AAUSAT4 – Generic Flight Computer

Juni 2013

Projektrapport

Gruppe 416

Aalborg Universitet
Institut for Elektroniske Systemer
Fredrik Bajers Vej 7
DK-9220 Aalborg

AALBORG UNIVERSITET
STUDENTERRAPPORT

Institut for Elektroniske Systemer
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://es.aau.dk

Titel:
AAUSAT4 – Generic Flight Computer

Tema:
Design af digitale systemer

Projektperiode:
Forårssemestret 2013

Projektgruppe:
416

Deltager(e):
Henrik Aarup Vesterager
Jacob Møller Hjerrild Hansen
Søren Bøgeskov Nørgaard
Malthe Bilgram Jensen
Igor Aleksandrovich Syrytsin
Lasse Thomsen

Vejleder(e):
Jens Dalsgaard Nielsen

Oplagstal: 2

Sidetal: 168

Afleveringsdato: 30. maj 2013

Synopsis:

I denne rapport udvikles et LOG-modul til AAUSAT4, med udgangspunkt i udviklingen af en Generic Flight Computer.

Projektet tager del i Aalborg Universitets AAUSAT4 projekt, hvor det ønskes at der laves en Generic Flight Computer, på en FPGA. Fordelene ved sådant et design belyses i henhold til fokus på muligheden for at opnå optimeret ekseveringshastighed og et minimeret effektforbrug.

LOG-systemet er designet således at det indeholder et logbibliotek som kan modtage statusbeskeder fra delmoduler på AAUSAT4 og kommunikerer med et S25FL128S flashmodul med dertil udviklet softwaredriver. Kommunikationen går via et dertil udviklet software SPI-bibliotek med tilhørende hardwaremodul, designet i VHDL. Der er udviklet et software I^2C bibliotek som tiltænkes at redesignes til et hardwaremodul for at optimere den interne kommunikationshastighed.

For at imødekomme dette, er det valgt at benytte en MicroBlaze MCS mikrocontroller, da denne kan implementeres på en FPGA, og opfylder de krav som er stillet til satellittens CPU og til delsystemernes grænseflader, samt til kompatibiliteten med kodesproget C.

Denne rapport vil afspejle, hvorledes det har været muligt at konstruere et grundlæggende system til implementering af subsystemer til en satellit, eksemplificeret ved design, udvikling og implementering af et protoype LOG-subsystem på en MicroBlaze MCS. Denne prototype indeholder SPI-bibliotek og hardware, flashbibliotek, I^2C-bibliotek og log-kode, der benytter sig af disse. Der har været udfordringer ved implementering, og et samlet LOG-modul er derfor ikke funktionelt.

Henrik Aarup Vesterager | Jacob Møller Hjerrild Hansen | Søren Bøgeskov Nørgaard

Malthe Bilgram Jensen | Igor Aleksandrovich Syrytsin | Lasse Thomsen

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Forord

Denne rapport er udarbejdet af gruppe 416 i samarbejde med Aalborg Universitet, i foråret 2013. Formålet med projektet er at påbegynde udviklingen af AAUSAT4, mere specifikt, udviklingen af en Generic Flight Computer implementeret på en FPGA. Da en Generic Flight Computer er et abstrakt system, tages der i denne rapport udgangspunkt i udviklingen af et specifikt delelement, nemlig et LOG-delmodul til AAUSAT4. Dette beskrives igennem en case, med seperat kravsspecifikation, udvikling, testspecifikation, test og implementering. Gennem denne rapport bliver der beskrevet forskellige idéer og metoder til udviklingen af en Generic Flight Computer, samt endeligt valg og konstruktion af et LOG-delmodul. Hensigten er at vise hvordan et delmodul kan implementeres på en FPGA som del af en Generic Flight Computer.

Der skal rettes en særlig tak til Jens Frederik Dalsgaard Nielsen for vejledningen til dette projekt.

Læsevejledning

Opdeling: Rapporten påbegyndes med en indledning samt et initierende problem, der fører til en problemanalyse. I problemanalysen analyseres AAUSAT3 for at give et overblik over hvilke delmoduler, som indgår i en satellit. På baggrund af analysen opstilles en problemformulering, som beskriver hvad der skal opnås med projektet. Herefter opstilles en funktionel kravspecifikation til Generic Flight Computeren, hvorefter de specifikke krav først bliver opstillet til Case: LOG. Efter den funktionelle beskrivelse af Flight Computeren, påbegyndes Case: LOG, som har sin egen kravspecifikation, samt beskrivelse af testspecifikation. Efter dette afsnit, påbegyndes udviklingen af LOG-delmoduler, samt beskrivelse og test af disse delmoduler. Herefter påbegyndes implementeringsfasen, og hvert delmodul sammensættes og testes på systematisk vis, dette ender ud i en accepttest og delkonklusion af casen. Casen afsluttes, og der gives en overordnet konklusion på problemformuleringen, og om mulighederne for at implementere delmoduler på en FPGA.

Bilag Gennem projektet er der lavet bilag og arbejdsblade, disse er vedlagt bagerst i rapporten og der vil være referencer til disse gennem rapporten. Der er sammen med rapporten vedlagt en bilags-CD, som vil indeholde alle bilag, anvendt i rapporten, samt alt kode udviklet gennem projektet. Når der refereres til bilags-CD'en vil følgende symbol bruges

`path/to/stuff`

hvor `path/to/stuff` er stien til indholdet fra CD'ens rod.

Kildeangivelse Alle kilder i rapporten er udformet efter nummer-metoden, og vil se ud som følgende: [kildenummer]. Kildehenvisninger der står før et afsluttende punktum, gælder for pågældende sætning. Står henvisningen efter et afsluttende punktum, gælder kildehenvisningen for det påbegyndte afsnit til selve kildehenvisningen. En litteraturliste over kilderne, vil være at finde til sidst i rapporten

Indhold

Indhold **iii**

1 Indledning **1**
1.1 Initierende problem . . . 1

2 Problemanalyse **3**
2.1 Formålet med AAUSAT4 . . . 3
2.2 Analyse af AAUSAT3 . . . 3
2.2.1 Platform . . . 4
2.2.2 Payloads . . . 5
2.2.3 Kriterier for valg af mikrocontroller på AAUSAT3 . . . 5
2.2.4 Hukommelse . . . 6
2.2.5 Intern kommunikation . . . 6
2.3 FPGA . . . 7
2.3.1 Softcore processors . . . 7
2.4 Målgruppe . . . 8
2.5 Miljø . . . 8

3 Problemformulering **10**

4 Kravspecifikation **11**
4.1 Generel beskrivelse . . . 11
4.1.1 Xilinx Spartan-3 Starter Board . . . 11
4.1.2 Blokopdeling . . . 12
4.1.3 Mikrocontroller . . . 12
4.1.4 Ekstern kommunikation . . . 13
4.1.5 Intern kommunikation . . . 14
4.1.6 Hukommelse . . . 14
4.1.7 Logik . . . 15
4.2 Funktionelle krav . . . 15
4.2.1 Mikrocontroller . . . 15
4.2.2 Ekstern kommunikation . . . 16
4.2.3 Intern kommunikation . . . 16
4.2.4 Hukommelse . . . 16
4.2.5 Logik . . . 16
4.3 Specifikke krav . . . 16

I Case: LOG **18**

5 Kravspecifikation **21**
5.1 LOG . . . 21
5.1.1 Funktionelle krav . . . 21
5.1.2 Specifikke krav for LOG . . . 21
5.2 SPI . . . 23
5.2.1 Funktionelle krav . . . 23
5.2.2 Specifikke krav . . . 23
5.3 Flash . . . 24
5.3.1 Funktionelle krav . . . 24
5.3.2 Specifikke krav . . . 25
5.4 I^2C . . . 25
5.4.1 Funktionelle krav for I^2C . . . 26
5.4.2 Specifikke krav for I^2C . . . 26

6 Testspecifikation **27**
6.1 LOG . . . 27
6.1.1 Test af funktionelle krav . . . 27
6.1.2 Test af specifikke krav . . . 27
6.2 Testspecifikation til SPI . . . 28
6.2.1 Test af funktionelle krav . . . 28
6.2.2 Test af specifikke krav . . . 28
6.3 Flash . . . 28
6.3.1 Test af funktionelle krav . . . 28
6.3.2 Test af specifikke krav . . . 28
6.4 I^2C . . . 29
6.4.1 Unit test . . . 29
6.4.2 Global test . . . 29
6.4.3 Test med Arduino som slave. . . . 30

7 Udvikling **33**
7.1 LOG . . . 33
7.1.1 Databeskrivelse . . . 33
7.1.2 Kommandobeskrivelse . . . 34
7.1.3 Kode . . . 35
7.2 SPI . . . 43
7.2.1 SPI opbygning . . . 43
7.2.2 Master . . . 43
7.2.3 Analyse af open source modul . . . 43
7.2.4 Driver . . . 48
7.2.5 Funktionelle krav til driver . . . 48
7.2.6 Grænseflader . . . 50
7.2.7 Kode . . . 51
7.2.8 Deltest . . . 53
7.2.9 IP blok . . . 53
7.2.10 SPI C bibliotek . . . 58
7.2.11 SPI C bibliotek . . . 60

7.2.12 Implementering af hardware og software . . . 62
7.2.13 Indførelse af SR-latch . . . 64
7.2.14 Opsummering af hardwareændringer . . . 71
7.2.15 Opsummering af softwareændringer . . . 71
7.3 Flash . . . 73
7.3.1 Hardware interface . . . 73
7.3.2 Software interface . . . 77
7.4 I^2C . . . 87
7.4.1 Baggrunden for I^2C . . . 87
7.4.2 I^2C– kode . . . 90

8 Test **97**
8.1 LOG . . . 97
8.1.1 Test på PC, version 1 . . . 97
8.1.2 Test på MicroBlaze, version 1 . . . 101
8.1.3 Testresultater . . . 101
8.1.4 Test på PC, version 2 . . . 102
8.1.5 Test på MicroBlaze, version 2 . . . 102
8.2 SPI . . . 102
8.2.1 Test af software . . . 102
8.3 Flash . . . 105
8.3.1 Test med Arduino . . . 105
8.3.2 Test 1 . . . 106
8.4 I^2C . . . 109
8.4.1 Test af I^2C . . . 109
8.4.2 Testresultater . . . 110

9 Implementering **121**
9.1 Trin 1: Flash og SPI . . . 121
9.1.1 Beskrivelse af kode . . . 121
9.1.2 Implementeringstests . . . 122
9.2 Trin 2: Flash, SPI og LOG . . . 133
9.2.1 Beskrivelse af kode . . . 133
9.2.2 Afslutning på Case . . . 136

II Generic flight computer **137**

10 Accepttest **138**

11 Konklusion **140**

A Kode **142**
A.1 LOG-kode, version 2 . . . 142

B Flash **155**

C Arbejdsblade **157**
C.1 Clock Divider . . . 157

C.2 Komponentfil 158
C.3 Microblaze - Hello world UART program 160
C.4 Dual microblaze 163
C.5 Programmering af Flash PROM 166

Litteratur **168**

1 Indledning

Denne rapport indeholder dokumentation af udviklingen og konstruktionen af projektforslaget AAUSAT4's Generic Flight Computer (GFC).

AAUSAT projektet, er et studenterprojekt udviklet og konstrueret på Aalborg Universitet. Projektet har været igang siden 2001, hvor udviklingen af AAU-CubeSat blev påbegyndt og siden da, har projektet omhandlet udviklingen af to pico-satellitter: AAUSAT2 og AAUSAT3. AAUSAT2 satellitten blev opsendt i 2008 og er nu i 2013 stadigvæk aktiv. Den senest udviklede satellit, AAUSAT3, blev opsendt i starten af indeværende år og kommunikerer på nuværende tidspunkt, efter 90 dage, uden problemer. Efter opsendelsen af AAUSAT3, er udviklingen af den fjerde version, AAUSAT4, påbegyndt.

I dette projekt dokumenteres udviklingen og konstruktionen af en GFC til AAUSAT4, der følger studieordningens rammer indenfor projektet. Derudover vil der også lægges vægt på, at projektet følger målene for 4. semesterkurserne, således at kurserne supplerer med den nødvendige teori igennem projektperioden.

For at opnå den bedst mulige sammenhæng mellem kurser og projektet, er det derfor valgt at anvende en FPGA (Field Programmable Gate Array) i projektet. Dette valg er også taget på baggrund af, at det fra AAUSAT's projektforslag foreslås, at der anvendes en FPGA i udviklingen af projektet og for at se om det er muligt.

For at udvikle en GFC til AAUSAT4-satellitten, er der søgt inspiration fra AAUSAT3-projektet. Problemanalysen indeholder derfor en analyse af AAUSAT3-satellitten, for at skabe et overblik over funktionerne af denne. Derudover vil problemanalysen indeholde en beskrivelse af formålet med AAUSAT4 og en analyse af flight controllerens målgruppe.

1.1 Initierende problem

Det initierende problem har til formål at initialisere en problemanalyse, som skal afklare hvilke funktioner og krav der kan stilles til en Generic Flight Computer.

Der tages i høj grad udgangspunkt i AAUSAT3, og det undersøges derfor hvilke forskelle der er på AAUSAT4 og AAUSAT3. Det undersøges i øvrigt, hvordan GFC'en vil blive brugt senere i AAUSAT4's udvikling.

Problemet er formuleret som følgende spørgsmål:

- Formålet med AAUSAT4:
  - Hvad er formålet med AAUSAT4?
- Analyse af AAUSAT3:
  - Mikroprocessor: Hvilken type processor er anvendt i AAUSAT3, og skal en tilsvarende anvendes til AAUSAT4?

  - Kommunikation: Hvilke grænseflader kommunikerer AAUSAT3 igennem, og skal AAUSAT4 kunne anvende samme grænseflader?
  - Hukommelse: Hvilken type hukommelse (fast lager, midlertidig) er implementeret, og skal disse typer implementeres på AAUSAT4?

- Målgruppe:
  - Hvem er målgruppen for, og skal senere arbejde med, GFC'en?

2 Problemanalyse

2.1 Formålet med AAUSAT4

AAUSAT4 er Aalborg Universitets fjerde projekt med satellitter, hvor studerende arbejder sammen i udviklingen af CubeSats. Under udviklingen af AAUSAT4, deltager Aalborg Universitet i et udviklingsprojekt der hedder QB50, hvilket er et internationalt projekt omhandlende udviklingen af 50 satellitter, som er en blanding af studentprojekter samt professionelt udviklede payloads (satellittens last). QB50 projektet samarbejder med FP7, som er et EU projekt kaldet det 7. rammeprogram. Igennem dette kan de forskellige CubeSat projekter søge økonomiske midler til at financiere udviklingen af deres CubeSat [15].

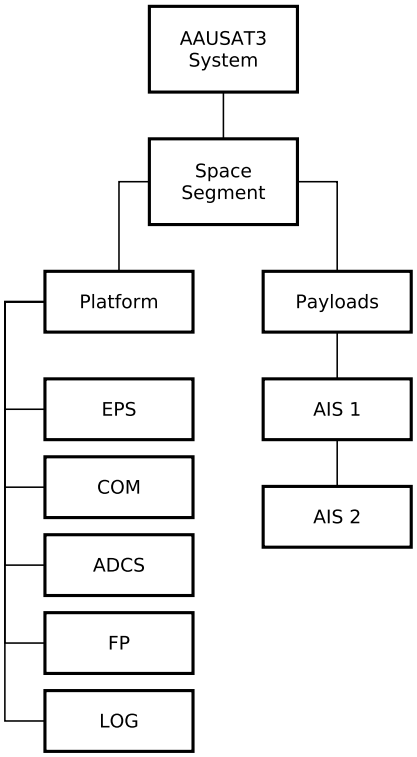
Som tidligere nævnt lægger Aalborg Universitet vægt på, at AAUSAT4 udelukkende udvikles af de studerende. Der er derfor blevet fremstillet forskellige projektforslag til henholdsvis 4, 6 og 8 semester på Elektronik & IT-Ingeniør studiet. Alle projektforslag har til formål, at de studerende skal udvikle kerneelementer til AAUSAT4. Meningen er, at de studerende selv skal komme med forslag til hvordan de forskellige projekter skal løses, så de på den måde får større erfaring med ingeniørarbejdet.

Projektet QB50 vil opsende CubeSat satellitter af større dimensioner end tidligere. En typisk CubeSat satellit har dimensioner på $10 \times 10 \times$ cm. QB50 projektet vil opsende dobbelte CubeSat satellitter, altså CubeSats som er sat sammen, for på den måde at have en større payload med. AAUSAT4 vil blive en af disse dobbelt CubeSats. CubeSat'en, der bliver udviklet på Aalborg Universitet, vil blive koblet sammen med en CubeSat der indeholder en payload, som er købt gennem QB50 projektet, med støtte fra EU's 7. rammeprogram. Denne payload indsamler og behandler selv sit data, men benytter sig af AAUSAT4's kommunikationsudstyr til at transmiterer den indsamlede data til jordstationer tilknyttet projektet. Så ved at sætte flere CubeSats sammen, kan der medbringes større payloads og på den måde fås mere ud af CubeSat-opsendelserne.

Forgængeren AAUSAT3 indeholder AIS som den primære payload, der kort fortalt er et tracking system der benyttes til skibe. Et af de foreslåede mål med den kommende AAUSAT4, er at have en nyere og bedre udgave af den foregående AIS som payload, da AIS i rummet på nuværende tidspunkt er et varmt emne. Aalborg Universitet vil sammen med Farvandsstyrelsen fremstille et forslag til den næste generation af AIS og implementere AIS'en i AAUSAT4 satellitten. Indholdet af satellitten er dog ikke bestemt endnu, udover at den skal stå for kommunikationen til jorden.

2.2 Analyse af AAUSAT3

I det følgende undersøges AAUSAT3, således at delmodulernes funktionalitet beskrives og der undersøges hvilke kriterier der har dannet grundlag for valg af mikrocontroller.



Figur 2.1: AAUSAT3, delmoduler.

2.2.1 Platform

EPS

Den elektroniske strømforsyning EPS (Electronic Power Supply), skal, som navnet antyder, leverer de nødvendige spændinger og strømme til alle satellittens subsystemer. Strømforsyningen er i stand til at kontrollere de individuelle forsyninger til subsystemerne, samt overvåge disse. Det er også EPS'en, som står for opladning af batterier fra solcellerne placeret på satellitten.[14]

COM

COM er det modul der benyttes til at kommunikere med jorden. På AAUSAT3 kommunikeres der over UHF-båndet (Ultra High Frequency, mellem 300 MHz og 3000 MHz [16]) med Forward Error Correction.

Systemet er opbygget således, at alle subsystemer kan kommunikerer med jorden – både starte en kommunikation til jorden, og modtage kommandoer fra jorden, direkte.[14]

ADCS

Det primære formål med ADCS (Attitude Determination Control System) systemet er at stabilisere satellittens opførsel i forhold til dens rotationskraft, efter den er frigivet fra raketten. Det er ligeledes ADCS-systemets ansvar at sørge for finjustering af satellittens altitude. Til dette formål anvendes sensorer og aktuatorer som sikrer stabilitet[14].

FP

FP'ens (Flight Planner) funktion er at styre, hvornår de forskellige subsystemer skal tændes eller slukkes. Der er to muligheder for at planlægge styringen af subsystemer igennem flightplanneren. Flightplanneren kan have preprogrammerede kommandoer, som tænder eller slukker subsystemer på faste tidspunkter. Derudover er der også mulighed, for at planlægge styringen af flightplanneren fra jordstationen via CAN kommunikation, så der kan planlægges nye tænd- eller slukketider, for de forskellige subsystemer.

Da flightplanneren er en vigtig del af AAUSAT3, skal det være muligt at lagre CAN-beskeder fra jordstationen, så der i tilfælde af kommunikationsafbrydelse, ikke opstår problemer med afvikling af CAN-beskederne.

LOG

Loggen gemmer log oplysninger fra alle subsystemer og gemmer dem i loggens hukommelse. Disse logoplysninger indeholder info om status på de forskellige subsystemer, såsom advarsler eller fejl. I loggen påføres der et tidspunkt, for hvornår de enkelte oplysninger er blevet gemt, samt de inddeles i kategorier, så jordstationen kan bede om bestemte oplysninger fra et givent tidspunkt. loggen bruges kun til at gemme system oplysninger og altså ikke data fra sensorer etc.

2.2.2 Payloads

AIS

AIS (Automatic Identification System) er et automatiserende sporingssystem, som anvendes primært af skibstrafiktjenester for at overvåge skibenes placering. På AAUSAT3 er indbygget et AIS1-modul, som er en AIS modtager. Den er baseret på ADF7021-serie integreret radio, hvilket er indbygget i AT90CAN128 mikrocontroller.

2.2.3 Kriterier for valg af mikrocontroller på AAUSAT3

Flere subsystemer på AAUSAT3, har sin egen mikrocontroller og disse subsystemer, har derfor ikke brug for at låne regnekraft fra hinanden.

Valget af mikrocontroller tages på baggrund af krav om processorkraft fra det enkelte subsystem. Ligeledes er en af de vigtigste parametre effektforbruget. I effektforbrugssammenhæng skal der undersøges hvilke subsystemer, der er de store forbrugere. CSP modulet (CAN Space Protocol) bruger 30 % af processorkraften ved 8 MHz[2].

Valget af mikrocontroller udføres på baggrund af følgende parametre:[2]):

- Processor-arkitektur: Vigtig parameter med hensyn til udviklingsværktøj og hvor mange clock-cykler der bruges på udførsel af instruktioner.
- Processor-hastighed: Den maksimale hastighed, som processoren kan operere ved er afgørende for om processoren kan udføre alle de ønskede opgaver.
- Mikrocontrollerens hukommelse: Det er vigtigt at der internt og eksternt kan opnås den fornødne hukommelse, volatil såvel som non-volatil.
- Antal I/O pins tilgængelig til ekstern hardware: Antallet af pins er vigtigt for at kunne betjene det eksterne hardware der ønskes.
- USART-kompatibel: Der bliver anvendt en USART forbindelse til COM-U modulet.
- Temperaturtolerance: Satellitten bliver anvendt i rummet hvor store temperaturudsving kan forekomme.

Den anvendte mikrocontroller på AAUSAT3 er en AT90CAN128. Denne kører med en clockfrekvens på 8 MHz og kan klare driftstemperaturer op til 125 °C. De vigtigste overvejelser i valget af AT90CAN128 har været effektforbruget, hvor clockfrekvensen er holdt så lav som muligt og den lave temperaturfølsomhed.[2]

2.2.4 Hukommelse

AAUSAT3 platformen har forskellige hukommelsestyper, både ekstern og intern.

Den interne hukommelse er bestemt af den valgte mikrocontroller, som er af typen AT90CAN128. Denne mikrocontroller har en intern flashhukommelse på 128 KiB, intern S-RAM hukommelse på 4 KiB og 16 arbejdsregistre.

Da der ikke er nok plads i den interne hukommelse til både logning og Flight Planner [2], er der implementeret noget eksternt hukommelse. Den eksterne hukommelse er implementeret på to forskellige non-volatile hukommelseschips på 65 KiB.

Der er brugt en latch fra 74AHC-serien. Derudover er der valgt, at anvende ekstern hukommelse på 64 KiB, da denne hukommelse kan håndteres af mikrocontrolleren med 16 adresselinjer. Temperaturen i satellitten, kan variere fra -10 °C til 60 °C[9]. Derfor er det vurderet at alle komponenter i satellitten, skal kunne holde til en temperatur på 60 °C. I det færdige design bruges en AT29LV512 hukommelseskreds, der virker ved op til 85 °C og kan læse fra kredsen mens temperaturen er 125 °C.

2.2.5 Intern kommunikation

AAUSAT3 er opbygget af flere delmoduler, og disse må nødvendigvis kommunikere internt. Dette gøres over nogle standardiserede kommunikationsbusser med fastlagte protokoller. I følgende afsnit vil der kigges på hvilke busser og protokoller, der anvendes i AAUSAT3.

CAN bus

Til kommunikation mellem delmoduler i AAUSAT3 bruges blandt andet CAN (Controller Area Network) bus, hvilket er en teknologi primært udviklet til køretøjer, men nu også anvendt i mange andre områder heriblandt rumfart. CAN bus er designet til at være meget robust, og til at kunne modstå barske forhold.

Bussen er en differentialbus, og består fysisk af kun to ledere hvor signalerne vil ses som et spændingsdifferentiale mellem disse. Dette er blandt andet med til at gøre bussen modstandsdygtig overfor elektromagnetisk støj.[2]

I AAUSAT3 bruges CAN sammen med en overliggende protokol CSP (CAN Space Protocol), udviklet netop til AAUSAT3[2].

Serial Peripheral Interface Bus

Serial Peripheral Interface (SPI), bruges i AAUSAT3, enkelte steder til kommunikation mellem mikrocontroller og eksterne IC'er. Ved kommunikation over SPI, findes der et master/slave forhold, hvor masteren har mulighed for at sende/modtage data fra en række slaver på bussen. Specifikt anvendes SPI i AAUSAT3 til kommunikation mellem en mikrocontroller og en port-udvider[2]. Fysisk består SPI af fire ledninger, hvor man har to til dataoverførsel, samt en clock og en slave-select. En af fordelene ved SPI er fuld dupleks kommunikation.

I^2C

I AAUSAT3 bruges også I^2C-interface (hvis interface også kaldes TWI). Det bruges i satellitten enkelte steder blandt andet til at måle temperaturen. Bussen er bi-direktional og den tillader op til 127 enheder koblet på samme

bus. Den benytter sig, som navnet antyder, kun af to linier – en til clockfrekvensen og en til data. Der er ligesom med SPI et master/slave hierarki, hvor det er masteren som leverer clocken, og starter/stopper dataoverførslen.

2.3 FPGA

Field-programmable gate array (FPGA) er et integreret kredsløb, som gør det muligt at programmere på hardware/gate niveau med et højniveau sprog, som VHDL eller Verilog. Der er mange fordele ved at implementere specifikke delkredsløb i hardware, frem for at udføre det i software, fordi software er mere krævende for CPU'en og derfor bruger flere clock-perioder til afvikling af funktioner.

Når der udvikles hardware til FPGA's bygges der såkaldte IP-blokke (Intellectual property), hvilket giver et højere abstraktionsniveau i forhold til det nedre hardwarelag. Der fås bl.a. IP-blokke som indeholder hele mikroprocessorer, såkaldte "Softcores", som derefter kan programmeres i standard C. Der kan således laves flere softcores på en FPGA med konfigurerbar hardware til krævende funktioner.

For at øge effektiviteten, har mange FPGA's indbygget RAM, addere, multipliere og processorer. Der kan være mange fordele ved at bruge en FPGA til specifikke applikationer, en ulempe er dog prisen, idet FPGA's er dyrere end almindelige mikrocontrollere.

2.3.1 Softcore processors

En softcore processor er i sin enkelthed en processor, som er skrevet i Hardware Descriptive Language (HDL), som derefter kan implementeres på en FPGA eller en Application-Specific Integrated Circuit (ASIC). Softcore processorer er meget fleksible og kan tilpasses til specifikke formål, hvor en almindelig mikrocontroller ikke egner sig. Softcore processorer er typisk designet på en FPGA og implementeret på en ASIC i forbindelse med store produktioner. Der findes mange forskellige softcore processorer, men her vil der blive set på de fire oftest brugte inden for området: Nios II fra Altera, MicroBlaze og PicoBlaze fra Xilinx og Xtensa fra Tensilica. Samtidig findes der en del af Open-Source softcore processorer.

- Nios II er en softcore processor fra Altera, baseret på Reduced Instruction Set Computer (RISC), som understøtter 32-bit Instruction Set Architecture (ISA). Der er også indbygget specifikke instruktioner for 64 og 128-bit multiplikation og der kan laves op til 256 nye instruktioner på Nios II.
- Xilinx har to populære softcore processorer: MicroBlaze og PicoBlaze, hvor MicroBlaze er en 32-bit processor og PicoBlaze 8-bit mikrocontroller. Der er indbygget mange funktioner i MicroBlaze softcore processor, hvilke nemt kan vælges eller fravælges ved hjælp af Xilinx Embedded Development Kit (EDK). Xilinx tilbyder også en gratis version af MicroBlaze processoren der har nogle begrænsninger. PicoBlaze, har knapt så mange funktioner, men er til gengæld mere simpel og der kan implementeres mange på en FPGA, hvor man på den måde får en rigtig parallelisering .
- Xtensa er en softcore processor fra Tensilica, som er mest kendt for dens fleksibilitet. Extensa processoren kan udvides med uendelig mange brugerdefinerede instruktioner, som kan tilføjes ved hjælp af Tensilica Instruction Extension (TIE) programmeringssprog.

Der kan konkluderes at hver af disse processorer har sit eget område hvor dets funktioner bliver udnyttet bedst. Nios II kan have den højeste clockfrekvens når den er implementeret på en FPGA. Ydermere har denne processor et antal brugerdefinerbare instruktioner. Dog er Xtensa den mest fleksible af disse softcore processorer, og er beregnet til implementation på både ASIC's og FPGA'er, hvor andre processorer er lavet til implementation på én FPGA. MicroBlaze og PicoBlaze er til gengæld nemme at implementere og der findes gratis versioner af dem, dog er de ikke så fleksible som Nios II eller Xtensa.

2.4 Målgruppe

AAUSAT3 har været udviklet i årene 2007–2013, og da AAUSAT4's udvikling først er startet i 2013, er udviklingen af GFC'en kun det første skridt. Dette betyder, at det må overvejes, hvem der skal arbejde videre med den. I denne sammenhæng er overvejet følgende:

- Hvordan der arbejdes videre med GFC'en.
- Programmeringssprog og metode.
- Dokumentation af grænseflader, pinout og kode.

Når dette projekt er færdigt, forventes det, at GFC'en er en færdigdesignet computer, som der skal kunne implementeres software på. Det er som udgangspunkt ikke meningen, at fremtidige brugere skal dykke ind i hardwaredelen af denne, men nærmere tilslutning af moduler og software.

Programmeringssproget GFC'en skal programmeres i, afhænger af hvilken processor der implementeres på FPGA'en – det forventes at der anvendes en allerede udviklet compiler til denne, til et sprog de studerende på Aalborg Universitet har fået undervisning i. Dette uddybes nærmere i kravspecifikationen.

Dokumentation af FPGA-kode skulle ikke være nødvendig, da den senere skulle kunne anvendes udenom dette, så derfor gøres der ikke særlige overvejelser omkring dette, udover selvfølgelig at gøre koden overskuelig og forståelig for eget vedkommende. Det er selvfølgelig muligt, at FPGA-koden skal åbnes op igen, hvilket også bidrager til velkommenteret kode.

Pinout på FPGA'en er meget afgørende i forbindelse med udlægning af print, men da dette ikke forventes gjort før i slutfasen af AAUSAT4-forløbet, lægges disse ikke fast, men må tydeliggøres i denne rapport, for at senere udviklere let kan finde de forbindelser i FPGA-koden, der skal føres ud på de fysiske ben. Det samme gælder grænsefladerne til senere moduler, der i øvrigt udvikles efter nogle af de standarder og protokoller, der er beskrevet tidligere i problemanalysen.

Denne rapport vil hermed være hoveddokumentation for senere udviklere, der har med flight computeren af gøre.

2.5 Miljø

Når der designes en satellit, som skal fungerer i rummet, er der flere forskellige krav og forbehold, der skal tages højde for i udviklingen. Satellitten bliver påvirket af forskellige faktorer ude i rummet og når den sendes ud i rummet. Derudover skal der også tages højde for systemdesignet af de forskellige delmoduler, når den skal foretage operationer i rummet.

En af de vigtige faktorer, der skal tages hensyn til, er temperatursvingninger i rummet. I følge måledata, fra AAUSAT3, bliver satellitten udsat for temperatursvingninger fra $-10\,^{\circ}\mathrm{C}$ til $60\,^{\circ}\mathrm{C}$.[9]. Forudsat at temperaturen på AAUSAT4, kommer til at svinge i samme interval som for AAUSAT3, skal der tages højde for at alle subsystemer, kan holde til indvendige temperatursvingninger fra $-10\,^{\circ}\mathrm{C}$ til $60\,^{\circ}\mathrm{C}$. Det kan bl.a. være kritisk hvis batteriet bliver udsat for en temperatur på $-10\,^{\circ}\mathrm{C}$, men temperaturen ønskes heller ikke for høj, da solpanelerne er mest effektive, ved en lav temperatur. Herudover er det også vigtigt at tage hensyn til stråling i rummet, da stråling kan blive en kritisk faktor, hvis satellitten skal ligge i en høj bane. AAUSAT3 satellitten ligger i en bane i højden 700–800 km, hvor der ikke er så meget stråling, men hvis det besluttes, at AAUSAT4 satellitten skal ligge i en højere bane, skal der tages hensyn til dette. I dette projekt, hvor der anvendes en FPGA, er det muligt at tage hensyn til stråling, ved at anvende nogen af de nyere chips fra Xilinx. Xilinx har igennem den seneste tid, udviklet en strålingshærdet chip, som er specialfremstillet til rumfarten og er derfor relevant i det større perspektiv[7].

Udover krav til temperatursvingninger og stråling, er der også vigtige forbehold der skal tages, i forhold til opsendelsen og den fysiske tilstand af satellitten ude i rummet. Satellitten skal konstrueres robust, så den kan modstå G-påvirkningen ved opsendelsen og for at kunne holde, til de vibrationer der kan forekomme i rummet.

Ydermere er satellitten meget skrøbelig, overfor tekniske fejl, da det er begrænset hvor meget vedligehold og reparationer der kan udføres, efter satellitten er sendt ud i rummet. Det er derfor vigtigt, at der i designet tages højde for, at satellitten kan fortsætte sine operationer, selvom der opstår en fejl i et af systemerne.[9]

3 Problemformulering

Igennem problemanalysen er det initierende problem i Afsnit 1.1 besvaret. Den indsamlede viden anvendes til at opstille kriterier for at udvikle en hensigtsmæssig løsning, og beskrive hvilke krav der skal overholdes.

Den indsamlede viden giver anledning til udvikling af en computerstruktur, som sikrer et lavt effektforbrug samtidig med at processeringshastigheden er høj. Dette danner grundlaget for den tekniske problemformulering. Den tekniske løsning vil derfor bygge på en undersøgelse af følgende spørgsmål:

> *Hvordan designes en Generic Flight Computer, således denne kan duplikeres på en FPGA og kommunikere internt såvel som eksternt via standardiserede grænseflader?*

Mere specifikt ønskes følgende designet:

- En mikrocontrollerstruktur med biblioteker og IP-blokke, der gør det muligt at kommunikere mellem mikrocontrollere internt på FPGA'en.
- Standardiseret kommunikation ind og ud af FPGA'en, til eksterne moduler som sensorer, ADC'er o.l.

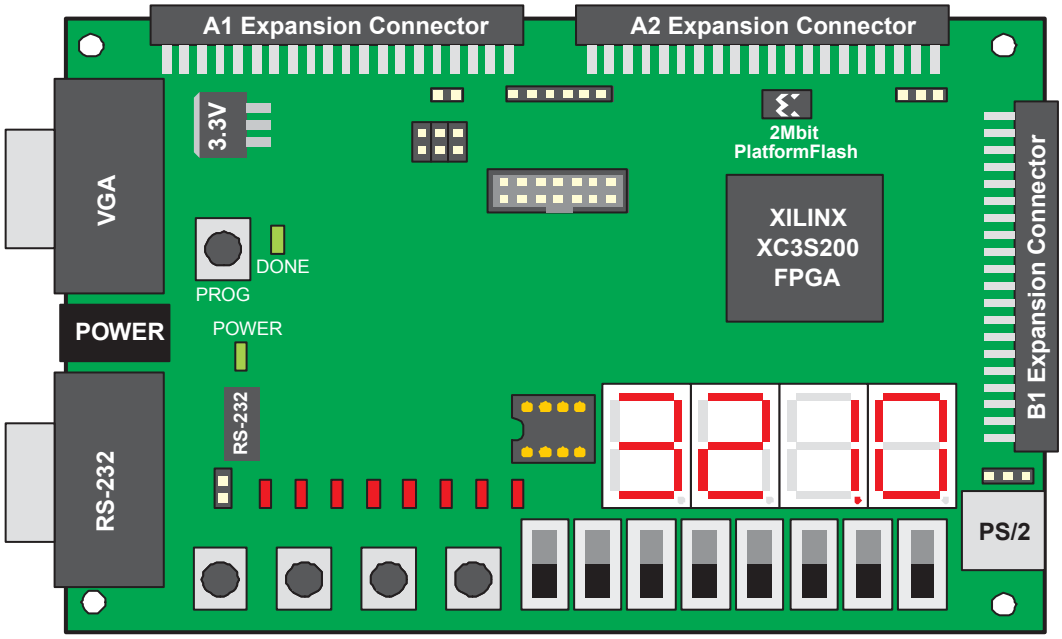
4 Kravspecifikation

4.1 Generel beskrivelse

4.1.1 Xilinx Spartan-3 Starter Board

Der er fra Aalborg Universitet stillet et Xilinx Spartan-3 starter board til rådighed og det vælges derfor at arbejde med dette board.

Spartan-3 boardet er en udviklingsplatform, som bl.a. indeholder en 1000E3 gate FPGA, I/O moduler og 1 MiB hurtig asynkron SRAM. Disse features gør boardet velegnet til udviklingen af alt fra simple logiske kredsløb, til indlejrede processorkerner. Herudover er boardet egnet i dette projekt, da det er kompatibelt med den gratis version af Xilinx ISE udviklingsværktøjet. Se Figur 4.1 for et billede af boardet.[5]



Figur 4.1: Billede af Xilinx Spartan-3 Starter Board (med XC3S200)

Endvidere er der herunder en liste, med de overordnede features på FPGA-boardet.

- Stik:
  - Tre 40 pins udvidelses porte
  - VGA port

  - RS-232 seriel port
  - PS/2 port
- Programmering:
  - Boardet kan arbejde med JTAG3, JTAG USB, JTAG USB Full Speed kabler, P4 og MultiPro kabler fra Xilinx
- Xilinx Spartan-3 FPGA med 18 bit-Multipliers, 216 Kibit blok RAM, og op til 500 MHz intern clock hastighed
- 1000E3 gates FPGA
- 4 Mibit Platform Flash
- 8 slide switches, 4 trykknapper, 9 LEDs og 4 syv-segment displays
- Tre spændingsregulatorer (3,3 V,2,5 V og 1,2 V)
- 1 MiB on-board 10 ns SRAM (256 Kibit x 32)[5]

Bemærk at Spartan FPGA'en, er en XC3S1000. Spartan-3 XC3S1000 FPGA'en, har følgende features:

- System Gates: 1E6
- Logik Blokke: 17 280
- CLB arrays: 1920
- Distribuerede RAM Bits: 120 Kibit
- Blok RAM Bits: 432 Kibit
- Dedikerede Multipliers: 24
- Digitale clock managers: 4
- Max bruger I/O: 391
- Max differential I/O par: 175

[1]

4.1.2 Blokopdeling

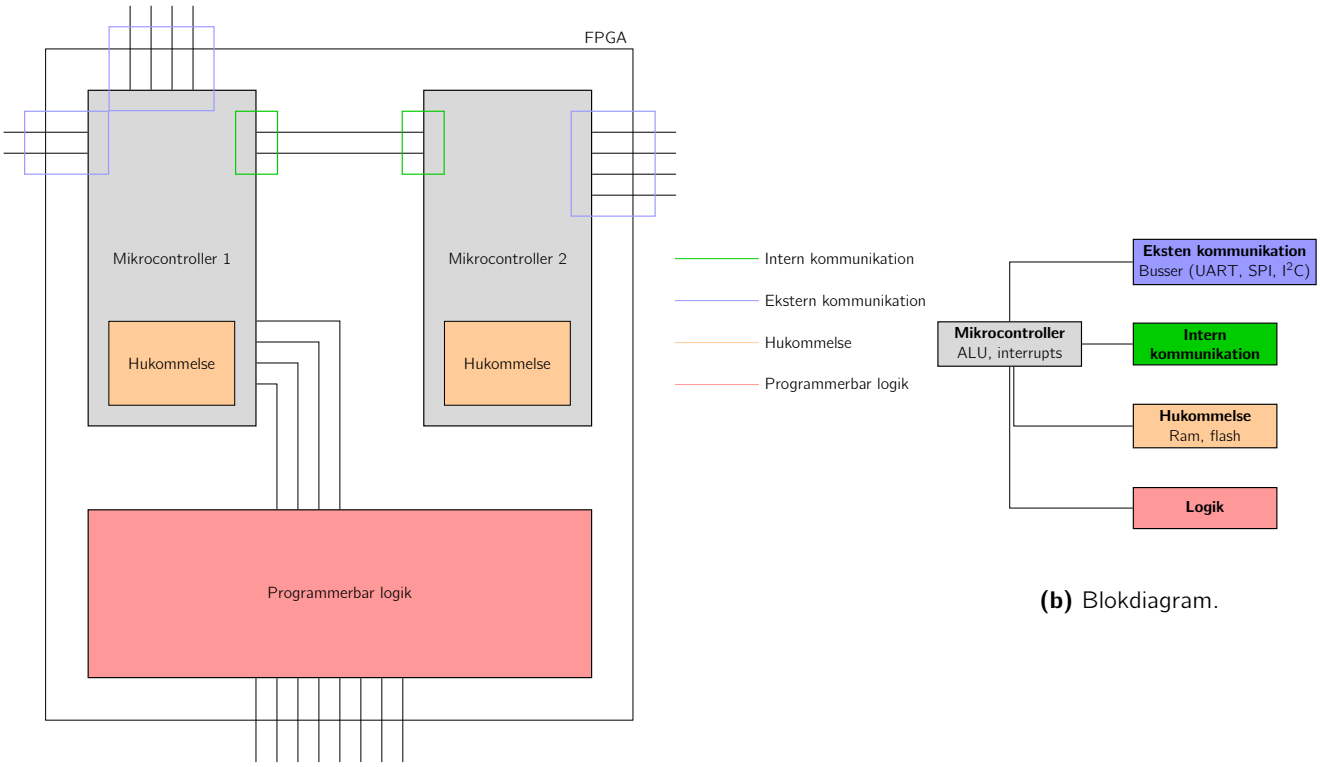
I følgende afsnit vil det beskrives, hvordan flight computeren opdeles, for at kunne implementeres på en FPGA.

I AAUSAT3 benyttedes en AT90CAN128, som udgangspunkt for submodulerne beskrevet i Afsnit 2.1. Formålet med dette projekt er, at udvikle et alternativ til denne mikrocontroller, som kan implementeres på en FPGA – altså skal den udviklede mikrocontroller kunne duplikeres, på samme måde som AT90CAN128'en blev det, men internt på FPGA'en, eller over flere FPGA'er, om nødvendigt.

På Figur 4.2, ses systemet opdelt i blokke (4.2a), og et eksempel på blokkene i brug (4.2b). De enkelte blokkes indhold vil herunder uddybes.

4.1.3 Mikrocontroller

En af de blokke der skal udvikles i forbindelse med dette projekt er en mikrocontroller-blok. Som det fremgår af Figur 4.2, skal den være i stand til at kommunikere internt med logik og mikrocontrollere, samt være i stand



(b) Blokdiagram.

(a) Eksempel på system.

Figur 4.2: Blokopdeling af FPGA.

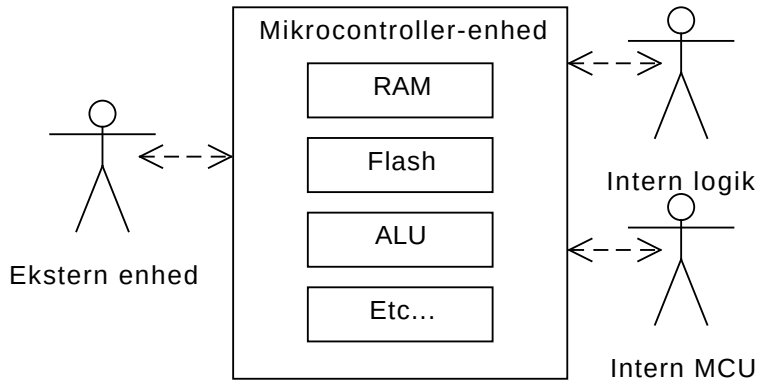
til at kommunikere eksternt gennem digitale I/O porte. Dermed bemærkes det, at der skal kunne indlejres flere mikrocontrollere på en FPGA.

Grænseflader

Softcore mikrocontrolleren, har en række interne og eksterne grænseflader, disse er illustreret på Figur 4.3. Af denne ses det, at der internt i mikrocontrolleren er nogle enheder, såsom timere, flash, ram. Der er på Figur 4.3 også vist de grænseflader mikrocontrolleren har internt i FPGA'en. Her skal den blandt andet kommunikere internt med mikrocontrollere, internt FPGA-logik, samt kommunikere med eksterne enheder som eksempelvis analog/digital konvertere eller CANBUS-transceivere. I tilfælde af at kommunikation via SPI, kan der implementeres hardware, således at SPI bliver designet som en FPGA-logikblok. På denne måde kan en ellers relativ simpel softcore processor generaliseres og udbygges efter behov.

4.1.4 Ekstern kommunikation

Ekstern kommunikation, foregår mellem den anvendte mikrocontroller og eksterne enheder, som eksempelvis RAM, flash og sensorer. Denne kommunikation kan foregå gennem busser, som både kan være parallelle og serielle.



Figur 4.3: Mikrocontroller diagram

Parallelle busser er velegnet til hurtig kommunikation, fordi de har en højere dataoverførselshastighed end serielle forbindelser. Enheder der kommunikerer via parallelle busser, skal sidde tæt sammen, for at undgå busskew.

En ulempe ved parallelle busser er, at der kræves mange signalledninger for at fører data over parallelt.

Serielle busser bruges mellem enheder, som ikke kræver specielt hurtig dataoverførsel. De har den fordel, at der kun bruges få datalinjer til at overføre data, og ved asynkron operation, elimineres problemet med busskew. Serielle busser kræver mere behandling for at udtrække data.

Eksempler på serielle busser er: RS-232, SPI, I^2C og CANBUS.

4.1.5 Intern kommunikation

Intern kommunikation, er kommunikationen inde i FPGA'en imellem de forskellige subsystemer, som kan bestå af en eller flere kerner. For at opnå fordele, ved at have to eller flere kerner, i forhold til at benytte flere CPU'er, er det vigtigt at den interne kommunikation er hurtig og effektiv. Hvis to eller flere subsystemer skal kommunikerer internt på FPGA'en, vil eksekveringstiden af programmerne i høj grad afhænge af kommunikationen. Man ønsker derfor i de fleste tilfælde en hurtig og effektiv intern kommunikation, for ikke at forsinke eksekveringstiden. Dog kan der være særlige krav omkring strømbesparelse, hvorved man kan være nødsaget til at indgå et kompromis mellem effektivitet og strømbesparelse.

4.1.6 Hukommelse

En softcore mikrocontroller i FPGA'en skal have noget hukommelse til rådighed, formentlig både internt og eksternt. Hukommelsen skal bl.a. bruges til at lagre programdata, som mikrocontrolleren skal have mere eller mindre hurtig adgang til. For at mikrocontrolleren skal have hukommelse den har hurtig tilgang til, er det en fordel at have intern hukommelse frem for ekstern hukommelse. Den interne hukommelse er hurtigere end den eksterne da den ligger inde i FPGA'en. Dermed skal et data-request kun sendes igennem få busser. Da den interne hukommelse kan være begrænset i størrelse, kan det være en fordel, at have ekstern hukommelse, da der her er mulighed for at vælge større lagertyper, såsom flash. I flash kan der også gemmes programdata, som skal forblive når strømmen til FPGA'en afbrydes.

4.1.7 Logik

XC3S1000 logikblokke

En FPGA er reprogrammerbart hardware, som indeholder et array af CLB-enheder (Configurable Logic Block). En FPGA kan programmeres til at udføre logiske operationer og dermed logiske netværk, herunder både kombinatoriske og sekventielle netværk. Hardwaren beskrives ved HDL, (Hardware Descriptive Language). HDL gør at hardwaren kan beskrives og konfigureres på programmeringsform, hvilket giver fleksibilitet ved designændringer og højniveau-algoritmer oversættes direkte til specifikke hardware konfigurationer.

På logik-niveau findes der på Spartan-3 boardet 1920 CLB-arrays som hver indeholder et krydsfelt med 4 slices. Dermed indeholder Spartan-3 boardet 7680 slices. Disse CLB-arrays indeholder LUT's (Look Up Table) som anvendes til at implementere logik og lagerelementer. En slice indeholder to logiske funktionsgeneratorer, to lagerelementer, MUX'er, carry-logik og aritmetiske gates. Disse elementer anvendes til logik, beregninger og ROM-funktioner[1]. Med logiske netværk som ikke er clock-styrede kan der opnås de højeste kommunikationshastigheder, hvis de hardware-implementerede algoritmer er designet til at afvikles parallelt. Dette omfatter også parallel eksekvering af flere sekventielle netværk[4]. På logikniveau er kommunikationshastigheden kun begrænset af propageringstiden for de enkelte gates og dermed også begrænset af compilerens evne til at route effektivt.

Logikblok til GFC

I forbindelse med mikrocontrollernes hardware design, betragtes den programmerbare logikblok som et krydsfelt med faste opkoblinger til logiske kontrol-netværk, routing-netværk og beregningsnetværk. Denne blok er opbygget uafhængigt af systemets clock signal. Den er opbygget af simple gates, multiplexere, dekodere, komparatorer og aritmetiske kredsløb[13]. Disse elementer vil altid optræde i en fast konfiguration og vil med faste tilkoblinger til andre system-blokke, også eksternt, udføre operationer på baggrund af signaldetektioner. Med et sådant logisk netværk som ikke er clock-styret, kan der opnås de højeste kommunikations- og eksekveringshastigheder. Optimering af hastigheden udføres ved at designe de hardware-implementerede algoritmer til at eksekverer parallelt.

4.2 Funktionelle krav

I det følgende udarbejdes en beskrivelse af funktionelle- og specifikke krav. Nogle af disse krav vil tage udgangspunkt i AAUSAT3-designet og problemanalysen Afsnit 2.1. Ved at tage udgangspunkt i det tidligere projekt, sikres det at eventuelle delmoduler anvendt på AAUSAT3, kan implementeres på FPGA'en. Dette gøres også for at sikre, at Spartan-3 boardet, kan kommunikere med eksterne moduler.

4.2.1 Mikrocontroller

De funktionelle krav til mikrocontrolleren beskriver hvilke funktioner mikrocontrolleren skal indeholde.

- Skal kunne programmeres i ANSI C.
- Skal indeholde en ALU, som kan multiplicere, subtrahere, addere og udføre logiske operationer, herunder AND, XOR, OR og NOT.
- Det skal være muligt for programmøren at vælge imellem eksterne kommunikationsbusser.
- Skal have mulighed for at have dvale tilstand.

4.2.2 Ekstern kommunikation

De funktionelle krav til ekstern kommunikation, beskriver i høj grad hvilke protokoller som anvendes til kommunikation, ind og ud ad FPGA'en. Dette skyldes, at der skal kommunikeres med standardmoduler (AD-convertere, sensorer mm.), og standardprotokoller skal derfor være understøttede.

De funktionelle krav til ekstern kommunikation er:

- Skal kunne kommunikere serielt via følgende protokoller: SPI, I^2C, RS-232 og CANBUS.
- Skal kunne kommunikere direkte (parallelt) med eksterne moduler.

4.2.3 Intern kommunikation

De funktionelle krav til intern kommunikation beskriver, hvordan der skal kommunikeres internt imellem modulerne i FPGA'en og hvor udbredt denne kommunikation skal være.

De funktionelle krav for intern kommunikation er listet herunder:

- Alle enheder skal have mulighed, for at kommunikere med hinanden.
- Alle enheder skal have ét eller flere unikke ID numre.
- Det skal være muligt for flere enheder at modtage det samme data på en gang.
- Den interne kommunikation skal kunne udvides til flere FPGA'er

4.2.4 Hukommelse

De funktionelle krav til hukommelsen beskriver hvilke muligheder, der skal være for at tilgå hukommelse, både internt og ekstern.

- Skal kunne tilgå intern hukommelse i FPGA'en
- Mulighed for at tilgå eksterne hukommelsesmoduler.
- Mulighed for at gemme programmer i en flash hukommelse, svarende til den på Spartan-3 boardet.
- Den eksterne hukommelse, skal kunne tilgås fra en enhed af gangen.

4.2.5 Logik

De funktionelle krav til FPGA-logik blokkene, beskriver hvilke muligheder brugeren skal have for at lave brugerdefineret logik til operationer, som kræves for at implementere loggen i softwaren.

De funktionelle krav for brugerdefinerede logik-blokke:

- Mulighed for at lave brugerdefinerede FPGA-logikblokke, som kan kobles på mikrocontroller porte eller eksterne I/O porte.

4.3 Specifikke krav

Kravspecifikationen, herunder den generelle beskrivelse og de funktionelle krav, tager udgangspunkt i den overordnede FPGA opbygning. Da de specifikke krav i høj grad skal beskrive kravene til et konkret system, med henblik på en testspecifikation, er det derfor ikke muligt at opstille de specifikke krav endnu. Kravene opstilles

derfor først under i Del I: Case: LOG , hvor der tages udgangspunkt i implementeringen af et konkret system på FPGA'en.

Del I

Case: LOG

Indledning til Case: LOG

Som led i udviklingen af en Generic Flight Computer er det valgt at udvikle et LOG-system til AAUSAT. Dette er gjort for at få mere indsigt i de udfordringer der kan optræde, når der udvikles satellitmoduler, og derved stille krav til GFC'en.

På Figur 4.4 ses et blokdiagram over casen, og de dele der i løbet at Del I (med grønt) vil udvikles. *I anførselstegn under hvert at disse moduler står nogle kortere navne. I de følgende afsnit vil de korte navne benyttes til at beskrive blokkene fra diagrammet. LOG betyder altså hovedprogrammet til loggen, og ikke hele casen.*

Figuren viser desuden hvordan data bevæger sig igennem systemet. Når data skifter navn, foretager en blok en ændring eller sortering i dataene. Præcis hvordan dette sker beskrives under udviklingen af blokkene.

Udviklingen foregår parallelt i tre grupper:

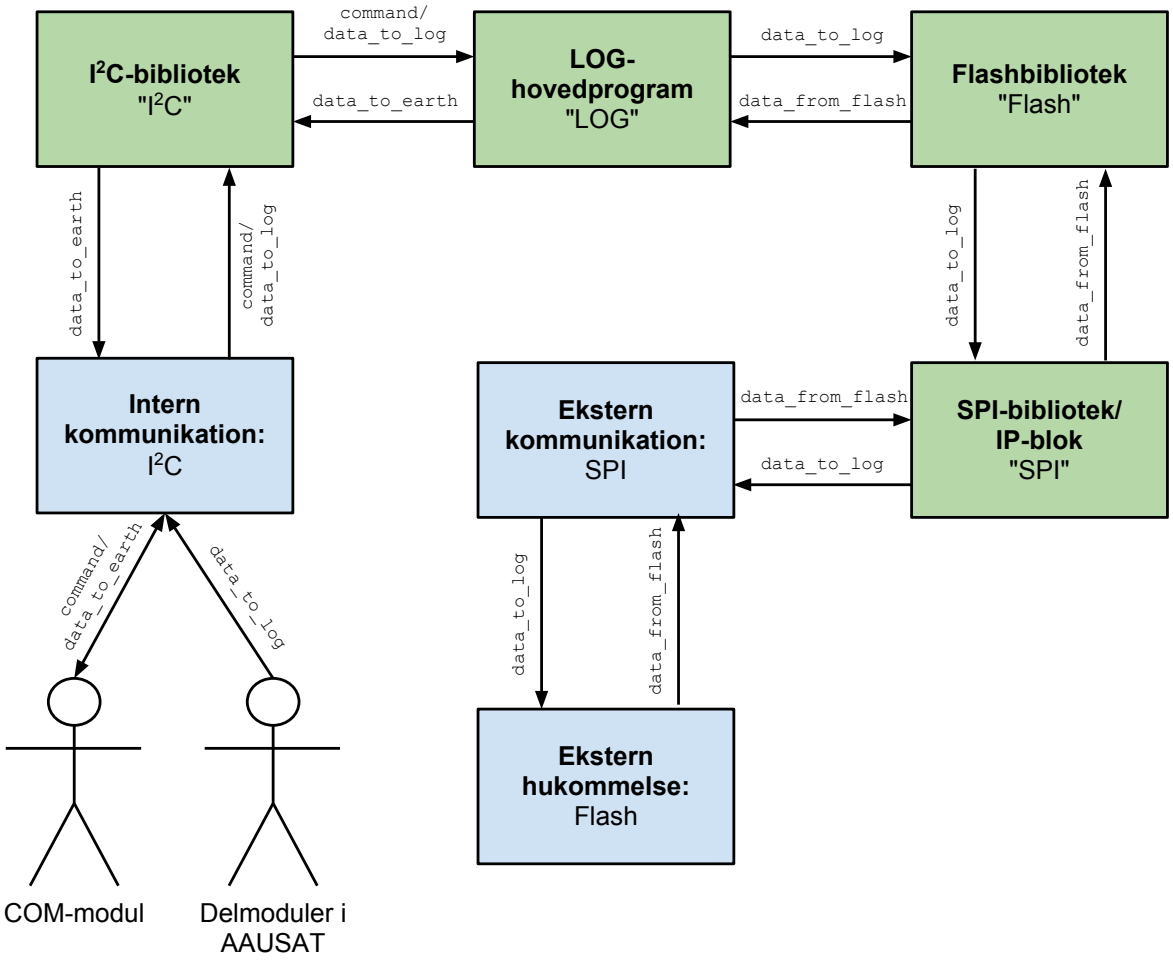
SPI SPI-bibliotek/IP-blok: Interface mellem mikrocontroller og SPI-slaver. Denne designes i hardware, men med biblioteker der giver programmører en simpel grænseflade til SPI-slaver.

Flash Flashbibliotek: Grænseflade mellem LOG-modulet og flashen, via ovennævnte SPI-bibliotek. Flashbiblioteket udvikles først på en Arduino, og implementeres senere med SPI-biblioteket.

LOG LOG-hovedprogram: Programmet har to funktioner: At logge data, som delmoduler på AAUSAT'en gerne vil gemme og at sende data til jorden, afhængigt at hvilken data der ønskes (givet ved en kommando).

Udviklingen af I^2C-biblioteket er overladt til den gruppe der først bliver færdigt, og udvikles derfor senere end de tre andre.

Det er valgt at benytte en MicroBlaze MCS fra Xilinx som mikrocontroller til projektet, da det udleverede udviklingsboard ifølge Xilinx understøtter denne, og den samtidig understøtter programmering i C.



Figur 4.4: Oversigt over Case: LOG. Diagrammet viser med grønt de dele der udvikles igennem casen – i anførselstegn ved hver af disse, står navnet der anvendes om de afsnit, som omhandler de givne moduler.

5 Kravspecifikation

5.1 LOG

Dette afsnit omhandlende kravspecifikationen for loggen, vil bl.a. indeholde en generel beskrivelse, for at give et overblik over det valgte case-modul. Herudover vil beskrivelsen indeholde funktionelle og specifikke krav, som giver anledning til udviklingen af loggen.

Usecase-diagram for LOG

Herunder er der udarbejdet et usecase-diagram over loggen (5.1), for at give et overblik over loggens funktioner. En mere uddybende og specifik beskrivelse af loggens funktioner, vil være beskrevet under funktionelle krav (5.1.1).

Beskrivelse af LOG

LOG-modulet har til opgave, at håndtere og lagre data fra delmoduler og herefter sende den håndterede data ned til jorden udfra en forespørgsel, som modtages fra jorden. Loggen står kun for håndtering og lagring af data og det vil derfor være op til udviklerne, af de forskellige delmoduler, hvilke og hvor meget data de ønsker at lagre. Dataen fra delmodulerne indeholder statusoplysninger og disse kan eksempelvis være oplysninger omkring batteriniveau eller temperaturmålinger.

5.1.1 Funktionelle krav

I dette afsnit vil der være en beskrivelse af de funktionelle krav til loggen. De funktionelle krav, beskriver de funktioner, som loggen skal have og her er der søgt inspiration, fra det tidligere AAUSAT3 projekt.[2]

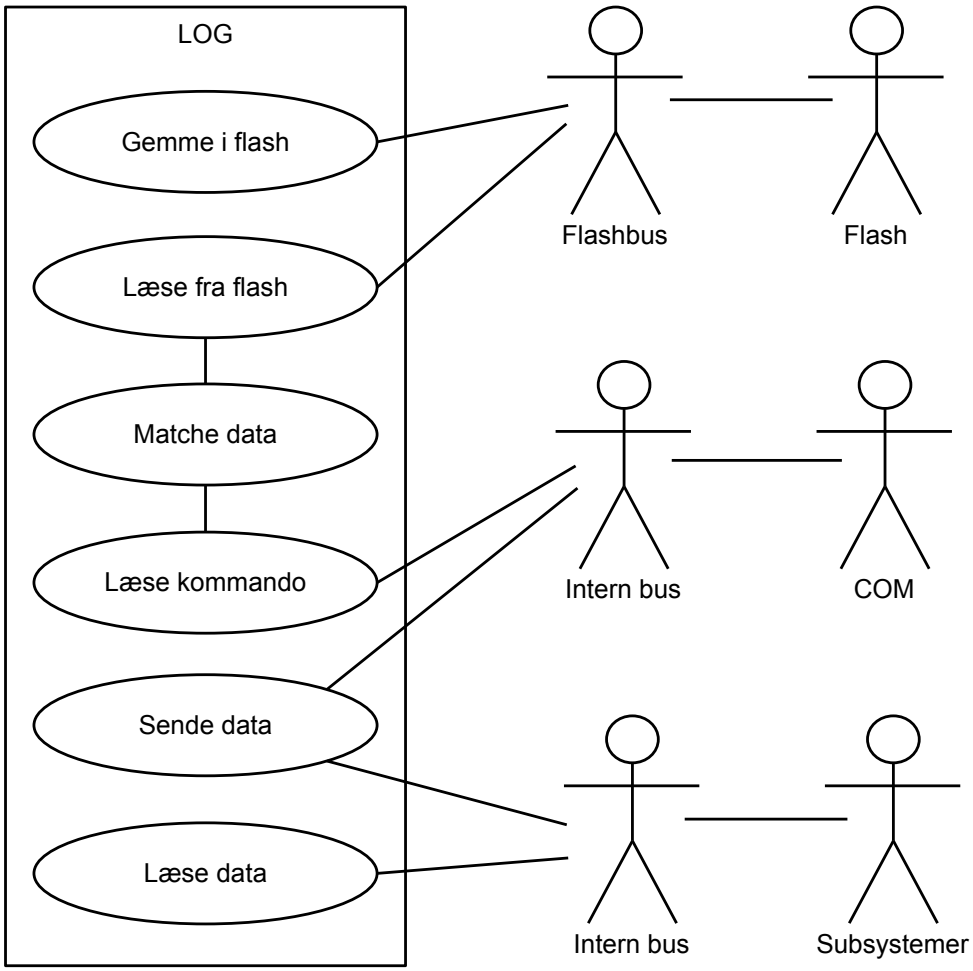
De funktionelle krav til loggen er listet herunder:

- Mulighed for at gemme data fra alle delmoduler
- Modtage kommando besked fra jorden.
- Mulighed for at sortere den lagrede data, ud fra kommando beskeden fra jorden.
- Sende sorteret data til jorden.
- Alle datapakker, skal have samme størrelse.

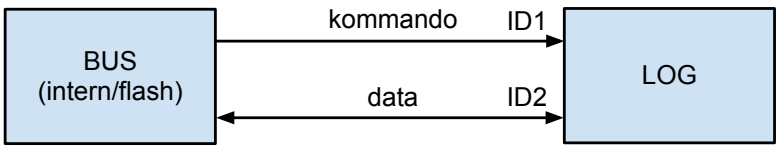
5.1.2 Specifikke krav for LOG

Herunder vil der være en beskrivelse af de specifikke krav til loggen.

Krav 1 Alle data pakker, skal have en størrelse på 16 B



Figur 5.1: Usecase diagram for loggen.



Figur 5.2: Kontekstdiagram for loggen.

Krav 2 Gemme alle logbeskederne fra delmoduler, med en Unix tid i størrelsen 4 B

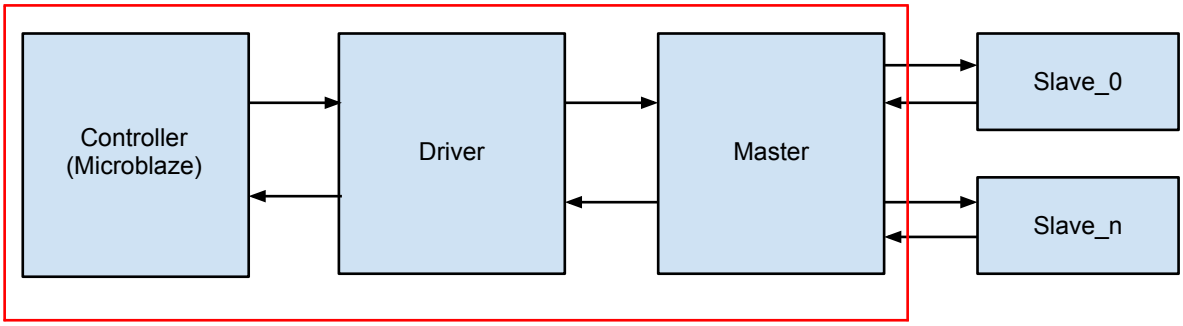
Krav 3 Plads til 200 forskellige ID numre, idet det antages at hver delmodul, kan have flere ID numre.

Krav 4 Sende dele eller hele loggen til jorden, alt efter forespørgslen i kommandobeskeden.

Krav 5 Tildele numre i numerisk rækkefølge til logbeskederne, inden de sendes til jorden

5.2 SPI

Som det fremgår vil SPI som helhed blive delt op i tre under moduler. Controlleren bestemmer hvad og hvornår data skal sendes over SPI bussen. Driveren skal fungere som en mellemmand mellem controller og SPI-master modulet, og skal sørge for at controlleren har et data register som der kan skrives og læses fra, samt sørge for at controlleren, kan overholde de timingkrav SPI sætter. Sidste delmodul er selve SPI-master modulet, dette sørger for, at få paralleltdata fra driveren omsat til serielt SPI data, så den overholder standarden. Herunder er udarbejdet et diagram over SPI modulet (5.3), for at give et overblik over dets funktioner. En mere uddybende og spicifik beskrivelse af SPI modulets funktioner, vil være beskrevet under funktionelle krav (5.2.1).



Figur 5.3: Overblik over SPI-master. Omkranset af den røde linie ses de tre moduler som tilsammen udgør SPI-master. Bemærk at alle kommandoer fra controller går gennem driver-modulet. Slaverne planlægges det at implementere eksternt i forhold til FPGA-chippen.

5.2.1 Funktionelle krav

Herunder vil der være en beskrivelse af de funktionelle krav for SPI-master modulet.

- Tilkobling af flere slaver.
- Polaritet skal kunne konfigureres
- Konfigurerbar fase.
- Modificerbar overførselshastigheden.
- Continuous mode skal kunne anvendes ved overførsel af store pakker.
- Der skal kunne sendes og modtages én pakke af gangen.

5.2.2 Specifikke krav

Herunder vil der være en beskrivelse af de specifikke krav for SPI-master modulet.

Krav 6 Modulet skal have 8 bit data bredde.

Krav 7 Det skal være muligt, at dividere clocken ned med 0,2,4 og 8.

Krav 8 Forsyningsspænding skal være på 3,3 V.

Krav 9 Skal minimum kunne køre med en clockfrekvens på 4 Mhz, uden at clocken er divideret.

Specifikke krav SPI-software

SPI-biblioteket skal kunne:

Krav 10 Sende en pakke på 8 bit af gangen.

Krav 11 Sætte clock-divideren, til de prædefinerede værdier fra krav 7.

Krav 12 Sætte modulet i continuous mode til at sende større pakker.

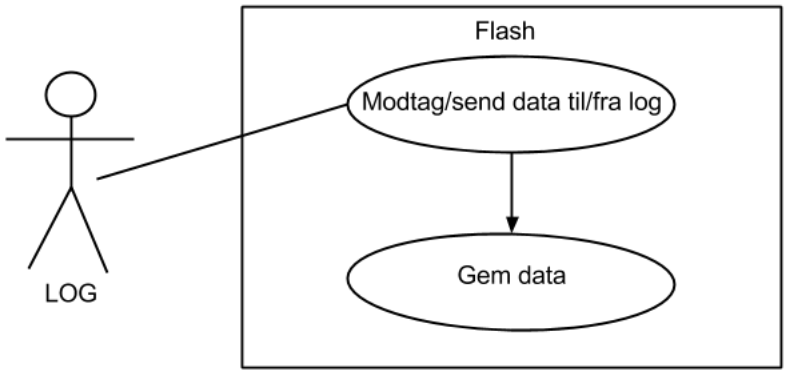
Krav 13 Indeholde en samlet funktion til at sende og modtage data over SPI.

5.3 Flash

Dette afsnit omhandlende kravspecifikationen for flashmodulet, vil bl.a. indeholde en generel beskrivelse, for at give overblik over den valgte IC. Udover en generel beskrivelse af flashmodulet, vil der ligeledes blive opstillet generelle krav og specifikke krav, som giver anledning til udvikling af flashmodulet.

Usecase-diagram for flash

Herunder er der udarbejdet et usecase-diagram for flashmodulet, som ses på Figur 5.4, for at give et overblik over flashmodulets funktioner. Under funktionelle krav (5.3.1) vil der være en mere uddybende og specifik forklaring af flashmodulets funktioner.



Figur 5.4: Usecase diagram for flashmodulet

Beksrivelse af flash

Flashmodulet har til opgave, at lagre de data, der modtages af logmodulet. Ud over at lagre de data der modtages fra logmodulet, skal flashmodulet også kunne sende tidligere gemt data tilbage til logmodulet, samt være mulighed for, på anmodning fra loggen, at slette gemt data fra hukommelsen.

5.3.1 Funktionelle krav

Herunder vil der være en beskrivelse af de funktionelle krav for flashmodulet.

- Mulighed for kommunikation via SPI.

- Skal være stor nok til at gemme logbeskeder fra alle delmoduler i 14 dage, uden at gammelt data slettes.
- Skal have mulighed for at slette alle data fra flashen.
- Skal kunne sende og modtage data til og fra LOG modulet.
- Skal have mulighed for at læse alt data fra den adresse som sidst er skrevet til flashen.
- Skal have mulighed for at skubbe ny data på næste ledige adresse.

5.3.2 Specifikke krav

Herunder vil der være en beskrivelse af de specifikke krav til flashmodulet.

Krav 14 Lagerstørrelsen skal minimum være 16 MiB.

Krav 15 Skal minimum kunne skrive 16 B data af gangen.

Krav 16 Skal kunne læse minimum 8 bit data af gangen.

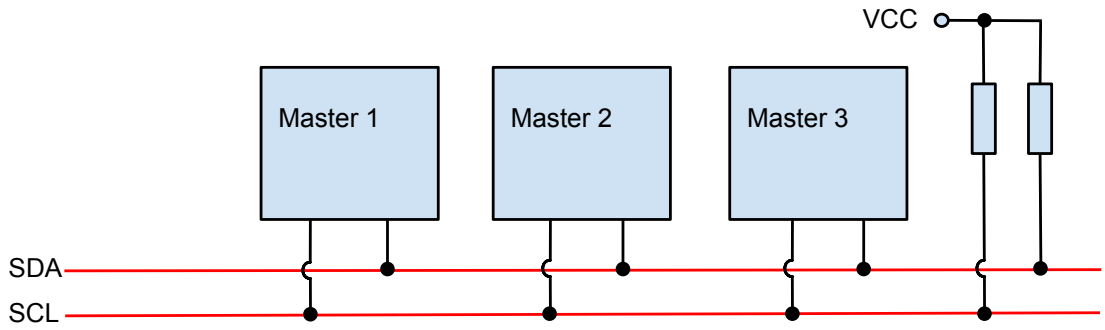
Krav 17 Skal kunne operere med en forsyningsspænding fra 3,3 V.

Krav 18 Skal minimum kunne overføre med en hastighed på 25 MHz.

5.4 I²C

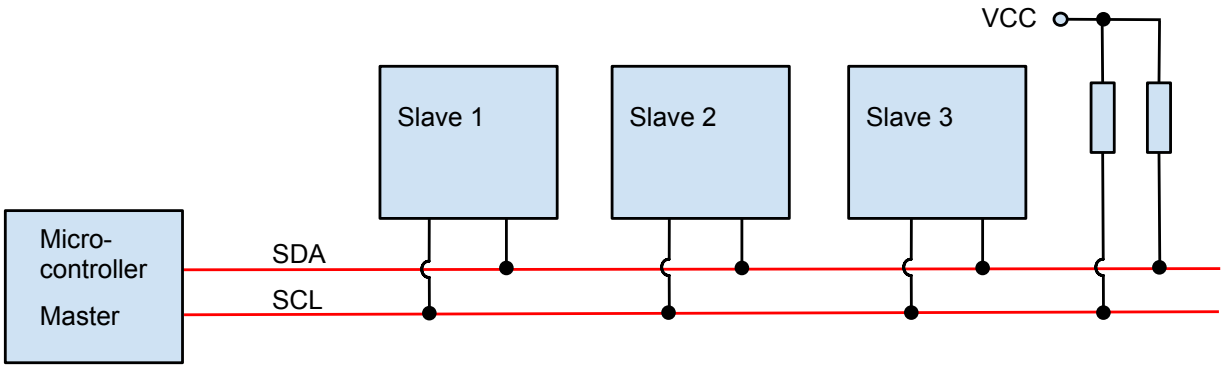
Kommunikationen består af to forbindelser, kaldet SDA (Seriel Data) og SCL (Seriel Clock). SCL og SDA forbindelserne bliver via pull-up modstande hævet til forsyningsspændingen, hvilket betyder at SCL og SDA som udgangspunkt er høje. Forbindelserne skal derved trækkes ned til stel for at ændre spændingsniveau, hvilket betyder at der også anvendes en stel forbindelse.[6]

Som det fremgår af Figur 5.6, har alle enheder forbindelse til I²C-bussen, hvilket betyder at alle enhederne kan kommunikerer direkte med hinanden. I de fleste tilfælde anvendes I²C kommunikation, som vist på Figur 5.6, hvor der kun er en master (i de fleste tilfælde en mikrocontroller) og flere slave enheder. I dette projekt anvendes der flere masterenheder, hvilket betyder, at alle enhederne kan opføre sig som slave- eller masterenheder.[3]



Figur 5.5: Blokdiagram I²C-master

Som beskrevet anvendes I²C-bussen i høj grad i næsten alle indlejrede systemer. FPGA'en er i sig selv et indlejret system, hvor der skal være mulighed for kommunikation mellem flere MicroBlazes. I²C-bussen er derfor ideel i dette projekt, da alle MicroBlaze enhederne, skal have mulighed for at tilgå hinanden. De funktionelle- og specifikke krav til I²C-bussen tager derfor udgangspunkt i den standardiserede I²C-bus designet af Philips[10], for at udnytte kompatibiliteten ved denne kommunikationsform.



Figur 5.6: Blokdiagram I²C-slave

5.4.1 Funktionelle krav for I²C

De funktionelle krav til I²C-bussen, beskriver de overordnede funktioner af bussen og hvilke muligheder de forskellige enheder skal have.

- Alle enheder skal have mulighed, for at skrive eller læse fra alle enheder forbundet til I²C-bussen.
- Der skal ikke være mulighed, for at to eller flere master enheder, kan eje bussen samtidig.
- Slaveenheder skal ikke have mulighed for at kommunikere på bussen, før de tilgås af masteren.

5.4.2 Specifikke krav for I²C

Herunder opstilles de specifikke krav til den interne kommunikation (I²C). Kravene tager udgangspunkt i den standardiserede I²C-bus [10], som beskrevet under den generelle beskrivelse af I²C'en 5.4.

Krav 19 Bussen skal bestå af to forbindelser, en data forbindelse (SDA) og en clock forbindelse (SCL).

Krav 20 100 kHz kommunikationshastighed, med en tolerance på $\pm 10\,\%$.

Krav 21 Mulighed for flere forskellige adresser per enhed, alt efter hvilken data det ønskes at tilgå. Adresserne er 7 bit.

Krav 22 Hver byte der sendes ud på bussen, skal være 8 bit lang, efterfulgt af et acknowledge bit.

Krav 23 Ingen begrænsning på mængden af databytes pr. overførsel.

6 Testspecifikation

6.1 LOG

Herunder vil der være en testspecifikation, som beskriver hvordan de funktionelle og specifikke krav testes. Kravene testes på samme måde, både for koden på PC'en og ved implementering på en MicroBlaze.

6.1.1 Test af funktionelle krav

De funktionelle krav opstillet i Afsnit 5.1.1, testes ved brug af en blackbox-test. Denne metode vælges da de funktionelle krav afhænger af kommandobeskeden fra jorden, hvilket gør det muligt, at teste kravene i samme test. Ved blackbox-testen, kigges der kun på input, forventet output og output og på den måde, kan det verificeres, om de funktionelle krav er overholdt. Blackbox testen, er illustreret på Figur 6.1.

Figur 6.1: Blackbox diagram

En mere specifik beskrivelse af denne testmetode, vil være beskrevet i testafsnittet, Afsnit 8.1.1

6.1.2 Test af specifikke krav

Test af Krav 1 – Pakke størrelse Dette krav testes ikke, da pakkerne designes udfra Krav 1, så de er på 16 B.

Test af krav 2 – Tilføjelse af Unix-tid Kravet testes ved at kigge på dataene, der skrives til flash-hukommelsen. Dette gøres ved at printe dataene på skærmen.

Test af krav 3 – Plads til ID Da pakkerne designes efter kravene, testes der ikke for dette krav.

Test af krav 4 – Forspørgsel af data Testes ved at sende forskellige kommandobeskeder til loggen og tjekke om returdataene matcher. De forskellige kommandobeskeder, kan ses under testafsnittet, Afsnit 8.1.1.

Test af krav 5 – Counter Testes ved at kigge på retur dataene til jorden. Dette gøres ved at printe dataene på skærmen.

6.2 Testspecifikation til SPI

I dette afsnit, vil testspecifikationen til SPI blive beskrevet. Denne beskriver hvordan de funktionelle og specifikke krav testes. Kravene testes på samme måde, både for koden på PC'en og ved implementering på en MicroBlaze.

6.2.1 Test af funktionelle krav

De funktionelle krav opstillet i Afsnit 5.2.1, testes udfra en blackbox-test. Der testes funktioner defineret i softwaren, og hvordan outputtet stemmer overens med et forventet resultat. Denne type test er illustreret på Figur 6.1. Under hvert krav, vil der være en specifik beskrivelse af testmetoden.

6.2.2 Test af specifikke krav

Test af krav 6 – Pakkestørrelse Denne variabel er sat i VHDL-koden. Dette kan testes ved at se størrelsen på pakkerne på en Logic Analyzer eller på et oscilloskop.

Test af krav 7 – Clock divider Denne variabel er sat i VHDL-koden og kan varieres fra controlleren. Signalet måles på et oscilloskop.

Test af krav 8 – Spændinger Dette krav er afhængigt af FPGA, netop Spartan-3 kører med 3,3 V og derfor er dette krav opfyldt.

Test af krav 9 – Minimum clock Dette krav er afhængigt af FPGA, og hvilken clockhastighed den kører med. Det valgte udviklingsboard kører med en Spartan-3 på 50 Mhz og derfor kører SPI maks med 50 Mhz.

6.3 Flash

I dette afsnit vil der være en testspecifikation der beskriver hvordan de funktionelle krav og de specifikke krav testes.

6.3.1 Test af funktionelle krav

De funktionelle krav opstillet i Afsnit 5.3.1, testes ved brug af en blackbox-test. Denne metode vælges da de funktionelle krav afhænger af kommandobeskeden fra loggen og flashmodulet, hvilket gør det ideelt, at teste kravene i samme test. Ved blackbox testen, kigges der kun på input, forventet output og output og på den måde, kan det verificeres, at de funktionelle krav er overholdt. Blackbox testen, er illustreret på Figur 6.1.

En mere specifik beskrivelse af denne testmetode, vil være beskrevet i Afsnit 8.3.1.

6.3.2 Test af specifikke krav

Test af krav 14 – Hukommelsesstørrelse Dette krav testes ikke, da det er et valg der bliver taget ud fra kravene.

Test af krav 15 – Skrive til flash Der bliver skrevet 16 B til flashen fra et Arduino-bibliotek, hvor der via en serial monitor bliver aflæst om det korrekte bliver skrevet til flashen.

Test af krav 16 – Læse fra flash Der bliver aflæst 8 bit fra flashen og skrevet på en serial monitor.

Test af krav 17 – Forsyningsspænding til flashmodulet Dette krav testes ikke, da modulet er valgt ud fra kravet til forsyningsspændingen.

Test af krav 18 – Kommunikationshastighed Dette krav testes ikke, da modulet er valgt ud fra kravet til overførselshastighed.

6.4 I²C

I dette afsnit, vil der være en beskrivelse af testspecifikationen til I²C-biblioteket. Testspecifikationen vil beskrive hvordan kravene testes, og derudover vil der være en beskrivelse af hvordan selve koden testes.

I²C-bussen skal stå håndtere den interne kommunikation i FPGA'en og det er derfor vigtigt, at lave en dybdegående test, hvor testen deles op i tre dele. Den første del består af en unit test, hvor hver funktion i koden testes individuelt. I anden del vil der være en global test, hvor alle funktionerne samt kravene testes samlet, og i tredje test testes I²C-modulet med en I²C-slave.

Testen udføres på en ATmega32 mikrocontroller, med et tilhørende testboard med otte knapper og otte dioder. Som I²C-slave benyttes en Arduino Mega 1280.

6.4.1 Unit test

Unit testen anvendes på alle de funktioner, der kan testes ved forskellige inputs. Funktionerne der testes, testes så individuelt som muligt, ved at undgå at kalde andre funktioner. Det er i dette tilfælde ikke muligt, at teste alle funktionerne uden at kalde andre funktioner, men hvis muligheden er der, så testes der individuelt.

Ved brug af denne test, testes alle funktionerne individuelt og det er derfor muligt at opdage fejl, som ikke vil være synlige i en global test af hele systemet. I forhold til en global test, hvor der skal for at lave en dybdegående test, anvendes utallige test kombinationer, er det er med en unit test, forholdsvis nemt og overskueligt at teste hver funktion for sig.

Unit tests gør at koden i sidste ende er mere robust og man ikke er så udsat for fejl senere i udviklingen.

Unit testene udføres med et valg af inputs, forventede outputs og output-resultater, som verificeres og måles ved brug af et Rigol DS1102E oscilloskop. Et overblik over testopstillingen, kan ses på Figur 6.2.

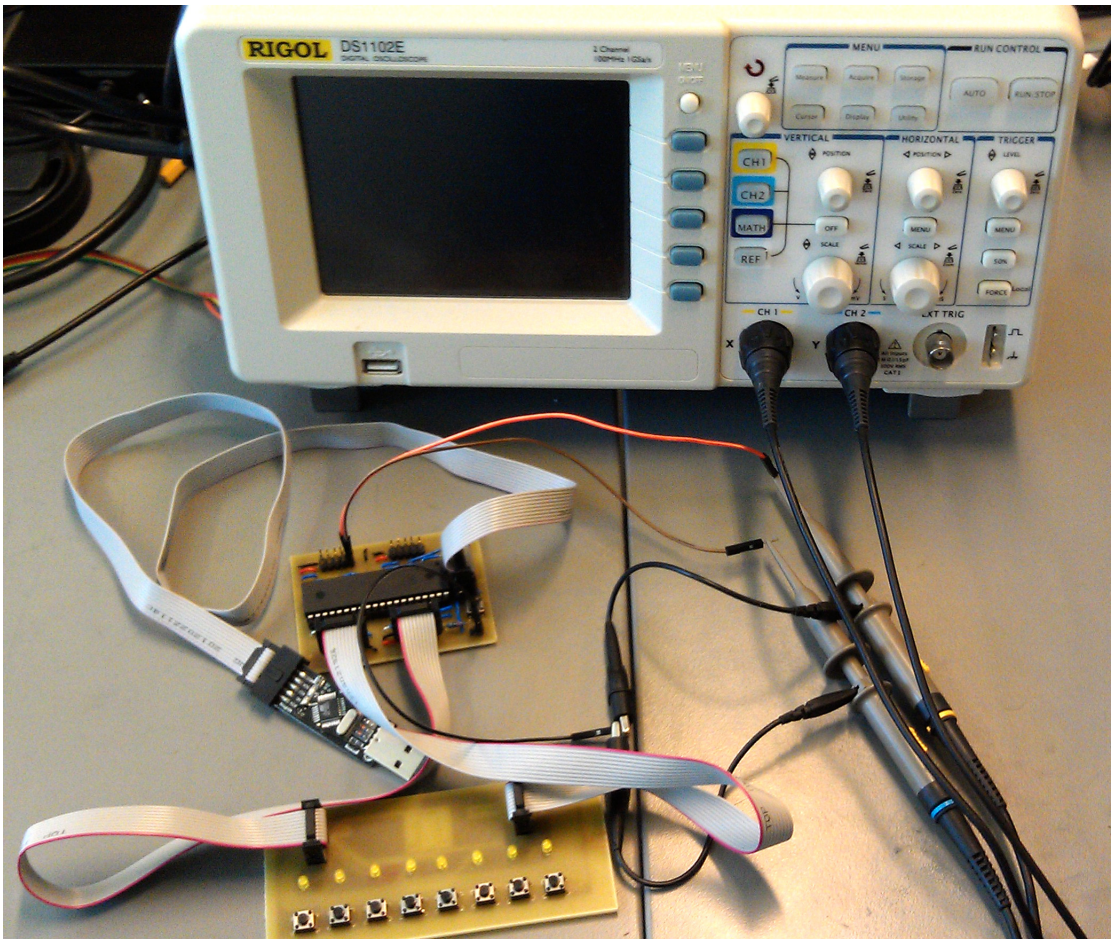
Input, forventet output og output, kan være forskelligt, alt efter hvilken funktion der testes og vil derfor først være beskrevet under testafsnittet 8.6. [11]

6.4.2 Global test

Som navnet antyder, anvendes den globale test, til at teste de globale funktioner. Testen fokuserer på at teste sammenspillet mellem alle funktionerne. Herudover testes de globale funktioner også, for at verificere at I²C-bussen overholder de funktionelle- og specifikke krav. Da der også i den globale test anvendes forskellige inputs, er disse først beskrevet under testafsnittet, Afsnit 8.6, hvor der også vil være en beskrivelse af de forventede outputs og testresultaterne.

Test af specifikke krav

Test af krav 19 – Busforbindelser I²C-bussen designes efter den standardiserede I²C-bus fra Philips[10], bussen udvikles derfor, så der kun er to busforbindelser, som beskrevet i kravet hertil.



Figur 6.2: I^{2}C testopstilling

Test af krav 20 – Overførselshastighed Testes ved at kigge på periodetiden under transaktion. Dette gøres ved at måle på SDA- og SCL linjen ved brug af et oscilloskop.

Test af krav 21 – Adressering Dette krav testes ved at tilegne flere forskellige adresser til en enhed og herefter tjekke om alle adresserne kan tilgås fra en anden enhed.

Test af krav 22 – Bytestørrelse Kravet testes ved at kigge på SDA- og SCL linjen ved en igangværende overførsel og verificere længden af byten og om der modtages eller sendes et acknowledge bit.

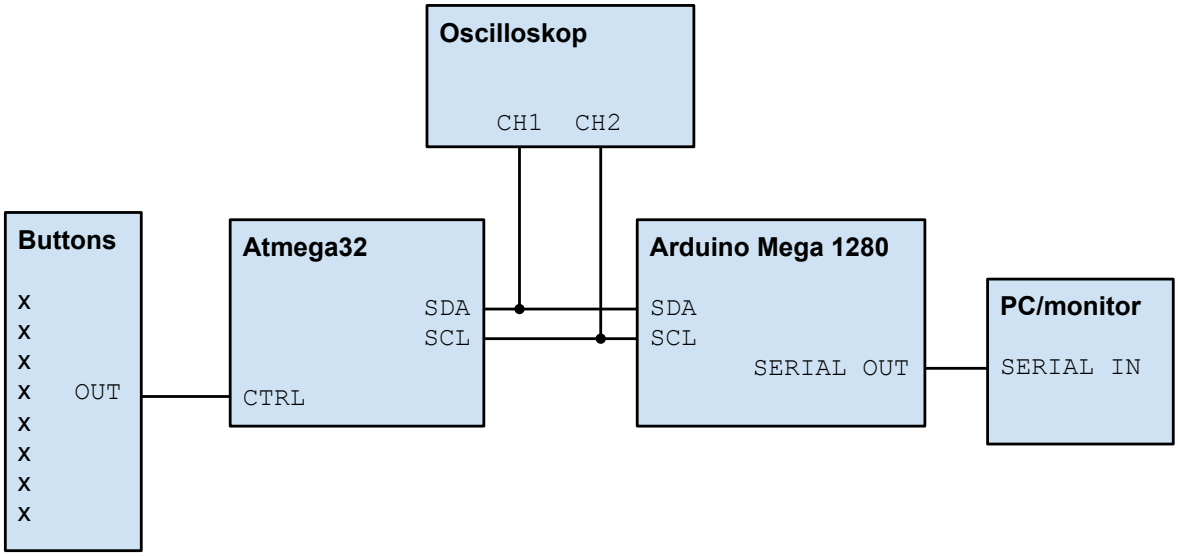
Test af krav 23 – Datastørrelse Kravet testes ikke, da I^{2}C-bussen designes, så mængden af datapakker ikke er begrænset af andet end hukommelsen.

6.4.3 Test med Arduino som slave.

For at teste, om I^{2}C fungerer med den standardiserede I^{2}C-bus, udføres en test, hvor I^{2}C-master-modulet stadig er på en ATmega32, og har forbindelse til den indbyggede I^{2}C-bus på en Arduino Mega 1280.

Arduinoen sættes op som slave, med en bestemt adresse og sættes til at sende al den data den modtager over I^{2}C serielt til en PC, hvor dataene kan monitoreres. Samtidig overvåges I^{2}C-linjerne på et oscilloskop.

Testopstillingen ses på Figur 6.3.



Figur 6.3: Test med Arduino som slave.

Der foretages to tests:

1. Arduino modtager data fra master.
2. Arduino sender data til master.

som hver deles op i nogle mindre tests.

Arduino modtager data fra master

Følgende vil blive testet, for at sikre, at Arduinoen kan *modtage* data fra masteren (ATmega32), der benytter det udviklede I^2C-bibliotek:

- Arduinoen godkender sin egen adresse, og kun den.
- Arduinoen trækker et acknowledge efter at have modtaget sin egen adresse.
- Når Arduinoen modtager én eller flere databytes, printes disse på computerskærmen.

Testene foretages over tre omgange, som følger.

Matchende adresse og ACK

1. Master: Send matchende adresse + write – ingen data sendes med.
2. Tjek ACK på skop.
3. Arduino: Print, hvis adresse matcher.
4. Master: Send andre adresser + write, og observer om disse kommer igennem.

Enkelt databyte

1. Master: Send matchende adresse + write + data (1 B).
2. Tjek ACK på skop.
3. Arduino: Print hvis adresse matcher.
4. Arduino: Print data modtaget.

Flere databytes

1. Master: Send matchende adresse + write + data (2 B).
2. Tjek ACK på skop.
3. Arduino: Print hvis adresse matcher.
4. Arduino: Print data modtaget.
5. Master: Der sendes flere bytes, og foretages samme tests.

Arduino sender data til master

Følgende vil blive testet, for at sikre, at Arduinoen kan *sende* data fra masteren (ATmega32), der benytter det udviklede I^2C-bibliotek.

- Arduinoen trækker et ACK, når dens egen adresse modtages.
- Arduinoen sender én eller flere databytes, når masteren beder om det.

Testene foretages over to omgange, som følger.

Send 1 B til master

1. Master: Send matchende adresse + read.
2. Master: Clock 1 B ind fra Arduino.
3. Tjek ACK på skop.
4. Tjek databyte på skop.

Send 2 B til master

1. Master: Send matchende addresse.
2. Master: Clock 2 B ind fra Arduino.
3. Tjek ACK på skop.
4. Tjek databytes på skop.
5. Udfør samme test med flere bytes.

Alle ovenstående tests med Arduinoen dokumenteres senere i Afsnit 8.4.1.

7 Udvikling

7.1 LOG

Herunder beskrives udviklingen af hovedkoden til det LOG-system, der implementeres på FPGA'en, som led i udviklingen af systemet på FPGA'en.

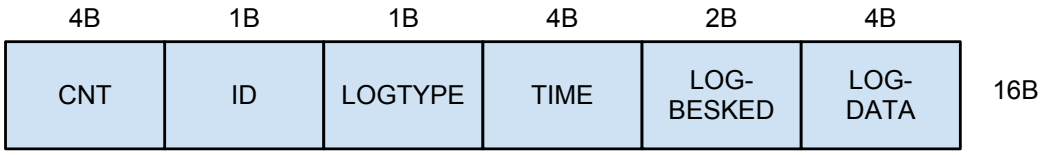
Hovedkoden har grænseflader mod den interne bus imellem mikrocontrollerne og mod flash. Gennem den interne bus modtager den beskeder fra andre moduler, der vil gemme noget i flashen. Når der skal sendes log-data til jorden, foregår det også via den interne bus, ved at lade jordkommunikationen foregå gennem et andet modul på bussen (COM). Flashen bruges til at læse og skrive fra.

Fra jorden kan der sendes en kommando til LOG-modulet, der beskriver, hvad der skal sendes ned.

Det er ovenstående det der her vil beskrives i detaljer.

7.1.1 Databeskrivelse

Dataene der lagres i flashen, og sendes til jorden, gemmes i pakker på 16 B, som det ses på Figur 7.1. De enkelte blokke dækker over, og lagrer data således:



Figur 7.1: LOG data-pakke.

CNT En tæller, der automatisk tæller op, når data sendes ned. Denne værdi er tilfældig i flashen, men bliver tildelt når pakker sendes ned. Denne bruges til at se, om nogle pakker er gået tabt på vej ned.

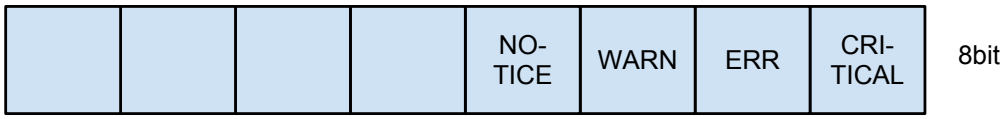
ID Alle moduler på den interne bus har ét eller flere ID'er. I denne blok står ID'et på det modul, der har skrevet log-beskeden.

LOGTYPE Der findes fire typer log, som kan stå her: CRITICAL, ERROR, WARNING og NOTICE. LOGTYPE-blokken bruges som et register, der har en bit til at repræsentere hver logtype. Dette register ses på Figur 7.2.

TIME Tid for log i "Unix time". Skrives på log-tidspunktet. I datapakken fra delmodulerne, er disse bytes tilfældige.

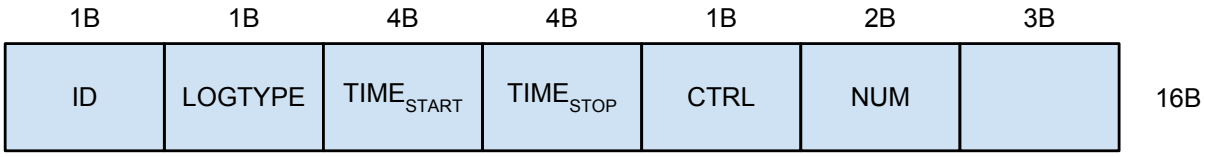
LOGBESKED ID-afhængige fejlnumre, der repræsenterer fejl der er givet ved de forskellige moduler. Et eksempel kunne være overophedning, der knytter sig til EPS.

LOGDATA Data der knytter sig til LOGBESKED'en. Ved overophedning kunne LOGDATA eksempelvis være temperaturen i °C.



Figur 7.2: LOGTYPE-register.

7.1.2 Kommandobeskrivelse



Figur 7.3: LOG kommando.

Kommandoen sendes til loggen, og bruges til at vælge, hvilke pakker fra loggen der skal sendes ned. For at have konsistente pakkestørrelser vælges det også at bruge 16 B på kommandopakken, der bestemmer hvilke data der skal sendes til jorden. Kommandoen ses på Figur 7.3, og består af følgende. Navnene stemmer overens med de, brugt i databeskrivelse.

ID Hvis ID sættes til et ID $\neq$ `0x00`, sendes kun dette ID's pakker ned. Hvis ID = `0x00`, sendes alle ID'er ned.

LOGTYPE Der sættes et register svarende til det på Figur 7.2. Et 1-tal betyder at logtypen på denne plads sendes ned – et 0-tal og denne logtype sendes ikke ned. Sættes registeret f.eks. til `0x04` , sendes kun warnings ned – `0x06` for errors og warnings osv.

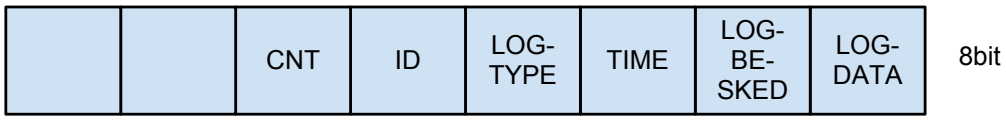
TIME-START Her er starttidspunkt for data der skal sendes ned, angivet i Unix-tid.

TIME-STOP Her er stoptidspunkt for data der skal sendes ned, angivet i Unix-tid.

CTRL Dette kontrolregister (se Figur 7.4) sætter hvilke dele af datapakken, Figur 7.1, der sendes ned. Sættes et 1-tal ud for eksempeltvis ID, og TIME, men nuller i resten, sendes kun ID og TIME ned.

NUM Sættes dette til NUM = `0x00`, sendes alt der matcher ned. Ellers sendes kun de første NUM datapakker der matcher ned – er NUM = `0x07`, sendes kun de seneste 7 datapakker ned.

<Tomme bytes> Disse er med for at give 16 B i kommandopakken.



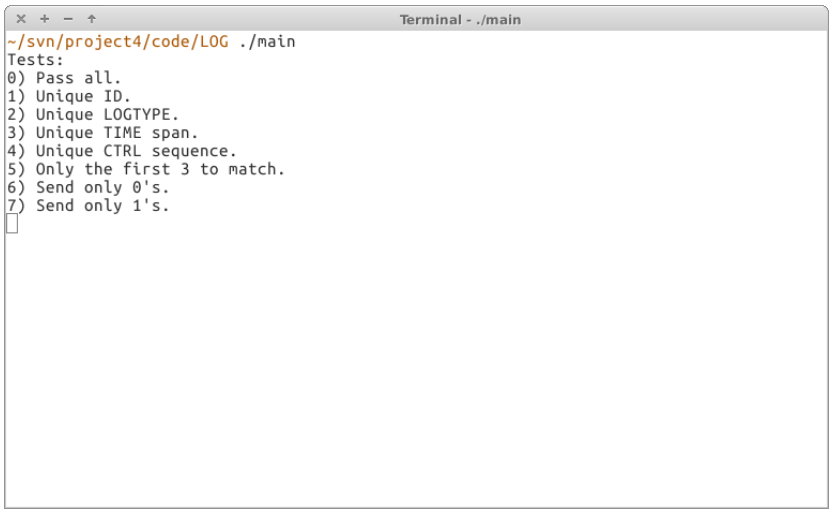
Figur 7.4: Kontrolregister i kommando.

7.1.3 Kode

Udviklingen af koden foregår i 3 omgange, da den interne bus og SPI-bussen udvikles parallelt med loggens hovedprogram. Faserne er:

1. Log-kode udvikles til at virke på en PC. Her benyttes prædefinerede kommandoer og data, som er repræsentative for at teste funktionaliteten af logsystemet. Der benyttes standard C-biblioteker til tiden. Flash- og bus-outputs printes ud på skærmen.
2. Log-kode implementeres på MicroBlaze. Der benyttes stadig faste værdier for data og kommandoer, og da MicroBlazen i sig selv ikke kan klokken, benyttes en fast tid. In- og outputs foregår via kontakter og led'er/7-segments-display/RS-232.
3. Log implementeres med flash og intern bus. Dette går ind under implementering, som følger senere.

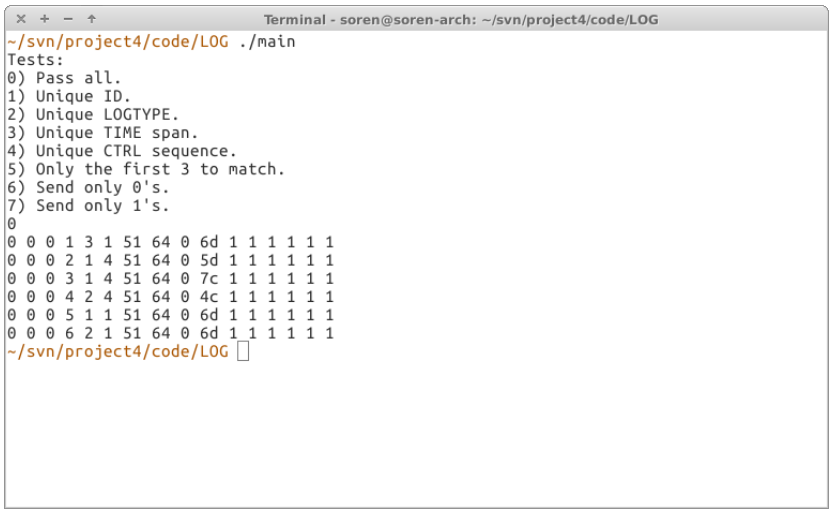
Log-kode udviklet til PC



Figur 7.5: Brugergrænseflade for logprogram på PC - her vises opstartsskærmen.

Logkoden udviklet til PC, er lavet for at teste, at data bliver sorteret korrekt, ved en given kommando.

Brugergrænsefladen ser ud, som på Figur 7.5, under opstart. Den giver mulighed for at vælge imellem forskellige test-kommandoer, der sorterer data fra "flashen" (test-data). Testdata og testkommandoer er defineret i to biblioteker, `testdata_main.h` og `testdata_log.h`, og beskrives under test i Afsnit 8.1.1. Figur 7.6 viser den kommando der printer hele "flashen".



Figur 7.6: Brugergrænseflade for logprogram på PC - her vises funktionen der printer hele loggen.

```c
/**
 * LOG main program for AAUSAT4.
 * @author G416
 * @date 2013-04-09
 */
#include <stdio.h>
#include <stdlib.h>
#include "log.h"
#include "testdata_main.h"

/* #define FLASH_SIZE 0x11111111 */
#define FLASH_SIZE 6                    /* TODO: Set to actual flash size */

/* Main */
int main(void)
{
    unsigned char *cmd  = (unsigned char*)calloc(COMMAND_LGT, sizeof(unsigned char));
    unsigned char *data = (unsigned char*)calloc(DATA_LGT,   sizeof(unsigned char));
    unsigned int cnt = 0;
    unsigned int flashCnt = 0;
    unsigned int num;
    unsigned char i;

    /* DETERMINE ID */
    switch (busID) {
    case ID_LOG_COMMAND:
        testGetCommand(cmd);
        num = cmd[11] << 8 | cmd[12];       /* Max number of packages to send down */

        while (flashCnt < FLASH_SIZE &&     /* Loop until max number of packages are sent, and flash not finished */
                (cnt < num || num == 0x00)) {
            readFromFlash(data, flashCnt++);
            sendToEarth(cmd, data, &cnt);
        }
        break;
    case ID_LOG_TOFLASH:
        for (i = 0; i < 6; i++) {           /* Write six test-packages to flash */
            writeToFlash(busData_toFlash[i]);
        }
        break;
    default:
```

```
42          break;
43      }
44
45      return 0;
46  }
```

Listing 7.1: Logkode på PC - her vises main.c

Mainprogrammet har til opgave at vælge imellem at skrive til flashen, og læse fra flashen/modtage en kommando. I Listing 7.1 ses koden for mainprogrammet. En switch-statement benyttes til at bestemme hvad der skal udføres, bestemt ud fra det det ID, dataene er kommet ind med fra bussen (`busID`). Kode findes også på cd under:

`code/LOG/{main.c, log.h, log.c}`

De forskellige bus-ID'er, der benyttes rundt omkring i AAUSAT4, er defineret i filen `bus_id.h`*. Dette skal bemærkes når der siden skal kobles flere moduler på.*

ID_LOG_COMMAND: Når data kommer ind med dette ID, er det en kommando fra jorden der kommer ind. Først gemmes NUM, som er det højeste antal pakker der skal sendes ned, og herefter sendes der pakker ned – enten indtil NUM er nået, eller til der ikke er flere pakker der matcher. Bemærk, at er NUM = `0x00`, er der ingen begrænsning på, hvor mange pakker der må sendes ned.

ID_LOG_TOFLASH: Når data kommer ind med dette ID, skal hele inputpakken skrives ind i flashen. I testprogrammet, Listing 7.1, skrives 6 testpakker fra `testdata_main.h` ind i flashen/ud på skærmen.

Selve logbiblioteket findes i `log.h` og `log.c`. Funktionerne ses i Listing 7.2

```
void writeToFlash(unsigned char *data);
void testGetCommand(unsigned char *command);
void sendToEarth(unsigned char *command, unsigned char *data, unsigned int *cnt);
char readFromFlash(unsigned char *data, unsigned int address);
int getRTC();
```

Listing 7.2: Funktioner i log-bibliotek

Den totale kode ses i Listing 7.3.

```
1   /**
2    * LOG function library for AAUSAT4
3    * @author G416
4    * @date 2013-04-09
5    */
6   #include <stdio.h>
7   #include "log.h"
8   #include <time.h>
9   #include "testdata_log.h"
10
11  /* Functions */
12  void writeToFlash(unsigned char *data)
13  {
14      /* TODO: Write to flash instead of screen */
15      char i;
16      int timeNow;
17
18      /* Replace time in data with data from RTC */
19      timeNow = getRTC();
20      data[6] = (timeNow & 0xff000000) >> 24;
21      data[7] = (timeNow & 0x00ff0000) >> 16;
22      data[8] = (timeNow & 0x0000ff00) >> 8;
23      data[9] = (timeNow & 0x000000ff) >> 0;
```

```
24
25      /* Debug print */
26      printf("FLASH WRITE: ");
27      for (i = 0; i < DATA_LGT; i++) {
28          printf("%x ", data[i]);
29      }
30      printf("\n");
31  }
32
33  void testGetCommand(unsigned char *command)
34  {
35      /* A command is incoming from earth. Store it in "command" */
36      /* TODO: Get actual command */
37      char i;
38
39      /* TODO: Nasty nasty temporary command number */
40      printf("Tests:\n0) Pass all.\n1) Unique ID.\n2) Unique LOGTYPE.\n3) Unique TIME span.\n4) Unique CTRL sequence.\n5) Only
            the first 3 to match.\n6) Send only 0's.\n7) Send only 1's.\n");
41      char tmpCommandNum;
42      scanf("%c", &tmpCommandNum);
43      tmpCommandNum -= '0';
44
45      for (i = 0; i < COMMAND_LGT; i++) {
46          if (tmpCommandNum < 8)
47              command[i] = busCommands[tmpCommandNum][i];
48          else
49              command[i] = busCommands[0][i];
50      }
51  }
52
53  void sendToEarth(unsigned char *command, unsigned char *data, unsigned int *cnt)
54  {
55      unsigned char dataToEarth[DATA_LGT];
56      int j;
57      int index = 0;
58
59      /* Clear dataToEarth array */
60      for (j = 0; j < DATA_LGT; j++) dataToEarth[j] = '\0';
61
62      /* Compare with command */
63      /* CNT */
64      if(command[10] & 0x20) {              /* CNT-bit set in CTRL register */
65          /* Make room for counter. Counter is only set
66           * if the rest of the checks below passes.
67           * Counter is incremented, and written just
68           * before package is sent to earth. */
69          dataToEarth[index++] = 0x00;       /* Room for counter */
70          dataToEarth[index++] = 0x00;
71          dataToEarth[index++] = 0x00;
72          dataToEarth[index++] = 0x00;
73      }
74      /* ID */
75      if(command[10] & 0x10) {              /* ID-bit set to CTRL register */
76          if(command[0] == 0x00) {            /* Send all down */
77              dataToEarth[index++] = data[4];    /* Send ID down */
78          } else if (command[0] == data[4]) {  /* ID in command == ID in package */
79              dataToEarth[index++] = data[4];    /* Send ID down */
80          } else {                            /* ID doesn't match - drop package */
81              return;
82          }
83      }
84      /* LOGTYPE */
85      if(command[10] & 0x08) {              /* LOGTYPE-bit set in CTRL register */
86          if(command[1] & data[5]) {          /* If {crit,err,war,not}-bit set in LOGTYPE-register */
87              dataToEarth[index++] = data[5];    /* Send LOGTYPE down */
```

```
88          } else {                            /* LOGTYPE doesn't match - drop package */
89              return;
90          }
91      }
92      /* TIME */
93      if(command[10] & 0x04) {                /* TIME-bit set in CTRL register */
94          unsigned int intStartTime;
95          unsigned int intStopTime;
96          unsigned int intDataTime;
97
98          /* Convert 4 chars to int */
99          intStartTime = command[2] << 24 | command[3] << 16 | command[4] << 8 | command[5];
100         intStopTime  = command[6] << 24 | command[7] << 16 | command[8] << 8 | command[9];
101         intDataTime  = data[6] << 24    | data[7] << 16    | data[8] << 8    | data[9];
102
103         if(intDataTime >= intStartTime       /* If TIME is in interval given by command */
104                 && intDataTime <= intStopTime) {
105             dataToEarth[index++] = data[6];    /* Send TIME down */
106             dataToEarth[index++] = data[7];
107             dataToEarth[index++] = data[8];
108             dataToEarth[index++] = data[9];
109         } else {                            /* TIME doesn't match - drop package */
110             return;
111         }
112     }
113     /* LOGBESKED */
114     if(command[10] & 0x02) {                /* LOGBESKED-bit set in CTRL register */
115         dataToEarth[index++] = data[10];     /* Send LOGBESKED down */
116         dataToEarth[index++] = data[11];
117     }
118     /* LOGDATA */
119     if(command[10] & 0x01) {                /* LOGBESKED-bit set in CTRL register */
120         dataToEarth[index++] = data[12];     /* Send LOGDATA down */
121         dataToEarth[index++] = data[13];
122         dataToEarth[index++] = data[14];
123         dataToEarth[index++] = data[15];
124     }
125     /* Write CNT to package */
126     if(command[10] & 0x20) {                /* CNT-bit set in CTRL register */
127         *cnt += 1;                            /* Count up CNT */
128
129         /* Split cnt into 4 chars */
130         dataToEarth[0]                        /* Send CNT down */
131             = (*cnt & 0xff000000) >> 24;
132         dataToEarth[1]
133             = (*cnt & 0x00ff0000) >> 16;
134         dataToEarth[2]
135             = (*cnt & 0x0000ff00) >> 8;
136         dataToEarth[3]
137             = (*cnt & 0x000000ff) >> 0;
138     }
139
140     /* Send to Earth */
141     /* TODO: Command to send to internal bus, goes here.
142      * To sendt to earth, send to COM-module, with it's
143      * BUSID.
144      * Include ability to send variable length packages
145      * to earth. */
146     printf("%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x\n",
147            dataToEarth[0]  , dataToEarth[1]  , dataToEarth[2]  , dataToEarth[3]  ,
148            dataToEarth[4]  , dataToEarth[5]  , dataToEarth[6]  , dataToEarth[7]  ,
149            dataToEarth[8]  , dataToEarth[9]  , dataToEarth[10] , dataToEarth[11] ,
150            dataToEarth[12] , dataToEarth[13] , dataToEarth[14] , dataToEarth[15]
151           );
152 }
```

```
153
154 char readFromFlash(unsigned char *data, unsigned int address)
155 {
156     /* TODO: Read from actual flash */
157     char i;
158     for (i = 0; i < DATA_LGT; i++) {
159         data[i] = busData[address][i];
160     }
161     return 1;
162 }
163
164 int getRTC()
165 {
166     /* TODO: Get time from RTC, instead of computer system time */
167     return (int)time(NULL);
168 }
```

Listing 7.3: Logkode på PC - her vises log.c

writeToFlash(unsigned char *data) Skriver tiden-lige-nu, hentet via `getRTC()`, ind i `*data`, og gemmer i flashen. I test-programmet betyder det, at `*data` bliver printet på skærmen som hexadecimaltal.

testGetCommand(unsigned char *command) Henter testkommando ud fra et tal brugeren indtaster. I virkeligheden er denne kommando lig med den data der kommer ind over bussen, så denne kommando er ikke nødvendig. Brugerinterfacet er defineret heri.

sendToEarth(unsigned char *command, unsigned char *data, unsigned int *cnt) Former `*data` til en streng, på baggrund af kommandoen `*command`, og sender denne til jorden/COM-modulet, hvis alle kommando-kriterier er opfyldt. Samtidig skrives der et tællernummer, `*cnt`, til pakken, så det fra jorden let kan ses, om alle data er kommet ned. Er de ikke det, mangler der et tal på tællerens plads.

Pakken formes alt efter hvordan CTRL-registeret i kommandoen er sat, og overskydende pladser i pakken, fyldes med nuller.

readFromFlash(unsigned char *data, unsigned int address) Kommando til at læse data på flash-adressen `address` ind i `*data`. Returnerer 1 hvis dette lykkes.

getRTC() Returnerer Unix-tid-lige-nu. Denne hentes fra et tidsmodul, der her simpelthen udgøres af `time(NULL)` fra `time.h`.

Log-kode udviklet til Microblaze

For at få koden til at fungere på Microblazen fremfor en PC, er der ændret nogle enkelte ting for at kunne teste programmet. Dette skyldes hovedsageligt, at brugergrænsefladen til Microblazen ikke indeholder skærm og tastatur.

Brugergrænsefladen er ændret således:

- Input via switche. Switchene angiver et binært tal fra 0–8 – altså bestemt ud af de første tre switche – og bestemmer hvilken test der udføres.
- Output via RS-232: Data sendes via RS-232 til en pc.

MicroBlaze-koden findes på CD under:

`code/LOG_mb_standalone/workspace/log/src/{main.c, log.c, log.h}`

Da ændringerne i koden, i forhold til koden på PC'en, er ganske få, vises i Listing 7.4 og Listing 7.5 kun en diff[1] af de tilsvarende filer på PC og MicroBlaze.

```
4c4
<  * @date 2013-04-09
---
>  * @date 2013-04-23
6,7d5
< #include <stdio.h>
< #include <stdlib.h>
9a8
> #include "platform.h"
17,18c16,17
<     unsigned char *cmd  = (unsigned char*)calloc(COMMAND_LGT, sizeof(unsigned char));
<     unsigned char *data = (unsigned char*)calloc(DATA_LGT,   sizeof(unsigned char));
---
>     unsigned char cmd[COMMAND_LGT];
>     unsigned char data[DATA_LGT];
22a22,24
>
>     /* Init */
>     init_platform();
```

Listing 7.4: Diff af main.c efter omskrivning til MicroBlaze.

```
4c4
<  * @date 2013-04-09
---
>  * @date 2013-04-23
8c8
< #include <time.h>
---
> #define DUMMY_TIME 1366733256
9a10,11
> #include "xparameters.h"
> #include "xiomodule.h"
15c17
<     char i;
---
>     unsigned char i;
26c28
<     printf("FLASH WRITE: ");
---
>     print("FLASH WRITE: ");
28c30
<         printf("%x ", data[i]);
---
>         putnum(data[i]);
30c32
<     printf("\n");
---
>     print("\n");
37c39
<     char i;
---
>     unsigned char i;
39,43c41,46
<     /* TODO: Temporary command number */
```

[1] `diff` er et program der læser to filer, og printer forskellen på de to i et givent format.

```
34  <     printf("Tests:\n0) Pass all.\n1) Unique ID.\n2) Unique LOGTYPE.\n3) Unique TIME span.\n4) Unique CTRL sequence.\n5) Only
          the first 3 to match.\n6) Send only 0's.\n7) Send only 1's.\n");
35  <     char tmpCommandNum;
36  <     scanf("%c", &tmpCommandNum);
37  <     tmpCommandNum -= '0';
38  ---
39  >     /* Choose command with switches. */
40  >     XIOModule gpi1;
41  >     int sw;
42  >     sw = XIOModule_Initialize(&gpi1, XPAR_IOMODULE_0_DEVICE_ID);
43  >     sw = XIOModule_Start(&gpi1);
44  >     sw = XIOModule_DiscreteRead(&gpi1, 1);
45  44a48
46  >     /* Set binary number on switches, to choose command. */
47  46,47c50,51
48  <         if (tmpCommandNum < 8)
49  <             command[i] = busCommands[tmpCommandNum][i];
50  ---
51  >         if (sw < 8)
52  >             command[i] = busCommands[sw][i];
53  50a55
54  >
55  56c61
56  <     int j;
57  ---
58  >     unsigned char j;
59  149,154c154,155
60  <     printf("%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x\n",
61  <            dataToEarth[0]  , dataToEarth[1]  , dataToEarth[2]  , dataToEarth[3]  ,
62  <            dataToEarth[4]  , dataToEarth[5]  , dataToEarth[6]  , dataToEarth[7]  ,
63  <            dataToEarth[8]  , dataToEarth[9]  , dataToEarth[10] , dataToEarth[11] ,
64  <            dataToEarth[12] , dataToEarth[13] , dataToEarth[14] , dataToEarth[15]
65  <           );
66  ---
67  >     for (j = 0; j < DATA_LGT; j++) putnum(dataToEarth[j]);
68  >     print("\n");
69  160c161
70  <     char i;
71  ---
72  >     unsigned char i;
73  169,170c170,171
74  <     /* TODO: Get time from RTC, instead of computer system time */
75  <     return (int)time(NULL);
76  ---
77  >     /* TODO: Get time from RTC, instead of DUMMY_TIME */
78  >     return DUMMY_TIME;
```

Listing 7.5: Diff af log.c efter omskrivning til MicroBlaze.

Det viser sig, at pladsen på MicroBlazen er ret begrænset, og at `calloc`-funktionen fylder en del. Derfor benyttes et fast-størrelse array i stedet, da der ikke er nogen grund til at bruge dynamisk allokering til dette formål (Listing 7.4).

Der bruges nogle få funktioner, der er MicroBlaze-specifikke. For at skrive til RS-232, benyttes MicroBlaze-biblioteket `stdio.h`. For at læse fra switchene, benyttes to biblioteker: `xparameters.h` og `xiomodule.h`. Funktionerne der benyttes til test er:

print(s) Sender strengen `s` til RS-232-porten.

putnum(n) Sender hex-værdien af integeren `n` til RS-232-porten.

XIOModule_Initialize(&gpiOrGpo, XPAR_IOMODULE_0_DEVICE_ID) Initialiserer et input eller output, `gpiOrGpo`, af typen `XIOModule`.

XIOModule_Start(&gpiOrGpo) Start input eller output, `&gpiOrPgo`.

XIOModule_DiscreteRead(&gpi, gpiNum) Læser status på `gpi`-porten. `gpiNum` bestemmer hvilken port der benyttes, af dem der er oprettet med MicroBlazen – for eksempel `1`, for "General Purpose Input 1".

Logkode udviklet til PC og MicroBlaze, Version 2

For i fremtiden bedre at kunne vedligeholde `log`-biblioteket, er det omskrevet til et mere funktionsbaseret layout, med inspiration fra Linux-kernelen. Koden ses på Bilag A.1, og på cd:

- `code/LOG_v2/{log.c, log.h}`
- `code/LOG_v2_mb/workspace/log/src/{log.c, log.h}`

Koden er indvendigt den samme som i version 1, men er opdelt i "lokale funktioner" (`static`), som kun kan ses inde i kildefilen selv, `log.c`. De "globale funktioner", beskrevet i Afsnit 7.1.3, er stadig de samme, men er nu mere simple. Det er hovedsageligt `sendToEarth()`-funktionen, der er blevet væsentligt simplere. Desuden er alle konstante værdier nu defineret i header-filen, `log.h` – det vil sige bitmasker og byteplacering i pakker.

Både version 1 og version 2 testes i Afsnit 8.1.

7.2 SPI

I det følgende beskrives udviklingen af SPI-bussen. Denne designes som en IP-blok. SPI-bussen vil håndtere kommunikationen med eksterne enheder på AAUSAT4. Disse eksterne enheder vil blive opkoblet som slaver på den designede SPI-master. Dette modul har grænseflader mod den interne bus imellem mikrocontrollerne og mod flash. Gennem den interne bus modtages beskeder fra andre moduler, der vil gemme eller læse noget i flashen. Dette modul designes som som VHDL IP-blok. Dette uddybes i det følgende.

7.2.1 SPI opbygning

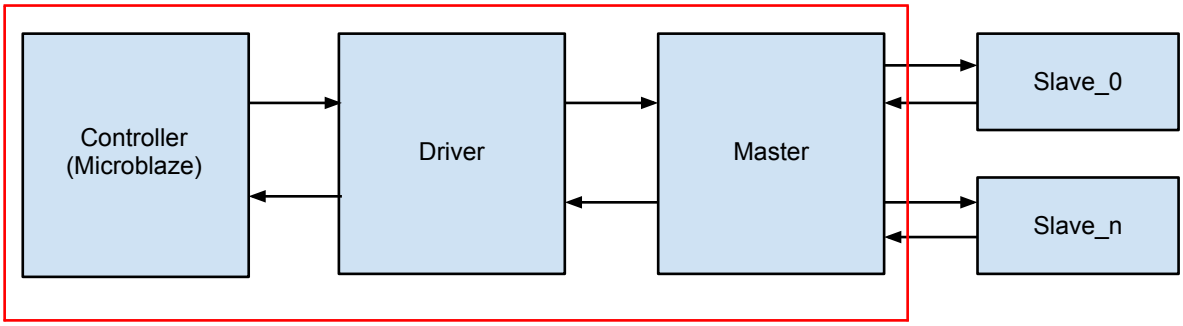
SPI-masteren opdeles i tre moduler, controller, driver og mastermodul. Disse er illustreret som overordnet SPI-system på Figur 7.7.

7.2.2 Master

For at opfylde kravspecifikation 5.2.1, er der valgt et open source VHDL-modul, som kan fungere som SPI-mastermodul. Kildekode og dokumentation til dette findes i litteraturlisten[8]. Til dette modul designes VHDL-driver og en software controller, implementeret på en MicroBlaze MCS. Først analyseres mastermodul, ved simulering af en driver. Derefter designes en driver og der simuleres en controller. Slutteligt designes en controller som MCS, hvori softwaren implementeres og systemet testes i henhold til testspecifikation, Afsnit 6.2.

7.2.3 Analyse af open source modul

SPI mastermodulet er udviklet som VHDL-logik. Udgiverens dokumentation findes i litteraturlisten [8]. Dette modul testes ved en række simuleringer i 'ISE-design - iSim'. Der tages udgangspunkt i VHDL-koden og derudfra bygges en test bench, hvor der simuleres en driver. Dermed klarlægges det, hvilke signaler der er nødvendige og

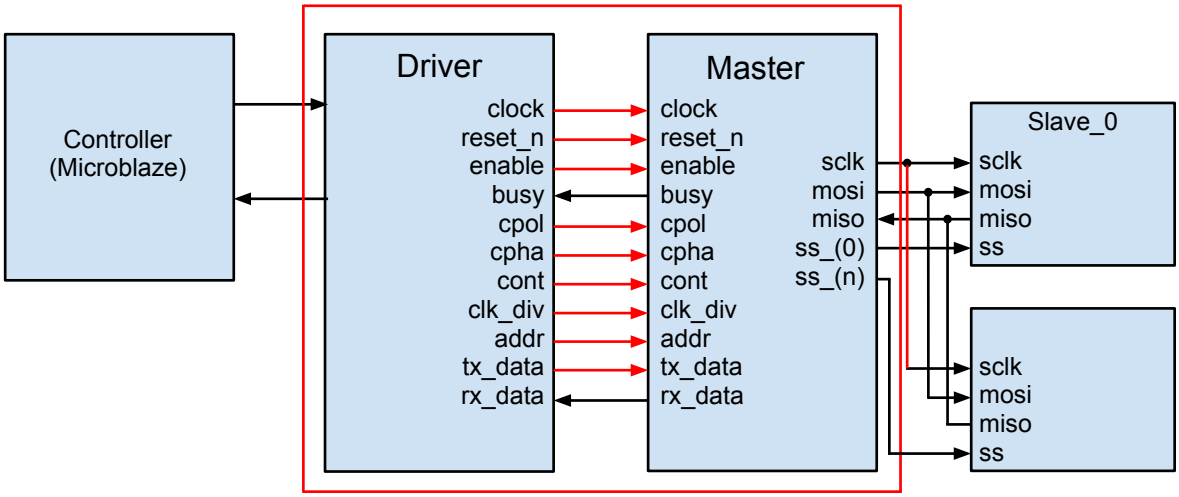


Figur 7.7: Overblik over SPI-master. Omkranset af den røde linie ses de tre moduler som tilsammen udgør SPI-master. Bemærk at alle kommandoer fra controller går gennem driver-modulet. Slaverne planlægges det at implementere eksternt i forhold til FPGA-chippen.

på hvilke tidspunkter i transaktionsprocessen det enkelte signal er nødvendigt, for at der eksempelvis overføres en besked fra master til slave. Her lægges vægt på at samtlige outputs fra master til slave overholder SPI-standarden. De specifikke signaler ind og ud af masteren som er testet i denne 'test bench', visualiseres på Figur 7.8. Et resultat af driver-simuleringen kan betragtes nærmere på Figur 7.9. Det ønskes at opsætte følgende 'test benches':

- Driver intialiserer og gennemfører en overførsel af en datapakke på masterens mosi ben.
- Flere pakker overføres direkte efter hinanden ved aktivering af continous mode.
- Tx_buffer overvåges og det undersøges om MSB eller LSB outputtes kommer først?

I kildekoden ses det, at der er mulighed for variabel databredde og slaveantal. Ligeledes findes et input til hendholdsvis fase og polaritet. Et output til sclk, mosi, miso og slave select.

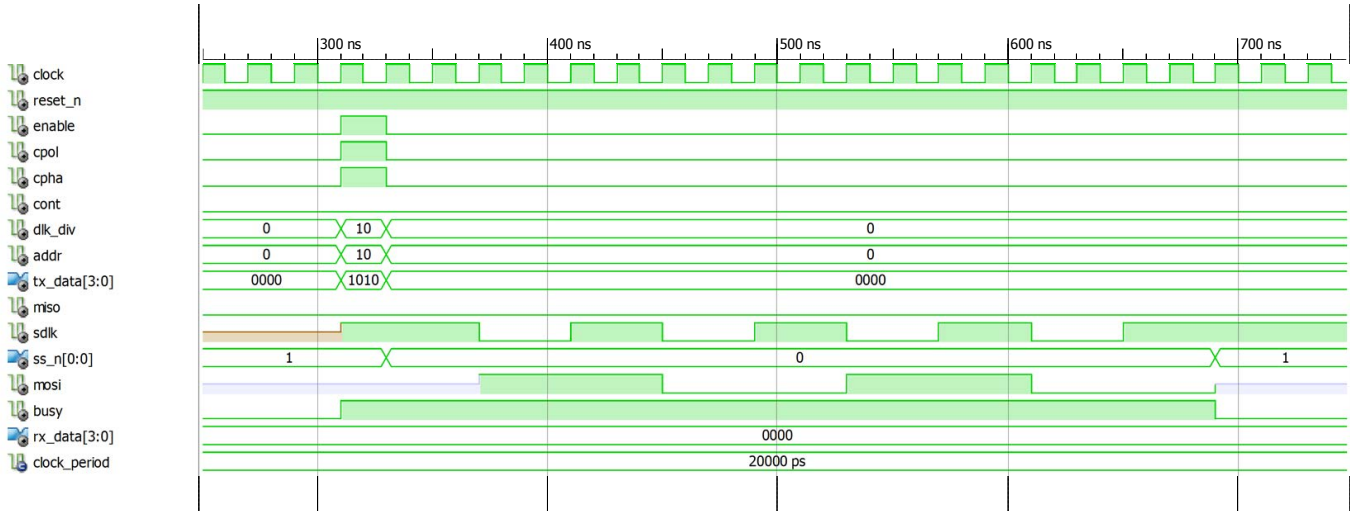


Figur 7.8: Signalerne angivet med rødt er alle simuleret i de følgende timing-diagrammer Figur 7.9, Figur 7.20 og Figur 7.11. Disse er udført ved at simulere en driver med disse inputs til mastermodulet.

Beskrivelse af SPI kommunikation

Modulet er opbygget som en state machine og overførsel af datapakke sker under følgende proces. Først skal modulet intialiseres med 'reset, høj', for at masteren går i 'ready state'. Clock divider, mode, adresse og data bliver alle latched når 'enable' går høj. Ved 'continue mode' bliver disse latched på sidste stigende flanke af 'sclk'. Når 'enable' går høj, startes 'sclk' og modulet går i 'execute' mode. Slaven vælges inden overførsel på 'mosi', som sker bitvis på de følgende stigende flanker af 'sclk'. På hvilken flanke læsning og skrivning foretages varierer, afhængigt af den valgte mode. Til sidst gives der signal om at modtaget data er klar. Hele kildekoden kan ses i bilag eller på udviklernes egen side[8].

bilag/eewiki.pdf



Figur 7.9: Timing-diagram over en succesfuld afsending af en 4-bit datapakke. Først sættes reset, derefter clkdiv, mode, adresse og data. Disse bliver alle 'latched' af master på enable-pulsen, som her er simuleret til en clockperiode. På enable starter sclk som er 'høj' i tre clock-perioder, hvorefter data sættes på mosi-benet. Mosi-signalet skrives til læsning af slave0 på de følgende 4 rising edges af sclk. Busy er 'høj under' under hele transaktionen. Bemærk også at reset er 'høj'. Dette er for at master kan gå i 'ready' mode.

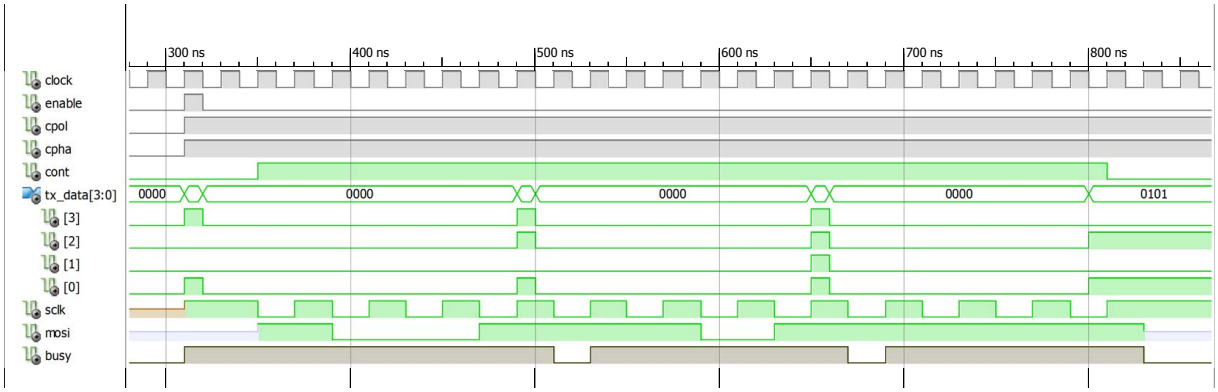
Ready state

Master venter på inputs, mens data output er højimpedant. Når 'enable' sættes høj, sættes adresse, hastighed, mode og fase. Data læses ind i bufferen, tælleren intialiseres og pakkestørrelsen defineres. Herefter skiftes state til 'execute'.

```
1        WHEN ready =>
2          busy     <= '0';             --clock out not busy signal
3          ss_n     <= (OTHERS => '1'); --set all slave select outputs high
4          mosi     <= 'Z';             --set mosi output high impedance
5          continue <= '0';             --clear continue flag
6
7          --user input to initiate transaction
8          IF(enable = '1') THEN
9            busy    <= '1';           --set busy signal
10           IF(addr < slaves) THEN    --check for valid slave address (fx. slaves=8 addresses: 7 downto 0)
11             slave <= addr;          --clock in current slave selection if valid
12           ELSE
13             slave <= 0;             --set to first slave if not valid
```

```
14            END IF;
15            IF(clk_div = 0) THEN      --check for valid spi speed
16              clk_ratio <= 1;         --set to maximum speed if zero
17              count <= 1;             --initiate system-to-spi clock counter
18            ELSE
19              clk_ratio <= clk_div;   --set to input selection if valid
20              count     <= clk_div;   --initiate system-to-spi clock counter
21            END IF;
22            sclk        <= cpol;      --set spi clock polarity
23            assert_data <= NOT cpha;  --set spi clock phase
24            tx_buffer   <= tx_data;   --clock in data for transmit into buffer
25            clk_toggles <= 0;         --initiate clock toggle counter
26            last_bit_rx <= d_width*2 + conv_integer(cpha) - 1; --set last rx data bit
27            state       <= execute;   --proceed to execute state
28          ELSE
29            state <= ready;           --remain in ready state
30          END IF;
```

Listing 7.6: Kildekode fra Master. Her ses at master er i 'ready state' og venter på inputs.



Figur 7.10: Dette timing-diagram viser en succesfuld afsending af tre datapakker på henholdsvis 4 bit. Den første transaktion initialiseres med enable-pulsen. To clock-cycles senere sættes cont-benet 'høj', for at fortsætte med flere pakker. Bemærk at tx_data kun udsættes en halv clock-cycle ad gangen for at understrege hvornår masteren latcher ind i tx_buffer. Dette medfører at næste pakkes data skal være tilgængeligt på tx_data-benet på sidste rising_edge af sclk i hver pakke. På denne flanke bliver alle kommandoer læst af masteren. Bemærk at master går 'lav' på netop denne flanke ved sidste pakke og afslutter transaktion med succes.

Execute state

Masteren signalerer at den er optaget og vælger slave. 'Sclk' resettes herefter og denne 'toggles', indtil sidste bit er overført. rx_buffer er designet som skifteregister, som efter hver toggle sammenkædes med sidste modtaget bit. Ligeledes sammenkædes tx_buffer med et '0' når data er skrevet på mosi. Når sidste bit er overført slutter transaktionen og modtaget data sættes ud på data benene hvormed master går i 'ready' state. Hvis continue er høj overføres næste pakke uden pause og data sættes ud på data benene efter hver pakke er overført. Når data sættes ud går 'busy' altid lav og dette skal udnyttes til at kommunikere med controlleren.

```
1         WHEN execute =>
2           busy <= '1';        --set busy signal
3           ss_n(slave) <= '0'; --set proper slave select output
4
5           --system clock to sclk ratio is met
6           IF(count = clk_ratio) THEN
```

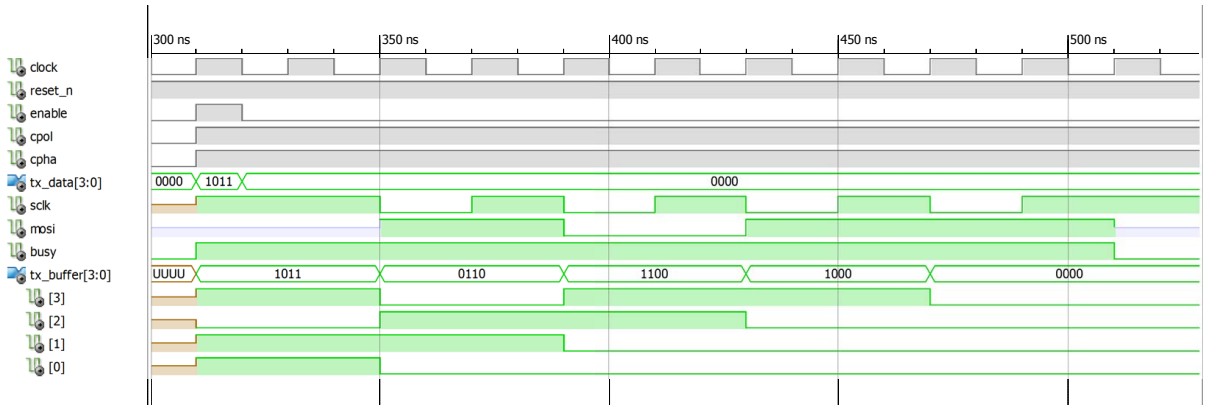
```
7           count <= 1;                    --reset system-to-spi clock counter
8           assert_data <= NOT assert_data; --switch transmit/receive indicator
9           clk_toggles <= clk_toggles + 1; --increment spi clock toggles counter
10
11          --spi clock toggle needed
12          IF(clk_toggles <= d_width*2 AND ss_n(slave) = '0') THEN
13            sclk         <= NOT sclk; --toggle spi clock
14          END IF;
15
16          --receive spi clock toggle
17          IF(assert_data = '0' AND clk_toggles < last_bit_rx + 1 AND ss_n(slave) = '0') THEN
18            rx_buffer   <= rx_buffer(d_width-2 DOWNTO 0) & miso; --shift in received bit
19          END IF;
20
21          --transmit spi clock toggle
22          IF(assert_data = '1' AND clk_toggles < last_bit_rx) THEN
23            mosi        <= tx_buffer(d_width-1);                    --clock out data bit
24            tx_buffer   <= tx_buffer(d_width-2 DOWNTO 0) & '0'; --shift data transmit buffer
25          END IF;
26
27          --last data receive, but continue
28          IF(clk_toggles = last_bit_rx AND cont = '1') THEN
29            tx_buffer   <= tx_data;                       --reload transmit buffer
30            clk_toggles <= last_bit_rx - d_width*2 + 1; --reset spi clock toggle counter
31            continue    <= '1';                            --set continue flag
32          END IF;
33
34          --normal end of transaction, but continue
35          IF(continue = '1') THEN
36            continue <= '0';      --clear continue flag
37            busy     <= '0';          --clock out signal that first receive data is ready
38            rx_data  <= rx_buffer; --clock out received data to output port
39          END IF;
40
41          --end of transaction
42          IF((clk_toggles = d_width*2 + 1) AND cont = '0') THEN
43            busy    <= '0';             --clock out not busy signal
44            ss_n    <= (OTHERS => '1'); --set all slave selects high
45            mosi    <= 'Z';             --set mosi output high impedance
46            rx_data <= rx_buffer;    --clock out received data to output port
47            state   <= ready;          --return to ready state
48          ELSE                       --not end of transaction
49            state <= execute;        --remain in execute state
50          END IF;
51
52        ELSE                         --system clock to sclk ratio not met
53          count <= count + 1; --increment counter
54          state <= execute;   --remain in execute state
55        END IF;
```

Listing 7.7: Kildekode fra master. Her ses at masteren er i 'execute state' og overfører data.

Egenskaber for SPI-master

Ved at simulere forskellige værdier af driver-signaler til masteren er det bekræftet at dette modul kan styres ved at bygge en controller til denne. Den tilgængelige dokumentation til dette modul[8] er bekræftet og de væsentlige egenskaber for master VHDL-filen er.

- sclk, mosi, miso og ss overholder generel SPI-standard.
- Mulighed for 'continous mode' til overførsel af store datapakker.



Figur 7.11: I dette Timing-diagram overvåges tx_buffer, som er en intern buffer i masteren. Det ses tydeligt ud fra denne simulering at MSB sendes først på miso-benet.

- Variabelt antal tilslutninger af slaver.
- Mulighed for 4 'transfer modes'.
- Variabel clock.
- Variabel databredde.

Da mastermodulets funktionalitet nu er bekræftet ved analyse af kildekode og ved simulering af en driver, vil der påbegyndes udvikling af et drivermodul, hvilket designes som simpel logikblok.

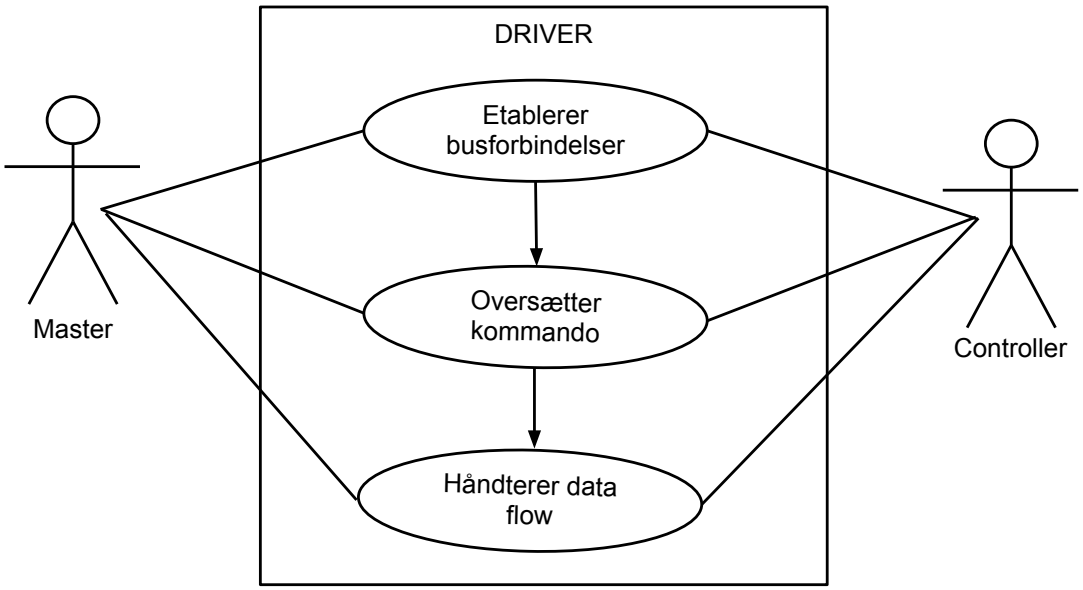
7.2.4 Driver

I dette afsnit beskrives udviklingen af drivermodulet. Først opstilles simple krav til funktionalitet ved opstilling af use cases. Grænseflader og funktionelle krav bliver ligeledes defineret, før dokumentationen af den reelle udvikling påbegyndes.

I grænsefladen mellem controller og master designes en driver som oversætter kommandoer fra MicroBlazen og sender disse til masteren. Driveren starter og stopper masterkommunikation med signalerne 'reset', 'enable' og 'cont'. Driveren må tage højde for om masteren er optaget og i så fald vente. Driveren har til opgave konstant at læse kommando og data fra controlleren og oversætte og videresende dette til VHDL-mastermodulet. Den må ligeledes sørge for at gøre controlleren opmærksom på hvornår ny data er klar til læsning fra buffer og hvornår data til afsendelse er indlæst i buffer.

7.2.5 Funktionelle krav til driver

For at definere de funktionelle krav til SPI-drivermodulet opstilles der først use cases over driveren. På denne måde dannes et overblik over grænsefladerne og de enkelte funktionaliteter. Som det ses på Figur 7.12 skal driver modtage kommando fra controlleren, oversætte denne og videresende kommandoen til masteren. Grænsefladen mod masteren er defineret i Afsnit 7.2.3. Grænsefladen mod controller implementeres som et register, der kan indeholde de nødvendige data til at definere en kommando fra controlleren. Denne kommando er analyseret i Afsnit 7.2.3. Da driveren bliver udviklet som hardware, er data- og opsætningsforbindelser hardwired mellem controller og master. Symbolsk er dette en del af kontrol-registeret, da denne grænseflade designes i driver-koden. Dette kan betragtes på use case Figur 7.14. For at starte en transaktion, må driveren vurdere om masteren er klar, før transaktionen begyndes, hvilket er specificeret i use case Figur 7.14.



Figur 7.12: Use case diagram over driveren. Driveren har grænseflade mod controlleren og masteren og har til opgave at etablere forbindelse og oversætte kommando. Driveren må også håndtere hvornår data er afsendt og modtaget, således at controller modtager og afsender alle de ønskede pakker

Use Case	Etablerer forbindelser
Description	Faste forbindelser til bus etableres mellem controller og master
Assumptions	
Actors	Controller & Master
Steps	1. REPEAT 1.1 IN PARALLEL 1.1.1 Modtag datasignaler 1.1.2 Modtag addressesignaler 1.1.3 Modtag Clock-divider signaler 1.1.4 Modtag Mode signaler 1.1.5 Modtag Continue signaler 1.1.6 Output datasignaler til Master 1.1.7 Output addressesignaler til Master 1.1.8 Output Clock-divider til Master 1.1.9 Output Mode til Master 1.1.10 Output Continue til Master 1.1.11 Overtag variablen 'databredde' fra Master 1.1.12 Overtag variablen 'slaveantal' fra Master 1.1.13 Send Clock-signal til Master

Figur 7.13: Use case over forbindelserne som er hardwired mellem controller og master. I denne use case er specificeret at databredde og slaveantal skal opsættes i denne fil, fremfor implementationen i mastermodulet.

Use Case	Oversætter kommando
Description	Kommandoer fra controller omsættes til afvikling af transaktion
Assumptions	
Actors	Controller & Master
Steps	1. REPEAT 1.1 IF controller anmoder om transaktion AND master er klar til transaktion 1.1.1 Aktiver master til transaktion

Figur 7.14: Use case for oversætter kommando

Use Case	Håndterer data-flow
Description	Notificèr controller om at ny data er klar og om at sendt data er modtaget.
Assumptions	
Actors	Controller & Master
Steps	1. REPEAT 1.1 IF pakke netop er skrevet på rx_data på master 1.1.1 Send signal til controller om at der skal læses 1.2 IF pakke er indlæst i tx_buffer på master 1.2.1 Send signal til controller om at der skal gives ny kommando

Figur 7.15: Use case for håndtering af data flow

Opsummering af krav

- Den skal 'resette' masteren ved initialisering
- Den skal koble clk_div, mode, cont, adresse, rx_data og tx_data på masteren, direkte fra MicroBlazen.
- Den skal afvikle transaktion når startbit er 'høj' og master ikke er 'busy'.
- Den skal fortælle controlleren at transaktion er igangsat, så denne kan præsentere ny data inden sidste rising_edge af sclk eller ligeledes vente med at skifte data i tilfælde af ventetid.
- Den skal fortælle controlleren at pakken er modtaget og ny data er klar.

7.2.6 Grænseflader

I grænsefladen mellem controller og driver defineres et kontrolregister og et driverregister med fast størrelse. Kontrolregisteret indeholder de nødvendige parametre til at styre master. Følgende krav stilles til denne.

Krav til kontrolregister

- Den skal bygges som general purpose register.
- Den skal indeholde 8 bit med data til afsending.
- Den skal indeholde 4 bit til clock divider.
- Den skal indeholde 5 bit til slaveadresser.
- Den skal indeholde 2 til mode. (polarity og phase)
- Den skal indeholde en startbit som angiver om der skal transmitteres.
- Den skal indeholde en continue bit til overførsel af store pakker

Krav til driverregister

Driverregisteret indeholder de nødvendige parametre for at håndtere data-flow og det designes som listet her.

- Det skal indeholde otte bit til data
- Det skal indeholde et data klar flag
- Det skal indeholde et data modtaget flag

Da SPI-modulets indre grænseflader nu er fast defineret, kan udvikling af et drivermodul og et controller-modul påbegyndes. Kontrolregisteret opbygges med SI21bit som illustreret på Figur 7.16

Figur 7.16: SPI-controller sender en kommandobesked på SI21bit til VHDL driveren. Kommandoen indeholder 4 bit til clock divider, 2 bit til mode, 8 bit til data, 5 bit til adresse, en startbit hvorved 'høj' angiver at der skal transmitteres og til sidst en continue bit som efter endt overførsel, fortsætter sclk indtil cont går 'lav'.

7.2.7 Kode

```
1  ----------------------------------------------------------------------------
2  -- SPI DRIVER
3  ----------------------------------------------------------------------------
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6  use IEEE.std_logic_unsigned.all;
7
8
9  entity spi_driver is
10     GENERIC(
11                slaves       : INTEGER := 1;                    -- number of spi slaves
12                d_width      : INTEGER := 8                     -- data bus width
13            );
14     Port( clk          : IN  STD_LOGIC;
```

```
15          ctrl_reg     : IN  STD_LOGIC_VECTOR(20 downto 0); -- register from controller
16          drvr_reg     : OUT STD_LOGIC_VECTOR(9 downto 0);  -- output to controller
17          sclk         : out STD_LOGIC;
18          mosi         : out std_logic;
19          miso         : in std_logic := 'Z';
20          ss_n         : out STD_LOGIC_VECTOR(slaves-1 DOWNTO 0)
21      );
22
23  end spi_driver;
24
25  architecture Behavioral of spi_driver is
26      SIGNAL  busy       : STD_LOGIC;                                    -- busy / data ready signal
27      SIGNAL  reset_n    : STD_LOGIC:='0';                               -- asynchronous reset
28      SIGNAL  enable     : STD_LOGIC;                                    -- initiate transaction
29      SIGNAL  cont       : STD_LOGIC:='0';                               -- continuous mode command
30      SIGNAL startbit    : STD_LOGIC;                                    -- transaction begin flag
31      SIGNAL sclk_to_out : STD_LOGIC;                                    -- for converting buffer to out
32      SIGNAL slaveselect : STD_LOGIC_VECTOR(slaves-1 DOWNTO 0);
33    ---------------------------------------------------------------------------------
34    --signals for driver_register (drvr_reg)
35      SIGNAL rx_data_rdy_for_ctrl : STD_LOGIC:='0';                      -- if '1' recieved data package from selected slave
36      SIGNAL tx_data_latched      : STD_LOGIC:='0';                      -- if '1' data has been latched into buffer, driver is
            ready for new data input
37      SIGNAL rx_data              : STD_LOGIC_VECTOR(d_width-1 DOWNTO 0); -- input data from slave -- not tested yet
38  ---------------------------------------------------------------------------------
39  begin
40
41      master: entity work.spi_master(logic)
42      generic map (d_width   => d_width,            -- data width of master is controlled from this file
43                   slaves    => slaves )            -- number of spi slaves is controlled from this file
44      port map(      busy    => busy,
45                     clock   => clk,
46                     reset_n => reset_n,
47                     enable  => enable,
48                     cont    => ctrl_reg(20),
49                     rx_data => rx_data,
50                   --interface from spi_driver spi_master.vhdl (ctrl_reg)
51                     clk_div => CONV_INTEGER(ctrl_reg(19 downto 16)),
52                     cpol    => ctrl_reg(15),
53                     cpha    => ctrl_reg(14),
54                     tx_data => ctrl_reg(13 downto 6),
55                     addr    => CONV_INTEGER(ctrl_reg(5 downto 1)),
56                   -- interface to slaves
57                     sclk    => sclk_to_out,
58                     mosi    => mosi,
59                     miso    => miso,
60                     ss_n    => slaveselect
61                );
62
63      sclk <= sclk_to_out;        --conversion of type: 'buffer' from master
64      ss_n <= slaveselect;  --conversion of type: 'buffer' from master
65    ---------------------------------------------------------------------------------
66    -- Process of transmission
67      startbit <= ctrl_reg(0);               --transaction begin flag
68
69      process( startbit, busy )
70      begin
71          reset_n<='1';
72          if  startbit='1' and busy='0' then  --check for ready signal from master, before intializing transaction
73              enable<='1';
74          else
75              enable<='0';
76          end if ;
77      end process;
78
```

```
79   ---------------------------------------------------------------------------------
80   ------ Driver register ------
81     drvr_reg(0)          <= rx_data_rdy_for_ctrl; --'1' data package recieved from selected slave
82     drvr_reg(1)          <= tx_data_latched;      --'1' tx_data latched into buffer, ready for new data package
83     drvr_reg(9 downto 2) <= rx_data;              -- data byte from slave to controller
84
85  -- Process of data flow control
86     Process(busy)
87     begin
88         if  busy='1' then
89             tx_data_latched <= '1';           -- flag for controller to change data output
90         else
91             tx_data_latched <= '0';
92         end if ;
93         if busy='0' then
94             rx_data_rdy_for_ctrl <= '1';    -- flag for controller to read data
95         else
96             rx_data_rdy_for_ctrl <= '0';
97         end if ;
98     end Process;
99
100 end Behavioral;
```

Listing 7.8: Kilde kode for driver.

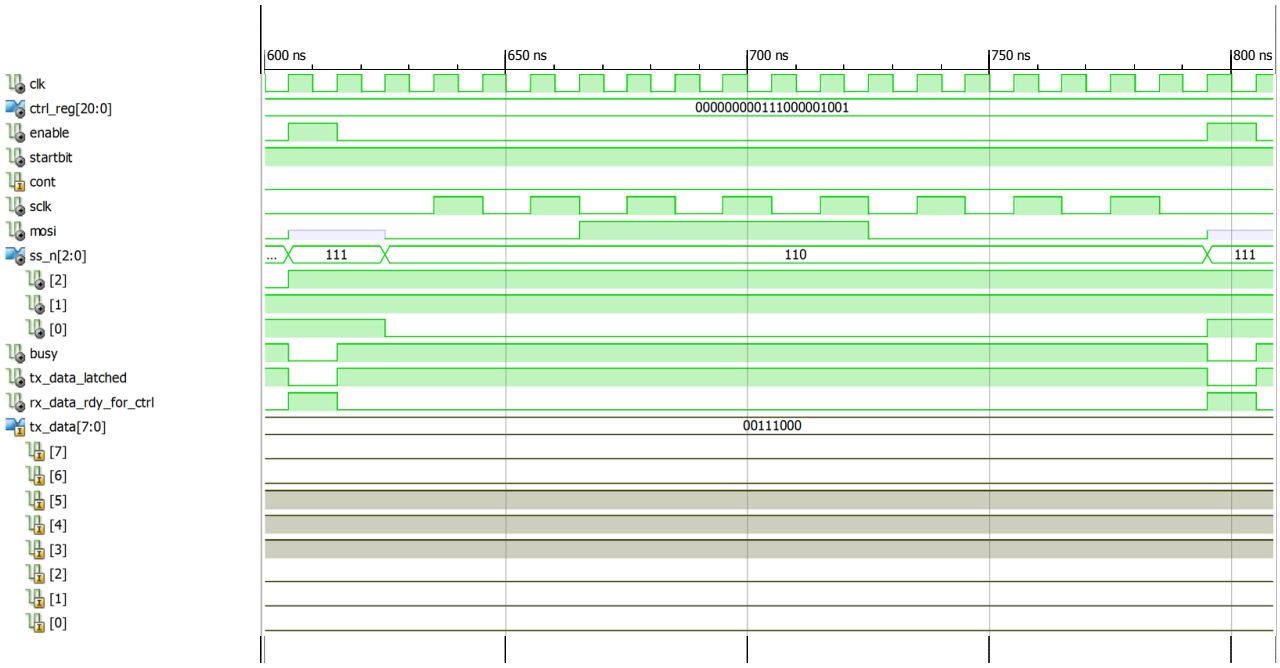
7.2.8 Deltest

Det ønskes at teste driveren ved at simulere et kontrolregister og derfor opstilles følgende test benches.

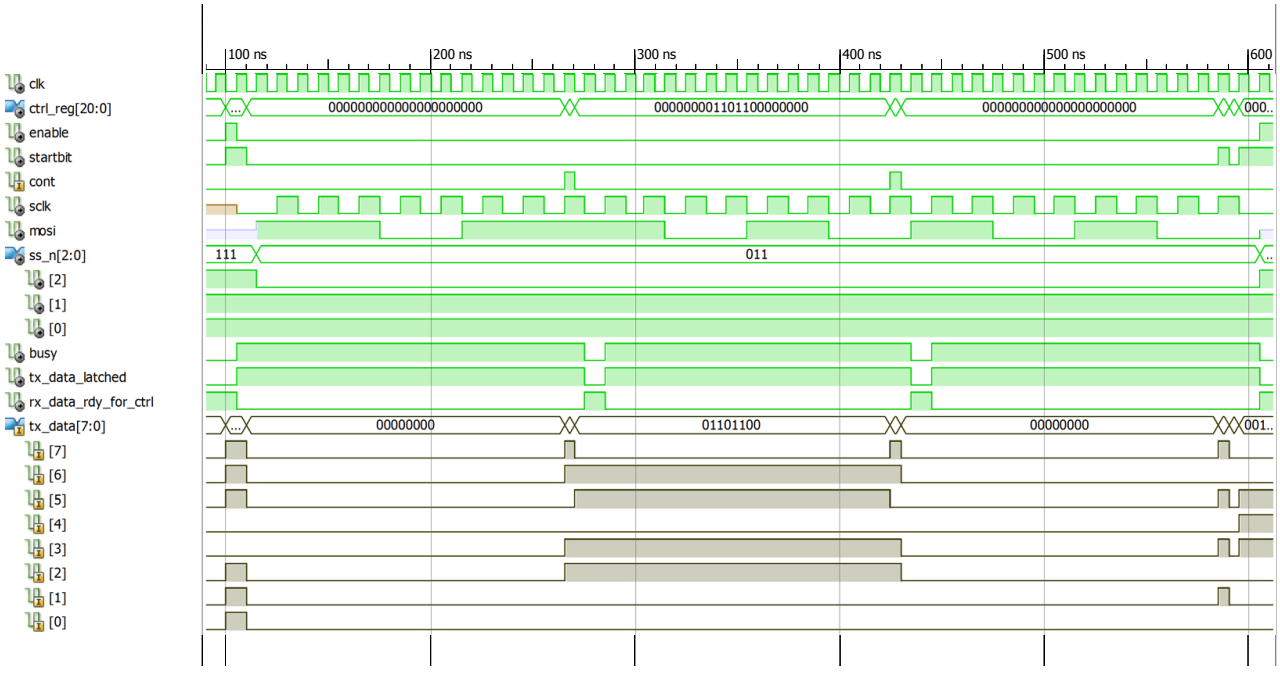
- Overførsel af en pakke
- Overførsel af tre pakker ved continuos mode. Det skal defineres præcist hvornår der latches kommando og data ind i master. Derfor skal der kun sendes 'cont' høj, når denne skal latches på sidste rising edge af 'sclk'.

7.2.9 IP blok

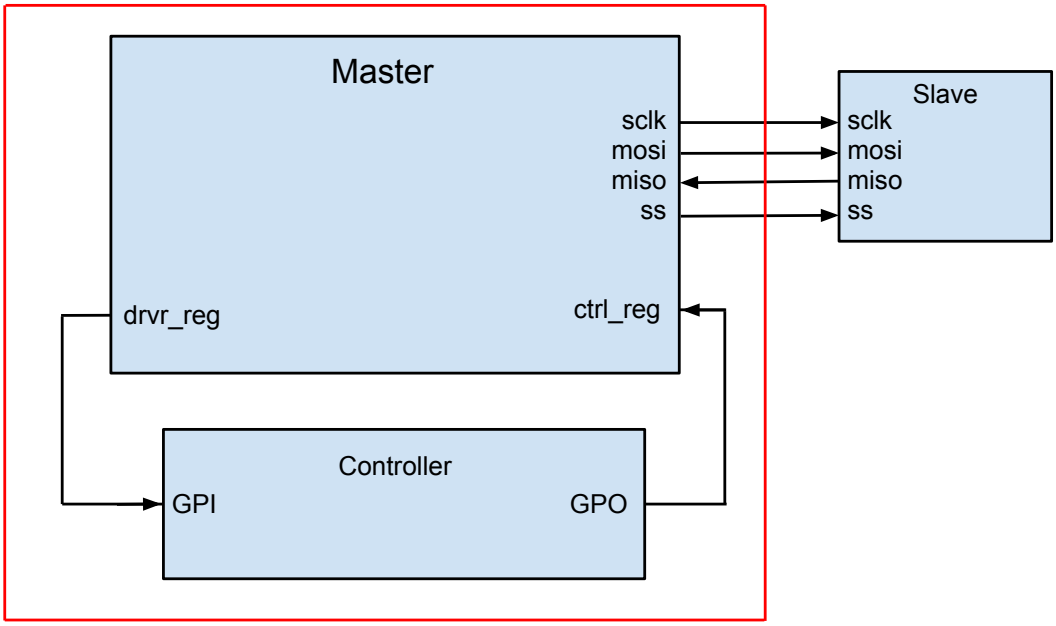
Til sidst desiges en IP-blok hvor controller, master og driver alle er implementeret i. Denne vil derfor have grænseflader mod den interne bus og mod slaverne. Kan ses som Figur 7.19.



Figur 7.17: Resultat af test hvor der overføres en pakke. Kontrolregisteret er simuleret. Resultatet er som forventet at der overføres en pakke efter enable har startet transaktionen.



Figur 7.18: Resultat af test hvor der overføres tre pakker. Kontrolregisteret er simuleret. Resultatet er som forventet at der overføres tre pakker. Continue mode er kun sat høj på sidste stigende flanke af 'sclk' for at understrege latch-tidspunkt.



Figur 7.19: Det færdige SPI-modul er omkranset af den røde linie. Modulet har grænsflader til slaverne og til den interne bus mellem mikrocontrollerne.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4
5  ENTITY driver_test IS
6      END driver_test;
7
8  ARCHITECTURE behavior OF driver_test IS
9
10     -- Component Declaration for the Unit Under Test (UUT)
11
12     COMPONENT spi_driver
13         PORT(
14                 clk      : IN  std_logic;
15                 ctrl_reg : IN  std_logic_vector(20 downto 0);
16                 sclk     : OUT  std_logic;
17                 mosi     : OUT  std_logic;
18                 miso     : IN  std_logic;
19                 ss_n     : OUT  std_logic_vector(0 downto 0)
20             );
21     END COMPONENT;
22
23
24    --Inputs
25     signal clk : std_logic := '0';
26     signal ctrl_reg : std_logic_vector(20 downto 0) := (others => '0');
27     signal miso : std_logic := '0';
28
29    --Outputs
30     signal sclk : std_logic;
31     signal mosi : std_logic;
32     signal ss_n : std_logic_vector(0 downto 0);
33
```

```
34    -- Clock period definitions
35     constant clk_period : time := 10 ns;
36     constant sclk_period : time := 10 ns;
37
38  BEGIN
39
40    -- Instantiate the Unit Under Test (UUT)
41     uut: spi_driver PORT MAP (
42                                  clk      => clk,
43                                  ctrl_reg => ctrl_reg,
44                                  sclk     => sclk,
45                                  mosi     => mosi,
46                                  miso     => miso,
47                                  ss_n     => ss_n
48                               );
49
50    -- Clock process definitions
51     clk_process :process
52     begin
53         clk <= '0';
54         wait for clk_period/2;
55         clk <= '1';
56         wait for clk_period/2;
57     end process;
58
59
60    -- Stimulus process
61     stim_proc: process
62     begin
63       -- hold reset state for 100 ns.
64         wait for 100 ns;
65         ctrl_reg <="00000011100111000001";
66
67         wait for clk_period*1;
68         ctrl_reg <="00000000111100000001";
69
70         wait for clk_period*10;
71         ctrl_reg <="00000000000000000000";
72
73
74       -- insert stimulus here
75         wait;
76     end process;
77
78  END;
```

Listing 7.9: Test bench med simuleret kontrolregister som vil overføre en pakke.

```
1     -- Stimulus process
2   stim_proc: process
3   begin
4         -- hold reset state for 100 ns.
5     wait for 100 ns;
6     ctrl_reg <="00000011100111000001";
7
8     wait for clk_period*1;
9     ctrl_reg <="00000000000000000000";
10
11    wait for 155 ns;
12    ctrl_reg <="10000011001100000010";
13    wait for 5 ns;
14    ctrl_reg <="00000001101100000000";
15    wait for 155 ns;
16    ctrl_reg <="10000011001100000000";
17    wait for 5 ns;
```

```
18      ctrl_reg <="00000000000000000100";
19      wait for 155 ns;
20      ctrl_reg <="00000010101010000000";
21      wait for 5 ns;
22      ctrl_reg <="00000000000000000000";
23        -- insert stimulus here
24      wait;
25  end process;
```

Listing 7.10: Test bench hvor der overføres med continue mode

```
1   ----------------------------------------------------------------------------------
2   -- spi implementation of master, driver and controller
3   ----------------------------------------------------------------------------------
4   library IEEE;
5   use IEEE.STD_LOGIC_1164.ALL;
6
7   entity spi_block is
8       GENERIC(
9                  slaves  : INTEGER : = 1;                     --number of spi slaves
10                 d_width : INTEGER : = 8 );
11      Port (       clk   : in  STD_LOGIC;
12                   miso  : in  STD_LOGIC;
13                   mosi  : out  STD_LOGIC;
14                   sclk  : out  STD_LOGIC;
15                   ss_n  : out  STD_LOGIC_VECTOR(slaves-1 DOWNTO 0);
16                   reset : in  STD_LOGIC);
17  end spi_block;
18
19  architecture implementation of spi_block is
20      SIGNAL ctrl_reg    : STD_LOGIC_VECTOR(20 downto 0);
21      SIGNAL drvr_reg    : STD_LOGIC_VECTOR(9 downto 0);
22
23      COMPONENT spi_ctrl
24          PORT (
25                   Clk  : IN STD_LOGIC;
26                   Reset: IN STD_LOGIC;
27                   GPO1 : OUT STD_LOGIC_VECTOR(20 DOWNTO 0);
28                   GPI1 : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
29                   GPI1_Interrupt : OUT STD_LOGIC
30              );
31      END COMPONENT;
32
33
34  begin
35      drvr_mstr: entity work.spi_driver(behavioral)
36      generic map (d_width => d_width,            -- data width of master is controlled from this file
37                   slaves  => slaves )
38      port map(   clk      => clk,
39                  ctrl_reg => ctrl_reg,  -- register from controller to driver
40                  drvr_reg => drvr_reg,  -- output to controller from driver
41                  sclk     => sclk,
42                  mosi     => mosi,
43                  miso     => miso,
44                  ss_n     => ss_n
45              );
46
47      spi_controller : spi_ctrl
48      PORT MAP (
49                   Clk            => clk,
50                   Reset          => reset,
51                   GPO1           => ctrl_reg,
52                   GPI1           => drvr_reg,
53                   GPI1_Interrupt => open
54              );
```

```
55
56 end implementation;
```

Listing 7.11: Kilde kode for SPI IP-blok.

7.2.10 SPI C bibliotek

I det følgende beskrives udviklingen af SPI-software controlleren. Denne er designet på en MikroBlaze. Hensigten er at gøre SPI-bussen tilgængelig for fremtidige AAUSAT-programmører. Derfor udvikles der et bibliotek til en MicroBlaze softcore MCU. Bibliotekets funktion er at håndtere kommunikationen mellem kode-lag og hardware-register, således, at det er tilgængeligt for programmøren at igangsætte kommunikation via SPI modulet, blot ved at kalde enkelte funktioner. Koden skrives i C, og der er gjort brug af Xilinx proprietære biblioteker til I/O funktionaliteten. Opsummeret skal biblioteket:

- Kodes i C til en MicroBlaze MCS.
- Biblioteket skal styre om der transmitteres data over SPI-bussen.
- Biblioteket skal styre master VHDL-driveren med en datapakke.

Biblioteket indeholder funktioner til opsætning og initialisering af SPI modulet: `spiInit()`, en funktion til at sende enkelte pakker: `spiSend(data)`, en funktion til at modtage enkelte pakker: `spiRecv()`, samt en funktion til at lave en "hel" overførsel, denne funktion har således funktionskald til de ovennævnte pakker: `spiTrans(data)`.

spiInit()

`spiInit()` har til funktion, at modtage de opsætningsparametre, som programmøren ønsker at SPI modulet skal bruge, herunder CPHA, CPOL samt mulighed for at dividere clocken med 2, 4 og 8 – som tidligere beskrevet i Afsnit 7.2.1. For at lave denne opsætning skal parameterne, placeres bitvis korrekt i pakken defineret på Figur 7.16

spiInit funktionen er blevet implementeret på følgendevis:

```
1  #define SPI_CLKDIV2 0x20000
2  #define SPI_CLKDIV4 0x40000
3  #define SPI_CLKDIV8 0x80000
4  #define SPI_MODE0 0x0
5  #define SPI_MODE1 0x4000
6  #define SPI_MODE2 0x8000
7  #define SPI_MODE3 0xC000
8  u32 = spiCond;
9
10 void spiInit(u32 initCond)
11 {
12   XIOModule_DiscreteWrite(&spiOut, SPI_CH, initCond);
13   spiCond = initCond;
14 }
```

Som det fremgår af koden, tager funktionen `spiInit` kun imod en 32 bit unsigned variabel. Til gengæld er der en række defines, som skal `OR`'s sammen, for at konstruerer opsætningen til SPI-modulet. Dvs at et typisk funktionskald kunne se ud på følgende måde: `spiInit(SPI_CLKDIV2|SPI_MODE2)`.

spiSend() Denne funktion skal sende 8 bit ud til registeret, samt starte SPI-overførslen. Funktionen tager således kun imod et argument, nemlig datapakken, som er en 8 bit unsigned integer. Igen skal det huskes at, datapakken skal indsættes det korrekte sted i kontrolregisteret, som følge af definitionen på Figur 7.16.

Funktionen `spiSend()` implementeres således, at funktionen ikke afsluttes før BUSY-benet fra hardware-modulet går lavt. På denne vis sikres at der ikke overskrives datapakker eller læses datapakker før modtaget data er tilstede.

`spiSend` er implementeret som følgende:

```
1  #define SPI_SEND 0x1
2  #define SPI_CH 1
3  #define SPI_BUSY_PIN 4
4  #define SPI_BUSY_MSK 0b0001000
5
6  int spiSend(u32 slave, u32 data)
7  {
8    //Shift bits to correct location.
9    slave = slave << 1;
10   data = data << 6;
11   //Send to SPI module
12   XIOModule_DiscreteWrite(&spiOut, SPI_CH, spiCond|slave|data|SPI_SEND);
13   while(!((XIOModule_DiscreteRead(&spiIn,SPI_CH)|SPI_BUSY_MSK)>>(SPI_BUSY_PIN-1)));
14   //Restore the initial conditions
15   XIOModule_DiscreteWrite(&spiOut, SPI_CH, spiCond);
16 }
```

Listing 7.12: spiSend()

Igen `OR`'s de forskellige opsætninger for at samle konfigurationen, og til sidst resettes der til den oprindelige opsætning, så SPI-modulet ikke bliver ved med at sende det samme data ud.

spiRecv() `spiRecv`, skal læse den data, som der er blevet læst af hardware-modulet. Dette er gøres ved at læse 8 bit fra driverregister, hvilket er defineret med `SPI_CH`. Funktionen retunerer en unsigned 8 bit integer. Denne er implementeret som vist i det følgende:

```
1  u8 spiRecv()
2  {
3    u32 dataReg;
4    dataReg = XIOModule_DiscreteRead(&spiIn, SPI_CH);
5    return (dataReg & 0b11111111);
6  }
```

Listing 7.13: spiRecv()

`dataReg` `AND`'s med et bitmønster, således at det matcher de første 8 bit i GPI-register fra MicroBlaze MCU'en, som er 32 bit og det retuneres derefter.

spiTran() Denne funktion, implementerer en samlet pakke, der bliver både skrevet og læst fra SPI modulet, funktionen indeholder blot funktionskald til `spiInit`, `spiSend` og `spiRecv`.

```
1  u8 spiTran(int setup, int slave , int data, int delay)
2  {
3    spiInit(setup);
4    spiSend(slave, data);
5    sleep(delay);
6    return spiRecv();
7  }
```

Listing 7.14: spiTran()

Kommentarer

Som det fremgår af kodeeksemplerne, er der ikke nogen form for fejlhåndtering i softwaren. Derfor skal programmøren være varsom med parametrene som sendes til de forskellige funktioner, således der ikke sker et overflow.

7.2.11 SPI C bibliotek

I det følgende beskrives udviklingen af SPI-software controlleren. Denne er designet på en MikroBlaze. Hensigten er at gøre SPI-bussen tilgængelig for fremtidige AAUSAT-programmører. Derfor udvikles der et bibliotek til en MicroBlaze softcore MCU. Bibliotekets funktion er at håndtere kommunikationen mellem kode-lag og hardware-register, således, at det er tilgængeligt for programmøren at igangsætte kommunikation via SPI modulet, blot ved at kalde enkelte funktioner. Koden skrives i C, og der er gjort brug af Xilinx proprietære biblioteker til I/O funktionaliteten. Opsummeret skal biblioteket:

- Kodes i C til en MicroBlaze MCS.
- Biblioteket skal styre om der transmitteres data over SPI-bussen.
- Biblioteket skal styre master VHDL-driveren med en datapakke.

Biblioteket indeholder funktioner til opsætning og initialisering af SPI modulet: `spiInit()`, en funktion til at sende enkelte pakker: `spiSend(data)`, en funktion til at modtage enkelte pakker: `spiRecv()`, samt en funktion til at lave en "hel" overførsel, denne funktion har således funktionskald til de ovennævnte pakker: `spiTrans(data)`.

spiInit()

`spiInit()` har til funktion, at modtage de opsætningsparametre, som programmøren ønsker at SPI modulet skal bruge, herunder CPHA, CPOL samt mulighed for at dividere clocken med 2, 4 og 8 – som tidligere beskrevet i Afsnit 7.2.1. For at lave denne opsætning skal parameterne, placeres bitvis korrekt i pakken defineret på Figur 7.16

spiInit funktionen er blevet implementeret på følgendevis:

```
1  #define SPI_CLKDIV2 0x20000
2  #define SPI_CLKDIV4 0x40000
3  #define SPI_CLKDIV8 0x80000
4  #define SPI_MODE0 0x0
5  #define SPI_MODE1 0x4000
6  #define SPI_MODE2 0x8000
7  #define SPI_MODE3 0xC000
8  u32 = spiCond;
9
10 void spiInit(u32 initCond)
11 {
12   XIOModule_DiscreteWrite(&spiOut, SPI_CH, initCond);
13   spiCond = initCond;
14 }
```

Som det fremgår af koden, tager funktionen `spiInit` kun imod en 32 bit unsigned variabel. Til gengæld er der en række defines, som skal `OR`'s sammen, for at konstruerer opsætningen til SPI-modulet. Dvs at et typisk funktionskald kunne se ud på følgende måde: `spiInit(SPI_CLKDIV2|SPI_MODE2)`.

spiSend() Denne funktion skal sende 8 bit ud til registeret, samt starte SPI-overførslen. Funktionen tager således kun imod et argument, nemlig datapakken, som er en 8 bit unsigned integer. Igen skal det huskes at, datapakken skal indsættes det korrekte sted i kontrolregisteret, som følge af definitionen på Figur 7.16.

Funktionen `spiSend()` implementeres således, at funktionen ikke afsluttes før BUSY-benet fra hardware-modulet går lavt. På denne vis sikres at der ikke overskrives datapakker eller læses datapakker før modtaget data er tilstede.

`spiSend` er implementeret som følgende:

```
1  #define SPI_SEND 0x1
2  #define SPI_CH 1
3  #define SPI_BUSY_PIN 4
4  #define SPI_BUSY_MSK 0b0001000
5
6  int spiSend(u32 slave, u32 data)
7  {
8    //Shift bits to correct location.
9    slave = slave << 1;
10   data = data << 6;
11   //Send to SPI module
12   XIOModule_DiscreteWrite(&spiOut, SPI_CH, spiCond|slave|data|SPI_SEND);
13   while(!((XIOModule_DiscreteRead(&spiIn,SPI_CH)|SPI_BUSY_MSK)>>(SPI_BUSY_PIN-1)));
14   //Restore the initial conditions
15   XIOModule_DiscreteWrite(&spiOut, SPI_CH, spiCond);
16 }
```

Listing 7.15: spiSend()

Igen `OR`'s de forskellige opsætninger for at samle konfigurationen, og til sidst resettes der til den oprindelige opsætning, så SPI-modulet ikke bliver ved med at sende det samme data ud.

spiRecv() `spiRecv`, skal læse den data, som der er blevet læst af hardware-modulet. Dette er gøres ved at læse 8 bit fra driverregister, hvilket er defineret med `SPI_CH`. Funktionen retunerer en unsigned 8 bit integer. Denne er implementeret som vist i det følgende:

```
1  u8 spiRecv()
2  {
3    u32 dataReg;
4    dataReg = XIOModule_DiscreteRead(&spiIn, SPI_CH);
5    return (dataReg & 0b11111111);
6  }
```

Listing 7.16: spiRecv()

`dataReg` `AND`'s med et bitmønster, således at det matcher de første 8 bit i GPI-register fra MicroBlaze MCU'en, som er 32 bit og det retuneres derefter.

spiTran() Denne funktion, implementerer en samlet pakke, der bliver både skrevet og læst fra SPI modulet, funktionen indeholder blot funktionskald til `spiInit`, `spiSend` og `spiRecv`.

```
1  u8 spiTran(int setup, int slave , int data, int delay)
2  {
3    spiInit(setup);
4    spiSend(slave, data);
5    sleep(delay);
6    return spiRecv();
7  }
```

Listing 7.17: spiTran()

Kommentarer

Som det fremgår af kodeeksemplerne, er der ikke nogen form for fejlhåndtering i softwaren. Derfor skal programmøren være varsom med parametrene som sendes til de forskellige funktioner, således der ikke sker et overflow.

7.2.12 Implementering af hardware og software

Dette afsnit beskriver implementering af SPI-controller og SPI-master. Beskrivelsen omhandler de i denne forbindelse uforudsete hastighedsproblemer og løsningen på disse.
Softwarebiblioteket og SPI VHDL modulet, er designet sideløbende. Derfor har det ikke været muligt, at lave løbende tests af grænsefladen mellem disse moduler. I implementeringsfasen har det vist sig, at der er lavet nogle grundlæggende fejl i forbindelse med timingen. Det blev antaget, at mikrocontrolleren kunne "slukke" for SPI hardwaren, før modulet kunne nå at sende adskillige pakker. I praksis kunne softwaren ikke følge med hastigheden på hardwaren. Derfor blev der afsendt flere af samme pakke for hver funktionskald.

Løsningsstrategi

Som løsningsstrategi er der valgt at undersøge timingen mellem hardware og software, ved at udføre en række af tests. I det følgende beskrives to mulige løsninger på denne udfordring.

- Underclocking af SPI-hardware, således at mikrocontrolleren kan følge med.
- Implementering af latch, for at give controlleren total kontrol over overførselsprocessens dataflow.

Valg af testmetoder

For at lave en test, af dataoverførslen mellem software- og hardwaremodulet, er det valgt at teste denne ved at overføre i dette tilfælde fire pakker fra controlleren gennem masteren videre til en slave. Slaven er i dette tilfælde en Seeduino ATmega1280. Datapakkerne som Seeduinoen modtager overvåges over en RS-232 terminal. Datapakkerne skal herefter sammenlignes med det ønskede format. Dette gøres for at verificere, at timingen mellem software og hardware er konsistent. Kommandobeskederne fra controlleren, er som beskrevet under udviklings afsnittet, Afsnit 7.2.3, opdelt i 21 bit. Denne test er udført ved at sende en pakke ad gangen fra controlleren, gentaget fire gange i træk. Det ønskede output er disse fire pakker sammenkædet direkte, således at de kan anvendes som kommando til flash hukommelsen beskrevet i Afsnit 7.3.1.

Der testes med følgende fire pakker fra controller:

- 255
- 2
- 100
- 4

```
1 255 2 100 4 255 2 100 4 255 2 100 4 255 2 100 4
2 255 2 100 4 255 2 100 4 255 2 100 4 255 2 100 4
3 255 2 100 4 255 2 100 4 255 2 100 4 255 2 100 4
4 255 2 100 4 255 2 100 4 255 2 100 4 255 2 100 4
```

Listing 7.18: Ønsket output.

Test 1

Resultat: Det ses at den ønskede overførsel af fire pakker resulterer i gentagne pakker af samme type. Dette ses i Listing 7.19.

```
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100 4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4
4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   255 255 255 255 255
```

Listing 7.19: Testdata fra Arduino. Det ses at der overføres gentagne pakker efter hinanden og antallet er ikke konstant. Derfor konstateres det at timing mellem hardware software ikke er konstant.

Testkonklusion: Det kan konstateres at de enkelte pakkers format bliver overført korrekt. Ud fra testdata ses at hardwaren arbejder med en højere hastighed end software-modulet. Derfor besluttes det at nedsætte hastigheden på SPI-hardwaren for at undersøge om der findes en fast tidsgrænse i denne sammenhæng. Dette gøres ved at implementere en clock divider i hardwaren og koble clock output fra denne op på inputtet til clocken på SPI-driver og SPI-master.

```
entity clkdivider is
    generic (divideby : natural := 2);
    Port ( clk : in std_logic;
           slow_pulse : out std_logic
       );
end clkdivider;

architecture Behavioral of clkdivider is
signal cnt : natural range 0 to divideby-1;
signal toggle : std_logic := '0';
begin

process(clk)
begin
  if rising_edge(clk) then
    if (cnt = (divideby-1)/2)  then
      toggle <= not toggle;
      cnt <= 0;
    else
      cnt <= cnt+1;
    end if;
  end if;
end process;
slow_pulse <= toggle;
end Behavioral;
```

Listing 7.20: Den implementerede clock divider anvendt til at nedsætte SPI-hastigheden.

Test 2 – Nedskalering af hardwarens hastighed

Resultat: Det kan konstateres at nedskaleringen fungerer således at der kun overføres en pakke ad gangen som ønsket, men dog konstateres det at disse pakker nu er efterfulgt af gentagne pakker indeholdende nuller.

```
0   0   255 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   2   0   0   0   0   0   100
0   0   0   0   0   0   4   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   255
0   0   0   0   0   0   0   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
100 0   0   0   0   0   0   0   4   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Listing 7.21: Testdata fra Arduino med nedskalering af hardwarens clock-hastighed.

Testkonklusion: Der er stadig et timing problem.

Test 3 – Opskalering af softwarens afviklingstid

Denne situation undersøges nærmere ved først at udkommentere nogle prints i softwaren. På denne måde afvikles softwaren hurtigere.

Resultat: Dette resulterer i flere nuller mellem pakkerne. Derfor besluttes at nedsætte hardware-modulet til forskellige hastigheder og giver følgende resultater.

Testkonklusion: Der er stadig et timingproblem.

Test 4 – decrementering af hardwarens clock-hastighed

Det ønskes at undersøge om der er en liniær sammenhæng mellem clock hastighed og antal af nuller mellem pakkerne. Dette undersøges nærmere ved at nedsætte clock-hastigheden på hardwaren som vist i Listing 7.1.

Clock delt med	Antal nuller mellem pakker
10	1000
30	600
40	260
42	1300
50	800
60	700
2	0

Tabel 7.1: Test data for varieret clock-hastighed. Der er ikke en direkte liniær sammenhæng mellem hastighed og antal nuller. Bemærk at en nedskalering med 2 giver det ønskede pakkeformat. Dette betragtes som en tilfældighed som kun virker specifikt under de opstillede omstændigheder.

Resultat:

Testkonklusion: Ud fra testen konluderes det at der må findes en anden løsning på timing-problemet end hidtil planlagt. Denne løsning er at implementere hukommelse, i forbindelse med signalering mellem controller og driver. Dette gøres således at det er totalt overladt til controller, hvornår hver transaktion gennemføres. Controller får mulighed for at udføre et vilkårligt antal operationer, imellem transaktionerne af datapakker.

7.2.13 Indførelse af SR-latch

Først bygges en SR-latch som ses i Listing 7.22. Denne latch skal skal overlade data-flowet til controlleren. Masteren skal igennem latchen meddele controlleren at pakken er færdig. Derefter skal masteren overføre når controlleren giver kommando. Dette gøres ved at definere præcis hvornår en pakke er overført og på dette tidspunkt sende en puls ind på set-benet af latchen og afbryde videre overførsel til slaven. Kodeændringer er vist i Listing 7.23 og 7.24.

```
1 ENTITY sr_latch IS
2   PORT (S      : in std_logic;
3         R      : in std_logic;
4         Q      : out std_logic;
5         Q_not  : out std_logic);
```

```
END sr_latch;

architecture behavioral of sr_latch is
  signal Q_nor_S     : std_logic :='0';
  signal Q_not_nor_R : std_logic :='1';

begin
  Q_nor_S     <= R nor Q_not_nor_R;
  Q           <= Q_nor_S;
  Q_not_nor_R <= S nor Q_nor_S;
  Q_not       <= Q_not_nor_R;
end behavioral;
```

Listing 7.22: Den implementerede SR-latch.

I master-filen, designes et signal som sender en puls, netop på tidspunktet hvor en pakke er færdig. Denne puls kaldes package_end_pulse og designet kan ses i Listing 7.23

```
          to_latch_in_S : OUT STD_LOGIC;

...
          to_latch_in_S <= package_end_pulse;
...

WHEN ready =>
          busy              <= '0';             --clock out not busy signal
          ss_n              <= (OTHERS => '1'); --set all slave select outputs high
          mosi              <= 'Z';             --set mosi output high impedance
          continue          <= '0';             --clear continue flag
          package_end_pulse <= '0';             -- for sr_latch

...

            IF((clk_toggles = d_width*2 + 1) AND cont = '0') THEN
              busy              <= '0';             --clock out not busy signal
              ss_n              <= (OTHERS => '1'); --set all slave selects high
              mosi              <= 'Z';             --set mosi output high impedance
              rx_data           <= rx_buffer;       --clock out received data to output port
              package_end_pulse <= '1';             --for sr_latch
              state             <= ready;           --return to ready state
            ELSE                                    --not end of transaction
              state <= execute;                     --remain in execute state
            END IF;
```

Listing 7.23: Det implementerede puls-signal som signalerer at en pakke er overført.

Dette puls-signal routes ud i driveren, hvor det implementeres med SR-latchen og dermed også med controlleren gennem denne. Dette ses i Listing 7.24.

```
  SIGNAL set_latch_with_end_of_package : STD_LOGIC;
  SIGNAL latch_out                     : STD_LOGIC:='0';

...

  latch: entity work.sr_latch(behavioral)
  port map( S     => set_latch_with_end_of_package,
            R     => ctrl_reg(22),--switch_R
            Q     => latch_out,
            Q_not => open
            );

...

```

```
15   master: entity work.spi_master(logic)
16   generic map (d_width   => d_width,           -- data width of master is controlled from this file
17                slaves  => slaves )            -- number of spi slaves is controlled from this file
18   port map(    busy    => busy,
19                clock   => clk,
20                reset_n => reset_n,
21                enable  => enable,
22                cont    => ctrl_reg(20),
23                rx_data => rx_data,
24                --interface from spi_driver spi_master.vhdl (ctrl_reg)
25                clk_div => CONV_INTEGER(ctrl_reg(19 downto 16)),
26                cpol    => ctrl_reg(15),
27                cpha    => ctrl_reg(14),
28                tx_data => ctrl_reg(13 downto 6),
29                addr    => CONV_INTEGER(ctrl_reg(5 downto 1)),
30                -- interface to slaves
31                sclk    => sclk_to_out,
32                mosi    => mosi,
33                miso    => miso,
34                ss_n    => slaveselect,
35                to_latch_in_S => set_latch_with_end_of_package
36            );
37
38 ...
39
40   process( startbit, busy, latch_out )
41     begin
42    reset_n<='1';
43       if  startbit='1' and busy='0' and latch_out ='0' then  --check for ready signal from master, before intializing
              transaction
44         enable<='1';
45         else
46           enable<='0';
47           -- evt reset_n <='0';
48       end if ;
49   end process;
```

Listing 7.24: Det implementerede puls-signal som signalerer at en pakke er overført. Bemærk at output på latch nu er bestemmende for om en transaktion kan begynde.

Test 5 – Simulering af ctrl_reg med ny latch i hardwaren

Der opstilles en testbench hvor controller-registeret er simuleret. Dette gøres for at undersøge om der bliver overført en pakke, hvorefter masteren stopper og venter på reset signal fra controlleren.

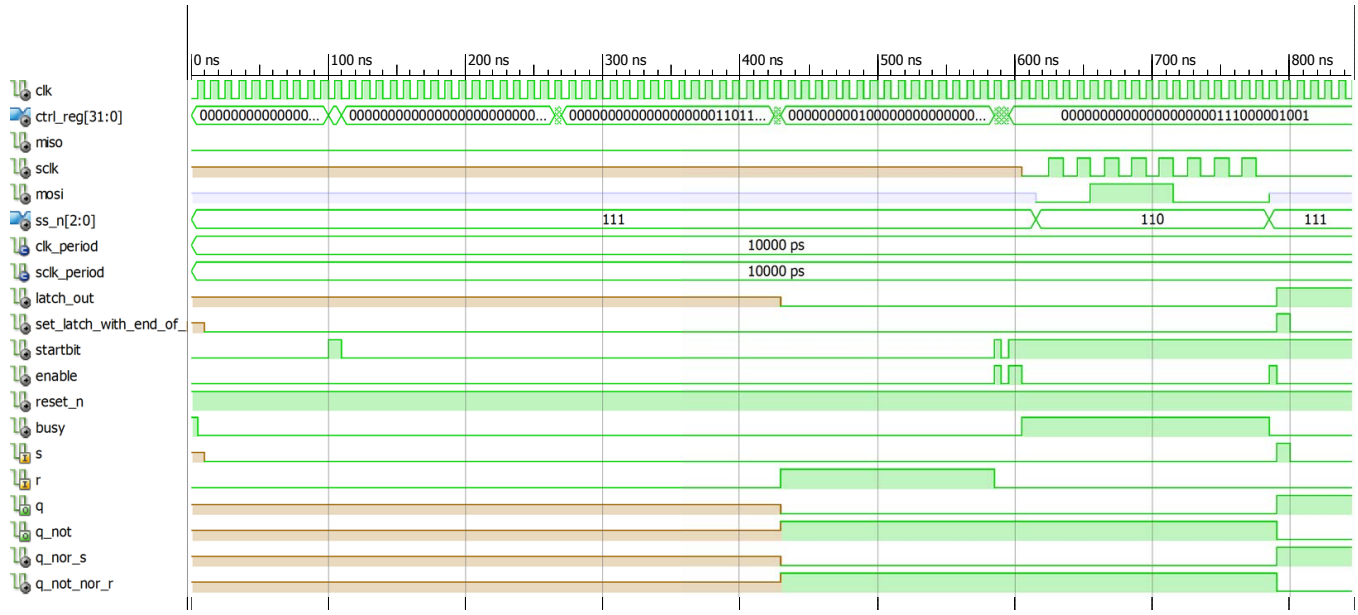
Resultat: Data-flowet er som ønsket. En pakke bliver overført, hvorefter controller bestemmer hvornår en ny pakke skal afsendes

Testkonklusion: Der er nu optimal timing mellem software og hardware. Software-controlleren skal blot starte processen ved at sende et reset signal til SR-latchen, hver gang en pakke ønskes overført.

Test 6 – Implementering af software og hardware med ny latch

Her udføres en implementeringstest af SPI-modulet som helhed. Denne test er udført uden Arduino, da Arduinoen er for langsom. Testen udføres ved maksimal hastighed, som er givet af software-controllerens hastighed. Der testes med følgende fire pakker fra controlleren:

- 255

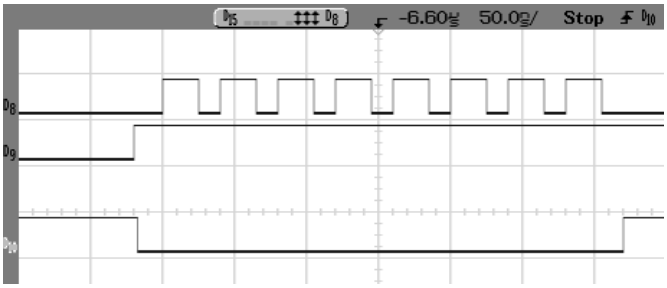


Figur 7.20: Dette timing diagram viser at starbit nu ikke kan aktivere en overførsel uden at outputtet på latchen er gået lav først. Dette signal skal sættes lavt af controller og højt af master-modulet

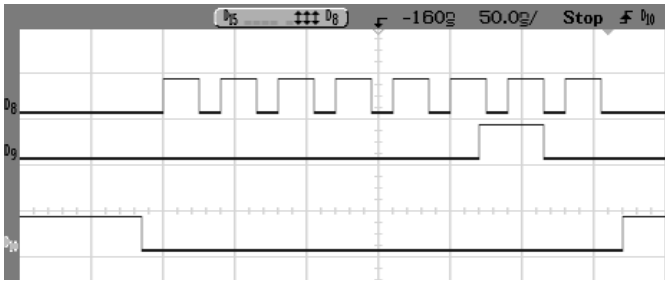
- 2
- 100
- 4

Resultat: Data-flowet er som ønsket. En pakke bliver overført, hvorefter controller bestemmer hvornår en ny pakke skal afsendes. Alle pakker bliver modtaget som forventet. Disse pakker kan ses på Figur 7.21.

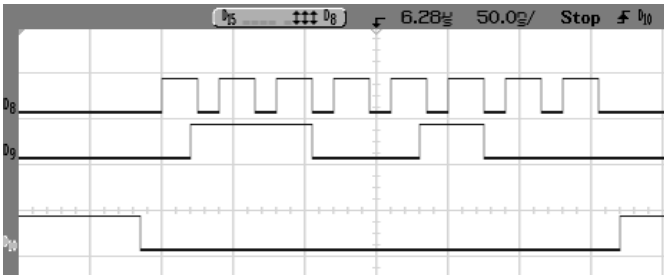
Testkonklusion: Denne løsning med SR-latch, hvor det asynkrone dataflow styres af controlleren, anvendes i dette projekt, mellem software-controller og hardware modul. Bemærk at der er mulighed for at designe et SPI-system, hvor continue mode anvendes, som er beskrevet i Afsnit 7.2.4



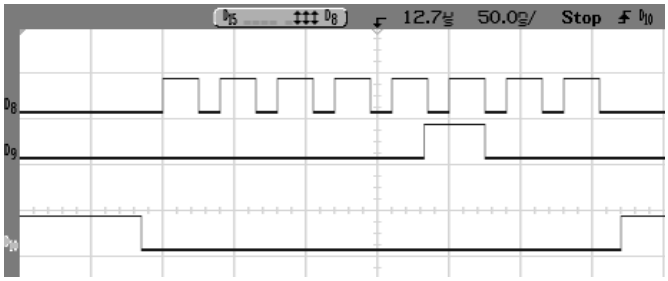
(a) Pakke 1 (255)



(b) Pakke 2 (2)

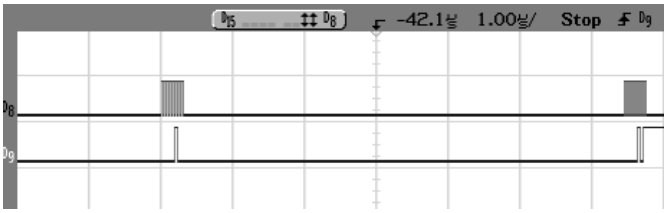


(c) Pakke 3 (100)

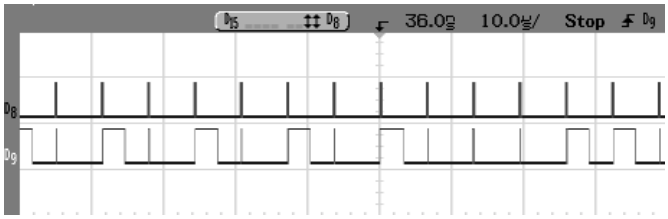


(d) Pakke 4 (4)

Figur 7.21: Data fra logic analyzer, som viser succesfuld afsending af fire pakker. Apparatet er indtillet til 50 ns pr. tern. D8 viser SPI-clock. D9 viser mosi ben. D10 viser slaveselect



(a) Afstand mellem to pakker (1 us pr. tern)



(b) Den konstante afstand mellem pakkerne (10 us pr. tern)

Figur 7.22: Data fra logic analyser som viser den tidsmæssige afstand mellem pakkerne.

```
1  #include <stdio.h>
2  #include "platform.h"
3  #include "xparameters.h"
4  #include "xbasic_types.h"
5  #include "xiomodule.h"
6  //Settings
7  #define SPI_CONT 0x100000
8
9  #define SPI_CLKDIV0 0x0
10 #define SPI_CLKDIV2 0x20000
11 #define SPI_CLKDIV4 0x40000
12 #define SPI_CLKDIV8 0x80000
13
14 #define SPI_MODE0 0x0
15 #define SPI_MODE1 0x4000
16 #define SPI_MODE2 0x8000
17 #define SPI_MODE3 0xC000
18 #define SPI_SEND 0x1
```

```
19
20 #define SPI_CH 1
21
22 #define SPI_BUSY_PIN 9
23 #define SPI_BUSY_MSK 0b100000000
24
25 #define SPI_DATA_LATCH_PIN 9 // Kobles p? pin 8 GPI1
26 #define SPI_DATA_LATCH_MSK 0b100000000
27 //Globals vars...
28 XIOModule spiIn;
29 XIOModule spiOut;
30 u32 spiCond;
31
32 //Init spi
33 int spiInit(u32 initCond){
34   XIOModule_DiscreteWrite(&spiOut, SPI_CH, initCond);
35   spiCond = initCond;
36   putnum(initCond);
37  return 0;
38 }
39
40 //Set data and the begin transfer signal
41 //The function will only send single packet of 1 byte.
42 int spiSend(u32 slave, u32 data){
43   //Shift bits to correct location.
44   slave = slave << 1;
45   data = data << 6;
46   //u8 i;
47   //Send to SPI module
48   XIOModule_DiscreteWrite(&spiOut, SPI_CH, spiCond|slave|data|SPI_SEND);
49   print("\nSend data with startbit high: ");
50   putnum(spiCond|slave|data|SPI_SEND);
51   XIOModule_DiscreteWrite(&spiOut, SPI_CH, 0xf);
52
53   // debugging
54   u32 dataIn;
55   print("\n 1. dataIn (SPI_CH): ");
56   dataIn = XIOModule_DiscreteRead(&spiIn, SPI_CH);
57   putnum(dataIn);
58   print("\n 2. dataIn (dataIn&SPI_BUSY_MSK): ");
59   dataIn = dataIn&SPI_BUSY_MSK;
60   putnum(dataIn);
61   print("\n 3. dataIn(dataIn >> (SPI_BUSY_PIN-1)): ");
62   dataIn = dataIn >> (SPI_BUSY_PIN-1);
63   putnum(dataIn);
64   //
65 while(((XIOModule_DiscreteRead(&spiIn, SPI_CH)&SPI_BUSY_MSK)>>(SPI_BUSY_PIN-1))){
66   print("\nDetecting if master is ready for new data or datapins to be reset\n");
67 }
68   //Restore the initial conditions
69   print("\nResetting to initial conditions\n");
70   XIOModule_DiscreteWrite(&spiOut, SPI_CH, spiCond);
71
72 }
73
74 //MAIN TEST FUNCTION//
75 ////////////////////
76
77     void print(char *str);
78     int main()
79     {
80       init_platform();
81       //print("\n\nSending 255 to slave 1 \n");
82       print("\nInitializing platform\n\r");
83
```

```
84          XIOModule_Initialize(&spiIn, XPAR_IOMODULE_0_DEVICE_ID);
85          XIOModule_Start(&spiIn);
86          XIOModule_Initialize(&spiOut, XPAR_IOMODULE_0_DEVICE_ID);
87          XIOModule_Start(&spiOut);
88
89       print("\nInit spi: ");
90       spiInit(SPI_CLKDIV8|SPI_MODE3);
91
92  while(1){
93         //spiSendLargeTest(1, 0xE700E700, 4);
94         spiSend(1,255);
95         spiSend(1,2);
96         spiSend(1,100);
97         spiSend(1,4);
98          }
99
100      cleanup_platform();
101      return 0;
102    }
```

Listing 7.25: Test software anvendt til implementeringen af software og hardware.

```
1   #include <avr/interrupt.h>
2   #include <avr/io.h>
3   #include "pins_arduino.h"
4
5   #define SDRDY 9 // slave data ready
6
7   byte t = 0;
8   byte r = 0;
9
10  void setup() {
11     SPI_SlaveInit();
12     pinMode(9, OUTPUT);
13     digitalWrite(9, LOW);
14     Serial.begin(9600);
15  }
16
17  void loop() {
18    r = SPI_SlaveReceive();
19    Serial.print(r);
20    Serial.print(' ');
21
22
23    //t = SPI_SlaveTransmit(r+1);
24  }
25
26  void SPI_SlaveInit(void)
27  {
28     pinMode(MISO, OUTPUT);
29     digitalWrite(MISO, LOW);
30     SPCR |= _BV(SPE);
31  }
32
33  char SPI_SlaveReceive(void)
34  {
35    /* Wait for reception complete */
36    while(!(SPSR & (1<<SPIF)))
37      ;
38    /* Return Data Register */
39    return SPDR;
40  }
41
42  char SPI_SlaveTransmit(byte cData)
43  {
```

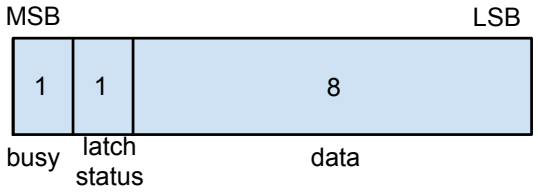
```
44    /* Start transmission */
45    SPDR = cData;
46    // assert drdy signal(to master)
47    digitalWrite(9, HIGH);
48
49    /* Wait for transmission complete */
50    while(!(SPSR & (1<<SPIF)))
51    ;
52
53    // deassert drdy signal
54    digitalWrite(9, LOW);
55
56    return SPDR;
57  }
```

Listing 7.26: Arduino software som er anvendt til at teste det samlede SPI-modul mod.

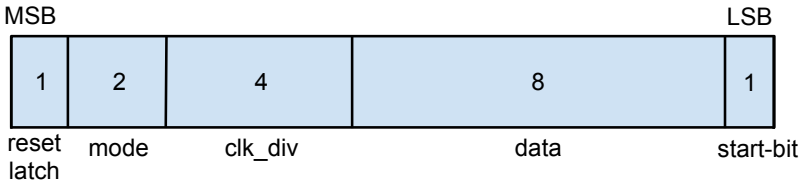
7.2.14 Opsummering af hardwareændringer

I forbindelse med implementeringen af software og hardware, blev der foretaget følgende ændringer i forhold til det tidligere design.

- Kontrolregisteret er ændret hvilket ses på Figur 7.24.
- Driverregisteret er ændret, hvilket ses på Figur 7.23.
- Dataflowet er ændret med indførelsen af SR-latch. Denne implementering er beskrevet i Afsnit 7.2.13
- Slaven skal hardwires eksternt i forhold til SPI-modulet. Dermed vælges den fra det respektive software.



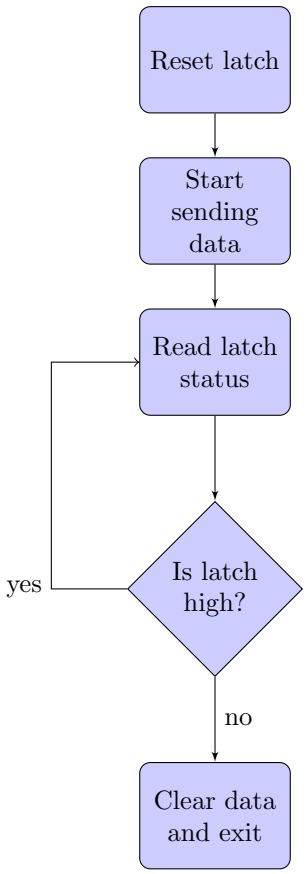
Figur 7.23: Formatet på driverregisteret.



Figur 7.24: Formatet på kontrolregisteret.

7.2.15 Opsummering af softwareændringer

Som beskrevet i Afsnit 7.2.14, var der problemer med timingen, hvilket løses med en SR-latch. Denne latch skal styres fra softwaren, således at controlleren kan resette latchens output. Ydermere skal softwaren være i stand til



Figur 7.25: Flow chart over ny send-funktion

at kunne læse status på latchen, altså om den er høj eller lav. Implementeringen af latch-funktionaliteten, gør at der skal ændres i send-funktionen, således at flowet bliver som vist på Figur 7.25. Et andet valg, som blev truffet er at lade mikrocontrolleren vælge slaven, og ikke selve SPI-modulet, hvilket kan ses som kode i Listing 7.2.15.

```
1  static int send(u32 data)
2  {
3      data = data << 6;
4      XIOModule_DiscreteWrite(&spiOut, SPI_CH, SPI_RESET_LATCH);
5      XIOModule_DiscreteWrite(&spiOut, SPI_CH, 0x0);
6      XIOModule_DiscreteWrite(&spiOut, SPI_CH, spiCond|slave|data|SPI_SEND);
7      while(!((XIOModule_DiscreteRead(&spiIn, SPI_CH)&SPI_LATCH_MSK)>>(SPI_LATCH_PIN-1)));
8      XIOModule_DiscreteWrite(&spiOut, SPI_CH, 0x0);
9      return 0;
10 }
```

Som det fremgår af både flowchart og koden, så resettes latchen først, derefter sættes data på de interne output pins, hvorefter der ventes på at latchen går høj, hvilket vil sige at SPI-modulet har overført en pakke. Derefter nulstilles data på de interne output pins og latchen forbliver høj, indtil næste pakke sendes. Dette løste timing-problemet mellem hardware og softwaren. Der kan nu sendes enkelte pakker uden noget problem. Det færdige SPI-bibliotek kan ses på bilags CD'en

- `code/spiLibMicroBlaze` .

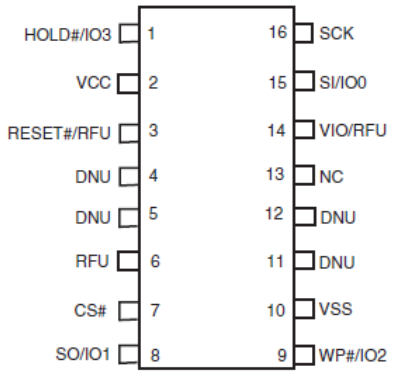
Der er foretaget nogle mindre væsentlige ændringer i koden. Disse er blot funktioners navne og andre ikke funktionelle ændringer.

7.3 Flash

Hukommelsesmodulet der anvendes til dette projekts LOG-system er en Spansion S25FL128S. Ud fra krav 14 ses det at der kræves en størrelse på minimum 16 MiB, så det er derfor valgt at bruge en 16 MiB flash. Modulet kan fås i en WSON , BGA-24 ogSOIC-16 pakke, sidst nævnte kan ses på Figur 7.26, og har følgende karakteristiske specifikationer[12]:

Størrelse	16 MiB
Bus interface	SPI
Max. clockfrekvens	133 MHz
V_{cc}	2,7 V 3,6 V
Pakke	SOIC-16

Tabel 7.2: S25FL128S specifikationer.[12]



Figur 7.26: S25FL128S SOIC-16 pakke.[12]

7.3.1 Hardware interface

En uddybende forklaring af de vigtigste funktioner på hukommelseschippen findes i Tabel 7.3.

Som det ses på Tabel 7.3 og Figur 7.26 har nogle af modulets ben flere funktioner. Det er bl.a. fordi , at flashmodulet har mulighed for at operere i Single SPI, Dual SPI og Quad SPI mode.

Single SPI, Dual SPI og Quad SPI Single-, Dual- og Quad SPI er en databus-mulighed, der tillader overførsel af en-, to-, eller fire bit ad gangen, mellem SPI masteren og flashmodulet. Se Figur 7.27, som viser de 3 forskellige opkoblingsmetoder.

Ved Single SPI benyttes SO og SI på henholdsvis ben 8 og 15, hvor der modtages kommandoer og data fra masteren via SI og sendes data tilbage til masteren med SO. Alle kommandoer begynder med en opcode på en byte, der vælger hvilken type operation der skal foretages i flashmodulet. Denne er som regel efterfulgt af en adresse og derefter data, hvis der skal overføres data til en bestemt adresse i flashen.[12]

Benyttes der derimod Dual SPI bliver SI og SO brugt som IO0 og IO1. IO0 og IO1 er begge input/output porte, som beskrevet, overføres med 2 bit. Ved Dual SPI overføres adressen ligesom ved single SPI, under

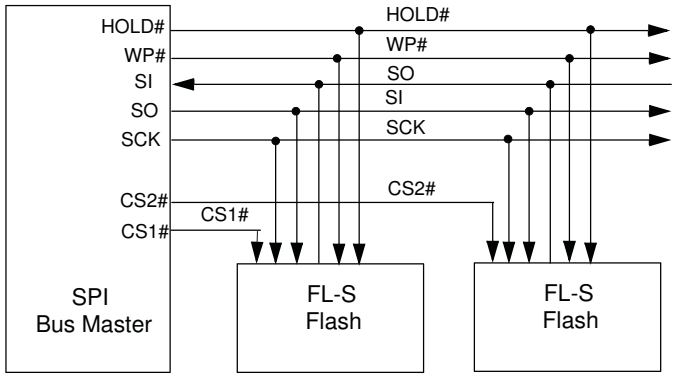
Ben nr.	Forkortelse	Betydning
1, 8, 9 og 15	IO3, IO1, IO2 og IO0	Benyttes som input/output hvis der benyttes Dual SPI eller Quad SPI.
3, 4, 5, 6, 11, 12, 13 og 14	DNU, RFU og NC	Står for henholdsvis "Do Not Use", "Reserved for Future Use" og "Not Connected". Disse ben kan ikke benyttes.
7	CS#	"Chip Select" skal sættes lav for at der kan skrives instruktioner til flashen.
15	SI	"Serial In" bruges når der benyttes almindelig Single SPI, som serial in.
8	SO	"Serial Out" bruges, som "Serial In", når der benyttes almindelig Single SPI, som serial out.
16	SCK	"Serial Clock" er et input signal der benyttes som en clock til SPI interfacet.
10	V_{ss}	Modulets GND ben.
9	WP#	"Write Protect" tillader at der kan beskyttes sektorer inde i flashen fra at blive overskrevet.
1	HOLD#	Benyttes til at sætte seriel kommunikation til modulet på pause.
3	RESET#	Dette ben er en hardware metode til at resette modulet, dog uden at slette data.

Tabel 7.3: Ben beskrivelse

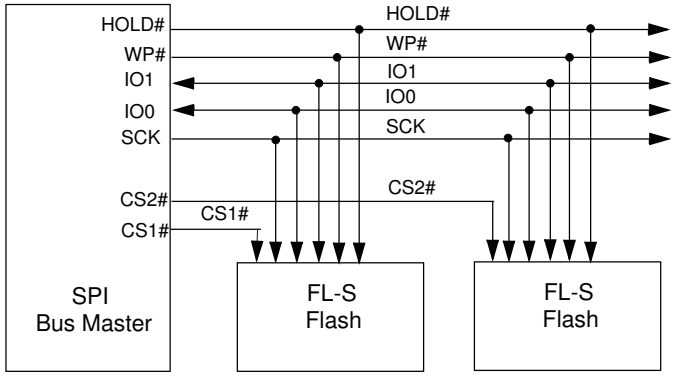
kommunikationen fra master til slave (Dual SPI output kommunikation), hvilket vil sige at der kun overføres en adresse på SI signalet. Dataet derimod overføres i bit-par på IO0 og IO1. Er det derimod en input/output kommando, benyttes både IO0 og IO1 til at sende adresse og data på i bit-par.[12]

Quad SPI fungerer på samme måde som Dual SPI, hvor den eneste markante forskel er antallet af bits der kan kommunikeres med af gangen. Som beskrevet tidligere tillader Quad SPI master/slave kommunikation med 4 bit ad gangen. Det vil sige at der nu benyttes flere signaler; IO0, IO1, IO2 og IO3. Måden at sende data frem og tilbage på er den samme som forklaret ved Dual SPI. [12]

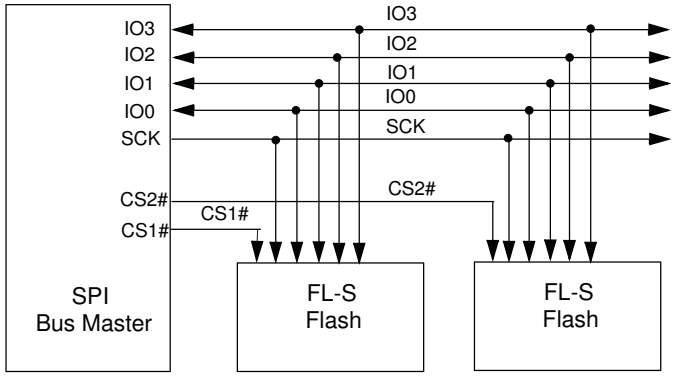
Timingdiagrammer For at få et bedre overblik over hvad forskellen er på de 3 kommunikationsformer, kan det være en fordel at kigge på de forskellige timingdiagrammer, der hører til hver form for kommunikation. En ting som er generel for alle timingdiagrammer, er måden hvorpå Chip Select skal konfigureres. Så snart der ønskes at skrive noget til det pågældende flashmodul, er det nødvendigt at Chip Select bliver sat logisk lav og holdes lav alt den tid der skrives til modulet. Bliver Chip Select ikke sat lav før en instruktion skrives til flashmodulet, læser modulet ikke instruktionen og foretager derfor intet. På Figur 7.28 ses et timingdiagram for Single SPI kommunikation, nærmere bestemt et timingdiagram for en output kommando, hvor der bedes om data fra slaven til master modulet, hvor slaven er flashmodulet og masteren er SPI modulet. Det ses på timingdiagrammet, at Chip Select bliver trukket lav, umiddelbart før der sendes instruktion og adresse til flashmodulet og holdes lav indtil den forespurgte data er afsendt. På Figur 7.29 og 7.30 vises via timingdiagram hvordan en output og I/O kommando ser ud for henholdsvis Dual SPI og Quad SPI, som er forklaret i ovenstående afsnit. [12]



(a) Single SPI blokdiagram

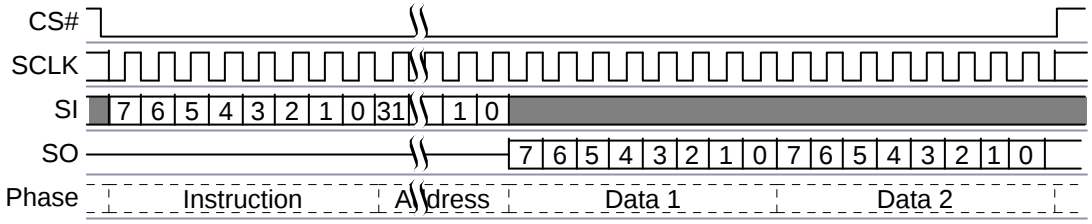


(b) Dual SPI blokdiagram

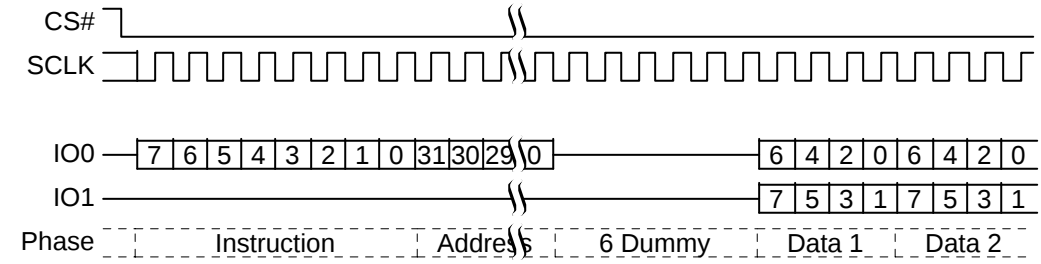


(c) Quad SPI blokdiagram

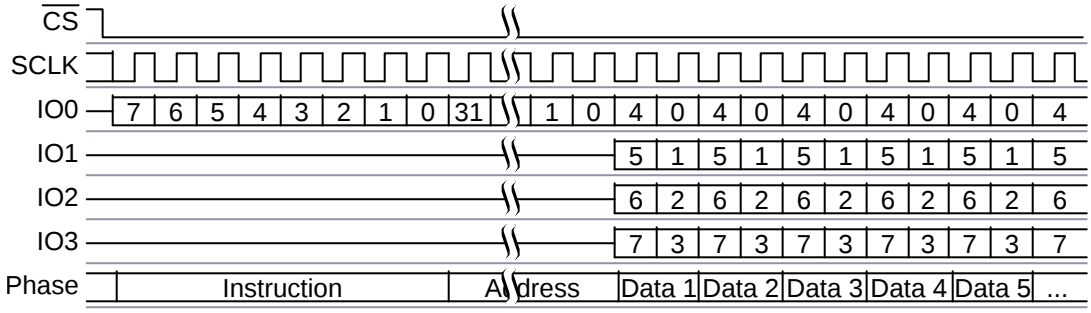
Figur 7.27: Blokdiagrammer for Single-, Dual- og Quad SPI.[12]



Figur 7.28: Enkelt bit I/O kommando.[12]

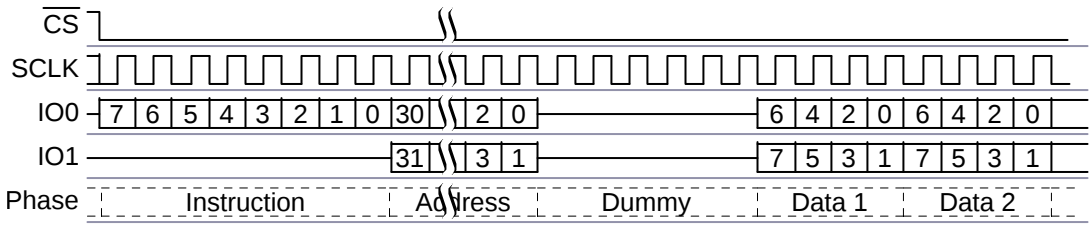


(a) Dual SPI output kommando.

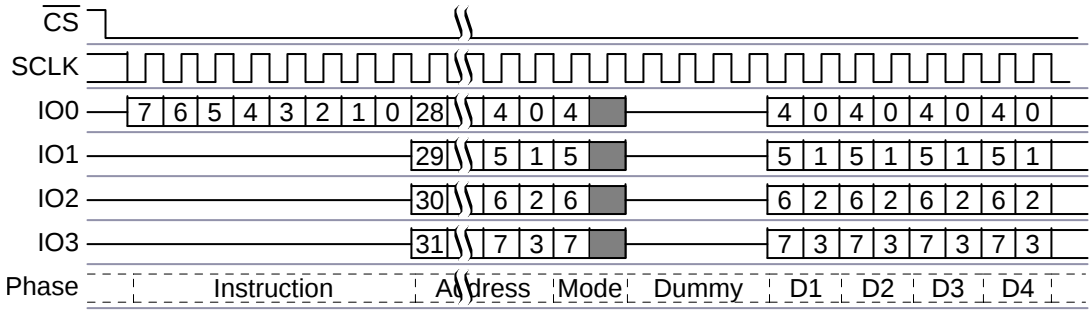


(b) Quad SPI output kommando.

Figur 7.29: Dual og Quad SPI output kommandoer.[12]



(a) Dual SPI I/O kommando.



(b) Quad SPI I/O kommando.

Figur 7.30: Dual- og Quad SPI I/O kommandoer.[12]

7.3.2 Software interface

I dette kapitel beskrives software interfacet til flashmodulet S25FL128S.

Generel beskrivelse af software interface

- Flashmodulet S25FL128S understøtter 32 bit og 24 bit adresser. Ved brug af 24 bit er det muligt at adressere op til 16 MiB, og ved en opløsning på 32 bit er det muligt at adressere op til 4 GiB (dette er dog ikke nødvendigt, da flashen kun er 16 MiB).
- Flash hukommelsesmatricen er delt i sektorer. Disse er organiseret, som en hybrid kombination af 4 KiB og 64 Kibit sektorer, eller som en 256 KiB sektorer, se evt. Figur 7.31. Inddeling af sektorer afhænger af chippens model. Der findes inddeling med 32 sektorer af 4 KiB og 254 sektorer af 64 KiB, eller inddeling med 64 sektorer af 256 KiB.
- Hukommelsesmodulet har 1024 B lang OPT (One Time Program) adresse lager, som er placeret separat fra hukommelsesmatricen. Dette lager er delt i 32 låsbare 32 B lange regioner.
- Flash-hukommelsesmodulet har forskellige registre, som anvendes til at konfigurere, eller angive status for flashmodulet. Det er muligt at få adgang til disse registre ved at anvende de passende opcodes (se Tabel 7.5).

Sector Size (kbyte)	Sector Count	Sector Range	Address Range (Byte Address)	Notes
64	254	SA00	0000000h-000FFFFh	Sector Starting Address — Sector Ending Address
		:	:	
		SA253	00FD0000h-00FDFFFFh	
4	32	SA254	00FE0000h-00FE0FFFh	
		:	:	
		SA285	00FFF000h-00FFFFFFh	

Figur 7.31: Flashmodulets sektor opdeling.[12]

Kommandoer

Kommunikation mellem masteren og S25FL128S modulet foregår over SPI, og der sendes kommandoer alt efter hvilken operation, som skal foretages. Kommando kan have adresse, instruction modifier, latency periode, dataoverførsel til eller fra hukommelse. Hver kommando begynder med en instruktion, som bestemmer hvilken type af information der skal overføres.

Alle kommandoer er struktureret således:

- Kommando begynder med 8 bit instruktionskode.
- Instruktionen kan stå alene, eller kan følges med en adresse, som kan være 24 bit eller 32 bit lang.
- Der kan tilføjes en for mulighed at overføre en, to eller fire bits parallelt af gangen.
- Instruktionen bestemmer hvor bred en overførsel kan være.
- Enkeltstående bits eller parallel-bit-grupper er overført med mindst betydende bit først.
- Nogle instruktioner sender instruction modifier efter adressen, hvilket indikerer at den næste kommando skal være af samme type som den igangværende.

- Afhængig af instruktion kan data undlades efter adressen.
- Der kan sendes dummy-bytes, hvilke kan blive brugt til at ændre forsinkelse fra 0 til flere clock-cykler og lave et bestemt delay.
- Instruktion, adresse, mode og data skal sendes i byte-størrelser. Alle adresser er overført med MSB først. Datapakker med lave adresser, bliver overført før pakker med højere adresser.
- Alle forsøg på at læse hukommelse mens der skrives eller slettes fra hukommelsesmatricen, bliver ignoreret.

Instruktionssæt

Der er forudsat to kommandosæt: en til 32 bit adresser og en til 24 bit adresser. Instruktionerne til disse kommandosæt har samme format, forskellen er blot navngivningen af disse og deres hexadecimale repræsentation. Navne på instruktioner efterfulgt af 4 B adresser, begynder med tallet "4"hvilket ses i Tabel 7.4. Navne på instruktioner efterfulgt af 3 B undlader et 4-tal. Disse kan ses i Tabel 7.5.

Instruktionsnavn	Beskrivelse	Kode (Hex)
4FAST_READ	Læs hurtig	0x0C
4READ	Læs	0x13
4DOR	Læs output med to parallele forbindelser	0x3C
4QOR	Læs output med fire parallele forbindelser	0x6C
4DIOR	Læs I/O med to parallele forbindelser	0xBC
4QIOR	læs I/O med fire parallele forbindelser	0xEC
4DDRFR	Hurtig læsning af DDR I/O med fire parallele forbindelser	0x0E
4DDRDIOR	læs DDR I/O med fire parallele forbindelser	0xBE
4DDRQIOR	læs DDR I/O med fire parallele forbindelser	0xEE
4PP	Page program	0x12
4P4E	Slet 4 KiB parameter	0x21
4SE	Slet 64 KiB eller 256 KiB segment	0xDC

Tabel 7.4: Instruktioner med 4 B adresser.[12]

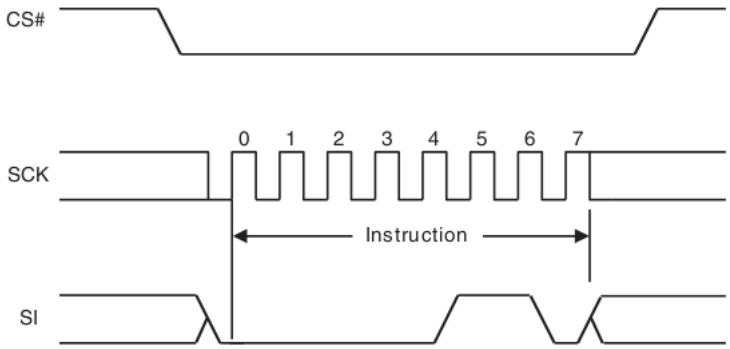
Instruktionsnavn	Beskrivelse	Kode (Hex)
FAST_READ	Læs hurtig	0x0B
READ	Læs	0x03
DOR	Læs output med to parallele forbindelser	0x3B
QOR	Læs output med fire parallele forbindelser	0x6B
DIOR	Læs I/O med to parallele forbindelser	0xBB
QIOR	læs I/O med fire parallele forbindelser	0xEB
DDRFR	Hurtig læsning af DDR I/O med fire parallele forbindelser	0x0D
DDRDIOR	læs DDR I/O med fire parallele forbindelser	0xBD
DDRQIOR	læs DDR I/O med fire parallele forbindelser	0xED
PP	Page program	0x02
P4E	Slet 4 KiB parameter	0x20
SE	Slet 64 KiB eller 256 KiB segment	0xD8

Tabel 7.5: Instruktioner med 3 B adresser.[12]

Det er valgt at kigge på de vigtigste instruktioner til at skrive, læse og slette flashen. Der kan ses, at der kan bruges en, to eller fire SPI data forbindelser. Flashmodulet har en størrelse på 16 MiB – derfor er det passende at anvende instruktioner med 24 bit adresser.

Write enable Write enable kommando (`0x06`) sætter write enable latch bit af status registeret til "1". Write enable kommandoen gør det muligt for flashmodulet at modtage write, program og erase kommandoer. CS# input-benet skal sættes til logisk høj efter de 8 bit af instruktionen latched til serial input-benet.

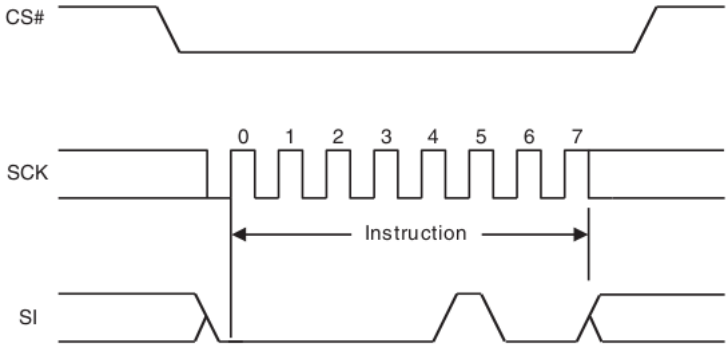
Timingdiagram for write enable kommandoen kan ses på Figur 7.32.



Figur 7.32: Timingdiagram for write enable kommando.[12]

Write disable Write disable kommandoen (`0x04`) sætter write enable latch bit af status registeret til "0". Write disable kommandoen gør det umuligt for flashmodulet at modtage write, program og erase kommandoer. CS# input-benet skal sættes til logisk høj efter de 8 bit af instruktionen latched til serial input ben.

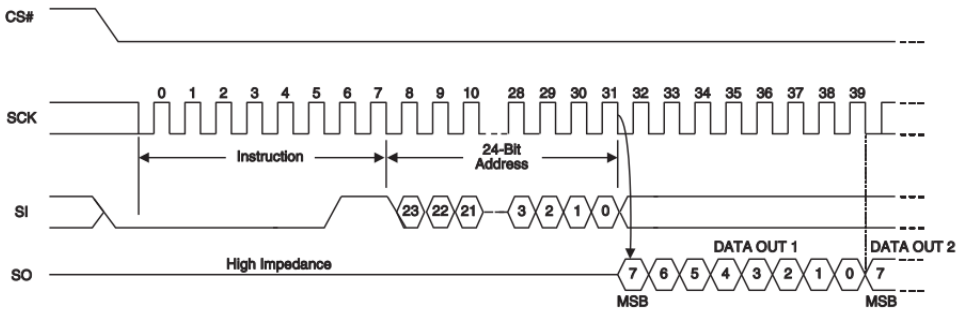
Timingdiagram for write disable kommandoen kan ses på Figur 7.33.



Figur 7.33: Timingdiagram for write disable kommando.[12]

Read Read-kommandoen (`0x03`) flytter hukommelsesindholdet i en bestemt adresse til serial output. Clock-frekvensen må maksimum være på 50 MHz for at udføre en read-kommando. Startadressen kan være en tilfældig adresse i hukommelsen, men efter hver byte er skrevet ud, bliver den næste adresse sat til næste adresse i flashen. Hele hukommelsen kan derfor læses, ved hjælp af en instruktion, som begynder på adressen `0x000000` og læser til data fra den sidste byte, der er skrevet.

Timingdiagram for read kommandoen kan ses på en Figur 7.34.

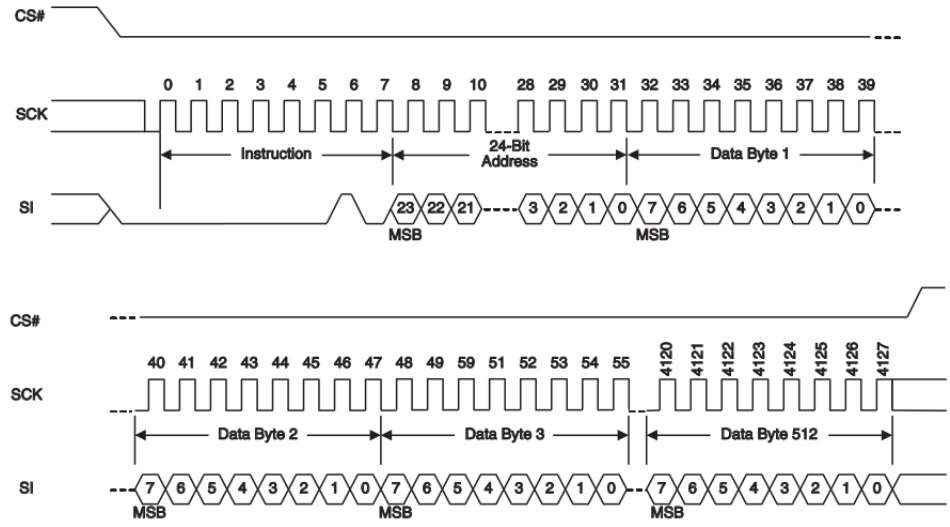


Figur 7.34: Timingdiagram for read kommando.[12]

Page program Page program kommandoen (`0x02`) skifter bittene i hukommelsen fra "1" til "0". Denne kommando kan kun afvikles efter "write enable" kommandoen er afviklet og write enable latchen er sat til logisk høj.

Page størrelse kan være 256 B eller 512 B afhængig af modulets type. Hvis der er sent data som er mindre end en page, bliver alle bytes i data skrevet sekventielt ind i pagen uden at konflikte med hinanden. Hvis der skal skrives data, som er lige så lang som pagen, behøver der kun at skrives en startadresse for pagen.

Timingdiagram for page program kommandoen kan ses på en Figur 7.35.

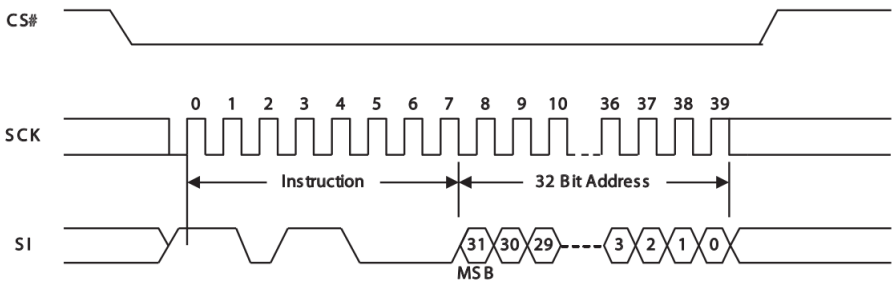


Figur 7.35: Timingdiagram for page program kommando.[12]

Sector erase Sector erase kommandoen (`0xD8`) sætter alle bittene i en bestemt adressesektor til "1". Write enable kommandoen skal køres før sector erase kommando, kan begynde at slette sektorer. CS# skal sættes til logisk høj efter den sidste adressebit er sendt til serial input. Modulets opbygning bestemmer om sektoren, der skal slettes er på 256 KB eller 64 KiB.

Timingdiagram for sector erase kommandoen kan ses på en Figur 7.36.

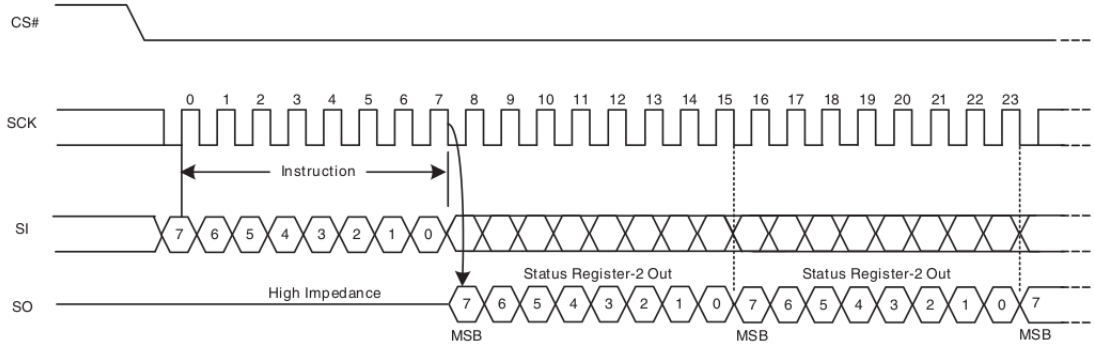
Read status register Read status register kommandoen (`0x05`) læser indholdet af status registret og sender det til serial output. Chip select skal sættes til logisk lav før kommandoen kan køres, og sættes høj



Figur 7.36: Timingdiagram for erase segment kommando.[12]

efter kommandoen er færdig. Read status register kommandoen kan køres på hvilket som helst tidspunkt – også når systemet er i gang med at slette eller programmere. Dette er væsentligt, hvis der er brug for at vide om systemet er i gang med at slette eller programmere i hukommelsesmatricen. Det er muligt at læse status register kontinuerligt, ved at sende et multiplum af otte pulser af klokken for hver byte, som skal læses.

Timingdiagram for read status register kommandoen kan ses på Figur 7.37.



Figur 7.37: Timingdiagram for read status register kommando.[12]

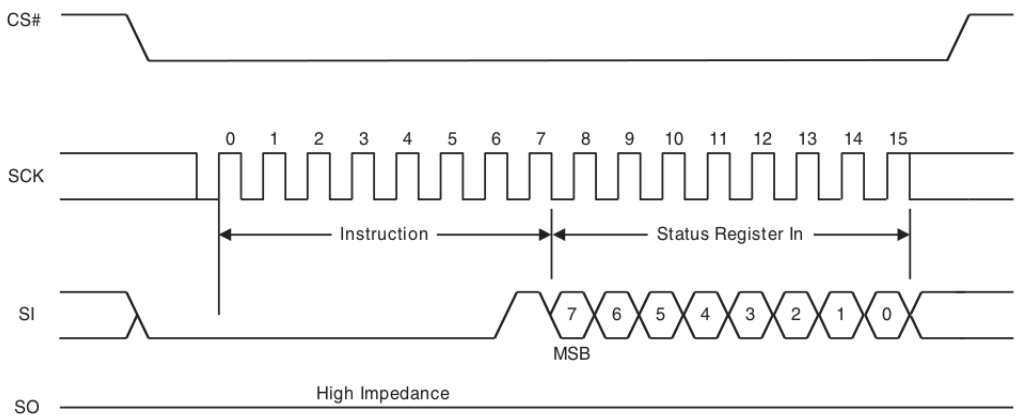
Write registers Write registers kommandoen (`0x01`) har mulighed for at skrive en byte til status- og/eller konfigurationsregister. Denne kommando kan være brugbar, når der skal nulstilles status- og konfigurationregister. Write enable kommandoen skal køres, før systemet accepterer Write registers kommandoen. Det betyder også at det er muligt at sende en kommando i to forskellige formater. Der kan sendes en instruktion og en byte af data, som skal skrives i status register, eller der kan sendes en instruktion og en byte data til status registret og en byte data til konfigurations registret.

Timingdiagram for write registers kommandoen kan ses på Figur 7.38.

Beskrivelse af komponenten

Herunder er en beskrivelse af flashmodulet:

- Flash hukommelsen er på 16 MiB.
- Maksimal hastighed er 133 MHz
- Pakken er 16-pin SOIC.



Figur 7.38: Timingsdiagram for write registers kommando.[12]

- Komponenten kan holde til temperaturforskellen fra $-40\,^{\circ}\mathrm{C}$ til $85\,^{\circ}\mathrm{C}$.
- Komponenten er lavet af blyfri materiale.
- Sektorer er en hybrid af $32 \times 4\,\mathrm{KiB}$ og de resterende sektorer er på 64 KiB, med 256 B programmerbare buffer.

I dette tilfælde er hukommelsesmatricen opbygget i sektorer på 64 KiB og 4 KiB. Sektorer på 64 KiB begynder på adresse `0x000000` og slutter på adresse `0xFDFFFF` , hvor en sektor er `0x00FFFF` adresser lang, så er der i alt er 253 af disse slags sektorer. De udgør 16,65E6 adresser og på hver adresse kan der være en byte data, dette betyder at det er muligt at gemme 16 MiB af data.

Resten af hukommelsesmatricen er blevet brugt til 32 sektorer på 4 KiB, hvilke starter på adresse `0xFE0000` og ender på adresse `0xFFFFFF` , og en sektor er `0x000FFF` lang. De udgør 131,1E3 adresser, dette betyder at det er muligt at gemme 128 KiB data i disse sektorer.

Det er også væsentlig at understrege, at det kun er muligt, som maks at skrive en page (256 B) af gangen til hukommelsesmatricen, men det er muligt at slette en sektor.

Status Register

Status registeret, som kan benyttes i forbindelse med flashmodulet, har flere forskellige funktioner, som kan bruges til brugerens fordel, alt efter hvilken funktion status registeret bliver bedt om at udføre. Status registeret består af 8 bit, som hver i sær har et formål. De forskellige funktioner, som status registeret indeholder, er:

- Status Register Write Disable
- Programming Error Occurred
- Erase Error Occurred
- Block Protection
- Write Enable Latch
- Write In Progress

Nogle af funktionerne kan brugeren selv påvirke, ved at sætte dem til enten 1 eller 0, andre er “read-only”.

For at få et overblik over hvad de forskellige funktioner gør, vil de kort blive forklarede.

Status Register Write Disable Denne funktion ligger på status registerets 7. bit. Hvis denne funktion bliver sat til 1 og WP# bliver sat lav, bliver modulet sat i en hardware mæssig beskyttet tilstand, og bittene i status registeret bliver "read-only". Hvis WP# derimod sættes høj, kan Status Register Write Disable of Block Protection bittene, i status registeret, ændres efter brugerens behov. Sættes WP# lav men SRWD forbliver 0, har WP# ingen effekt, og der oprettes ikke hardware mæssig beskyttet tilstand.[12]

Program Error Denne funktion kan læses på status registerets 6. bit. Læses denne del af status registeret, kan det ses om der er sket en fejl under en programmering eller omprogrammeringen af modulet er afviklet som forventet. Læses der et 1 tal, er der opstået en fejl. Læses der derimod et 0 er programmering forløbet som forventet.[12]

Erase Error Denne funktion kan læses på status registerets 5. bit. Læses denne del af status registeret, kan det ses om der er sket en fejl under en sletning i modulet, eller om sletningen er afviklet som forventet. Læses der et 1 tal, er der opstået en fejl med sletningen. Læses der derimod et 0 er sletningen forløbet som forventet.[12]

Block Protection Denne funktion ligger på status registerets bit 4, 3 og 2. Denne del af status registeret fylder hele 3 bit. Med disse kan der softwaremæssigt bestemmes forskellige dele af flashmodulet, som skal beskyttes imod programmering og sletning. Er en eller flere af disse bits aktiverede kan Bulk Erase kommandoen ikke eksekveres før block protection bittene er blevet slettet til 0 igen.[12]

Write Enable Latch Denne funktion ligger på status registerets 1. bit. For at skrive noget til modulet skal denne bit sættes til 1, for at give en beskyttelse imod skrivning og sletning. For at sætte den til 1, benyttes Write Enable kommandoen. Skal den igen sættes til 0, benyttes Write Disable kommandoen.[12]

Write In Progress Denne funktion kan læses på status registerets 0. bit. Denne bit kan tjekkes for at se om modulet er igang med at udføre en operation. Står der her et 1 tal, er modulet igang med at udføre en operation, og alle andre operation ignoreres imens. Står der derimod et 0, er der ingen operation.[12]

SPI Arduino-bibliotek

For at udvikle SPI og flash parallelt, er der valgt at udvikle og teste flashmodulet ved hjælp af Arduino SPI bibliotek. Der er valgt et Seeduino Mega udviklingsbord, som kan operere ved 3,3 V.Dette udviklingboard har et indbygget SPI interface, som bruges til at udvikle biblioteket.

Arduino-biblioteket indeholder seks funktioner, hvor den vigtigste af disse er "transfer"-funktionen. Denne funktion sender en byte med data til "MOSI" pinsene, og returnerer værdien af den byte, som er modtaget på "MISO"-benet. Arduinobiblioteket giver også mulighed for at opdele klokken. Funktionen "setClockDivider" kan dividere Arduino klokken med 2, 4, 8, 16, 32, 64 eller 128, hvilket betyder, at den hurtigste clockfrekvens SPI kan køre med, er 4 MHz.

Det er også muligt at skifte polariteten og fasen for SPI-overførslen. Data kan sendes med MSB eller LSB i starten, hvilket kan bestemmes af funktionen "setBitOrder". For at alle disse kommandoer virker korrekt, skal "begin"-funktionen afvikles. Denne sætter SCK, MOSI, SS til outputs og sætter SCK og MOSI til logisk lav, og SS til logisk høj.

Beskrivelse af kode

Der er implementeret en funktion for hver kommando. Funktioner kan indeholde andre funktioner, eksempelvis indeholder "Page Program" funktionen et kald til "Write Enable" funktionen.

```
1
2  void FlWE (unsigned char FCS)
3  {
4      digitalWrite(FCS,LOW);
5      SPI.transfer(WREN);
6      digitalWrite(FCS, HIGH);
7  }
8
9  /**
10  * Write disable.
11  * @param FCS chip select pin
12  */
13 void FlWD (unsigned char FCS)
14 {
15     digitalWrite(FCS,LOW);
16     SPI.transfer(WRDE);
17     digitalWrite(FCS,HIGH);
18 }
19
20 /**
21  * Reset flash.
22  * @param FCS chip select pin
23  */
24 void FlReset (unsigned char FCS)
25 {
26     digitalWrite(FCS,LOW);
27     SPI.transfer(RESET);
28     digitalWrite(FCS,HIGH);
29 }
30
31 /**
32  * Read status register and run loop as long as there is any bits in status register that set to "1!"
33  * @param FCS chip select pin
34  */
35 void FlReadStatReg (unsigned char FCS)
36 {         int i = 1;
37     digitalWrite(FCS,LOW);
38     SPI.transfer(RDSR1);
39     while(i==1){
40         unsigned char read = SPI.transfer(DUMMY);
41         // Serial.print("status register:  ");
42         //Serial.println(read,BIN);
43         if(read == DUMMY){
44             digitalWrite(FCS,HIGH);
45             i=0;
46         }
47     }
48 }
49
50 /**
51  * Read data on 3 byte address.
52  * @param FCS chip select pin
53  * @param adr First 3 rightmost bytes contain address.
54  * @return readed Data on address adr.
55  */
56 unsigned char FlRead (unsigned char FCS, unsigned int adr)
57 {
58     char read = 0;
59     digitalWrite(FCS, LOW);
60     SPI.transfer(READ);
```

```
61      SPI.transfer((adr & 0x00FF0000) >> 16);
62      SPI.transfer((adr & 0x0000FF00) >>  8);
63      SPI.transfer((adr & 0x000000FF) >>  0);
64      read =  SPI.transfer(DUMMY);
65      digitalWrite(FCS, HIGH);
66      return read;
67  }
68
69  /**
70    * Write data to 3 byte address.
71    * @param FCS chip select pin
72    * @param data Data to be written to address.
73    * @param adr First 3 rightmost bytes contain address.
74    */
75  unsigned char FlProgram(unsigned char FCS,unsigned int adr, unsigned char data)
76  {
77      FlWE(FCS);
78      digitalWrite(FCS, LOW);
79      SPI.transfer(PP);
80      SPI.transfer((adr & 0x00FF0000) >> 16);
81      SPI.transfer((adr & 0x0000FF00) >>  8);
82      SPI.transfer((adr & 0x000000FF) >>  0);
83      SPI.transfer(data);
84      digitalWrite(FCS,HIGH);
85      FlReadStatReg(FCS);
86
87
88  }
89
90  /**
91   * Write page of data in form of an unsigned char data array .
92   * @param FCS chip select pin
93   * @param adr First 3 rightmost bytes contain address.
94   * @param *data  Pointer to data array to program, may max be 256 bytes long!
95   * @param datalen Defined number of loops must be done to program data,
96   * must be as big as array(if array is data[256], datalen must be 256)
97   */
98  unsigned char FlProgramEx (unsigned char FCS,unsigned int adr,unsigned char *data,unsigned char datalen)
99  {
100   unsigned char i;
101   FlWE(FCS);
102   digitalWrite(FCS,LOW);
103   SPI.transfer(PP);
104   SPI.transfer((adr & 0x00FF0000) >> 16);
105   SPI.transfer((adr & 0x0000FF00) >>  8);
106   SPI.transfer((adr & 0x000000FF) >>  0);
107   for(i=0 ; i<=datalen ; i++){
108     SPI.transfer(data[i]);
109   }
110   digitalWrite(FCS,HIGH);
111   FlReadStatReg(FCS);
112 }
113
114 /**
115  * Read a page of data from flash.
116  * @param FCS chip select pin
117  * @param adr First 3 rightmost bytes contain address.
118  * @param *data  Pointer to data array to read to, may max be 256 bytes long!
119  * @param datalen Defined number of loops must be done to program data,
120  * must be as big as array(if array is data[256], datalen must be 256)
121  */
122 void FlReadEx (unsigned char FCS,unsigned int adr, unsigned char *read, unsigned char datalen)
123 {
124   unsigned char i;
125   digitalWrite(FCS,LOW);
```

```
126   SPI.transfer(READ);
127   SPI.transfer((adr & 0x00FF0000) >> 16);
128   SPI.transfer((adr & 0x0000FF00) >>  8);
129   SPI.transfer((adr & 0x000000FF) >>  0);
130   for (i = 0; i<=datalen; i++){
131     read[i] =  SPI.transfer(DUMMY);
132   }
133   digitalWrite(FCS,HIGH);
134 }
135 
136 /**
137  * Delete byte of data at 3 byte address.
138  * @param FCS chip select pin
139  * @param adr First 3 rightmost bytes contain address.
140  */
141 void FlErase (unsigned char FCS,unsigned int adr)
142 {
143     FlWE(FCS);
144     digitalWrite(FCS,LOW);
145     SPI.transfer(SE);
146     SPI.transfer((adr & 0x00FF0000) >> 16);
147     SPI.transfer((adr & 0x0000FF00) >>  8);
148     SPI.transfer((adr & 0x000000FF) >>  0);
149     digitalWrite(FCS,HIGH);
150     FlReadStatReg(FCS);
151 }
152 
153 /**
154  * Delete everything in flash memory array.
155  * @param FCS chip select pin
156  */
157 void FlEraseAll (unsigned char FCS)
158 {
159     FlWE(FCS);
160     digitalWrite(FCS,LOW);
161     SPI.transfer(BE);
162     digitalWrite(FCS,HIGH);
163     FlReadStatReg(FCS);
164 }
165 
166 /**
167  * Resets Status and Instruction registers
168  * @param FCS chip select pin
169  */
170 void FlWriteReg (unsigned char FCS)
171 {
172 
173     FlWE(FCS);
174     digitalWrite(FCS,LOW);
175     SPI.transfer(WRR);
176     SPI.transfer(0x00);
177     SPI.transfer(0x00);
178     digitalWrite(FCS,HIGH);
179     FlReadStatReg(FCS);
180 
181 }
```

Listing 7.27: Test 0 kode

Alle funktioner har et argument "FCS" , som sættes af chip select benet. Dette gør det muligt at sætte chip select benet fra main filen, uden at ændre i biblioteket.

- Listing 7.27 begynder med at "FlWE" funktionen på linje 2, sætter chip select benet til logisk lav. Derefter

afvikles "Write enable" funktionen som sætter chip select til logisk høj, efter det ønskede data er overført.

- Funktionen "FlWD" (Write disable) kan ses på linje 13. Den ligner meget "Write Enable" funktionen, blot med forskellen den respektive opcode.
- "FlReset" funktionen resetter flashmodulet, og kan køres en gang i starten, så brugeren ved i hvilken tilstand flashen begynder at arbejde fra. Funktion ligner igen "FlWE" funktionen på linje 2, og forskellen er kun i hex-instruktionskoden.
- "FlReadStatReg" er en af det vigtigste funktioner i dette bibliotek. Dette funktion læser status registerets indhold, og hvis status registeret er tom, afslutter funktion. Ellers virker den som en delay og kører et while loop indtil status register retunere 0. Denne funktion er blev brugt til at sikre for at der ikke bliver skrevet og slettet fra hukommelsesmatricen samtidig.
- "FlRead" funktion på linje 56 har et argument mere, som er en adresse. Dette adresse er en int-størrelse. Det betyder at den skal skrives i format `0x00XXXXXX` , hvor "X" er en hexidecimal tal, og sidste to pladser skal være tomme, fordi der er brugt 24 B adresser. Denne funktion retunerer en byte af data, som blev læst fra flashmodulet.
- "FlProgram" funktionen på linje 75 skriver en byte af data til hukommelsesmatricen. Derfor har den et argument mere, som repræsenterer en byte af data. Her er første gang "FlReadStatReg" funktionen bliver brugt. Den sørger for at der bliver ventet, mens der blev udført en "Page Program" kommando.
- Funktionen "FlProgramEx" på linje 98 er en udvidet udgave af "FlProgram" funktionen. Denne funktion kan programmere en page af data til hukommelsesmatricen.
- Funktion "FlReadEx" på linje 122 er en udvidet udgave af "FlRead" funktionen. Denne funktion kan læse data fra hukommelsesmatricen. Længde af data der skal læses, kan bestemmes af argumentet "datalen".
- "FlErase" funktionen på linje 141 sletter en sektor af hukommelsesmatricen (sætter alle bittene til "1").
- "FlEraseAll" funktionen på linje 157 sletter alt data i hukommelsesmatricen (sætter alle bittene til "1").
- "FLWriteReg" funktionen på linje 170 skriver en byte til konfigurationsregisteret og en byte til statusregisteret.

7.4 I²C

7.4.1 Baggrunden for I²C

Start og stop betingelser

Som beskrevet under den generelle beskrivelse af I²C-bussen, Afsnit 5.4, har alle enheder forbindelse til SDA (data) og SCL (clock) og alle enheder kan derved se ændringer på disse pins. Hvis en masterenhed ønsker at udføre en transaktion til eller fra en anden enhed, startes denne transaktion med en startbetingelse og stoppes med en stopbetingelse. En startbetingelse fra en masterenhed, sætter bussen til busy, således at masteren ejer bussen indtil transaktionen stoppes ved en stopbetingelse fra masteren.

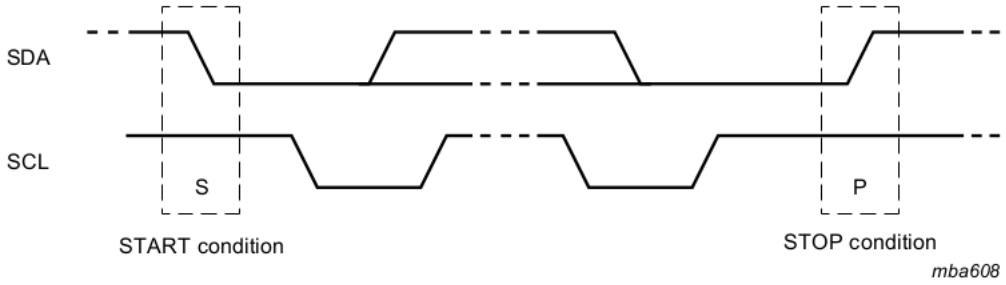
En startbetingelse består af en nedadgående flanke på SDA linjen, imens SCL linjen er høj. Stopbetingelsen består af en opadgående flanke på SCL linjen, imens SCL linjen er høj. Start- og stop betingelserne er illustreret på Figur 7.39, og samtlige tilstande ses i Tabel 7.6.

Dataoverførsel

Bitoverførsel Hvert databit overføres, som illustreret på Figur 7.40. Dataen læses fra SDA linjen, når SCL linjen er høj. Dette betyder at SDA linjen, skal være stabil, så længe SCL linjen er høj. Dataen kan derfor kun

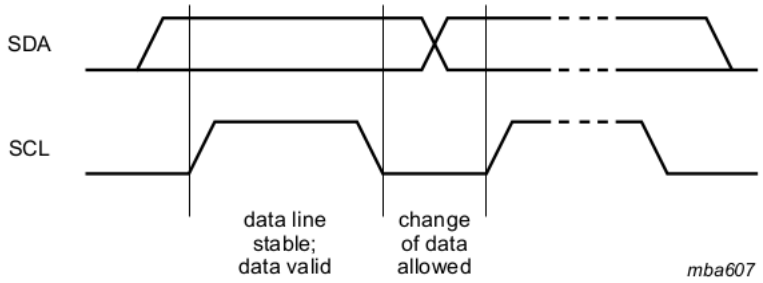
Tilstand	SDA	SCL
Start	HIGH->LOW	HIGH
Stop	LOW->HIGH	HIGH
Data gyldig	HIGH / LOW	HIGH
Dataændring tilladt	HIGH / LOW / HIGH->LOW / LOW->HIGH	LOW

Tabel 7.6: Tilstande på I^{2}C-bussen.



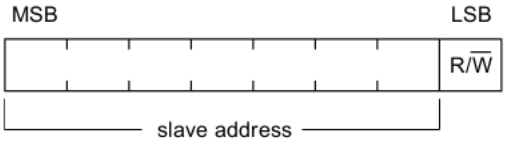
Figur 7.39: I^{2}C-bus start og stop betingelser

skifte værdi, når clocken er lav. En vigtig faktor i denne sammenhæng, er timingen mellem SDA- og SCL linjen, for at dataen aflæses korrekt.



Figur 7.40: I^{2}C-bus bit-overførsel

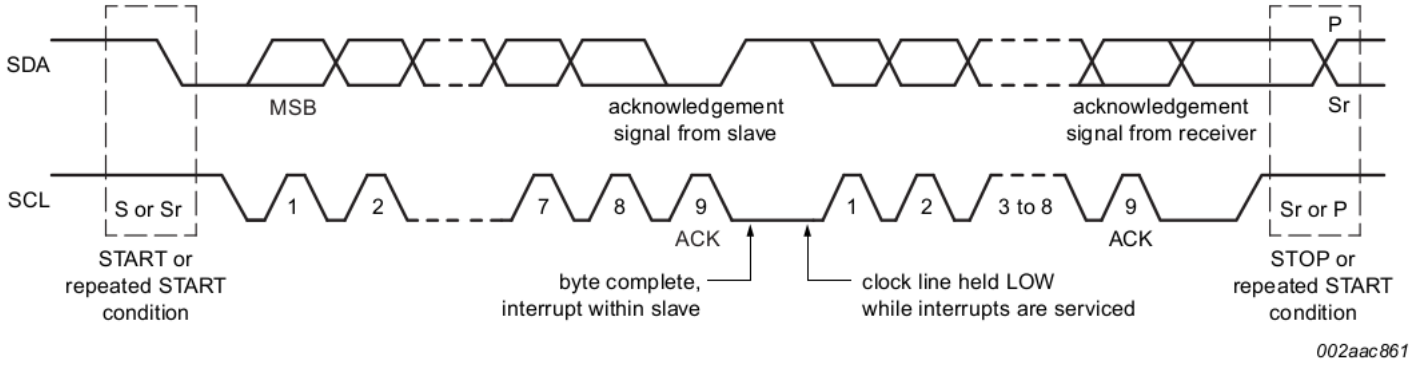
Byteoverførsel Hver dataoverførsel er bestående af en byte på otte bit efterfulgt af et acknowledge-bit og sendes/modtages, med det mest betydende bit (MSB) først. Den første byte efter en startbetingelse, er adresseringsbyten og er opdelt, så de første syv bytes indeholder adressen på den enhed der ønskes at kommunikere med, efterfulgt af et ottende bit, som beskriver data retningen (Read/Write). Et nul/lav på det ottende bit, indikerer en transmission og betyder, at der skrives til fra masteren til en anden enhed og modsat indikerer et et/høj en forespørgsel om data. Adresseringsbyten kan ses på Figur 7.41



Figur 7.41: Adressebyte efter startbetingelsen

Alle enheder der har forbindelse til I^2C-bussen "lytter" på SDA linjen og efter en startbetingelse fra en master, er alle enheder klar til at modtage den første adresseringsbyte. Enhederne der modtager adresseringsbyten, sammenligner adressen med deres egen adresse og svarer tilbage på SDA linjen, med et acknowledge bit, som er det niende bit i en overførsel. Det niende bit, kan enten være nul/lav, for at indikerer et acknowledge (ACK) eller et/høj for not acknowledge (NACK). Modtager masteren et ACK, kan der fortsættes ved at sende eller modtage data, men modtages der et NACK signal, kan masteren enten vælge at sende en stopbetingelse eller gentage startbetingelsen, hvis der ønskes at oprette en ny overførsel. Hele overførslen kan ses på Figur 7.42. Som udgangspunkt er acknowledge bitten altid høj pga. pull-up modstandende på bussen, dette betyder at der kan være flere grunde til at masteren modtager et NACK signal på SDA linjen. Fem af de hyppige årsager til et NACK-signal er beskrevet herunder:

- Der er ingen enheder med den anmodede adresse, hvilket betyder at der ikke er nogen enheder der svarer.
- Modtageren er ikke klar til at modtage eller sende data, da den har travlt med at udføre andre funktioner og derfor ikke er klar til at kommunikere med masteren.
- Modtageren får data, som den ikke forstår.
- Modtageren kan ikke modtage flere databytes, dette kan evt. skyldes en øvre grænse sat af brugeren, eller en begrænsning sat af hukommelsen.
- Hvis masteren læser data fra en anden enhed, er det masterens opgave at sende et acknowledge og der modtages derfor altid et NACK signal fra den anden enhed.



Figur 7.42: I^2C-Bus byte overførsel

Der skal fra modtagerens side, altid sendes et acknowledge bit efter hver overført byte, hvis kommunikationen forløber som den skal. Dette gælder også for databytes, som også er af længden otte bit. Efter hver modtaget databyte, sendes der et acknowledge bit for at verificere at data'en er modtaget korrekt og at modtageren er klar til næste overførsel.[10]

Det er værd at notere sig, at når en slave/master trækker et acknowledge (trækker sda ned), er det altid masteren der styrer clocken.

Alternativ til multi-master I^2C

Det blev i den generelle beskrivelse af I^2C-bussen, Afsnit 5.4, beskrevet at I^2C-bussen skulle designes, så alle enhederne fungerede som multimaster-enheder. Det har senere vist sig, at multimaster-metoden er for omfattende og at der derfor skal tages hensyn til alt for mange detaljer, i forbindelse med udvikling af denne metode.

Det vælges derfor, at lave I^{2}C-bussen på den lidt mere standardiserede måde, hvor der anvendes en master mikrocontroller, til at styre alt kommunikation mellem slave/MicroBlaze enhederne, som vist på Figur 5.6. Det er bl.a. valgt at anvende denne metode, fremfor multimaster-metoden da denne metode er simplere og mindre tidskrævende at udvikle og teste. Dette betyder også at nogle af de funktionelle- og specifikke krav er tilpasset den nye master-slave metode.

Idéen med denne metode er, at have en fælles master på bussen, som står for al intern kommunikation. Masteren læser efter tur en pakke fra hver slave, for at se, om de vil kommunikere med andre delmoduler. Vil det givne slavemodul det, foretages en transaktion på én pakke, og masteren rykker videre til næste slavemodul.

Et flowchart for denne kommunikation ses på Figur 7.43. Pakken, som masteren læser fra slaven ved hver omgang, ses på Figur 7.44. Her betyder `read`, at slaven ønsker at læse en pakke data fra slaven med adressen `Address` – modsat betyder `write`, at slaven ønsker at skrive en pakke data, allerede givet ved `Data`, til slaven med adresse `Address`.

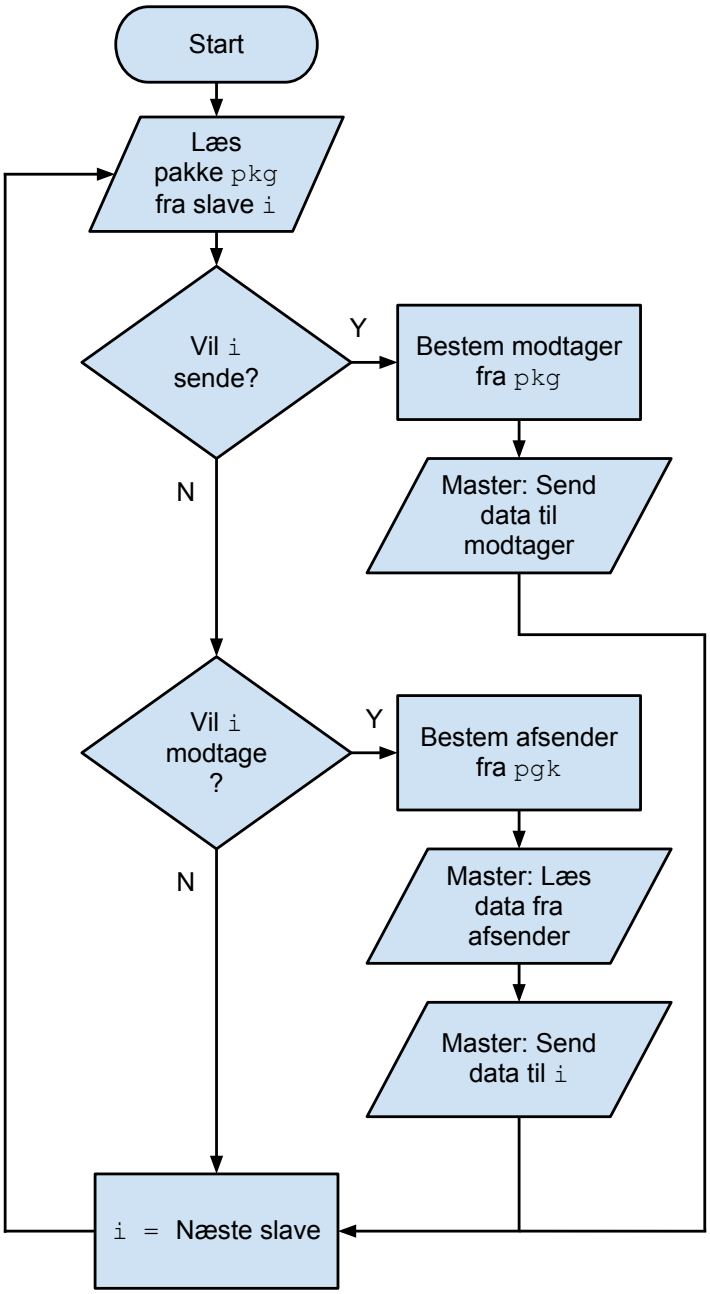
Ulempen ved denne metode er, at det tager længere tid at få data frem og tilbage imellem slaverne, end hvis de kunne snakke direkte sammen.

7.4.2 I^{2}C– kode

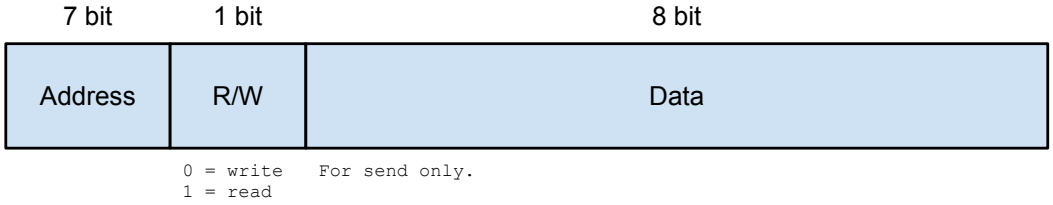
Koden til I^{2}C-biblioteket findes i Listing 7.28 og Listing 7.29. Koden findes også på CD under:

- `code/i2c_avr/{i2c.h, i2c.c}` .

Koden herunder er udviklet til en ATmega32, og skal omskrives før den kan bruges på en MicroBlaze.



Figur 7.43: Fælles master – flowchart.



Figur 7.44: Fælles master – slave-pakke.

```
#define I2C_T        10 // Period time in us with 100khz.
#define I2C_TH        5 // Halfperiod time in us with 100khz.
#define I2C_CH        4 // Clock High for bit transfer.
#define I2C_CL        3 // Clock Low for bit transfer.
#define I2C_READ      1 // Read from slave.
#define I2C_WRITE     0 // Write to slave.

#define I2C_ACK       1
#define I2C_ENACK    -2
#define I2C_EALLOC   -3

#define HIGH  1
#define LOW   0
#define TRUE  1
#define FALSE 0

/**
 * Transmit or request to receive data from slave.
 * @param[in,out] sdaline Pin of serial data.
 * @param[in,out] sclline Pin of serial clock.
 * @param[in] addr_rw Address of slave.
 * Bit [7-1] address Bit [0] 1=read,0=write.
 * @param[in,out] data Place to read or write data.
 * @param[in] cnt Lenght of data.
 *
 * @return 0 = sucess, I2C_EALLOC = allocation error,
 *             I2C_ENACK = no ACK/NACK.
 */
/* signed char i2c_transceive(unsigned char sdaline, unsigned char sclline,  */
/*                            unsigned char addr_rw, unsigned char *data, */
/*                            unsigned long cnt); */
/**
 * Receive data from bus - wait for start and execute command, if address
 * matches.
 * @param sdaline Pin of serial data.
 * @param sclline Pin of serial clock.
 * @param own_addr Own master/slave adress.
 *
 * @return 0 = sucess, -X = Error
 */
/* unsigned char *i2c_receive(unsigned char sdaline, unsigned char sclline,  */
/*                            unsigned char own_addr); */
```

Listing 7.28: I^{2}C-kode – her vises `i2c.h`

```
#define F_CPU 8000000UL

#include "i2c.h"
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>

/**
 * PLATFORM SPECIFIC.
 */
#define udelay(us) _delay_us(us)
// static inline void udelay (unsigned long us)
// {
//     _delay_us (us);
// }

/**
 * PLATFORM SPECIFIC.
 * @return 0 = sucess
```

```
 */
static signed char setsda(unsigned char sda, unsigned char state)
{
    DDRD = 0xFF;                  /* Port D output */
    if (state > 0)
        PORTD |= (1 << sda);
    else
        PORTD &= ~ (1 << sda);

    return 0;
}

/**
 * PLATFORM SPECIFIC.
 * @return 0 = sucess
 */
static signed char setscl(unsigned char scl, unsigned char state)
{
    DDRD = 0xFF;                  /* Port D output */
    if (state > 0)
        PORTD |= (1 << scl);
    else
        PORTD &= ~ (1 << scl);

    return 0;
}

/**
 * PLATFORM SPECIFIC.
 * @return 1 = High, 0 = Low
 */
static signed char getsda(unsigned char sda)
{
    DDRD &= ~ (1 << sda); /* Port D input. */
    PORTD |=  (1 << sda); /* Port D pullup */
    return (PIND >> sda) & 0x01;
}

/**
 * PLATFORM SPECIFIC.
 * @return 1 = High, 0 = Low
 */
static signed char getscl(unsigned char scl)
{
    DDRD &= ~ (1 << scl); /* Port D input. */
    PORTD |=  (1 << scl); /* Port D pullup */
    return (PIND >> scl) & 0x01;
}

/**
 * @return 0 = sucess
 */
static signed char starttrans(unsigned char sda, unsigned char scl)
{
    setsda(sda, HIGH); /* Initial state */
    setscl(scl, HIGH);

    udelay(I2C_TH);
    setsda(sda,LOW);
    udelay(I2C_TH);
    setscl(scl,LOW);
    return 0;
}

/**
```

```
85  * @return 0 = sucess
86  */
87 static signed char stoptrans(unsigned char sda, unsigned char scl)
88 {
89     setsda(sda, LOW);
90     setscl(scl, HIGH);
91     udelay(I2C_TH);
92     setsda(sda, HIGH);
93     udelay(I2C_TH);
94     return 0;
95 }
96
97 /**
98  * @return 0 = sda LOW, 1 = sda HIGH
99  */
100 static signed char readbit(unsigned char sda, unsigned char scl)
101 {
102     unsigned char ret;
103
104     setscl(scl, LOW);
105     udelay(I2C_CL);
106     setscl(scl, HIGH);
107     ret = getsda(sda);
108     udelay(I2C_CH);
109     setscl(scl, LOW);
110     udelay(I2C_CL);
111
112     return ret;
113 }
114
115 /**
116  * @return 0 = sucess
117  */
118 static signed char writebit(unsigned char sda, unsigned char scl,
119                             signed char bit)
120 {
121     if (bit > 0)
122         bit = 0x01;
123     else
124         bit = 0x00;
125
126     setscl(scl, LOW);
127     setsda(sda, bit);
128     udelay(I2C_CL);
129     setscl(scl, HIGH);
130     udelay(I2C_CH);
131     setscl(scl, LOW);
132     udelay(I2C_CL);
133
134     return 0;
135 }
136
137
138 /**
139  * @return Byte that is read.
140  */
141 static unsigned char readbyte(unsigned char sda, unsigned char scl)
142 {
143     unsigned char c = 0x00;
144     signed char i;
145
146     for (i = 7; i >= 0; i--) {
147         c |= readbit(sda, scl) << i;
148     }
149     writebit(sda, scl, LOW);
```

```
150
151     return c;
152 }
153
154 /**
155  * @return I2C_ACK = ACK, I2C_ENACK = NACK.
156  */
157 static signed char writebyte(unsigned char sda, unsigned char scl,
158                              unsigned char c)
159 {
160     signed char i;
161     unsigned char tmp;
162
163     for (i = 7; i >= 0; i--) {
164         tmp = (c >> i) & 0x01;
165         writebit(sda, scl, tmp);
166     }
167
168     setsda(sda, HIGH);
169     //    setsda (sda, LOW); /* For auto-acking */
170     if (readbit(sda, scl) == LOW)
171         return I2C_ACK;
172     else
173         return I2C_ENACK;
174 }
175
176 /**
177  * TODO: Make not static for global use. Comment in in header-file.
178  */
179 static signed char i2c_transceive(unsigned char sdaline, unsigned char sclline,
180                                   unsigned char addr_rw, unsigned char *data,
181                                   unsigned long cnt)
182 {
183     unsigned char i;
184     signed char ret;
185
186     starttrans(sdaline, sclline);
187     if (writebyte(sdaline, sclline, addr_rw) == I2C_ENACK)
188         goto nack;
189
190     if ((addr_rw & 0x01) == I2C_READ) {
191
192         /* TODO:
193          * Should there be a dynamic memory
194          * allocation here?
195          */
196
197 //         free(data);
198 //         if ((data = (unsigned char *) malloc(sizeof(char) * cnt)) == NULL)
199 //             return I2C_EALLOC;
200         for (i = 0; i < cnt; i++)
201             data[i] = readbyte(sdaline, sclline);
202     } else {            /* I2C_WRITE */
203         for (i = 0; i < cnt; i++)
204             if ((ret = writebyte(sdaline, sclline, data[i])) == I2C_ENACK)
205                 goto nack;
206     }
207
208     stoptrans(sdaline, sclline);
209     return 0;
210
211 nack:
212     stoptrans(sdaline, sclline);
213     return I2C_ENACK;
214
```

```
215 }
216
217 // /**
218 //  * TODO: Make not static for global use. Comment in in header-file.
219 //  */
220 // static unsigned char *i2c_receive (unsigned char sdaline, unsigned char sclline,
221 //                                    unsigned char own_addr)
222 // {
223 //
224 //     return 0;
225 // }
```

Listing 7.29: I^{2}C-kode – her vises `i2c.c`

8 Test

8.1 LOG

8.1.1 Test på PC, version 1

Dette afsnit indeholder en test af loggen og testdokumentationen hertil, indeholder en beskrivelse af testmetoder, testudførsel og testresultater. Loggen testes individuelt, uden at programmet implementeres i en MicroBlaze og uden implementering af eksterne enheder. Programmet skrives i C og testen udføres på en PC, ved brug af GCC-kompileren. Der udføres først test på version 1 af softwaren, og senere version 2, som beskrevet i slutningen af Afsnit 7.1.3.

Valg af testmetoder

For at lave en gennemgående test, af alle loggens funktioner, er det valgt at teste loggen udfra forskellige testkommandoer fra jorden. Testkommandoerne skal herefter sammenlignes med retur-dataene, som sendes til jorden udfra den enkelte kommando. Dette gøres for at verificere, at retur-dataene matcher den enkelte kommando.

Kommandobeskederne fra jorden, er som beskrevet under udviklingsafsnittet 7.1, opdelt i seks forskellige blokke. En gennemgående test af de forskellige blokke, omfatter derfor seks testscenarier. Her bemærkes det, at der bruges en kommandobesked, til test af et specifikt tidsrum og en til test af "alle data".

Udover de seks forskellige testscenarier, anvendes der to forskellige tests, for at teste for worst case scenarie. Her testes der for en kommandobesked bestående af nuller og en kommandobesked bestående af et taller. Formålet med kommandobeskederne til worst case testscenarierne, er at begge beskeder ikke skal retunere noget til jorden.

De seks forskellige blokke i kommandopakken, kan ses under udviklingsafsnittet 7.1.2. Bemærk at kommando-beskeden består af syv pakker, men den syvende pakke er tom og tages derfor ikke med i testen.

Der testes for følgende kommandoer fra jorden:

- Alle data
- Specifikt ID
- Logtyper af typen warnings
- Specifikt tidsrum
- Bestemt kontrolsekvens: Kun ID og TID
- Kun de tre sidste beskeder

De seks forskellige testscenarier, kan ses på Listing 8.2

For at teste de seks testscenarier, er der skrevet seks forskellige datapakker, som symboliserer dataet i flash-hukommelsen. Disse datapakker er lavet, så de matcher de forskellige kommandobeskeder. På den måde, er det nemt at verificere om retur-dataene er korrekt. Datapakkerne kan ses på Listing 8.1

```
#ifndef TESTDATA_MAIN_H
#define TESTDATA_MAIN_H
#include "bus_id.h"
#include "log.h"
unsigned char busID = ID_LOG_COMMAND;
unsigned char busData_toFlash[6][DATA_LGT] = {
{0x00, 0x00, 0x00, 0x00, 0x03, 0x01, 0x51, 0x64, 0x00, 0x6D, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01},//Test:Unique ID = 3
{0x00, 0x00, 0x00, 0x00, 0x01, 0x04, 0x51, 0x64, 0x00, 0x5D, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01},//Test:Unique LOGTYPE = 4
{0x00, 0x00, 0x00, 0x00, 0x01, 0x04, 0x51, 0x64, 0x00, 0x7C, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01},//Test:Unique LOGTYPE = 4
{0x00, 0x00, 0x00, 0x00, 0x02, 0x04, 0x51, 0x64, 0x00, 0x4C, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01},//Test:Unique LOGTYPE = 4
{0x00, 0x00, 0x00, 0x00, 0x01, 0x01, 0x51, 0x64, 0x00, 0x6D, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01},//Test:Unique TIMEspan
{0x00, 0x00, 0x00, 0x00, 0x02, 0x01, 0x51, 0x64, 0x00, 0x6D, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01},//Test:Unique TIMEspan
```

Listing 8.1: Kommando beskeder

```
#ifndef TESTDATA_LOG_H
#define TESTDATA_LOG_H
#include "log.h"
unsigned char busCommands[8][COMMAND_LGT] = {
{0x00, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00},//Pass all through
{0x03, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00},//Unique ID = 0x03
{0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00},//Unique LOGTYPE = 0x04, only
     warnings
{0x00, 0xff, 0x51, 0x64, 0x00, 0x6c, 0x51, 0x64, 0x00, 0x6e, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00},//Unique TIME interval, from
    0x5164006c to 0x5164006e
{0x00, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0x14, 0x00, 0x00, 0x00, 0x00, 0x00},//Unique CTRL sequence = 0x14
     - only ID and TIME
{0x00, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x03, 0x00, 0x00, 0x00},//Only the last NUM = 0x0003
    messages are sent down
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},//Only zeros
{0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff} //Only ones
```

Listing 8.2: Test data

Test 0

Her udføres testen svarende til, at brugeren taster '0', som vist tidligere (Figur 7.5). I følgende tests, vil testnummeret ligeledes svare til det nummer brugeren taster i brugergrænsefladen.

Input: Kommandoen `0x00, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00` har til formål, at lukke al data igennem.

Forventet output: Af dataene præsenteret i Listing 8.1, forventes det, at samtlige data kommer igennem:

```
0 0 0 1 3 1 51 64 0 6d 1 1 1 1 1 1
0 0 0 2 1 4 51 64 0 5d 1 1 1 1 1 1
0 0 0 3 1 4 51 64 0 7c 1 1 1 1 1 1
0 0 0 4 2 4 51 64 0 4c 1 1 1 1 1 1
0 0 0 5 1 1 51 64 0 6d 1 1 1 1 1 1
0 0 0 6 2 1 51 64 0 6d 1 1 1 1 1 1
```

Listing 8.3: Forventet resultat af test 0

Resultat: Alle data i Listing 8.3 kom igennem.

Test 1

Input: Kommandoen `0x03, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00` har til formål, kun at lukke data med ID = `0x03` igennem.

Forventet output: Af dataene præsenteret i Listing 8.1, forventes det, at alle data med ID = `0x03` lukkes igennem:

```
0 0 0 1 3 1 51 64 0 6d 1 1 1 1 1 1
```

Listing 8.4: Forventet resultat af test 1

Resultat: Kun den ene datalinje i Listing 8.4 kom igennem.

Test 2

Input: Kommandoen `0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00` har til formål, kun at lukke data med LOGTYPE = `0x04` igennem.

Forventet output: Af dataene præsenteret i Listing 8.1, forventes det, at alle data med LOGTYPE = `0x03` lukkes igennem:

```
0 0 0 1 1 4 51 64 0 5d 1 1 1 1 1 1
0 0 0 2 1 4 51 64 0 7c 1 1 1 1 1 1
0 0 0 3 2 4 51 64 0 4c 1 1 1 1 1 1
```

Listing 8.5: Forventet resultat af test 2

Resultat: De tre datalinjer med `0x04` som logtype i Listing 8.5 kom igennem.

Test 3

Input: Kommandoen `0x00, 0xff, 0x51, 0x64, 0x00, 0x6c, 0x51, 0x64, 0x00, 0x6e, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00` har til formål, kun at lukke data indenfor tidsintervallet [`0x5164006c` : `0x5164006e`] igennem.

Forventet output: Af dataene præsenteret i Listing 8.1, forventes det at følgende data, indenfor det givne interval, lukkes igennem:

```
0 0 0 1 3 1 51 64 0 6d 1 1 1 1 1 1
0 0 0 2 1 1 51 64 0 6d 1 1 1 1 1 1
0 0 0 3 2 1 51 64 0 6d 1 1 1 1 1 1
```

Listing 8.6: Forventet resultat af test 3

Resultat: De tre datalinjer i Listing 8.6 med TIME indenfor tidsintervallet, kom igennem.

Test 4

Input: Kommandoen `0x00, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0x14, 0x00, 0x00, 0x00, 0x00, 0x00` har til formål, at lukke alle datalinjer igennem, men kun sende/printe ID og TIME.

Forventet output: Af dataene præsenteret i Listing 8.1, forventes det at alle datalinjer, men kun ID og TIME, kommer igennem:

```
3 51 64 0 6d 0 0 0 0 0 0 0 0 0 0 0
1 51 64 0 5d 0 0 0 0 0 0 0 0 0 0 0
1 51 64 0 7c 0 0 0 0 0 0 0 0 0 0 0
2 51 64 0 4c 0 0 0 0 0 0 0 0 0 0 0
1 51 64 0 6d 0 0 0 0 0 0 0 0 0 0 0
2 51 64 0 6d 0 0 0 0 0 0 0 0 0 0 0
```

Listing 8.7: Forventet resultat af test 4

Resultat: Alle datalinjers ID og TIME i Listing 8.7 kom igennem.

Test 5

Input: Kommandoen `0x00, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x03, 0x00, 0x00, 0x00` har til formål, at matche alt, som kommandoen i test 0, men kun sende/printe de første 3 datalinjer.

Forventet output: Af dataene præsenteret i Listing 8.1, forventes det at de første tre beskeder i loggen printes ud:

```
0 0 0 1 3 1 51 64 0 6d 1 1 1 1 1 1
0 0 0 2 1 4 51 64 0 5d 1 1 1 1 1 1
0 0 0 3 1 4 51 64 0 7c 1 1 1 1 1 1
```

Listing 8.8: Forventet resultat af test 5

Resultat: De første tre datalinjer, som i Listing 8.8, kom igennem.

Test 6

Input: Kommandoen `0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00` har til formål, at teste hvad der sker, når der kun er nuller i kommandoen.

Forventet output: Det forventes ikke, at der kommer nogle data igennem.

```
```

Listing 8.9: Forventet resultat af test 6

Resultat: Resultatet var ikke som forventet. Resultatet blev som følger:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Listing 8.10: Resultat af test 6

Da det ikke ønskes at der sendes noget til jorden, når kontrolregistret er ene af nuller, skrives kode der sørger for, at dette er det første der tjekkes, når en besked skal sendes ned. Følgende indsættes, og findes fremover på cd:

code/LOG/log.c

```
/* If command is only zeros. */
if(command[10] == 0x00) return;
```

Listing 8.11: Kode tilføjet som følge af test 6

Test 7

Input: Kommandoen `0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff` har til formål, at teste hvad der sker, når der kun er 1-taller i kommandoen.

Forventet output: Det forventes ikke, at der kommer nogle data igennem.

Listing 8.12: Forventet resultat af test 7

Resultat: Der kom ingen data igennem.

Delkonklusion

Loggen er nu færdiggjort, så den fungerer (skriver data, læser data bestemt af kommandobeskeden), på en PC. Næste skridt er, at få koden implementeret på en MicroBlaze, og blive uafhængig af PC'ens funktioner som tid og skærm. Implementeringen med andre moduler, der står for tid, data og flash, vil først komme senere under Implementering, og i første omgang vil koden altså benytte tilfældige tider og fast input-output-data, da tid og data ikke kan hentes eller skrives uden disse grænseflader.

8.1.2 Test på MicroBlaze, version 1

For at lave en gemmengående test af loggens implementering på en MicroBlaze, vælges det at anvende samme testmetoder, som ved testen af loggen på en PC, Afsnit 8.1.1. Denne testmetode giver den fordel, at testresultaterne fra MicroBlaze testen, kan sammenlignes med testresultaterne fra pc-testen. På den måde kan det nemt verificeres, om testene giver de ønskede testresultater. Implementering af loggen på MicroBlazen testes derfor også, som en blackbox-test 6.1, med de samme kommandobeskeder og datapakker som er anvendt i test på PC'en.

Testen udføres med UART-kommunikation til en PC. Dette fungerer ved at en kommando vælges via switchene på boardet, og på baggrund af kommandobeskeden, sender MicroBlazen retur-data'en fra delmodulerne tilbage til PC'en.

8.1.3 Testresultater

Da testene udføres på samme måde, som ved PC testen, kan input og forventede output, ses under Afsnit 8.1.1.

Testresultaterne viser, at alle testene giver det forventede output, hvorved testresultaterne også kan ses under Afsnit 8.1.1.

Delkonklusion

Loggen er nu implementeret på en MicroBlaze og fungerer med ekstern UART-kommunikation til en PC (jordstation). En sammenligning af testresutaterne fra "test på log" og "test på MicroBlaze" viser, at implementeringen af loggen på en MicroBlaze var vellykket og fungerer uden fejl. For at afsætte den nødvendige tid, til videre implementering mellem LOG, SPI-kommunikation og flash-hukommelsen, er det i første omgang valgt at afgrænse for tilføjelsen af Unix-tiden til databeskederne.

Loggen er nu klar til implementering med resten af systemet, herunder SPI-kommunikation og flash hukommelse.

8.1.4 Test på PC, version 2

For at verificere, at den mere vedligeholdelsesvenlige version 2 at log-koden fungerer, vil samme tests som ovenfor, udføres igen, og resultatet skulle derfor gerne give det samme.

For PC-koden viser det sig, at koden leverer samme resultater som tidligere.

8.1.5 Test på MicroBlaze, version 2

Version 2 testes ligeledes på samme måde som version 1 og vil derfor ikke være beskrevet ydligere. Testen af loggen version 2, gav ikke de samme resultater som version 1 og fungerede derfor ikke efter planen. Den første antagelse, var at den gratis MicroBlaze version ikke havde plads nok, da version 2 af loggen fyldte mere end version 1. Denne antagelse blev bl.a. taget, da der ikke kunne findes nogen fejl i koden efter længere tids fejlfinding og da den gratis version af MicroBlaze, så ud til at være begrænset. Senere har det vist sig at der uden problemer, kan eksekveres større mængde kode på den gratis MicroBlaze MCS og derfor konkluderes, at der må være en overset fejl et sted i softwaren. Fejlen blev ikke fundet, da det pga. tidsmangel og prioritering blev valgt at bruge tiden på andre dele af projektet. Denne prioitering blev bl.a. valgt, da testen af loggen version 2, på en PC gav de ønskede resultater og denne test derfor ikke prioiteres så højt.

8.2 SPI

8.2.1 Test af software

Dette afsnit indeholder tests af SPI-biblioteket, og dokumentation samt beskrivelse af testmetoder. Til de initialiserende tests, er der skrevet en replikation af biblioteket til PC, hvor den er sat op til at printe de bitmønstre, der sættes ud på MicroBlazens GPO porte.

Vælg af testmetoder

Testen går ud på, at se om softwaren giver de rigtige bitmønstre, der skal sættes ud på MicroBlazens GPO porte, for at følge protokollen beskrevet på Figur 7.16. Testen foregår ved at, der indsættes kendte parametre i et funktionskald og derefter ses om outputtet stemmer overens med det forventede bitmønster.

Test koden kan ses i følgende

```
1 #include <stdio.h>
2 //Settings
3 #define SPI_CONT 0x100000
4
5 #define SPI_CLKDIV0 0x0
6 #define SPI_CLKDIV2 0x20000
7 #define SPI_CLKDIV4 0x40000
8 #define SPI_CLKDIV8 0x80000
9
```

```
10 #define SPI_MODE0 0x0
11 #define SPI_MODE1 0x4000
12 #define SPI_MODE2 0x8000
13 #define SPI_MODE3 0xC000
14 #define SPI_SEND 0x1
15
16 #define SPI_GPIO_REG 1
17
18 unsigned int spiInit(int initCond){
19   printf("0x%x\n",initCond);
20 }
21
22 unsigned int spiSend(int slave, int data){ //u32 int
23   //Set data and the begin transfer signal
24   //Shift bits to correct location.
25   slave = slave << 1;
26   data = data << 6;
27   printf("0x%x\n", slave|data|SPI_SEND);
28 }
29
30 unsigned int spiRecv(){
31   //Read from specefic ports and return value as raw 8bit, needs error handling.
32   int dataReg; //u32
33   int data; //u8 int
34   dataReg = 0b00000001010010100000000;  // Dummy data
35   // extract the 8 data bits
36   data = (dataReg & 0b11111111);
37   printf("0x%x\n", data);
38 }
39
40 unsigned int spiTran(int slave, int data, int delay){
41   //This function is a complete transfer implementation, that can be used to send commands and retrive data over the SPIbus.
42   spiSend(slave, data);
43   sleep(delay);
44   return spiRecv();
45 }
46
47
48 unsigned int spiSendLarge(int slave, int data, int nPack){ //u32 int
49   int n;
50   int pack;
51   //Set data and the begin transfer signal
52   //Shift bits to correct location.
53   slave = slave << 1;
54   unsigned int shiftMask = 0b11111111 << (8*(nPack-1));
55   printf("Shift mask is: 0x%x \n",shiftMask);
56   for(n = 0; n<nPack;n++){
57     pack = ((data & shiftMask)>> (8*(nPack-1))); //<< 6;
58     printf("Package %d contains: 0x%x\n",n, pack); //slave|pack|SPI_SEND);
59     data = data << 8;
60   }
61 }
```

Test af spiInit() : Der er valgt ikke at teste med alle permutationer af parametre, men blot at teste med et udvalg. Resultatet kan ses i Tabel 8.1

Test af spiSend() : Der er valgt ikke at teste med alle permutationer af parametre, men blot at teste med et udvalg. Resultatet kan ses i Tabel 8.2

Test af spiRecv() : Denne funktion kræver at, den får data fra en GPI port, dette simuleres blot med noget dummy-data.Der vælges ikke at teste med alle permutationer af parametre, men blot at teste med et udvalg.

Input	Output (Binær)	Forventet (Binær)	Bestået
SPI_CONT|SPI_CLKDIV0|SPI_MODE0	10000000000000000000	10000000000000000000	✓
SPI_CLKDIV0|SPI_MODE0	00000000000000000000	00000000000000000000	✓
SPI_CONT|SPI_CLKDIV2|SPI_MODE1	10010010000000000000	10010010000000000000	✓
SPI_CLKDIV2|SPI_MODE1	00010010000000000000	00010010000000000000	✓
SPI_CLKDIV4|SPI_MODE2	00100100000000000000	00100100000000000000	✓
SPI_CLKDIV8|SPI_MODE3	01000110000000000000	01000110000000000000	✓

Tabel 8.1: Test data for spiInit

Input	Output (Binær)	Forventet (Binær)	Bestået
spiSend(2, 255)	00000001111111000101	00000001111111000101	✓
spiSend(15, 21)	00000000010101011111	00000000010101011111	✓
spiSend(25, 110)	00000001101110110011	00000001101110110011	✓

Tabel 8.2: Test data for spiSend

Resultatet kan ses i Tabel 8.3

Input	Output (Binær)	Forventet (Binær)	Bestået
00000010100101000000	01000000	01000000	✓
10101101010010101010	01010101	01010101	✓
00011001010011111111	11111111	11111111	✓

Tabel 8.3: Test data for spiRecv

Test af spiTrans() : Denne funktion består blot af funktionskald til de ovennævnte funktioner og derfor testes denne kun med et sæt data, da de tidligere funktioner er gennemtestet.

Input	Output og input (Hex)	Forventet (Hex)	Bestået
spiTran(31 ,255, 2)	3FFF og 55	3FFF og 55	✓

Tabel 8.4: Test data for spiTrans

Test af spiSendLarge() : Der er valgt ikke at teste med alle permutationer af parametre, men blot at teste med et udvalg. Resultatet kan ses i Tabel 8.5

Der testes med parametre `spiSendLargeTest(1, 0xF1F2F3F4, 4)`

Test af kode på Microblaze

Efter koden er omskrevet, således at de generede bitmønstre sættes ud på GPO porten, er disse mønstre blevet verificeret ved at sætte lys-dioder til, for på den måde at få et visuelt overblik. Resultatet stemmer overens med resultatet fra testen på PC.

Test af hardware

Hardwaren er løbende testet i forbindelse med udviklingen, og kan læses i Afsnit 7.2.1.

Pakke	Output (Hex)	Forventet (Hex)	Bestået
Pakke 0:	F1	F1	✓
Pakke 1:	F2	F2	✓
Pakke 2:	F3	F3	✓
Pakke 3:	F4	F4	✓

Tabel 8.5: Test data for spiSendLarge

8.3 Flash

8.3.1 Test med Arduino

Dette afsnit indeholder en test af flashmodulet og dokumentation hertil og beskrivelse af testmetoder. Testen udføres ved hjælp af et Arduino udviklingsboard og Arduino software. Alt koden er skrevet i C++ med Arduino biblioteker og der er anvendt et SPI bibliotek.

Valg af testmetoder

De vigtigste funktioner til flashmodulet skal testes. Disse er at skrive, læse og slette data. Testene er blevet delt i tre minisprints.

Først testes de simple funktioner, som er at programmere en byte til flashmodulet, læse en byte fra flashmodulet og slette en byte fra flashmodulet.

I næste minisprint programmeres en page på 256 B til flashmodulet, slette en sektor på 64 KiB, og læse hele pagen ud efterfølgende.

I sidste minisprint forsøges der at slette hele flashen og efterfølgende skrives til en arbitrær sektor.

Test 0

Formålet med denne test er at skrive tal til de første 256 pladser. Meningen er at der først skal slettes data fra hver plads i flashmodulet og herefter skrives ny data til hver enkelt plads. For at realisere dette skal der benyttes read, write og delete funktioner. For at tjekke om testen udføres som forventet, skal der læses fra flashen før sletning, samt når der er skrevet til flashen igen.

Implementering

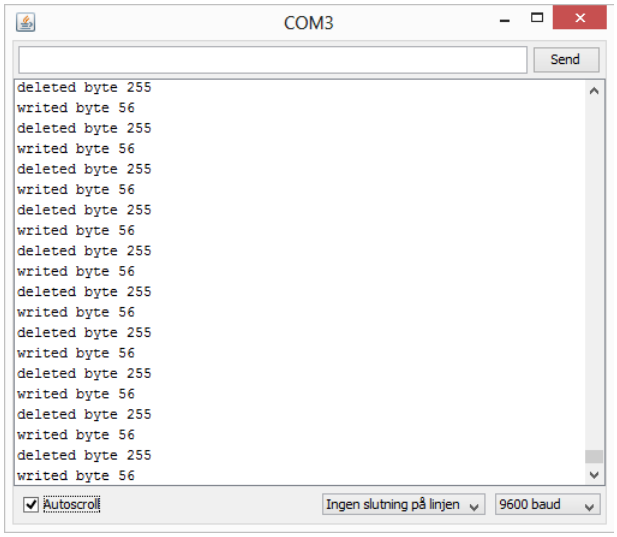
```
unsigned char run = 1;
void loop ()
{
  if(run){
    run = 0;
    unsigned char data = 56;
    unsigned int i;
    FlWriteReg(FCS);
    FlErase(FCS,0);
    for( i = 0; i<=255;i++){
      Serial.print("deleted byte ");
      Serial.println(FlRead(FCS,i));
      FlProgram(FCS,i,data);
      Serial.print("writed byte ");
      Serial.println(FlRead(FCS,i));
    }
```

```
  }
}
```

Listing 8.13: Test 0 kode

Testen er implementeret ved hjælp af et for loop, som kører fra 0 til 255. "FlErase" funktionene er kørt for at slette en hel sektor fra adresse 0. Derefter skrives indholdet fra adresse "i" ud på en serial monitor, som er integreret med Arduino softwaren. Efterfølgende kan det ses, at der er skrevet 56 på de første 255 pladser, og det der er skrevet vises på en serial monitor. Koden kan ses på Listing 8.13.

Resultater Resultater af testen kan ses på Figur 8.1, vor tal 255 reperesenterer den slettede byte inde i hukommelsesmatrisen, som ert slettet til `0xFF` og tal 56 repræsenterer tallet, som er programmeret til hukommelsesmodulet. Ud fra denne test kan der ses at det er muligt at slette flashmodulet sektorvis, skrive 8 bit af gangen og læse 8 bit af gangen.



Figur 8.1: Resultater af test0

8.3.2 Test 1

I denne test skal data inde i en sektor slettes (alle bytes sættes til `0xFF`), og bagefter programmeres 16 B af ny data. Efter både sletning og programmering skal indholdet af de 16 pladser læses og vises på en serial monitor.

Implementering

```
unsigned char run = 1;
void loop ()
{
  if(run){
    run = 0;
    unsigned char i;
    unsigned int adr=0;
    unsigned char dataRead[16],dataWrite[16],datalen=16;
    dataWrite[0] = 0;
    dataWrite[1] = 1;
    dataWrite[2] = 2;
```

```
    dataWrite[3] = 3;
    dataWrite[4] = 4;
    dataWrite[5] = 5;
    dataWrite[6] = 6;
    dataWrite[7] = 7;
    dataWrite[8] = 8;
    dataWrite[9] = 9;
    dataWrite[10] = 10;
    dataWrite[11] = 11;
    dataWrite[12] = 12;
    dataWrite[13] = 13;
    dataWrite[14] = 14;
    dataWrite[15] = 15;
    FlWriteReg(FCS);
    FlErase(FCS,adr);
    delay(1);
    FlReadEx(FCS,adr,dataRead,datalen);
     Serial.println("deleted page ");
    for(i=0;i<datalen;i++){
    Serial.println(dataRead[i]);
    }
    FlProgramEx(FCS,adr,dataWrite,datalen);
    FlReadEx(FCS,adr,dataRead,datalen);
    Serial.println("writed page ");
    for(i=0;i<datalen;i++) {
    Serial.println(dataRead[i]);
    }
 }
}
```

Listing 8.14: Test 0 kode

På Listing 8.14 kan der ses en implementering af test nummer 1. I starten af koden bliver "dataWrite" arrayet fyldt op med 16 tal, som går fra 0 til 15. Derefter bliver status- og konfigurationsregistrene nulstillet og den første sektor bliver slettet. Dernæst læses data og skrives ud på serial monitor. 16 B af data bliver programmeret til flashmodulet. Efter alle 16 B bliver programmeret, læses det der står på adresserne og skrives ud på en serial monitor.

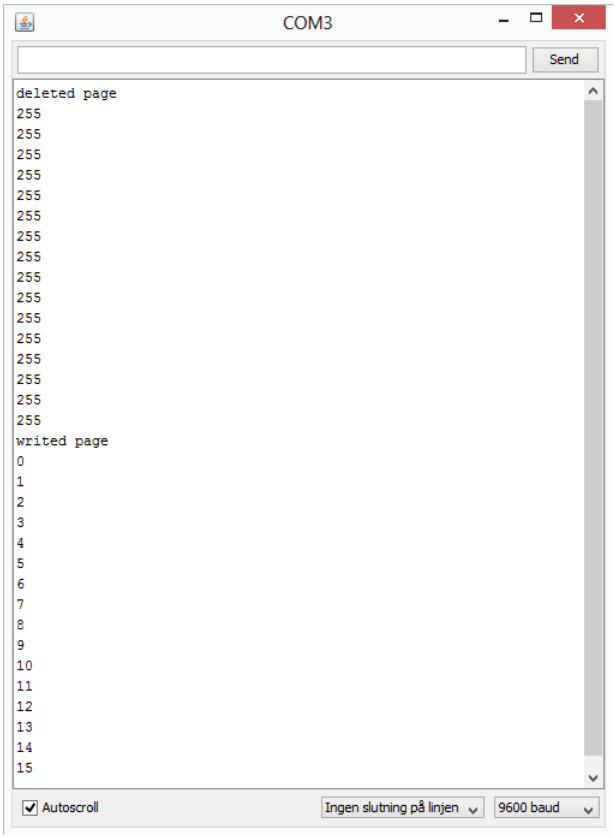
Resultater På Figur 8.2 kan det ses at alle bytes i "deleted page" er lige med 255, som betyder at alle bytes er slettet til `0xFF` . Efter skrive funktionen har kørt, vises der på serial monitor, det der stå på adresserne. Alle bytes i "programmed page" er dem, som er bestemt fra starten.

Der kan konkluderes at Krav 15 er overholdt. Samtidig kan der konkluderes at det er muligt at læse et antal af bytes ud af flashmodulet til et char array. Dette betyder at Krav 16 er overholdt. Det er også muligt at slette fra flashen sektorvis.

Test 2

I denne test er der testet en ny slettefunktion, som sletter alle sektorer i hukommelsesmatricen. På Listing 8.15 kan ses en ny slettefunktion "FlEraseAll", som kører en "FlErase" funktion på alle 255 sektorer i hukommelsesmatricen.

Implementering



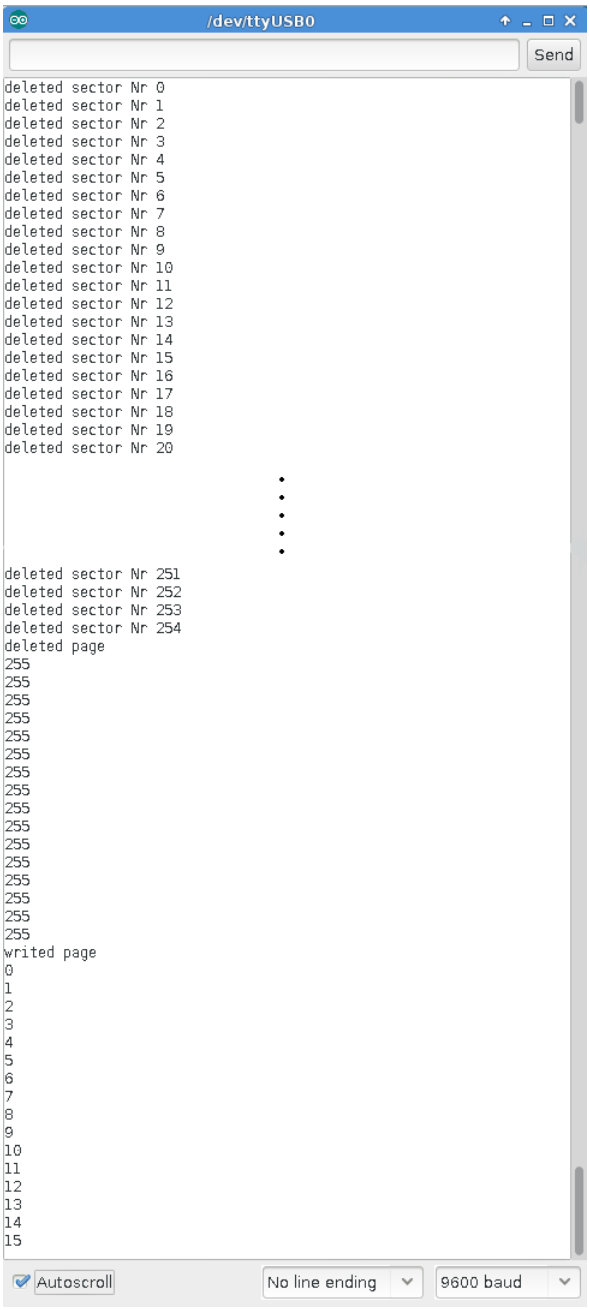
Figur 8.2: Resultater af test1

```
/**
 * Delete everything in flash memory array.
 * @param FCS chip select pin
 */
void FlEraseAll (unsigned char FCS)
{
  unsigned int i,adr;
  for (i = 0; i < 255 ; i++){
    adr = 0x0000FFFF * i;
    FlErase(FCS,adr);
    Serial.print("deleted sector Nr ");
    Serial.println(i);
  }
}
```

Listing 8.15: Erase all funktion

For at teste denne funktion, er der anvendt en tidligere kode fra Listing 8.14, hvor adressen var ændret til adressen på sektor nummer 3, som gør det muligt at undersøge om ikke kun den første sektor er slettet.

Resultater Resultater af testen kan ses på Figur 8.3. Der kan ses at samtlige sektorer i flashen er slettet og der er skrevet data til sektor nummer 3.



Figur 8.3: Resultater af test2

8.4 I²C

8.4.1 Test af I²C

Som beskrevet i testspecifikationen for I²C 6.4, testes I²C-bussen, med en unit test og en global test, for at minimere risikoen for fejl. Igennem unit testen og den globale test, vil hver funktion testes med forskellige inputs, se Tabel 8.6. Alle funktioner, testes i worst case scenarie, så vidt muligt, for at få en gennemgående test af hver funktion. Som det fremgår af testtabellen, Tabel 8.6, beskrives det forventede output, for hvert input og der vil efterfølgende være en delkonklusion, der samler op på alle test resultaterne. Derudover testes I²C-bussen,

som beskrevet i testspecifikationen 6.4, på en ATmega32. Efter den globale test og unit testen anvendes der en Arduino, der skal fungerer som en slaveenhed, for at teste kommunikationen imellem en master og en slave, hvor der bl.a. tjekkes om slaven kan sende et acknowledge til masteren.

8.4.2 Testresultater

Unit test

Testen udføres som beskrevet under testspecifikationen 6.4. Clockfrekvensen på ATmega32 testmodulet, er som standard indstillet til 1 MHz og testene blev derfor i første omgang udført ved denne clockfrekvens. Ved test af den første funktion `udelay`, viste det sig, at clockfrekvensen på 1 MHz, var for langsom og eksekveringstiden af koden derfor gav for meget delay til at clocke med 100 kHz. For at løse dette problem, blev clockfrekvensen indstillet til 8 MHz, hvilket gav mere præcise testresultater og hurtigere eksekvering af koden. Udover en hurtigere clockfrekvens, opstod der også nogen problemer mht. datatyper i koden, da datatypen int kun er 2 B stor, på ATmega32 mikroprocessoren. Dette gav problemer, da int-datatypen, blev brugt som en 4 B variable i koden og der måtte derfor i stedet anvendes unsigned long, de steder der var brug for 4 B datatyper.

Koden til testene findes på CD:

`code/i2c_avr/{testbench.c, unittests.c}`

og ses i Listing 8.16 og Listing 8.17.

```
1  #define F_CPU 8000000UL
2  #include <avr/io.h>
3  #include <util/delay.h>
4  #include "unittests.c"
5
6  /* Main */
7  int main (void)
8  {
9      unsigned char sw;
10     DDRD  = (1 << SDALINE) | (1 << SCLLINE); /* Output */
11     PORTD = (1 << SDALINE) | (1 << SCLLINE); /* High */
12
13 //     MCUCSR = (1<<JTD); /* Disable JTAG on PORTC */
14 //     MCUCSR = (1<<JTD);
15
16     DDRC  = 0x00; /* Input buttons */
17     PORTC = 0xFF; /* Pullup, because buttons are active LOW. */
18     DDRA  = 0xFF; /* Output LED's */
19     PORTA = 0x00;
20
21     for (;;) {
22         PORTA &= ~ 0xFE; /* Necessary hack. */
23         PORTA &= ~ 0x01; /* Don't know why. */
24
25         sw = ~ PINC;
26         if (sw & 0x01) {
27             /* ut_udelay(); */
28             /* ut_setsda(); */
29             /* ut_getsda(); */
30             /* ut_starttrans(); */
31             /* ut_stoptrans(); */
32             /* ut_readbit(); */
33             ut_writebit();
34             /* ut_readbyte(); */
35             /* ut_writebyte(); */
36             /* ut_i2c_transcieve(); */
37             /* ut_i2c_receive(); */
38             _delay_ms(1000);
```

```
        }
    }

    return 0;
}
```

Listing 8.16: Hoveprogram til unit tests – her ses `testbench.c`

```
#include "i2c.c"
const unsigned char SDALINE     = 0x00;
const unsigned char SCLLINE     = 0x01;
const unsigned char OWN_ADDR    = 0x21;

const unsigned int TEST_TIMEOUT = 2000;
const unsigned char TEST_ADDR_W = 0xAA; /* 0x55 + write */
const unsigned char TEST_ADDR_R = 0xAB; /* 0x55 + read  */

static inline void ut_udelay(void)
{
    unsigned char i;

    PORTD &= ~ 0x01;
    for (i = 0; i < 16; i++) {
        PORTD ^=  (1 << SDALINE);
        udelay(0); /* <- Unit test variable */
        PORTA |= 0x01;
    }
    PORTD &= ~ 0x01;
}

static inline void ut_setsda(void) /* For both setsda and setscl */
{
    PORTD &= ~ 0x7F; /* <- Initial value         */
    _delay_ms(1);
    setsda(-1, 1);  /* <- Unit test variables   */
    _delay_ms(1);
    PORTD &= ~ 0x7F; /* <- Back to initial value */
}

static inline void ut_getsda(void) /* For both getsda and getscl */
{
    setsda(0, HIGH);
    udelay(100);
    signed char ret = getsda(0);
    if (ret == HIGH)
        PORTA = 0x02;
    else
        PORTA = 0x01;
}

static inline void ut_starttrans(void)
{
    setsda(0, HIGH);
    setscl(1, HIGH);
    udelay(100);
    starttrans(0, 1);

    udelay(100);
    setsda(0, HIGH);
    setscl(1, HIGH);
}

static inline void ut_stoptrans(void)
{
    setsda(0, LOW);
```

```
58      setscl(1, LOW);
59      udelay(100);
60      stoptrans(0, 1);
61
62      udelay(100);
63      setsda(0, LOW);
64      setscl(1, LOW);
65  }
66
67  static inline void ut_readbit(void)
68  {
69      setsda(0, LOW);
70      setscl(1, LOW);
71
72      if (readbit(0, 1) == 0)
73          PORTA |= 0x01;
74      else
75          PORTA |= 0x02;
76  }
77
78  static inline void ut_writebit(void)
79  {
80      setsda(0, HIGH);
81      setscl(1, HIGH);
82      udelay(100);
83
84      writebit(0, 1, 0); /* <- Unit test variable. */
85  }
86
87  static inline void ut_readbyte(void)
88  {
89      setsda(0, HIGH);
90      setscl(1, HIGH);
91      udelay(100);
92
93      readbyte(0, 1);
94  }
95
96  static inline void ut_writebyte(void)
97  {
98      setsda(0, HIGH);
99      setscl(1, HIGH);
100     udelay(100);
101
102     if (writebyte(0, 1, 0xAA) == I2C_ACK)
103         PORTA = 0x01;
104     else
105         PORTA = 0x02;
106 }
107
108 static inline void ut_i2c_transcieve(void)
109 {
110     setsda(0, HIGH);
111     setscl(1, HIGH);
112     udelay(100);
113
114     /* writebyte() is modified
115      * in test 1 and 2 to
116      * automatically do an ACK
117      * after a byte is sent. */
118
119     /* Write tests */
120
121     // unsigned char data[1] = {0xCC};/* Test 1 + 3 */
122     // i2c_transceive (0, 1, TEST_ADDR_W, data, 1);
```

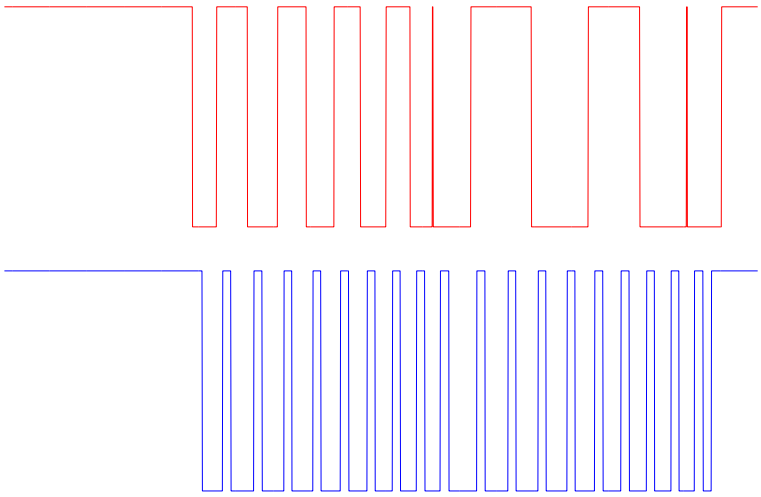
```
123
124     //   unsigned char data[3] = {0xCC, 0xBB, 0x4E}; /* Test 2 */
125     // i2c_transceive (0, 1, 0xAA, data, 3);
126
127     /* Read tests */
128
129     unsigned char data[2]; /* Test 4 */
130     i2c_transceive(0, 1, TEST_ADDR_R, data, 2);
131 }
132
133 static inline void ut_i2c_receive(void)
134 {
135 }
```

Listing 8.17: Hoveprogram til unit tests – her ses `unittests.c`

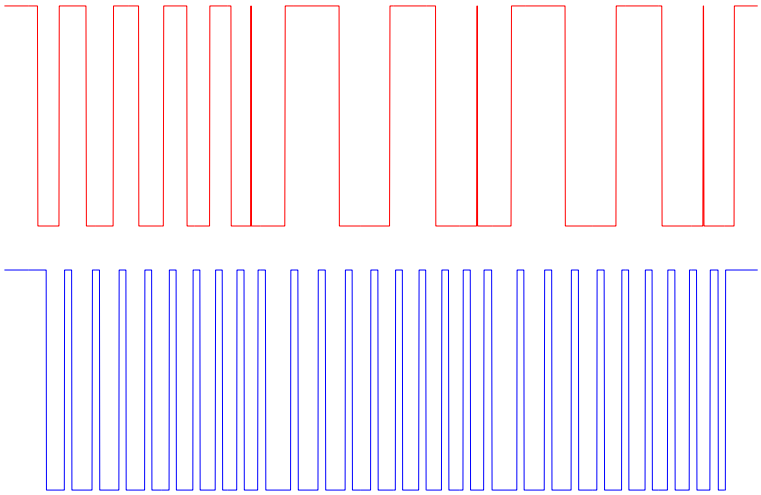
Funktion/test	Forventet output	Output
Hvis ikke andet nævnt, er `sda = 0, scl = 1`.		
`udelay(us)`		
`us = -1`	Delay på `0xFFFFFFFF` µs (> 30 s testes.)	Delay: 1,8 µs.
`us = 0`	Intet delay.	Delay: 736 ns.
`us = 1`	Delay på 1 µs.	Delay: 1,7 µs.
`us = 5`	Delay på 5 µs.	Delay: 5,6 µs.
`us = 0xFFFFFFFF`	Delay på `0xFFFFFFFF` µs. (> 30 s testes.)	Intet respons efter 30 s.
`setsda(sda, state)`		
`sda = 0, state = 0`	`sda`-pin på `PORTD` går lav.	Pin 0 går lav som forventet.
`sda = 0, state = 1`	`sda`-pin på `PORTD` går høj.	Pin 1 går høj som forventet.
`sda = 10, state = 1`	Ingen ændringer på `PORTD`.	Ingen pins ændrer sig.
`sda = -1, state = 1`	Ingen ændringer på `PORTD`.	Ingen pins ændrer sig.
`getsda(sda)`		
`PORTD(sda) = 0`	Status på `sda`-pin på `PORTD` tænder `LED 0` på testboard.	`LED 0` tænder.
`PORTD(sda) = 1`	Status på `sda`-pin på `PORTD` tænder `LED 1` på testboard.	`LED 1` tænder.
`starttrans(sda, scl)`		
`sda = 0, scl = 1`	`sda` går lav, mens `scl` er høj.	`sda` går lav 7,8 µs efter `scl`.
`stoptrans(sda, scl)`		
`sda = 0, scl = 1`	`sda` går høj, mens `scl` er høj.	`sda` går høj 5,2 µs efter `scl`
`readbit(sda, scl)`		
`PORTD(sda) = 0`	Funktionen returnerer 0, `LED 0` tænder.	`LED 0` tænder.
`PORTD(sda) = 1`	Funktionen returnerer 1, `LED 1` tænder.	`LED 1` tænder.
`writebit(sda, scl, bit`		
`bit = 1`	`scl = 0`, `sda = 1`. Efter 3 µs kommer en 4 µs puls på `scl`.	Efter 3,4 µs kommer en 4,2 µs puls på `scl`. Det giver en frekvens på $f = 1/(2 \cdot 3{,}2\text{E}{-9} + 4{,}2\text{E}{-9}) = 90{,}90\,\text{kHz}$.
`bit = 0`	`sda = 0` mens `scl`-puls kommer.	`sda` er lav mens der kommer `scl`-puls.
`bit = -1`	`sda = 1` mens `scl`-puls kommer.	`unsigned char bit -> signed char bit`. `sda` er nu lav, mens der kommer en `scl`-puls.

`readbyte(sda, scl)`		
`PORTD(sda) = 0`	`sda` kobles lav. På oscilloskopet ses, at otte pulse kommer på `scl`, mens `sda` er lav.	I `readbit()`, sættes `scl` lav før puls. Som forventet kommer der otte pulser på `scl`, mens `sda` er lav.
`PORTD(sda) = 1`	`sda` kobles høj. På oscilloskopet ses, at otte pulse kommer på `scl`, mens `sda` er høj.	`sda` er høj, mens der kommer otte pulser på `scl`.
`writebyte(sda, scl, c)`		
`c = 0xAA, PORTD(sda) = 1`	Otte `scl`-pulser kommer, med skiftevis `sda = 1` og `sda = 0`. Returnerer 1, da et ACK modtages. Der indsættes et delay inden der læses et ACK, for at have tid til manuelt at sætte `sda` lav. `LED 0` tændes, når ACK modtages.	Oscilloskopet viser, at `0xAA` clockes ind, og `LED 0` tændes.
`c = 0xAA, PORTD(sda) = 0`	Otte `scl`-pulser kommer, med skiftevis `sda = 1` og `sda = 0`. Returnerer 0, da et NACK modtages. `LED 1` tændes, når NACK modtages.	Oscilloskopet viser, at `0xAA` clockes ind, og `LED 1` tændes.
`i2c_transceive(sdaline, sclline, addr_rw, *data, cnt)`		
`addr_rw = 0xAA(write), data[1] = {0xCC}, cnt = 1`	En startbetingelse sendes. Adressen `0xAA` sendes først. `writebyte()` er modificeret til selv at ACKe, så et ACK modtages. Databyten `0xCC` sendes. En stopbetingelse sendes.	Det forventede modtages. Se Figur 8.4.
`addr_rw = 0xAA(write), data[2] = {0xCC, 0xCC}, cnt = 2`	Det samme som forrige, men med to databytes.	To datapakker modtages, som det ses på Figur 8.5
`addr_rw = 0xAA(write), data[1] = {0xCC}, cnt = 1`	En startbetingelse sendes. Adressen `0xAA` sendes først. `writebyte()` er ændret tilbage og giver et NACK og transaktionen slutter efter adressebyten. En stopbetingelse sendes.	`stoptrans()` sat til at starte med `sda = LOW`. `i2c_transceive()` er lavet om til at lave så der også sendes en stopbetingelse ved NACK. Se Figur 8.6.
`i2c_receive(sdaline, sclline, own_addr)`		

Tabel 8.6: Unit tests af I^{2}C-modul.



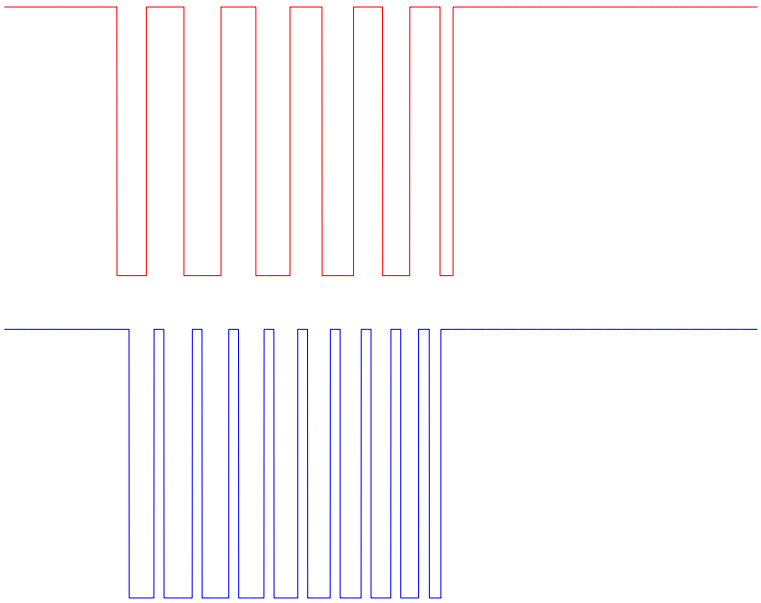
Figur 8.4: Unit test af `i2c_transceive()`.



Figur 8.5: Unit test af `i2c_transceive()` – test 2.

Test med Arduino som slave

Som nævnt i testspecifikationen, Afsnit 6.4, sættes en Arduino op som SPI slave, og videresender modtaget data til en PC, som viser dataene på skærmen. Koden der gør dette ses i Listing 8.18.



Figur 8.6: Unit test af `i2c_transceive()` – test 3.

```
#include <Wire.h>

void setup ()
{
    Wire.begin(0x55);
    Wire.onReceive(receiveEvent);
    Wire.onRequest(requestEvent);
    Serial.begin(9600);
}

void loop ()
{
    delay (100);
}

void receiveEvent(int howMany)
{
    Serial.println("Receiving...");
    while(1 < Wire.available()) {
        char c = Wire.read();
        Serial.print(c);
    }
    int x = Wire.read();
    Serial.println(x);
}

void requestEvent()
{
    Serial.println("Sending...");
    Wire.write((unsigned char) 0xCC);

    /* FIXME:
     * On the oscilloscope, no matter what byte
     * we send, the output is a repetition of the
     * address: 0xAB. */
}
```

Listing 8.18: Kode til Arduino som I^2C-slave.

Forud for testen blev følgende fejl rettet:

- Når der fra masteren læses fra SDA og SCL-pinsene, sættes pullup modstande på pinsene, foruden at sætte den til inputs (`DDRD &= ~ (1 << sda)`, `PORTD |= (1 << sda)` og lignende for `scl`).
- Dynamisk allokering – `malloc` og `free` – er erstattet med arrays.

Herefter blev testene fra testspecifikationen, Afsnit 6.4, udført.

Matchende adresse og ACK

1. Master: Send matchende adresse + write – ingen data sendes med.
   `0xAA = 0x55` + write.
2. Tjek ACK på skop.
   Arduinoen trækker et ACK.
3. Arduino: Print, hvis adresse matcher.
   Arduino kører funktionen `receiveEvent()`, og printer på skærmen når der modtages.
4. Master: Send andre adresser + write, og se, at disse ikke kommer igennem.
   Arduinoen ACK'er kun, på sin egen adresse, og printer kun på skærmen hvis egen adresse sendes.

Enkelt databyte

1. Master: Send matchende adresse + write + data (1 B).
   `0xAA + 0xCC` .
2. Tjek ACK på skop.
   Arduinoen trækker et ACK ved både adresse og data.
3. Arduino: Print hvis adresse matcher.
   Arduinoen printer intet på skærmen, selvom data ses på skopet.
4. Arduino: Print data modtaget.
   Arduino printer heller ikke data.

Flere databytes

1. Master: Send matchende adresse + write + data (2 B).
   `0xAA + 0xCC + 0xCC` .
2. Tjek ACK på skop.
   Arduinoen ACK'er på de adressen og første byte, men ikke på sidste byte.
3. Arduino: Print hvis adresse matcher.
   Der printes intet på skærmen.
4. Arduino: Print data modtaget.
   Der printes intet på skærmen.

5. Master: Der sendes flere bytes, og foretages samme tests.
   Arduinoen ACK'er stadig efter anden byte, og der sendes derfor ikke mere end de første to. Der printes intet på skærmen.

Send 1 B **til master**

1. Master: Send matchende adresse + read.
   `0xAB` = `0x55` + read.
2. Master: Clock 1 B ind fra Arduino.
   Der bliver clocket 8 bit og en ACK ind fra Arduinoen.
3. Tjek ACK på skop.
   Arduinoen trækker et ACK.
4. Tjek databyte (`0xCC`) på skop.
   Databyten matcher ikke med det forventede `0xCC` . I stedet sender Arduinoen adressen som databyte (`0xAB`). Dette sendes også hvis `0xCC` erstattes med `0xFF` .

Send 2 B **til master**

1. Master: Send matchende addresse.
   `0xAB` .
2. Master: Clock 2 B ind fra Arduino.
   Der clockes 8 bit + 8 bit = 16 bit ind. Masterkode er ændret til at trække et ACK efter hver byte.
3. Tjek ACK på skop.
   Arduinoen trækker et ACK efter adressen er modtaget. Masteren trækker et ACK efter hver modtage databyte.
4. Tjek databytes på skop.
   Første databyte er adressen `0xAB` , og anden byte er `0xFF` , selvom `0xCC` og `0xCC` forventes sendt fra Arduinoen.

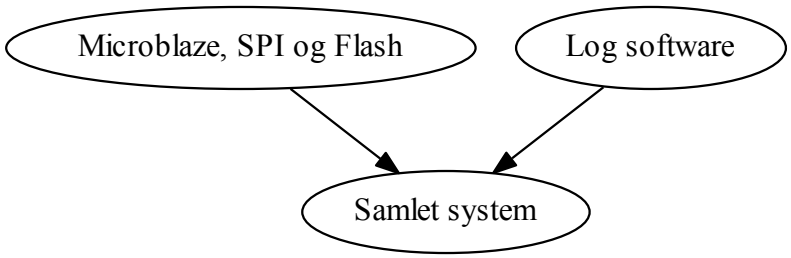
Delkonklusion

Som det fremgår af testresultaterne, er der flere at kommunikationstestene mellem slave- og masterenhederne, som ikke gav de forventede resultater. De uforventede testresultater, skyldes en dataoverførsels fejl i slaveenheden, hvilket ingår i en stor del af testresultaterne. For at udbedre disse fejl, skal der bruges noget tid på slaveenheden, da denne ikke fungere optimalt. I^2C-bussen har taget mere tid end planlagt og det har derfor været nødvendigt, at prioritere hvilke dele af I^2C-bussen og resten af projektet, der skulle lægges vægt på. I^2C-master receive funktionen og koden til den interne kommunikation til implementering på FPGA'en, er derfor ikke påbegyndt og mangler derfor også, før I^2C-bussen er klar til brug. Dette skyldes også nogen ikke forventede problemer, som beskrevet i starten af dette afsnit8.4.2. Problemerne opstod igennem udviklingen og der har desværre ikke været afsat nok tid til sådanne problemer. I fremtiden vil det også være en fordel, at afsætte mere tid til udviklingen af en I^2C-bus, når der skal foretages en unit test, da en sådan test er meget tidskrævende at lave.

9 Implementering

Dette afsnit beskriver implementering af de 3 systemer, udviklet under CASE:LOG udviklings afsnittet 7.1. Beskrivelsen omhandler implementeringen af loggen, Flash-modulet og SPI kommunikation. Under hele implementeringafsnittet anvendes en Spartan-6 på et Nexys3 udviklingsbræt. Dette er en konsekvens af de tidligere beskrevne pladsproblemer i forbindelse med Spartan-3 og MicroBlazens software.

Implementeringen vil foregå i 2 faser, først vil SPI og flash biblioteket blive sammensat og testet, derefter bliver log-modulet implementeret med det forrige og efterfølgende testet.



Figur 9.1: Implementeringsproces

9.1 Trin 1: Flash og SPI

9.1.1 Beskrivelse af kode

Den første implementering er mellem den eksterne flash og SPI-modulet. Flash IC'en kobles til SPI-modulet via de digitale I/O-porte på Nexys-3 udviklingsboardet. Internt i FPGA´en er SPI-modulet koblet til MicroBlazen.

Herefter sammensættes softwaren internt i MicroBlazen og da flash-biblioteket er kodet efter SPI-biblioteket, kan det nemt implementeres, således at det kompilerer uden fejl. Herefter testes funktionaliteten når delmodulerne er sammensat. En overordnet liste over testene kan ses nedenfor, disse er beskrevet i detaljer senere i dette kapitel:

- Test af læse funktion.
- Test af status register skriv.

- Test af slet funktion.
- Test af program funktion.

Til implementeringen er der udviklet en funktion til håndtering af Chip Select. Denne funktion (`digitalWrite()`), kan ses i Listing 9.1.

```
1  u8 digitalWrite(XIOModule *InstancePtr,unsigned channel,u8 pin, u8 value)
2  {
3    u32 pinOut,pinIn;
4    pinIn = XIOModule_DiscreteRead(InstancePtr,channel);
5    pinOut = 0x0001 << pin;
6    if (value == 0) pinOut = ~pinOut | ~pin;
7    if (value == 1) pinOut =  pinOut |  pin;
8    else return 1;
9    XIOModule_DiscreteWrite(InstancePtr,channel,pinOut);
10 }
```

Listing 9.1: Test 0 kode

digitalWrite Funktionen "digitalWrite" tager imod tre parametre: "channel", "pin" og "value". "channel" beskriver hvilken GPO port, der skal opereres på , som kan være 1, 2, 3 eller 4. Parameteren "pin" beskriver hvilken pin der skrives til og kan have værdien 0 til 31.Parameteren "value" beskriver den værdi, der skal skrives til outputtet, værdien kan enten være høj eller lav. Funktionen "digitalWrite" har til at formål at bestemme hvilken pin, der skal skrives til, samt hvilken GPO kanal der skal benyttes. Ydermere bestemmes output pinen ud fra om "value" er høj eller lav.

9.1.2 Implementeringstests

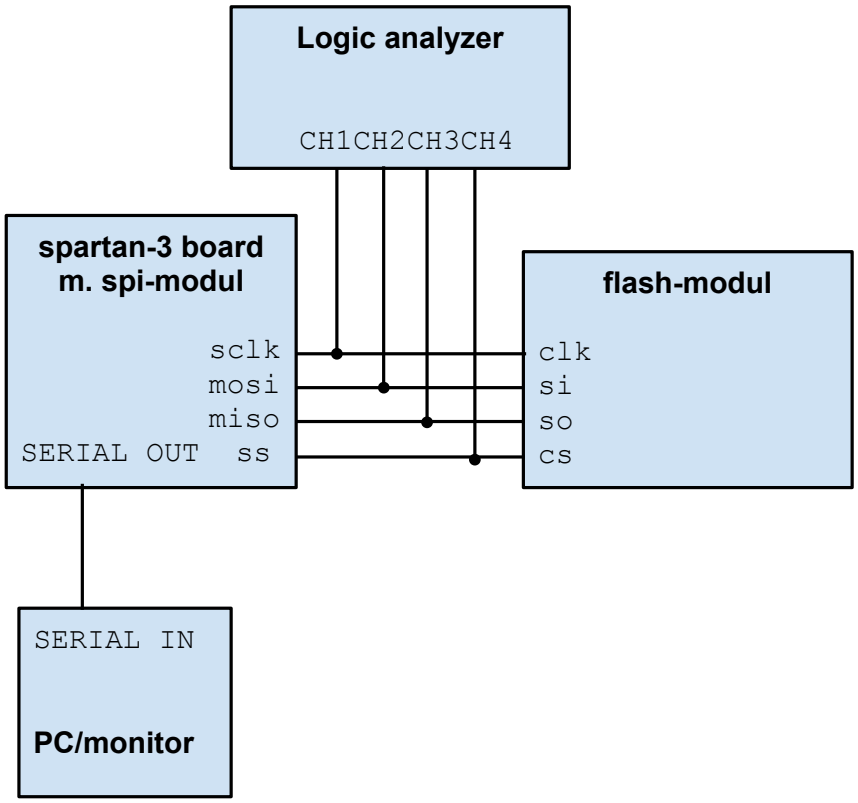
Ved anvendelsen af Spartan-3 udviklingsboardet konstateres det ved implementeringen, at der er pladsmangel når softwaren kompileres. Derfor udføres følgende to tests "test 1" og "test 2", udelukkende med overvågning på en Logic Analyzer. Dermed er det fravalgt at anvende RS-232 debugging i disse, dvs. "putnum" og "print" funktioner. Outputtet aflæses mellem flashmodulet og SPI-modulet, på de digitale I/O-ben på Spartan-3 boardet. På Logic Analyzeren blev der kigget på sclk, slave in, slave out og chip select. I de efterfølgende tests anvendes et Spartan-6 board, hvorfor der også her debugges med RS-232. Denne implementeringstest blev afbrudt, da kommunikationen ind og ud af flash-modulet, blev utilregnelig i forhold til de første fire testopstillinger. Derfor er der i det følgende vist disse fire tests, som gruppen nåede at udføre inden den nævnte kommunikationstilstand.

Test 1 – Read funktion

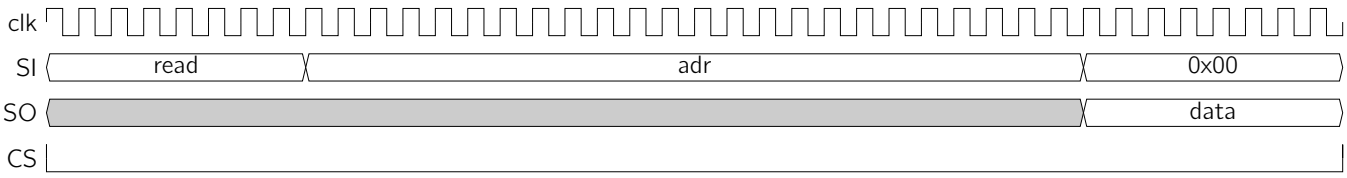
I den første test bliver læsefunktionen testet sammen med "digitalWrite" funktionen til chip select. Formålet er at der skal læses "FF" på en bestemt adresse, en adresse som det vides på forhånd er tom. På Figur 9.3 ses et timing diagram af det forventede resultat.

Forventet resultat Det forventes at mosi sender opcode "read" efterfulgt af tre byte med adresser. Flashen skriver derefter et output på 255 i en byte. 255 er dataen på en tom plads i flashen. Samtidigt med denne byte sender master en byte på 0. Dette vises på 9.3.

Resultat På Figur 9.4 ses resultat af testen, som resultat fra Logic Analyzeren "Agilent 54621D", hvor linje 1 er sclk, linje 2 er mosi, linje 3 er chip select og linje 4 er miso. Som det ses på Figur 9.4 bliver der som forventet læst "FF".



Figur 9.2: Testopstilling til den følgende implementering.



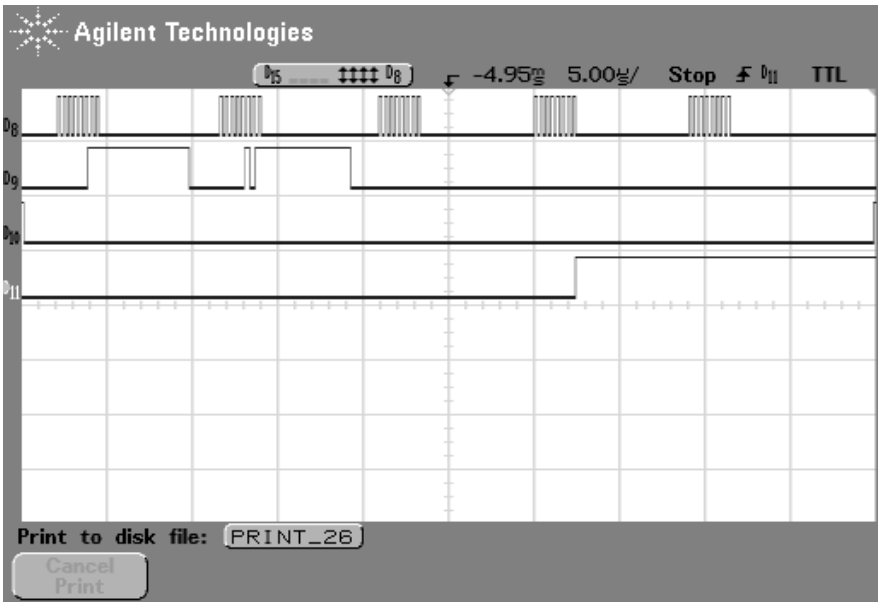
Figur 9.3: Timing diagram af forventet output

Test 2 – Read status register

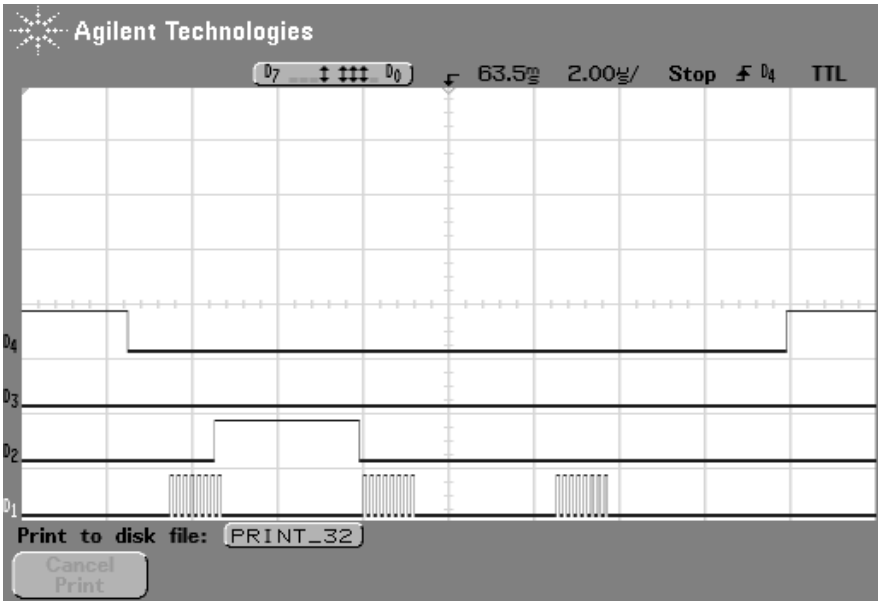
Under denne test afvikles s25WriteReg funktionen. Denne funktion afvikler også s25ReadStatReg funktionen. Testen er implementeret ved en uendelig for-løkke, hvori der er placeret en s25WriteReg funktion.

Forventet resultat Det forventes at SPI-master sender en opcode "writeReg", efterfulgt af to bytes som nulstiller henholdsvis statusregister og konfigurationsregister. I næste pakke forventes at funktion s25ReadStatReg skal afvikles og dermed skabe det nødvendige delay, som det kræves før WIP-benet (write in progress) går lav efter endt skrivning.

Resultater På Figur 9.5 kan der ses en s25WriteReg funktion, hvor der sendt opcode " `0x01` " efterfulgt af to bytes med " `0x00` ".



Figur 9.4: Resultat på logic analyser, som viser en læseinstruktion med adresse og svar fra slaven. Bemærk at slaven svarer med data = 255, som er indholdet på en tom plads.

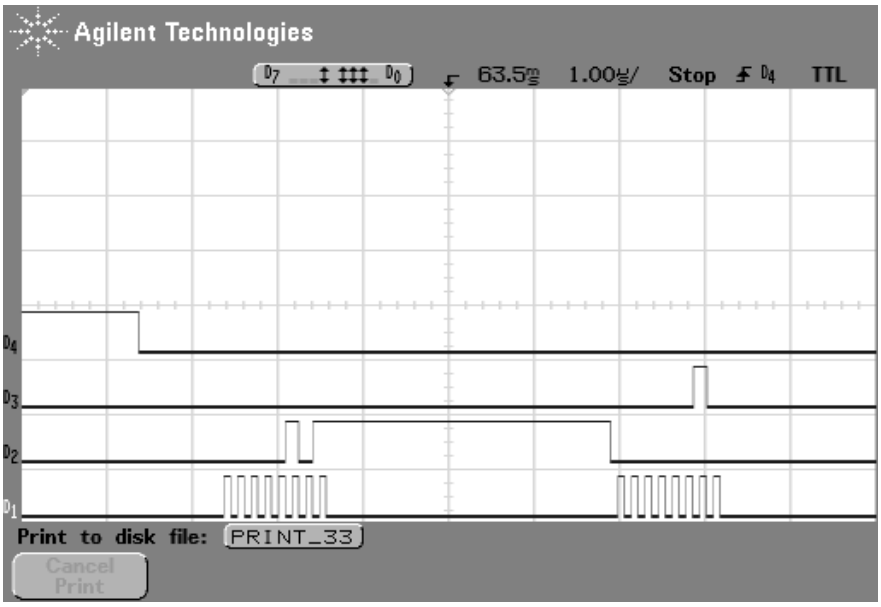


Figur 9.5: s25WriteReg funktion

På Figur 9.6 kan der ses en s25ReadStatReg funktion, som afvikles med opcode " `0x05` ", derefter aflæses indeholdet af statusregisteret indtil dette bliver `0x00` .

Udskiftining af udviklingsboard

For at kunne udføre en mere effektiv debugging under den følgende implementering, udskiftes udviklingsboardet under implementeringen med en nyere model. Her anvendes et Nexys-3 board med en Spartan-6 FPGA. På denne FPGA kan MicroBlazen implementeres således, at den udviklede software kan fylde de 64 KiB, som også



Figur 9.6: s25ReadStatReg funktion

var forventeligt at implementere på en Spartan-3 MicroBlaze. Da dette ikke kunne lade sig gøre, blev Spartan-3 hermed udskiftet og Spartan-6, vil blive anvendt i de følgende implementeringstests.

Test 3 – slet og skriv funktioner

Under denne test, testes de nye slet og skriv funktioner. Fremgangsmåden er først at slette en byte fra et memory-array, derefter tjekke at der står `0xFF` på denne plads, ved hjælp af read funktionen. MicroBlazens indbyggede print-funktion skriver resultatet på en serial monitor. Det næste skridt i testen, er at skrive et tal ved hjælp af skrivefunktionen og tjekke om dette tal, skrives korrekt ved brug af samme read-funktion, som overvåges på serial monitor.

Gennemgang af testkode Den anvendte kode til denne test er vist som Listing 9.2. Først er der kørt en slette kommando på adressen `0x00050000` . Alle skrive kommandoer afvikler en statusregister funktion. Statusregisterfunktionen er i denne test, derfor sat til at printe indholdet af statusregisteret ud på en serial monitor, når denne står i sit delay-loop. Når den hopper ud af delay-loopet, læses der fra flash-modulet og den læste data printes på en serial monitor. Næste procedure i koden er at programmere en byte til memory-arrayet på en specifik adresse og derefter læse fra denne adresse, for at printe det læste data til serial monitor. Resultatet er vist i Listing 9.3.

```
int main (void)
{
  XIOModule_Initialize(module1, XPAR_IOMODULE_0_DEVICE_ID);
  XIOModule_Start(module1);
  print("XIOModule started");
  s25WriteReg(module1,FCS);

  for(;;)
  {
    u32 adr=0x00050000;
    u8 data = 0x1F,red;

    s25Erase(module1,FCS,adr,data);
```

```
    red = s25Read(module1,FCS,adr,data);
    print("Read after erase ");
    putnum((int)red);


    s25Program(module1,FCS,adr,data);
    red = s25Read(module1,FCS,adr,data);
    print("Read the written data ");
    putnum((int)red);
  }
}
```

Listing 9.2: Test 3 kode

Resultater Resultatet er som forventet, at s25ReadStatReg skaber et delay og s25FlErase kan slette en adresse. Sletning af sektor tager længere tid end skrivning. Det er hermed også konstateret at read-funktionen læser det ønskede data, efter dette data netop er skrevet ind i flashen.

```
XIOModule started

statusreg contains  = 00000000
statusreg contains  = 00000002
statusreg contains  = 00000004
statusreg contains  = 00000008
statusreg contains  = 00000010
statusreg contains  = 00000020
statusreg contains  = 00000040
statusreg contains  = 00000000
Read after erase    000000ff

statusreg contains  = 00000002
statusreg contains  = 00000004
statusreg contains  = 00000000
Read the written data   0000001F
```

Listing 9.3: Test 3 resultater

Test 3.2 – forskellig data skrives og aflæses efterfølgende

Efterfølgende udvidedes denne test med forskellige data som inputs til flashen. Testkoden afvikles med forskellige data, og de efterfølgende outputs fra s25Read-funktionen aflæses.

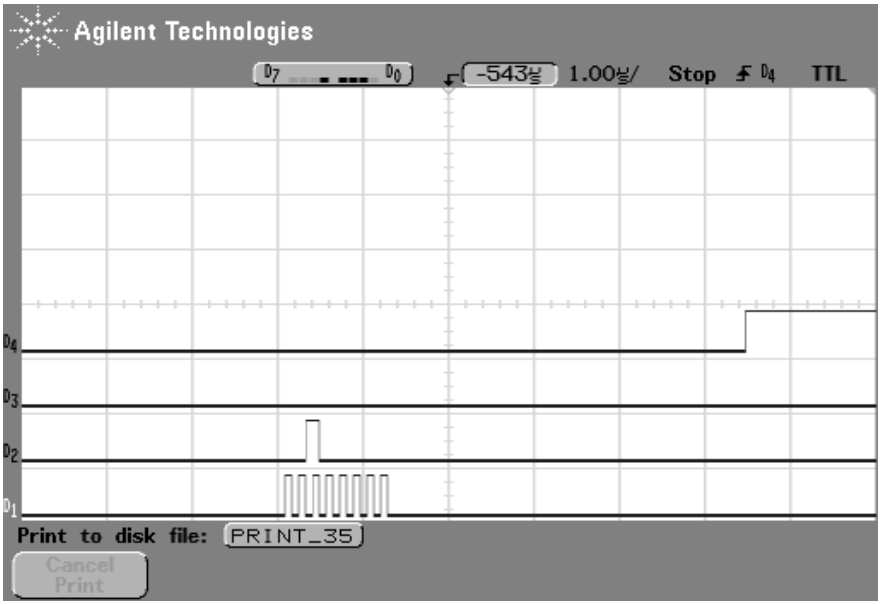
Resultater Prints fra s25Read kan ses i Tabel 9.1. Nogle af disse prints fra s25Read er som forventet, mens andre ikke er som forventet. Dette kan også ses på Listing 9.3, hvor statusregisteret skriver andre værdier end de forventede, mens selve funktionen virker, fordi s25Read læser `0x00` korrekt. Ud fra dette er det svært at drage en færdig konklusion. Derfor vælges det at ændre sætte SPI-hastigheden op i den efterfølgende test.

Test 3.3 – SPI-hastighed 50 MHz

Clockfrekvensen på SPI-modulet blev ændret til 50 Mhz. Samtidigt anvendtes logic analyzeren til at undersøge den overførte byte. Testkoden blev afviklet med overførsel af en byte indeholdende `0x20` .

Forventet resultat Dette skulle give en forventet output på `0x20` på både logic analyzer og serial monitor på PC.

Resultat På Figur 9.7 ses resultatet fra logic analyzeren. Resultatet er som forventet, der skrives `0x20` til flashen. Under denne test var det ikke muligt, at finde pakken hvori der blev skrevet tilbage fra flashen. Dette var besværliggjort pga. den høje hastighed, som blev anvendt i forhold til den tidsmæssige opløsning på logic analyzeren. På serial monitoren blev der i dette tilfælde printet samme resultat fra s25Read funktionen som i test, UnderAfsnit 9.1.2. Resultatet var stadig `1` , hvilket kan tyde på softwarefejl i forbindelse med read-funktionen. Derfor vælges det at undersøge situationen nærmere i Test 4, UnderAfsnit 9.1.2.



Figur 9.7: Input til flash med forventet data `0x20`

Forventet output	output
0	0
1	1
F	F
10	1
1F	1F
20	1
2F	1F
30	2

Tabel 9.1: Sammenligning af forventet output og output af s25Read funktion

Test 4

For at undersøge om der kan findes et mønster, i fejlene i de printede data fra s25Read, vælges det at skrive alle kombinationer af otte bit, sekventielt ind i flashen. Derefter aflæses disse med s25Read-funktionen. Testen udføres med en for-løkke, hvor data inkrementerer fra 0 til 255.

Forventet resultat Den ønskede overførsel af de inkrementerede 256 pakker, som skal skrives til flashen enkeltvis og læses igen enkeltvis, er alle tal i inkrementeret rækkefølge fra 0 til 255.

Implementering Denne test blev implementeret på den samme måde som test 3, udover at der i denne test anvendes en for-løkke. Dette loop kører fra 0 til 255 og ses i linje 25 på Listing 9.4.

```
1  #include "platform.h"
2  #include "xparameters.h"
3  #include "xbasic_types.h"
4  #include "xiomodule.h"
5  #include "S25FL128S.h"
6
7  u8 FCS = 31;       // chip select pin
8  XIOModule module1;
9  void putnum(int);
10 void print(char *str);
11
12 int main (void)
13 {
14   init_platform();
15   spiInit(SPI_MODE0|SPI_CLKDIV0);
16   XIOModule_Initialize(&module1, XPAR_IOMODULE_0_DEVICE_ID);
17   XIOModule_Start(&module1);
18   print("\nXIOModule started\n");
19
20   u8 red;
21   u32 adr=0x00090000;
22
23   int i =0;
24
25     for ( i = 0; i < 255; i++)
26     {
27     u8 data = i;
28
29     s25WriteReg(&module1,FCS);
30     s25Erase(&module1,FCS,adr);            // Erases data at the address adr
31     red = s25Read(&module1,FCS,adr);       // read page of data at adress adr
32      print("data   ");
33      putnum(data);
34     s25Program(&module1,FCS,adr,data);
35     red = s25Read(&module1,FCS,adr);       // read page of data at adress adr
36      blue = (int)red;
37      print("\nRead the written data   ");
38      putnum(blue);                         //print page to serial port
39      print("\n");
40   }
41   cleanup_platform();
42   return 0;
43 }
```

Listing 9.4: Kode anvendt til test af skrivning af data 0 til 255 som for-løkke.

Resultater Det ses i Listing 9.5, at der læses flere pakker med det samme indhold, hvilket tyder på timing-problemer mellem SPI-modul og flashmodul eller mellem softwarefunktioner i flashmodulet eller mellem software og hardware i flashmodulet.

```
1  XIOModule started
2  data   00000000 - Read the written data   00000000
3  data   00000001 - Read the written data   00000000
4  data   00000002 - Read the written data   00000001
5  data   00000003 - Read the written data   00000001
6  data   00000004 - Read the written data   00000002
7  data   00000005 - Read the written data   00000002
8  data   00000006 - Read the written data   00000003
9  data   00000007 - Read the written data   00000003
10 data   00000008 - Read the written data   00000004
```

```
data   00000009 - Read the written data   00000004
data   0000000a - Read the written data   00000005
data   0000000b - Read the written data   00000005
data   0000000c - Read the written data   00000006
data   0000000d - Read the written data   00000006
data   0000000e - Read the written data   00000007
data   0000000f - Read the written data   00000007
data   00000010 - Read the written data   00000008
data   00000011 - Read the written data   00000008
data   00000012 - Read the written data   00000009
data   00000013 - Read the written data   00000009
data   00000014 - Read the written data   0000000a
data   00000015 - Read the written data   0000000a
data   00000016 - Read the written data   0000000b
data   00000017 - Read the written data   0000000b
data   00000018 - Read the written data   00000000
data   00000019 - Read the written data   0000000c
data   0000001a - Read the written data   0000000d
data   0000001b - Read the written data   0000000d
data   0000001c - Read the written data   0000000e
data   0000001d - Read the written data   0000000e
data   0000001e - Read the written data   0000000f
data   0000001f - Read the written data   0000000f
data   00000020 - Read the written data   00000010
data   00000021 - Read the written data   00000010
data   00000022 - Read the written data   00000011
data   00000023 - Read the written data   00000011
data   00000024 - Read the written data   00000012
data   00000025 - Read the written data   00000012
data   00000026 - Read the written data   000000ff
data   00000027 - Read the written data   00000013
data   00000028 - Read the written data   00000014
data   00000029 - Read the written data   00000014
data   0000002a - Read the written data   00000015
data   0000002b - Read the written data   00000015
data   0000002c - Read the written data   00000016
data   0000002d - Read the written data   00000016
data   0000002e - Read the written data   00000017
data   0000002f - Read the written data   00000017
data   00000030 - Read the written data   00000018
data   00000031 - Read the written data   000000ff
data   00000032 - Read the written data   00000019
data   00000033 - Read the written data   00000019
data   00000034 - Read the written data   0000001a
data   00000035 - Read the written data   0000001a
data   00000036 - Read the written data   0000001b
data   00000037 - Read the written data   0000001b
data   00000038 - Read the written data   0000001c
data   00000039 - Read the written data   0000001c
data   0000003a - Read the written data   0000001d
data   0000003b - Read the written data   0000001d
data   0000003c - Read the written data   0000001e
data   0000003d - Read the written data   0000001e
data   0000003e - Read the written data   0000001f
data   0000003f - Read the written data   0000001f
data   00000040 - Read the written data   00000020
data   00000041 - Read the written data   00000020
data   00000042 - Read the written data   00000021
data   00000043 - Read the written data   00000021
data   00000044 - Read the written data   000000ff
data   00000045 - Read the written data   00000022
data   00000046 - Read the written data   00000023
data   00000047 - Read the written data   00000023
data   00000048 - Read the written data   00000024
data   00000049 - Read the written data   00000024
```

```
data   0000004a - Read the written data   000000ff
data   0000004b - Read the written data   00000025
data   0000004c - Read the written data   00000026
data   0000004d - Read the written data   00000026
data   0000004e - Read the written data   00000027
data   0000004f - Read the written data   00000027
data   00000050 - Read the written data   00000028
data   00000051 - Read the written data   00000028
data   00000052 - Read the written data   00000029
data   00000053 - Read the written data   00000029
data   00000054 - Read the written data   0000002a
data   00000055 - Read the written data   0000002a
data   00000056 - Read the written data   0000002b
data   00000057 - Read the written data   0000002b
data   00000058 - Read the written data   0000002c
data   00000059 - Read the written data   0000002c
data   0000005a - Read the written data   0000002d
data   0000005b - Read the written data   0000002d
data   0000005c - Read the written data   0000002e
data   0000005d - Read the written data   000000ff
data   0000005e - Read the written data   0000002f
data   0000005f - Read the written data   0000002f
data   00000060 - Read the written data   000000ff
data   00000061 - Read the written data   000000ff
data   00000062 - Read the written data   00000031
data   00000063 - Read the written data   00000031
data   00000064 - Read the written data   00000032
data   00000065 - Read the written data   00000032
data   00000066 - Read the written data   00000033
data   00000067 - Read the written data   00000033
data   00000068 - Read the written data   00000034
data   00000069 - Read the written data   00000034
data   0000006a - Read the written data   00000035
data   0000006b - Read the written data   00000035
data   0000006c - Read the written data   00000036
data   0000006d - Read the written data   00000036
data   0000006e - Read the written data   00000037
data   0000006f - Read the written data   00000037
data   00000070 - Read the written data   00000038
data   00000071 - Read the written data   00000038
data   00000072 - Read the written data   00000039
data   00000073 - Read the written data   00000039
data   00000074 - Read the written data   0000003a
data   00000075 - Read the written data   0000003a
data   00000076 - Read the written data   0000003b
data   00000077 - Read the written data   0000003b
data   00000078 - Read the written data   0000003c
data   00000079 - Read the written data   000000ff
data   0000007a - Read the written data   0000003d
data   0000007b - Read the written data   0000003d
data   0000007c - Read the written data   0000003e
data   0000007d - Read the written data   0000003e
data   0000007e - Read the written data   0000003f
data   0000007f - Read the written data   00000000
data   00000080 - Read the written data   000000c0
data   00000081 - Read the written data   000000c0
data   00000082 - Read the written data   000000c1
data   00000083 - Read the written data   000000c1
data   00000084 - Read the written data   000000c2
data   00000085 - Read the written data   000000c2
data   00000086 - Read the written data   000000c3
data   00000087 - Read the written data   000000c3
data   00000088 - Read the written data   000000c4
data   00000089 - Read the written data   000000c4
data   0000008a - Read the written data   000000c5
```

```
data   0000008b - Read the written data   000000c5
data   0000008c - Read the written data   000000c6
data   0000008d - Read the written data   000000c6
data   0000008e - Read the written data   000000c7
data   0000008f - Read the written data   000000c7
data   00000090 - Read the written data   000000c8
data   00000091 - Read the written data   000000c8
data   00000092 - Read the written data   000000c9
data   00000093 - Read the written data   000000c9
data   00000094 - Read the written data   000000ca
data   00000095 - Read the written data   00000000
data   00000096 - Read the written data   000000cb
data   00000097 - Read the written data   000000cb
data   00000098 - Read the written data   000000ff
data   00000099 - Read the written data   000000cc
data   0000009a - Read the written data   000000ff
data   0000009b - Read the written data   000000cd
data   0000009c - Read the written data   000000ce
data   0000009d - Read the written data   000000ce
data   0000009e - Read the written data   000000cf
data   0000009f - Read the written data   000000cf
data   000000a0 - Read the written data   000000d0
data   000000a1 - Read the written data   000000d1
data   000000a2 - Read the written data   000000d2
data   000000a3 - Read the written data   000000d3
data   000000a4 - Read the written data   000000d4
data   000000a5 - Read the written data   000000d5
data   000000a6 - Read the written data   000000d6
data   000000a7 - Read the written data   000000ff
data   000000a8 - Read the written data   000000d8
data   000000a9 - Read the written data   000000d9
data   000000aa - Read the written data   000000da
data   000000ab - Read the written data   000000db
data   000000ac - Read the written data   000000dc
data   000000ad - Read the written data   00000000
data   000000ae - Read the written data   000000de
data   000000af - Read the written data   000000df
data   000000b0 - Read the written data   000000d0
data   000000b1 - Read the written data   000000d1
data   000000b2 - Read the written data   000000d2
data   000000b3 - Read the written data   000000ff
data   000000b4 - Read the written data   000000d4
data   000000b5 - Read the written data   000000d5
data   000000b6 - Read the written data   000000d6
data   000000b7 - Read the written data   000000d7
data   000000b8 - Read the written data   000000d8
data   000000b9 - Read the written data   000000d9
data   000000ba - Read the written data   000000da
data   000000bb - Read the written data   000000db
data   000000bc - Read the written data   000000dc
data   000000bd - Read the written data   000000dd
data   000000be - Read the written data   000000de
data   000000bf - Read the written data   000000df
data   000000c0 - Read the written data   000000e0
data   000000c1 - Read the written data   000000e0
data   000000c2 - Read the written data   000000e1
data   000000c3 - Read the written data   000000e1
data   000000c4 - Read the written data   000000e2
data   000000c5 - Read the written data   000000e2
data   000000c6 - Read the written data   000000e3
data   000000c7 - Read the written data   000000e3
data   000000c8 - Read the written data   000000e4
data   000000c9 - Read the written data   000000e4
data   000000ca - Read the written data   000000e5
data   000000cb - Read the written data   000000e5
```

```
206 data   000000cc - Read the written data   000000e6
207 data   000000cd - Read the written data   000000e6
208 data   000000ce - Read the written data   000000e7
209 data   000000cf - Read the written data   000000e7
210 data   000000d0 - Read the written data   000000e8
211 data   000000d1 - Read the written data   000000e8
212 data   000000d2 - Read the written data   000000e9
213 data   000000d3 - Read the written data   000000eb
214 data   000000d4 - Read the written data   000000ec
215 data   000000d5 - Read the written data   000000ff
216 data   000000d6 - Read the written data   000000ee
217 data   000000d7 - Read the written data   000000ef
218 data   000000d8 - Read the written data   000000ff
219 data   000000d9 - Read the written data   000000ec
220 data   000000da - Read the written data   000000ea
221 data   000000db - Read the written data   000000ed
222 data   000000dc - Read the written data   000000ee
223 data   000000dd - Read the written data   000000ed
224 data   000000de - Read the written data   000000ee
225 data   000000df - Read the written data   000000ef
226 data   000000e0 - Read the written data   000000f0
227 data   000000e1 - Read the written data   000000ff
228 data   000000e2 - Read the written data   000000f1
229 data   000000e3 - Read the written data   000000f1
230 data   000000e4 - Read the written data   000000f2
231 data   000000e5 - Read the written data   000000f2
232 data   000000e6 - Read the written data   000000f3
233 data   000000e7 - Read the written data   000000f3
234 data   000000e8 - Read the written data   000000f4
235 data   000000e9 - Read the written data   000000f4
236 data   000000ea - Read the written data   000000f5
237 data   000000eb - Read the written data   000000f5
238 data   000000ec - Read the written data   000000f6
239 data   000000ed - Read the written data   000000f6
240 data   000000ee - Read the written data   000000f7
241 data   000000ef - Read the written data   000000f7
242 data   000000f0 - Read the written data   000000f8
243 data   000000f1 - Read the written data   000000f8
244 data   000000f2 - Read the written data   000000f9
245 data   000000f3 - Read the written data   000000f9
246 data   000000f4 - Read the written data   000000fa
247 data   000000f5 - Read the written data   000000fa
248 data   000000f6 - Read the written data   000000fb
249 data   000000f7 - Read the written data   000000fb
250 data   000000f8 - Read the written data   000000fc
251 data   000000f9 - Read the written data   000000fc
252 data   000000fa - Read the written data   000000fd
253 data   000000fb - Read the written data   000000fd
254 data   000000fc - Read the written data   000000fe
255 data   000000fd - Read the written data   000000fe
256 data   000000fe - Read the written data   000000ff
```

Listing 9.5: Resultat af testen i 9.4. Af denne test fremgår det at der sker en forsinkelse i forbindelse med læsningen af flashcellerne når denne kombination af kommandoer udføres. Bemærk desuden at der sker et spring fra `0x3F` til `0xC0` på outputtet fra s25Read-funktionen.

Ikke testede funktioner

Flashbiblioteket har flere funktioner end der er testet i dette afsnit. De funktioner der ikke vil blive testet i forbindelse med implementeringen er:

s25ProgramEx Via denne funktion kan en hel page i flashmodulet skrives, hvilket vil sige at der kan skrives 256 B på en gang.

s25ReadEx Modsat ovenstående funktion, læser denne en hel page fra flashmodulet ad gangen.

s25EraseAll Denne funktion sletter alt hvad der står i flashmodulet.

Uforudsete udfordringer med flash-modul

Herefter blev al kommunikation med flash-modulet utilregnelig i forhold til disse udførte og beskrevne test 1 til test 4. Der blev derfor efterfølgende forsøgt at skrive til flashen med samme opstilling som i test 1 til test 4, uden at opnå de forventede resultater. Det vil sige at intet var som tidligere testet og der blev derfor af tidsmæssige årsager, vurderet at chippen ikke kunne anvendes og der blev derfor truffet valg om at udføre resten af implementeringen på anden vis end tidligere planlagt. I forbindelse med implementeringen af log-modulet, SPI og flash, var det indtil dette stadie planlagt at teste ud fra resultater for inputs og outputs på flash-modulet. Denne test blev på grund af situationen valgt at undlade, fordi de enkelte funktionaliter for alle tre delmoduler allerede er testet tidligere og fordi det pga. situationen ikke var muligt.

9.2 Trin 2: Flash, SPI og LOG

9.2.1 Beskrivelse af kode

I den næste implementeringsfase skal flash og log-koden implementeres. Da der har været problemer med at implementere SPI og flash, skrives der lidt om på nogle funktioner i log-biblioteket, så de printer ud på en skærm i stedet for at skrive i flash.

Desuden er det bestemt, at logbeskeder læses bagud i flashen, for at få nyeste beskeder først. Det har medført omskrivning af readFromFlash-funktionen, der nu kan ses på Listing 9.6.

Denne funktion bruger s25Read-funktionen fra flash biblioteket og er implementeret ved hjælp af en for-løkke, hvilket begynder i den nuværende adresse, læser hele flashen igennem baglæns, og stopper når startplaceringen nåes igen.

```
1  char readFromFlash(XIOModule *InstancePtr, unsigned char FCS
2                     unsigned long addr, unsigned char *data)
3  {
4    unsigned long i;
5    unsigned char a;
6
7    for (i = addr; i > addr - 16; i--) {
8      if (addr == 0x0000FFFF) addr = 0x00FDFFFF;
9      data[addr - i] = s25Read (InstancePtr, FCS, addr);
10   }
11   return 1;
12 }
```

Listing 9.6: readFromFlash funktion

For at bevare flashens position, selv hvis strømmen går, er det valgt hele tiden at gemme nuværende position i første sektor af flashen. Funktionerne “read_flash_pos” og “ write_flash_pos” sørger for at læse og skrive denne position, og ses på Listing 9.7.

Funktionen "read_flash_pos" retunerer en unsigned long, som udgør den nuværende position. Da første sektor i flashen nu bruges til at bevare positionen i flashen, er det aktive dataområde for flashen fra `0x00010000` til `0x00FDFFFF` . Både read_flash_pos og write_flash_pos wrapper derfor rundt ved disse yderpunkter.

Funktionen "write_flash_pos" først sletter den første sektor og programmerer derefter nuværende position på de første tre adresser i flashen – tre adresser, da alle adresser/positioner kan beskrives med 3 B.

```
/**
 * Read current position/index from the
 * first 3 bytes of the flash.
 */
static unsigned long read_flash_pos (XIOModule *port,
                                     unsigned char FCS)
{
  return ( (s25Read (port, FCS, 0x00000002) << 16)
         | (s25Read (port, FCS, 0x00000001) << 8)
         | (s25Read (port, FCS, 0x00000000) << 0)
         );
}

/**
 * Write current position/index to the
 * first 3 bytes of the flash.
 */
static void write_flash_pos (XIOModule *port,
                             unsigned char FCS,
                             unsigned long pos)
{

    s25Erase (port, FCS, 0x00000000);
    s25Program (port, FCS, 0x00000000,
                (pos >> 0) & 0x000000FF);
    s25Program (port, FCS, 0x00000001,
                (pos >> 8) & 0x000000FF);
    s25Program (port, FCS, 0x00000002,
                (pos >> 16) & 0x000000FF);
}
```

Listing 9.7: read_flash_pos og write_flash_pos funktioner

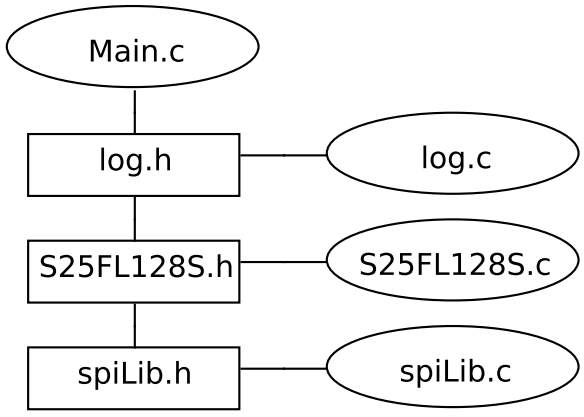
En ny main funktion for denne implementering kan ses på Listing 9.8.

```
  int main (void)
{
  XIOModule_Initialize (spi_port, XPAR_IOMODULE_0_DEVICE_ID);
  XIOModule_Start (spi_port);

  unsigned char cmd[COMMAND_LGT];
  unsigned char data[DATA_LGT];
  unsigned int cnt = 0;
  unsigned int num;
  unsigned long pos;
  unsigned long startpos;
  unsigned char i;

  init_platform ();

  /* DETERMINE ID */
  switch (busID) {
  case ID_LOG_COMMAND:
    testGetCommand (cmd);
    num = cmd[11] << 8 | cmd[12];
    startpos = pos = read_flash_pos (spi_port, FCS);
    while (pos != startpos + 16 && (cnt < num || num == 0x00)) {
```

```
23        if (pos - 16 < 0x00010000) /* Beginning of flash data.     */
24          pos = 0x00FDFFFF - 16; /* Go to last package in flash. */
25        else
26          pos -= 16;
27        readFromFlash (&spi_port, FCS, pos, data);
28        sendToEarth (cmd, data, &cnt);
29      }
30      break;
31    case ID_LOG_TOFLASH:
32      pos = read_flash_pos (spi_port, FCS);
33      writeToFlash (&spi_port, FCS, pos, busData_toFlash[i]);
34      if (pos + 16 > 0x00FDFFFF) { /* End of flash.                 */
35        pos = 0x00010000;        /* Go to first package in flash. */
36        s25Erase (&spi_port, FCS, pos);
37      }
38      else if ((pos + 16) % 0x10000 == 0) {  /* New sector. */
39        pos += 16;
40        s25Erase (&spi_port, FCS, pos);
41      }
42      else
43        pos += 16;
44      write_flash_pos (&spi_port, FCS, pos);
45      break;
46    default:
47      break ;
48    }
49    return 0;
50  }
```

Listing 9.8: LOG main funktion

Den nye main fil anvendes som et top modul i systemet. Når main funktionen skal udføre en kommando, skal den først hentes fra biblioteket "log.h". Hvis funktionen fra "log.h" biblioteket skal interagere med flashmodulet, skal der kaldes en af funktionerne fra "S25FL128S", som bruger "spiLib.h" biblioteket. Det samlede system, kan ses på Figur 9.8. Disse ændringerne i koden, betyder at SPI kommunikationen bliver langsommere og afstanden mellem pakkerne bliver længere. Dette betyder at SPI clocken kan køre langsommere uden at påvirke kommunikationshastigheden, hvilket medfører at det kan reducerer energiforbruget af det samlede system.



Figur 9.8: Opbygning af samlet software til LOG system

Resultater Fordi at læsningen fra flashmodulet ikke fungerer som forventet er det valgt, at implementere alle biblioteker sammen med en ny main fil på MicroBlazen. Herved kunne det konstateres at alt kode kan kompileres og at det ikke fylder mere end den tilgængelige plade på MicroBlazen. Dog har selve implementeringen ikke været mulig at teste, da SPI-modulet og flashen ikke er ordentligt implementeret.

9.2.2 Afslutning på Case

Under implementeringsfasen har det vist sig, at alle delmoduler ikke kan sammensættes på en Spartan-3, derfor blev designet migreret til en Spartan-6 og herefter kunne delmodulerne implementeres.

Alle delmodulerne er testet for sig og virkede, men igennem implementeringsfasen opstod der en række udfordringer med flashen, heriblandt læsefunktionen. Dette gjorde, at implementering af log-koden blev besværliggjort og blev derfor ikke testet færdig som tidligere planlagt, men blot kompileret sammen med resten af koden, for give et helhedsindtryk over størrelsen af softwaren. Når koden er kompileret fylder den cirka 23 KiB, hvilket udgør en tredjedel af pladsen på den største MicroBlaze MCS.

Der er implementeret et SPI-modul i VHDL sammen med MicroBlaze MCS, med dertilhørende bibliotek skrevet i C og en driver udviklet med VHDL. SPI-modulet er testet ved modultest og fungerer som ønsket.

Ydermere blev I^2C, kun testet for sig selv og aldrig implementeret sammen med resten af systemet. I^2C blev implementeret som kode, og det ville egne sig til at blive porteret over i hardware (VHDL) og lave et tilhørende bibliotek. Når det udelukkende er skrevet i software, er det svært at overholde timingkrav og det er mere tidskrævende for mikrocontrolleren.

Del II

Generic flight computer

10 Accepttest

I dette afsnit vil der blive samlet op på de forskellige tests der er lavet. Der vil blive tjekket om resultaterne heraf overholder kravene der blev sat i kravspecifikationen.

✓ = bestået, × = ikke bestået, ⊗ = ikke testet.

LOG modul		
Krav 1	Størrelse på data pakker	✓
Krav 2	Gemme LOG-beskeder med Unix tid	×
Krav 3	ID numre	✓
Krav 4	Sende indhold af loggen til jorden	×
Krav 5	Numeriske numre på logbeskeder	✓
SPI		
Krav 6	Data bredde	✓
Krav 7	Dividere clocken	✓
Krav 8	Arbejde med spændinger på 3,3 V	✓
Krav 9	Clockfrekvens	✓
Krav 10	Sende pakker på 8 bit	✓
Krav 11	Sætte clock-divideren med softwaren	✓
Krav 12	Continuous mode	×
Krav 13	Funktion til at sende og modtage data over SPI	✓
Flash		
Krav 14	Størrelse	✓
Krav 15	Minimum af data der skrives pr. gang	✓
Krav 16	Minimum af data der skal læses pr. gang	×
Krav 17	Forsyningsspænding	✓
Krav 18	Hastighed	✓
I^2C		
Krav 19	To forbindelser på bussen	✓
Krav 20	Kommunikationshastighed	✓
Krav 21	Flere adresser pr. enhed	✓
Krav 22	Byte længde	✓
Krav 23	Ingen begrænsning af databytes	⊗

Tabel 10.1: Accepttest

1 Div. beskrivelser

Der er blevet samlet op på de forskellige tests og der kan nu laves en konklusion på det hele.

11 Konklusion

Rapportens udgangspunkt bunder i en antagelse om, at det er muligt at udvikle en Generic Flight Computer, implementeret på en FPGA, som kan anvendes til alle delsystemer på næste studentersatellit, AAUSAT4. Med udgangspunkt i at anvende FPGA-teknologi til at udvikle en Generic Flight Computer til delsystemerne på AAUSAT4, er AAUSAT3´s delsystemer blevet analyseret og det er undersøgt hvilke kriterier, der danner grundlag for valg af mikrocontroller på denne. Denne analyse har ledt til problemformuleringen i Kapitel 3, som lyder:

> *"Hvordan designes en Generic Flight Computer, således denne kan duplikeres på en FPGA og kommunikere internt såvel som eksternt via standardiserede grænseflader?"*

Med udgangspunkt i problemformuleringen er der valgt en løsning indeholdende en MicroBlaze MCS mikrocontroller, hvori computersystemet, som er udviklet i C, kan implementeres på. Denne rapport dokumenterer design- og udviklingsfasen af et LOG-delsystem til AAUSAT4, implementeret i en XC3S1000 FPGA. Dette delsystem indeholder kode til håndtering af logning af statusbeskeder fra satellittens delmoduler, en driver til S25FL128S-flashhukommelse og et standardiset SPI-modul med tilhørende hardware, anvendt til kommunikation mellem log og flash. Udover disse er der designet en prototype til et I^2C system til den interne kommunikation på satellitten. Disse moduler kan anvendes som enheder, til en Generic Flight Computer.

Logsystemet fungerer som beskrevet i udviklingsafsnittet, Afsnit 7.1. Loggen er i stand til at modtage en kommando om hvad der skal sendes til jorden, læse data fra midlertidig hukommelse, sortere disse efter kommandoforespørgsel, og sende de data ned der ønskes. Ydermere er den i stand til at skrive modtaget data og gemme disse uden tid i midlertidig hukommelse. En log-arkitektur er fastlagt.

Loggen er ikke i stand til at modtage data fra/sende data på en bestemt, intern bus, og kan ikke kommunikere med et flashmodul. Loggen kan ikke skrive tiden/klokken ind i pakkerne som de gemmes, da der ikke er implementeret et RTC-modul (Real Time Clock). En version 2 af log-koden, som er mere overskuelig, er udviklet, og fungerer som forventet på en PC. På MicroBlaze MCS er det dog kun version 1 der virker som forventet.

Flashbiblioteket fungerer som beskrevet i udviklingsafsnittet, Afsnit 7.3. Et bibliotek er udviklet til flashchippen S25FL128S. Dette bibliotek indeholder simple funktioner til at slette, programmere og læse. Flashmodulet er testet med et Arduino udviklingsboard og tilhørende SPI bibliotek, hvor alle funktioner fungerede som forventet. Under implementering med SPI-modulet på MicroBlazen måltes på logic analyzer, ligeledes pakker med det ønskede format ind og ud af flashmodulet, indtil en uventet ændring hvorefter det blev besluttet ikke at udføre flere tests.

I^2C systemet fungerer som beskrevet i udviklingsafsnittet, Afsnit 7.4. I^2C-biblioteket er udviklet til en ATmega32. Alle masterfunktioner er fungerende. Disse er testet og målt på et oscilloskop. I sammenkobling med en slave, reagerer slaven ikke som forventet på givne inputs. Årsagen til problemet er ikke klar, men ligger sandsynligvis på slavesiden.

Slavefunktioner til I^2C-biblioteket er ikke udviklet.

SPI-modulet fungerer som beskrevet i udviklingsafsnittet, Afsnit 7.2. Ved test med interface mod flashchippen og mod flash-biblioteket, er det bekræftet at der kan oversættes en given kommando fra flash-biblioteket, overføre denne kommando serielt i form af et standardiseret full-duplex, SPI-format ud af FPGA'en til flashchippen. Standardiseringen indebærer at SPI-biblioteket kan styre opsætningen af variabel overførselshastighed, opsætning af polaritet og fase, samt at der er mulighed for tilkobling af flere slaver til SPI-masteren.

Løsningen udarbejdet i dette projekt, giver satellitprogrammøren direkte adgang til at oprette flere mikrocontrollere på samme FPGA og designe software til satellitten i C. Den udviklede prototype kan øge mulighederne for et mindre effektforbrug om bord på satellitten, set i forhold til den nuværende teknologi på AAUSAT3, og da løsningen er kompatibel med C-koden ombord på AAUSAT3, giver det derfor ikke satellitprogrammøren nogen anledning til at ændre på sit workflow under udviklingen af software eller hardware til den nye satellit.

Der er udført en accepttest, med henblik på systematisk at afklare hvilke krav der er overholdt i prototypen. Her blev det fundet at prototypen overholder de mest relevante krav opstillet i kravspecifikationen.

Det har vist sig, at FPGA'en på det valgte Spartan-3 udviklingsboard ikke er kompatibelt med MicroBlazens fulde hukommelseskapacitet på 64 KiB, men at den er begrænset og derfor ikke kan indeholde hele logsystemet og dermed bør der vælges en nyere FPGA, hvis flere delsystemer skal implementeres på samme FPGA i en MicroBlaze. Konceptet bag en Generic Flight Computer på en FPGA kan have mange fordele og vælges der en stor FPGA, kan flere delsystemer køre på samme chip, hvilket vil kunne reducere strømforbruget og man vil kunne udlicitere bestemte opgaver til dedikeret hardware. Dette vil i sidste ende kunne spare på effektforbruget, og der vil kunne udvikles mere komplekse og krævende funktioner end tidligere muligt, set i forhold til eksekveringshastigheden.

Rapporten besvarer således problemformuleringen igennem et design af en teknisk løsning i form af et delsystem på FPGA. Delsystemet er testet og ud fra de opnåede resultater, kan det konkluderes at der er basis for videreudvikling af et system hvor alle delsystemer på satellitten kan implementeres i et multi-mikrocontrollersystem på en enkelt ellere flere FPGA'er.

A Kode

A.1 LOG-kode, version 2

Herunder ses version 2 af koden udviklet til loggen. Koden er mere funktionsbaseret end version 1, er mere opdelt, og menes derfor at være lettere at vedligeholde.

```
/**
 * LOG header file for AAUSAT4.
 * @author G416
 * @date 2013-04-09
 */
#ifndef LOG_H
#define LOG_H

#define DATA_LGT    16
#define COMMAND_LGT 16

/* Command byte index */
#define CMD_ID        0
#define CMD_LOGTYPE   1
#define CMD_T_START1  2
#define CMD_T_START2  3
#define CMD_T_START3  4
#define CMD_T_START4  5
#define CMD_T_STOP1   6
#define CMD_T_STOP2   7
#define CMD_T_STOP3   8
#define CMD_T_STOP4   9
#define CMD_CTRL      10
#define CMD_NUM1      11
#define CMD_NUM2      12
#define CMD_DUM1      13
#define CMD_DUM2      14
#define CMD_DUM3      15

/* Data byte index */
#define DAT_CNT1     0
#define DAT_CNT2     1
#define DAT_CNT3     2
#define DAT_CNT4     3
#define DAT_ID       4
#define DAT_LOGTYPE  5
#define DAT_TIME1    6
#define DAT_TIME2    7
#define DAT_TIME3    8
#define DAT_TIME4    9
#define DAT_LOGBESK1 10
#define DAT_LOGBESK2 11
#define DAT_LOGDATA1 12
#define DAT_LOGDATA2 13
#define DAT_LOGDATA3 14
```

```
46 #define DAT_LOGDATA4 15
47
48 /* Control register masks */
49 #define CTRL_CNT      0x20
50 #define CTRL_ID       0x10
51 #define CTRL_LOGTYPE  0x08
52 #define CTRL_TIME     0x04
53 #define CTRL_LOGBESK  0x02
54 #define CTRL_LOGDATA  0x01
55
56 /* Logtype masks */
57 #define LOGT_NOTICE  0x08
58 #define LOGT_WARN    0x04
59 #define LOGT_ERR     0x02
60 #define LOGT_CRIT    0x01
61
62 /* Prototypes */
63 /**
64  * Write data package to flash.
65  * @return void
66  * @param *data Data to be written to flash.
67  */
68 void writeToFlash(unsigned char *data);
69 /**
70  * Test command to get different commands from user input.
71  * @return void
72  * @param *command Pointer to where command should be stored.
73  */
74 void testGetCommand(unsigned char *command);
75 /**
76  * Send enumerated data to earth, if matches command.
77  * @return void
78  * @param *command 16 byte command received from earth.
79  * @param *data 16 byte data to be matched and sent.
80  * @param cnt 4 byte counter to enumerate packages, as they are sent down.
81  */
82 void sendToEarth(unsigned char *command, unsigned char *data, unsigned int *cnt);
83 /**
84  * Read from flash.
85  * @return char 1 if successful.
86  * @param *data Pointer to where data should be stored.
87  * @param address Address of data to be read (4B).
88  */
89 char readFromFlash(unsigned char *data, unsigned int address);
90 /**
91  * Get real time clock from RTC-module.
92  * @return int 4 byte Unix-time
93  */
94 int getRTC();
95 #endif
```

Listing A.1: Logkode til PC version 2. `log.h`

```
1  /**
2   * LOG function library for AAUSAT4
3   * @author G416
4   * @date 2013-04-09
5   */
6  #include <stdio.h>
7  #include "log.h"
8  #include <time.h>
9  #include "testdata_log.h"
10
11
12 /* ---------------------------------------------------------------------- *
```

```
13  * Auxillary functions
14  * ------------------------------------------------------------------------ */
15 /**
16  * Add a byte to an array.
17  * @param[out] pkg Array to add to.
18  * @param[in] byte Byte to be added.
19  * @param[in,out] index Place in array to add to/start from.
20  *
21  * @return 1 = ok, 0 = index out of range
22  */
23 static char addtopkg(unsigned char *pkg, unsigned char byte,
24                      unsigned int *index)
25 {
26     if (*index > DATA_LGT)
27         return 0;
28     else
29         pkg[(*index)++] = byte;
30 
31     return 1; /* WARNING: Return value never used. */
32 }
33 
34 /**
35  * Is OK-functions.
36  *
37  * Determines wether the package should
38  * be sent down to earth, or not, based
39  * on *_ok-criteria.
40  *
41  * If it should, the *-data is added to
42  * the package, that is sent to earth.
43  *
44  * @param[out] pkg Place to gather data going to earth.
45  * @param[in] cmd Command from earth.
46  * @param[in] data Data from flash.
47  * @param[in,out] i Index in pkg-array.
48  *
49  * @return 1 = ok, 0 = drop package
50  */
51 static char cnt_ok(unsigned char * pkg, unsigned int *i)
52 {
53     addtopkg(pkg, 0x00, i); /* Data is later written in these cells */
54     addtopkg(pkg, 0x00, i);
55     addtopkg(pkg, 0x00, i);
56     addtopkg(pkg, 0x00, i);
57 
58     return 1;
59 }
60 static char id_ok(unsigned char *pkg, unsigned char *cmd,
61                   unsigned char *data, unsigned int *i)
62 {
63     if (cmd[CMD_ID] == 0x00 || cmd[CMD_ID] == data[DAT_ID]) /* 0x00 = send
64                                                                all down */
65         addtopkg(pkg, data[DAT_ID], i);
66     else
67         return 0;
68 
69     return 1;
70 }
71 static char logt_ok(unsigned char *pkg, unsigned char *cmd,
72                     unsigned char *data, unsigned int *i)
73 {
74     if (cmd[CMD_LOGTYPE] & data[DAT_LOGTYPE]) /* Matches all log messages
75                                                  {crit,err,war,not} */
76         addtopkg(pkg, data[DAT_LOGTYPE], i);
77     else
```

```
78          return 0;
79
80      return 1;
81  }
82  static char time_ok(unsigned char *pkg, unsigned char *cmd,
83                      unsigned char *data, unsigned int *i)
84  {
85      unsigned int intStartTime;
86      unsigned int intStopTime;
87      unsigned int intDataTime;
88
89      /* Convert 4 chars to int */
90      intStartTime = cmd[CMD_T_START1] << 24 | cmd[CMD_T_START2] << 16
91                     | cmd[CMD_T_START3] << 8 | cmd[CMD_T_START4];
92      intStopTime  = cmd[CMD_T_STOP1] << 24  | cmd[CMD_T_STOP2] << 16
93                     | cmd[CMD_T_STOP3] << 8  | cmd[CMD_T_STOP4];
94      intDataTime  = data[DAT_TIME1] << 24   | data[DAT_TIME2] << 16
95                     | data[DAT_TIME3] << 8   | data[DAT_TIME4];
96
97      if (intDataTime >= intStartTime
98              && intDataTime <= intStopTime) {
99          addtopkg(pkg, data[DAT_TIME1], i);
100         addtopkg(pkg, data[DAT_TIME2], i);
101         addtopkg(pkg, data[DAT_TIME3], i);
102         addtopkg(pkg, data[DAT_TIME4], i);
103     } else
104         return 0;
105
106     return 1;
107 }
108 static char logmsg_ok(unsigned char *pkg, unsigned char *data,
109                       unsigned int *i)
110 {
111     addtopkg(pkg, data[DAT_LOGBESK1], i);
112     addtopkg(pkg, data[DAT_LOGBESK2], i);
113
114     return 1;
115 }
116 static char logdata_ok(unsigned char *pkg, unsigned char *data,
117                        unsigned int *i)
118 {
119     addtopkg(pkg, data[DAT_LOGDATA1], i);
120     addtopkg(pkg, data[DAT_LOGDATA2], i);
121     addtopkg(pkg, data[DAT_LOGDATA3], i);
122     addtopkg(pkg, data[DAT_LOGDATA4], i);
123
124     return 1;
125 }
126
127 /**
128  * Put bit-mask on a byte-register,
129  * and see if the masking makes
130  * a match.
131  *
132  * @param[in] mask Bitmask to apply.
133  * @param[in] reg Byte to apply mask to.
134  *
135  * @return 1 = match, 0 = no match
136  */
137 static char matches(unsigned char mask, unsigned char reg)
138 {
139     return mask & reg ? 1 : 0;
140 }
141
142 /**
```

```
143  * Does data match what the earth wants from command?
144  * @param[out] pkg Place to store data goint to earth.
145  * @param[in] cmd Command from earth.
146  * @param[in] data Data from flash.
147  *
148  * @return 1 = package matches, 0 = data doesn't match.
149  */
150 static char datamatches(unsigned char *pkg, unsigned char *cmd,
151                         unsigned char *data)
152 {
153     unsigned int i;
154     unsigned char status = 0; /* number of high CTRL-bits */
155 
156     for (i = 0; i < 6; i++)   /* 6 = active bits in CTRL-register */
157         if ((cmd[CMD_CTRL] & (0x01 << i)))
158             status++;
159 
160     i = 0;
161 
162     if (matches(CTRL_CNT, cmd[CMD_CTRL])     && cnt_ok(pkg, &i))
163         status--;
164     if (matches(CTRL_ID, cmd[CMD_CTRL])      && id_ok(pkg, cmd, data, &i))
165         status--;
166     if (matches(CTRL_LOGTYPE, cmd[CMD_CTRL]) && logt_ok(pkg, cmd, data, &i))
167         status--;
168     if (matches(CTRL_TIME, cmd[CMD_CTRL])    && time_ok(pkg, cmd, data, &i))
169         status--;
170     if (matches(CTRL_LOGBESK, cmd[CMD_CTRL]) && logmsg_ok(pkg, data, &i))
171         status--;
172     if (matches(CTRL_LOGDATA, cmd[CMD_CTRL]) && logdata_ok(pkg, data, &i))
173         status--;
174 
175     return status ? 0 : 1;
176 }
177 
178 /**
179  * Write package-to-earth counter to
180  * package, before sending it down.
181  * @param[out] pkg Package for counter to be written into.
182  * @param[in,out] cnt Number of package sent down in a row.
183  */
184 static void cnttopkg(unsigned char *pkg, unsigned int *cnt)
185 {
186     unsigned int index = DAT_CNT1;
187     *cnt += 1;
188 
189     addtopkg(pkg, (*cnt & 0xff000000) >> 24, &index);
190     addtopkg(pkg, (*cnt & 0x00ff0000) >> 16, &index);
191     addtopkg(pkg, (*cnt & 0x0000ff00) >>  8, &index);
192     addtopkg(pkg, (*cnt & 0x000000ff) >>  0, &index);
193 }
194 
195 /**
196  * Clears a string.
197  * @param[out] str String to be cleared.
198  * @param[in] len Length of string.
199  */
200 static void clrstr(unsigned char * str, int len)
201 {
202     for (; len >= 0; len--) {
203         str[len] = '\0';
204     }
205 }
206 
207 /* ---------------------------------------------------------------------- *
```

```
208  * Global functions
209  * ------------------------------------------------------------------------ */
210 void writeToFlash(unsigned char *data)
211 {
212     /* TODO: Write to flash instead of screen */
213     unsigned char i;
214     int timeNow;
215 
216     timeNow = getRTC();
217     data[DAT_TIME1] = (timeNow & 0xff000000) >> 24;
218     data[DAT_TIME2] = (timeNow & 0x00ff0000) >> 16;
219     data[DAT_TIME3] = (timeNow & 0x0000ff00) >> 8;
220     data[DAT_TIME4] = (timeNow & 0x000000ff) >> 0;
221 
222     printf("FLASH WRITE: ");          /* Debug print */
223     for (i = 0; i < DATA_LGT; i++) {
224         printf("%x ", data[i]);
225     }
226     printf("\n");
227 }
228 
229 void testGetCommand(unsigned char *command)
230 {
231     /* A command is incoming from earth. Store it in "command" */
232     /* TODO: Get actual command */
233     char i;
234 
235     /* TODO: Temporary command number */
236     printf("Tests:\n0) Pass all.\n1) Unique ID.\n2) Unique LOGTYPE.\n3) Unique TIME span.\n4) Unique CTRL sequence.\n5) Only
          the first 3 to match.\n6) Send only 0's.\n7) Send only 1's.\n");
237     char tmpCommandNum;
238     scanf("%c", &tmpCommandNum);
239     tmpCommandNum -= '0';
240 
241     for (i = 0; i < COMMAND_LGT; i++) {
242         if (tmpCommandNum < 8)
243             command[i] = busCommands[tmpCommandNum][i];
244         else
245             command[i] = busCommands[0][i];
246     }
247 }
248 
249 void sendToEarth(unsigned char *command, unsigned char *data,
250                  unsigned int *cnt)
251 {
252     unsigned char dataToEarth[DATA_LGT];
253     int i;
254 
255     clrstr(dataToEarth, DATA_LGT);
256 
257     if (command[CMD_CTRL] == 0x00)
258         return;  /* No data sent if CTRL-register is empty. */
259 
260     if (datamatches(dataToEarth, command, data) == 0)
261         return;
262 
263     if (matches(CTRL_CNT, command[CMD_CTRL]))
264         cnttopkg(dataToEarth, cnt);
265 
266     /* Send to Earth */
267     printf("%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x\n",
268            dataToEarth[0]  , dataToEarth[1]  , dataToEarth[2]  , dataToEarth[3]  ,
269            dataToEarth[4]  , dataToEarth[5]  , dataToEarth[6]  , dataToEarth[7]  ,
270            dataToEarth[8]  , dataToEarth[9]  , dataToEarth[10] , dataToEarth[11] ,
271            dataToEarth[12] , dataToEarth[13] , dataToEarth[14] , dataToEarth[15]
```

```
272          );
273  }
274
275  char readFromFlash(unsigned char *data, unsigned int address)
276  {
277      /* TODO: Read from actual flash */
278      unsigned char i;
279      for (i = 0; i < DATA_LGT; i++) {
280          data[i] = busData[address][i];
281      }
282      return 1;
283  }
284
285  int getRTC()
286  {
287      /* TODO: Get time from RTC, instead of computer system time */
288      return (int)time(NULL);
289  }
```

Listing A.2: Logkode til PC version 2. `log.c`

```
1   /**
2    * LOG header file for AAUSAT4.
3    * @author G416
4    * @date 2013-05-03
5    */
6   #ifndef LOG_H
7   #define LOG_H
8
9   #define DATA_LGT    16
10  #define COMMAND_LGT 16
11
12  /* Command byte index */
13  #define CMD_ID        0
14  #define CMD_LOGTYPE   1
15  #define CMD_T_START1  2
16  #define CMD_T_START2  3
17  #define CMD_T_START3  4
18  #define CMD_T_START4  5
19  #define CMD_T_STOP1   6
20  #define CMD_T_STOP2   7
21  #define CMD_T_STOP3   8
22  #define CMD_T_STOP4   9
23  #define CMD_CTRL      10
24  #define CMD_NUM1      11
25  #define CMD_NUM2      12
26  #define CMD_DUM1      13
27  #define CMD_DUM2      14
28  #define CMD_DUM3      15
29
30  /* Data byte index */
31  #define DAT_CNT1     0
32  #define DAT_CNT2     1
33  #define DAT_CNT3     2
34  #define DAT_CNT4     3
35  #define DAT_ID       4
36  #define DAT_LOGTYPE  5
37  #define DAT_TIME1    6
38  #define DAT_TIME2    7
39  #define DAT_TIME3    8
40  #define DAT_TIME4    9
41  #define DAT_LOGBESK1 10
42  #define DAT_LOGBESK2 11
43  #define DAT_LOGDATA1 12
44  #define DAT_LOGDATA2 13
```

```
45 #define DAT_LOGDATA3 14
46 #define DAT_LOGDATA4 15
47
48 /* Control register masks */
49 #define CTRL_CNT      0x20
50 #define CTRL_ID       0x10
51 #define CTRL_LOGTYPE  0x08
52 #define CTRL_TIME     0x04
53 #define CTRL_LOGBESK  0x02
54 #define CTRL_LOGDATA  0x01
55
56 /* Logtype masks */
57 #define LOGT_NOTICE  0x08
58 #define LOGT_WARN    0x04
59 #define LOGT_ERR     0x02
60 #define LOGT_CRIT    0x01
61
62 /**
63  * Write data package to flash.
64  * @param *data Data to be written to flash.
65  */
66 void writeToFlash(unsigned char *data);
67
68 /**
69  * Test command to get different commands from user input.
70  * @param *command Pointer to where command should be stored.
71  */
72 void testGetCommand(unsigned char *command);
73
74 /**
75  * Send enumerated data to earth, if matches command.
76  * @param *command 16 byte command received from earth.
77  * @param *data 16 byte data to be matched and sent.
78  * @param cnt 4 byte counter to enumerate packages, as they are sent down.
79  */
80 void sendToEarth(unsigned char *command, unsigned char *data, unsigned int *cnt);
81
82 /**
83  * Read from flash.
84  * @param *data Pointer to where data should be stored.
85  * @param address Address of data to be read (4B).
86  * @return 1 = success, 0 = fail.
87  */
88 char readFromFlash(unsigned char *data, unsigned int address);
89
90 /**
91  * Get real time clock from RTC-module.
92  * @return 4 byte Unix-time
93  */
94 int getRTC();
95 #endif
```

Listing A.3: Logkode til Microblaze version 2. `log.h`

```
1  /**
2   * LOG function library for AAUSAT4
3   * @author G416
4   * @date 2013-05-03
5   */
6  #include <stdio.h>
7  #include "log.h"
8  #include "testdata_log.h"
9  #include "xparameters.h"
10 #include "xiomodule.h"
11 #define DUMMY_TIME 1366733256
```

```
12
13 /* ---------------------------------------------------------------------------- *
14  * Auxillary functions
15  * ---------------------------------------------------------------------------- */
16 /**
17  * Add a byte to an array.
18  * @param[out] pkg Array to add to.
19  * @param[in] byte Byte to be added.
20  * @param[in,out] index Place in array to add to/start from.
21  *
22  * @return 1 = ok, 0 = index out of range
23  */
24 static char addtopkg(unsigned char *pkg, unsigned char byte,
25                      unsigned int *index)
26 {
27     if (*index > DATA_LGT)
28         return 0;
29     else
30         pkg[(*index)++] = byte;
31
32     return 1; /* WARNING: Return value never used. */
33 }
34
35 /**
36  * Is OK-functions.
37  *
38  * Determines wether the package should
39  * be sent down to earth, or not, based
40  * on *_ok-criteria.
41  *
42  * If it should, the *-data is added to
43  * the package, that is sent to earth.
44  *
45  * @param[out] pkg Place to gather data going to earth.
46  * @param[in] cmd Command from earth.
47  * @param[in] data Data from flash.
48  * @param[in,out] i Index in pkg-array.
49  *
50  * @return 1 = ok, 0 = drop package
51  */
52 static char cnt_ok(unsigned char * pkg, unsigned int *i)
53 {
54     addtopkg(pkg, 0x00, i); /* Data is later written in these cells */
55     addtopkg(pkg, 0x00, i);
56     addtopkg(pkg, 0x00, i);
57     addtopkg(pkg, 0x00, i);
58
59     return 1;
60 }
61 static char id_ok(unsigned char *pkg, unsigned char *cmd,
62                   unsigned char *data, unsigned int *i)
63 {
64     if (cmd[CMD_ID] == 0x00 || cmd[CMD_ID] == data[DAT_ID]) /* 0x00 = send
65                                                                all down */
66         addtopkg(pkg, data[DAT_ID], i);
67     else
68         return 0;
69
70     return 1;
71 }
72 static char logt_ok(unsigned char *pkg, unsigned char *cmd,
73                     unsigned char *data, unsigned int *i)
74 {
75     if (cmd[CMD_LOGTYPE] & data[DAT_LOGTYPE]) /* Matches all log messages
76                                                  {crit,err,war,not} */
```

```
77          addtopkg(pkg, data[DAT_LOGTYPE], i);
78      else
79          return 0;
80
81      return 1;
82  }
83  static char time_ok(unsigned char *pkg, unsigned char *cmd,
84                      unsigned char *data, unsigned int *i)
85  {
86      unsigned int intStartTime;
87      unsigned int intStopTime;
88      unsigned int intDataTime;
89
90      /* Convert 4 chars to int */
91      intStartTime = cmd[CMD_T_START1] << 24 | cmd[CMD_T_START2] << 16
92                   | cmd[CMD_T_START3] << 8 | cmd[CMD_T_START4];
93      intStopTime  = cmd[CMD_T_STOP1] << 24  | cmd[CMD_T_STOP2] << 16
94                   | cmd[CMD_T_STOP3] << 8  | cmd[CMD_T_STOP4];
95      intDataTime  = data[DAT_TIME1] << 24   | data[DAT_TIME2] << 16
96                   | data[DAT_TIME3] << 8   | data[DAT_TIME4];
97
98      if (intDataTime >= intStartTime
99              && intDataTime <= intStopTime) {
100         addtopkg(pkg, data[DAT_TIME1], i);
101         addtopkg(pkg, data[DAT_TIME2], i);
102         addtopkg(pkg, data[DAT_TIME3], i);
103         addtopkg(pkg, data[DAT_TIME4], i);
104     } else
105         return 0;
106
107     return 1;
108 }
109 static char logmsg_ok(unsigned char *pkg, unsigned char *data,
110                       unsigned int *i)
111 {
112     addtopkg(pkg, data[DAT_LOGBESK1], i);
113     addtopkg(pkg, data[DAT_LOGBESK2], i);
114
115     return 1;
116 }
117 static char logdata_ok(unsigned char *pkg, unsigned char *data,
118                        unsigned int *i)
119 {
120     addtopkg(pkg, data[DAT_LOGDATA1], i);
121     addtopkg(pkg, data[DAT_LOGDATA2], i);
122     addtopkg(pkg, data[DAT_LOGDATA3], i);
123     addtopkg(pkg, data[DAT_LOGDATA4], i);
124
125     return 1;
126 }
127
128 /**
129  * Put bit-mask on a byte-register,
130  * and see if the masking makes
131  * a match.
132  *
133  * @param[in] mask Bitmask to apply.
134  * @param[in] reg Byte to apply mask to.
135  *
136  * @return 1 = match, 0 = no match
137  */
138 static char matches(unsigned char mask, unsigned char reg)
139 {
140     return mask & reg ? 1 : 0;
141 }
```

```
142
143 /**
144  * Does data match what the earth wants from command?
145  * @param[out] pkg Place to store data goint to earth.
146  * @param[in] cmd Command from earth.
147  * @param[in] data Data from flash.
148  *
149  * @return 1 = package matches, 0 = data doesn't match.
150  */
151 static char datamatches(unsigned char *pkg, unsigned char *cmd,
152                         unsigned char *data)
153 {
154     unsigned int i;
155     unsigned char status = 0; /* number of high CTRL-bits */
156
157     for (i = 0; i < 6; i++)   /* 6 = active bits in CTRL-register */
158         if ((cmd[CMD_CTRL] & (0x01 << i)))
159             status++;
160
161     i = 0;
162
163     if (matches(CTRL_CNT, cmd[CMD_CTRL])     && cnt_ok(pkg, &i))
164         status--;
165     if (matches(CTRL_ID, cmd[CMD_CTRL])      && id_ok(pkg, cmd, data, &i))
166         status--;
167     if (matches(CTRL_LOGTYPE, cmd[CMD_CTRL]) && logt_ok(pkg, cmd, data, &i))
168         status--;
169     if (matches(CTRL_TIME, cmd[CMD_CTRL])    && time_ok(pkg, cmd, data, &i))
170         status--;
171     if (matches(CTRL_LOGBESK, cmd[CMD_CTRL]) && logmsg_ok(pkg, data, &i))
172         status--;
173     if (matches(CTRL_LOGDATA, cmd[CMD_CTRL]) && logdata_ok(pkg, data, &i))
174         status--;
175
176     return status ? 0 : 1;
177 }
178
179 /**
180  * Write package-to-earth counter to
181  * package, before sending it down.
182  * @param[out] pkg Package for counter to be written into.
183  * @param[in,out] cnt Number of package sent down in a row.
184  */
185 static void cnttopkg(unsigned char *pkg, unsigned int *cnt)
186 {
187     unsigned int index = DAT_CNT1;
188     *cnt += 1;
189
190     addtopkg(pkg, (*cnt & 0xff000000) >> 24, &index);
191     addtopkg(pkg, (*cnt & 0x00ff0000) >> 16, &index);
192     addtopkg(pkg, (*cnt & 0x0000ff00) >>  8, &index);
193     addtopkg(pkg, (*cnt & 0x000000ff) >>  0, &index);
194 }
195
196 /**
197  * Clears a string.
198  * @param[out] str String to be cleared.
199  * @param[in] len Length of string.
200  */
201 static void clrstr(unsigned char * str, int len)
202 {
203     for (; len >= 0; len--) {
204         str[len] = '\0';
205     }
206 }
```

```
207
208 /* ------------------------------------------------------------------------ *
209  * Global functions
210  * ------------------------------------------------------------------------ */
211 void writeToFlash(unsigned char *data)
212 {
213     /* TODO: Write to flash instead of screen */
214     unsigned char i;
215     int timeNow;
216
217     timeNow = getRTC();
218     data[DAT_TIME1] = (timeNow & 0xff000000) >> 24;
219     data[DAT_TIME2] = (timeNow & 0x00ff0000) >> 16;
220     data[DAT_TIME3] = (timeNow & 0x0000ff00) >> 8;
221     data[DAT_TIME4] = (timeNow & 0x000000ff) >> 0;
222
223     print("FLASH WRITE: ");          /* Debug print */
224     for (i = 0; i < DATA_LGT; i++) {
225         putnum(data[i]);
226     }
227     print("\n");
228 }
229
230 void testGetCommand(unsigned char *command)
231 {
232     unsigned char i;
233     int sw; /* Choose command with switches. */
234
235     XIOModule gpi1;
236     sw = XIOModule_Initialize(&gpi1, XPAR_IOMODULE_0_DEVICE_ID);
237     sw = XIOModule_Start(&gpi1);
238     sw = XIOModule_DiscreteRead(&gpi1, 1);
239
240     for (i = 0; i < COMMAND_LGT; i++) {
241         if (sw < 8)
242             command[i] = busCommands[sw][i];
243         else
244             command[i] = busCommands[0][i];
245     }
246 }
247
248 void sendToEarth(unsigned char *command, unsigned char *data,
249                  unsigned int *cnt)
250 {
251     unsigned char dataToEarth[DATA_LGT];
252     int i;
253
254     clrstr(dataToEarth, DATA_LGT);
255
256     if (command[CMD_CTRL] == 0x00)
257         return;  /* No data sent if CTRL-register is empty. */
258
259     if (datamatches(dataToEarth, command, data) == 0)
260         return;
261
262     if (matches(CTRL_CNT, command[CMD_CTRL]))
263         cnttopkg(dataToEarth, cnt);
264
265
266     for (i = 0; i < DATA_LGT; i++)
267         putnum(dataToEarth[i]);  /* Send to Earth */
268     print("\n");
269 }
270
271 char readFromFlash(unsigned char *data, unsigned int address)
```

```
272 {
273     /* TODO: Read from actual flash */
274     unsigned char i;
275     for (i = 0; i < DATA_LGT; i++) {
276         data[i] = busData[address][i];
277     }
278     return 1;
279 }
280 
281 int getRTC()
282 {
283     /* TODO: Get time from RTC, instead of computer system time */
284     return DUMMY_TIME;
285 }
```

Listing A.4: Logkode til Microblaze version 2. `log.c`

B Flash

B.0.1 Flash Command Summary

På Figur B.1 og B.2 ses et overblik over de forskellige kommandoer der kan sendes til flashmodulet.

Instruction (Hex)	Command Name	Command Description	Maximum Frequency (MHz)
01	WRR	Write Register (Status-1, Configuration-1)	133
02	PP	Page Program (3- or 4-byte address)	133
03	READ	Read (3- or 4-byte address)	50
04	WRDI	Write Disable	133
05	RDSR1	Read Status Register-1	133
06	WREN	Write Enable	133
07	RDSR2	Read Status Register-2	133
0B	FAST_READ	Fast Read (3- or 4-byte address)	133
0C	4FAST_READ	Fast Read (4-byte address)	133
0D	DDRFR	DDR Fast Read (3- or 4-byte address)	66
0E	4DDRFR	DDR Fast Read (4-byte address)	66
12	4PP	Page Program (4-byte address)	133
13	4READ	Read (4-byte address)	50
14	ABRD	AutoBoot Register Read	133
15	ABWR	AutoBoot Register Write	133
16	BRRD	Bank Register Read	133
17	BRWR	Bank Register Write	133
18	Reserved-18	Reserved	
20	P4E	Parameter 4 kB-sector Erase (3- or 4-byte address)	133
21	4P4E	Parameter 4 kB-sector Erase (4-byte address)	133
2B	ASPRD	ASP Read	133
2F	ASPP	ASP Program	133
30	CLSR	Clear Status Register - Erase/Program Fail Reset	133
32	QPP	Quad Page Program (3- or 4-byte address)	80
34	4QPP	Quad Page Program (4-byte address)	80
35	RDCR	Read Configuration Register-1	133
38	QPP	Quad Page Program (3- or 4-byte address)	80
3B	DOR	Read Dual Out (3- or 4-byte address)	104
3C	4DOR	Read Dual Out (4-byte address)	104
41	DLPRD	Data Learning Pattern Read	133
42	OTPP	OTP Program	133
43	PNVDLR	Program NV Data Learning Register	133
4A	WVDLR	Write Volatile Data Learning Register	133
4B	OTPR	OTP Read	133
60	BE	Bulk Erase	133
6B	QOR	Read Quad Out (3- or 4-byte address)	104
6C	4QOR	Read Quad Out (4-byte address)	104
75	ERSP	Erase Suspend	133
7A	ERRS	Erase Resume	133
85	PGSP	Program Suspend	133
8A	PGRS	Program Resume	133
90	READ_ID (REMS)	Read Electronic Manufacturer Signature	133
9F	RDID	Read ID (JEDEC Manufacturer ID and JEDEC CFI)	133
A3	MPM	Reserved for Multi-I/O-High Perf Mode (MPM)	133

Figur B.1: Oversigt over de forskellige kommandoer til flashmodulet

Instruction (Hex)	Command Name	Command Description	Maximum Frequency (MHz)
A6	PLBWR	PPB Lock Bit Write	133
A7	PLBRD	PPB Lock Bit Read	133
AB	RES	Read Electronic Signature	50
B9	BRAC	Bank Register Access (Legacy Command formerly used for Deep Power Down)	133
BB	DIOR	Dual I/O Read (3- or 4-byte address)	104
BC	4DIOR	Dual I/O Read (4-byte address)	104
BD	DDRDIOR	DDR Dual I/O Read (3- or 4-byte address)	66
BE	4DDRDIOR	DDR Dual I/O Read (4-byte address)	66
C7	BE	Bulk Erase (alternate command)	133
D8	SE	Erase 64 kB or 256 kB (3- or 4-byte address)	133
DC	4SE	Erase 64 kB or 256 kB (4-byte address)	133
E0	DYBRD	DYB Read	133
E1	DYBWR	DYB Write	133
E2	PPBRD	PPB Read	133
E3	PPBP	PPB Program	133
E4	PPBE	PPB Erase	133
E5	Reserved-E5	Reserved	
E6	Reserved-E6	Reserved	
E7	PASSRD	Password Read	133
E8	PASSP	Password Program	133
E9	PASSU	Password Unlock	133
EB	QIOR	Quad I/O Read (3- or 4-byte address)	104
EC	4QIOR	Quad I/O Read (4-byte address)	104
ED	DDRQIOR	DDR Quad I/O Read (3- or 4-byte address)	66
EE	4DDRQIOR	DDR Quad I/O Read (4-byte address)	66
F0	RESET	Software Reset	133
FF	MBR	Mode Bit Reset	133

Figur B.2: Oversigt over de forskellige kommandoer til flashmodulet

C Arbejdsblade

C.1 Clock Divider

18. marts 2013

C.1.1 Problem/formål

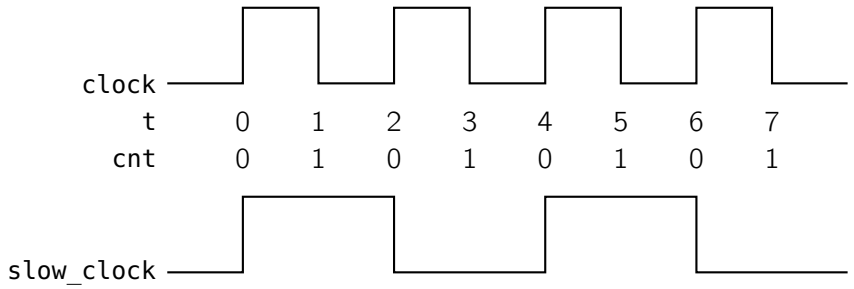
At skalere clocken ned, så mennesker kan følge med i hvad der sker for hver clockcyklus.

C.1.2 Løsning/resultat

Løsningen er, at skrive en funktion i VHDL, der tager et heltal, som der det, clocken skal divideres med. I C-pseudokode, ser funktionen således ud:

```
1 std_logic clockdivide (std_logic clock, natural divideby){
2   return clock/divideby;
3 }
```

– altså returneres en clock, der er `divideby` gange langsommere end den fysiske clock (eller hvad der ellers bliver givet som input.) Se eventuelt timingdiagram, Figur C.1.



Figur C.1: Timingdiagram for clock divider, med `divideby` = 2.

Funktionen implementeres i VHDL i en IP-blok, så den kan hentes ind til at lave flere forskellige clocks. Funktionen ligger i IP-blok-biblioteket på cden:

`code/IP/clk_divider/`

Koden, der dividerer clocken, ser således ud:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity clkdivider is
5     generic (divideby : natural := 2);
```

```
6     Port ( clk : in std_logic;
7            slow_pulse : out std_logic
8                           );
9 end clkdivider;
10
11 architecture Behavioral of clkdivider is
12 signal cnt : natural range 0 to divideby-1;
13 signal toggle : std_logic := '0';
14 begin
15
16 process(clk)
17 begin
18         if rising_edge(clk) then
19                 if (cnt = (divideby-1)/2)  then
20                         toggle <= not toggle;
21                         cnt <= 0;
22                 else
23                         cnt <= cnt+1;
24                 end if;
25         end if;
26 end process;
27 slow_pulse <= toggle;
28 end Behavioral;
```

Listing C.1: Clock divider fil

```
1 --  instantiering af clkdivider, hvor der divideres med 2
2
3   clk_divided : clkdivider generic map (divideby => 2)
4   port map (clk=>clk, slow_pulse => your_signal);
```

Listing C.2: Hovedkode

Det viser sig, at man ikke kan bruge ordet `clock` eller `clk` i andre inputs/outputs end clocken selv. Derfor benyttes ordet `pulse` i stedet.

C.1.3 Konklusion

Ved at anvende en Clock Divider kan der returneres en clock, der er `divideby` gange langsommere end den fysiske clock. Det er vigtigt at tage højde for at dele med et lige tal. I tilfælde af at `divideby` er et ulige tal, vil clock divider runde op til næste lige tal.

C.2 Komponentfil

18. marts 2013

C.2.1 Problem/formål

For at skabe overblik, vil hvert projekt have en komponentfil, som alle komponentfiler beskrives i.

C.2.2 Løsning/resultat

Der er tre dele i at bruge komponenter i VHDL:

1. Komponentbeskrivelse.
2. Inkludering af komponentfil.

3. Instantiering af komponent.

Dette svarer til at lave en header-fil i C, og gøres for at skabe mere overblik i hovedkoden. Fordeligen af disse tre områder, er således:

Hovedprogram	**Komponentfil**
Inkludering af komponentfil	Komponentbeskrivelse
Instantiering af komponent	

```
1  -- Komponentbeskrivelse
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  package my_components is
6
7  component kcpsm3
8      Port (      address       : out std_logic_vector(9 downto 0);
9                  instruction   : in std_logic_vector(17 downto 0);
10                 port_id       : out std_logic_vector(7 downto 0);
11                 write_strobe  : out std_logic;
12                 out_port      :  out std_logic_vector(7 downto 0);
13                 read_strobe   : out std_logic;
14                 in_port       : in std_logic_vector(7 downto 0);
15                 interrupt     : in std_logic;
16                 interrupt_ack : out std_logic;
17                 reset         : in std_logic;
18                 clk           : in std_logic);
19   end component;
20
21   component p1
22     Port (      address     : in std_logic_vector(9 downto 0);
23                 instruction : out std_logic_vector(17 downto 0);
24                 clk         : in std_logic);
25   end component;
26
27   component p2
28     Port (      address     : in std_logic_vector(9 downto 0);
29                 instruction : out std_logic_vector(17 downto 0);
30                 clk         : in std_logic);
31   end component;
32
33   component clkdivider
34     generic (   divideby  : natural := 2);
35     Port(       clk       : in std_logic;
36                 slow_clk  : inout std_logic);
37   end component;
38
39 end my_components;
```

Listing C.3: Eksempel på komponentfil

```
1  use my_components.all; -- Inkludering af komponentfil
2  ...
3  -- Instantiering af komponenter
4    processor1: kcpsm3
5    port map(      address      => address1,
6                   instruction  => instruction1,
7                   port_id      => port_id1,
8                   write_strobe => write_strobe1,
9                   out_port     => out_port1,
10                  read_strobe  => read_strobe1,
```

```
                in_port      => in_port1,
                interrupt    => interrupt1,
                interrupt_ack => interrupt_ack1,
                reset        => reset1,
                clk          => new_clk);

  processor2: kcpsm3
  port map(     address      => address2,
                instruction  => instruction2,
                port_id      => port_id2,
                write_strobe => write_strobe2,
                out_port     => out_port2,
                read_strobe  => read_strobe2,
                in_port      => in_port2,
                interrupt    => interrupt2,
                interrupt_ack => interrupt_ack2,
                reset        => reset2,
                clk          => new_clk);

  program1: p1
  port map(     address     => address1,
                instruction => instruction1,
                clk         => new_clk);

  program2: p2
  port map(     address     => address2,
                instruction => instruction2,
                clk         => new_clk);

  d_ff_in: entity work.d_ff(arch)
  port map(     d   => out_port1,
                q   => in_port2,
                ce  => ce_in);

  d_ff_led: entity work.d_ff(arch)
  port map(     d   => out_port1,
                q   => led,
                ce  => ce_led);
...
```

Listing C.4: Eksempel på hovedkode

C.2.3 Konklusion

Ved at benytte en ekstern komponent-fil til beskrivelse af komponenter, vil hovedkoden blive simplere at se på. Projekter vil derfor fremover have en ekstern komponentfil til dette.

C.3 Microblaze - Hello world UART program

26. marts 2013

C.3.1 Problem/formål

Formålet er at få oprettet en `Microblaze_MCS` mikrocontroller i VHDL, med tilhørende software som er skrevet i C. Denne skal kommunikere med PC ved hjælp af en UART rs232 forbindelse.

C.3.2 Løsning/resultat

Der er flere trin i denne process, disse er listet som følgende punkter

1. Opret en MCS IP-blok i ISE.
    - Tilføj ny source og vælg MCS.
    - indstil clock til 50 MHz og memory til 16 kB.
    - UART indstilles til `en_reciever` og `en_transmiter` og baud indstilles som ønsket
    - Der kan oprettes inputs og outputs ved at vælge GPO og GPI, vælg use GPO og use GPI med otte bits hver.
2. Opret toplevel.vhdl.
    - Tilføj ny source (VHDL)
    - Marker `microblaze_mcs` → Processes: `microblaze_mcs` (vinduet under) → Dobbelt klik på "View HDL Instantiation Template" under "CORE Generator"
    - indsæt instantiation template i toplevel VHDL filen
    - tryk Synthezise
3. Opret merged BMM-fil og update værktøj til at anvende til BMM-fil.
    - Find tcl console
    - skriv source ipcore_dir/microblaze_mcs_setup.tcl
    - skriv source `ipcore_dir/microblaze_mcs_setup.tcl`
4. Implement Design.
5. Opret C-program til FPGA-platformen
    - Start XSDK applikation
    - vælg `file/new/xilinx` hardware platform specification
    - lokaliser xml-filen i `ip_core_dir`
    - vælg `file/new/xilinx` board support package
    - standalone
    - CPUen har det navn som du har givet i ISE
    - vælg `file/new/xilinx/C` application project
    - vælg i dette tilfælde templaten `hello_world`
6. Generer en bit-fil til at downloade på FPGA-board
    - Opret en passende Constraint-fil i ISE
    - Tryk Generate Programming File
    - Find nu tcl console og skriv `microblaze_mcs_data2mem workspace/sdk-program/Debug/sdk-program.elf`, hvor sdk-program i dette tilfælde hedder `hello_world`
7. download på FPGA

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity MCS_UART_toplevel is
5 port(
6     clk      : IN STD_LOGIC;
7     reset    : IN STD_LOGIC;
8     rx       : IN STD_LOGIC;
9     tx       : OUT STD_LOGIC;
```

```
    leds     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    switches : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
);
end MCS_UART_toplevel;

architecture Behavioral of MCS_UART_toplevel is

------------- MCS COMPONENT Declaration ------
COMPONENT MCS_UART
  PORT (
    Clk       : IN STD_LOGIC;
    Reset     : IN STD_LOGIC;
    UART_Rx   : IN STD_LOGIC;
    UART_Tx   : OUT STD_LOGIC;
    GPO1      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    GPI1      : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    GPI1_Interrupt : OUT STD_LOGIC
  );
END COMPONENT;
 ------ End MCS COMPONENT Declaration ------------

begin
------------- MCS INSTANTIATION Template -----
mcs_0 : MCS_UART
  PORT MAP (
    Clk       => clk,
    Reset     => reset,
    UART_Rx   => rx,
    UART_Tx   => tx,
    GPO1      => leds,
    GPI1      => switches,
    GPI1_Interrupt => open
  );
------ End MCS INSTANTIATION Template ------------

end Behavioral;
```

Listing C.5: Eksempel på toplevel-fil

```
#include <stdio.h>
#include "platform.h"


void print(char *str);

int main()
{
    init_platform();

    print("Microblaze is writing with function print()\nWhich is transmitting through the UART\n\r");

    cleanup_platform();

    return 0;
}
```

Listing C.6: Eksempel på main.c fil

C.3.3 Konklusion

Ved at benytte de ISE og XSDK kan der nu oprettes microblaze mikroprocessor-systemer.

C.4 Dual microblaze

27. marts 2013

C.4.1 Problem/formål

Får to Microblazes til at snakke sammen. Hvor en af dem skal aflæse switches og sende de aflæste data til den anden Microblaze. Denne sender det videre via uart til computer og printer det ud på skærmen.

C.4.2 Løsning/resultat

VHDL design

1. Der blev tilføjet to Microblaze processorer med forskellige navne og instanteringsnavne (`microblaze_mcs` + `microblaze_mcs1` og `mcs_0` + `mcs_1`) i ISE ved hjælp af add source wizard.
2. Begge processorer blev inkluderet ind i topfil og forbundet ud til signaler.
3. GPI fra `mcs_0` er forbundet til sliders og dens output til inputtet på den næste (`mcs_1`). Dennes output går ud på LED'erne, og den har desuden et UART-output.
4. Derefter syntetiseres designet og følgende kommando køres i TCL konsollen:

   ```
   source ipcore_dir/microblaze_mcs_setup.tcl
   ```

VHDL-kode for topfil findes på CD:

`code/microblaze_sw-led-uart_dualcore`

og ses herunder:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity topEntity is
5    Port ( sliders  : in  STD_LOGIC_VECTOR (7 downto 0);
6           leds     : out  STD_LOGIC_VECTOR (7 downto 0);
7           uart_rx  : in  STD_LOGIC;
8           uart_tx  : out  STD_LOGIC;
9           clk      : in  STD_LOGIC;
10          reset    : in  STD_LOGIC);
11 end topEntity;
12
13 architecture arch of topEntity is
14   COMPONENT microblaze_mcs
15     PORT (
16             Clk            : IN STD_LOGIC;
17             Reset          : IN STD_LOGIC;
18             UART_Rx        : IN STD_LOGIC;
19             UART_Tx        : OUT STD_LOGIC;
20             GPO1           : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
21             GPI1           : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
22             GPI1_Interrupt : OUT STD_LOGIC
23           );
24   END COMPONENT;
25   COMPONENT microblaze_mcs1
26     PORT (
27             Clk     : IN STD_LOGIC;
28             Reset   : IN STD_LOGIC;
29             UART_Rx : IN STD_LOGIC;
```

```
30            UART_Tx : OUT STD_LOGIC;
31            GPO1    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
32            GPI1    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
33            GPI1_Interrupt : OUT STD_LOGIC
34          );
35    END COMPONENT;
36
37        --processor 0 signaler:
38    signal  uart_rx_0        : STD_LOGIC;
39    signal  uart_tx_0        : STD_LOGIC;
40    signal GPO1_0            : STD_LOGIC_VECTOR(7 DOWNTO 0);
41    signal GPI1_0            : STD_LOGIC_VECTOR(7 DOWNTO 0);
42    signal GPI1_Interrupt_0  : STD_LOGIC;
43        --processor 1 signaler:
44    signal  uart_rx_1        : STD_LOGIC;
45    signal  uart_tx_1        : STD_LOGIC;
46    signal GPO1_1            : STD_LOGIC_VECTOR(7 DOWNTO 0);
47    signal GPI1_1            : STD_LOGIC_VECTOR(7 DOWNTO 0);
48    signal GPI1_Interrupt_1  : STD_LOGIC;
49
50
51  begin
52    mcs_0 : microblaze_mcs
53    PORT MAP (
54              Clk            => Clk,
55              Reset          => Reset,
56              UART_Rx        => UART_Rx_0,
57              UART_Tx        => UART_Tx_0,
58              GPO1           => gpo1_0,
59              GPI1           => gpi1_0,
60              GPI1_Interrupt  => open
61            );
62    mcs_1 : microblaze_mcs1
63    PORT MAP (
64              Clk            => Clk,
65              Reset          => Reset,
66              UART_Rx        => UART_Rx_1,
67              UART_Tx        => UART_Tx_1,
68              GPO1           => gpo1_1,
69              GPI1           => gpi1_1,
70              GPI1_Interrupt  => open
71            );
72
73  -- Fysisk mapping
74  gpi1_0      <= sliders;
75  gpi1_1      <= gpo1_0;
76  leds        <= gpo1_1;
77  UART_Rx_1   <= UART_Rx;
78  UART_Tx     <= UART_Tx_1;
79
80  end arch;
```

Listing C.7: Topfil

C design

C-designet sker i XSDK-programmet.

1. Der laves nyt hardware platform specifikation projekt for hver processors program(program0 og program1).
2. Der laves der en application projekt for hver hardware platform, som tager udgangspunkt i "hello world"-skabelonen. Husk at vælge en rigtig hardware platform.

3. Der rettes i `src/hello_world.c`-filerne. Koden ses herunder. Dette er C-koden til processorerne.
4. Filene gemmes, og compiles derved automatisk.

Programmerne herunder findes på CD:

`code/microblaze_sw-led-uart_dualcore/workspace/{program0,program1}`

```
1
2  #include <stdio.h>
3  #include "platform.h"
4  #include "xparameters.h" // add
5  #include "xiomodule.h"   // add
6
7  void print(char *str);
8
9  int main()
10 {
11         init_platform();
12
13         u32 data;
14         XIOModule gpi;
15         XIOModule gpo;
16
17         data = XIOModule_Initialize(&gpi, XPAR_IOMODULE_0_DEVICE_ID);
18         data = XIOModule_Start(&gpi);
19         data = XIOModule_Initialize(&gpo, XPAR_IOMODULE_0_DEVICE_ID);
20         data = XIOModule_Start(&gpo);
21
22         while (1){
23                 data = XIOModule_DiscreteRead(&gpi, 1); // read switches (channel 1)
24                 XIOModule_DiscreteWrite(&gpo, 1, data); // writes to GPO (channel 1)
25         }
26         cleanup_platform();
27
28
29         return 0;
30 }
```

Listing C.8: program0

```
1
2  #include <stdio.h>
3  #include "platform.h"
4  #include "xparameters.h" // add
5  #include "xiomodule.h"   // add
6
7  void print(char *str);
8
9  int main()
10 {
11         init_platform();
12         xil_printf("Starting program1");
13         u32 data;
14         XIOModule gpi;
15         XIOModule gpo;
16
17         data = XIOModule_Initialize(&gpi, XPAR_IOMODULE_0_DEVICE_ID);
18         data = XIOModule_Start(&gpi);
19         data = XIOModule_Initialize(&gpo, XPAR_IOMODULE_0_DEVICE_ID);
20         data = XIOModule_Start(&gpo);
21
22         while (1) {
23                 data = XIOModule_DiscreteRead(&gpi, 1); // read switches (channel 1)
```

```
24                  XIOModule_DiscreteWrite(&gpo, 1, data); // turn on LEDs (channel 1)
25                  xil_printf("%s", data);
26          }
27          cleanup_platform();
28          return 0;
29  }
```

Listing C.9: program1

VHDL design

Tilbage i ISE programmet.

1. I TCL konsollen køres der kommando:

   ```
   microblaze_mcs_data2mem sti_til_program0/Debug/program0-fil.elf sti_til_program1/Debug/program1-
   ```

2. "Generete programming file" køres.
3. Programmet lægges på FPGA'en med Impact.

C.4.3 Konklusion

Microblaze nummmer 0 modtager sliderdata og sender til Microblaze nummer 1. Denne printer, med succes, de modtagne data ud på LED'erne,

På seriel-terminalen ses de data der modtages. Disse skulle gerne representere den konfiguration sliderne er i. Disse ser dog ikke helt ud som forventet, men skifter så snart sliderne står i en anden position. Derfor antages det, at det er noget der er galt i C-programmet, da der skiftes lidt rundt imellem datatyper, inden de sendes til PC'en.

C.5 Programmering af Flash PROM

26. marts 2013

C.5.1 Problem/formål

Flash hukommelsen skal programmeres, så programmet loades over på FPGA'en, når spartan3 boarded forsynes.

C.5.2 Løsning/resultat - Fremgangsmetode ISE iMPACT

PROM filen genereres og loades i ISE iMPACT programmet. PROM filen gemmes, som en .MCS filtype og loades over i Flash hukommelsen på samme måde, som FPGA'en programmeres.

Fremgangsmåden til programmering af Flash hukommelsen er listet herunder:

- iMPACT Flows(vinduet øverst til venstre) : Create PROM File (PROM File Formatter)
- Step 1
  1. Xilinx Flash/PROM
- Step 2

    1. PROM Familly: Platform Flash
    2. Device (bits): xcf04s [4M] (gælder kun for vores board)
    3. Add Storage Device
- Step 3
    1. Output File Name
    2. Output File Location
    3. File Format: MCS
- Add Device : OK
- Add bit filen, der hører til projektet
- Would you like to add another device file to : NO
- iMPACT(vinduet nede til venstre) : Generate File...
- Boundary Scan
- initialize chain
- add bit filen hørende til projektet
- add MCS filen hørende til projektet(Flash/PROM filen)
- Program xcs31000
- Program xcf04s

C.5.3 Konklusion

Flashen blev programmet uden fejl. Programmet loades fra Flash hukommelsen til FPGA'en, så snart der er strøm til boarded. Alt i alt fungerer det, som det skal.

Litteratur

[1] Spartan-3 fpga family: Complete data sheet. http://pdf1.alldatasheet.com/datasheet-pdf/view/159178/XILINX/XC3S1000-4FT256C.html, juli 2004.

[2] 08gr414. *AAUSAT3*. Aalborg Universitet, 2008.

[3] Embedded Systems Academy. i2c-generel. http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/i2c-bus.html, 1999-2010.

[4] Pong P. Chu. *FPGA Prototyping by VHDL examples*. Wiley-Interscience, 2008.

[5] DIGILENT. Spartan-3 board. http://www.digilentinc.com/Products/Detail.cfm?Prod=S3BOARD, 2004.

[6] Robot Electronics. Using the i2c bus. http://www.robot-electronics.co.uk/acatalog/I2C_Tutorial.html.

[7] Kent Krøyer. Strålingshærdet chip på vej til rumfarten. september 2010.

[8] Scott Larsson. Serial peripheral interface (spi) slave (vhdl) - logic - eewiki. http://eewiki.net/x/YIA_/.

[9] Jens Dalsgaard Nielsen. Aausat3 & 4.

[10] Philips. I2c-bus specification and user manual. http://da.wikipedia.org/wiki/I%C2%B2C#cite_note-i2c-rev2-1-4, 2012.

[11] J. Ranade and A. Nash. *The elements of C programming style*. McGraw-Hill, 1993.

[12] Spansion. Datablad for s25fl128s - flash modul. http://pdf1.alldatasheet.com/datasheet-pdf/view/466526/SPANSION/S25FL128S.html.

[13] Andrew S. Tanenbaum. *Structured Computer Organisation*. Pearson, 2010.

[14] Aalborg Universitet. Aausat3 home. http://aausat3.dk.

[15] Aalborg Universitet. Aausat4 home. http://aausat4.dk.

[16] Wikipedia. Ultra high frequency. http://en.wikipedia.org/wiki/Ultra_high_frequency.