoren. Dersom hastigheten er over 24.000 RPM er det altså ikke nødvendig å ha nøyaktige hastighetsmålinger. At hastigheten er høyere enn 24.000 RPM er da tilstrekkelig informasjon.

Når det gjelder pwm-modulen, så kreves det av denne at svitjsefrekvensen er høyere enn 20kHz, slik at hørbar akustisk støy fra motoren kan unngåes. Ut over dette har man bestemt at input til motoren skal være en byte langt, hvorav 7 bit utgjør ønsket duty-cycle. Det siste bittet angir ønsket kjøreretning for motoren. Dette formatet er valgt for å minimere datatrafikken mellom AVR og FPGA, i tillegg til at det antas tilstrekkelig å kunne regulere pådraget over 127 nivåer.

Verning av drivtrinnet (h-broen):
I 1995 utførte Jan Gunnar Dyrset og Per Øivind Eger et prosjekt ved tittelen ”Digital krets for motorstyring”. De var her de første under veiledning av Øyvind Stavdahl som påpekte at drivtrinnet må vernes mot for hurtige skift på inngangene. Også Håkon Skjelten fikk under sitt arbeid med den mikrokontrollerbaserte løsningen i 2005 erfare akkurat dette. I figur 15 vises drivtrinnets interne koblinger, og koblingen til motoren. Tenkt virkemåte her er som kjent at dersom kurs A benyttes, skal kun den ene av transistorene være åpen. Hvis man velger å åpne A^+, så kan B^- eller C^- åpnes. Hvis begge transistorer på samme kurs åpnes, så gir dette en kortslutning direkte til jord.

Figur 15



Kobling av transistorene for å oppnå full 4- kvadrant styring

Problemet kommer av det at transistorene i drivtrinnet bruker litt tid på å skifte mellom lukket og åpen tilstand. Dersom kommuteringsskiftene utføres for raskt vil dette føre til at flere transistorer enn tilsiktet er aktive, og at situasjonen beskrevet ovenfor derfor kan inntreffe i kortere tidsintervaller. Det man oppnår er altså en periodisk kortslutning av drivtrinnet, der hele eller deler av strømmen ikke går via motoren som ønsket. Dette vil i milde tilfeller representere et energitap, og ekstra varme i drivtrinnet. I verste fall vil drivtrinnet ødelegges. Problemet kan oppstå både ved for raske kommuteringsskift, og ved direkte endring av kjøreretning. Dersom skiftene aldri skjer raskere en transistorenes åpne / lukke- forsinkelse, så vil ikke problemet oppstå.

Det er altså nødvendig at motorstyringen tar høyde for drivtrinnets egenskaper, og innfører en forsinkelse av kommuteringsskiftene. Både for normal drift og ved endring av kjøreretning. Om disse tilfellene skiller seg fra hverandre på noen måte må undersøkes nøyere. For å gjøre systemet mest mulig fleksibelt for bruk sammen med ulike drivtrinn, samt for å kunne ta høyde for at systemet klokkefrekvens ikke er bestemt ved implementering, så bør denne

funksjonen implementeres på en måte som gjør endring / tilpasning enkelt. Det er Forøvrig tilstrekkelig at endringer kan utføres ved kompilering / syntese av systemet.

Dead-lock-guard:
I systemets første versjon skal kun motorstyringen foregå i FPGA, mens reguleringen skjer i mikrokontrolleren. Mikrokontrolleren sender da som nevnt et pådrag til styringslogikken, hver gang det er hensiktsmessig å endre pådraget. I mellomtiden drives motoren med utgangspunkt i det pådraget som sist ble sendt. Dette skaper et potensielt problem dersom mikrokontrolleren av en eller annen grunn slutter å sende data regelmessig. Konsekvensen av dette, hvis problemet ikke håndteres, vil bli at motoren forsetter sin operasjon i den tilstanden den ble forlatt. Altså kjører til batteriet går tomt, eller mikrokontrolleren vender tilbake. I NRWD ville ikke dette bety mekaniske ødeleggelser, siden protesen fritt kan rotere over 360 grader. Men det bør allikevel ikke kunne skje. En mekanisme for å beskytte mot dette er altså nødvendig. – Den enkleste måten å løse dette på er nok å benytte seg av mikrokontrollerens innebygde watch-dog-timer. Denne fungerer slik at en teller må nullstilles regelmessig for at "vakt-bikja" ikke skal vekkes. Hvis ikke telleren nullstilles i tide, så kan for eksempel en global reset initieres. Gjerne også reset av FPGA. På den måten vil mikrokontrolleren bringes ut av tilstanden der den hadde hengt seg, og tilbake til initialtilstanden. Det samme med FPGA. Sannsynligvis vil dette kunne utføres så raskt at protesebrukeren ikke ville merke det, såfremt det ikke skjer ofte. Data som eventuelt går tapt mens reset utføres ville vært tapt uansett, siden mikrokontrolleren hadde sluttet å svare. Det antas at dette er en tilstrekkelig løsning, og at det ikke er nødvendig å implementere noen egen watch-dog-timer i FPGA.

Grensesnitt mellom AVR og FPGA:
Som beskrevet i kapittelet om FPSLIC, så er det AVR som er master for trafikk på databussen. Databladet forteller at styresignalene så vel som dataene på bussen kun skal foreligge i en klokkeperiode á 40ns (25 MHz). Dette er noe FPGA-grensesnittet også må oppfylle, slik at det ikke kan oppstå konflikter på bussen. Kommunikasjonsenheten må altså designes slik at både tolking av kall og utveksling av data utføres innenfor en klokkesykkel. Det betyr at alle data som skal sendes ut fra FPGA må være bufret opp på forhånd. Dersom svaret på et kall fra AVR først foreligger på et senere tidspunkt, så må et nytt kall sendes for å kunne motta svaret. Enten på et kjent tidspunkt, eller at FPGA melder fra at svaret er klart (interrupt). For alle typer overføringer vil mikrokontrolleren måtte sende separate kall for hver byte som skal overføres, uavhengig av overføringsretning.(1) Det er derfor nødvendig med interne pakketellere i FPGA, slik at den vet hvor i overføringen man er. Det må også kunne håndteres at neste kall fra mikrokontrolleren ikke foreligger umiddelbart etter det foregående, eller at andre kall kan inntreffe før overføringen fullføres. Dette fordi mikrokontrolleren kan ha fått interrupt eller lignende å håndtere i mellomtiden som må håndteres først. Det forutsettes at mikrokontrolleren aldri forkaster eller glemmer en påbegynt overføring, siden dette ville føre til feil i senere overføringer.

Siden det er AVR som er master over bussen, kan ikke FPGA sende data på eget initiativ. Da må eventuelt de 16 interrupt-linjene benyttes, slik at AVR kan varsles om at ferske data foreligger. Deretter kan AVR velge å lese ut data, eller den kan la være. Det er også en mulighet at AVR kan lese ut data periodisk, uten signalement fra FPGA. Men siden behovet for overføring av måledata vil variere stort med motorenes hastighet, er det antakelig å foretrekke at en løsning med interrupts benyttes. På den måten vil man spare CPU-tid ved lave hastigheter, og samtidig opprettholde en optimal tilbakekobling til regulatorene ved høye hastigheter. Dette er altså noen betraktninger som bør taes med ved implementering av logikken. Det kan også være relevant å innføre et støyfilter, eller en terskelverdi for minste

endring i måledata som kreves før AVR varsles. På den måten unngår man for eksempel å sende gjentatte interrupts for en motor hvis hastighet drifter mellom to verdier på grunn av måleunøyaktighet.

Brikkens delte RAM kunne også vært benyttet på ganske samme måte som databussen. Gjerne i kombinasjon med interrupts. Men det antas at overføring over bussen vil være en vesentlig raskere måte å utveksle data på. Så siden brorparten av trafikken vil være hastighetsdata til bruk i reguleringssløyfen, vil raskeste løsning være å foretrekke. I vertfall så lenge bussen har tilstrekkelig kapasitet til all trafikken; noe som ikke forventes å bli noe problem med de beskjedne datamengdene hver måling representerer.

5. Systemdesign

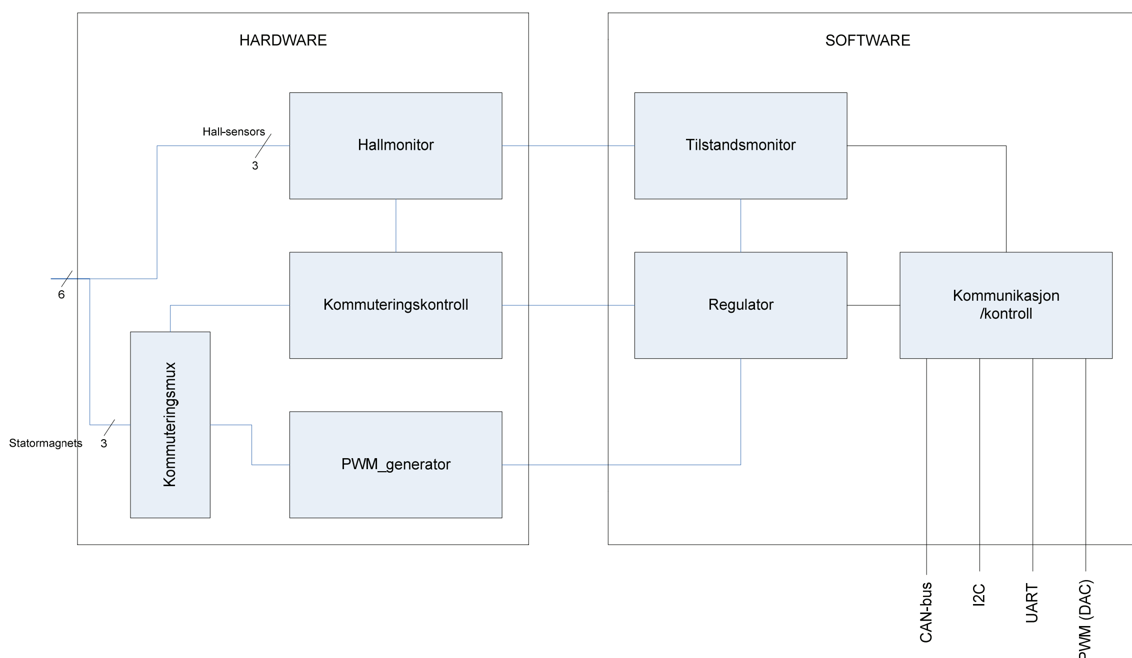
Oppgavetekstens punkt 2 ber om en utredning av mulighetene for samtidig styring av flere uavhengige motorer, samt et skalerbart design med samme perspektiv. Når man ser på virkemåten for en FPGA, så er det ideelt sett ingen ting som skulle være til hinder for dette. Det eneste som kunne sette en stopper er begrenset plass eller I/O. En utredning ville da i stor grad dreid seg om sammenligning av areal på grunnlag av syntetiserte moduler.

Det ble da vurdert sånn at siden designet i alle tilfeller skulle gjøres mest mulig skalerbart, så ville det antakelig være like greit å ta sikte på et system for flere motorer fra første stund. Til dels fordi dette vil gi et mer korrekt bilde av arealbehovet enn å multiplisere opp tall fra synteseresultatene. Men aller mest fordi man da samtidig vil få testet om systemets støttefunksjoner fungerer korrekt for flere motorkontrollere. Dette vil sannsynligvis også være arbeidsbesparende fremfor å først teste systemet for en motor, for så å skalere det opp til et flere motorer etterpå.

I samarbeid med veileder kom man frem til at det meste man kunne se for seg å ha bruk for i en protese ville være fem uavhengige motorer; og fem motorer ble derfor målet man satte for arbeidet.

5.1 Strukturelt design – Funksjonell partisjonering

I [Mossum 2005] ble en funksjonell partisjonering for systemet foreslått. Det foreslåtte systemet er vist i figur 16. Her ble de mest grunnleggende funksjonene for motorstyring tatt med, og gruppert med det perspektiv at kun en motor skulle styres (NRWD). I ettertid har en som kjent tatt sikte på å implementere et styringssystem for fem motorer, med mulighet for videre skalering. Grunnstrukturen i dette designet kan fremdeles brukes; skjønt noen justeringer er nødvendig. Først og fremst for å muliggjøre flere styringsmoduler med et felles grensesnitt til AVR. Men også for å tilfredsstille kravet til skalerbarhet. I tillegg kreves noen strukturelle endringer i forbindelse med utvidet funksjonalitet.

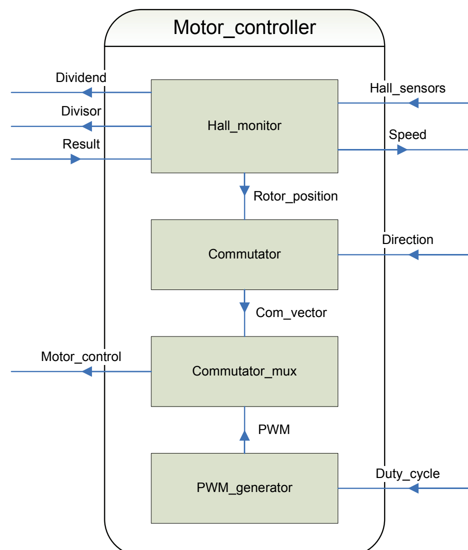


Figur 16 - Foreslått design

5.1.1 Moduler i hardware

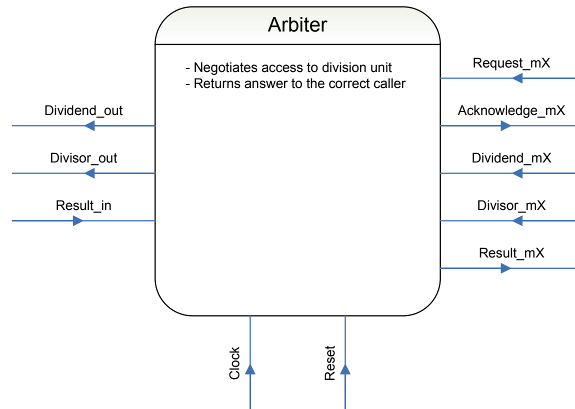
Motorkontroller som samlemodul:
Den kanskje viktigste grunnen til de forandringene man har gjort fra det foreslåtte designet er gjenbruk og skalerbarhet, og da med tanke på styring flere uavhengige motorer. Overordnet struktur er beholdt, mens enkelte av funksjonene er blitt flyttet på, og grensesnittene har blitt utvidet. Tanken har vært at strukturen fra [Mossum 2005] skulle danne grunnlaget for en selvberget samlemodul, inneholdende alt som relaterer seg til grunnleggende motorstyring. Dette perspektivet innebærer at kun elementer som gjentar seg for alle motorer kan inkluderes. Dette for at ingen interne tilpasninger av koden skal være nødvendig ved skalering. Modulen skal kunne instansieres i ønsket antall, uten behov for kjennskap til dens indre struktur. Samlemodulen har altså fått navnet ”motor_controller”, og er avbildet i figur 17. Her er samtidig modulens indre struktur beskrevet.

Den viktigste endringen som er gjort er at man har definert et selvstendig grensesnitt til databussen og mikrokontrolleren, slik at motorkontrollerne ikke trenger tenke på protokoll, og hvilken adresse de selv måtte ha. Samtidig slipper motorkontrollerne å forholde seg til de plattformspesifikke egenskapene ved databussen; slik at modulen ikke behøver endres ved eventuelt skifte av plattform. Videre har man også definert seg et selvstendig grensesnitt, kalt ”arbiter”, til den nylig innførte divisjonsmodulen; slik at motorkontrolleren ikke behøver spesifikke tilpasninger av verken grensesnitt eller timing. *Detaljene her vil bli presentert senere i kapittelet.*



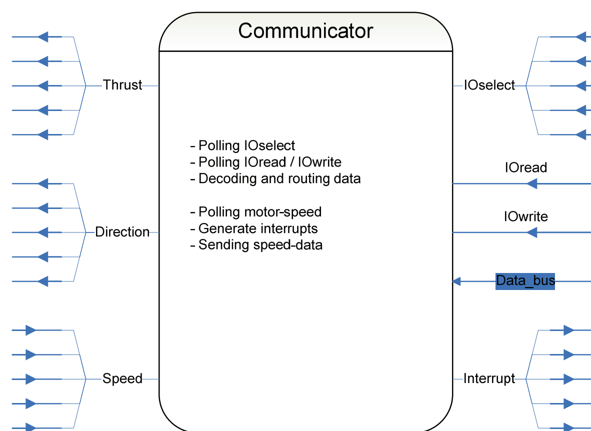
Figur 17 - Motor_controller - Interface and structure

Divisjon og ressursdeling:
Som nevnt i utredningen av funksjonelle krav har besluttet å implementere hastighetsestimering for motorene i hardware. *Dette var en funksjon det ikke ble tatt stilling til i [Mossum 2005]*. Funksjonen er valgt lagt til modulen ”hall_monitor”, og vil således ikke gjøre endringer av den overordnede strukturen nødvendig. Men med måten man har valgt å estimere hastigheten så er divisjon påkrevd. Dette er en krevende prosess, som vil gjøre beslag på betydelig areal i FPGA. Derfor har en tatt sikte på at kun en slik modul inkluderes i designet, og at motorene må dele denne. Implementering av en form for ressursdeling er derfor nødvendig. *I figuren nedenfor vises ressursdelingsmodulens eksterne grensesnitt, som har fått navnet ”arbiter”.*



Figur 18 - Arbiter

Kommunikasjon med AVR:
Det primære grensesnittet mellom FPGA og AVR er som kjent en delt databuss med tilhørende kontrollsignaler. Naturlig nok kan databussen kun forholde seg til én enhet i styringslogikken. Denne abstraherer da vekk håndtering av grensesnittet til databussen fra motorkontrollerne, og opptrer sånn sett som en ruter for datatrafikk. Dette er i med på å gjøre systemet mer skalerbart, og minske mengden redundant funksjonalitet. I figur 19 illustreres modulens grensesnitt.



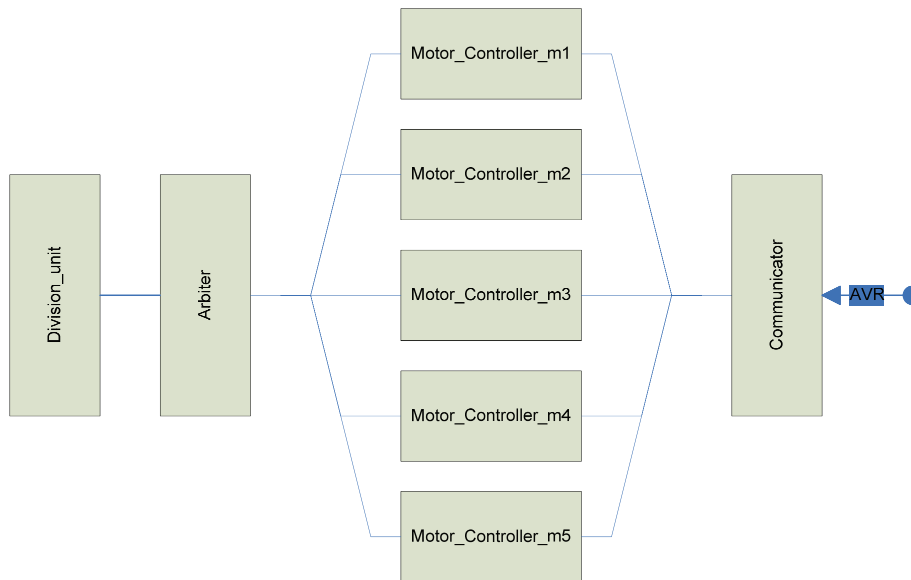
Figur 19 - Communicator interface

Denne funksjonaliteten kunne også vært realisert som en intern buss, med individuell polling av IOselect, og interne tristate-buffere i hver motorkontroller. Men dette ville sannsynligvis bli atskillig mer komplekst, og gjøre beslag på større areal enn en dedikert kommunikasjonsmodul. I vertfall for et såpass moderat antall brukere som fem enheter.

Brikkens delte minne inngår også som en del av grensesnittet mellom FPGA og AVR. Men siden man ikke har valgt å benytte denne, så er heller ingen spesifikk struktur definert for eventuell håndtering av denne. Hvis denne skulle vært tatt i bruk, burde man av hensyn til modulariteten definert en egen kontrollenhet for tilgang til dette minnet. En arbiter med tristate-buffere, slik som i grensesnittet til divisjonsenheten, kunne vært en mulighet.

Det samlede systemet:
Det totale systemet for motorstyring av flere motorer vil da være sammensatt av de tremodulene beskrevet ovenfor, pluss en divisjonsenhet som vil bli beskrevet senere. Motorkontrolleren foreligger i fem instanser for individuell styring av fem motorer; og

tilstøtende logikk er tilpasset dette antallet. Strukturen i systemet blir da som beskrevet i figur 20 nedenfor.



Figur 20 - Top entity

5.1.2 Software:

Selv om software er utenfor søkelyset til denne oppgaven, så er det allikevel hensiktsmessig å sette opp en struktur over de oppgavene som skal utføres i mikrokontrolleren, og som angår motorstyringen. Med tanke på relevans for dette prosjektets arbeid er disse funksjonene blitt gruppert i fire kategorier: Regulator, målinger, applikasjon og kommunikasjon / I/O. Applikasjonen, som får protesen til å kunne utføre noe nyttig, er utenfor oppgavens søkelys, og vil ikke bli kommentert nærmere.

Regulatoren angår hardwaren i den forstand at man er nødt for å definere en slags protokoll mellom denne og motorstyringen. Også det at regulatoren på sikt kan tenkes å bli flyttet til hardware gjør denne hensiktsmessig å definere som egen modul. I tillegg ansees kommunikasjonen regulatoren i software som det direkte grensesnittet til motorkontrolleren i hardware, da det foreløpig bare er denne som utveksler data med FPGA. Dersom annen selvstendig tilleggsfunksjonalitet blir implementert i FPGA, så bør antakelig grensesnittet til bussen gjøres generisk også på mikrokontrollersiden. Men inntil videre ansees altså regulatoren som enheten motorstyringen skal forholde seg til direkte.

Kategorien **målinger** gjelder hovedsakelig funksjoner som ble diskutert i [Mossum 2005], og har her mest relevans som grunnlag for anbefalingene til det videre arbeidet. Ut over dette kan enkelte målinger på sikt bli aktuelt å flytte over til hardware, og blir derfor nevnt for sammenhengens skyld. Dette blir altså kommentert nærmere i kapittel 7.

Kommunikasjon og I/O innbefatter alle definerte kommunikasjonsfunksjoner totalsystemet i følge kravspesifikasjonen skal støtte. Til syvende og sist er det applikasjonen som nyttiggjør seg disse. Men de har også en viss relevans for dette prosjektet med tanke på anbefalingene - for det videre arbeidet, for vurderingene av brikken som mulig plattform, og vurderingene rundt implementering av CAN-bus i hardware.

5.1.3 Hardware / software - Partisjonering:

Bakgrunnen for de valgene av funksjonell partisjonering som er gjort er hovedsakelig vurderinger av hva som ansees hensiktsmessig. Implementering i hardware innebærer som oftest mer komplekse design og lengre utviklingstid. Dette kan vanskelig rettferdiggjøres hvis en softwareløsning allikevel er god nok. Derfor har utgangspunktet vært å ta tak i de tyngste funksjonene, samt de med strengest sanntidskrav, og implementere disse i hardware. Primært dreier dette seg om den grunnleggende motorstyringen. Men også estimering av hastighet, siden dette ville gjort beslag på uhensiktsmessig mye CPU-tid i mikrokontrolleren.

Forøvrig er assembly-kode for utførelse av divisjon i FPSLIC å finne på Atmel sine hjemmesider, sammen med en oversikt over nødvendig CPU-tid. En 16/16 bit divisjon av unsigned ville her kreve 173 klokkesykler á 40ns (25 MHz). Dette gir ca 7 ms, som er i størrelsesorden passeringstiden i motorens hall-sensorer. Med andre ord ville mikrokontrolleren ikke vært i stand til å gjøre stort annet enn å estimere hastighet. I FPGA bør dette kunne utføres meget raskere. Av den grunn var det også et nokså enkelt valg å flytte alle slike beregninger fra den planlagte ”Tilstandsmonitor” i software, til ”Hallmonitor” i hardware. Derfor eksisterer ikke lenger ”Tilstandsmonitor” i blokkdiagrammet. Lettere beregninger kan eventuelt inngå i applikasjonen, og blir således programvareutviklers ansvar.

Regulator ble blant annet lagt til software av den grunn at det foreløpig er ukjent, og utenfor oppgavens søkelys, hva slags regulator som bør benyttes. Det er da langt mer fleksibelt å overlate denne til software i første omgang; slik at motorstyring med et ferdig definert grensesnitt kan implementeres. Software er også atskillig mer fleksibelt for raskt å kunne teste ulike regulatorparametere. At eksisterende programvare utviklet for NRWD kan gjenbrukes er også et poeng. Så kan eventuell overflytting til hardware vurderes senere. I mellomtiden kan antakelig den innebygde hardware multiplikatoren i FPSLIC avlaste reguleringen en del.

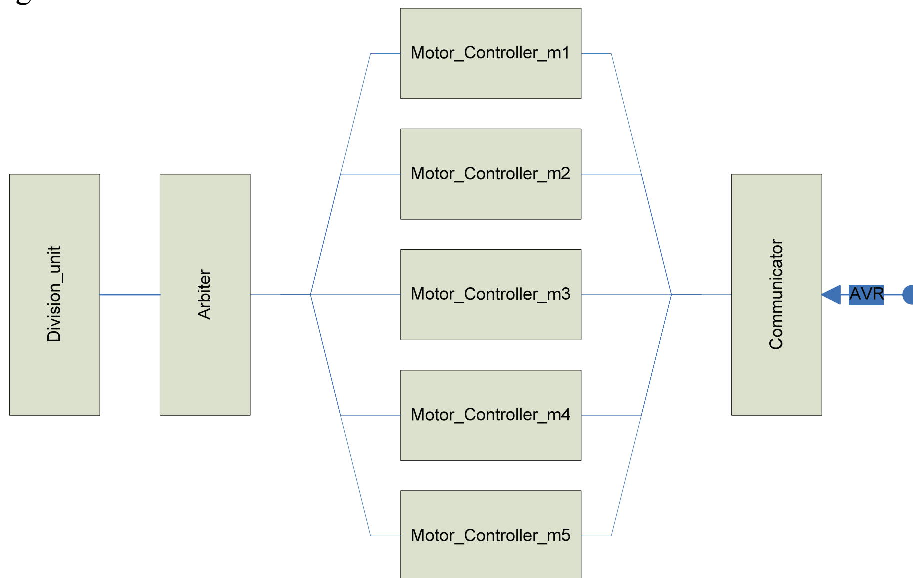
Alle former for ekstern **kommunikasjon** var selvsagte kandidater til software av den enkle grunn at de fleste grensesnittene allerede er støttet med dedikerte hardwareressurser som periferi til mikrokontrolleren i FPSLIC, i tillegg til at det foreløpig ikke er kjent hva disse kommunikasjonskanalene skal brukes til. Dette blir applikasjonsutviklers ansvar å klargjøre. Et annet argument for å håndtere **kommunikasjon** i software er at det er atskillig lettere å teste ut og videreutvikle egendefinerte høynivå protokoller på den måten. Med en gang man skal gjøre noe i hardware, så kompliseres ting med ekstra testing og verifikasjon.

Det at de nødvendige standardene er støttet med dedikert hardware som periferi til mikrokontrolleren gjør at disse funksjonene i liten grad laster prosessoren. Dersom protokollene i sin helhet måtte vært kjørt i software, så hadde saken stilt seg ganske annerledes. Antakelig ville da mikrokontrolleren hatt sin fulle hyre med å innfri krav til timing, og hadde ikke hatt kapasitet til stort annet enn håndtering av én enkelt protokoll. Derfor vil lavnivå protokoller nødvendigvis måtte utføres i hardware, mens de designspesifikke protokollene enklest håndteres i software.

Funksjonene som omfattes av softwaremodulen **målinger** har ikke vært noe direkte fokus for oppgaven, annet enn som medvirkende faktorer for anbefalingene til det videre arbeidet. Modulen er således tatt med for oversiktens skyld. Forøvrig omfatte blokken funksjoner som global vinkelmåling for protesen, eventuell måling av motorstrøm, og måling av pådrag fra analoge EMG-sensorer. De to sistnevnte er målinger som innbefatter bruk av ADC; noe som i tilfelle kan gjøres via ekstern brikke med I2C-grensesnitt.

5.2 Implementasjon – Systemets virkemåte

I dette kapittelet vil det implementerte systemets indre struktur og virkemåte bli beskrevet. Tidligere i rapporten har gjerne de laveste nivåene blitt beskrevet først, før de høyere nivåene har blitt bygget på. Grunnen til dette er som nevnt at arbeidet tok utgangspunkt i det strukturelle designet fra [Mossum 2005], og bygde videre på dette. Dette kapittelet starter heller på toppen av designet og jobber seg nedover i strukturen. Imidlertid er ikke hierarkiet her så dypt at utgjør den helt store forskjellen. Etter hvert som man kommer til detaljbeskrivelsen av de enkelte modulene, så vil også noen av de designspesifikke kravene bli utredet og diskutert.



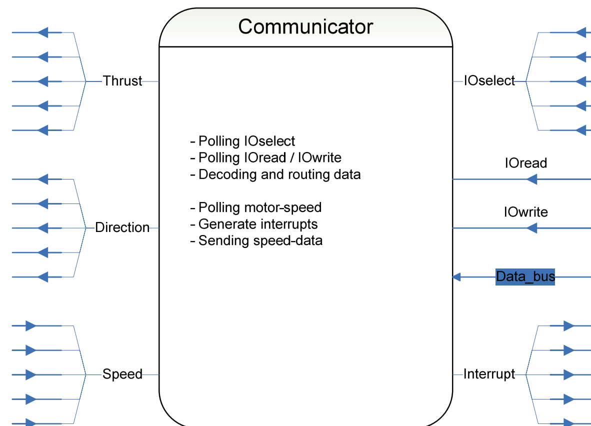
Figur 21 - Top-level entity

Toppnivået i designet består som kjent av de fire modulene Communicator, motor_controller, arbiter og division_unit. Siden Communicator er modulen nærmest mikrokontrolleren funksjonelt sett, og siden stimuli til drift i første omgang kommer derfra, så vil denne bli utredet først. Deretter vil motor_controller sin overordnede virkemåte og struktur bli presentert. Imidlertid vil ikke motor_controller sin indre struktur berøres før de fire toppnivå modulene er beskrevet. Videre beskrives altså arbiters struktur og virkemåte, samtidig som division_unit til slutt kommenteres raskt. Division_unit blir også kommentert nærmere i kapittel 5.3.2. Dette fordi modulen ikke er eget design, men innhentet fra eksterne kilder.

5.2.1 Kommunikasjonsmodulen

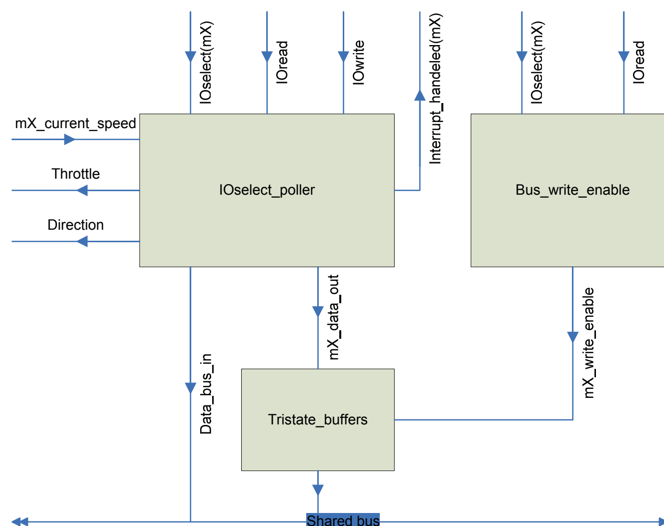
Kommunikasjonsmodulens oppgave er som kjent å opptre som et grensesnitt mellom den delte databussen og motorstyringens indre struktur. Dette innebærer at de nødvendige protokoller må implementeres her, og at modulen står for ruter av data mellom enhetene. Databussen er som kjent fast definert i hardware, og må interfaces som beskrevet i databladet. Motorstyringens grensesnitt er med hensikt designet så enkelt som mulig, slik at all kommunikasjonsrelatert kompleksitet abstraheres fra denne. Dette blir isteden kommunikasjonsmodulens ansvar. Motorkontrolleren presenterer løpende sine ferskeste data på utgangen, mens kommunikasjonsmodulen foruten å følge med på bussen må sørge for at mikrokontroller blir gjort klar over at ferske data foreligger. Modulen er i all hovedsak kombinatorisk, for i best mulig grad å kunne tilpasse seg mikrokontrollerens operasjon. Videre består den av to parallelle tråder. Den ene tråden har som primæroppgave å overvåke databussens kontrollinnganger, samt å utveksle data over bussen. Den andre tråden står for overvåkning av motorene for endrede hastigheter, generering av interrupt til

mikrokontrolleren, samt å levere nye pådragsdata til styringsmodulene. I figur 22 ser vi modulens overordnede grensesnitt beskrevet. Vi ser her at høyresiden samsvarer med figur 2 i kapittel 2.1, og det fastsatte grensesnittet i brikken. Venstresiden av figuren beskriver grensesnittet til resten eget design.



Figur 22 - Motor_controller Overview

Grensesnitt til databussen:
Databussen kontrolleres som kjent fra mikrokontrolleren. Dette gjøres ved hjelp av IOselect med de tilhørende kontrollsignalene IOread og IOwrite. Som forklart i kapittel 4.2 så vil kontrollsignalene så vel som data på bussen kun foreligge i en klokkeperiode á 40ns (25MHz). Kun en select-pinne kan være høy av gangen, og da i kombinasjon med enten IOread eller IOwrite. 5 av de tilgjengelige IOselect er koblet indirekte til hver sin motorkontroller via kommunikasjonsmodulen, og opptrer således som kontrollerens adresse ovenfor mikrokontrolleren. I figur 23 er et funksjonelt blokkdiagram over grensesnittet til databussen fremstilt. For å gjøre figuren mer oversiktlig er strukturen her noe forenklet med tanke på innganger og utganger for 5 motorer. At en port eller et signal består av flere kanaler er indikert direkte på signalene. At navnet avsluttes med (mX) betyr at signalet er en vektor med et dedikert bit for hver kontroller. At navnet begynner med mX betyr derimot at hver kontroller har sin egen vektor. Både prosessnavnene og signalnavnene er autentiske, og kan gjenfinnes i kildekoden.

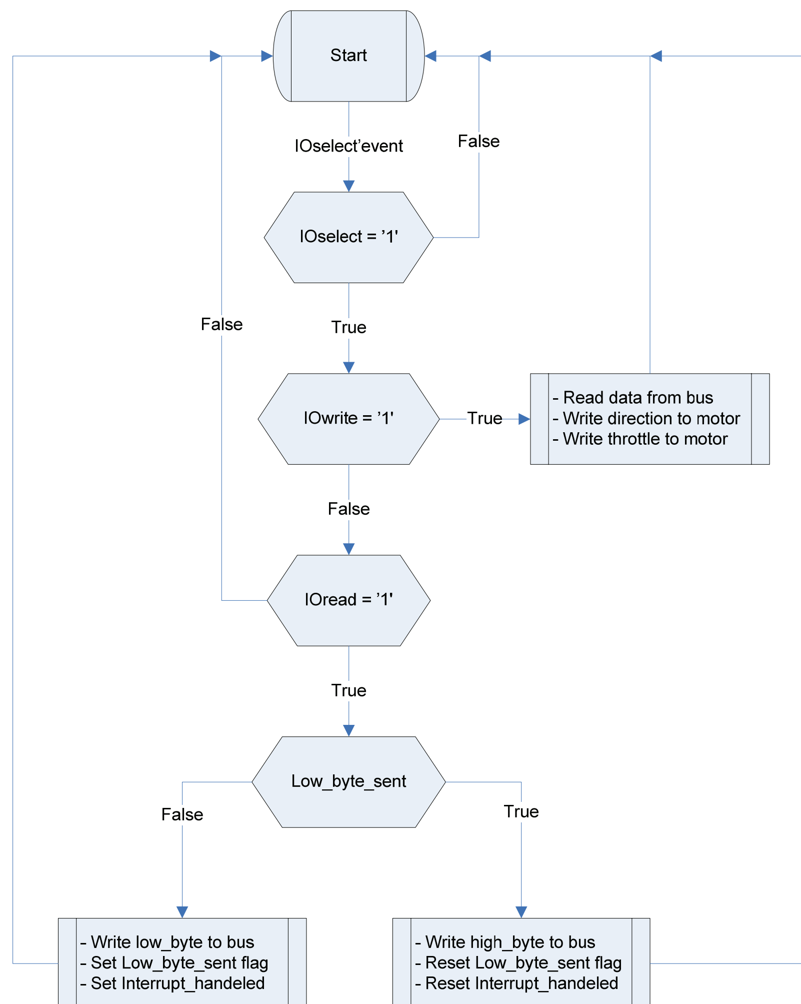


Figur 23 - Bus interface

Alle de tre prosessene som er med på håndteringen av databussen er kombinatoriske. Den viktigste av disse er **IOselect_poller**, som er kjernen i funksjonaliteten. Den overvåker kontrollsignalene fra AVR, og handler umiddelbart i henhold til dennes ønsker. Dersom mikrokontrolleren ønsker å sende, så leses data fra bussen i inneværende klokkeperiode, og rutes videre i henhold til IOselect. Dersom mikrokontrolleren ønsker å lese, så sender modulen data fra den motoren det spørres etter. Det er forøvrig sist beregnede hastighet som sendes, og ikke en beregning utført på grunnlag av kallet i inneværende periode.

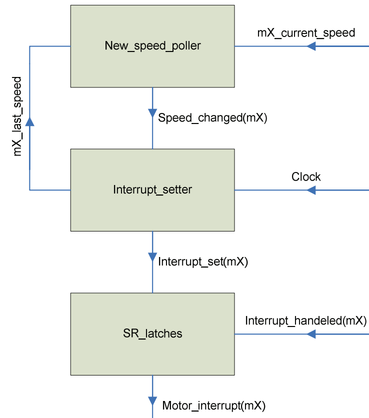
I foreliggende implementasjon består pakker som mottas av FPGA bare av en byte. Hver byte kan således sees på som en selvstendig pakke. Derimot består pakker som sendes mikrokontroller av to byte. Her benyttes interne flagg for å huske tilstanden, i henhold til kravene i kapittel 4.

Misjonen med **bus_write_enable** og **tristate_buffers** er å sørge for delt tilgang til databussen. Tristate_buffers fremstår i realiteten som fem separate buffere, og fungerer som tilkoblingspunkt mellom hver enkelt motorkontroller og bussen. Dersom enable til et buffer er høy, så får verdien på inngangen passere ut til bussen. Er enable lav, så påtrykkes verdien "Z", som i praksis vil si frakoblet tilstand. **Bus_write_enable** genererer da buffernes enable-signal i henhold til IOselect og IOread, slik at mikrokontrolleren har kontroll over tilgangen. I figur 24 er rutinens virkemåte beskrevet i form av et flytdiagram.



Figur 24 - Handeling of bus-interface

Grensesnitt til motorkontrollerne:
Som nevnt i forrige delkapittel så har man valgt å gjøre motorkontrollerne så enkle at de til en hver tid bare presenterer sine ferskeste måledata på utgangen, uten å bry seg med om noen leser dem. På samme måte drives motoren i forhold til det siste mottatte pådraget, helt til et nytt blir mottatt. All protokollhåndtering er således overlatt til kommunikasjonsmodulen. I figur 25 er det funksjonelle blokkdiagrammet over rutinen for overvåkning av motorkontrollerne fremstilt. Også her er både prosessnavn og signalnavn identiske med de i kildekoden.



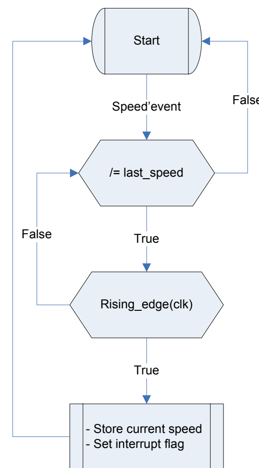
Figur 25

Gangen i denne kjeden er av figuren nokså intuitiv. Det som imidlertid ikke er så selvsagt er hvorfor man har delt rutinen opp i tre blokker. Måten dette fungerer på er at **new_speed_poller** kontinuerlig følger med på signalet *mX_current_speed*, som er en direkte kobling til motorkontrollernes målte hastighet. Denne sammenlignes med forrige observerte verdi, i form av en tilbakekobling fra **interrupt_setter**. Dersom disse avviker fra hverandre, så settes *speed_changed(mX)* høy for gjeldende motor. I neste omgang, når tilbakekoblingen vil samsvare med nåværende hastighet, så settes *speed_changed(mX)* lavt igjen.

Interrupt_setter er den eneste prosessen i kommunikasjonsmodulen som er klokket. Modulen trigger på stigende klokkeflanke og på *speed_changed(mX)*. Dersom et bitt på *speed_changed(mX)* går høyt, aktiveres prosessens rutine på første stigende klokkeflanke. Foruten å lagre gjeldende hastighet for senere sammenligning, settes da det korresponderende bittet i vektoren *interrupt_set(mX)*.

Av måten prosessen er implementert, så vil dette flagget bli slettet igjen ved neste stigende klokkeflanke. Her har man derfor valgt å benytte SR-vipper for å holde på verdien til situasjonen er håndtert. *Interrupt_set(mX)* kobles altså til Set-inngangen på vippen. SR-vippene er således egentlig en utstrekning av interrupt_setter sin funksjon, og kan sees under ett med denne. Videre resettes vippen av signalet *interrupt_handeled(mX)*, som genereres av IOselect_poller.

Fem av brikkens seksten interrupts er benyttet til nettopp dette, å gi mikrokontrolleren beskjed om at ferske måledata foreligger. Det at man har valgt å benytte individuelle interrupt her gjør at mikrokontrolleren umiddelbart vet hvilken motorkontroller det gjelder. Protokollen for datautveksling blir derfor en del enklere, samtidig som nødvendig datatrafikk reduseres.



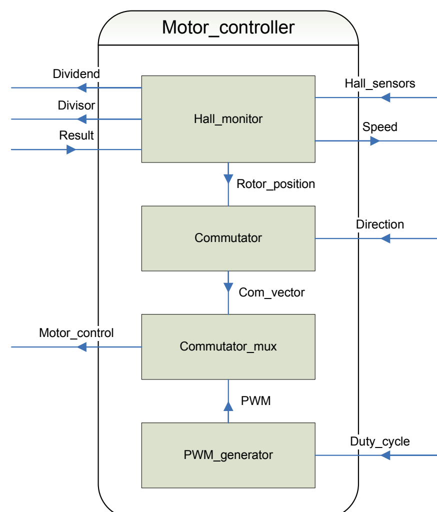
Figur 26 - Motor-interface

Noen ord om kommunikasjonsmodulen som helhet:
Det er antatt at behovet for overføring ikke vil bli så stort at avansert bussarbitrering er nødvendig. Systemet er implementert slik at det alltid er mikrokontrolleren som bestemmer hvilken motor som får sende data, uten at dette sjekkes mot noen historikk for tidligere tilgang. Det vil til en hver tid være motorkontrolleren med lavest indeks som får tilgang, så sant denne ikke hadde tilgang i forrige klokkeperiode. Av hensyn til måten bussen er definert på er dette den enkleste måten å gjøre det på.

5.2.2 Motorkontroller – Overordnet grensesnitt:

Denne modulen er systemets kjerne, og skal inneholde det meste av den funksjonaliteten oppgaven spesifikt ber om. Det vil si alt som har med lavnivåfunksjoner for styring av en børsteløs motor å gjøre. Dette innebærer først og fremst overvåkning av hall-sensorene og generering av kommuteringsvektorene. Andre krav som skal innfris av modulen er verning av drivtrinnet og håndtering av dead-lock i mikrokontrolleren. Videre vil estimering av motorhastighet og støyfiltrering av måledata bli utført i denne modulen.

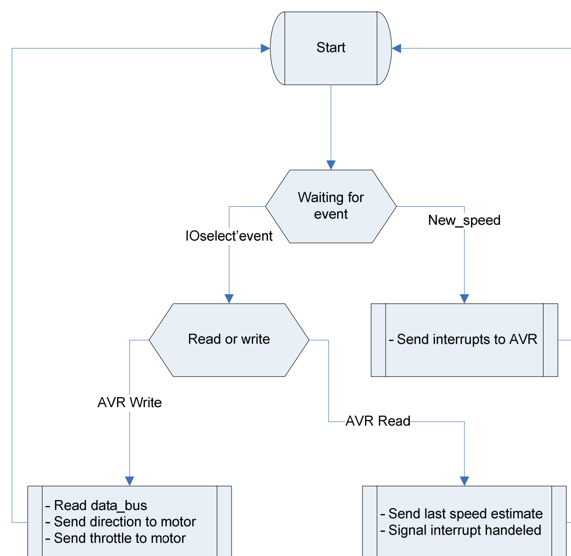
Modulens primære grensesnitt består av inngang for hall-sensorene, inngang for pådrag og retning, samt utgang for styring av drivtrinnet til motoren. Videre har man i relasjon til hastighetsestimeringen et grensesnitt mot divisjonsenheten via arbiter. I tillegg kommer det indirekte grensesnittet til mikrokontrolleren via kommunikasjonsenheten.



Figur 27 - Motor_controller

Modulens oppgave er i essens å følge med på hall-sensorene, og å utføre kommutering på grunnlag av dette. Kommutering utføres i henhold til kapittel 3 om motorens oppbygning og virkemåte, ved hjelp av en enkel simulert mekanisk kommutering. Dette vil si at kommuteringsskift utføres på et forhåndsdefinert tidspunkt relativt til entring av en ny hall-sone. Forøvrig samme kommuteringsmetode som ble benyttet i Skjeltens design. Dette gir opphav til de tre modulene ”hall-monitor”, ”commutator” og ”commutator_mux”. Disse tre submodulene vil bli nærmere beskrevet i kapittel 5.2.4.

I henhold til kapittel 3.3.2 om pådragsstyring av BLDC-motorer, så benyttes spenningsregulering ved hjelp av **pulsbreddemodulasjon** som det primære pådragsorgan. En egen modul for generering av et slikt signal er derfor inkludert i Motor_controller. Modulen som er benyttet her er forøvrig en applikasjonsnote hentet fra Atmel. Denne vil bli kommentert nærmere i kapittel 5.3 – Innhentede moduler.



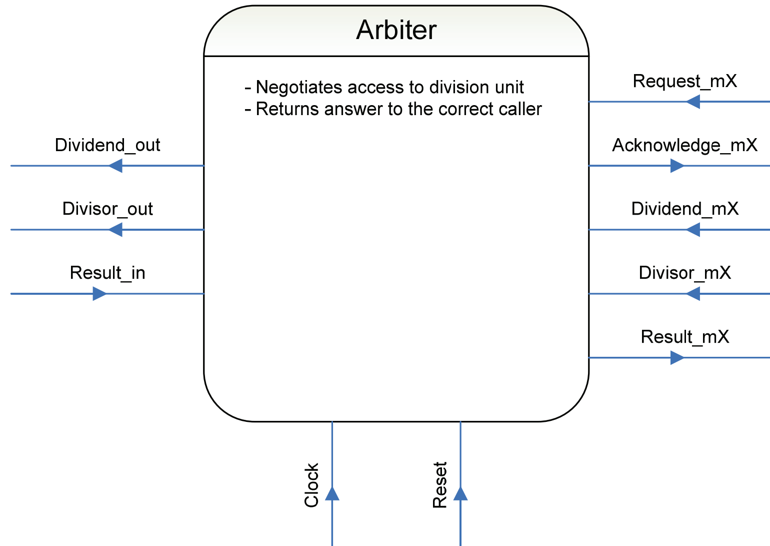
Figur 28 - Motor_controller - Simplified operation

Meget forenklet så beskriver flytdiagrammet i figur 28 kretsens overordnede virkemåte. Man har her abstrahert bort alle de interne prosessene og det utadgående grensesnittet til arbiter / divisjon. Man ser kun det innkommende svaret som en hendelse, type New_speed, som danner grunnlaget for et interrupt til AVR. Sånn sett beskriver flytdiagrammet kun det grensesnittet som direkte angår AVR.

Dead-lock-guard er en funksjon som får motorene til å stanse ved opphør av innkomne styredata. Etter all sannsynlighet vil årsaken til dette være at mikrokontrolleren har hengt seg. For håndtering av slike tilfeller ville man sannsynligvis benyttet mikrokontrollerens watchdog-timer med global reset for gjenoppretting av normal drift. Siden FPSLIC gir mikrokontrolleren mulighet til å initiere reset av FPGA, antas det tilstrekkelig for oppfyllelsen av kravet å benytte denne funksjonen. Det vil si at alt som kreves av logikken er at reset-rutine implementeres i alle sentrale prosesser for motorstyringen. Imidlertid vil dette ikke nødvendigvis være en mulig løsning dersom en ekstern FPGA ble benyttet.

5.2.3 Arbiter og divisjonsmodul:

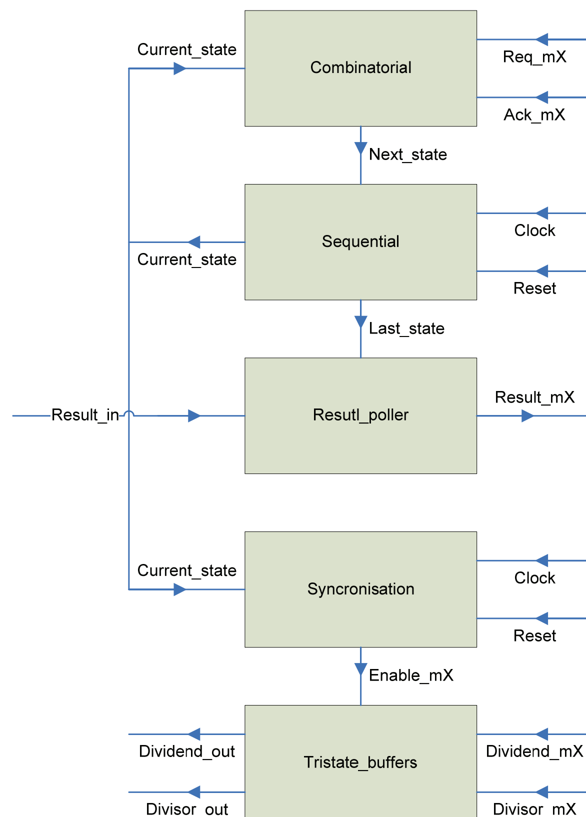
Arbiter sin oppgave er som kjent å opptre som grensesnitt mellom divisjonskretsen og motorkontrollerne. Primært for å håndtere delingen av divisjonsmodulen mellom et ukjent antall motorkontrollere. Men Arbiter har samtidig en funksjon som en slags sokkel; slik at Motor_controller ikke behøver implementeres i forhold til den valgte divisjonsmodulen. I dette tilfellet dreier det seg om håndtering av ulik timing. Dette er gjort primært for å lette skalerbarheten av systemet. Men også for å gjøre systemet mest mulig modulært, slik at man enkelt kan skifte ut divisjonsmodulen. I figur 29 nedenfor er grensesnittet illustrert:



Figur 29 - Arbiter interface

Hver av inngangene på høyre side, hvis navn avslutter med mX, er i realiteten et enkeltstående signal for hver motorkontroller. Kort fortalt så er gangen i prosessen som følger: På et vilkårlig tidspunkt ber en eller flere motorkontrollere om å få utført en divisjon. Så snart arbiter ser at divisjonsenheten er ledig, så sender den en bekreftelse til den høyest prioriterte enheten. Deretter leses data inn fra denne; før arbiter betjener neste forespørsel. Videre klokkes nye beregninger inn for hver stigende klokkeflanke. Svarene fra divisjonen kommer på samme måte på hver stigende klokkeflanke, men først to sykler etter at gjeldende divisjon ble iverksatt. Når så et nytt svar ankommer fra divisjonsenheten, så tar arbiter seg av å levere det til rett adressat.

Arbiter består av både kombinatoriske og synkrone prosesser. De kombinatoriske prosessene har til oppgave å overvåke inngangene fra verden rundt; og sikre at ingen innkomne data går tapt. Den synkrone har til oppgave å sørge for data påtrykkes divisjonsmodulen på rett tidspunkt, og å sørge for at svarene finner veien til rett kaller. I figur 30 er modulens indre struktur beskrevet med et forenklet blokkdiagram. Dette diagrammet viser prosessenes sammenkoblinger og interne kommunikasjon. På samme måte som i figur 29 er de inngangene som har en instans for hver motorkontroller utvidet med mX til slutt.



Figur 30 - Arbiter structure

Av figur 30 ser vi at arbiter består av fem interne moduler eller prosesser. Comb og Seq utgjør til sammen en tilstandsmaskin, som står for prioritering av tilgang til divisjonsmodulen. Sync står for å generere enable signaler i rette tid, som i sin tur aktiverer Tristate_buffers. Det er disse som sørger for at kun én motorkontroller om gangen har anledning til å påtrykke data på bussen. Result_poller tar inn svarene løpende, og videresender dem til rett kaller.

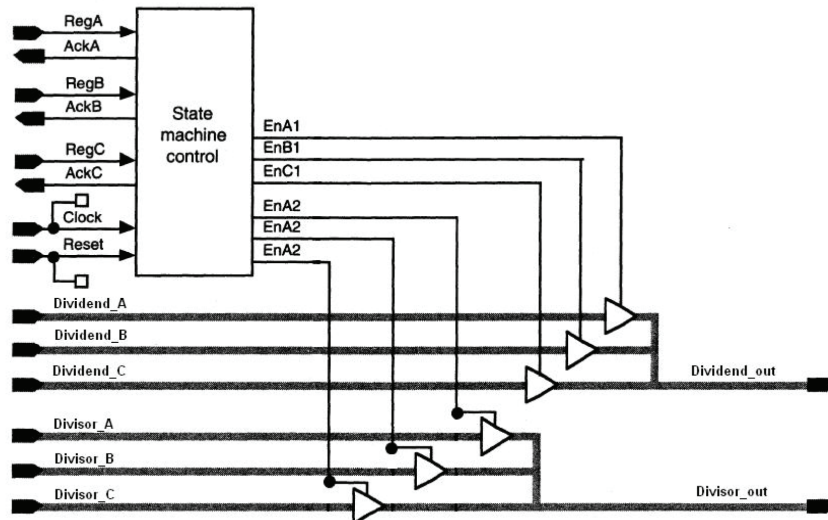
Tilstandsmaskinen, bestående av **Comb** og **Seq**, har altså til oppgave å gi tilgang til divisjonsmodulen. Comb er som navnet tilsier den kombinatoriske delen. Denne overvåker kontinuerlig nye forespørsler. Så snart en ny eller uhåndtert forespørsel oppdages, så starter en evaluering på grunnlag av gjeldende tilstand. Hvis divisjonsmodulen viser seg å være opptatt, så drøyes avgjørelsen til den igjen er ledig. En bekreftelse på innvilget tilgang sendes høyest prioriterte kaller, samtidig som neste tilstand avgjøres. Og her tar Seq over for Comb. På hver stigende klokkeflanke oppdaterer Seq gjeldende tilstand, i henhold til signalet Next_state fra Comb. Gjeldende tilstand sendes da tilbake til Comb via signalet Current_state. Dette signalet er som vi ser også koblet til modulen Sync.

Sync tar utgangspunkt i neste tilstand, og gir den aktuelle motorkontrolleren tilgang til bussen ved å sette opp to enable-signaler. Imidlertid sender ikke motorkontrollerne sine data før ved neste klokkeflanke, når tilstanden er et faktum. På mange måter så er Sync og Tristate_X funksjonelt sett knyttet sammen på lignende måte som Comb og Seq. Sync sørger for at enable-signalene til tristate-bufferne aktiveres på stigende klokkeflanke. Tristate_X symboliserer sånn sett de til sammen 10 tristate-bufferne som styrer skrivetilgangen til bussen.

Result_poller følger kontinuerlig med om nye resultater fra divisjonsmodulen foreligger. Modulen er kombinatorisk, og sender umiddelbart det innkomne svaret videre på grunnlag av

Last_state. Denne tilstanden forteller Sync hvilken motorkontroller som hadde tilgang til divisjonsmodulen når de korresponderende påtrykkene ble sendt. Modulen forenkler dermed oppgaven for motorkontrollerne på en sånn måte at disse ikke trenger å tenke noe mer på divisjonen de har bedt om før svaret ankommer.

I figur 31 nedenfor er den logiske strukturen i en tilsvarende arbiter illustrert. Figuren viser en arbiter med tre innganger, men uten håndtering av svarene.



Figur 31 - Arbiter Eksample-structure

5.2.4 Motorkontroller – Indre struktur:

I dette kapittelet vil de tre egendesignede modulene Motor_controller består av bli inngående beskrevet. Momenter som har vært belyst tidligere vil ikke nødvendigvis bli repetert her. For nærmere detaljer om den innhentede pulsbredde-modulen, se kapittel 5.3.1.

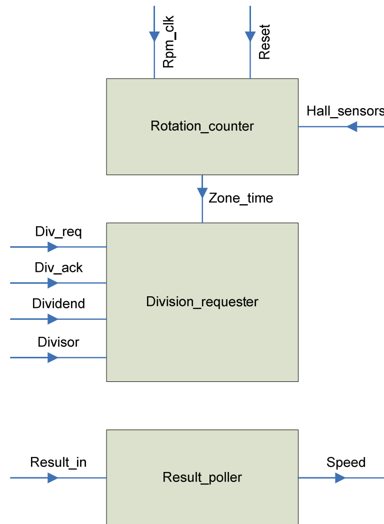
Når strukturen for motorkontroller ble foreslått i [Mossum 2005], så var dette også med enkelte fremtidige funksjoner i tankene. Ellers ville det av hensyn til størrelse og kompleksitet vært naturlig å slå sammen et par av submodulene. Dette vil bli kommentert under beskrivelsen av de enkelte modulene.

Hall_monitor:

Modulens primære oppgave var opprinnelig å overvåke hall-sensorene og rapportere endringer videre til Commutator. Dette er i seg selv en såpass enkel oppgave at en egen modul ikke er påkrevd. Imidlertid var tanken at denne modulen enkelt skulle kunne byttes ut med en versjon inneholdende funksjoner for mer avansert kommutering. Den ble altså opprettet for å gjøre et fremtidig design modulært.

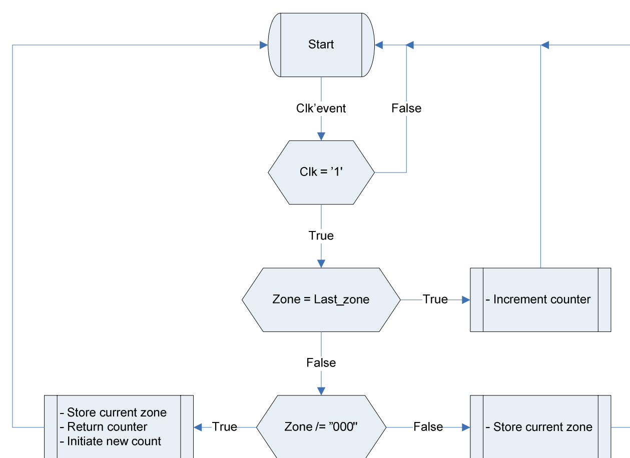
Ettersom det ble besluttet å implementere hastighetsestimering i hardware, viste Hall_monitor seg raskt som den mest naturlige plassen å utføre dette. Og så lenge ikke mer avansert kommutering er implementert her, så er dette i bunn og grunn også alt den gjør. Det som angår sensoravlesning er i foreliggende versjon ikke annet enn kanalisering av signaler. Sånn sett ville det kanskje være fornuftig å gi modulen et annet navn. Men det har en altså valgt å ikke gjøre. I alle tilfeller er det derfor hastighetsestimeringen som vil bli beskrevet her.

Hastighetsestimeringen fungerer på følgende måte: Entring av en ny hallsone trigger en klokket rutine som starter en teller. Klokken som benyttes har en kjent svingefrekvens som telleren er dimensjonert i forhold til. For hvert klokkeslag inkrementeres da telleren, helt til en ny hallsone entres. Verdien telleren har nådd lagres da i et annet buffer, før ny telling starter. Den lagrede telleren representerer da det antallet tidsenheter motoren brukte på å passere sist hallsone. Ved å dividere klokkefrekvensen kretsen benytter med tellerverdien, så får man isteden antall hallsoner motoren passerer per sekund. Ved å multiplisere dette svaret med 6 (antallet hallsoner) får man størrelsen oppgitt i enheten runder per sekund, eller Hertz. Av årsaker som vil bli utredet senere i rapporten, så er klokkefrekvensen lagret som en konstant, allerede dividert med 6.



Figur 32 - Speed-estimation

I figur 32 er Hall_monitor sin indre struktur beskrevet. Den består altså av tre prosesser, hvorav to er direkte koblet. Den tredje er indirekte koblet til de andre to igjennom svaret som ankommer fra divisjonsmodulen. **Rotation_counter** er prosessen som utfører tellingen. I figur 33 er denne prosessens virkemåte beskrevet med et flytdiagram.



Figur 33 - Rotation_counter

Når en telling er utført, og måltilstanden nederst til venstre er nådd, så overlates verdien til **Division_requester**. Denne forespør da Arbiter om tilgang til Ddivision_unit ved å sende et *div_req* (division request). Så snart *div_ack* (division acknowledge) går høy, så overlates data til Division_unit. Denne funksjonen er delt opp i to prosesser av den grunn at

Rotation_counter ikke skal behøve vente på acknowledge fra Arbiter. Ved raskere å kunne returnere til sine tellinger, vil også målingens nøyaktighet bli bedre.

Division_requester er av samme årsak som Rotation_counter implementert på en slik måte at den fortsetter med sitt så snart acknowledge er mottatt. Isteden har man altså innført modulen **Result_poller**, som har til hovedoppgaven å vente på svaret. Det er også denne modulen som har implementert støyfilteret, og som står for videresending av de ferske målingene. Med utgangspunkt i en ekstern konstant, må en endring over en viss terskel for at sending skal initieres. Hvis terskelen overgåes, blir allikevel målingen oversendt med full nøyaktighet. Hensikten er bare å ikke sende tilnærmet samme måling gang på gang.

Estimering av nødvendig nøyaktighet:
Prinsippet er som forklart ovenfor at kretsen teller hvor mange klokkeflanker som forekommer mellom to hallsoner eller kommuteringsskift. Når man vet hva klokkefrekvens som vil bli benyttet, så har man altså tilgjengelig en størrelse med enheten [tid / hallsone]. Eventuelt [tid / rotasjon] hvis man multipliserer med 6 for en full rotasjon. Dersom man dividerer klokkefrekvensen på størrelsen [tid / rotasjon] så får man isteden den størrelsen man er ute etter; nemlig [rotasjoner / tid], eller runder per sekund for å være spesifikk.

Enheten [tid / rotasjon] kunne også vært brukt til å regulere på. Men det ville gjøre regulatoren ulineær. Når målingene presenteres på formatet [rotasjoner / tid] så slipper man altså denne ulineariteten. Men til prisen av det ekstra arealet og kompleksiteten en divisjonskrets innebærer. Utrykket som benyttes ser altså slik ut:

(0.3) $$R_{motor} = \frac{f_{cpu}}{counter \times 6} [RPS]$$

For å unngå behov for en dedikert multiplikasjonsenhet, implementeres funksjonen som vist i formel 0.4. Dette fordi multiplikasjoner med toerpotenser kan utføres ved et venstreskift. I det følgende vil allikevel formelen uttrykkes uten denne detaljen, for å fremstå mest mulig oversiktlig.

(0.4) $$R_{motor} = \frac{f_{cpu}}{counter \times 4 + counter \times 2} [RPS]$$

Man har videre definert at regulatoren skal få reliable målinger opp til 24.000 RPM.

(0.5) $$\frac{24.000 rpm}{60 s} \times 6 = 2.400 [com/s]$$

Et minstekrav på 2.400 kommuteringer per sekund er altså det tellerfrekvensen må beregnes ut ifra. Imidlertid kan ikke samplingsteoremet brukes direkte, siden det ikke er tilstrekkelig å få med seg at et soneskift inntreffer. Simuleringer har vist at bruk av en samplingsfrekvens på ca 3,2x kommuteringsfrekvensen kunne gi så mye som 50 % målefeil. Denne målefeilen oppstår da på grunn av klokkeforskyvning (clock-skew). Hadde klokken vært synkron med hall-flankene, så hadde ikke målefeilen oppstått.

Clock-skew er altså ikke til å unngå i denne sammenhengen. Man må dermed bare finne en minste tellerverdi der denne driftingen ikke lengre har noen vesentlig betydning. Vi definerer

løst at en målefeil maksimalt kan ligge i området rundt 10 % for de høyeste hastighetene. Det antas at en tellerfrekvens i størrelsesorden 10x kommuteringsfrekvensen vil kunne innfri dette kravet.

(0.6) $$2.400[Hz]\times 10 = 24.000[Hz] = 24[kHz]$$

Så lenge tellerfrekvensen er over 24 kHz, så er altså målefeilen denne delen av prosessen bidrar med under 10 %. På et utviklingskortet Atmel har laget for denne brikken finnes 32,768 kHz klokkefrekvens tilgjengelig, ment for sanntidsapplikasjoner. Denne er da naturlig å ta som utgangspunkt.

(0.7) $$\frac{32.768[Hz]}{2.400[Hz]} \approx 13,65$$

Denne er altså innenfor det man har definert seg som minstekravet. Hvis motoren kjører konstant på 24.000 rpm vil da telleren oscillere mellom 13 og 14, grunnet klokkeforskyvning. I formel 0.8 og 0.9 er måleverdien for de to tilfellene beregnet. Samtidig påpekes det at det i foreliggende implementasjon ikke foreligger desimaler for svarene.

(0.8) $$\frac{32.768[1/s]}{13\times 6} = 420,103[1/s] \Rightarrow 420[r/s]\times 60[s/m] = 25.200[r/m]$$

(0.9) $$\frac{32.768[1/s]}{14\times 6} = 390,095[1/s] \Rightarrow 390[r/s]\times 60[s/m] = 23.400[r/m]$$

Hvis man deler målingene på den reelle hastigheten, så finner man målefeilen:

(0.10) $$\frac{25.200}{24.000} = 1,050 \Rightarrow \underline{\underline{5,0\%}}$$

(0.11) $$\frac{23.400}{24.000} = 0,975 \Rightarrow \underline{\underline{2,5\%}}$$

Målefeilen vil med 32,768 kHz tellerfrekvens være fra 2,5 % til 5 %. Dette må sies å være akseptabelt i forhold til kravene. I kapittel 7 er allikevel en del tiltak for bedring av nøyaktigheten presentert. I alle tilfeller må man i software ta hensyn til at målefeilen kan bli relativt stor ved høye hastigheter, siden kretsen i foreliggende versjon også sender målinger over 24.000 RPM.

Videre må tellerens nødvendige størrelse beregnes. Her er det den laveste hastighetene man ønsker å kunne måle som setter grenseverdien. I samarbeid med veileder har man satt kravet her til 800 rpm. Med 32,768 kHz som tellerfrekvens blir da tellerens størrelse som følger:

(0.12) $$Counter = \frac{f_{cpu}}{R_{\min}\times 6} = \frac{32.768}{\frac{800}{60}\times 6} = 409,6 \Rightarrow 2^9 = 512$$

Man trenger altså en teller på 9 bit for å dekke de laveste hastighetene. Men som vi ser blir allikevel telleren noe større enn påkrevd. Hvis man snur om litt på formel 0.12, så kan man raskt finne hva som vil være den reelt laveste målbare hastigheten før man får overflyt.

(0.13) $$R_{\min} = \frac{f_{cpu}}{Counter \times 6} \times 60 = \frac{32.768[1/s]}{512 \times 6} \times 60[s/m] = 640[1/m]$$

Med en tellerfrekvens på 32,768 kHz, og en teller på 9 bit kan altså hastigheter ned til 640 rpm måles før man vil få overflyt i registeret. Ved 800 rpm reell hastighet ville målingen da hevde følgende:

(0.14) $$\frac{32.768[1/s]}{409 \times 6} = 13,35[1/s] \Rightarrow 13[r/s] \times 60[s/m] = 780[r/m]$$

(0.15) $$\frac{32.768[1/s]}{410 \times 6} = 13,32[1/s] \Rightarrow 13[r/s] \times 60[s/m] = 780[r/m]$$

Vi ser her at klokkeforskyvning ikke lengre har noen nevneverdig betydning når tellerfrekvensen blir vesentlig høyere enn kommuteringsfrekvensen. Derimot blir avrundingsfeilen nå av større betydning. Målefeilen som ligger i metoden blir:

(0.16) $$\left(1 - \frac{\frac{32.768[1/s]}{410 \times 6}}{\frac{800[1/m]}{60[s/m]}}\right) \times 100 = \left(1 - \frac{13,3203}{13,3333}\right) \times 100 \approx \underline{0,10\%}$$

Tilsvarende utregning for 409 gir 0,15 % avvik. Altså innbefatter selve målemetoden med nevnte parametere en gjennomsnittlig målefeil på 0,13 % og et maksimalt avvik på 0,15 %. Når avrundingsfeilen taes med, blir avviksprosenten:

(0.17) $$\frac{780}{800} = 0,975 \Rightarrow \underline{2,5\%}$$

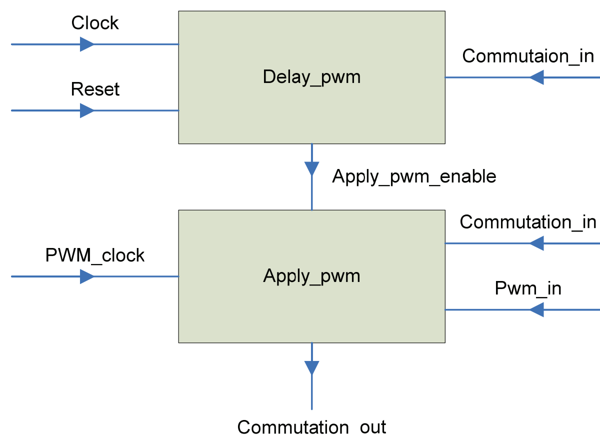
Det er altså avrundingen som begrenser målenøyaktigheten for de lave hastighetene, mens det tellerfrekvens og klokkeforskyvning som setter begrensningen for de høyeste hastighetene. Flere mulige tiltak for å bedre nøyaktigheten vil bli utredet i kapittel 7.

Commutator:
Denne modulens ansvar er generer kommuteringsvektorene på grunnlag av tilbakekoblingen fra hall-sensorene. Den foreliggende versjonen utfører da også kommuteringen utelukkende på bakgrunn av denne tilbakekoblingen; og da så fort et soneskift er registrert. Altså en simulert mekanisk kommutering. Måten dette gjøres er ved at modulen inneholder en oppslagstabell med kommuteringsvektorer, der oppslaget gjøres på grunnlag av gjeldende hallsone og ønsket kjøreretning. Disse vektorene er seks bitt lange, som beskrevet i kapittel 3. De laveste tre bittene angir kobling til jord, mens de tre høyeste bittene angir kobling til drivspenningen. For nærmere detaljer om kommuteringsmetoden, se kapittel 3.

Såfremt kun enkel simulert mekanisk kommutering benyttes, så vil det antakelig være greit å slå denne modulen sammen med "Commutator_mux". Derimot ville mer avanserte kommuteringsteknikker være naturlig å implementere i "commutator". Modulen er altså opprettholdt for å gjøre et mulig fremtidig design mest mulig modulært.

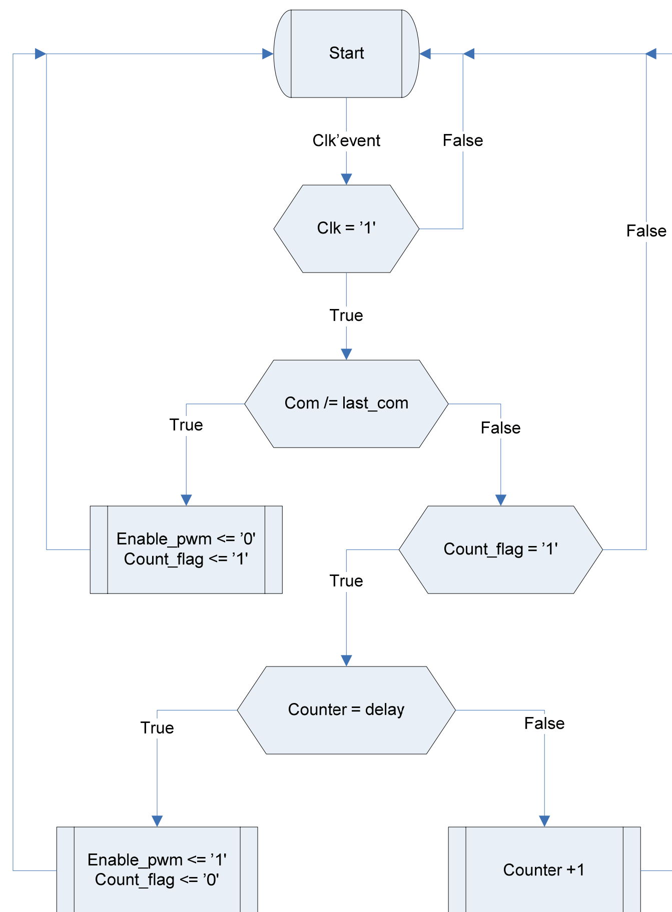
Commutator_mux:
Denne modulen har da to oppgaver: For det første har den til oppgave å mikse pwm-signalet mellom de tre spolene i motoren etter behov. For det andre skal modulen innføre den nødvendige forsinkelsen for å beskytte drivtrinnet mot overoppheting.

Dersom man hadde nøyd seg med kommuteringsmodulen slik som den foreligger, så ville motoren i praksis kjørt på full effekt hele tiden. Derfor innfører man spenningsregulering. Spenningsregulering utføres i henhold til de funksjonelle kravene med pulsbreddemodulasjon. Men for å slippe tre pwm-generatorer per motor, svitsjes heller signalet fra en slik modul til den til en hver tid aktive spolen i motoren. Med dette oppnår man altså å spare areal i FPGA. PWM-signalet kobles altså til drivtrinnets kontrollinnganger. Imidlertid svitsjes kun de tre høyeste bittene, som styrer kobling til drivspenning. De tre laveste bittene styres med konstant 1 eller 0. Denne funksjonaliteten er implementert i modulen **Apply_pwm**, vist i figur 34.



Figur 34 - Commutator_mux

Den andre funksjonen modulen skulle inkludere var kommuteringsforsinkelse for vern av drivtrinnet. **Delay_pwm** i figur 34 er prosessen som realiserer denne funksjonaliteten. Prosessen fungerer på den måten at en ved ny innkommende kommuteringsvektor fra Commutator, så avbrytes gjeldende kommutering, samtidig som en intern teller startes. Inntil denne telleren har nådd sin målverdi, så forblir Apply_pwm deaktivert. Målverdien er da en konstant som enkelt kan endres før syntese. Først når denne målverdien er nådd, så utføres kommuteringsskiftet. Man har altså innført en tidsluke der alle transistorer ideelt sett er avslått. Lengden av denne luken er altså avhengig av forsinkelsesfaktoren og systemets klokkefrekvens. Figur 35 viser et flytdiagram for prosessen Delay_pwm. Her er *com* en forkortelse for *commutation_in*, og *last_com* tilsvarende for forrige verdi. *Count_flag* er en intern indikator på at telling pågår, og sammenligningsfaktoren *delay* er den omtalte forsinkelsen.

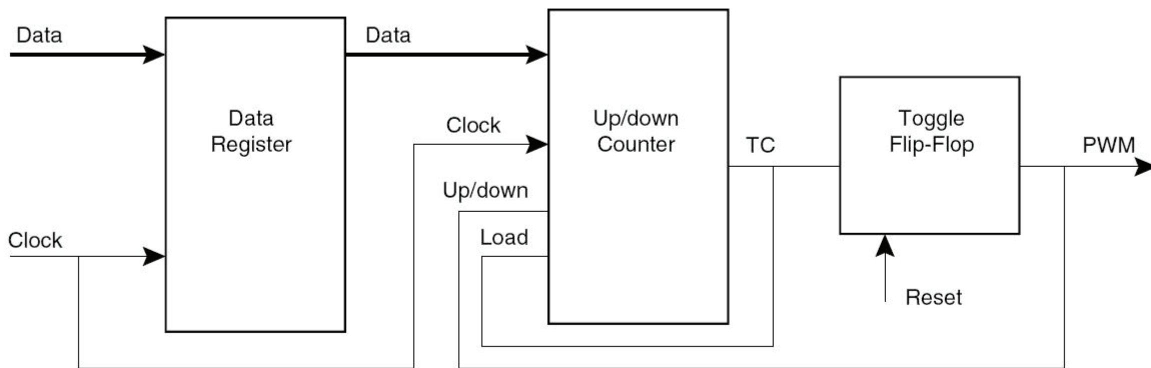


Figur 35 - Apply_pwm process

5.3 Innhentede moduler

5.3.1 PWM:

PWM-modulen som er benyttet i designet er som nevnt hentet fra Atmel sine websider. For nærere informasjon om prinsippet bak pulsbredde modulasjon, se kapittel 3. Kort fortalt tar modulen et 8 bits input som duty-cycle. De 7 bittene mikrokontrolleren sender blir da venstreskiftet en plass før de påtrykkes modulen, slik at pådraget blir et 8 bits ord. Siden modulen er designet for 8 bit, så man dette som en mer hensiktsmessig løsning enn å skrive om modulen. I figur 36 er et blokkdiagram av modulen vist.



Figur 36

Applikasjonsnoten denne modulen er hentet fra er skrevet med tanke på denne brikken. Uten nærmere studier av implementasjonen antas det derfor at modulen er implementert på en hensiktsmessig måte; og at det ikke er noen stor grunn til å redesigne den. En omskriving til 7 bit kunne imidlertid vært nyttig for å spare areal. Å redusere oppløsningen vil også gjøre at modulen får høyere effektiv svitsjefrekvens. Svitsjefrekvensen beregnes med uttrykket:

(0.18)
$$f_{pwm} = \frac{f_{system}}{2^{bitwidth}}$$

I Leonardo Spectrum har PWM_modulen syntetisert til å kunne kjøre med en klokkefrekvens på 36,5 MHz. Dette gir:

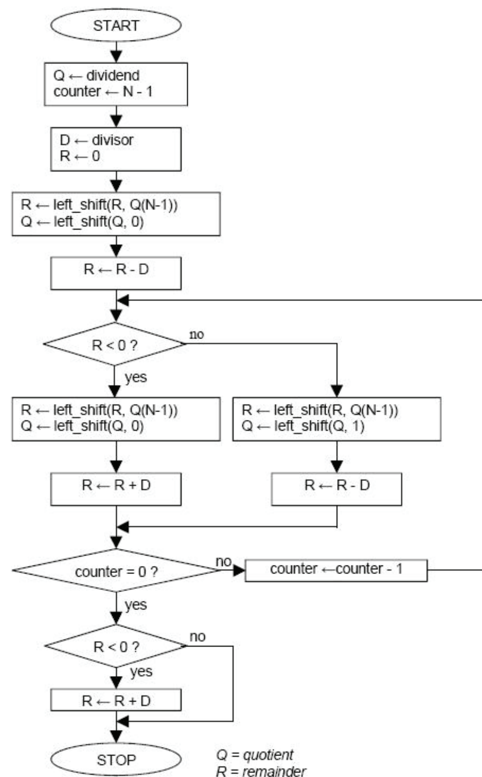
(0.19)
$$f_{pwm} = \frac{36,5 \times 10^6 [Hz]}{2^8} = 142.578 [Hz] \approx \underline{143 [kHz]}$$

Dette kan trygt sies å være innenfor kravene. Dersom modulen ble omskrevet til å reelt bare benytte 7 bits oppløsning, som jo strengt tatt er det den har, så ville denne frekvensen vært doblet. Imidlertid ville nok arealbesparelsen vært mer lukrativ enn den økte svitsjefrekvensen. Om ønskelig kunne man gått så høyt som til 10 bits oppløsning, og fremdeles holdt kravet om svitsjefrekvens over 20 kHz. I det tilfellet ville den blitt ca 36 kHz. For 11 bit ville ikke kravet lengre være oppfylt, med en svitsjefrekvens på 17,8 kHz.

5.3.2 Divisjon:

Divisjonsmodulen er hentet fra OpenCores.org, og later til å stamme fra firmaet DeverSys. I følge forfatteren Vladimir V. Erokhin (PhD) skal modulen være grundig testet og verifisert, skjønt det ikke har lyktes undertegnede å kompilere den medfølgende testbenken. (Feil antas å skyldes variasjoner i tilgjengelige biblioteker). Kun en enkel autogenerert testbenk er benyttet for å analysere timing-diagrammet i forkant av integrering. Imidlertid har alle praktiske beregninger fra modulen latet til å være korrekte. Ut over det har man altså sett seg nødt til å stole på forfatterens ord om at modulen er testet og verifisert. Algoritmen skal etter forfatters personlige erfaring være benyttet i to reelle industridesign.

Modulen er i essens kombinatorisk, og designet for å utføre én divisjon per klokkesykkel. Altså er kjernen kombinatorisk, mens rammen rundt er synkron. Dividend og divisor leses inn på positiv klokkeflanke, mens svaret foreligger på utgangen to klokkeflanker senere. Nye data kan klokkes inn for hver påfølgende klokkeflanke om ønskelig. På samme måte presenteres svarene løpende, to klokkesykler etter data ble påtrykket. I figur 37 er algoritmen illustrert ved et flytdiagram.



Figur 37 – Non-restoring unsigned division algorithm

Enheten er designet for å ta inn en dividend på 32 bit og en divisor på 16 bit. Bredden på disse kan imidlertid endres ved å forandre en enkelt konstant. I alle tilfeller vil dividend være dobbel bredde av divisor. Svaret er 33 bit langt; eller 1 bit lengre enn dividend. Dette ekstra bittet indikerer at divisjon på null er forsøkt utført. I tilfellet med 32 / 16 ordbredde, vil de 16 øverste bittene være heltalssvaret, mens de 16 laveste bittene vil være resten fra divisjonen. Denne resten fra divisjonen har man i dette designet valgt å forkaste. I tilfellet at en divisjon på null er blitt forsøkt utført, vil heltalssvaret bestå av bare enere, i tillegg til indikatoren selv. Rest-delen vil være bare nuller.

Optimalisering av areal:
I enkelte tekster funnet på nett har det vært hevdet at nødvendig arealet for en kombinatorisk divisjonskrets stiger proporsjonalt med ordbredden. Dette betyr at det sannsynligvis er mye å tjene på å gjøre optimaliseringer her. Med full bredde har forfatteren syntetisert kretsen til 4600 gates, og til en maksimal klokkefrekvens på 14 MHz. Imidlertid er vesentlig bedre resultater enn dette oppnådd ved syntetisering med Leonardo Spectrum for AT94K (FPSLIC). Disse resultatene vil bli kommentert i kapittel 6.

Foruten det at divisjonsmodulen er implementert kombinatorisk, så er det modulens ordbredde som påvirker arealet i størst grad. Det mest effektive man kan gjøre for å begrense ordbredden er da å velge en så lav tellerfrekvens som mulig for hastighetsestimatoren. Grunnen til denne direkte innvirkningen er at telleren ved redusert tellerfrekvens ikke lengre vil oppnå like høye verdier innenfor en hall-sone. Betraktningene rundt dette er beskrevet i kapittel 5.2.4. Her ble 32.768 Hz valgt som tellerfrekvens, mye fordi denne finnes tilgjengelig på utviklingskortet som er tenkt benyttet til testformål.

Som beskrevet i kapittel 5.2.4 vil en tellerfrekvens på 32.768 Hz kreve en teller på 9 bit for å innfri kravene. Denne telleren skal i følge algoritmen multipliseres med 6 (antallet hall-soner) før divisjonen utføres med resultatet som divisor. Multiplikasjonen var tenkt utført før divisjonen for å maksimere målenøyaktigheten. Men denne viste seg i alle tilfeller å være så god sammenlignet med avrundingsfeilen, at man har valgt å heller prioritere optimalisering av areal. Telleren på 512 multiplisert med 6 gir maksimalt 3072 som divisor, et tall som krever 12 bit (4096) for å representeres binært. Dividend er altså 32.768, et tall som krever 15 bit (32.768) for binær representasjon. Det er altså dividend som presser ordbredden. Det første tiltaket som er gjort for å redusere behovet er således å dividere tellerfrekvensen med 6 før divisjonen utføres. Dette ofrer som sagt litt presisjon; men gjøres for å spare areal.

$$\frac{32.768}{6} = 5461{,}33 \approx 5461 \Rightarrow 2^{12} = 8192 \tag{0.20}$$

Dermed kan altså dividend potensielt reduseres til 12 bit istedenfor 15 bit. Samtidig reduseres divisors nødvendige ordbredde fra 12 til 9 bit. Det som da gjenstår er å se på den nødvendige ordbredden for svaret fra divisjonen. Dette er nemlig styrt av den samme faktoren som styrer bredden for dividend og divisor. I følge utregningene i kapittel 5.2.4 vil telleren ved 24.000 rpm ha en gjennomsnittelig verdi på 13,65. Hvis en da velger 14 som verdi, så kan vi beregne maksimal ordbredde for svaret som følger:

$$\frac{5461}{14} = 390{,}071 \Rightarrow 2^{9} = 512 \tag{0.21}$$

Også svaret vil maksimalt bli 9 bit bredt. Altså er det dividend som setter nedre grense for ordbredden i divisjonsmodulen, som altså blir 24 /12 / 25. Vi observerer samtidig at målenøyaktigheten knapt påvirkes av endringene vi har gjort. (390,071 vs 390,095).

5.4 Systemets skalerbarhet

En del av oppgaven var å legge opp til et mest mulig skalerbart design. En del er sagt om dette under veis i rapporten, ved beskrivelse av den enkelte modulen. Men her kommer allikevel et en liten sammenfatning av tingene som er gjort. Man kan si at systemet er implementert med tanke på skalerbarhet på to områder. Det mest innlysende er med tanke på flere motorer. Men skalerbarhet har også vært et tema når det gjelder enkel tilpassning av presisjonen i hastighetsestimatoren, og divisjonsmodulen.

Mye er allerede sagt om skalerbarheten i forhold til flere motorer. Hovedtanken har vært at motorkontrolleren skal være en selvberget modul, som inneholder alle nødvendige funksjoner for å drive motorene. Videre skal denne kunne dupliseres i det ønskede antallet. Altså er det kun toppentiteten som må justeres i forhold til ønsket antall motorer på dette feltet. Den tilstøtende logikken, som vil si communicator og arbiter, krever derimot litt mer manuell tilpasning i den foreliggende implementasjonen. Disse består i stor grad av switch-case og nøstede if-strukturer. Som oftest med en alternativ gren for hver motorkontroller. Her vil det altså være nødvendig med litt kopiering og liming for å tilpasse antallet.

I disse gjentagende blokkene, og for så vidt i designet ellers også, er alias og konstanter benyttet en del. På den måten har man klart å få en gren som for eksempel angår motorkontroller_1 til å bare inneholde 1-tall, og ingen andre indekser. For eksempel benyttes da m1 som indeks når et signal skal tilordnes verdi. På den måten blir det meget enkelt å benytte "finn og erstatt" i editoren når blokkene skal tilpasses, i tillegg til at koden blir mer strukturert og lettlest.

Videre krever det også litt manuelle endringer med tanke på grensesnitt til omverdenen, og signalføringer. Men mye er her gjort ved at globale konstanter benyttes for å deklarere bredden av en del porter og signaler.

Når det gjelder presisjonen i hastighetsestimatoren så kan denne enkelt modifiseres, kun ved å endre fire konstanter, samt å koble til den nødvendige tellerfrekvensen. Ut over dette er ingen endring av kode nødvendig. Konstantene som må endres ligger i fila *packages.vhd*, og bærer følgende navn: Freq_div_6, counter_width, data_width og speed_width. Freq_div_6 er som navnet impliserer tellerfrekvensen dividert på 6. Counter_width er tellerens bredde, og beregnes i henhold til formel 0.14 på side 44. Data_width er konstanten som benyttes for å sette bredden av de fleste porter relatert til hastighetsestimeringen og divisjon. Speed_width er en egen konstant til samme formål, som gjør det mulig å justere hvor høye hastigheter man ønsker å kunne ta vare på før man får overflyt. Fordelen med dette er at man har muligheten til å ikke bruke ressurser på målinger hvis presisjon uansett er for dårlig til å regulere på. Avrundingsfeilen man innfører kan imidlertid ikke påvirkes med disse virkemidlene. Da må algoritmen forandres, noe som vil bli kommentert nærmere i kapittel 7.

6. Systemtest / simulering / evaluering

Testene i dette kapittelet søker som oppgaven spesifiserer å verifisere at designet fungerer korrekt under stasjonære forhold, og antar med det at verifikasjonen også er gyldig for normal drift med varierende pådrag. Man tester altså logikken for situasjoner der motorens hastighet har stabilisert seg.

6.1 Testmetode og testbenker

Siden man ikke har noen modell av motoren, så vil ikke et pådrag få den fiktive motoren til å bevege seg, slik en virkelig motor ville gjort. Isteden må testbenken imitere sensorskift fra en fiktiv motor. De to hendelsene, pådrag og sensorskift, er sånn sett ikke knyttet direkte til hverandre i testbenken. Samvirkningen oppstår først internt i den simulerte logikken. Sensorskiftene er i realiteten implementert helt uavhengige av pådraget, siden en modell for dette i seg selv ville innført en mengde potensielle feil.

I første omgang imiterer testbenken brikkens databuss, og at mikrokontrolleren ønsker å sende pådrag over denne. Ulike innledende pådrag sendes til de fem motorkontrollerne i henhold til databladet. På den måten får man først testet om Communicator responderer rett på stimuli, og om data blir sendt til rett mottaker. Man får også testet at PWM_module generer et signal i henhold til pådraget. Men kommuteringen, som jo simulerer et mekanisk system, tar ikke til før testbenken påtrykker verdier for hall-sensorene til den fiktive motoren.

I neste omgang må testbenken derfor imitere motorens hallsensorer. Man påtrykker logikken suksessivt påfølgende de verdier som sensorene ville hatt ved normal drift på en gitt hastighet. Man får således ikke testet samsvar mellom pådrag og motorhastighet. Men ut ifra timing-diagrammet ser vi om logikken utføres korrekt i henhold til sensorskiftene. I tillegg til at vi får verifisert at Commutator virker korrekt, får vi også satt Commutator_mux på prøve. I første omgang er vi her ute etter at som et minimum er en kopi av inngangen. Det neste vi ønsker å observere er at de tre øverste bittene i vektoren ut fremstår som kopier av pwm-signalet. Samtidig ønsker vi å observere at verken pwm-signalet eller de tre laveste bittene slipper igjennom før etter den gitte forsinkelsen vi innførte for vern av drivtrinnet.

Videre får man testet hele rekken av funksjoner som inngår i målingen av motorhastigheten. Dette innebærer da både Arbiter, Division_unit og grensesnittene mellom modulene. Med utgangspunkt i de definerte forsinkelsene mellom sensorskiftene fra testbenkens side kan man sammenlignet beregnet hastighet med reell hastighet. Man får også mulighet til å studere virkningene av det omdiskuterte fenomenet klokkeforskyvning.

Sist, men ikke minst, får man testet om Communicator reagerer korrekt på innkommende hastighetsmålinger fra motorkontrollerne, og om interrupt genereres i henhold til disse. Når interruptene inntreffer, så responderer testbenken med å initiere utlesing av data, på samme måte som mikrokontrolleren er ment å skulle gjøre det.

Testbenken inneholder altså en rekke funksjoner for å imitere en mikrokontrollers og databussens virkemåte i den virkelige kretsen. Men siden disse har vært under konstant forandring igjennom arbeidet, så har man ikke villet utrede disse, annet enn deres funksjonelle virkemåte. For nærmere detaljer henvises det her til testbenkens kildekode, *Top_module_tb2.vhd*.

Men før systemet har kommet så langt som til denne testbenken, så har de enkelte modulene blitt testet grunnleggende hver for seg. Dette gjelder samtlige moduler, unntatt Arbiter. Grunnen til det er at en fullverdig testbenk for denne måtte imitere nærmest hele resten av systemets virkemåte; noe som ville vært mer nærmest like krevende å implementere som systemet selv. Denne ble av den grunn heller debugget direkte i totalsystemet. Derfor vil ikke individuelle simuleringsresultater for Arbiter være å finne i rapporten. Resten av systemets moduler er som sagt simulert, og vil bli beskrevet i påfølgende kapittel.

6.2 Simuleringsresultater

Her vil først systemets enkeltmoduler og deres simuleringsresultater bli presentert, før de sammensatte modulene testes. Først ut er Motor_controller sine moduler. Deretter Motor_controller som helhet, før Communicator og Division_unit. Til sist vises altså simuleringsresultatene av hele motorstyringen under ett.

6.2.1 Motor_controller – Undermoduler:

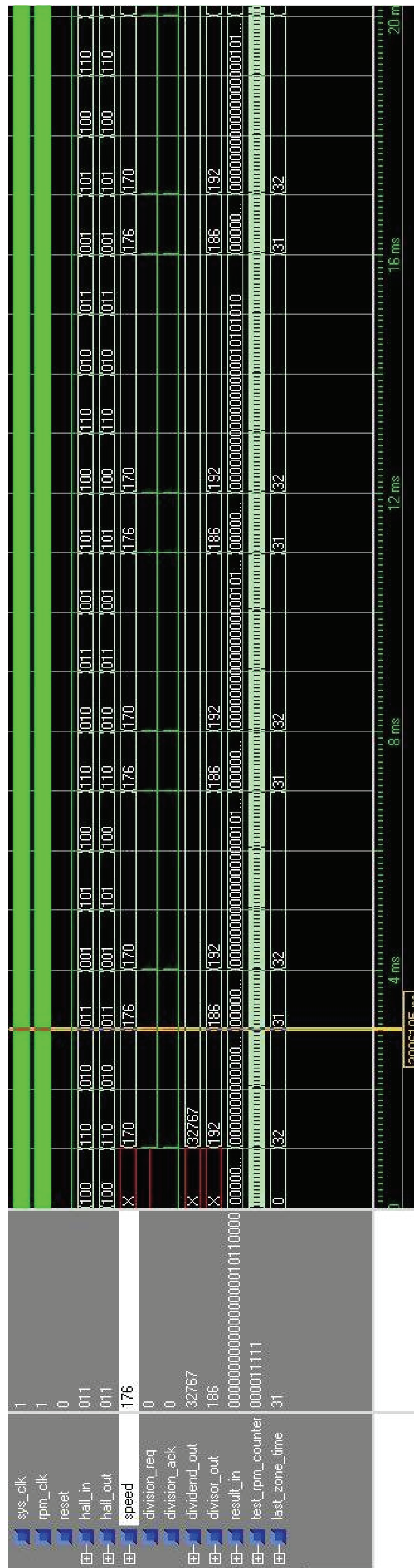
Testing av Hall_monitor:

Hall_monitor er som beskrevet i kapittel 5 en modul som i foreliggende versjon primært står for hastighetsestimeringen. Modulens primære inngang, foruten klokke og reset, er signalet fra hall-sensorene. Videre har den grensesnitt til Arbiter, og utgang til Communicator for hastighet.

Testbenken er designet slik at den først påtrykker suksessive stimuli på inngangen hall_sensors, som om motoren kjørte med en konstant hastighet på 10.000 RPM eller 167 RPS. Dette starter da beregning av hastighet internt i modulen. Videre har testbenken implementert en funksjonell oppførselsbeskrivelse av divisjonsenheten, men uten den forsinkelsen den virkelige kretsen har. Svaret foreligger en klokkesykkel etter at divisjon ble forespurt. Deretter presenterer modulen forhåpentligvis resultatet den fikk fra testbenken på sin utgang Speed. I figur 38 er et utsnitt av simuleringen illustrert. Denne simuleringen er forøvrig utført før forhåndsdivisjon av dividend ble innført.

Her ser vi modulens operasjon helt fra oppstart og initialisering. Vi ser også at motoren må kjøre en full kommuteringssone før alle signaler blir gitt verdi. Forøvrig blir reset av kretsen utført før oppstart, skjønt dette ikke fremkommer av figuren.

Etter at den fiktive motoren har passert en full kommuteringssone, ser vi at både *dividend_out* og *divisor_out* blir gitt verdi. Vi ser også at dividend blir gitt verdi tilsvarende tellerfrekvensen (32.768 Hz), og at divisor blir gitt verdi tilsvarende *last_zone_time* x 6. Siden tidslinjen er såpass kraftig zoomet ut, så synes ikke resultatet sendt fra divisjonen i dette plottet. Men vi ser av *Speed* hva svaret ble; nemlig 170 RPS, eller 10.200 RPM. I neste omgang kan vi observere at målingen har driftet til 176 RPS, eller 10.560 RPM. Simuleringen viser altså en maksimal målefeil på 5,6 % for gjeldende hastighet.



Figur 38 - Hall_monitor

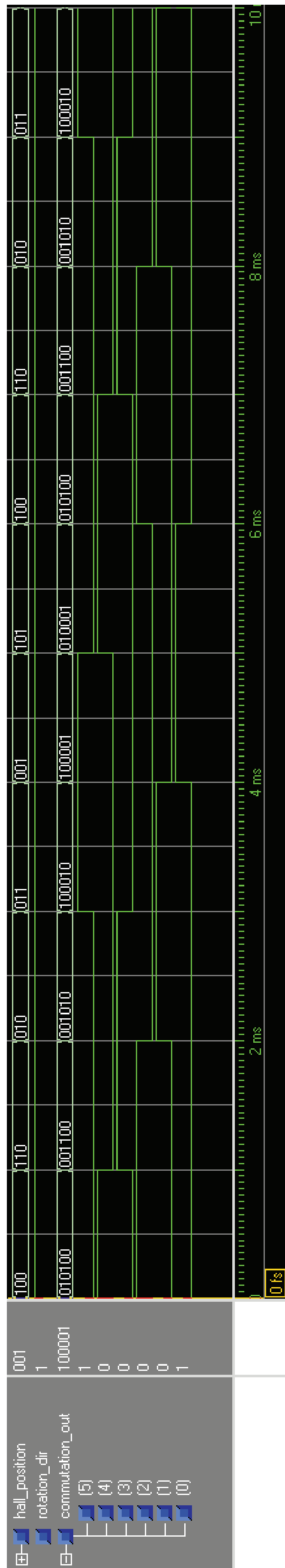
Testing av Commutator:
Commutator er som tidligere nevnt den enkleste modulen i systemet, og er i essens ikke annet enn én oppslagstabell. Sånn sett er eneste nytteverdi med en simulering å se at signaltransisjonene samsvarer med tabellverdiene. I figur 39 er modulens simuleringsresultat illustrert for 10 kommuteringssoner, og ved en rotasjonshastighet på 10.000 RPM.

Testing av Commutator_mux:
Modulen tar som kjent inn kommuteringsvektoren fra Commutator, og pwm-signalet fra PWM_module. Testbenken påtrykker derfor et fiktivt pwm-signal med periodetid på 100us og 50 % duty-cycle. Videre påtrykkes nye kommuteringsvektorer suksessivt med 1ms mellomrom, tilsvarende ved 10.000 RPM rotasjonshastighet. Simuleringsresultatene for de 10 første kommuteringssonene er illustrert i figur 40. Kommuteringsvektorene benyttet her samsvarer forøvrig med de Commutator genererte i forrige simulering.

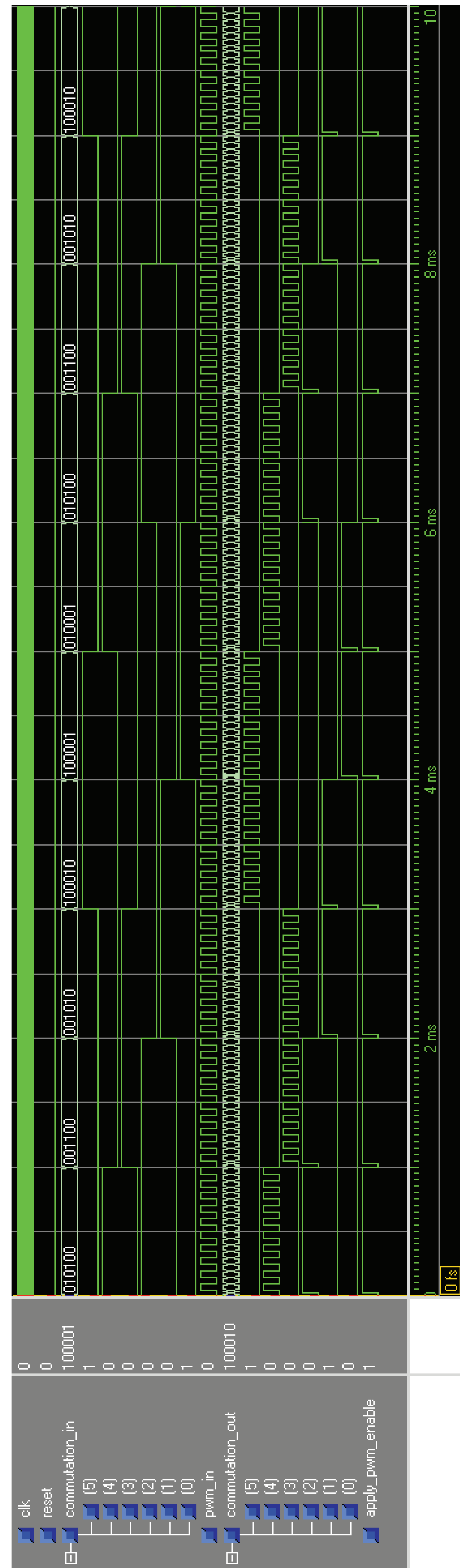
Simuleringen viser at pwm-signalet multiplekses til de tre øverste bittene i *commutation_out*, i henhold til *commutation_in*. Modulen fungerer altså som tilsiktet på dette feltet. Simuleringen viser også at de tre laveste bittene viderekobles fra inngang til sin korresponderende utgang.

Commutator_mux var også ment å skulle innføre en forsinkelse av kommuteringen, til vern for drivtrinnet. Dette er ikke så lett å se av pwm-signalet, siden denne i alle tilfeller oscillerer med en frekvens som går opp kommuteringsfrekvensen. Men av de laveste tre bittene er det lett å observere at kommuteringen effektivt forsinkes. Imidlertid kan det også observeres en kort spiker i signalet ut, ved transisjonen til hver nye kommuteringssone. Denne spikeren er i gjeldende tilfelle 30ns (33 MHz) bred, som er klokkefrekvensen testbenken benytter.

Årsaken til denne feilen er ikke identifisert, men skyldes sannsynligvis måten Apply_pwm er implementert på. I følge prosjektleder skaper ikke en så kort spiker noe nevneverdig problem for drivtrinnet. Det representerer således bare litt ekstra varmeutvikling. Men i alle tilfeller er dette er en feil som med fordel kunne vært utbedret.



Figur 39 - Commutator



Figur 40 - Commutator_mux

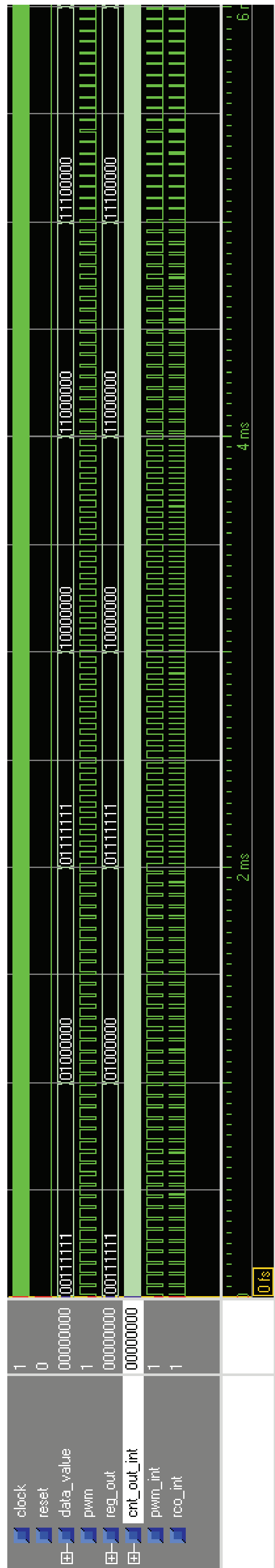
Testing av PWM_module:
Denne modulen ble testet med testbenken levert av Atmel. Siden modulen ble hentet fra en såpass reliabel kilde som Atmel, så ble den til å begynne med ikke testet for andre pådrag enn de som var skrevet inn i testbenken. Såfremt tellingen fungerte som den skulle, så forventet man ikke å finne noe interessant ved å teste for flere ulike pådrag. Men når modulen nå ble gått litt nærmere i sømmene oppdaget man et par uventede ting.

I figur 41 kan vi lese av signalet *data_value* for å se hva pådrag modulen har vært utsatt for. Det som var litt overraskende å oppdage var at modulen tilsynelatende genererer et pwm-signal hvis duty-cycle er invers av pådraget. Som vi ser øker *data_value* i seks trinn mot høyre, mens *pwm* faktisk avtar i tetthet. Dette er for så vidt ikke være noe problem, bare programvareutvikler er klar over det.

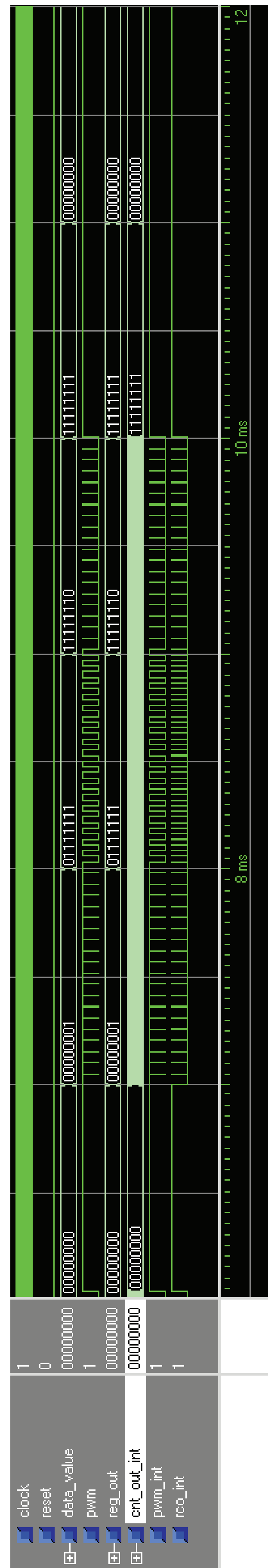
For sikkerhets skyld testet man også for grenseverdiene 0, 1, 254 og 255. Som figur 42 viser, stoppet ikke overraskelsene med modulens inverse virkemåte. Simuleringen viser at den inverse virkemåten stemmer for verdiene 1 og 254. Men for ytterpunktene 0 og 255 var saken en annen. Figuren viser her at modulen genererer 100 % duty-cycle i begge tilfeller, noe det ville være rimelig å anta at kun skulle være tilfelle for 0 som pådrag.

Dette var altså noe som ikke ble oppdaget før mot slutten av arbeidet; og har således ikke blitt gått nøyere i sømmene. Om virkemåten er tilsiktet fra Atmels side er uvisst. At modulen virker inverst av pådraget kunne kanskje være. Men sistnevnte særegenhet gjør det litt vanskelig å tro at oppførselen er tilsiktet. Siden modulen internt ikke er manipulert under arbeidet, er det vanskelig å se at lokale faktorer skulle kunne ha forårsaket dette. Eneste mulige forklaring man kan se for seg på at dette er en feil med lokale årsaker er at andre biblioteker er benyttet enn de Atmel designet modulen for. I forbindelse med implementasjon av divisjonsmodulens oppførselsbeskrivelse ble nemlig et annet bibliotek importert, som kan tenkes å ha overlappet enkelte andre definisjoner. Men før saken er nærmere undersøkt blir dette bare spekulasjoner.

I alle tilfeller bør virkemåten kunne håndteres, slik at modulen fortsatt kan benyttes. Det kan i tilfelle være ønskelig å deaktivere modulen ved 0 og 255 i pådrag. Sprang antas ikke å være noe problem, så lenge en slik funksjon er implementert. Ellers ville det beste selvfølgelig vært å endret modulen til forventet virkemåten.



Figur 41 - PWM_module



Figur 42 - PWM_module

6.2.2 Motor_controller – Sammensatt modul:

Her testes altså Motor_controller, med alle subbmoduler tilkoblet. Verifisering av Motor_controller vil innebære en oppfyllelse både av kravet om at systemet skal kunne styre en børsteløs motor, og kravet om motorens hastighet skal estimeres i kjøretid.

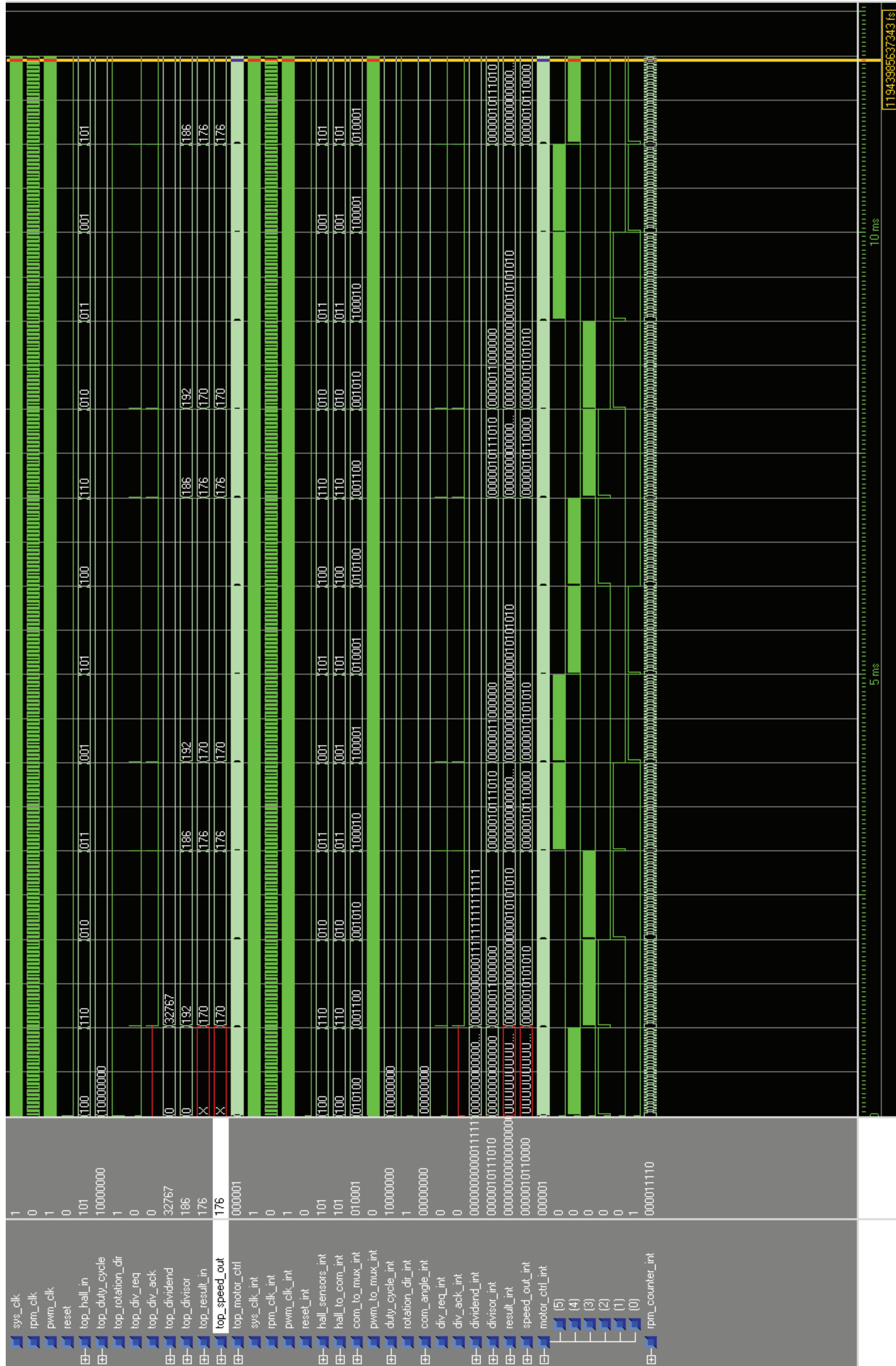
Testbenken påtrykker modulen klokkefrekvensen 33 MHz for systemklokken, og 32.768 Hz for hastighetsmålingen. Videre påtrykkes motoren ønsket duty-cycle og retning. På lik linje med testbenken for Hall_monitor, så er oppførselsbeskrivelsen av divisjonskretsen inkludert, slik at hele kjeden kan testes. Testbenken påtrykker også fiktive sensordata, tilsvarende en rotasjonshastighet på 10.000 RPM (166,67 RPS) som vanlig.

Figur 41 illustrerer simulering av modulen, med to fulle rotasjoner fra oppstart. Før testingen startes opp har testbenken utført reset av kretsen; skjønt det ikke synes i plottet. Som figuren viser, blir ønsket pådrag og retning levert påtrykt modulens innganger ved oppstart. Testbenken hopper så direkte til antatt stasjonære forhold, og påtrykker sensordata som om motoren løp med 10.000 RPM.

Som signalet *motor_ctrl_int* viser, så svitsjes pwm-signalet til utgangen, i henhold til hall-sensorene. Det vises også av samme signal at forsinkelsen her oppfører seg på samme måte som i den individuelle simuleringen.

Av signalene top_req og top_ack, vises det at modulen kaller divisjon hver gang den interne tellerens verdi er ulik forrige verdi. Dette skjer, som forventet, umiddelbart etter et kommuteringsskift. Men altså ikke for hvert eneste skift, så lenge verdien ikke er endret siden sist. Signalet top_divisor viser tellerens verdi multiplisert med 6. Signalet top_speed_out viser den estimerte hastigheten i øyeblikket. Denne varierer her mellom 170 og 176, mens den reelle verdien som nevnt er 166,67. Altså et maksimalt avvik på 5,6 %. Dette er som forventet av beregningene i kapittel 5.

Totalt sett må simuleringene sies å verifisere korrekt virkemåte, med unntak av den allerede påpekte mangelen med forsinkelsen. Unøyaktigheten i hastighetsmålingen er også noe man forventet, med tanke på den benyttede algoritmen. Imidlertid vil forslag til forbedringer av målingen bli presentert i kapittel 7. Så får det bli opptil prosjektleder å avgjøre om disse skal implementeres.



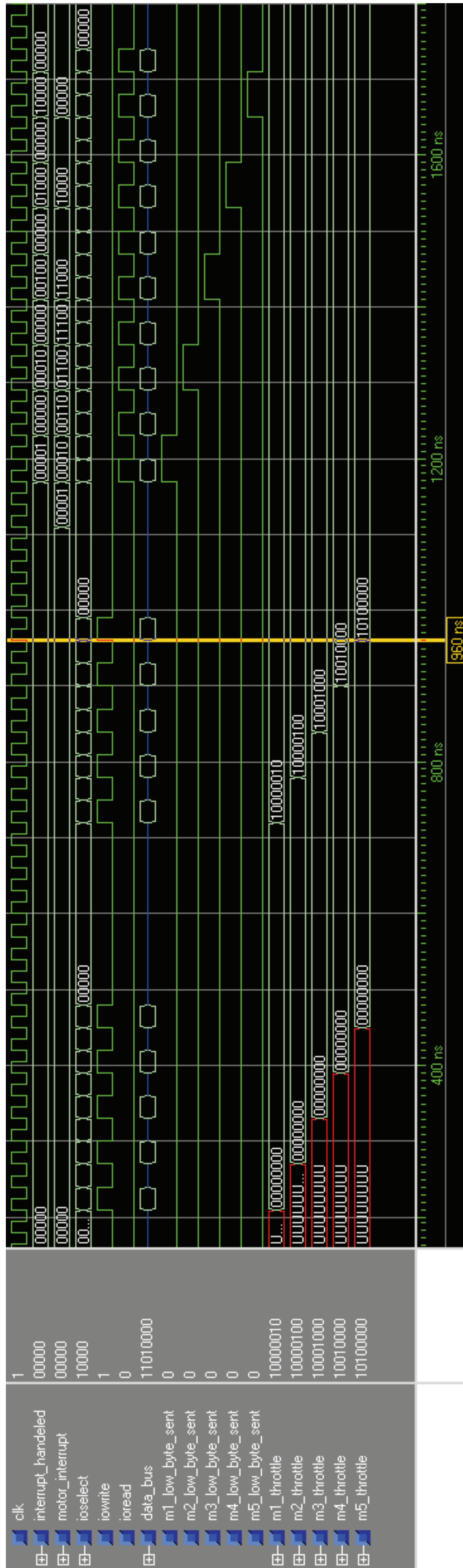
Figur 43 - Motor_controller - Simulation

6.2.3 Communicator:

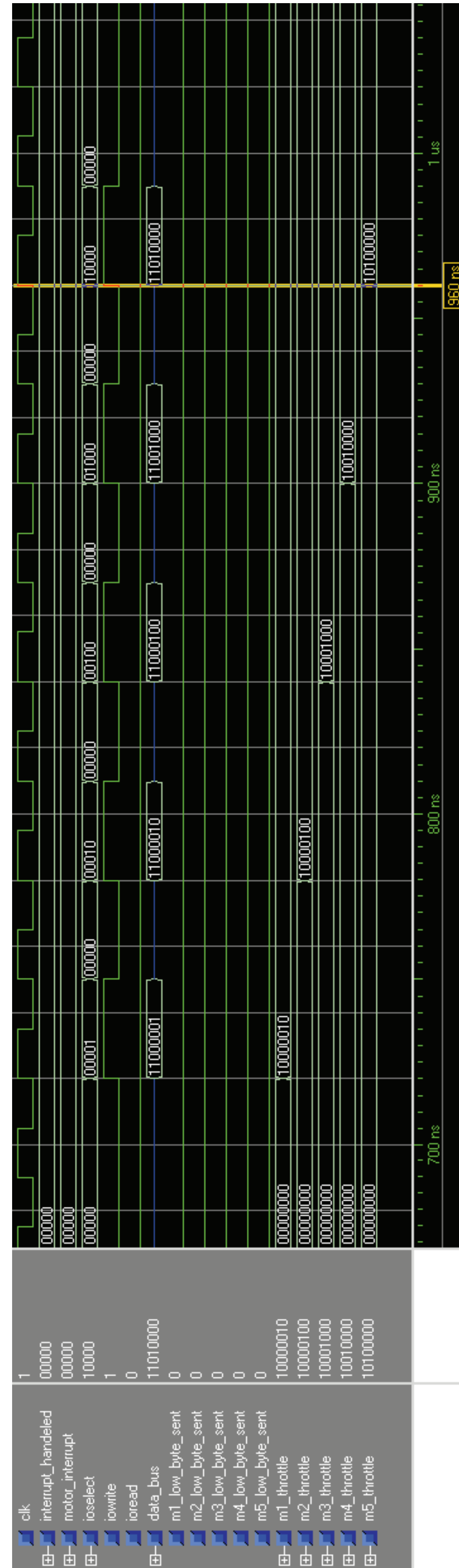
Testbenken for Communicator er utvilsomt den så langt mest komplekse av testbenkene. Den skal nemlig imitere bussens og mikrokontrollerens oppførsel ved normal drift. Denne består da av en prosess som lytter håndterer interrupt, en prosess som står for kommunikasjon med Communicator, og til sist en prosess som styrer påtrykk av stimuli. For nærmere detaljer her henvises det til dokumentasjon i testbenkens kildekode.

Det første testbenken gjør er å sende en nullvektor til samtlige fem fiktive motorkontrollere. Dette for å sikre at alle registre har en kjent verdi, også de som ikke blir direkte affektert av reset. Figur 44 viser hvordan fem påfølgende pakker sendes over bussen, og hvordan de tilhørende kontrollsignalene oppfører seg. Samtidig vises det av signalene mX_throttle at nullvektorene er mottatt som ønsket pådrag.

I neste omgang påtrykker testbenken reelle testvektorer. Her har testvektorene en systematikk som skal gjøre det mulig å se om data havner på rett sted. Alle vektorene starter med 11, mens et ettall er lagt til på slutten av vektoren som en slags indeks. For motorkontroller m1 vil dette altså si 11000001. Videre er dette avsluttende ettallet skiftet en plass til venstre for hver indeks mX. På den måten blir den videre gangen 11000010, 11000100, 11001000 og 11010000. Når resultatene studeres må man i tillegg huske at det er de 7 laveste bittene som etter et venstreskift blir påtrykket motorkontrolleren. Altså vil alle ha fått skiftet en 0 inn til slutt i vektoren, når de har nådd sin destinasjon. Av figur 45 kommer det for det første frem av *data_bus* hva de sendte data var. Samtidig vises det at den samme vektoren blir skrevet til mX_throttle i venstreskiftet versjon, i henhold til IOselect øverst i plottet. Det kan også observeres hvordan overføringen startes idet kontrollsignalet IOread går høyt. Og med det ansees modulen å være verifisert for mottak og ruting av data til fem motorkontrollere.



Figur 44



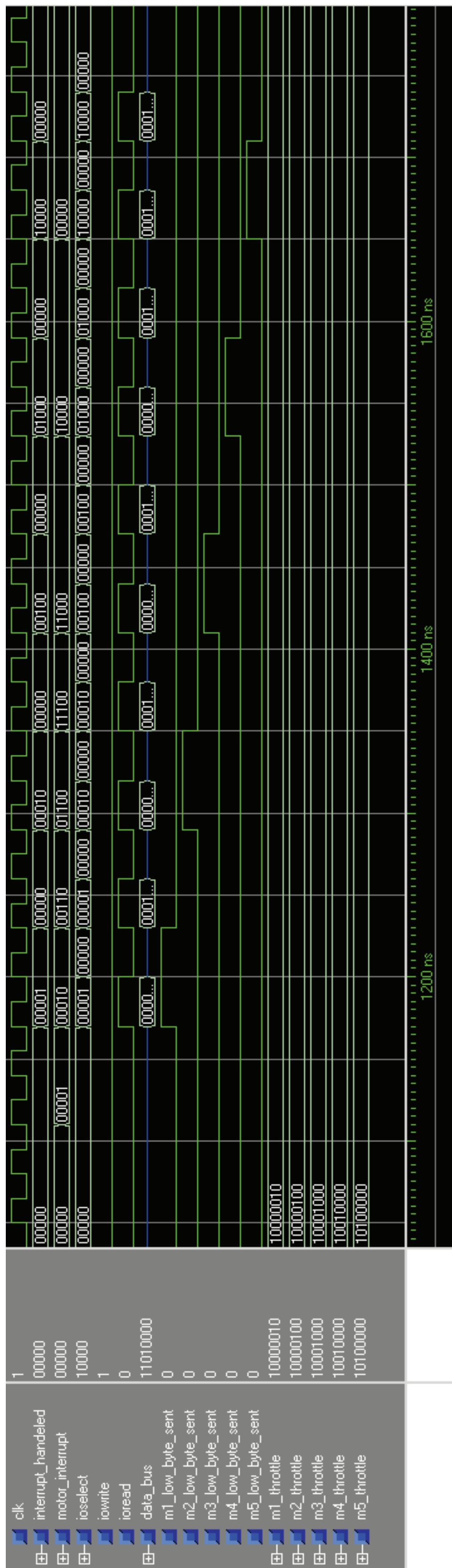
Figur 45

Communicator – Fortsettelse:

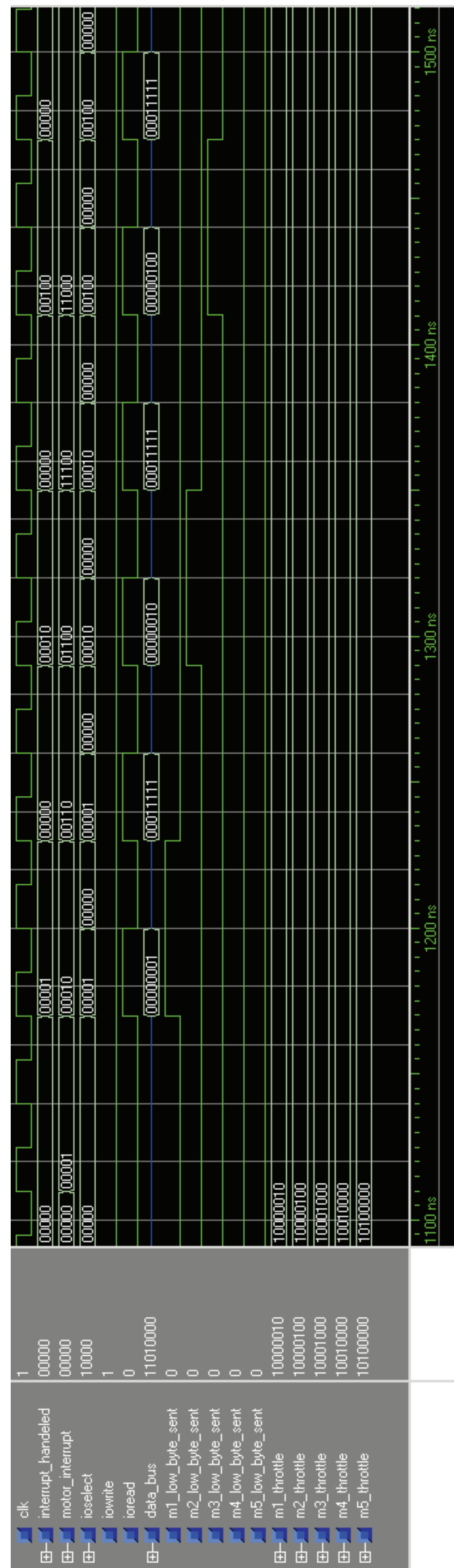
For å teste kretsens evne til sende data ut, samt å generere interrupt, ble fem fiktive hastigheter påtrykket de inngangene som ellers ville vært koblet til individuelle motorkontrollere. I siste halvdel av figur 44 vises det at *motor_interrupt* får verdi ved omtrent 1100ns. Denne endres videre i forhold til påtrykk av nye data, og utlesning fra testbenken sin side. Videre vises 10 påfølgende dataoverføringer på bussen. Her kan det observeres hvordan IOwrite og mX_low_byte_sent går høy i sammenheng med overføringene.

I figur 46 og 47 er forstørrede varianter av de samme plottene vist. I figur 46 kan det leses av hva verdi IOselect til en hver tid har, mens det i figur 47 også er mulig å lese hva verdi som ligger på bussen under overføringene. På samme måte som i ste, har man påtrykt identifiserbare vektorer, med en avsluttende indeks som skiftes til venstre for hver motorkontroller.

Det er også gjort simuleringer av situasjoner der flere motorkontrollere rapporterer ferske måledata. Den eneste forskjellen som vil være å observere da er på interruptene. Ellers skjer kall og utlesning på akkurat samme vis som i figur 46. Derfor har man altså ikke prioritert plass til plott av begge situasjoner. Man valgte den med hendelser spredt i tid, siden det er den mest sannsynlige driftssituasjonen i et virkelig tilfelle. Det er ikke gjort tester for umiddelbart påfølgende hendeler fra en og samme motorkontroller. Men det vil heller aldri skje i praksis, og ansees således ikke som en relevant problemstilling. Dermed ansees det som verifisert at modulen virker som tilsiktet.



Figur 46



Figur 47

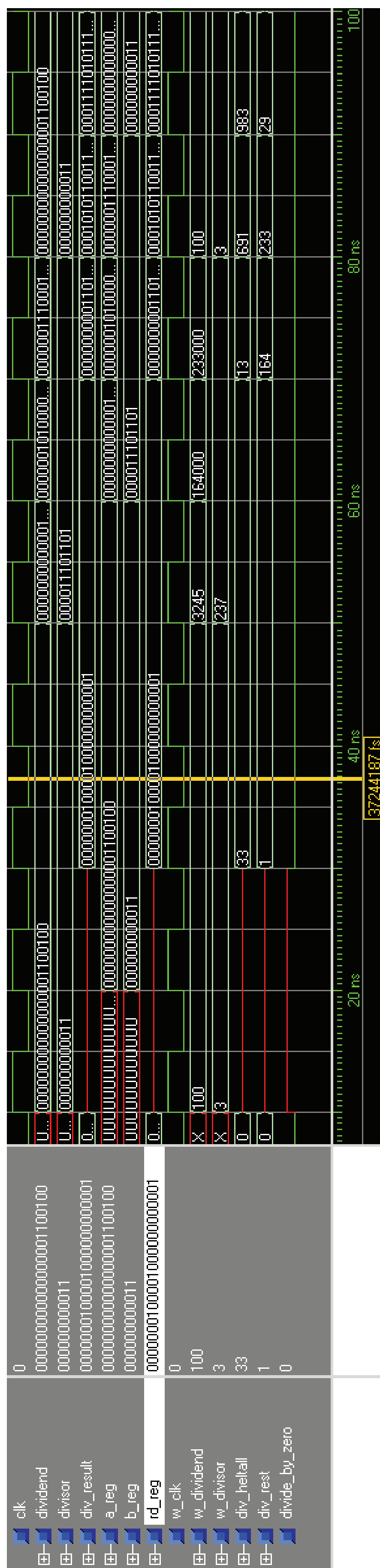
6.2.4 Division_unit:

Hensikten med testing av denne modulen var som nevnt ikke å verifisere dens korrekthet; men å studere timing-diagrammet, for å vite hvordan modulen i praksis fungerer. Det man har forsøkt her er først å utføre en individuell divisjon av 100/3, uten umiddelbart påfølgende påtrykk. Deretter følger en løpende divisjon av 3245/237, og umiddelbare divisjoner av gjenstående rest fra sist divisjon.

Før eksempelet taes videre kan det være nyttig å påpeke at a_reg og b_reg er divisjonsmodulens interne registre for tallene som skal deles. Videre er rd_reg registeret for svaret som vil bli returnert. – For eksempelet 100/3 kan man observere at data først leses inn på neste stigende flanke. Videre viser figuren at svaret er ferdig beregnet ved påfølgende flanke igjen. I samme omgang blir svaret skrevet til utgangen. Det vises her at det tar to klokkesykler fra data påtrykkes inngangen til svaret foreligger på utgangen.

Når man så forsøker å dividere 3245 på 237, og videre løpende påtrykk, så vises det at kretsen presenterer ett nytt resultat på utgangen for hver klokkesykkel. Her er også hver påfølgende divisjon etter den første, resten fra den forrige multiplisert med 1000. Dette gjøres altså i to omganger; slik at man dermed har beregnet at 3245/237 = 13,691983.

Modulen presenterer altså resten fra divisjonen som et heltall, og ikke et flyttall. Noen av strategiene for forbedret målenøyaktighet som vil bli foreslått i kapittel 7 forutsetter tilgjengelig flyttalssvar fra divisjonen. For å realisere dette kreves altså en multiplikasjonsenhet, og logikk for å tilbakekobling av resten fra forrige divisjon.



Figur 48

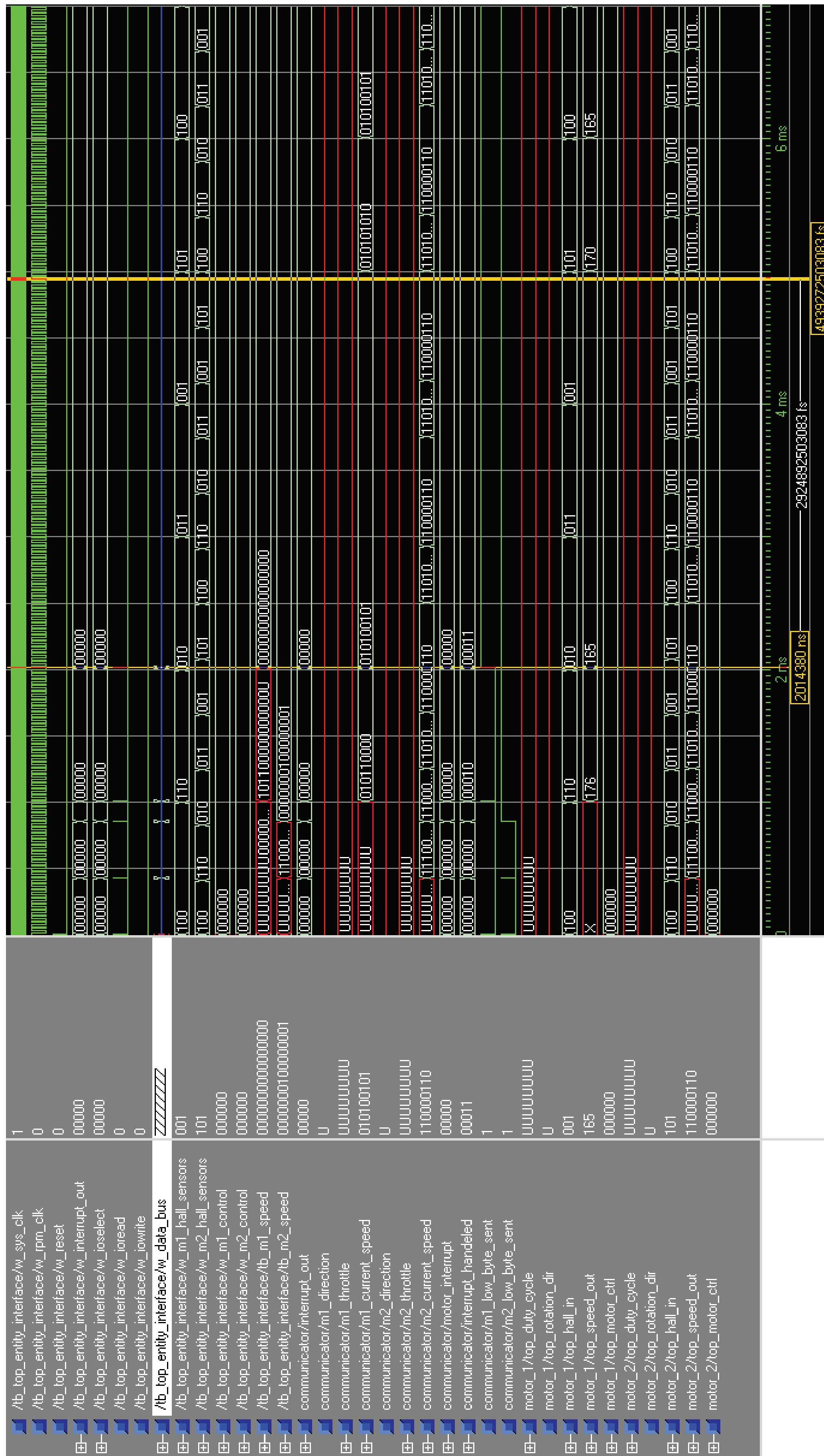
6.2.5 Komplett system:

Testbenken for totalsystemet har mye til felles med testbenken for Communicator. Den mest komplekse delen her er imiteringen av databussen og mikrokontrolleren, som i grove trekk er felles for modulene. Videre påtrykker testbenken sensordata fra fiktive motorer med konstant turtall. Utover å sende initielle pådrag til motoren, så foretar ikke testbenken seg noe før interrupt foreligger. Først da igangsettes utlesing av data. Sånn sett er dette altså en fullverdig test av implementasjonen, med parallell styring av fem motorer under stasjonære forhold.

Dessverre har det ikke lyktes å frembringe korrekte simuleringsresultater med siste versjon av systemet. Hva spesifikt som feiler er det litt vanskelig å sette fingeren på. Men det antas å være enten testbenk eller toppentiteten. Det som i vertfall er å observere av simuleringene er at ingen av pådragene når korrekt frem til sine destinasjoner, og få overføringer av data ut gjennomføres korrekt. Imidlertid fungerer hastighetsberegningen og medfølgende funksjoner utmerket!

Det man har kommet frem til er at mer normal drift kan oppnåes ved å sette signalverdier manuelt rundt i kretsen. Man får for eksempel kommuteringen til å operere korrekt på den måten. Men pwm-modulen starter av uviss årsak ikke opp på den måten. Det antas at denne krever ”events” for å vekkes opp. I figur 49 er et plott fra simuleringene vist. Grunnen til at dette er tatt frem er at man her kan påpeke en konkret feiltilstand. Nemlig at de interne flaggene som benyttes i Communicator, mX_low_byte_sent, ikke går lave igjen etter de har gått høye. Dette er en hendelse som er nødt for å skyldes at testbenken lar være å gjennomføre en overføring, og således noe man har forutsatt at mikrokontrolleren aldri vil gjøre. Feil oppstår også før dette. Men der er det ikke noe konkret å sette fingeren på.

Man har altså ikke lykkes fullt ut i verifiseringen av systemet. Men det antas med relativt stor sikkerhet at feilen ligger i testbenkens forsøk på å emulere bussens virkemåte, og ikke i systemet selv. Eventuelt kan det tenkes at noe har skjedd galt under sammenkoblingen i toppentiteten; skjønt det relativt sett ikke er så mange potensielle feilkilder her. Det skal også sies at man har oppnådd tilnærmet perfekte simuleringsresultater med tidligere og mer manuelle versjoner av testbenken.



Figur 49

6.3 Synteseresultater

Tabell 2 viser en oppsummering av systemets synteseresultater. Både individuelt, og samlet. Man har imidlertid ikke syntetisert Motor_controller sine subbmoduler hver for seg.

Tabell 2 - Synthesis

Modul	Maks frequency	Gate equivalent	LUTs	ibuf	obuf	Accum. instances
Division_unit	49,8 MHz	75	794	37	25	776
Arbiter	63,3 MHz	288	76	197	86	631
Motor_controller	58,1 MHz	81	206	26	49	321
Communicator	87,7 MHz	203	238	54	50	473
Total system	53,5 MHz	940	1989	25	35	2526

Det som her er litt merkelig å observere er at klokkefrekvensen for det samlede systemet er satt høyere enn den laveste individuelle frekvensen i systemet, nemlig Division_unit. Det antas at det også må være denne som hindrer systemet fra å kunne komme enda høyere i frekvens. Hvis man da setter 50 MHz som systemfrekvens, for å ha litt margin, så er dette i alle tilfeller et meget tilfredsstillende resultat i forhold til anvendelsen.

Systemet er i foreliggende versjon delt inn i tre klokkedomener: En for det generelle systemet, en for hastighetstelleren, og en for pwm-modulen. Systemklokken settes innledningsvis til 50 MHz, mens hastighetstelleren i foreliggende implementasjon skal ha 32.768 Hz. Hva PWM-frekvensen bør være er avhengig av hva svitsjefrekvens man ønsker ut, og taes ikke stilling til. Maksimal frekvens for denne er imidlertid er 36,5 MHz. Antakelig kan det være å foretrekke at flere klokkedomener innføres. For eksempel kunne det være gunstig å kjøre Communicator på samme frekvens som AVR, nemlig 25 MHz. Det kan også tenkes at Division_unit burde vært senket. Både fordi den allikevel ville være mer enn kraftig nok, og fordi man da ville spare strøm.

For nærmere resultater henvises det til synteseloggene i det elektroniske vedlegget.

7. Diskusjon og videre arbeid

7.1 Mulige utvidelser og forbedringer av designet

I forbindelse med implementasjon av systemet har hovedfokus vært motorstyringens grunnleggende funksjoner, samt at systemet skal kunne overvåkes og styres fra AVR. Men i utstrekning av dette arbeidet, og i forbindelse med testingen, har naturligvis en del tanker og ideer dukket opp. Dette dreier seg både om mulige forbedringer og ulike former for utvidet funksjonalitet. I dette kapittelet vil altså de viktigste tankene man har gjort seg bli presentert. Hvorvidt forslagene er nyttige eller nødvendige får bli opptil prosjektleder å vurdere.

7.1.1 Mulige forbedring av designet

Før mulige endringer og forbedringer av designet diskuteres, så er det et moment som må trekkes frem. Når det gjelder designets funksjonelle inndeling og struktur så vil det være en vesentlig forskjell på hva som er optimalt med foreliggende design, og hva som vil være optimalt ved eventuell implementering av utvidet funksjonalitet. Hvor man ønsker å ta prosjektet videre er derfor av avgjørende betydning for hva som er optimalt i denne sammenhengen.

Omstrukturering av designet:
Hvis man tar utgangspunkt den funksjonaliteten som ligger i systemet per i dag, og at en forholder seg til FPSLIC som plattform, så ser en allikevel for seg noen ting som kunne vært gjort enda bedre. Det første er en omfordeling av funksjonalitet i motorkontroller. For eksempel gjør ikke Hall_monitor det den først var tenkt å gjøre; nemlig å ta del i kommuteringen ved å sette kommuteringsvinkelen. Denne funksjonen er på ingen måte uaktuell for fremtidig implementasjon. Men det man her tenker på er at det allikevel ikke er nødvendig at overvåkningen av hall-sensorene går via Hall_monitor. Dette kunne godt vært gjort i commutator. Videre kunne commutator med fordel være slått sammen med Commutator_mux. Dersom styring av kommuteringsvinkel skal implementeres, så er antakelig Hall_monitor den mest naturlige plassen å gjøre dette. I essens dreier dette seg om å beregne rotorvinkelen i kjøretid, og å forsinke kommuteringen ut ifra kjennskapen til rotors vinkelhastighet. Derfor er det da like greit å bare sende et enable / disable til commutator som å forsinke sensoravlesningen. Igjen ville man altså stått med tre moduler istedenfor fire. Om den praktiske betydningen hadde vært så stor er heller usikkert. Men kunne kanskje gitt et ryddigere design.

Forbedret kommuteringsforsinkelse:
En annen forbedring, eller rettere sagt utbedring, er å fikse problemet med kommuteringsforsinkelsen som kapittel 6 påpeker. Nemlig det at commutator_mux påtrykker drivtrinnet en puls på én klokkesykkel før forsinkelsen innføres. I følge veileder dreier dette seg om en såpass kort puls at det kun representerer litt varmeutvikling i drivtrinnet. Men det kan allikevel være verdt å se på det, om ikke annet enn for å spare strøm og redusere slitasjen på drivtrinnet.

Utskiftning, forbedring eller egenimplementering av innhentede moduler:
PWM-modulen ser i grove trekk ut til å gjøre nytten. Den eneste grunnen til å bytte ut denne måtte være for å spare areal, hvis man mener det er mulig. Det kunne i første omgang være en mulighet redusere den interne oppløsningen fra 8 til 7 bit, siden kun 7 bit utnyttes fra regulatorens side uansett. Videre bør det undersøkes om gjeldende svitsjefrekvens fra modulen befinner seg i nærheten av båndbredden for drivtrinnet. Hvis så er tilfelle, så vil ikke

drivtrinnet slå seg av / på fort nok, og man oppnår ikke den ønskede spenningsreguleringen. Konsekvensen blir da en motorhastighet som ikke er proporsjonal med pådraget, kanskje også ulineær over det aktive området. Som vist i kapittel 5.3.1 så har man god margin på svitsjefrekvensen i forhold til kravspesifikasjonen. Måten å tilpasse dette på er for eksempel å bruke en klokkedeler utenfor modulen, for å oppnå den ønskede svitsjefrekvensen.

Divisjonsmodulen ser ut til å yte meget tilfredsstillende. Imidlertid er den mye kraftigere enn hva man strengt tatt trenger her. For å realisere divisjon på en enkelt klokkesykkel har man måttet gjøre modulens kjerne kombinatorisk. En kombinatorisk divisjonsenhet skal etter sigende kreve vesentlig større areal enn en tilsvarende sekvensiell variant. Så siden denne allikevel står ledig det meste av tiden, kunne man klart seg med en sekvensiell og tregere divisjonsmodul. Hensikten er i tilfelle kun å redusere arealet. På alle andre områder fungerer kretsen utmerket.

Den foreliggende implementasjonen sender som kjent både dividend og divisor fra motorkontroller for hver divisjon som utføres. Men sånn som systemet fungerer nå, så vil dividend alltid være den samme verdien for alle divisjoner som utføres i relasjon til hastighetsestimeringen. Det ville derfor vært hensiktsmessig å benytte en **hardkodet dividend** i Arbiter, for alle divisjoner relatert til hastighetsestimering. Men divisjonsenheten bør forbli generell, slik at denne på sikt også kan benyttes til andre ting. For eksempel gjøres tilgjengelig for mikrokontrolleren.

Forbedret nøyaktighet av hastighetsmålingen:
Når det gjelder **hastighetsestimeringen** så er det vist i kapittel 5.2.4 at denne har en viss unøyaktighet. Om denne unøyaktigheten er akseptabel, får bli opptil prosjektleder å avgjøre. I alle tilfeller utredes her kilden til unøyaktighetene; og hva som eventuelt kan gjøres for å forbedre beregningene.

For høye hastigheter har vi problemet at telleren relativt sett antar så lave verdier at avrunding til heltals divisor gir betydelig utslag. Siden avrunding her av praktiske årsaker også alltid rundes nedover, så vil feilene man innfører konsekvent være et overestimat. Den gjennomsnittelige målingen vil derfor ikke være så riktig som man kanskje skulle forvente.

Ved høye hastigheter får også klokkeforskyvning betydelige konsekvenser. Ved en konstant høy hastighet vil telleren, som også er divisor i beregningen, da kunne veksle +/- 1. For allerede lave tall vil dette kunne gi en betydelig drifting i målingen, gjerne mellom to nokså forskjellige verdier. Imidlertid er avrunding til heltallssvar ikke av vesentlig betydning her, siden den målte hastigheten i seg representerer såpass store verdier i forhold til enheten RPS.

For lave hastigheter er problemet omvendt. Her vil telleren anta så høye verdier at avrunding til heltals divisor ikke gjør nevneverdig utslag. Heller ikke klokkeforskyvning betyr så mye her. Derimot er hastighetene som skal representeres så små at enheten RPS tar for dårlig vare på presisjonen i målingen, såfremt desimaler ikke inkluderes i beregningen. Feilen skyldes altså primært heltallsavrunding av det endelige svaret; noe som i foreliggende design vil si forkasting av desimaler. Her får man da et konsekvent underestimat av hastigheten, som vi også så i kapittel 5.2.4. Riktignok er ikke en målefeil på 2,5 % å anse som en stor. Men her innfører i alle tilfeller selve tallrepresentasjonen en feil som er 25 ganger større enn feilen metoden og parametrene selv innfører.

Hvis en da ønsker å gjøre disse målingene bedre, så er en nødt for å ta for seg de to endene av skalaen hver for seg. En løsning på problemene med lave hastigheter vil ikke påvirke unøyaktigheten for de høyeste hastighetene nevneverdig, og omvendt. Dermed kan en også velge om en har behov for å forbedre målenøyaktigheten i begge ender av hastighetsområdet eller ikke. I alle tilfeller er noen måter å bedre nøyaktigheten på presentert her:

For å ta for oss **høye hastigheter** først, så er løsningen her å gjøre endringer slik at telleren teller lengre før divisjonen utføres. Da minsker man både virkningen av klokkeforskyvning og heltalsavrunding av divisor. To måter å gjøre dette på er å øke tellerfrekvensen, eller å telle over lengre intervaller enn bare én enkelt hall-sone. I begge tilfeller så innebærer det at behov for større registre, og en økning av ordbredden til divisjonsenheten. Større registre utgjør ingen stor forskjell her. Men å utføre divisjon på større tall kan fort representere en vesentlig kostnadsøkning i form av logiske gates i hardware. – Kapittel 5.3.2 forteller at dividend med nåværende implementasjon vil være maksimalt 12 bit, mens divisor aldri vil overstige 9 bit for de aktuelle hastighetene. Dermed har man muligheten til å øke telleren med 3 bit før divisjonsenheten må skaleres opp. Hvis vi tar utgangspunkt i den maksimale verdien telleren i følge kapittel 5.2.4 vil få for de aktuelle hastighetene, så representerer dette følgende forhold:

(0.22)
$$\frac{2^{12}}{410} = \frac{4096}{410} \approx 10$$

Ved å maksimere divisor til 12 bitt kan man altså telle over et 10 ganger lengre intervall før divisjonskretsen må utvides. Men før man innfører telling over flere hall-soner, så er det en ting man må tenke på. Som beskrevet i kapittel 5.3.2 er den konstante dividenden allerede dividert med 6 for å redusere bredden av denne. Det man samtidig oppnådde med dette var å gjøre målingen relativ til hele runder istedenfor hall-soner. Siden en full rotasjon består av 6 hall-soner, så ville nok det umiddelbart mest intuitive vært å øke tellerintervallet til nettopp en full rotasjon. Men hvis man heller velger å øke tellerintervallet med en toerpotens som faktor, så vil dette snart vise seg å gjøre saken meget enklere. La oss si at man velger å telle over 8 hall-soner, siden 8 er den største toerpotensen under 10. Vi kaller den nye telleren "8counter" slik at det blir lettere skille. For å beholde enheten RPS som representasjonsform, vil regnestykket måtte se ut som følger:

(0.23)
$$\frac{f_{counter}/6}{\frac{8counter}{8}} = \frac{f_{counter}/6}{8counter} \times 8$$

Ergo kan vi bare utføre divisjonen med den oppskalerte telleren, så lenge svaret etterpå multipliseres med 8. Grunnen til at det var så lurt å velge en toerpotens er fordi multiplikasjon med toerpotenser som kjent kan utføres med et høyreskift; noe som går meget raskt i hardware. Den gjennomsnittelige telleren blir da:

(0.24)
$$\frac{32.768}{2.400} \times 8 = 109,227$$

Hvis vi da utfører beregningen fra formel 0.23 for verdiene 109 og 110 så vil algoritmen gi oss følgende svar:

$$\left[\frac{f_{counter}/6}{8counter}\right]\times 8=\left[\frac{5461}{109}\right]\times 8=[50{,}101]\times 8\Rightarrow 50\times 8=400\times 60=24.000[RPM] \quad (0.25)$$

$$\left[\frac{f_{counter}/6}{8counter}\right]\times 8=\left[\frac{5461}{110}\right]\times 8=[49{,}646]\times 8\Rightarrow 49\times 8=392\times 60=23.520[RPM] \quad (0.26)$$

Dette gir oss en maksimal målefeil på 2 % og en gjennomsnittelig feil på 1 %, noe som må sies å være meget bra. Men det kan her tenkes at vi blir litt lurt av at avrundingene er til vår fordel; og at drifting i måleverdiene grunnet klokkeforskyvning vil bli merkbart. For å redusere dette problemet har en tillatt seg å tenke enda litt lengre.

Som nevnt tidligere i rapporten er resten fra divisjonen bare blitt umiddelbart forkastet i foreliggende design. Som det fremkommer av kapittel 6, har man igjennom testing funnet at divisjonskretsen presenterer resten fra divisjonen som et heltall. Her kan man se for seg å implementere en modul som kjører resten til ny divisjon i et gitt antall iterasjoner. På den måten kan man generere flyttallssvar ut ifra divisjonen. Hvis man legger dette til grunn, slik at svaret med desimaler multipliseres med 8, så kan nøyaktigheten økes ytterligere. Hvis man for eksempel multipliserer med 1000, og kjører kun en ekstra divisjon, så vil det gi følgende resultater:

$$Counter=109\Rightarrow Speed=400{,}807\approx 400\times 60=24.000[RPM] \quad (0.27)$$

$$Counter=110\Rightarrow Speed=397{,}164\approx 397\times 60=23.820[RPM] \quad (0.28)$$

Dette gir en maksimal målefeil på 0,75 %, og en gjennomsnittelig målefeil på 0,38 %. I tillegg vil det sannsynligvis redusere driftingen i målingene, samt at de vil ligge nærere normalfordelt rundt reell hastighet.

For **lave hastigheten** er imidlertid historien en del kortere. Her har tidligere utregninger vist at målenøyaktigheten, inklusiv avrundingsfeilen av divisor, i snitt utgjør 0,21 %. Den totale unøyaktigheten er allikevel 2,5 %. Dette skyldes da i all hovedsak avrunding av det endelige svaret. Problemet er her som nevnt at enheten RPS er for liten for å utrykke så lave verdier som kravspesifikasjonen ber om. Den eneste muligheten man da har, hvis man skal unngå å øke divisjonsmodulens bredde, er å skalere opp svaret direkte fra divisjonen. Dette forutsetter igjen at svaret fra divisjonsmodulen enkelt kan leses ut som, eller konverteres til et flyttall. Allerede består hastighetsvektoren som oversendes AVR av 16 bit. Derfor kunne svaret vært oppskalert med en maksimal faktor på 163 før man fikk overflyt. En så høy faktor er imidlertid ikke hensiktsmessig på noen måte. Igjen bør man i tilfelle holde seg til en toerpotens, for å lette operasjonen for hardwaren. En får altså ikke RPM direkte, men kunne fått "runder per 64 sekund" ganske enkelt. Eneste betydningen dette har er i tilfelle for forståelsen. Regulatoren bryr seg naturligvis ikke om enheten, så lenge målingen fremdeles er lineær. Nedenfor er vinningen av den foreslåtte endringen beregnet. For å lette sammenlikningen med reell hastighet har man benyttet faktoren 60 istedenfor 64. Det gjøres samtidig oppmerksom på at tiltakene foreslått for høye hastigheter her ikke er tatt med; skjønt det heller ikke skulle ha nevneverdig betydning for resultatet.

Kapittel 5.3.2 gir oss at den gjennomsnittlige telleren for 800 RPM vil være 409,6 for 32.768 Hz. Med den forkortede dividend gir dette følgende eksakte svar, dersom divisjon med desimaler ble utført:

(0.29) $$\frac{5461}{409,6}\times 60 = 799,951 \Rightarrow 0,006\%$$

Dette er ment bare som et sammenligningsgrunnlag for hva feil vi har innført med å dele tellerfrekvensen på 6 før utregningen gjennomføres. Våre faktiske målinger blir da:

(0.30) $$\frac{5461}{409}\times 60 = 801,125 \approx 801 \Rightarrow 0,125\%$$

(0.31) $$\frac{5461}{410}\times 60 = 799,171 \approx 799 \Rightarrow 0,125\%$$

Den gjennomsnittlige målefeilen blir da nettopp 0,125 %. Dette er den samme nøaktigheten man har vist at målemetoden i seg selv gir. Dermed kan vi også konkludere med at målenøyaktigheten er maksimert for den gitte tellerfrekvensen.

Tiltaket med å øke tellerintervallet vil naturligvis føre til sjeldnere oppdatering av hastigheten tilbake til regulator. For høye hastigheter (24.000 RPM / 400 RPS) blir da oppdateringsraten følgende:

(0.32) $$\frac{400[1/s]\times 6}{8} = 300[1/s]$$

Hastigheten vil altså oppdateres 300 ganger per sekund i FPGA; noe som i alle tilfeller er langt mer enn man har behov for. For de laveste hastighetene (800 RPM / 13,65 RPS) får vi:

(0.33) $$\frac{13,65[1/s]\times 6}{8} = 10,24[1/s]$$

Det antas at dette også vil være tilstrekkelig. Men denne avgjørelsen overlates til prosjektleder å ta. Hvis man skiller høye og lave hastigheter har man altså potensielt 8 ganger høyere oppdateringsfrekvens tilgjengelig. Så vil det videre være avhengig av filterkonstanten i støyfilteret hvor ofte man effektivt sender oppdateringer.

Avrunding på grunnlag av flyttall:
En feilkilde som påpekes gjentatte ganger i ulike sammenhenger er avrunding. En ting som er litt uheldig her er at alle avrundingene foretaes nedover, slik at alle feilestimat som skyldes avrunding vil helle til den ene siden for det korrekte svaret. Avrunding av divisor er noe man ikke kan gjøre stort med her, siden dette dreier seg om en binær teller. Men avrundingsfeilen avslutningsvis kan sannsynligvis påvirkes. Dersom man har flyttallssvaret fra beregningen tilgjengelig, så bør det være en enkel sak å implementere avrunding basert på første desimal. På den måten ville man i det minste få en gjennomsnittelig feil som ligger nærmere realiteten. Altså et mer normalfordelt standardavvik.

7.1.2 Tiltak for øket funksjonalitet

Generering av flyttall fra divisjonen:
Etter at deler av dette kapittelet ble skrevet, har man funnet at divisjonsenheten presenterer resten fra divisjonen som et heltall, ikke som desimaler. For så vidt ikke så overraskende. I alle tilfeller betyr det at flyttall kan genereres, hvis man innfører en multiplikasjonsenhet i systemet. Denne kan for eksempel være på 10x10 bitt, slik at resten fra divisjonen kan multipliseres med 1000, for så å utføre en ny divisjon av denne. For hver gang man utfører dette, så får man altså tre nye desimaler til et eventuelt flyttall.

Setting av støyfilterparameter fra mikrokontrolleren:
Som forklart i kapittel 5.2.4 har man implementert et støyfilter, eller en terskel for hvor stor hastighetsendring som skal til før interrupt genereres. I nåværende design settes denne terskelverdien som en konstant ved syntetisering. Imidlertid kan en se for seg at denne parameteren kunne vært satt fra mikrokontrolleren i kjøretid, og gjerne relativ til operasjonsmodus eller hastighet. – For høye hastigheter vil man naturlig ha mer målestøy, og oppleve mer drifting enn man vil ved lave hastigheter. Det er derfor sannsynlig at man kunne oppnå en mer optimal utnyttelse av hastighetsmålingen for ulike hastigheter med en slik funksjon tilgjengelig; og det uten at det ville koste mikrokontrolleren nevneverdig CPU-tid.

Vinkelestimering og regulering av kommuteringsvinkel:
En ting som opprinnelig var planlagt som ekstra funksjon, men som man ikke fikk tid til å implementere, er vinkelestimering og styring av kommuteringsvinkel. Og da i form av en parameter som mikrokontrolleren kan sette i kjøretid. Mer korrekt er det kanskje å omtale funksjonen som forskyvning av kommuteringsvinkel, siden metoden ikke gir mulighet for fremskyndet kommutering. Nedenfor følger en beskrivelse av metoden:

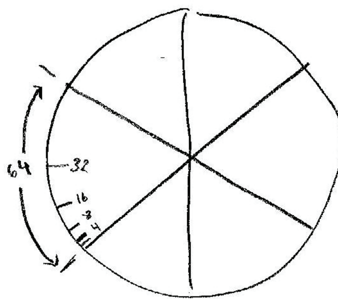
Grunnprinsippet for metoden er det samme som ligger bak hastighetsestimeringen. Ved å benytte en teller med kjent frekvens til å måle passeringstiden for en kommuteringssone under drift, så kan vi med hjemmel i motorens treghetsmoment anta at vi også kjenner den tilnærmede passeringstiden for påfølgende kommuteringssone. Med utgangspunkt i denne passeringstiden kan man da estimere gjeldende vinkelhastigheten for motoren; som igjen kan danne grunnlaget for en forsinkelse relativt til vinkel heller enn tid. Metoden foretrekker en tilgjengelig multiplikasjonsenhet for maksimal oppløsning, men kan også realiseres med skiftoperasjoner. Dette innebærer ikke tap av nøyaktighet, men gir redusert disponibel oppløsning for hva forsinkelse man ønsker å innføre. En tilgjengelig divisjonsenhet vil gi økt fleksibilitet ved valg av vinkeloppløsning, men er overflødig hvis man her velger en toerpotens i oppøsning. Da gjøres divisjonene raskere med skiftoperasjoner. Den beste løsningen med et minimum av ekstra hardware er antakelig divisjon ved skiftoperasjon, og multiplikasjon ved hjelp av multiplikasjonsenhet.

Først definerer man altså hvilken oppløsning / nøyaktighet man ønsker på beregning av vinkelfarten. Her må man velge en toerpotens, skal man klare seg uten en divisjonsenhet i kretsen. Den valgte vinkeloppløsningen multiplisert med maksimal kommuteringsfrekvens gir frekvensen til det fiktive signalet som skal samples. Tellerfrekvensen må da velges vesentlig høyere enn dette, for å minimere betydningen av klokkeforskyvning. Her er det ikke nødvendig å velge en toerpotens; noe som kanskje ikke vil være tilgjengelig i krystaller av aktuell størrelse. Man realiserer så en teller basert på den valgte frekvensen, som da skal telle antall klokkeslag mellom to kommuteringsskift eller hall-soner. Når en full kommuteringssone er overløpt, divideres tellerverdien man fikk på valgt vinkeloppløsning. Av den grunn bør altså vinkeloppløsningen være en toerpotens. Tallet man får etter divisjonen

er det antall klokkeslag motoren trenger for å tilbakelegge en vinkelenhet ved gjeldende rotasjonshastighet, eller invers vinkelhastighet om man vil. En vinkelenhet er da relativt til oppløsningen man valgte, og ikke nødvendigvis grader.

Videre behøver rutinen å få inn faktoren for hvor mange vinkelenheter man ønsker å forskyve kommuteringen med. – Det er denne faktoren som da kunne vært påtrykket av mikrokontrolleren i kjøretid. – Den inverse vinkelhastigheten multipliseres med ønsket antall vinkelenheter, og gir det antall klokkesykler neste kommutering må forsinkes med for å oppnå ønsket forskyvningsvinkel.

Hvis man hadde valgt en vinkeloppløsning på 64, ville forsinkelsen være gitt av forholdet X/64, der X er faktoren gitt av mikrokontrolleren. Med tilgjengelig multiplikasjonsenhet kunne X være en vilkårlig verdi mellom 0 og 63. Multiplikasjon ved skiftoperasjon gir derimot kun muligheten til å benytte en toerpotens faktor som forsinkelse. Altså 2/64, 4/64, 8/64, 16/64 eller 32/64. I figur 50 er sistnevnte tilfelle illustrert.



Figur 50

Forskyvningstelleren, for å gi den et navn, benyttes nå som målverdi for en ny tellerprosess. Denne prosessen må ha mulighet til å overstyre kommuteringsenheten på en sånn måte at kommuteringsskift kan utsettes. For hver nye kommuteringssone, deaktiveres kommuteringsenheten samtidig som telleren startes fra null. Først når telleren har nådd målverdien, så gjennomføres kommuteringsskiftet som kommuteringsenheten har liggende klart. Man har nå effektivt forsinket den ordinære kommuteringen med det ønskede antallet vinkelenheter. – Denne prosessen kan gjerne inkludere kompensasjon for tiden det tar å beregne nødvendig forsinkelse. Siden denne utregningstiden er konstant og kjent på forhånd, er det tilstrekkelig å la den gå til fratrekke fra den estimerte tidsforsinkelsen.

Som implisert under veis er den antatt beste løsningen å benytte skiftoperasjoner til divisjon, og en dedikert krets til multiplikasjon. Vinkeloppløsning basert på en toerpotens er i praksis ikke noe offer, siden den fulle oppløsningen allikevel kan utnyttes ved inkludering av en multiplikasjonsenhet. Denne anbefales det altså at man tar med, om ikke annet enn for å evaluere virkningen av ulike kommuteringsvinkler. Så kan heller oppløsningen reduseres til ett mer hensiktsmessig område i en eventuell endelig implementasjon. Grunnen til at man ikke har valgt å basere seg på den eksisterende divisjonsenheten i kretsen er at ordbredden denne tilbyr sannsynligvis ikke ville gitt tilfredsstillende nøyaktighet.

Estimering av protesens globale posisjon / vinkel:

Dette blir igjen mye av det samme som estimering av hastighet og vinkelhastighet. Men basert på protesens utveksling på 1:800, er det her ikke nødvendig med noe i nærheten av den samme nøyaktigheten for å kunne oppfylle kravspesifikasjonen på 10 bits oppløsning per

runde, som vil si 1024 punkter. Her er det mer enn godt nok å vite hvilken av de 6 kommuteringssonene motoren befinner seg innenfor, som multiplisert med utvekslingen gir 4.800 punkter per omdreining. Dette gir altså en oppløsning på mer enn 12 bit ($2^{12} = 4096$).

Eneste aber her, noe som også antas å kunne bli funksjonens bane, er at telleren må ha en referanse å telle fra. Denne referansen kunne vært gitt av vinkelmålingen som i nåværende prototyp utføres i software, en måling som ikke krever kalibrering av noe slag. Men siden denne målingen allerede tilfredsstiller kravene, er ikke nevnte funksjon noe umiddelbart behov. Først hvis posisjonsregulering skulle implementeres i hardware kunne denne vært meget nyttig, for å redusere målings og kommunikasjonslasten på mikrokontrolleren. Det hadde da vært tilstrekkelig å sende en enkelt referansemåling ved oppstart, samt ny referanse de gangene motorstyringen hadde vært satt i dvale. Hvis posisjonsregulering blir aktuelt å implementere i FPGA, så ansees det som direkte uhensiktsmessig å utelate denne funksjonen.

Hastighetsbasert tilstandsmaskin:
Som det har vært nevnt ved et par anledninger tidligere i rapporten, kan det i noen sammenhenger være fordelaktig å benytte ulike parametere ved forskjellige hastigheter. Dette kunne for eksempel være i forbindelse med hastighetsestimeringen. Det kunne også være aktuelt å benytte ulike kommuteringsstrategier for ulike hastigheter. I alle tilfeller vil slike anvendelser kreve en tilstandsmaskin som opererer på grunnlag av forhåndsdefinerte hastighetsintervall. Sånn sett er dette kun et strukturelt forslag, for å unngå gjentakende funksjonalitet rundt i kretsen.

Divisjonsenheten gjort tilgjengelig for mikrokontrolleren:
En ide som undertegnede har sett for seg kan forenkle reguleringen er å tilby mikrokontrolleren direkte tilgang til divisjonsenheten. Dette kunne enkelt vært innlemmet i kommunikasjonsenheten og arbiter, uten behov for å endre motorstyringen. Så fikk det bli opptil programvareutvikler hva funksjonen skulle brukes til. Det antas at svaret på divisjonen kunne vært returnert til mikrokontrolleren på 6 klokkesykler á klokkefrekvensen i FPGA, som antakelig vil være omtrent 50 MHz, mot mikrokontrollerens 25 MHz. Altså ca tre klokkesykler effektivt. Til sammenligning er det oppgitt i en applikasjonsnote fra Atmel at FPSLIC vil trenge i størrelsesorden 200 klokkesykler for å utføre en 16/16 divisjon, ved bruk av deres assembly-kode. Det eksakte tallet er litt avhengig av om rutinen er optimalisert for kodestørrelse eller hastighet, samt om tallene er SIGNED eller UNSIGNED.

7.1.3 Areal- og strømbesparende tiltak

Når man skal diskutere tiltak for strømsparing i motorstyringen, så har man to vidt forskjellige kontekster å forholde seg til. Den ene er når motoren går; den andre er når den i kortere eller lengre perioder står i ro. Hvis motoren går, så vil strømtrekkene i kretsen uansett være svært små i sammenligning med motorstrømmen; og man altså ikke all verdens å hente her. Hvis motoren derimot står i ro, så har man et langt mer betydelig potensial for prosentvis reduksjon i strømforbruk. Det er altså her man bør legge hovedinnsatsen i første omgang.

Det første som er naturlig å tenke på, uavhengig av kontekst, er at det primært er signaltransisjoner som koster strøm. Det betyr at man vil kunne spare en hel del strøm i det jevne ved å optimalisere kretsen mot **lavest mulig klokkefrekvens**. Men hvis man går til skritt som økt parallellisering for å senke klokkefrekvensen, så må man samtidig ta med i betraktningen at dette innebærer flere logiske blokker som skal skifte verdi per klokkeslag. Sånn sett vil også **kompakt design** i seg selv virke strømbesparende. Samtidig vil kompakt

design bety redusert logisk areal, som igjen kan gi lavere kretskortareal og komponentkostnad.

Videre vil et annet åpenbart tiltak være å **implementere dvalemoduls** for hele eller deler av kretsen, for bruk i perioder hvor disse av ulike grunner ikke er aktive. Dette er også tiltak som kan være relevante for begge kontekster, dersom man ønsker sparetiltak også når motoren er i drift. Synkrone moduler kan enkelt settes i dvalemodus ved å deaktivere klokkesignalet til modulen. For kombinatoriske moduler vil man måtte innføre et enable / disable. Dette kan også være relevant for synkrone moduler, dersom man ønsker at kun en del av en modul skal slåes av. For eksempel hvis man har behov for at en ekstern tilstand overvåkes mens man befinner seg i dvalemodus. Dette kunne også være nyttig for å få kretsen til å vekke opp seg selv igjen på grunnlag av en relevant hendelse.

Foruten funksjoner for dvalemodus i seg selv, så vil det sannsynligvis også være en del å hente på å implementere **gode rutiner for entring av dvalemodus**. Så lenge strømmen til kretsen ikke kuttes, så antas det at oppstartstiden fra dvalemodus ikke vil være mer enn tiden til neste positive klokkeflanke. Sånn sett kan det være aktuelt å ha en nokså lav terskel for når man setter enkelte moduler i dvale.

Et litt mer generelt tiltak, som man også har hatt i tankene under implementasjon av foreliggende løsning, er å **tilstrebe lavest mulig bittbredde** i komplekse blokker. Etter sigende skal dette være spesielt viktig for en rekke kombinatoriske strukturer. For eksempel har vi at nødvendig areal for å realisere en kombinatorisk divisjonsmodul er eksponentielt økende med ordbredden på dividend og divisor. Dette kan være verdt å ta med, for eksempel ved implementering av styrt kommuteringsvinkel. Det kan også være nyttig å kikke på muligheten for å **skifte ut den kombinatoriske divisjonsmodulen** med en tilsvarende sekvensiell variant. Det forventes at en slik vil kunne spare en del areal, og allikevel være tilstrekkelig rask for anvendelsen.

Dersom areal blir en virkelig knapp ressurs kan en eventuelt spørre seg om en vil ta seg råd til divisjon. Hastigheten kan estimeres også uten denne; men vil da kun foreligge på formatet [tid per rotasjon] i motsetning til [rotasjoner per tid] som man har nå. Dette vil gi en ulineær regulator, men skal i følge veileder være håndterbart. Så blir det en vurdering av hva som er viktigst: Lineær regulator eller lite fysisk areal.

7.1.4 Design for øket skalerbarhet

Et generelt tiltak man har sett for seg at kunne gjøre systemet enda mer skalerbart, og koden mer strukturert, er bruk av for-generate-strukturer. Hvis man hadde omstrukturert gjentakende rutiner til å være selvstendige arkitekturer, kunne disse vært rullet ut i ønsket antall med for-generate. Det kan imidlertid tenkes at dette vil komplisere designet en del, og dermed innføre flere potensielle feil. Siden undertegnede i alle tilfeller ikke har særlig erfaring med bruk av slike strukturer, så har man latt det være med å påpeke muligheten her.

7.2 Implementering av CAN-bus

Helt fra starten var man klar over at det ville bli arbeidskrevende og utfordrende å gjennomføre en grundig utredning av dette punktet. Derfor ble undertegnede enig med veileder om å heller bare gjøre en praktisk vurdering, og se hva en kunne finne ut av andres arbeid. Med andre ord var håpet at det skulle være mulig å finne en referanseløsning på nett. Det man primært ønsket å finne ut var om en slik implementasjon var realistisk å få plass til i en relativt liten FPGA, og var sånn sett ikke avhengig av å finne en 100 % ferdig

implementasjon. I forhold til en eventuell egen implementasjon eller videreutvikling av kontrolleren var det også av interesse å ha en viss formening om kompleksiteten i et slikt design. Imidlertid viste det seg etter hvert at prosjektleder Øyvind Stavdahl ikke ville være villige til å sette i gang egen utvikling av en slik modul. Med andre ord var man prisgitt tilgjengelighet av åpen kildekode.

Å finne slike moduler på nett innebærer et par utfordringer. Den første utfordringen er at Google gir relativt få relevante linker. Man er først nødt for å finne frem til åpne forum for utviklere, eller sider relatert til utdanning. Når man omsider kanskje har funnet noen slike sider, så er man allikevel ikke helt i mål. Et problem er at en del firma legger ut halvferdige løsninger med den hensikt å kapre kunder. Et annet problem, som kanskje er det mest relevante, er at de IP-modulene som ligger ute i mer eller mindre ferdig tilstand som oftest er dårlig dokumentert, både når det gjelder virkemåte og test / verifikasjon. Uten dokumentasjon på at modulen virker er det risikofylt å basere seg på den; og uten innsikt i modulens virkemåte blir det komplisert å sette opp gode tester. Ofte vil det være enklere å designe modulen selv enn å sette seg inn i og fullføre slike ufullstendige implementasjoner.

OpenCores.org er antakelig en av de mer kjente websidene av dette slaget. Her ligger et stort utvalg av IP-cores tilgjengelig, delt av personer som i mange tilfeller later til å være erfarne utviklere. Men mange av prosjektene lider av å aldri bli fullført helt; og at utviklerne sjelden tar seg tid til å dokumentere designene sine i særlig grad. Heldigvis har det vist seg at flere av utviklerne har vært mulig å komme i kontakt med via e-post, skjønt dette ikke kan kompensere for den mangelfulle dokumentasjonen.

Det var altså på OpenCores.org det lyktes undertegnede å finne både divisjonsmodulen og en CAN-kontroller. Divisjonsmodulen har vist seg å fungere godt sammen med designet; og har vært til stor hjelp i arbeidet. CAN-kontrolleren var derimot ikke like enkel å ta i bruk. Denne viste seg nemlig å kun være tilgjengelig i Verilog. En VHDL variant har vært tilgjengelig tidligere; men har blitt fjernet da den ikke var 100 % kompatibel med standarden. Eventuelt bare ikke ferdig. Selv om det ikke var noe definert ønske om å innlemme slik funksjonalitet i det nåværende designet, så står ikke en Verilog implementasjon like høyt i kurs. Undertegnede er ikke kjent med om det finnes verktøy for parallell bruk av VHDL og Verilog; men antar at dette uansett ikke ville vært noen god løsning. På grunn av mangelfull erfaring med Verilog kan ikke undertegnede uttale seg om eventuelle muligheter for å oversette modulen til VHDL; skjønt dette bør være mulig.

I den lille dokumentasjonen som fulgte CAN-kontrolleren er allikevel et par relevante detaljer nevnt. Det viktigste er kanskje at forfatter har syntetisert modulen til å kreve 12k gates, eller 930 flip-flops. Det er også oppgitt at modulen er testet på hardware, og verifisert med ”Bosch VHDL Reference System”. Dessverre har det av uviss grunn ikke lyktes undertegnede å syntetisere modulen i Leonardo Spectrum.

Konklusjonen på dette feltet blir derfor at alternativet med realisering av CAN-bus i FPGA ikke er umiddelbart tilgjengelig. Selv om det fantes vilje til å sette i gang egen utvikling av en slik modul, så ville det antakelig være en enklere jobb å oversette undertegnedes design til Verilog enn å skrive CAN-kontrolleren til VHDL. Med denne rapporten som grunnlag burde dette ikke være en uoverkommelig oppgave for en utvikler med erfaring i Verilog. Allikevel ville den mangelfulle dokumentasjonen av CAN-kontrolleren være et problem, som også muligens gjør det uforsvarlig å basere seg på den til en profesjonell anvendelse.

Hvis man skal gå ut ifra at CAN-kontrolleren er fullt kompatibel med standarden, og implementert på en hensiktsmessig måte, så kan allikevel det oppgitte arealet si en del. 12k gates er spiselig i den versjonen av FPSLIC som sitter på utviklingskortet. Denne har til sammen 40k gates; og antas å være romslig til anvendelsen. Men det kan dessverre se ut som denne brikken ikke er tilgjengelig i tilstrekkelig små pakker til å passe anvendelsen. Dersom dette er tilfelle, så betyr det at man må ned på en brikke som kun rommer 10k gates. Da sier det seg selv at CAN-bus aldri ville få plass, selv med redusert kompatibilitet.

7.3 Sett i lys av Skjeltens løsning

Når denne løsningen skal sees i lys av Skjeltens løsning, så er det en del forutsetninger som må avklares først. Det har lenge vært klart at en løsning basert på FPSLIC ikke ville kunne gjøre NRWD til en bedre protese, basert på de funksjonene denne per i dag er tiltenkt. Det vanskeligste å konkurrere på vil helt klart være brikkeareal. Det neste blir funksjonalitet. De to største hindrene her har vært mangel på ADC og CAN-bus. Når det gjelder ADC så er det rimelig å anta at denne på litt lengre sikt vil falle fra. Men først må EMG-sensorer med digitalt grensesnitt utvikles. Vinkelmåling ved avlesing av pottmeter har allerede falt fra, grunnet annen målemetode. Måling av motorstrømmen har også blitt kuttet ut fra nåværende prototyp, grunnet mangel på egnet målemetode. Men inntil videre må man altså forholde seg til at en ekstern ADC vil være nødvendig. At FPSLIC mangler CAN-bus er noe man ikke kommer utenom, da dette er kommunikasjonsprotokollen man jobber for at skal bli industristandard for kybernetiske proteser. – Siden CAN-bus i FPGA ikke er noe umiddelbart tilgjengelig alternativ, så betyr dette at ekstern brikke er nødvendig. Dermed røk vinningen i antall brikker. I tillegg har det vist seg umulig å skaffe til veie FPSLIC i like små pakker som Atmega128CAN (primærbrikken i Skjeltens design). I vertfall ikke før Atmel har presentert sine nyheter for FPSLIC, og trolig ikke da heller. Totalt sett må det sies at den funksjonelle kravspesifikasjonen for NRWD tilsynelatende er skrevet med tanke på Atmega128CAN. Det eneste denne mangler er evnen til å kommutere en BLDC-motor ved siden av alt det andre den skal gjøre. I NRWD kompenseres dette på en fullt ut tilfredsstillende måte med én AtTiny2313. Til tross for at dette utgjør en ekstra brikke i kretsen, så kan ikke FPSLIC konkurrere på fysisk areal. For de nødvendige hastighetene i NRWD yter den også mer enn godt nok. Det har således lenge vært åpenbart at dette prosjektet ikke kunne presentere noen bedre løsning for NRWD enn Skjeltens. Av den grunn ble da også fokus i arbeidet revet litt løs fra NRWD spesifikt, og mot proteseanvendelser litt mer generelt.

Kombinasjonen Atmega128CAN og AtTiny2313 gjør en utmerket figur for styring av én motor. Imidlertid vil ikke løsningen være like gunstig når flere motorer skal styres. Da vil brått brikketallet stige, siden én 2313 er nødvendig per motor i systemet. Da kan en FPGA være midt i blinken. Ikke bare kan denne by på parallell styring av flere motorer i en og samme krets. Den kan også tilby utvidet funksjonalitet for motorene, slik som den implementerte hastighetsestimeringen. Kapittel 7.1 utredet også funksjoner som høyoppløselig styring av kommuteringsvinkel og kjøretids estimering av global vinkel uten å gjøre beslag på CPU-tid. I det hele tatt kunne funksjonaliteten vært betydelig utvidet med en FPGA tilgjengelig. Spørsmålet er i så måte bare om hvilke ekstra funksjoner som kan gjøre en protese bedre. Men FPSLIC visse såpass betydelige mangler når det gjelder funksjonalitet og fysisk størrelse at den dessverre ikke er noen selvsagt erstatter.

I fare for å konkludere alt nå, så skal det sies at Atmega128CAN på mange måter er som skapt for NRWD, og oppfyllelse av protesens funksjonelle krav. Det blir derfor vanskelig å se for seg at en annen løsning skal kunne overgå Skjeltens design, uten på en eller annen måte inkludere Atmega128CAN. Dette vil da bli diskutert videre i kapittel 7.4 – Anbefalinger for det videre arbeidet.

7.4 Anbefalinger for det videre arbeidet.

Det har lenge vært klart at arbeidet ikke ville lede til en anbefaling av verken FPSLIC eller én ren FPGA-basert løsning. Så lenge NRWD er fokus, med sin ene BLDC-motor, så gjør Skjeltens mikrokontrollerbaserte løsning en utmerket jobb. En teknisk mer avansert løsning gir ikke nødvendigvis et bedre sluttprodukt i denne sammenhengen. Den ekstra kompleksiteten et design basert helt eller delvis på FPGA innfører vil neppe kunne rettferdiggjøres før en mer avansert protese er målet for hardwareplattformen. At en mikrokontrollerbasert prototyp allerede foreligger er heller ikke et uvesentlig moment, som gjør at alternative løsninger får en større jobb med å rettferdiggjøre sin eksistens.

Litt av det en opprinnelig håpet å kunne oppnå da dette arbeidet ble satt i gang var å redusere antallet brikker i kretsen, og samtidig kompleksiteten av kretsutlegget. Imidlertid har det vist seg, hovedsakelig igjennom utredningene i [Mossum 2005], at dette vanskelig lar seg gjøre. I beste fall kan man klare seg med tre primnærbrikker ved bruk av FPSLIC, mot Skjeltens to brikker. Siden FPSLIC i tillegg måler større fysisk areal (20x20mm) enn begge Skjeltens mikrokontrollere til sammen (8x8mm + 4x4mm), så skal det godt gjøres å få til en mer kompakt krets basert på brikken. Riktig nok finnes det ulike pakkestørrelser av FPSLIC, den minste på 14x14mm. Men den har da også kun ¼ så mange gates tilgjengelig som den evaluerte brikken. Det er således lite sannsynlig at denne brikken kan benyttes uten å redusere funksjonaliteten.

Til tross for at det er mye som peker mot fortsatt bruk av Skjeltens løsning for NRWD, så betyr det på ingen måte at videre arbeid med en hybrid løsning er bortkastet. Ankepunktet er mest at utvikling av en hybrid plattform vanskelig kan rettferdiggjøres kun tiltenkt NRWD. Hvis utviklingen derimot rettes andre proteser med flere motorer, så kan det også være gunstig å bruke den samme teknologien i en fremtidig utgave av NRWD. Men av hensyn til de funksjonelle og geometriske kravene FPSLIC ikke direkte kan innfri, så ville det antakelig være fornuftig å se i litt andre retninger enn til Atmel. *Det taes her forbehold om eventuell lansering at nye mer egnede utgaver av brikken i fjerde kvartal.*

På grunnlag av argumentene presentert ovenfor, og det faktum at Atmega128CAN oppfyller alle funksjonelle krav unntatt samtidig styring av flere motorer, så blir anbefalingen å beholde denne brikken som primærenhet, men heller i en hybrid løsning. Så bør man heller bytte ut Skjeltens AtTiny2313 med en ren FPGA. Det eneste som bør være på plass da er at brikkene er I/O-kompatible, og støtter samme drivspenning. Dette for å slippe nivåkonvertering og dobbel strømforsyning i kretsen, noe som ville være uhensiktsmessig plasskrevende. Med en slik tilnærming oppnår man at alle funksjonelle krav innfries med kun to primærbrikker i kretsen. Dette antallet vil sannsynligvis heller ikke behøve å øke med flere motorer i systemet, og kan således brukes direkte uavhengig av typen protese. Programvaren utviklet for foreliggende prototyp av NRWD vil også være tilnærmet direkte kompatibel med en slik løsning, såfremt man baserer kommunikasjon mellom AVR og FPGA på SPI, slik som Skjelten har gjort mellom sine to mikrokontrolleren. SPI som soft-core for FPGA er blant annet å finne på OpenCores.org. For ordens skyld skal det sies at denne ikke er studert, og at full kompatibilitet ikke kan garanteres. I alle tilfeller antas implementering av SPI i VHDL å være overkommelig sammenlignet med CAN.

Ved for eksempel å benytte en Xilinx Spartan III vil man få 50k gates disponibelt i en brikke som måler 8x8mm, eller 200k gates på 14x14mm. Den versjonen av FPSLIC dette arbeidet har tatt utgangspunkt i har til sammenligning 40k gates, og måler 20x20mm. Minste fysiske

pakketype av FPSLIC måler i følge databladet 14x14mm, men inneholder da maksimalt 10k gates, og 8 IOselect / Interrupts. – Det samlede arealet av en Atmega128CAN og en Spartan III blir altså litt større enn i Skjeltens løsning. Men to brikker á 8x8mm antas å kunne være akseptabelt også for NRWD.

Sammenlignet med det foreslåtte oppsettet ovenfor så sier det seg selv at Atmel er nødt for å presentere en kraftig oppdatering av FPSLIC-serien for å være konkurransedyktige. Men en annen ting, som neppe kommer til å endre seg selv om Atmel eventuelt lanserer en kraftig oppdatert serie, er at FPSLIC aldri har vært noe prioritert produkt fra deres side. Spesielt ikke i Europa. Til tross for at brikken ble lansert i USA i 2001, så har ikke brikken vært mulig å anskaffe fra norske leverandører før i 2005. Sånn sett så fremstår Xilinx Spartan III som et atskillig høyere prioritert produkt. Xilinx er da i tillegg et firma hvis primære satsningsområde er nettopp FPGA-teknologi; noe man ikke kan si om Atmel. Det er derfor rimelig å anta at både tilgjengelighet, pris, verktøy og support vil være bedre ved å satse på Xilinx. I tillegg har instituttet allerede lisens på nødvendig programvare for å utvikle mot disse brikkene. Alt som trenges er å kjøpe inn et utviklingskort, som koster i størrelsesorden 2000,-.

8. Konklusjon

Arbeidet med denne hovedoppgaven har ledet frem til en i stor grad fungerende prototyp for et styresystem i VHDL. Grunnleggende funksjoner for styring av børsteløs DC-motor og estimering av motorhastighet er verifisert korrekte ved simulering under stasjonære forhold. De ekstra modulene Arbiter og Communicator, som var nødvendig for samtidig styring av fem motorer, er foreløpig bare verifisert på individuell basis. Det har altså ikke lyktes å simulere totalsystemet feilfritt; skjønt man antar at systemet i essens fungerer som tilsiktet.

De problemene man observerer antas å hovedsakelig skyldes testbenkens forsøk på å emulere databussen i FPSLIC. Enkle tester med manuelle påtrykk har nemlig vist atskillig bedre resultater, skjønt dette ikke kan ansees som en fullstendig verifisering. Det antas derfor at man med en moderat innsats på feilsøking i testbenken også bør kunne få verifisert toppentiteten. Man antar også at atskillig bedre resultater ville vært oppnådd ved test på fysisk brikke; skjønt man da måtte lagt litt arbeid i å utvikle testrutiner for mikrokontrolleren.

I tillegg til det implementerte systemet har arbeidet også ledet frem til en del forslag til nyttige funksjoner og mulige forbedringer av systemet. Flere av de er ikke direkte påkrevde tiltak; men er foreslått fordi det kan gi en enda bedre løsning om ønskelig. Så får det bli opptil prosjektleder å vurdere.

Videre har man igjennom arbeidet gjort vurderinger av brikkens egnethet; både for NRWD spesifikt, og for mer generelle anvendelser innen lignende protesesystemer. Her har sammenligningsgrunnlaget i stor grad vært den foreliggende løsningen, basert på Skjeltens arbeider. De manglene ved FPSLIC som ble påpekt i [Mossum 2005] har blitt diskutert, og mulige løsninger utredet. Den viktigste manglene, at brikken ikke støtter CAN-bus, har også blitt en av de avgjørende faktorene for anbefalingen arbeidet har ledet frem til. Man har her kommet frem til at CAN-bus implementert i FPGA foreløpig ikke er noe tilgjengelig alternativ. Andre viktige faktorer er ting som brikkens fysisk størrelse, dokumentasjon og DAK-verktøy.

På grunnlag av disse vurderingene har man konkludert med at FPSLIC i foreliggende utgave ikke er en så egnet brikke som først antatt. Det anbefales derfor at man ikke satser videre på denne brikken, med mindre radikale nyheter presenteres av Atmel i fjerde kvartal. Isteden anbefales en løsning der man beholder Atmega128CAN som primærbrikke, og at man heller benytter en ren FPGA til motorstyring og kjøretids beregninger.

Med en løsning basert på en Atmega128CAN og for eksempel en Xilinx Spartan III ville systemet uten videre kunne oppfylle alle de samme funksjonelle kravene som Skjeltens løsning gjør. I tillegg vil systemet kunne tilby parallell styring av flere motorer, og mulighet for vesentlig utvidet funksjonalitet. Alt dette med bare to primære brikker i kretsen, uavhengig av antall motorer.

Selv om designtiden og kompleksiteten er antatt å være noe større for en slik løsning, så bør ikke kretsarealet bli nevneverdig større. Og når først en slik løsning er utviklet, ville det ikke være noe til hinder for å bruke den både i NRWD og andre proteser. All programvaren som i mellomtiden blir utviklet for NRWD kunne nærmest uendret vært gjenbrukt. Eneste forbehold er at samme busstandard benyttes i begge brikker.

Referanser

[1] Terje Wessel Pettersen – Digital Motorstyring, Hovedoppgave, Institutt for teknisk kybernetikk, NTNU, Januar 1997.

[2] Øyvind Stavdahl, Functional Requirements Specification for the NTNU Revolute Wrist Device (NRWD) v0.1. Report, NTNU, Department of Engineering Cybernetics, 2005 (Finnes som vedlegg til [3])

[3] Datablad for motor **Faulhaber 1628 012B** (aksessert 08.08.2005): http://www.faulhaber-group.com/upload/pdf_metric265_en.pdf

[4] Håkon Skjelten, Innvevd maskinvare for kybernetisk håndledd, Hovedoppgave, Institutt for teknisk kybernetikk, NTNU, 2005.

[5] Datablad for mikrokontroller AT90CAN128 (aksessert 08.08.2005): http://atmel.com/dyn/resources/prod_documents/doc4250.pdf

[6] ATtiny2313 – Datablad (www.atmel.com)

[7] FPSLIC (AT94K40AL) – Datablad (www.atmel.com)

[8] AT40K (AT40K40) – Datablad (www.atmel.com)

[9] Pulse Width Modulation – Appnote for Atmel FPGA (www.atmel.com)

[10] Øyvind Stavdahl. Optimal wrist prosthesis kinematics: Three-dimensional rotation statistics and parameter estimation. PhD thesis, Norwegian University of Science and Technology, 2002

[11] Pål Kastenes – Rapport 42190 Prosjekt – Institutt for teknisk kybernetikk, Januar 1997

[12] Lars E. Norum - Power Electronics (Utdrag)

[13] Trude Forsbak – Konstruksjon av et kontrollerkort for styring av underarmsprotese – Hovedoppgave, Institutt for teknisk kybernetikk, Februar 1998

[14] Trond Waage og Brede Vigdal – Posisjonsservo basert på mikrokontroller, Prosjekt Institutt for teknisk kybernetikk, Mai 1995

[15] Peter J Ashenden – The designers guide to VHDL (2nd edition)

[16] Marius Mossum – FPGA-basert motorstyring for protesehåndledd (2005)

[17] Jan Gunnar Dyrset og Per Øivind Eger – Digital krets for motorstyring realisert med programmerbre komponenter (1995)

[18] Anders Veberg – Lavnivåregulering av protesemotor (1998)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.Numeric_STD.all;

library work;
use work.hall_definitions.all;
use work.definitions.all;

--
***************************************************************************
******************************** --
-- *
-- *    Overvåker hall-sensorene, og sender data videre til kommutatoren.
Asynkront begge deler.
-- *    Dersom det skulle være ønskelig, er det mulig å interferere med
sendingen, feks ved avansert kommutering.
-- *
-- *    Teller forsinkelsen mellom hvert nye signal fra hall-sensorene, og
beregner motorens rotasjonshastighet.
-- *
-- *    Benytter ekstern divisjonskrets for å utføre deler av beregningene.

-- *    Ekstern logikk for ressursdeling av divisjonskretsen er nødvendig
for flere motorer.
-- *
-- *    Ekstern modul for generering av interrupt til AVR, og sending av
data er også nødvendig.
-- *
--
***************************************************************************
******************************** --


--
***************************************************************************
******************************** --
entity hall_monitor_interface is

        Port(
                sys_clk         : in    std_logic;      -- Klokke til andre
                funksjoner.
                rpm_clk         : in    std_logic;      -- Klokke til
                telleren. 32,768 kHz.
                reset           : in    std_logic;      -- Trengs til
                rpm-estimator

                hall_in         : in    std_logic_vector(2 downto 0);
                -- Inngang direkte tilkoblet motorens hall-sensorer.
                hall_out        : out   std_logic_vector(2 downto 0);
                -- Porter signalet ut i henhold til kommuteringsvinkel.
                speed           : out   std_logic_vector(speed_width-1
                downto 0);-- Tallet vil aldri være større enn 13bit. Men
                bør jeg bruke 16bit interface?

                division_req    : out   std_logic;      -- Sender
                forespørsel om å få utført divisjon.
                division_ack    : in    std_logic;      -- Tilgang til
                divisjon innvilget.

                dividend_out    : out   std_logic_vector((data_width*2)-1
                downto 0);      -- Teller
                divisor_out     : out   std_logic_vector((data_width*1)-1
                downto 0);      -- Nevner
                result_in       : in    std_logic_vector((speed_width -1)
                downto 0);      -- Svar         Redusert i Arbiter.

                rpm_counter     : out   std_logic_vector(counter_width-1
                downto 0) := (others => '0')            -- Kun for
                debugging.
        );
```

```
end hall_monitor_interface;
--
**************************************************************************
******************************* --



--
**************************************************************************
******************************* --
architecture behavioral of hall_monitor_interface is

        signal zone time        : std logic vector(counter width-1 downto
        0) := (others => '0');  -- Signal for å poste teller fra en prosess
        til en annen.
begin


-- ****************************************************** --
-- *    Porter hall-signalet videre til kommutator:
-- *    Hadde ikke trengt prosess for dette.
-- *    Men denne blir mer kompleks, så snart vinkel-kommutering er på
plass!
-- ****************************************************** --
stator_dumper : process (hall_in)
begin
        hall out <= hall_in;    -- Porter signalet rett ut igjen, til
        commutator.
end process;
-- ****************************************************** --



-- ****************************************************** --
-- *    RPM conter
-- ****************************************************** --
rotation_counter : process (rpm_clk, reset)

        variable curr count : std logic vector(counter width-1 downto 0) :=
        (others => '0'); -- Veldig usikker på størrelsen ennå.
        variable last count : std_logic_vector(counter_width-1 downto 0) :=
        (others => '0');

        variable curr hall_zone : std logic vector(2 downto 0);-- := "000";
        -- Prøver å droppe initiering, for å fikse feil.
        variable last hall_zone : std logic vector(2 downto 0);-- := "000";
        -- Prøver å droppe initiering, for å fikse feil.
begin

        ----------------------------------------------
        if      reset = '1' then
                curr count := (others => '0');
                last count := (others => '0');
                curr hall zone := "000";
                last hall zone := "000";
        ----------------------------------------------
        elsif   rising_edge(rpm_clk) then              -- (rpm_clk'event
        and clk = '1') then
                ------------------------------------
                curr hall zone := hall in;                     -- Leser av
                hall-inngangen, og gjør om til en intern variabel.
                ------------------------------------
                if      curr hall_zone = last hall_zone then    --
                Prosessen foretar normal telling:
                        ------------------------------
                        curr count := curr_count + 1;          -- Funker
                        dette? Eller burde jeg lage telleren med en int? Er
                        vel best å gjøre det med bit vil jeg tro.
                        rpm counter <= curr count;             -- Bare for
                        å gjøre telleren synlig i testbenk.
                ------------------------------------
```

```
                elsif   curr hall zone /= last hall zone
                and     last hall zone /= "000" then          --
                Prosessen setter igang rpm estimering, og restetes:
                        ------------------------------
                        last hall_zone := curr hall zone;      -- Denne
                        var visst glemt. Seriøs feil!
                        last count := curr count;              -- Tar vare
                        på telleren for videre beregninger.
                        curr count := (others => '0');         -- Klargjør
                        teller for neste periode. Skal jo telle
                        kontinuerlig.
                        curr count := curr count + 1;   -- Lagt til for å
                        bedre algoritmen!!!!
                        zone time <= last count;               -- Sender
                        telleren til prosessen som kaller divisjon.
                --------------------------------------
                elsif   last hall zone = "000" then           --
                Håndterer initialverdien, og unngår feil rpm_estimat pga
                oppstart/reset:
                        ------------------------------
                        last hall zone := curr hall zone;
                --------------------------------------
                --elsif curr hall zone = "000"
                --or    curr hall zone = "111" then null;      --
                Håndterer eventuelle feiltilstander:
                --------------------------------------
                end if;
        ----------------------------------------------
        end if;


end process;
-- ******************************************************* --

-- ******************************************************* --
-- ******************************************************* --
division_requester : process (zone_time, division_ack) is

        variable last_zone time : std_logic_vector(counter_width-1 downto
        0) := (others => '0');
begin
        ----------------------------------------
        if      zone time /= last zone time then
                --------------------------------
                division req    <= '1';
                last_zone_time := zone_time;

        ----------------------------------------
        elsif   division_ack = '1' then

                dividend out <= (others => '0');
                dividend_out(12 downto 0) <= freq_div_6;        -- Har
                allerede delt på 6.
                divisor_out(counter_width-1 downto 0) <= last_zone_time;

                division req <= '0';
        ----------------------------------------
        end if;
end process;
-- ******************************************************* --


-- ******************************************************* --
-- *    Poller resultat-inngangen kontinuerlig.
-- *    Hvis et verdien er ulik sist verdi, så leses denne inn.
-- *
-- *    Et enkelt støyfilter, basert på en konstant er implementert her.
-- *    Hvis endringen kun er gjeldende for de X laveste bitene, gitt av
konstanten, så ignoreres endringen.
-- *
-- *    Hastigheten sendes kun videre hvis endringen er større enn den
definerte grensen.
```

```
-- ******************************************************** --
result_poller : process (result_in) is

        constant null_vector : std_logic_vector(speed_width-1 downto 0) :=
        (others => '0');
        variable last_result_in : std_logic_vector(speed_width-1 downto 0)
        := (others => '0');
begin
        if      result_in(speed_width-1 downto noise_filter) /=
        last_result_in(speed_width-1 downto noise_filter)
        and     result_in /= null_vector then

                speed <= result_in;
                last_result_in := result_in;

        end if;
end process;
-- ******************************************************** --

end behavioral;
--
***************************************************************************
******************************** --



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--
***************************************************************************
******************************** --
-- *
* --
-- *    Modulen inneholder enkle kommuteringsvektorer. Oppslag baseres på
hall-vektorene               * --
-- *
* --
-- *    Modulen bør ikke implementere avansert kommutering. Det bør heller
skje i egne moduler.                * --
-- *
* --
--
***************************************************************************
******************************** --


--
***************************************************************************
************************************* --
entity commutator_interface is

        Port(
                hall_position   : in    std_logic_vector(2 downto 0);   --
                Graykodet soneangivning for rotor.
                rotation_dir    : in    std_logic;                      --
                1 for høyre, 0 for venstre.
                commutation_out : out   std_logic_vector(5 downto 0)    --
                Kobles til drivtrinnet (h-brua).
        );

end entity commutator_interface;
--
***************************************************************************
************************************* --


--
***************************************************************************
************************************* --
```

```
architecture commutator of commutator_interface is

        signal hall position int: std logic vector(2 downto 0);
        signal rotation dir int : std logic;-- := '1'; -- Settes statisk
        inntil videre. Kobler fra porten.
        signal commutation_int  : std_logic_vector(5 downto 0);

begin

-- ********************************************** --
-- Enkeltstående prosesser for signal-oppdateringer:
-- ********************************************** --
hall position int<= hall position;      -- Oppdaterer det interne signalet
commutation_out  <= commutation_int;    -- Porter tabel-oppslaget til
utgangen.
rotation dir int <= rotation dir;
-- ********************************************** --


--
****************************************************************************
--
-- Utfører tabell-oppslag, og porter verdiene til utgangen for styring av
H-bru.
-- Helt direkte kobling her. Tar ingen hensyn til krypstrøm i H-bru.
-- Bør kanskje endres før testing på fysisk drivkrets, så ikke H-brua
brenner.
--
****************************************************************************
--
commutator : commutation_int <=

        -- For rotasjon mot høyre:
        "100001" when hall_position_int = "001" and rotation_dir_int = '1'
        else
        "001010" when hall_position_int = "010" and rotation_dir_int = '1'
        else
        "100010" when hall_position_int = "011" and rotation_dir_int = '1'
        else
        "010100" when hall_position_int = "100" and rotation_dir_int = '1'
        else
        "010001" when hall_position_int = "101" and rotation_dir_int = '1'
        else
        "001100" when hall_position_int = "110" and rotation_dir_int = '1'
        else
        --"000000";

        -- For rotasjon mot venstre: (Tabell-verdier hentet fra Skjelten)
        "001100" when hall_position_int = "001" and rotation_dir_int = '0'
        else
        "010001" when hall_position_int = "010" and rotation_dir_int = '0'
        else
        "010100" when hall_position_int = "011" and rotation_dir_int = '0'
        else
        "100010" when hall_position_int = "100" and rotation_dir_int = '0'
        else
        "001010" when hall_position_int = "101" and rotation_dir_int = '0'
        else
        "100001" when hall_position_int = "110" and rotation_dir_int = '0'
        else
        "000000";
-- ********************************************** --


end architecture commutator;
--
****************************************************************************
************************************* --


library IEEE;
```

```
use IEEE.STD LOGIC 1164.ALL;
use IEEE.STD LOGIC ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.definitions.all;

--
****************************************************************************
******************************** --
-- *
-- *    PWM-signalet multiplekses til korrekt utgang, basert på
kommuteringsvektorene.
-- *    En forsinkelse mellom kommuteringsskift innføres, for å sikre at
kortslutning
-- *    i drivtrinnet ikke forekommer.
-- *
--
****************************************************************************
******************************** --


--
****************************************************************************
************************************* --
entity commutation_mux_interface is
        Port(
                clk             : in std logic;
                reset           : in std logic;
                commutation in  : in std_logic_vector(5 downto 0); -- Mitt
                select-signal.
                pwm in          : in std logic;
                commutation_out : out std_logic_vector(5 downto 0)
        );
end entity commutation_mux_interface;
--
****************************************************************************
************************************* --


--
****************************************************************************
************************************* --
-- Tar inn den kommuterte vektoren fra commutator, og multiplekser PWM'en
med disse.
-- Sørger for at kun en transistor skifter om gangen, for å unngå
krypstrøm.
--
****************************************************************************
************************************* --
architecture commutation_mux of commutation_mux_interface is

        signal apply pwm enable : std logic := '0';
        signal commutation_int  : std logic_vector(5 downto 0); -- En test
        signal pwm_int          : std_logic := '0';

        --
        /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/
        \/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
        --constant off on delay : integer := 1000;      -- Definerer
        kommuteringsforsinkelsen. Viktig!!!             -- Flyttet til
        packageges.vhd
        --
        \/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
        /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/

begin


pwm_int <= pwm_in;
```

```
-- ************************************************** --
-- Avbryter forrige kommutering, og aktiverer neste
-- kommutering etter en definert forsinkelse.
-- ************************************************** --

delay_pwm : process (clk, reset)        is

        variable delay_counter          : integer range 0 to off_on_delay
        := 0;
        variable last commutation_in    : std_logic_vector(5 downto 0) := (
        others => '0');
        variable count_flag             : std_logic := '0';     --
        Telleflagg

begin
------------------------------------------------------
if reset = '1' then                                     -- Plain resetting
av signaler og variable
        ----------------------------------------------
        apply pwm_enable <= '0';                        -- Deaktiverer
        gjeldende kommutering
        delay counter := 0;                             -- Nullstiller
        tellerne
        count flag := '0';                              -- Telleflagg
        last commutation in := "000000";                -- Tror jeg må ha
        med denne her for å unngå at motoren har mulighet for å aldri
        starte etter reset.
------------------------------------------------------
elsif rising edge(clk) then
        ----------------------------------------------
        if commutation in /= last commutation in then   -- Sjekker om ny
        kommutering foreligger fra Commutator.
                --------------------------------------
                apply pwm enable <= '0';                -- Deaktiverer
                forrige kommutering, siden ny kommutering straks skal
                utføres.
                last commutation in := commutation_in;  -- Husker verdien
                til neste eksekvering.
                delay counter := 0;                             -- Nuller
                ut tellerne før ny forsinkelse startes opp.
                count flag := '1';                      -- Flagg settes
                høyt for å indikere at telling pågår.
        ----------------------------------------------
        elsif count_flag = '1' then                     -- Hvis ny
        kommutering ikke foreligger, sjekkes det om telling pågår.
                --------------------------------------
                if delay counter = off on_delay then    -- Sjekker om
                teller ønsket forsinkelse er nådd.
                        ------------------------------
                        delay counter := 0;
                        count flag := '0';
                        apply pwm enable <= '1';
                --------------------------------------
                else delay counter := delay counter + 1;
                --------------------------------------
                end if;
        ----------------------------------------------
        end if;
------------------------------------------------------
end if;

end process;
-- ************************************************** --




-- ************************************************************ --
-- Aktiveres av "delay pwm" via signalet "apply_pwm_enable".
-- Påtrykker h-brua de nye kommuteringsdata.
-- ************************************************************ --
```

```
apply_pwm : process (apply_pwm_enable, commutation_in, pwm_int, clk) is

begin

----------------------------------------------------------------------------
if      rising edge(clk) then
        --------------------------------------------------------------------
        if      apply pwm enable = '1' then
                ------------------------------------------------------------
                commutation int(2 downto 0) <= commutation_in(2 downto 0);
                -- Tilordner rett vektor til pull-down
                ------------------------------------------------------------
                case commutation in(5 downto 3) is
                -- Tilordner pwm-signalet til rett utgang
                        ----------------------------------------------------
                        when "001" => commutation int(3) <= pwm int;
                        -- Funker, så lenge pwm int er i sens-lista.
                        when "010" => commutation int(4) <= pwm int;
                        when "100" => commutation_int(5) <= pwm_int;
                        when others => null;
                        ----------------------------------------------------
                end case;
        --------------------------------------------------------------------
        elsif   apply pwm enable = '0' then
                ------------------------------------------------------------
                commutation int <= (others => '0');
        --------------------------------------------------------------------
        end if;
----------------------------------------------------------------------------
end if;

end process;
-- ********************************************************************** --


-- ********************************************************** --
-- Prosess som påtrykker ny kommuteringsvektor utgangen.
-- ********************************************************** --
commutation out <= commutation int;
-- ********************************************************** --


end architecture;-- commutation_mux;
--
****************************************************************************
************************************* --




Library IEEE;
USE ieee.std logic 1164.all;
USE ieee.std logic arith.all ;
USE work.user_pkg.all;

--
****************************************************************************
************************* --
ENTITY pwm fpga IS
        PORT (  clock           : in STD LOGIC;
                reset           : in STD LOGIC;
                Data value      : in std_logic_vector(7 downto 0);      --
                Set-port for duty cycle.
                pwm             : out STD_LOGIC                         --
                Ferdig PWM-wave ut.
        );
END pwm_fpga;
--
****************************************************************************
************************* --
```

```
--
***********************************************************************
************************* --
ARCHITECTURE arch_pwm OF pwm_fpga IS

SIGNAL reg_out          : std_logic_vector(7 downto 0); -- Internt signal
for duty cycle.
SIGNAL cnt_out_int      : std_logic_vector(7 downto 0); -- Aner ikke!!!
SIGNAL pwm_int          : std_logic;                    -- Intern pwm-wave
tror jeg?
SIGNAL rco_int          : std_logic;                    -- Aner ikke!!!

BEGIN


--
***********************************************************************
*********** --
        -- 8-bit data register to store the data values .The data values
        -- will determine the duty cycle of PWM output
--
***********************************************************************
*********** --
        -- Slik forstår jeg denne: Leser inn ønsket duty cycle fra
        "data_value-porten",
        -- og legger disse tilgjengelig på "reg_out", et internt signal,
        for at telleren
        -- skal kunne bruke det.
--
***********************************************************************
*********** --
PROCESS (clock, reg_out, reset)
BEGIN

        IF (reset ='1') THEN
                reg_out <="00000000";   -- Nuller internt register.
        ELSIF (rising_edge(clock)) THEN -- Hver gang klokka slår:
                reg_out <= data_value;  -- Porten portes til internt
                signal.
        END IF;

END PROCESS;
--
***********************************************************************
*********** --


--
***********************************************************************
*********** --
        --8-bit up/down counter. Counts up or down based on the pwm_int
        signal
        --and generates terminal count whenever counter reaches the
        --maximum value or when it transists through zero. Terminal
        --count is uesd to automatically load the data value to generate
        --different pwm out with different duty cycle
        --INC and DEC are the two functions which are used for up and
        --down counting. they are defined in sepearate user_pakge library
--
***********************************************************************
*********** --
PROCESS (clock, cnt_out_int, rco_int, reg_out)
BEGIN

        IF (rco_int = '1') THEN         -- Slår til hvis cnt_out_int har
        nådd topp/bunn, og ved reset.
                cnt_out_int <= reg_out; -- Starter å telle på nytt.

        ELSIF rising_edge(clock) THEN
                IF (rco_int = '0' and pwm_int ='1' and cnt_out_int <
                "11111111") THEN
```

```
                        cnt_out_int <= INC(cnt_out_int);
                ELSE
                        IF (rco int ='0' and pwm_int ='0' and cnt_out_int >
                        "00000000") THEN
                                cnt_out_int <= DEC(cnt_out_int);
                        END IF;
                END IF;
        END IF;

END PROCESS;
--
***************************************************************************
*********** --


--
***************************************************************************
*********** --
PROCESS(cnt_out_int, rco_int, clock, reset)
BEGIN

        IF (reset ='1') THEN
                rco int <='1';          -- Nuller tellerne til init-verdi/
                omstart av PWM.
        ELSIF rising edge(clock) THEN
                IF ((cnt out int = "11111111") or (cnt_out_int
                ="00000000")) THEN
                        rco int <= '1'; -- Endestopp-bryter. Nuller
                        tellerne/omstart av PWM.
                ELSE
                        rco_int <= '0'; -- Deaktiverer endestopp-bryter.
                END IF;
        END IF;

END PROCESS;
--
***************************************************************************
*********** --


--
***************************************************************************
*********** --
-- Logic to Generate the PWM ouput.
PROCESS (clock,rco_int,reset)
BEGIN

        IF (reset = '1') THEN
                pwm int <='0';
        ELSIF rising edge(rco int) THEN
                pwm_int <= NOT(pwm_int);
        ELSE
                pwm_int <= pwm_int;
        END IF;

END PROCESS;

pwm <= pwm_int;

END arch_pwm;
--
***************************************************************************
************************ --



library IEEE;
use IEEE.STD LOGIC 1164.ALL;
use IEEE.STD LOGIC ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
library work;
use work.definitions.all;

--
**************************************************************************
************************************** --
entity motor_controller_interface is

        Port(
                sys clk         : in std logic;         -- Systemklokke.
                Kjører i første omgang høyeste tilgjengelige.
                rpm clk         : in std logic;         -- 32,768 kHz for
                beregning av RPM i sanntid.
                pwm clk         : in std logic;         -- Foreløpig ikke
                gjort noe beregninger her. Men denne bare trekker ekstra
                strøm ved å være for høy tror jeg.
                reset           : in std_logic;         -- Global reset av
                alle registre.

                top hall in     : in std logic_vector(2 downto 0);      --
                Tilkobles hall-sensorene.

                top duty cycle  : in std logic vector(7 downto 0);      --
                Pådrag fra regulator (Ønsket dutycycle) --throttle
                top rotation dir: in std_logic;                         --
                Ønsket rotasjonsretning.        -- rotation_dir

                top div req     : out std logic;
                top div ack     : in  std logic;
                top dividend    : out std_logic_vector((data_width*2)-1
                downto 0);
                top divisor     : out std_logic_vector((data_width*1)-1
                downto 0);
                top result_in   : in std_logic_vector((speed_width -1)
                downto 0);

                top_speed_out   : out std_logic_vector(speed_width-1 downto
                0);
                top motor ctrl  : out std logic_vector(5 downto 0)      --
                Styrer transistorene i H-brua.
        );
end entity motor_controller_interface;
--
**************************************************************************
************************************** --

architecture motor_controller of motor_controller_interface is

        -- ***************** Signal-section: ***************** --
        signal sys clk int      : std_logic := '0';             --
        Systemklokke. Kjører i første omgang høyeste tilgjengelige.
        signal rpm clk int      : std logic := '0';             -- 32,768
        kHz tilgjengelig på utviklingskortet.
        signal pwm clk_int      : std logic := '0';             --
        Foreløpig ikke gjort noe beregninger her. Men denne bare trekker
        ekstra strøm ved å være for høy tror jeg.
        signal reset int        : std_logic := '0';             -- Global
        reset av alle registre.

        signal hall_sensors int : std logic vector(2 downto 0) := (others
        => '0');        -- Tilkobles hall-sensorene.
        signal hall_to_com_int  : std_logic vector(2 downto 0) := (others
        => '0');        -- Signal fra hall monitor til commutator.
        signal com_to_mux int   : std logic vector(5 downto 0) := (others
        => '0');        -- Kommuteringsvektor til h-bridge
        signal pwm to mux int   : std_logic := '0';
        -- Fra pwm-modul til mux.

        signal duty_cycle int   : std_logic vector(7 downto 0) := (others
        => '0');        -- Pådrag fra regulator (Ønsket dutycycle)
        signal rotation dir int : std_logic := '0';
        -- Ønsket rotasjonsretning.
```

```
        signal div_req_int      : std_logic := '0';
        -- Interface til division_arbiter
        signal div_ack_int      : std_logic := '0';
        -- Interface til division_arbiter
        signal dividend_int     : std_logic_vector((data_width*2)-1 downto
        0) := (others => '0');  -- Interface til division_arbiter
        signal divisor_int      : std_logic_vector((data_width*1)-1 downto
        0) := (others => '0');  -- Interface til division_arbiter
        signal result_int       : std_logic_vector((speed_width-1) downto
        0) := (others => '0');  -- Denne reduseres til 13bit i arbiter!

--      signal speed_out_int    : std_logic_vector(speed_width-1 downto 0)
:= (others => '0');     -- Interface til communicator
        signal motor_ctrl_int   : std_logic_vector(5 downto 0) := (others
        => '0');                -- Styrer transistorene i H-brua.

        ------------------ Debugging-signals: -----------------
        signal rpm_counter_int  : std_logic_vector(8 downto 0) := (others
        => '0');                -- For debugging.
        -- **************** Signal-section: **************** --

--
***************************************************************************
************************************** --


---------------------------------------------------------------------------
---------
component hall_monitor_interface
        port(
                sys_clk         : in    std_logic;      -- Egen
                systemklokke, for å speede opp ting.
                rpm_clk         : in    std_logic;      -- Klokke til
                telleren. 32,768 kHz.
                reset           : in    std_logic;      -- Trengs til
                rpm-estimator

                hall_in         : in    std_logic_vector(2 downto 0);
                -- Inngang direkte tilkoblet motorens hall-sensorer.
                hall_out        : out   std_logic_vector(2 downto 0);
                -- Porter signalet ut i henhold til kommuteringsvinkel.
                speed           : out   std_logic_vector(speed_width-1
                downto 0);-- Tallet vil aldri være større enn 13bit. Men
                bør jeg bruke 16bit interface?

                division_req    : out   std_logic;      -- Sender
                forespørsel om å få utført divisjon.
                division_ack    : in    std_logic;      -- Tilgang til
                divisjon innvilget.

                dividend_out    : out   std_logic_vector((data_width*2)-1
                downto 0);      -- Teller
                divisor_out     : out   std_logic_vector((data_width*1)-1
                downto 0);      -- Nevner
                result_in       : in    std_logic_vector((speed_width -1)
                downto 0);      -- Svar -- Må være full lengde! Kuttes ned
                her.

                rpm_counter     : out   std_logic_vector(8 downto 0) := (
                others => '0')  -- Kun for debugging.
        );
end component;
---------------------------------------------------------------------------
---------

---------------------------------------------------------------------------
---------
component commutator_interface
        port(
                hall_position   : in    std_logic_vector(2 downto 0);   --
                Graykodet soneangivning for rotor.
```

```
                rotation dir     : in    std_logic;                          --
                1 for høyre, 0 for venstre.
                commutation out : out   std logic_vector(5 downto 0)     --
                Kobles til drivtrinnet (h-brua).
        );
end component;
---------------------------------------------------------------------------
---------

---------------------------------------------------------------------------
---------
component commutation_mux_interface
        port(
                clk              : in std logic;
                reset            : in std logic;
                commutation in   : in std_logic_vector(5 downto 0); -- Mitt
                select-signal.
                pwm in           : in std logic;
                commutation_out : out std_logic_vector(5 downto 0)
        );
end component;
---------------------------------------------------------------------------
---------

---------------------------------------------------------------------------
---------
component pwm_fpga
        port(
                clock            : in std logic;                          --
                Systemklokke spesifikt for PWM.
                reset            : in std logic;
                Data value       : in std_logic_vector(7 downto 0);       --
                Set-port for duty cycle.
                pwm              : out std_logic                          --
                Ferdig PWM-wave ut.
        );
end component;
---------------------------------------------------------------------------
---------

begin

--
***************************************************************************
*** --
--hall1 : entity work.hall monitor_interface(behavioral)
hall_1 : hall monitor_interface
        port map(
                sys clk         => sys clk int,
                rpm clk         => rpm clk int,
                reset           => reset_int,

                hall in         => hall sensors int,
                hall out        => hall to com int,
                speed           => top_speed_out,

                division req    => div req int,
                division_ack    => div_ack_int,

                dividend out    => dividend int,
                divisor out     => divisor int,
                result_in       => result_int,

                rpm_counter     => rpm_counter_int      -- For debugging.
        );
--
***************************************************************************
*** --

--
***************************************************************************
*** --
```

```
--comm1 : entity atml work.commutator_interface(commutator)
comm_1 : commutator_interface
        port map(
                hall position   => hall to com int,
                rotation dir    => rotation dir int,
                commutation_out => com_to_mux_int
                );
--
***************************************************************************
*** --

--
***************************************************************************
*** --
--mux1 : entity atml work.commutation_mux_interface(commutation_mux)
mux_1 : commutation_mux_interface
        port map(
                clk             => sys clk int,
                reset           => reset int,
                commutation_in  => com to mux int,
                pwm in          => pwm to mux int,
                commutation_out => motor_ctrl_int
                );
--
***************************************************************************
*** --

--
***************************************************************************
*** --
--pwm1 : entity atml_work.pwm_fpga(arch_pwm)
pwm_1 : pwm fpga
        port map(
                clock           => pwm clk int,
                reset           => reset int,
                Data_value      => duty cycle int,
                pwm             => pwm_to_mux_int
                );
--
***************************************************************************
*** --

        -- ** Signaler ut fra hall-monitor:
        top div req     <= div req int;
        div ack int     <= top div ack;
        top dividend    <= dividend int;
        top divisor     <= divisor int;
        result_int      <= top_result_in;

        rotation_dir_int<= top_rotation_dir;

        -- ***** Oppdaterer signaler ***** --
        sys clk int     <= sys clk;
        rpm clk int     <= rpm clk;
        pwm clk int     <= pwm clk;
        reset_int       <= reset;

--      top speed out   <= speed out int;       -- Har koblet direkte
        top_motor_ctrl  <= motor_ctrl_int;

        hall_sensors_int<= top_hall_in;

        duty_cycle_int <= top_duty_cycle;

end architecture motor_controller;
--
***************************************************************************
*************************************** --
```

```
library IEEE;
use IEEE.STD LOGIC 1164.ALL;
use IEEE.STD LOGIC ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---------------------------------------------
package hall definitions is
        --constant cpu freq : integer := 32768; -- Usikker på om dette går.
        Men må ha noe på 32bit.
        --constant cpu freq : std logic vector(14 downto 0) :=
        "111111111111111";-- Feil!! "100000000000000"; -- Noen enklere måte
        å initialisere denne på?

        constant freq div 6 : std_logic_vector(12 downto 0) :=
        "1010101010101"; -- 5461
        --constant noise filter : integer := 0; -- Hvor mange bit som
        neglesjeres i hastighet før nytt interrupt genereres.

        --constant counter_width : integer := 9;
end package;
---------------------------------------------

package definitions is

        constant m1 : integer := 0;
        constant m2 : integer := 1;
        constant m3 : integer := 2;
        constant m4 : integer := 3;
        constant m5 : integer := 4;

        --constant freq div_6 : std_logic_vector(12 downto 0) :=
        "1010101010101"; -- 5461
        constant counter_width : integer := 9;


        -- data width kan reduseres til 12 bit, basert på at 5461/2 =
        2730,5 er største forekommende output.
        -- Dette ville gjøre 2^12 = 4096 tilstrekkelig. men ikke generelt.
        Teller vil antakelig bli endret.


        -- jeg lurer faktisk på om divisjonen gir kraftig overflow ved høye
        hastigheter, fordi 13 er for lavt.
        -- Nei, så lenge div_6 utføres på konstanten, så skal det bære!
        constant data width     : integer := 12;        -- Brukes i
        RPM estimator, og divisjonskrets.       -- Endret denne fra 12 til
        13 for å få kompilert.
        constant speed width    : integer := 9; -- Bruker 13 bit for å
        spare ressurser i FPGA. Svaret blir aldri større enn 13 bit.

        constant noise filter : integer := 0; -- Hvor mange bit som
        neglesjeres i hastighet før nytt interrupt genereres.
        constant off on delay   : integer := 1000;      -- Definerer
        kommuteringsforsinkelsen. Viktig!!! Aner ikke hva denne bør være
        ennå.

end package;




LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

PACKAGE user pkg IS
        function INC(X: STD LOGIC VECTOR) return STD LOGIC VECTOR;
        function DEC(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
END user_pkg ;

-- ************************************************************ --
PACKAGE BODY user_pkg IS
```

```
        -- ***************************************************** --
        function INC(X: STD LOGIC VECTOR) return STD LOGIC_VECTOR is
                variable XV: STD_LOGIC_VECTOR(X'LENGTH - 1 downto 0);
        begin
                XV := X;
                for I in 0 to XV'HIGH LOOP
                        if XV(I) = '0' then
                                XV(I) := '1';
                                exit;
                        else
                                XV(I) := '0';
                        end if;
                end loop;
        return XV;
        end INC;
        -- ***************************************************** --


        -- ***************************************************** --
        function DEC(X: STD LOGIC VECTOR) return STD LOGIC_VECTOR is
                variable XV: STD_LOGIC_VECTOR(X'LENGTH - 1 downto 0);
        begin
                XV := X;
                for I in 0 to XV'HIGH LOOP
                        if XV(I) = '1' then
                                XV(I) := '0';
                                exit;
                        else
                                XV(I) := '1';
                        end if;
                end loop;

        return XV;
        end DEC;
        -- ***************************************************** --

END user pkg;
-- ***************************************************************** --



---------------------------------------------------------------------------
----
--  File: div_unit.vhd
--
--
--
--  Copyright (C) Deversys, 2004
--
--
--
--  one-clock division algorithm
--
--
--
--  Author: Vladimir V. Erokhin, PhD,
--
--          e-mails: vladvas@deversys.com; vladvas@verilog.ru;
--
--
--
--  Synthesis results for 0.35u library:
--
-- -----------------------------------------
--
-- |  operands  |   delay   | combinational |
--
-- | dimension  |   (ns)    |  area (gates) |
--
-- |------------|-----------|---------------|
--
```

```
-- |   32/16    |   63.12   |      4,200    |
--
-- -----------------------------------------
--
---------------  Revision History
----------------------------------------
--
--
--          Date         Engineer: Marius Mossum  Description
--
--
--
----------------------------------------------------------------------------
----

library IEEE;
use IEEE.std logic 1164.all;
use IEEE.std logic arith.all;
use IEEE.std_logic_unsigned.all;

library work;
use work.definitions.all;

----------------------------------------------------------------------------
----
entity DIV UNIT is
        port (
                CLK     : in std logic;
                --ENABLE        : in std logic;         -- Added.
                DIVIDEND: in STD LOGIC VECTOR (data width*2 - 1 downto 0);
                DIVISOR : in STD LOGIC VECTOR (data width - 1 downto 0);
                DIV_RESULT: out STD_LOGIC_VECTOR (data_width*2 downto 0)
        );
end DIV UNIT;
----------------------------------------------------------------------------
----

architecture RTL of DIV_UNIT is

----------------------------------------------------------------------------
----
function division(DIVIDEND      : STD LOGIC VECTOR;
                  DIVISOR       : STD_LOGIC_VECTOR)
         return STD_LOGIC_VECTOR is

variable B      : STD LOGIC VECTOR(DIVISOR'length - 1 downto 0);
variable A      : STD LOGIC VECTOR(DIVIDEND'length - 1 downto 0);
variable QUOTIENT, REMAINDER : STD_LOGIC_VECTOR(DIVISOR'length - 1 downto
0);
variable VECT   : STD LOGIC VECTOR(DIVIDEND'length downto 0);
variable QI     : STD LOGIC VECTOR(0 downto 0);
variable OVFL   : STD_LOGIC;

----------------------------------------------------------------------------
----
-- div -- div -- div -- div -- div -- div -- div -- div -- div -- div --
div --
function div(A: STD LOGIC VECTOR;
             B: STD LOGIC VECTOR;
             Q: STD LOGIC VECTOR;
             EXT: STD LOGIC)
         return STD_LOGIC_VECTOR is

variable R              : STD LOGIC VECTOR(A'length - 2 downto 0);
variable RESIDUAL       : STD LOGIC VECTOR(A'length - 1 downto 0);
variable QN             : STD LOGIC VECTOR(Q'length downto 0);
variable S              : STD_LOGIC_VECTOR(B'length + Q'length downto 0);

----------------------------------------------------------------------------
----
-- div1 -- div1 -- div1 -- div1 -- div1 -- div1 -- div1 -- div1 -- div1 --
div1 --
```

```
function div1(A: STD_LOGIC_VECTOR; B: STD_LOGIC_VECTOR; Q:
STD_LOGIC_VECTOR; EXT: STD_LOGIC)
         return STD_LOGIC_VECTOR is

variable S      : STD_LOGIC_VECTOR(A'length downto 0);
variable REST   : STD_LOGIC_VECTOR(A'length - 1 downto 0);
variable QN     : STD_LOGIC_VECTOR(Q'length downto 0);

begin   -- Begining of "div1" i suppose.
  S := EXT & A - B;

  QN := Q & (not S(S'high));
  if S(S'high) = '1' then
    REST := A;
  else
    REST := S(S'high - 1 downto 0);
  end if;
  return QN & REST;
end div1;
-- div1 -- div1 -- div1 -- div1 -- div1 -- div1 -- div1 -- div1 -- div1 --
div1
---------------------------------------------------------------------------
----


begin -- Begining of "div" i suppose.
  S := div1(A(A'high downto A'high - B'high), B, Q, EXT);
  QN := S(S'high downto B'high + 1);

  if A'length > B'length then
    R := S(B'high - 1 downto 0) & A(A'high - B'high - 1 downto 0);
    return DIV(R, B, QN, S(B'high));    -- save MSB '1' in the rest for
    future sum
  else
    RESIDUAL := S(B'high downto 0);
    return QN(QN'high - 1 downto 0) & RESIDUAL;  -- delete initial '0'
  end if;
end div;
-- div -- div -- div -- div -- div -- div -- div -- div -- div -- div --
div --
---------------------------------------------------------------------------
----

begin -- Beginning of "division" i suppose
  A := DIVIDEND;                                      -- it is necessary to
  avoid errors during synthesis!!!!
  B := DIVISOR;
  QI := (others =>'0');

  VECT := div(A, B, QI, '0');

  QUOTIENT := VECT(VECT'high - 1 downto B'high + 1);
  REMAINDER := VECT(B'high downto 0);
  OVFL := VECT(VECT'high );
 return OVFL & QUOTIENT & REMAINDER;
-- return VECT;

end division;
---------------------------------------------------------------------------
----


signal A_REG: std_logic_vector(data_width*2 - 1 downto 0);
signal B_REG: std_logic_vector(data_width - 1 downto 0);
signal RD_REG: std_logic_vector(data_width*2 downto 0) := (others => '0');
-- Added initializing. Safe?

begin -- Begining of architecture?


aaa: process(CLK)
begin
```

```
if CLK'event and CLK = '1' then
--  if ENABLE = '1' then                -- Added
     A REG <= DIVIDEND;
     B_REG <= DIVISOR;

    RD_REG <= DIVISION(A REG, B REG);
        --  RD_REG <= IDIVISION(A_REG, B_REG);

--  end if; -- enable           -- Added
end if;

end process;

DIV_RESULT <= RD_REG;


end RTL;





library IEEE;
use IEEE.STD_Logic_1164.all, IEEE.Numeric_STD.all;

library work;
use work.definitions.all;

--
****************************************************************************
******************************************************************** --
entity arbiter_interface is
        port (  Clock, Reset    : in std_logic;

                Dividend 1,
                Dividend 2,
                Dividend 3,
                Dividend 4,
                Dividend 5      : in std_logic_vector(data_width*2 - 1
                downto 0);

                Divisor 1,
                Divisor 2,
                Divisor 3,
                Divisor 4,
                Divisor_5       : in std_logic_vector(data_width -1 downto
                0);

                Req 1,
                Req 2,
                Req 3,
                Req 4,
                Req_5           : in std_logic;

                Ack 1,
                Ack 2,
                Ack 3,
                Ack 4,
                Ack_5           : out std_logic;

                Result 1,
                Result 2,
                Result 3,
                Result 4,
                Result_5        : out std_logic_vector(speed_width -1
                downto 0);

                Dividend out    : out std_logic_vector(data_width*2 - 1
                downto 0);
                Divisor out     : out std_logic_vector(data_width   - 1
                downto 0);
                Result_in       : in  std_logic_vector(data_width*2 downto
```

```
                0)              -- foreløpig ikke implementert. går
                direkte.

        );
end entity arbiter_interface;
--
****************************************************************************
******************************************************************** --


--
****************************************************************************
******************************************************************** --
architecture division_arbiter of arbiter_interface is

        type ARB STATE_TYPE is (Idle,Grant_1,Grant_2,Grant_3, Grant_4,
        Grant 5);
        signal CurrentState, NextState, LastState: ARB STATE TYPE;
        signal En11, En12, En21, En22, En31, En32, En41, En42, En51, En52 :
        std_logic;

begin

-- ********************************************************* --
-- ********************************************************* --
answer_ready_poller : process(Result_in) is

        variable last_result    : std_logic_vector(speed_width-1 downto 0)
        := (others => '0');

        alias divide_by_zero    is Result in(data width*2);
        alias result            is Result_in(data_width+speed_width-1
        downto data width);
        alias remainder         is Result_in(data_width*1-1 downto 0);
begin
        -------------------------------------------------------
        if (result /= last result) and (divide by zero = '0') then
                -----------------------------------------------
                case LastState is
                        when Grant 1 => Result 1 <= result;
                        when Grant 2 => Result 2 <= result;
                        when Grant 3 => Result 3 <= result;
                        when Grant 4 => Result 4 <= result;
                        when Grant 5 => Result_5 <= result;
                        when others => null;
                end case;
                -----------------------------------------------
                last result := result;--current result;
        -------------------------------------------------------
        elsif (divide_by zero = '1') then                        --
        Gjør systemet bedre, men gjør også simuleringene vanskeligere å
        lese.
                Result 1 <= (others => '0');
                Result 2 <= (others => '0');
                Result 3 <= (others => '0');
                Result 4 <= (others => '0');
                Result 5 <= (others => '0');
        -------------------------------------------------------
        end if;

end process;
-- ********************************************************* --


-- ********************************************************* --
-- ********************************************************* --
ARBITER_COMB : process(Req_1, Req_2, Req_3, Req_4, Req_5,CurrentState)

begin
        Ack_1 <= '0'; Ack_2 <= '0'; Ack_3 <= '0'; Ack_4 <= '0'; Ack_5 <= '0
        ';
```

```
        case CurrentState is
                -- Her kunne jeg fint bygget inn noe litt mer rettferdig.
                Men tar meg ikke tid nå.
                -----------------------------------------------------
                when Idle =>
                        -----------------------------------------
                        if      Req_1 = '1' then NextState <= Grant_1;
                        elsif   Req_2 = '1' then NextState <= Grant_2;
                        elsif   Req_3 = '1' then NextState <= Grant_3;
                        elsif   Req_4 = '1' then NextState <= Grant_4;
                        elsif   Req_5 = '1' then NextState <= Grant_5;
                        else                     NextState <= Idle;
                        end if;
                -----------------------------------------------------
                when Grant_1 => Ack_1 <= '1';
                        -----------------------------------------
                        if      Req_2 = '1' then NextState <= Grant_2;
                        elsif   Req_3 = '1' then NextState <= Grant_3;
                        elsif   Req_4 = '1' then NextState <= Grant_4;
                        elsif   Req_5 = '1' then NextState <= Grant_5;
                        elsif   Req_1 = '1' then NextState <= Grant_1;
                        else                     NextState <= Idle;
                        end if;
                -----------------------------------------------------
                when Grant_2 => Ack_2 <= '1';
                        -----------------------------------------
                        if      Req_3 = '1' then NextState <= Grant_3;
                        elsif   Req_4 = '1' then NextState <= Grant_4;
                        elsif   Req_5 = '1' then NextState <= Grant_5;
                        elsif   Req_1 = '1' then NextState <= Grant_1;
                        elsif   Req_2 = '1' then NextState <= Grant_2;
                        else                     NextState <= Idle;
                        end if;
                -----------------------------------------------------
                when Grant_3 => Ack_3 <= '1';
                        -----------------------------------------
                        if      Req_4 = '1' then NextState <= Grant_4;
                        elsif   Req_5 = '1' then NextState <= Grant_5;
                        elsif   Req_1 = '1' then NextState <= Grant_1;
                        elsif   Req_2 = '1' then NextState <= Grant_2;
                        elsif   Req_3 = '1' then NextState <= Grant_3;
                        else                     NextState <= Idle;
                        end if;
                -----------------------------------------------------
                when Grant_4 => Ack_4 <= '1';
                        -----------------------------------------
                        if      Req_5 = '1' then NextState <= Grant_5;
                        elsif   Req_1 = '1' then NextState <= Grant_1;
                        elsif   Req_2 = '1' then NextState <= Grant_2;
                        elsif   Req_3 = '1' then NextState <= Grant_3;
                        elsif   Req_4 = '1' then NextState <= Grant_4;
                        else                     NextState <= Idle;
                        end if;
                -----------------------------------------------------
                when Grant_5 => Ack_5 <= '1';
                        -----------------------------------------
                        if      Req_1 = '1' then NextState <= Grant_1;
                        elsif   Req_2 = '1' then NextState <= Grant_2;
                        elsif   Req_3 = '1' then NextState <= Grant_3;
                        elsif   Req_4 = '1' then NextState <= Grant_4;
                        elsif   Req_5 = '1' then NextState <= Grant_5;
                        else                     NextState <= Idle;
                        end if;
                -----------------------------------------------------
                when others => null;
        end case;

end process;
-- ************************************************************ --


-- ************************************************************ --
```

```
-- ******************************************************** --
ARBITER_SEQ : process(Reset, Clock)

        variable temp_state : ARB_STATE_TYPE;
begin
        if(Reset = '1') then
                CurrentState <= Idle;
                LastState <= Idle;
                temp state := Idle;     -- Added. Smart?
        elsif rising edge(Clock) then
                LastState <= temp state;
                temp state := CurrentState;
                Currentstate <= NextState;
        end if;
end process;
-- ******************************************************** --


-- ******************************************************** --
-- ******************************************************** --
SYNC TRI_STATE_ENS : process(Reset, Clock)
begin
        if      (Reset = '1') then

                En11 <= '0'; En12 <= '0';
                En21 <= '0'; En22 <= '0';
                En31 <= '0'; En32 <= '0';
                En41 <= '0'; En42 <= '0';
                En51 <= '0'; En52 <= '0';

        elsif rising_edge(Clock) then

                En11 <= '0'; En12 <= '0';
                En21 <= '0'; En22 <= '0';
                En31 <= '0'; En32 <= '0';
                En41 <= '0'; En42 <= '0';
                En51 <= '0'; En52 <= '0';

                case NextState is
                        when Grant 1 => En11 <= '1'; En12 <= '1';
                        when Grant 2 => En21 <= '1'; En22 <= '1';
                        when Grant 3 => En31 <= '1'; En32 <= '1';
                        when Grant 4 => En41 <= '1'; En42 <= '1';
                        when Grant 5 => En51 <= '1'; En52 <= '1';
                        when others =>
                                En11 <= '0'; En12 <= '0';
                                En21 <= '0'; En22 <= '0';
                                En31 <= '0'; En32 <= '0';
                                En41 <= '0'; En42 <= '0';
                                En51 <= '0'; En52 <= '0';
                end case;
        end if;
end process;
-- ******************************************************** --


-- ******************************************************** --
-- ******************************************************** --
Dividend out <= Dividend 1 when (En11 = '1') else (others => 'Z');
Dividend out <= Dividend 2 when (En21 = '1') else (others => 'Z');
Dividend out <= Dividend 3 when (En31 = '1') else (others => 'Z');
Dividend out <= Dividend 4 when (En41 = '1') else (others => 'Z');
Dividend_out <= Dividend_5 when (En51 = '1') else (others => 'Z');

-- Gjør systemet bedre. Men blir vanskelig å lese simuleringene. Fjernes
midlertidig
--Dividend out <= (others => '0') when (En11 = '0' and En21 = '0' and En31
= '0' and En41 = '0' and En51 = '0') else (others => 'Z');
-- En bedre variant, som virker fra oppstart av.
Dividend out <= (others => '0') when (En11 /= '1' and En21 /= '1' and En31
/= '1' and En41 /= '1' and En51 /= '1') else (others => 'Z');
```

```
Divisor_out <= Divisor_1 when (En12 = '1') else (others => 'Z');
Divisor_out <= Divisor_2 when (En22 = '1') else (others => 'Z');
Divisor_out <= Divisor_3 when (En32 = '1') else (others => 'Z');
Divisor_out <= Divisor_4 when (En42 = '1') else (others => 'Z');
Divisor_out <= Divisor_5 when (En52 = '1') else (others => 'Z');

-- Gjør systemet bedre. Men blir vanskelig å lese simuleringene. Fjernes
midlertidig
--Divisor_out <= (others => '0') when (En12 = '0' and En22 = '0' and En32 =
'0' and En42 = '0' and En52 = '0') else (others => 'Z');
-- En bedre variant, som virker fra oppstart av.
Divisor_out <= (others => '0') when (En12 /= '1' and En22 /= '1' and En32
/= '1' and En42 /= '1' and En52 /= '1') else (others => 'Z');
-- ****************************************************** --

end architecture;
--
****************************************************************************
********************************************************************** --




library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.definitions.all;

--
****************************************************************************
************************************** --
entity motor_communication_interface is

port(   -- ***** System interface ***** --
        clk             : in std_logic;         -- Bør nok kjøre på samme
        klokke som AVR.
        reset           : in std_logic;

        -- ***** Calling interface ***** --
        IOselect        : in    std_logic_vector(4 downto 0);   --
        Interrupt som iverksetter utveksling av data. Throttle inn og Speed
        ut.
        IOwrite         : in    std_logic;                      -- Gir FPGA
        signal om at AVR er klar for mottak av data.
        IOread          : in    std_logic;                      -- Gir FPGA
        signal om at data blir sendt fra AVR
        data_bus        : inout std_logic_vector(7 downto 0);   -- Felles
        data-buss for sending og mottak av data.
        Interrupt_out   : out   std_logic_vector(4 downto 0);   -- Velger
        foreløpig å ha dedikerte interrupt.

        -- ***** Motor interface ***** --
        m1_direction    : out   std_logic;
        m1_throttle     : out   std_logic_vector(7 downto 0);
        m1_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m2_direction    : out   std_logic;
        m2_throttle     : out   std_logic_vector(7 downto 0);
        m2_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m3_direction    : out   std_logic;
        m3_throttle     : out   std_logic_vector(7 downto 0);
        m3_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m4_direction    : out   std_logic;
        m4_throttle     : out   std_logic_vector(7 downto 0);
        m4_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m5_direction    : out   std_logic;
```

```
        m5_throttle     : out   std_logic_vector(7 downto 0);
        m5_current_speed: in    std_logic_vector(speed_width-1 downto 0)

);

end entity motor_communication_interface;
--
****************************************************************************
************************************** --


--
****************************************************************************
************************************** --
architecture motor_communicatior of motor_communication_interface is

        signal motor_interrupt   : std_logic_vector(4 downto 0) := (others
        => '0');
        signal interrupt_handeled: std_logic_vector(4 downto 0) := "00000";
        -- Et eksperiment
        signal cancel_interrupt  : std_logic_vector(4 downto 0) := (others
        => '0');

        signal m1_low_byte_sent  : std_logic := '0';    -- Debug-signal
        signal m2_low_byte_sent  : std_logic := '0';    -- Debug-signal
        signal m3_low_byte_sent  : std_logic := '0';    -- Debug-signal
        signal m4_low_byte_sent  : std_logic := '0';    -- Debug-signal
        signal m5_low_byte_sent  : std_logic := '0';    -- Debug-signal

        -- De følgende blir ikke resat. Trengs det?
        signal m1_current_speed_int : std_logic_vector(speed_width-1 downto
        0) := (others => '0');
        signal m2_current_speed_int : std_logic_vector(speed_width-1 downto
        0) := (others => '0');
        signal m3_current_speed_int : std_logic_vector(speed_width-1 downto
        0) := (others => '0');
        signal m4_current_speed_int : std_logic_vector(speed_width-1 downto
        0) := (others => '0');
        signal m5_current_speed_int : std_logic_vector(speed_width-1 downto
        0) := (others => '0');

        signal m1_data_out : std_logic_vector(7 downto 0) := (others =>
        '0');
        signal m2_data_out : std_logic_vector(7 downto 0) := (others =>
        '0');
        signal m3_data_out : std_logic_vector(7 downto 0) := (others =>
        '0');
        signal m4_data_out : std_logic_vector(7 downto 0) := (others =>
        '0');
        signal m5_data_out : std_logic_vector(7 downto 0) := (others =>
        '0');

        signal m1_write_enable  : std_logic := '0';
        signal m2_write_enable  : std_logic := '0';
        signal m3_write_enable  : std_logic := '0';
        signal m4_write_enable  : std_logic := '0';
        signal m5_write_enable  : std_logic := '0';

        signal speed_changed    : std_logic_vector (4 downto 0) := "00000";
        signal interrupt_set    : std_logic_vector (4 downto 0) := "00000";
        signal interrupt_reset  : std_logic_vector (4 downto 0) := "00000";

        signal  m1_last_speed,
                m2_last_speed,
                m3_last_speed,
                m4_last_speed,
                m5_last_speed : std_logic_vector(speed_width-1 downto 0) :=
                (others => '0');


begin
```

```
--
***************************************************************************
************ --
-- *    Sjekker om hastighetene er endret. Sender beskjed hvis ja.
-- *    Kan ikke sette interrupt selv, siden de da ikke ville foreligge
samtidig som data ut.
--
***************************************************************************
************ --
new_speed_poller : process(     m1 current speed int,
                                m2 current speed int,
                                m3 current speed int,
                                m4 current speed int,
                                m5 current speed_int,
                                m1 last speed,
                                m2 last speed,
                                m3 last speed,
                                m4 last speed,
                                m5_last_speed) is

begin
        --speed_changed <= '0';

        -------------------------------------------------------------------
        ----------------------------
        if      m1 current_speed int /= m1_last_speed then speed_changed <=
        (m1 => '1', others => '0');

        -------------------------------------------------------------------
        ----------------------------
        elsif   m2 current_speed int /= m2_last_speed then speed_changed <=
        (m2 => '1', others => '0');

        -------------------------------------------------------------------
        ----------------------------
        elsif   m3 current_speed int /= m3_last_speed then speed_changed <=
        (m3 => '1', others => '0');

        -------------------------------------------------------------------
        ----------------------------
        elsif   m4 current_speed int /= m4_last_speed then speed_changed <=
        (m4 => '1', others => '0');

        -------------------------------------------------------------------
        ----------------------------
        elsif   m5 current_speed int /= m5_last_speed then speed_changed <=
        (m5 => '1', others => '0');

        -------------------------------------------------------------------
        ----------------------------
        else    speed_changed <= (others => '0');

        -------------------------------------------------------------------
        ----------------------------
        end if;

end process;
--
***************************************************************************
************ --


--
***************************************************************************
************ --
-- *
--
***************************************************************************
************ --
interrupt_setter : process (clk)--, reset) is   -- Added reset. Okei?
```

```
begin

        ----------------------------------------------------------
        if rising edge (clk) then
                --------------------------------------------------
--              if      reset = '1' then
--                      m1 last speed <= (others => '0');
--                      m2 last speed <= (others => '0');
--                      m3 last speed <= (others => '0');
--                      m4 last speed <= (others => '0');
--                      m5 last speed <= (others => '0');
--                      interrupt_set <= (others => '0');       -- Added
for testing.
                --------------------------------------------------
                --else
                        interrupt_set <= (others => '0');

                        ------------------------------------------
                        if      speed changed(m1) = '1' then
                                ----------------------------------
                                m1 last speed <= m1 current_speed;
                                interrupt set(m1) <= '1';
                        ------------------------------------------
                        elsif   speed changed(m2) = '1' then
                                ----------------------------------
                                m2 last speed <= m2 current_speed;
                                interrupt set(m2) <= '1';
                        ------------------------------------------
                        elsif   speed changed(m3) = '1' then
                                ----------------------------------
                                m3 last speed <= m3 current_speed;
                                interrupt set(m3) <= '1';
                        ------------------------------------------
                        elsif   speed changed(m4) = '1' then
                                ----------------------------------
                                m4 last speed <= m4 current_speed;
                                interrupt set(m4) <= '1';
                        ------------------------------------------
                        elsif   speed changed(m5) = '1' then
                                ----------------------------------
                                m5 last speed <= m5 current_speed;
                                interrupt set(m5) <= '1';
                        ------------------------------------------
                        end if;
                --------------------------------------------------
                --end if;
        ----------------------------------------------------------
        end if;
end process;
--
***************************************************************************
************ --



--
***************************************************************************
************ --
-- *
--
***************************************************************************
************ --
sr_latch : process (interrupt_set, interrupt_handeled)

        variable v srlatch_intern       : std_logic_vector(4 downto 0)  :=
        "00000"; -- Tester å initialisere!
        variable v srlatch_interrupt    : std_logic_vector(4 downto 0)  :=
        "00000"; -- Tester å initialisere!
begin
        v srlatch intern        := interrupt set nor v srlatch interrupt;
        v_srlatch_interrupt     := interrupt_handeled nor v_srlatch_intern;
```

```
        motor_interrupt         <= v_srlatch_interrupt;
end process;
--
***************************************************************************
************ --



--
***************************************************************************
************ --
-- *                    Genererer styresingalene til tristate-bufferne
--
***************************************************************************
************ --
Bus_write_enable : process (IOread, IOselect)

begin
        --------------------------------------------------------------
--      if      reset = '1' then        -- Added reset for debugging.
                ----------------------------------------------
--              m1 write enable <= '0';
--              m2 write enable <= '0';
--              m3 write enable <= '0';
--              m4 write enable <= '0';
--              m5 write enable <= '0';
        --------------------------------------------------------------
        if      IOread = '1' then
                ----------------------------------------------
                case    IOselect is
                        when "00001" => m1 write enable <= '1';
                        when "00010" => m2 write enable <= '1';
                        when "00100" => m3 write enable <= '1';
                        when "01000" => m4 write enable <= '1';
                        when "10000" => m5 write_enable <= '1';
                        when others => null;
                ----------------------------------------------
                end case;
        --------------------------------------------------------------
        elsif   IOread = '0' then
                ----------------------------------------------
                m1 write enable <= '0';
                m2 write enable <= '0';
                m3 write enable <= '0';
                m4 write enable <= '0';
                m5 write enable <= '0';
        --------------------------------------------------------------
        end if;
end process;
--
***************************************************************************
************ --
-- *                    Tristatebuffere for styre skriving til bussen:
--
***************************************************************************
************ --
data_bus <= m1 data out when (m1_write_enable = '1') and (reset /= '1')
else (others => 'Z');
data_bus <= m2 data out when (m2_write_enable = '1') and (reset /= '1')
else (others => 'Z');
data_bus <= m3 data out when (m3_write_enable = '1') and (reset /= '1')
else (others => 'Z');
data_bus <= m4 data out when (m4_write_enable = '1') and (reset /= '1')
else (others => 'Z');
data_bus <= m5 data out when (m5_write_enable = '1') and (reset /= '1')
else (others => 'Z');
data bus <= (others => 'Z') when m1 write enable /= '1' and m2 write_enable
/= '1' and m3 write enable /= '1' and m4_write_enable /= '1' and
m5 write_enable /= '1'; -- En test.
--
***************************************************************************
```

```
************ --



--
**************************************************************************
************ --
-- *    Ved endring på IOselect initieres utveksling av data.
--
**************************************************************************
************ --
IOselect_poller : process (IOselect, IOwrite, IOread) is

        variable        throttle  : std_logic_vector(7 downto 0) := (others
        => '0');
        variable        direction : std_logic := '0';

        variable        low_byte sent   : std logic_vector(4 downto 0) :=
        "00000";                -- Et eksperiment:

        variable        m1 speed int,
                        m2 speed int,
                        m3 speed int,
                        m4 speed int,
                        m5 speed int    : std logic vector(speed_width-1
                        downto 0) := (others => '0');
begin
--------------------------------------------------------------------------
-------------------------

        -----------------------------------------------------
--      if reset = '1' then
--              throttle := (others => '0');
--              direction := '0';
--              cancel interrupt <= (others => '0');
--              low_byte_sent := (others => '0');

--              m1 speed int := (others => '0');
--              m2 speed int := (others => '0');
--              m3 speed int := (others => '0');
--              m4 speed int := (others => '0');
--              m5_speed_int := (others => '0');

--              m1 direction    <= '0';
--              m2 direction    <= '0';
--              m3 direction    <= '0';
--              m4 direction    <= '0';
--              m5_direction    <= '0';

--              m1 throttle     <= (others => '0');
--              m2 throttle     <= (others => '0');
--              m3 throttle     <= (others => '0');
--              m4 throttle     <= (others => '0');
--              m5_throttle     <= (others => '0');

        -----------------------------------------------------
        -- AVR indikerer at den ønsker å sende data:
        -----------------------------------------------------
        if IOwrite = '1' then-- and reset /= '1' then
                ---------------------------------------------
                direction               := data bus(7);
                throttle(7 downto 1)    := data_bus(6 downto 0);
                throttle(0) := '0';
                ---------------------------------------------
                case IOselect is
                        when "00001" =>
                                m1 direction    <= direction;
                                m1 throttle     <= throttle;
                        when "00010" =>
                                m2 direction    <= direction;
                                m2 throttle     <= throttle;
                        when "00100" =>
```

```
                m3 direction    <= direction;
                m3 throttle     <= throttle;
        when "01000" =>
                m4 direction    <= direction;
                m4 throttle     <= throttle;
        when "10000" =>
                m5 direction    <= direction;
                m5 throttle     <= throttle;
        when others => null;
    end case;
    ---------------------------------------------

---------------------------------------------------
-- AVR indikerer at den ønsker å motta data:
-- AVR må kalle to ganger, for å få både low-byte og high-byte.
---------------------------------------------------
elsif IOread = '1' then --and reset /= '1' then
    ---------------------------------------------
    case IOselect is
        ---------------------------------------------
        when "00001" =>
                ------------------------------------
                if      low byte sent(m1) = '0' then
                -- Sjekker om low_byte er sendt for
                gjeldende motor.

                        m1_speed int := m1 current speed;
                        -- Lagrer hastigheten i lokal
                        variabel.
                        m1 data out <= m1 speed int(7
                        downto 0);-- Skriver low_byte til
                        data-bussen.

                        low byte sent(m1) := '1';
                        -- Setter flag som indikerer at
                        low byte er sendt.
                        m1_low byte sent <= '1';
                        -- Debug-signal
                        interrupt handeled(m1) <= '1';
                ------------------------------------
                elsif   low byte sent(m1) = '1' then
                -- Klart for å sende high_byte.

                        m1_data out <= (others => '0');
                        -- Nuller de øverste ubrukte
                        bitene.

                        m1_data out(speed width-9 downto 0)
                        <= m1 speed int(speed width-1
                        downto 8);-- Sender resten av
                        high_byte.

                        low byte sent(m1) := '0';
                        -- Deaktiverer flagget igjen.
                        m1_low byte sent <= '0';
                        -- Debug-signal
                        interrupt handeled(m1) <= '0';
                ------------------------------------
                end if;

        ---------------------------------------------
        when "00010" =>
                ------------------------------------
                if      low byte sent(m2) = '0' then
                -- Sjekker om low_byte er sendt for
                gjeldende motor.

                        m2_speed int := m2 current speed;
                        -- Lagrer hastigheten i lokal
                        variabel.
                        m2 data out <= m2 speed int(7
                        downto 0);-- Skriver low_byte til
```

```
                data-bussen.

                low byte sent(m2) := '1';
                -- Setter flag som indikerer at
                low byte er sendt.
                m2_low byte sent <= '1';
                -- Debug-signal
                interrupt handeled(m2) <= '1';
        -----------------------------------
        elsif   low byte sent(m2) = '1' then
        -- Klart for å sende high_byte.

                m2_data out <= (others => '0');
                -- Nuller de øverste ubrukte
                bitene.

                m2_data out(speed width-9 downto 0)
                <= m2 speed int(speed width-1
                downto 8);-- Sender resten av
                high_byte.

                low byte sent(m2) := '0';
                -- Deaktiverer flagget igjen.
                m2_low byte sent <= '0';
                -- Debug-signal
                interrupt handeled(m2) <= '0';
        -----------------------------------
        end if;

--------------------------------------------
when "00100" =>
        -----------------------------------
        if      low byte sent(m3) = '0' then
        -- Sjekker om low_byte er sendt for
        gjeldende motor.

                m3_speed int := m3 current speed;
                -- Lagrer hastigheten i lokal
                variabel.
                m3 data out <= m3 speed int(7
                downto 0);-- Skriver low_byte til
                data-bussen.

                low byte sent(m3) := '1';
                -- Setter flag som indikerer at
                low byte er sendt.
                m3_low byte sent <= '1';
                -- Debug-signal
                interrupt handeled(m3) <= '1';
        -----------------------------------
        elsif   low byte sent(m3) = '1' then
        -- Klart for å sende high_byte.

                m3_data out <= (others => '0');
                -- Nuller de øverste ubrukte
                bitene.

                m3_data out(speed width-9 downto 0)
                <= m3 speed int(speed width-1
                downto 8);-- Sender resten av
                high_byte.

                low byte sent(m3) := '0';
                -- Deaktiverer flagget igjen.
                m3_low byte sent <= '0';
                -- Debug-signal
                interrupt handeled(m3) <= '0';
        -----------------------------------
        end if;

--------------------------------------------
when "01000" =>
```

```
                ------------------------------------
                if      low byte sent(m4) = '0' then
                -- Sjekker om low_byte er sendt for
                gjeldende motor.

                        m4_speed int := m4 current speed;
                        -- Lagrer hastigheten i lokal
                        variabel.
                        m4 data out <= m4 speed int(7
                        downto 0);-- Skriver low_byte til
                        data-bussen.

                        low byte sent(m4) := '1';
                        -- Setter flag som indikerer at
                        low byte er sendt.
                        m4_low byte sent <= '1';
                        -- Debug-signal
                        interrupt handeled(m4) <= '1';
                ------------------------------------
                elsif   low byte sent(m4) = '1' then
                -- Klart for å sende high_byte.

                        m4_data out <= (others => '0');
                        -- Nuller de øverste ubrukte
                        bitene.

                        m4_data out(speed width-9 downto 0)
                        <= m4 speed int(speed width-1
                        downto 8);-- Sender resten av
                        high_byte.

                        low byte sent(m4) := '0';
                        -- Deaktiverer flagget igjen.
                        m4_low byte sent <= '0';
                        -- Debug-signal
                        interrupt handeled(m4) <= '0';
                ------------------------------------
                end if;

        --------------------------------------------
        when "10000" =>
                ------------------------------------
                if      low byte sent(m5) = '0' then
                -- Sjekker om low_byte er sendt for
                gjeldende motor.

                        m5_speed int := m5 current speed;
                        -- Lagrer hastigheten i lokal
                        variabel.
                        m5 data out <= m5 speed int(7
                        downto 0);-- Skriver low_byte til
                        data-bussen.

                        low byte sent(m5) := '1';
                        -- Setter flag som indikerer at
                        low byte er sendt.
                        m5_low byte sent <= '1';
                        -- Debug-signal
                        interrupt handeled(m5) <= '1';
                ------------------------------------
                elsif   low byte sent(m5) = '1' then
                -- Klart for å sende high_byte.

                        m5_data out <= (others => '0');
                        -- Nuller de øverste ubrukte
                        bitene.

                        m5_data out(speed width-9 downto 0)
                        <= m5 speed int(speed width-1
                        downto 8);-- Sender resten av
                        high_byte.
```

```
                                        low byte sent(m5) := '0';
                                        -- Deaktiverer flagget igjen.
                                        m5_low byte sent <= '0';
                                        -- Debug-signal
                                        interrupt handeled(m5) <= '0';
                                ------------------------------------
                                end if;

                        --------------------------------------------
                        when others => null;--exit;
                end case;
                ---------------------------------------------

        -----------------------------------------------------
        end if;

end process;
--
****************************************************************************
************ --

m1 current speed int <= m1 current speed;
m2 current speed int <= m2 current speed;
m3 current speed int <= m3 current speed;
m4 current speed int <= m4 current speed;
m5_current_speed_int <= m5_current_speed;

interrupt_out <= motor_interrupt;       -- Sender ut interrupt.

end architecture motor_communicatior;
--
****************************************************************************
**************************************** --



library IEEE;
use IEEE.STD LOGIC 1164.ALL;
use IEEE.STD LOGIC ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.definitions.all;

--
****************************************************************************
**************************************** --
entity top_entity_interface is

port(   sys clk         : in    std logic;
        rpm clk         : in    std logic;
        pwm clk         : in    std logic;      -- Added extra
        reset           : in    std_logic;

        Interrupt_out   : out   std logic vector(4 downto 0);
        IOselect        : in    std logic vector(4 downto 0);
        IOread          : in    std logic;
        IOwrite         : in    std logic;
        Data_bus        : inout std_logic_vector(7 downto 0);


        m1 hall sensors : in    std logic vector(2 downto 0);
        m2 hall sensors : in    std logic vector(2 downto 0);
        m3 hall sensors : in    std logic vector(2 downto 0);
        m4 hall sensors : in    std logic vector(2 downto 0);
        m5_hall_sensors : in    std_logic_vector(2 downto 0);

        m1 control      : out   std logic vector(5 downto 0);
        m2 control      : out   std logic vector(5 downto 0);
        m3 control      : out   std logic vector(5 downto 0);
        m4_control      : out   std_logic_vector(5 downto 0);
```

```
        m5_control      : out   std_logic_vector(5 downto 0)
);
end entity;
--
***************************************************************************
*************************************** --


architecture top_entity of top_entity_interface is

        -- Signaler og variable

        -- ***** Components ***** --
---------------------------------------------------------------------------
-------------------------------------
component motor_controller_interface
Port(
        sys clk         : in std logic;         -- Systemklokke. Kjører i
        første omgang høyeste tilgjengelige.
        rpm clk         : in std_logic;         -- 32,768 kHz for beregning
        av RPM i sanntid.
        pwm clk         : in std logic;         -- Foreløpig ikke gjort noe
        beregninger her. Men denne bare trekker ekstra strøm ved å være for
        høy tror jeg.
        reset           : in std_logic;         -- Global reset av alle
        registre.

        top hall in     : in std logic_vector(2 downto 0);      --
        Tilkobles hall-sensorene.

        top_duty cycle  : in std logic vector(7 downto 0);      -- Pådrag
        fra regulator (Ønsket dutycycle)        --throttle
        top rotation dir: in std logic;                         -- Ønsket
        rotasjonsretning.       -- rotation dir
        --top com angle : in std logic vector(7 downto 0);      -- Ønsket
        forskyvning av kommuteringsvinekelen.

        top div req     : out std logic;
        top div ack     : in  std logic;
        top dividend    : out std logic vector((data width*2)-1 downto 0);
        top divisor     : out std logic vector((data width*1)-1 downto 0);
        top result in   : in std logic vector((speed width -1) downto 0);
        -- Litt usikker på bredden her. Tror dette er rett!

        top speed out   : out std logic vector(speed width-1 downto 0);
        top motor ctrl  : out std_logic_vector(5 downto 0)      -- Styrer
        transistorene i H-brua.
);
end component;
---------------------------------------------------------------------------
-------------------------------------

---------------------------------------------------------------------------
-------------------------------------
component motor communication interface
port(   -- ***** System interface ***** --
        clk             : in std logic;
        reset           : in std_logic;

        -- ***** Calling interface ***** --
        IOselect        : in    std logic vector(4 downto 0);   --
        Interrupt som iverksetter utveksling av data. Throttle inn og Speed
        ut.
        IOwrite         : in    std logic;                      -- Gir FPGA
        signal om at AVR er klar for mottak av data.
        IOread          : in    std logic;                      -- Gir FPGA
        signal om at data blir sendt fra AVR
        Data bus        : inout std logic vector(7 downto 0);   -- Felles
        data-buss for sending og mottak av data.
        Interrupt_out   : out   std logic vector(4 downto 0);   -- Velger
        foreløpig å ha dedikerte interrupt.
```

```
        -- ***** Motor interface ***** --
        m1 direction    : out   std logic;
        m1 throttle     : out   std logic vector(7 downto 0);
        m1_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m2 direction    : out   std logic;
        m2 throttle     : out   std logic vector(7 downto 0);
        m2_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m3 direction    : out   std logic;
        m3 throttle     : out   std logic vector(7 downto 0);
        m3_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m4 direction    : out   std logic;
        m4 throttle     : out   std logic vector(7 downto 0);
        m4_current_speed: in    std_logic_vector(speed_width-1 downto 0);

        m5 direction    : out   std logic;
        m5 throttle     : out   std logic vector(7 downto 0);
        m5_current_speed: in    std_logic_vector(speed_width-1 downto 0)
);
end component;
---------------------------------------------------------------------------
-------------------------------------

---------------------------------------------------------------------------
-------------------------------------
component arbiter interface
port (  Clock, Reset    : in std_logic;

        Dividend 1,
        Dividend 2,
        Dividend 3,
        Dividend 4,
        Dividend_5      : in std_logic_vector(data_width*2 - 1 downto 0);

        Divisor 1,
        Divisor 2,
        Divisor 3,
        Divisor 4,
        Divisor_5       : in std_logic_vector(data_width -1 downto 0);

        Req 1,
        Req 2,
        Req 3,
        Req 4,
        Req_5           : in std_logic;

        Ack 1,
        Ack 2,
        Ack 3,
        Ack 4,
        Ack_5           : out std_logic;

        Result 1,
        Result 2,
        Result 3,
        Result 4,
        Result_5        : out std_logic_vector(speed_width -1 downto 0);

        Dividend out    : out std logic vector(data width*2 - 1 downto 0);
        Divisor out     : out std logic vector(data width   - 1 downto 0);
        Result_in       : in  std_logic_vector(data_width*2 downto 0)

);
end component;
---------------------------------------------------------------------------
-------------------------------------

---------------------------------------------------------------------------
-------------------------------------
component DIV_UNIT
```

```
port (
        CLK             : in  STD_LOGIC;
        DIVIDEND        : in  STD_LOGIC_VECTOR (data_width*2 - 1 downto 0);
        DIVISOR         : in  STD_LOGIC_VECTOR (data_width   - 1 downto 0);
        DIV_RESULT      : out STD_LOGIC_VECTOR (data_width*2 downto 0)
);
end component;
---------------------------------------------------------------------------
-------------------------------------

        signal w_sys_clk        : std_logic;
        signal w_rpm_clk        : std_logic;
        signal w_pwm_clk        : std_logic;
        signal w_reset          : std_logic;

--      signal w_IOselect       : std_logic_vector(4 downto 0);
--      signal w_IOread         : std_logic;
--      signal w_IOwrite        : std_logic;
--      signal w_Data_bus       : std_logic_vector(7 downto 0);
--      signal w_Interrupt      : std_logic_vector(4 downto 0);

        signal w_m1_hall        : std_logic_vector(2 downto 0);
        signal w_m1_throttle    : std_logic_vector(7 downto 0);
        signal w_m1_direction   : std_logic;
        --signal w_m1_angle     : std_logic_vector(7 downto 0);
        signal w_m1_speed       : std_logic_vector((speed_width-1) downto
        0);--   := (others => '0');

        signal w_m1_req         : std_logic;-- := '0';
        -- Added for init
        signal w_m1_ack         : std_logic;-- := '0';
        -- Added for init
        signal w_m1_dividend    : std_logic_vector((data_width*2 -1) downto
        0);--   := (others => '0');
        signal w_m1_divisor     : std_logic_vector((data_width*1 -1) downto
        0);--   := (others => '0');
        signal w_m1_result      : std_logic_vector(speed_width -1 downto
        0);--   := (others => '0');

        signal w_m2_hall        : std_logic_vector(2 downto 0);
        signal w_m2_throttle    : std_logic_vector(7 downto 0);
        signal w_m2_direction   : std_logic;
        --signal w_m2_angle     : std_logic_vector(7 downto 0);
        signal w_m2_speed       : std_logic_vector((speed_width-1) downto
        0);

        signal w_m2_req         : std_logic;
        signal w_m2_ack         : std_logic;
        signal w_m2_dividend    : std_logic_vector((data_width*2 -1) downto
        0);
        signal w_m2_divisor     : std_logic_vector((data_width*1 -1) downto
        0);
        signal w_m2_result      : std_logic_vector(speed_width -1 downto
        0);

        signal w_m3_hall        : std_logic_vector(2 downto 0);
        signal w_m3_throttle    : std_logic_vector(7 downto 0);
        signal w_m3_direction   : std_logic;
        --signal w_m3_angle     : std_logic_vector(7 downto 0);
        signal w_m3_speed       : std_logic_vector((speed_width-1) downto
        0);

        signal w_m3_req         : std_logic;
        signal w_m3_ack         : std_logic;
        signal w_m3_dividend    : std_logic_vector((data_width*2 -1) downto
        0);
        signal w_m3_divisor     : std_logic_vector((data_width*1 -1) downto
        0);
        signal w_m3_result      : std_logic_vector(speed_width -1 downto
        0);

        signal w_m4_hall        : std_logic_vector(2 downto 0);
```

```
        signal w_m4_throttle    : std_logic_vector(7 downto 0);
        signal w_m4_direction   : std_logic;
        --signal w_m4_angle     : std_logic_vector(7 downto 0);
        signal w_m4_speed       : std_logic_vector((speed_width-1) downto
        0);

        signal w_m4_req         : std_logic;
        signal w_m4_ack         : std_logic;
        signal w_m4_dividend    : std_logic_vector((data_width*2 -1) downto
        0);
        signal w_m4_divisor     : std_logic_vector((data_width*1 -1) downto
        0);
        signal w_m4_result      : std_logic_vector(speed_width -1 downto
        0);

        signal w_m5_hall        : std_logic_vector(2 downto 0);
        signal w_m5_throttle    : std_logic_vector(7 downto 0);
        signal w_m5_direction   : std_logic;
        --signal w_m5_angle     : std_logic_vector(7 downto 0);
        signal w_m5_speed       : std_logic_vector((speed_width-1) downto
        0);

        signal w_m5_req         : std_logic;
        signal w_m5_ack         : std_logic;
        signal w_m5_dividend    : std_logic_vector((data_width*2 -1) downto
        0);
        signal w_m5_divisor     : std_logic_vector((data_width*1 -1) downto
        0);
        signal w_m5_result      : std_logic_vector(speed_width -1 downto
        0);

        signal w_arb_dividend   : std_logic_vector((data_width*2 -1) downto
        0);
        signal w_arb_divisor    : std_logic_vector((data_width*1 -1) downto
        0);
        signal w_arbiter_result : std_logic_vector((data_width*2) downto
        0);
        --signal w_common_result        : std_logic_vector((data_width*2)
        downto 0);

        signal w_m1_ctrl        : std_logic_vector(5 downto 0);
        signal w_m2_ctrl        : std_logic_vector(5 downto 0);
        signal w_m3_ctrl        : std_logic_vector(5 downto 0);
        signal w_m4_ctrl        : std_logic_vector(5 downto 0);
        signal w_m5_ctrl        : std_logic_vector(5 downto 0);

begin

-- ************************************** --
communicator : motor_communication_interface
port map(

        clk             => w_sys_clk,
        reset           => w_reset,

        -- ***** Calling interface ***** --
        IOselect        => IOselect,            --w_IOselect,
        IOwrite         => IOwrite,             --w_IOwrite,
        IOread          => IOread,              --w_IOread,
        Data_bus        => Data_bus,            --w_Data_bus, -- Kobler
        portene direkte isteden. Funka!!!
        Interrupt_out   => Interrupt_out,       --w_Interrupt,

        -- ***** Motor interface ***** --
        m1_direction    => w_m1_direction,
        m1_throttle     => w_m1_throttle,
        m1_current_speed=> w_m1_speed,

        m2_direction    => w_m2_direction,
        m2_throttle     => w_m2_throttle,
        m2_current_speed=> w_m2_speed,
```

```
        m3 direction    => w m3 direction,
        m3 throttle     => w m3 throttle,
        m3_current_speed=> w_m3_speed,

        m4 direction    => w m4 direction,
        m4 throttle     => w m4 throttle,
        m4_current_speed=> w_m4_speed,

        m5 direction    => w m5 direction,
        m5 throttle     => w m5 throttle,
        m5_current_speed=> w_m5_speed
);
-- ************************************* --


-- ************************************* --
arbiter : arbiter_interface
port map(
        Clock           => w sys clk,
        Reset           => w_reset,

        Dividend 1      => w m1 dividend,
        Dividend 2      => w m2 dividend,
        Dividend 3      => w m3 dividend,
        Dividend 4      => w m4 dividend,
        Dividend_5      => w_m5_dividend,

        Divisor 1       => w m1 divisor,
        Divisor 2       => w m2 divisor,
        Divisor 3       => w m3 divisor,
        Divisor 4       => w m4 divisor,
        Divisor_5       => w_m5_divisor,

        Req 1           => w m1 req,
        Req 2           => w m2 req,
        Req 3           => w m3 req,
        Req 4           => w m4 req,
        Req_5           => w_m5_req,

        Ack 1           => w m1 ack,
        Ack 2           => w m2 ack,
        Ack 3           => w m3 ack,
        Ack 4           => w m4 ack,
        Ack_5           => w_m5_ack,

        Result 1        => w m1 result,
        Result 2        => w m2 result,
        Result 3        => w m3 result,
        Result 4        => w m4 result,
        Result_5        => w_m5_result,

        Dividend out    => w arb dividend,
        Divisor out     => w arb divisor,
        Result_in       => w_arbiter_result

);
-- ************************************* --


-- ************************************* --
div module : DIV_UNIT
port map(
        CLK             => w sys clk,
        DIVIDEND        => w arb dividend,
        DIVISOR         => w arb divisor,
        DIV_RESULT      => w_arbiter_result     -- Foreløpig direkte koblet
        til alle motorer.
);
-- ************************************* --


-- ************************************* --
```

```
motor_1 : motor_controller_interface
port map(
        sys_clk         => w_sys_clk,
        rpm_clk         => w_rpm_clk,
        pwm_clk         => w_pwm_clk,   --w_sys_clk,
        reset           => w_reset,

        top_hall_in     => w_m1_hall,

        top_duty_cycle  => w_m1_throttle,
        top_rotation_dir=> w_m1_direction,
        --top_com_angle => w_m1_angle,

        top_div_req     => w_m1_req,
        top_div_ack     => w_m1_ack,
        top_dividend    => w_m1_dividend,
        top_divisor     => w_m1_divisor,
        top_result_in   => w_m1_result,

        top_speed_out   => w_m1_speed,
        top_motor_ctrl  => w_m1_ctrl
);
-- ************************************** --


-- ************************************** --
motor_2 : motor_controller_interface
port map(
        sys_clk         => w_sys_clk,
        rpm_clk         => w_rpm_clk,
        pwm_clk         => w_pwm_clk,   --w_sys_clk,
        reset           => w_reset,

        top_hall_in     => w_m2_hall,

        top_duty_cycle  => w_m2_throttle,
        top_rotation_dir=> w_m2_direction,
        --top_com_angle => w_m2_angle,

        top_div_req     => w_m2_req,
        top_div_ack     => w_m2_ack,
        top_dividend    => w_m2_dividend,
        top_divisor     => w_m2_divisor,
        top_result_in   => w_m2_result,

        top_speed_out   => w_m2_speed,
        top_motor_ctrl  => w_m2_ctrl
);
-- ************************************** --


-- ************************************** --
motor_3 : motor_controller_interface
port map(
        sys_clk         => w_sys_clk,
        rpm_clk         => w_rpm_clk,
        pwm_clk         => w_pwm_clk,   --w_sys_clk,
        reset           => w_reset,

        top_hall_in     => w_m3_hall,

        top_duty_cycle  => w_m3_throttle,
        top_rotation_dir=> w_m3_direction,
        --top_com_angle => w_m3_angle,

        top_div_req     => w_m3_req,
        top_div_ack     => w_m3_ack,
        top_dividend    => w_m3_dividend,
        top_divisor     => w_m3_divisor,
        top_result_in   => w_m3_result,

        top_speed_out   => w_m3_speed,
```

```
        top_motor_ctrl  => w_m3_ctrl
);
-- ************************************ --


-- ************************************ --
motor 4 : motor_controller_interface
port map(
        sys_clk         => w_sys_clk,
        rpm_clk         => w_rpm_clk,
        pwm_clk         => w_pwm_clk,   --w_sys_clk,
        reset           => w_reset,

        top_hall_in     => w_m4_hall,

        top_duty_cycle  => w_m4_throttle,
        top_rotation_dir=> w_m4_direction,
        --top_com_angle => w_m4_angle,

        top_div_req     => w_m4_req,
        top_div_ack     => w_m4_ack,
        top_dividend    => w_m4_dividend,
        top_divisor     => w_m4_divisor,
        top_result_in   => w_m4_result,

        top_speed_out   => w_m4_speed,
        top_motor_ctrl  => w_m4_ctrl
);
-- ************************************ --


-- ************************************ --
motor 5 : motor_controller_interface
port map(
        sys_clk         => w_sys_clk,
        rpm_clk         => w_rpm_clk,
        pwm_clk         => w_pwm_clk,   --w_sys_clk,
        reset           => w_reset,

        top_hall_in     => w_m5_hall,

        top_duty_cycle  => w_m5_throttle,
        top_rotation_dir=> w_m5_direction,
        --top_com_angle => w_m5_angle,

        top_div_req     => w_m5_req,
        top_div_ack     => w_m5_ack,
        top_dividend    => w_m5_dividend,
        top_divisor     => w_m5_divisor,
        top_result_in   => w_m5_result,

        top_speed_out   => w_m5_speed,
        top_motor_ctrl  => w_m5_ctrl
);
-- ************************************ --




w_sys_clk <= sys_clk;
w_rpm_clk <= rpm_clk;
w_pwm_clk <= w_pwm_clk;
w_reset   <= reset;

w_m1_hall <= m1_hall_sensors;
w_m2_hall <= m2_hall_sensors;
w_m3_hall <= m3_hall_sensors;
w_m4_hall <= m4_hall_sensors;
w_m5_hall <= m5_hall_sensors;

m1_control <= w_m1_ctrl;
m2_control <= w_m2_ctrl;
```

```
m3_control <= w_m3_ctrl;
m4_control <= w_m4_ctrl;
m5_control <= w_m5_ctrl;

--w_data_bus <= Data_bus;
--Data_bus <= w_data_bus;

end architecture;
```