0.1 Arbejdsblad

Hej Søren

Vi vil gerne have at du skimmer design af lydmodulet igennem. Her vil vi især gerne have respons på strukturen(råd tråd), men hvis du har andet at påpege, tager vi gerne imod.

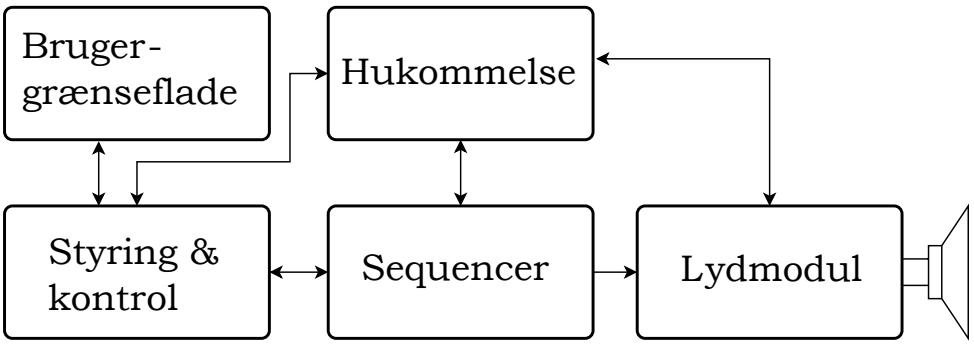
Derudover er der resten af designafsnitet. Der er måske lidt for meget af læse på så kort varsel, og noget af det har du vist også læst før.

Desuden vil vi gerne have lidt inspiration til test og deltests af trommemaskinen.

Indholdsfortegnelse

1 Design

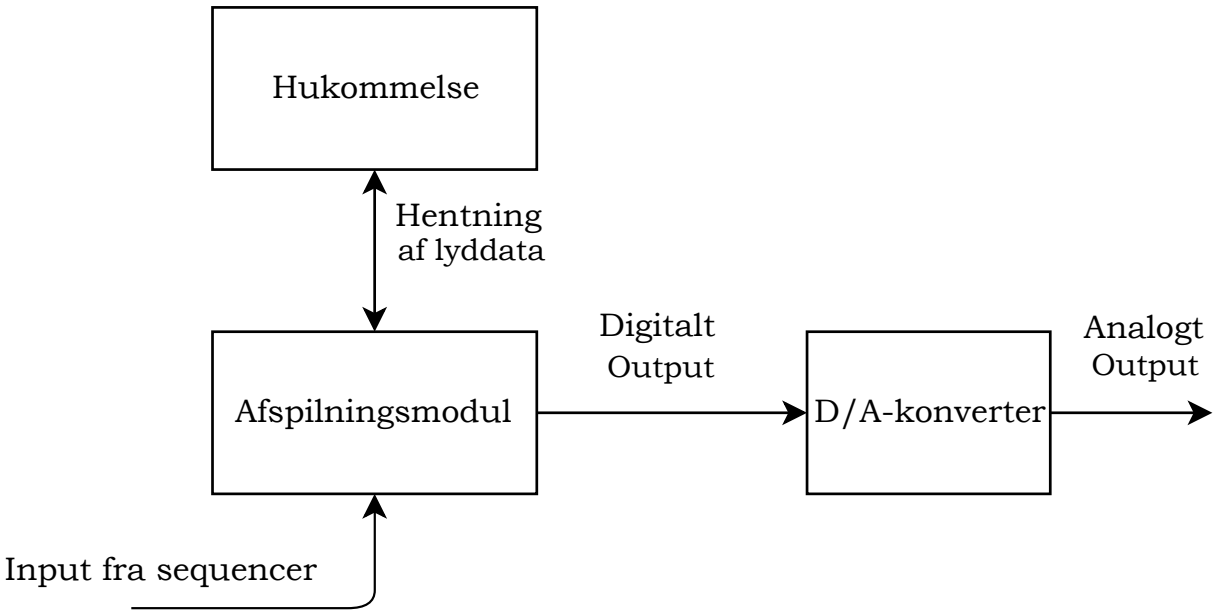
I dette kapitel beskrives trommemaskinens design og implementering. Designet er opdelt med udgangspunkt i tre delmodulers kravspecifikationer (kontrolmodul, sequencer og lydmodul) som det er beskrevet i afsnit **??**.



Figur 1.1: Blokdiagram over trommemaskinen som helhed

1.1 Design af lydmodul

En af trommemaskinens primære opgaver, er at være i stand til at lyde som et trommesæt ved afspilning af lyd. Denne opgave opdeles i er to elementære funktioner. For det første skal lydmodulet være i stand til at finde den korrekte lyd-fil baseret på et input fra en sequencer, og derefter skal lyd-fil konverteres til et analogt signal.



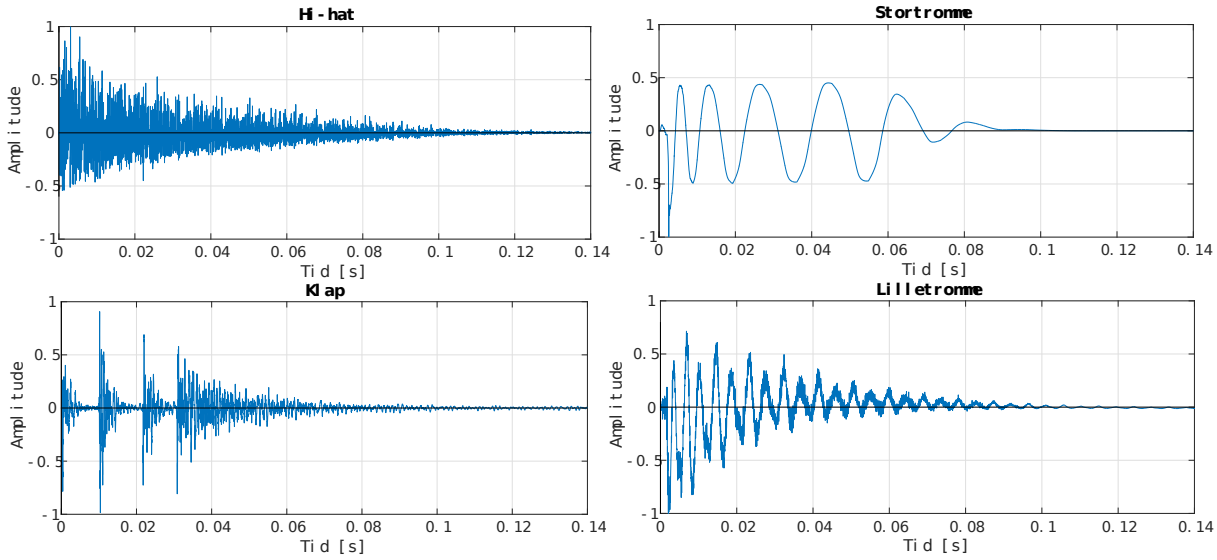
Figur 1.2: Illustration af samspil mellem delmoduler i lydmodulet.

1.1.1 Valg af samples

Ifølge kravspecifikationen i afsnit **??** skal trommemaskinen være i stand til at afspille fire forskellige trommelyde. Dette anses dom en tilstrækkelig mængde til at demonstrere konceptet bag en trommemaskine.

De fire lydfiler kan i princippet være hvilke som helst lyde, men der er i projektet ladet sig inspirere af de gamle trommemaskiner fra 80'erne, og der er af denne grund valgt fire af de elleve trommelyde fra en Roland-909. Disse fire trommelyde er en hi-hat, et klap, en stortromme og en snare.

Disse lyde er valgt, da hi-hat, stortrommen og snaren som regel danner grundstenene i de fleste rytmer. Derudover vælges et klap da klappelyden er meget ikonisk for ældre trommemaskiner. Derudover er de valgte lyde lette at høre forskel på og har meget forskellige frekvensresponser. Dette giver rig mulighed for kreative trommerytmer frem for for eksempel at have to forskellige hi-hats eller snares. Et plot af de forskellige filer kan ses på figur 1.3.



Figur 1.3: test - Den er ikke færdig endnu - hvis den havde været færdig havde den været himmelsk!

Hver lyd er 140 millisekunder lang. Der er valgt at bruge lige lange trommelyde, da der ikke skal holdes styr på hver enkelt trommelyd, og dens størrelse, under implementeringen på hardwaren. Der er taget udgangspunkt i det længste trommelyd, hvorefter de andre lyde er lavet lige så langt. Lydene er indspillet med en samplefrekvens på 44,1 kHz i en opløsning på 16 bits. Dette giver filerne en størrelse på 14.046 bytes (ca. 13.7kB), hvor af de første 46 bytes er head'eren på .wav-filerne, og de resterende bytes er selve lyden. Head'eren vil altså ikke blive brugt, når lydfilerne implementeres på hardwaren, da det ikke er nødvendigt at tage hensyn til information omkring lydfilerne, fordi hardwaren fremstilles specifikt til disse lydfiler som har samme specifikationer.

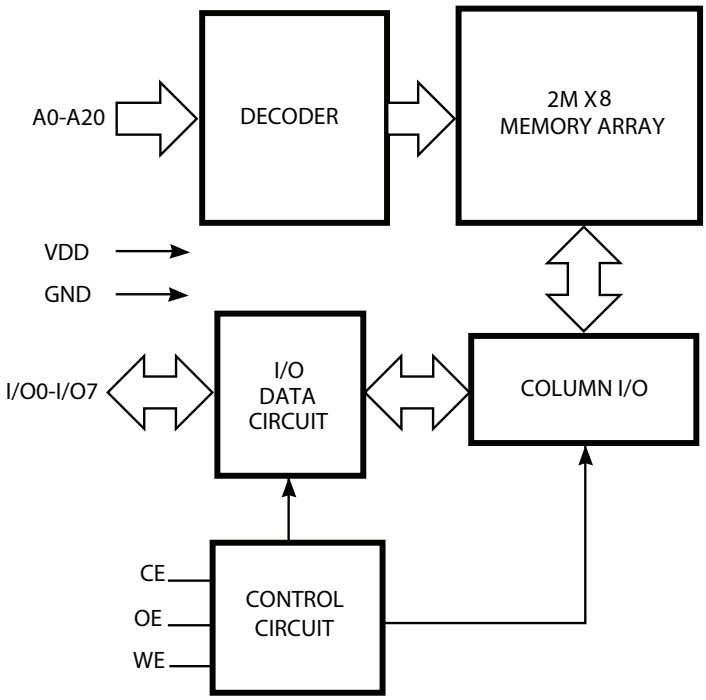
1.1.2 Hurtig Tilgang til Lydfiler

Når lydfilerne skal afspilles, skal filerne være klar til at blive tilgået med kort varsel. Derfor er det nødvendigt at have lydfilerne gemt et sted, hvor de hurtigt kan læses.

Dette kan gøres på flere forskellige måder. Det er muligt benytte selve FPGA'en som RAM på to forskellige måder. Enten som distribueret RAM eller blok-RAM. Distribueret RAM består af hukommelsesblokke fremstillet af FPGA'ens logikceller. Distribueret RAM er meget let at bruge, siden det kan forbindes direkte til systemets øvrige logik. Dette giver også en klar øvre grænse til hvor meget hukommelse der kan bruges. Det er kun muligt at få nogle få kilobytes hukommelse med distribueret RAM, hvilket gør denne type hukommelse uhensigtsmæssig til at lagre lydfiler.

En anden måde at lagre data på FPGA'en er ved benyttelse af dens blok-RAM[1]. Blok-RAM er et lille modul af hukommelse på en FPGA, der sidder separat fra FPGA'ens logikceller. På en Spartan-6 består hver blok-RAM af 18 kb hukommelse. På den anvendte Papilio Duo er der 32 af disse blok-RAM, hvilket svarer til 576 kb eller 72 kB. Dette er stadig ret lavt i forhold til de valgte wav-filer der i alt har en størrelse på 54,69 kB. Især hvis blok-RAMene skal benyttes til andet end lyd. Thomas siger, at dette nok er det eneste belæg vi har for at bruge SRAM.

På grund af manglende hukommelse vælges det at filerne skal lagres på eksternt hukommelse. Papilio Duo'en er udstyret med chippen IS61WV20488BLL[3], der er udstyret med 2 mB Statisk RAM[2]. Et blokdiagram over SRAM-chippen kan ses på figur 1.4



Figur 1.4: Blokdiagram over SRAM-chip. Billed fra kilde **??**

Chippen kontrolleres ved hjælp af fem signaler. Signalet CE står for chip-enable og styrer hvorvidt chippen skal være tændt eller slukket. OE (output-enable) sættes til lav, hvis der ønskes at læses fra chippen, og WE (write-enable) sættes til lav hvis der ønskes at skrive. Her har WE førsteprioritet over chippen, så der vil altid kunne skrives til chippen ligegyldigt hvad OE er. Det læste eller skrevne data sendes over en in/out-port på 8 bit, og nummeret på adressen der tilgås skrives over en 21 bits adresseport.

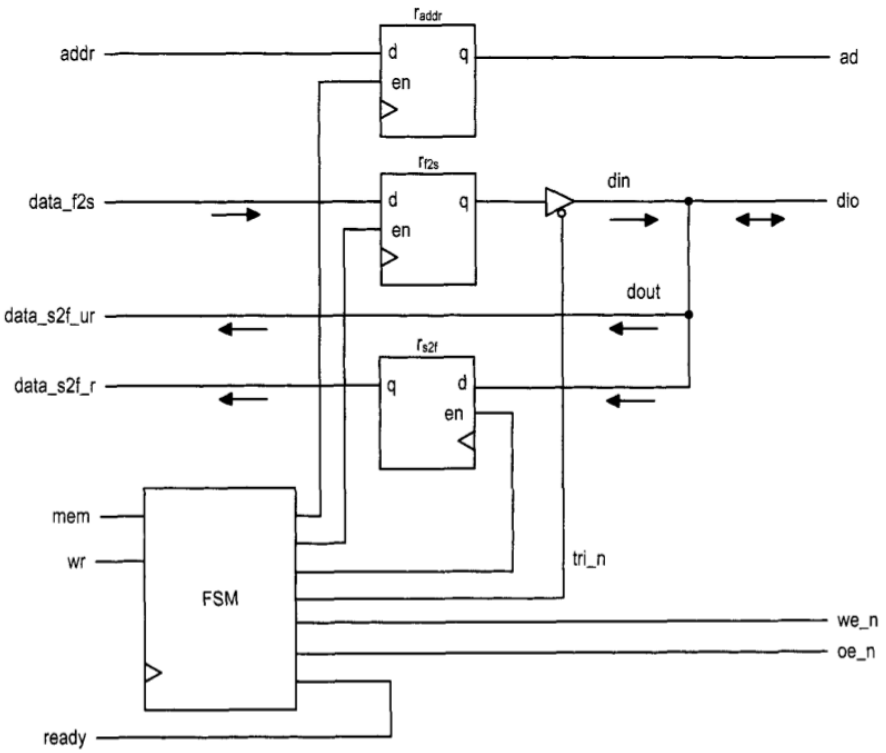
Den anvendte chip har en forholdsvis hurtigt access time på højest 20 ns. Da SRAM-chippen

[1] Nogen gange benævnt som BRAM
[2] SRAM

er asynkron kan der potentielt opstå problemer, når der udføres hukommelsesoperationer på baggrund af eksempelvis en klok fra resten af systemet.

Adresse- og kontrolbit'ene skal holdes fast i et specifikt stykke tid før in- og outputdata kan sendes korrekt. Til dette er der lavet et simpelt interface, til kommunikation med SRAM-chippen fra FPGA'en i VHDL. Interfacet er baseret på et kodeeksempel fra kilde [2] og modificeret til chippen IS61WV20488BLL.

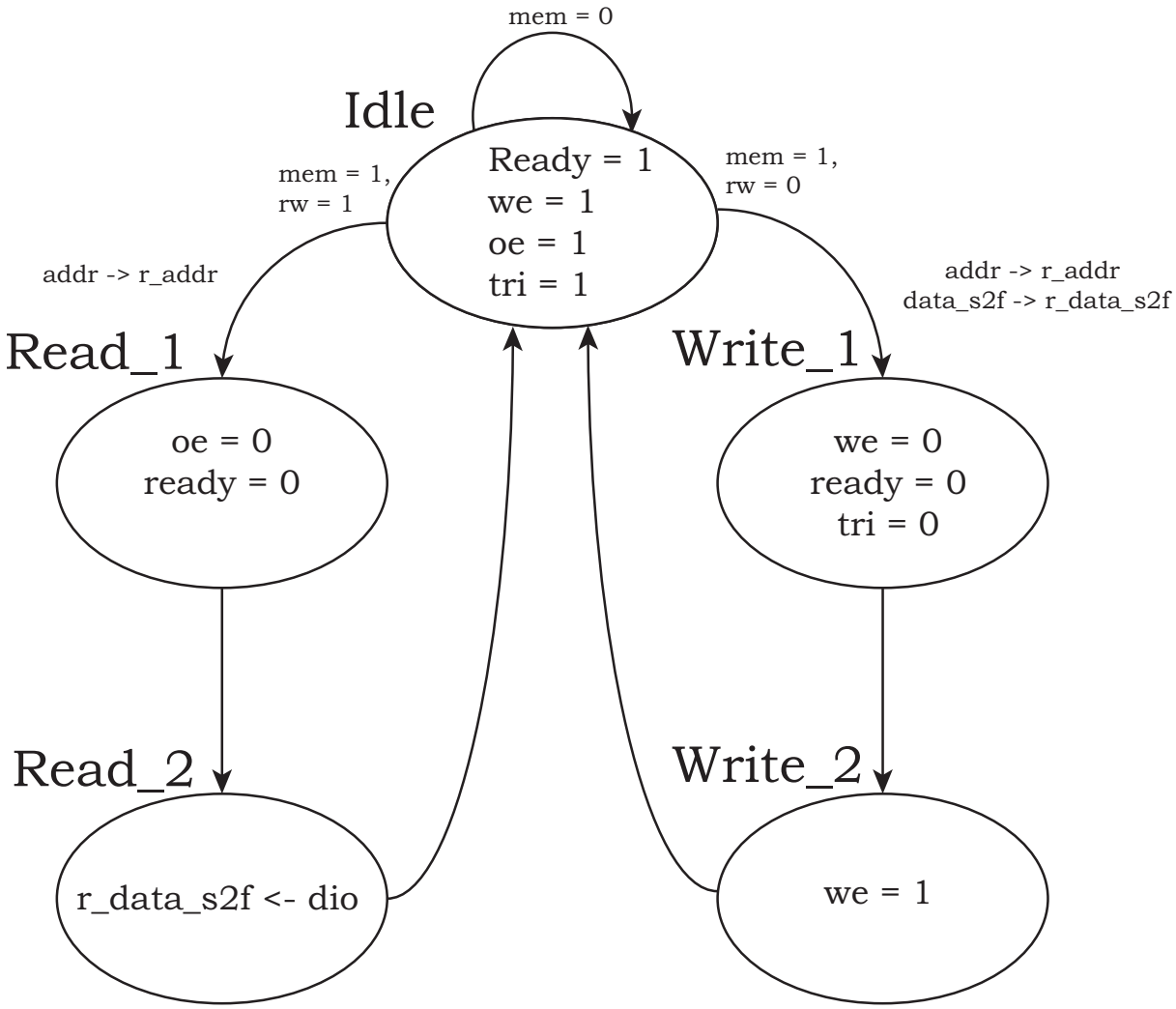
Et blokdiagram over interfacet mellem FPGA'en og SRAM-chippen kan ses på figur 1.5.



Figur 1.5: SRAM-interface. skal nok rettes til, så man kan se hvor mange bits hvert signal er Billede fra kilde [2]

Fra FPGA'en er der tilgang til SRAM'ens adresser ved hjælp af signalet "addr". SRAM'ens in-og output-data tilgås ved signalerne data_f2s (FPGA to SRAM), data_s2f_ur (SRAM to FPGA uregistreret) og data_s2f_ur (SRAM to FPGA registreret). Ud over data_s2f_ur, er alle signalerne koblet til registre for at holde deres værdier faste, indtil de påføres et bestemt signal.

SRAM-interfacet er styres af en en tilstandsmaskine, hvor et tilstandsdiagram over denne kan ses på figur 1.6. Signalet "mem"sættes højt når en hukommelsesoperation skal initieres. Hvorvidt denne operation skal læse eller skrive specificeres af "rw"(read/write). Tilstandsmakinen bruger derefter to klokcyklusser på først at sætte de relevante adresse- og kontrolbits til deres specificerede værdier, og derefter sende/modtage den ønskede data på et register. Da en hukommelsesoperation tager mere end en clockcyclus, kigges der på signalet "ready"som indikerer hvorvidt SRAM-interfacet er i "idle-tilstand og dermed om der sendes nye instrukser.



Figur 1.6: Illustration af tilstandsmaskine der styrer SRAM-interfacet.

Ved starten af hver SRAM-operation latches adresse-signalet "addr"ind på et register som er forbundet til SRAM-chippens adresseben. Dette gøres som en form for sikkerhed, da der på denne måde ikke kan ændres uhensigtsmæssigt i adressen.

Når der skal læses fra SRAM'en startes der med at aktivere "output-enable", hvor SRAM'ens in/output-port latches efter en clockcyclus.

Efter endnu en clockcyclus returnere tilstandsmakinen til "idle-tilstanden.

Når der skal skrives til SRAM'en latches, foruden adressen, signalet "data_f2s".

I tilstanden "Write_1"aktiveres "write-enable"samt signalet "tri"der aktivere forbindelsen mellem det latchede input og SRAM-chippens in/out-port.

Efter en clockcyclus deaktiveres "write-enable", men "tri"holdes stadig lavt. Denne tilstand kaldes "Write_2"Dette er med til at sikre at data-inputtet holdes fast i hele periode hvor "write enable"er aktivt.

Igen returnere tilstandsmakinen til sidst til "Idle-tilstanden.

Det fulde VHDL-modul, kan findes i appendix **??**

1.1.3 Lagring af lydfiler

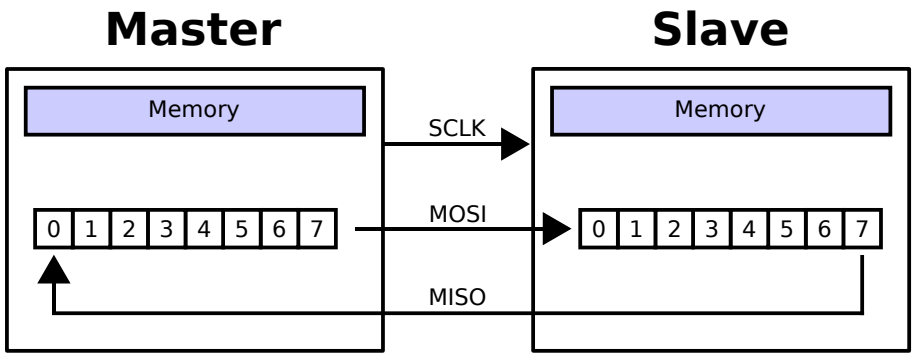
Med mulighed for at FPGA'en kan skrive filer til, og læse filer fra SRAM-chippen, er det oplagt at lydfilerne tilgås på papilioens SRAM, når de skal afspilles på trommemaskinen. SRAM'et har dog det problem, ligesom alt andet volatilt hukommelse, at det forsvinder når der slukkes for strømmen.

Derfor ønskers der udover SRAM'en noget hukommelse som ikke er volatilt, hvor lydfilerne kan gemmes permanent. Dette problem har umiddelbart to løsninger: Enten kan der bruges et eksternt SD-kort og SD-kortlæser, som opkobles til FPGA'en eller arduinoen på papilio'en, eller også bruges der noget af det flash hukommelse, som der i forvejen ligger på papilio'en. Hvad der er fælles for disse er at der under alle omstændigheder skal laves en opstartssekvens, hvor lydfilerne flyttes fra det permanente hukommelse, ind i SRAM'en.

Umiddelbart falder valget på papilio'ens flash hukommelse, da dette allerede er integreret sammen med FGPA'en. Papilio'ens flash hukommelse er på 8 MB. Bit-filer (VHDL kode) fylder hver 333 kB, altså vil der være mulighed for at lagerer flere bit-filer på hukommelsen, men da vi antager at det ikke er nødvendigt at fylde hele hukommelsen med bit-filer, vil der være overskydende plads på hukommelsen. Lydfilerne som man ønsker at ligger på flash hukommelsen er en størrelse på TBD, og da der ikke ønskes mere end 4-8 lydfiler, bør dette kunne lader sig at gøre. Valget falder derfor på papilio'ens flash hukommelse, således det ikke er nødvendigt at bruge tid på at integrer en SD-kortlæser.

Papilio'ens flash hukommelse er en "MX25L6445E"8-PIN SOP som bruger SPI til at kommuniker med. SPI står for "Serial Perpheral Interface bus". Det bygger på et master-slave forhold, hvor masteren kan tilgå flere slaves, dog kræver dette et dedikeret ben per slave. SPI kommunikation består typisk af 4 ben - nogle gange flere, f.eks. kan MX25L6445E bruge optil 6 ben. De 4 typiske ben består af: SCLK (clock), SS/CS (Slave Select/Chip Select), MOSI (Master Out Slave In) og MISO (Master In Slave Out). Når masteren skal kommunikerer med slaven, sættes CS typisk lav, herefter kan masteren sende data med MOSI benet og modtage data med MISO benet. Ved dataoverførsel skriver masteren typisk dens MSB til slavens LSB over MOSI, samtidig skriver slaven dens MSB til masterens LSB. Altså skrives og læses der samtidig, også selv om man blot ønsker at overføre data til slaven.

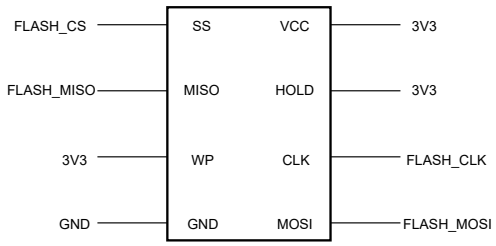
Opdater figur, +1 register



Figur 1.7: SPI kommunikation med 8 bits - billede fra [5]

Selv om MX25L6445E chippen ligger op til at det er muligt bruge 4 data-ben på samme tid, er det dog ikke en mulighed, da 2 af disse på papilio'en er sat til VCC. Dermed er det kun muligt

at bruge chippen på typisk vis. Forbindelserne af flash chippen kan ses på figur 1.8.



Figur 1.8: Flash chippens forbindelser på papilio duo'en

NOGET OM SPI-FLASH LOADER VHA. ARDUINO - det skal en tur med groft sandpapir

For at kunne hente lydfilerne fra flash hukommelsen over i SRAM, skal filerne først ligge på flash hukommelsen. Til at skrive disse filer til hukommelsen, tages en ekstern Arduino Due i brug, da denne har 3.3V logik og tilstrækkelig plads til at opbevare lydfilerne under programmering. Denne process udføres én gang, da det ikke er nødvendigt at ændre i lydfilerne når først de ligger på flash hukommelsen. Arduino Duen vil fremover blive benævnt duen.

For Duen kan få forbindelse til flash hukommelsen udarbejdes et program til FPGA'en, så denne giver forbindelse mellem flash hukommelsen og de eksterne ben hvor duen forbindes med ledninger.XXX REF Der er nu fysisk forbindelse mellem duen og flash hukommelsen.

I flash hukommelsens datablad findes et diagram over flowet når data skal skrives til hukommelsen. Dette kan også findes i appendiks **??**. Der tages udgangspunkt i dette flow, når lydfilerne skal skrives til hukommelsen. Alle kommandoer flash hukommelsen bruger er 8 bit og er angivet i hex-værdier jf. databladet. REF XXX

Når data skal skrives til flash hukommelsen, skal Write Enable (WREN) være sat. Dette gøres med kommandoen `0x06`. Når WREN er sat, er bit 1 i statusregistret være 1. Der tjekkes derfor om WREN lykkedes, ved at læse statusregistret med `0x05`. Når chippen er sat i Write Enable, kan selve programmeringen foregå. Chippen har to kommandoer der skal bruges til at skrive data, nemlig Page-Program og Erase. Grunden til de begge skal tages i brug er fordi Page-Program udelukkende er i stand til at skrive '0'-er til hukommelsen da den AND'er den nye data med den eksisterende. Er området der skrives til med Page-Program derfor ikke '1'ere på alle bits, vil den data der kommer til at ligge på området ikke være den ønskede. Et eksempel på dette kan ses herunder.

Eksempel:

	Eksisterende data	01010101
&	Ny data	11110000
	Resulterende data	01010000

Som det ses, er den nye data og den resulterende data på hukommelsen ikke ens. For Page-Program skriver det rigtige til hukommelsen skal eksisterende data derfor være ene '1'ere. Det er her Erase kommer ind i billedet, da denne skriver '1'ere til et område.

Flash hukommelsen har 4 forskellige Erase funktioner. For at være sikker på hvilken erase-funktion der skal tages i brug, er det nødvendigt at kigge på hvor meget hver funktion sletter sammenlignet med hvor meget lydfilerne fylder. En oversigt over erase-funktionerne kan ses i tabel 1.1. En overordnet tabel over områderne i flash hukommelsen kan ses i bilag **??**.

Tabel 1.1: Tabel over erase-funktionerne

Funktion	Område	Størrelse	Kommando
Sector erase	Sector	4 KB	`0x20`
Block 32 erase	Lille blok	32 KB	`0x52`
Block 64 erase	Stor blok	64 KB	`0xD8`
Chip erase	Alt	64 Mb	`0x60`

Lydfilerne der tages i brug fylder som bekendt? 14.000 byte, så de er for store til at kunne være på én sektor, men for små til at fylde en lille blok. Da flash hukommelsen har 128 store blokke, og det kun er en enkelt af dem der bliver brugt til at gemme programmet til FPGA'en, er der rigelig mange store blokke, så derfor vælges at dedikere en stor blok til hver lydfil, da dette gør adressering overskueligt samt simplificerer erase-processen før hver lydfil skrives til hukommelsen. Derfor bestemmes at lydfilerne får adresserne:

1. `0x110000`
2. `0x120000`
3. `0x130000`
4. `0x140000`

Det er nu gjort klart, at når lydfilerne skal lægges over på flash hukommelsen, skal området først slettes hvorefter der kan skrives data til området, i hele pages ad gangen. Et timingsdiagram kan ses herunder:

INDSÆT TIMINGSDIAGRAM

For at tjekke om det lykkedes at skrive data til flash hukommelsen, er der to forskellige metoder der kan tages i brug. Enten kan området læses, og sammenlignes med det ønskede data, eller `RDSCUR` (Security Register) kan læses for `P_FAIL/E_FAIL`. Skulle noget gå galt, som f.eks. hvis for få data-bit kommer frem, vil flash hukommelsen sætte disse flag. Disse kan læses når som helst og skal slettet af kommandoen `CLSR` (Clear Security Register).

Det vælges at gøre brug af en kombination af de to. Efter hver page er skrevet til hukommelsen, tjekkes `P_FAIL/E_FAIL` og hvis de er sat skrives en enkelte page igen. Når alle lydfiler er skrevet til hukommelsen, læses disse op og sammenlignes med de originale lydfiler. Når `P_FAIL/E_FAIL` tjekkes, gøres det i forbindelse med funktionen `pageProgram`, som returnerer en boolsk værdi hvor `true` betyder der ikke var fejl. Dette bruges som betingelse til et do-while loop som det kan ses i kodeeksempel 1.1

Kodeeksempel 1.1: Tjek af `P_FAIL/E_FAIL` efter pageProgram

```
1 do{
2       lykkesDetAtSkrive = pageProgram((tempAdresse & 0xFFFFFF00), antal256bytes, sampleNr, 0xFF);
3 } while(!lykkesDetAtSkrive);
```

Kodeeksempel kan ses i appendiks **??**.

1.1.4 picoBlaze

Nu mangler der bare en måde at få filerne på Flash-hukommelsen over på FPGA'en. Det logiske valg er at lade lydfilerne blive på FPGA'en så længe trommemaskinen er tændt, og da FPGA'ens hukommelse er statisk er det kun nødvendigt at overføre filerne én gang.

Da denne handling kun skal udføres én gang og at filerne kun kan sendes sekventielt på baggrund af at SPI-Flash og FPGA'ens interfaces, vælges der at denne handling skal laves som noget der kan udføre disse sekventielle instruktioner én gang, og derefter ikke gøre noget.

Derfor vælges det at anvende en mikroprocesser til at lægge lydfilerne fra flash-hukommelsen over til FPGA'en, når trommemaskinen tændes.

Der er to oplagte valg til en mikroprocesser givet det hardware, der er tilgængelig i projektet. For det første kunne man anvende arduino'en, Der er en 8-bits mikroprocesser, med en maksimal clock-frekvens på 16 MHz er dette rigtigt?. Fordelen ved at bruge arduino'en er at den er meget simpel at programmere, og at den allerede ligger klar på papilio-boardet.

Derudover kunne kan en processor implementeres af FPGA'ens logikceller (en såkaldt soft-processor). Dette har den fordel at der ikke skal interfaces flere eksterne moduler, og at koblingen mellem SRAM og Flash-hukommelse bliver nemt da disse i forvejen er forbundet til FPGA'en. Derudover kan soft-processorens clock-frekvens sættes til at være clock'en på papilio'en, som har frekvensen 32 MHz. Dette er en dobbelt eller hvad? så hurtigt som clock-frekvensen på Arduionen.

Xilinx har lavet to typer mikroprocessore, der frit kan anvendes på Xilinx's FPGA'er (såsom Spartan-6). Disse kaldes henholdsvist microBlaze og picoBlaze. MicroBlaze er en 32-bit processor, og en del mere kompleks end en picoBlaze der kun er en 8-bits processor. En MicroBlaze er designet til at blive programmeret i c, hvorimod picoBlaze er lavet til a køre mindre programmer skrevet i assembly. Dog fylder en microBlaze en stor del af en FPGA, og da mikroprocessoren blot skal udføre et meget simpelt arbejde, en enkelt gang under systemets opstart vælges der at anvende en picoBlaze.

Fordelen ved at implementere en soft-processor på FPGA'en, er at det ikke vil være nødvendigt, at implementerer fysiske I/O's på FPGA'en, men der kan dette gøres internet i FPGA'en, hvilket også betyder at det er muligt, at give soft-processoren lagt flere I/O, hvis dette skulle vise sig nødvendigt.

En picoBlaze er implemeteret som en fuldt udbygget microcontroller med program-hukommelse og en smule RAM til at gemme variabler.

Den specifikke udgave af picoBlaze skrevet til en spartan-6 FPGA, kaldes KCPSM6[3]. KCPSM6 kan eksekvere et program med op til 4096 instruktioner, hvor hver instruktion fylder 18 bit. Hver instruktion udføres på 2 clock-cyclusser hvilket i dette projekts tilfælde giver 16 Mips. Dog kan en reduceret clockfrekvens opstå med et program med over 2048 instruktioner på en Spartan-6 FPGA.

KCPSM6 har to registerbanke, A og B, af 16 registre til generelt brug. Registrerne benævnes "s1,s2,...,sF"[1].

[3]KCPSM står for "Constant(K) Coded Programmable State Machine"(førhen "Ken Chapman's PSM")

picoBlaze'ens input og output styres ved hjælp af tre kontrolsignaler: write_strope, k_write_strope og read_strope, der hver viser om processeren har taget imod et input/sendt et output. Desuden bruges den 8-bit lange vektor port_id, til at styre hvor der skal læses fra/skrives til.

Det er også muligt at bruge "interrupts"og "sleeps"i forbindelse med picoBlaze'en men da dette ikke nvende i dette projekt vil disse ikke blive beskrevet videre.

Et diagram over KCPSM6's arkitektur kan ses på figur 1.9

Figur 1.9: Blokdiagram over en picoBlaze Mikrokontroller. Billede fra [1]

Implementering af en PicoBlaze er relativt simpelt. Der kræves ikke andet en at tilføje to VHDL-

moduler: Selve KCPSM6'en samt et modul til programmet's ROM. Sammen med KCPSM6'en tilbyder Xilinx et program der automatisk læser en Assemply-fil og oversætter det til et stykke program-hukkomelse til picoBlaze'en.

Da picoBlaze er en RISC-processor[4], kan den kun eksevere en begrænset mængde maskininstruktioner. KCPSM6's instructionssæt kan ses i appendix **??**

picoBlaze i sammenspil med SPI-flash og SRAM

For at picoBlaze'en skal være i stand til at overføre lydfilerne fra Flash-hukommelsen til SRAM'en, skal den selvfølgelig have en hardware-forbindelse til disse enheder. Derudover ønskes det også at picoBlaze'en kan kommunikere med resten af trommemaskinen, for at fortælle systemet hvornår lydfilerne er klar til at blive afspillet.

En måde hvorpå en picoBlaze's Input- og Outputport, kan forbindes med forskellige signaler i FPGA'en, er ved en proces, der forbinder portene på baggrund af bit'ene i picoBlaze'ens "port_id".

Sammen med picoBlaze leveres en skabelon til dette, der er brugt til at implementere forbindelsen mellem picoBlaze, og resten af FPGA'en.

Dette implementeres som to separate processer til input-porte og output-porte.

Inputprocessen fungere ved at værdien af "port_id"bliver på hver clockcyclus . Input-porten sættes derefter til et specifikt signal henfør denne værdi. Dette er eksemplifiseret i kodeeksempel 1.2

Kodeeksempel 1.2: picoBlaze inputporte

```
1
2  input_ports: process(clk)
3  begin
4    if clk'event and clk = '1' then
5      case port_id(6) is
6        when '1' => in_port(7) <= FPGA_MISO;; -- reads the content of "fpga_MISO" on port "7"
7        when others => in_port <= "XXXXXXXX";
8      end case;
9    end if;
10 end process input_ports;
11
12
13
14
15
16                          When others => in_port <= "XXXXXXXX";
```

I eksemplet bruges port_id(6), eftersom port_id(5) bruges som output til flash hukommelsen. I princippet kan der bruges samme port_id, hvor forskellen er om det er in- eller output, men på denne måde er der en tydelig forskel, hvilket gør det lettere at kende forskel på in- og output. Herefter sættes dette input til in_port(7), ben 7 bruges fordi de i forvejen lavet rutiner af Xilinx, til SPI kommunikation ligger op til dette, dermed behøves der ikke ændres i kode, og dermed

[4]Reduced instruction set computer

mindsket muligheden for introduktion af fejl. Ben 7 bruges til MISO, altså kommunikation fra SPI-flash til picoBlaze processoren.

For outputportene er processen meget lignende den tidligere proces. Den eneste markante forskel[5] er at der også tjekkes om hvorvidt signalet "write strobe"er sat til høj, hvilket sker hvergang picoBlaze'en har eksekveret en output-instruktion, eftersom write strobe og port_id bliver AND'et sammen og viker som chip enable, Et eksempel på dette ses i kodeeksempel 1.3

Kodeeksempel 1.3: picoBlaze outputporte

```
output_ports: process(clk)
begin
  if clk'event and clk = '1' then
    if write_strobe = '1' then
       if port_id(5) = '1' then
                              FPGA_SCK <= out_port(0);
                              FPGA_CS <= out_port(1);
                              FPGA_MOSI <= out_port(7);
       end if;
    end if;
  end if;
end process output_ports;
```

Ligesom før bruges port_id til, at specificerer hvilken outputport man ønsker at skrive fra - i dette tilfælde port_id(5), som forklaret tidligere. Herefter bruges out_port(0) som SCK til SPI-flashen, out_port(1) som CS og out_port(7) til MOSI altså kommunikation fra picoBlaze processoren til SPI-flashen. Disse outputports er endnu en gang valgt, pga. Xilinx rutiner.

Disse processer danner grundlaget til hvordan hver separat port forbindes. Og der kan derfor kigges på hvordan picoBlaze processoren er programmeret.

I dette kodeeksempel, vises der hvordan der modtages data fra SPI-flashen til picoBlaze processoren.

Kodeeksempel 1.4: SPI-flash input

```

some code - som bare er fukin' laeaeaeaekkert
```

I det næste kodeeksmpel, vises der hvordan der sendes data til SPI-flashen fra picoBlaze processoren.

Kodeeksempel 1.5: SPI-flash output

```


some code
content...
```

I begge eksempler kaldes der til Xilinx's rutiner, dette vil vi her dykke lidt nærmere ned i.

Kodeeksempel 1.6: læse og skrive rutine kald

[5] Begge processer kunne i princippet være lavet ved hjælp af både if- og case-statements

```
1
2  write_spi_byte: LOAD sD, s2              ;Preserve data to be written
3                  CALL SPI_disable         ;ensure known state of bus and s0 register
4                  CALL set_spi_flash_WREN  ;set write enable latch
5                  LOAD s2, 02              ;PP instruction
6                  CALL SPI_FLASH_tx_rx     ;transmit instruction
7                  LOAD s2, s9              ;Transmit 24-bit address
8                  CALL SPI_FLASH_tx_rx
9                  LOAD s2, s8
10                 CALL SPI_FLASH_tx_rx
11                 LOAD s2, s7
12                 CALL SPI_FLASH_tx_rx
13                 LOAD s2, sD              ;Transmit data byte **
14                 CALL SPI_FLASH_tx_rx
15                 CALL SPI_disable         ;terminate PP transaction
16
17
18 read_spi_byte:  CALL SPI_disable         ;ensure known state of bus and s0 register
19                 LOAD s2, 03              ;READ instruction
20                 CALL SPI_FLASH_tx_rx     ;transmit instruction
21                 LOAD s2, s9              ;Transmit 24-bit address
22                 CALL SPI_FLASH_tx_rx
23                 LOAD s2, s8
24                 CALL SPI_FLASH_tx_rx
25                 LOAD s2, s7
26                 CALL SPI_FLASH_tx_rx
27                 CALL SPI_FLASH_tx_rx     ;read data byte **
28                 JUMP SPI_disable         ;terminate transaction (includes return)
```

Som man kan se minder "read_spi_byte"og "write_spi_byte"meget om hinanden. Den største forskel på de to, er at når der skal skrive til SPI-flashen, sende koden 02, og WREN kaldes. Når der skal læses fra SPI-flashen sendes koden 03. I begge funktions kald kan man se, at registerne s7-s9 bruges til at beskrive, den adresse man ønsker at læse fra eller skrive til. Herefter kaldes der på funktionen "SPI_FLASH_tx_rx", som skrive disse adresser til SPI-flashen, så den ved hvor der skal skrives/læses fra. Når der skrives til SPI-flashen, bruges der yderligere den data som står i register s2. Og når der læses fra den, bliver dataet på den givende adresse igen læse over i register s2. Altså bruges s2 konsekvent som en slags "data register"hvor s7-s9 bruges som en form for "adresse register".

Grunden til at disse to funktioner minder så meget om hinanden, er fordi man i princippet kan læse og skrive til SPI-flashen samtidig, som nævnt tidligere. Dette bliver langt tydeligere hvis man se på funktionen "SPI_FLASH_tx_rx"

Kodeeksempel 1.7: SPI_FLASH_tx_rx

```
1
2  SPI_FLASH_tx_rx: LOAD s1, 08                      ;8-bits to transmit and receive
3     next_SPI_FLASH_bit: LOAD s0, s2              ;prepare next bit to transmit
4                         AND s0, spi_mosi         ;isolates data bit and spi_cs_b = 0
5                         OUTPUT s0, SPI_output_port ;output data bit ready to be used on rising
                              clock edge
6                         INPUT s3, SPI_data_in_port ;read input bit
7                         TEST s3, spi_miso        ;carry flag becomes value of received bit
8                         SLA s2                   ;shift new data into result and move to next
                              transmit bit
```

```
9                   CALL SPI_clock_pulse       ;pulse spi_clk High
10                  SUB s1, 01                 ;count bits
11                  JUMP NZ, next_SPI_FLASH_bit ;repeat until last bit
12                  RETURN
```

Når SPI_FLASH_tx_rx kaldes defineres der først, at det er 8 bits der behandles. Det data der står på register s2 bliver kopieret over i s0, og herefter bliver MSB outputtet på "SPI_output_port"som tidligere er defineret til værdien 32, hvilket svare til 00100000 i binær, og dermed svare dette til port_id(5) i VHDL koden, som set tidligere. Ligeledes svare "SPI_data_in_port"til værdien 64, altså port_id(6) i VHDL. Data på input benet bliver altså læst over i s3, og carry'en mellem s3 og konstanten spi_miso, som har værdien 10000000 binært, bliver bit-shiftet ind i s2. Altså vil denne funktion både skrive og læse fra SPI-flashen samtidig, altså på samme clock-pulse, som bliver kaldt "SPI_clock_pulse"lige efter der er bit-shiftet. Dette gentages 8 gange, end til en byte er læst fra eller skrevet til s2 registeret. Hvorefter funktionen returnere til det sted hvor den blev kaldt.

Den eneste grund til, at der ikke skrives og læses samtidig, er at den første kode der sendes - altså 02 for at skrive og 03 for at læse fra SPI-flashen. Det er også derfor, at læse og skrive funktionskaldene read_spi_byte og write_spi_byte, minder så meget om hinanden, eftersom kerne funktionen "SPI_FLASH_tx_rx"essentielt gør begge dele samtidig.

Derudover skal picoBlaze'en have forbindelse til SRAM'en via SRAM-interfacet beskrevet i afsnit 1.1.2. Fra picoBlaze'ens side er det kun nødtvendigt at kunne skrive til SRAM'ens adresse via signalet "addr"og SRAM-interfacets data input. Andre signaler der styrer SRAM-interfacet kan implementeres ved hjælp af hardware i vhdl. Eksempelvis skal signalet "mem", der initiere en SRAM-operation, blot sættes høj hver gang der sendes en byte fra picoBlaz'en. Signalet "rw"skal altid være lav så længe picoBlaze'en er i gang med at sende lyd-filer.

Da man i VHDL kun kan ændre på signaler igennem én proces, og da SRAM-interfacet også skal tilgås når lyd skal afspilles er det nødvendigt at opdele signalerne sådan at SRAM'en udelukkende kan tilgås af picoBlaze'en under trommemaskinens opstart, og at SRAM'en kun kan tilgås efter opstart, når der skal afspilles lyde.

Dette gøres ved signalet "boot", som skal kunne ændres fra 0 til 1 af picoBlaze'en når alle filer er overført. Derfor skal denne kobles til et af picoBlaze'ens outputporte.

De eneste SRAM-signaler der skal tilgås både når picoBlaze'en sender lydfiler, er "mem"og "addr". Dette skyldes at picoBlaze'en udelukkende skal skrive til SRAM'en, og resten af systemet skal udelukkende læse fra SRAM'en. Dette medføre at "data_f2s"kun skal tilgås fra picoBlaze'en, og at "rw"konstant skal have samme værdi som "boot"[6].

Derfor skal "addr"og "mem", hver især opdeles i to signaler, der enten kan tilgås af picoBlaze'en eller resten af lydmodulet. Dertil forbindes kun et af de to signaler til SRAM-interfacet, på baggrund af værdien af "bbot".

Måden hvorpå dette er implementeret i VHDL, kan ses i kodeeksempel 1.8

Kodeeksempel 1.8: Separering af adgang til SRAM

```
1 with boot select
```

[6]Dette er fordi "rw"skal være 0 når der skal skrives til SRAM'en og 1 når der skal læses derfra.

```
2               mem <= bootmem when '0',
3                          playmem when '1';
4
5 with boot select
6               addr <= bootaddr when '0',
7                          playaddr when '1';
8 -- rw skal altid vaere 1 efter bootsekvensen
9 rw <= boot;
```

Da SRAM-chippens adresse, skal specificeres med 21 bits, og picoBlaze er en 8 bits processor er det nødvendigt at bruge tre instruktioner for at sætte adressen. Derfor speciferes der tre outputporte til de 8 lavest betydende bit på adressen, de 8 mellemste bits, og de sidste 5 højeste.

Selve Data'en der skal lægges i SRAM'ene skal også have et bestemt "port_id"der forbinder picoBlaze'ens output til data-inputtet på SRAM-interfacet "data_f2s". Hver gang der sendes en byte fra picoBlaze'en skal der udføres en skrive-operation. Hvis man dog bare sætter "mem"til 1 hver gang der skrives data, kan der opstå problemer hvis ikke "mem"sættes til 0 igen efter en operation er påbegyndt. Da picoBlaze'ens "port_id"kan tage længere tid om at opdatere end den tid det tager at fuldføre end SRAM-operation, kan dette give problemer hvis "mem"kun ændres ved en ændring af "port_id"Dette er på grund af at det tager flere omgange at sætte SRAM'ens adresse. Derfor skal man være sikker på at adressen har den rigtige værdi, før der påbegyndes en SRAM-operation.

Dette løses ved at indføre et nyt signal kaldet "input_ready", der sættes høj hvergang en databyte er blevet sendt fra picoBlaze'en.

Dertil oprettes en parallel proces der opdateres ved hver ændring i "input_ready"og "ready"signalet fra SRAM-interfacet. Processen sætter "bootmem"høj , når både "input_ready"og "ready"er høje. Ellers sættes "bootmem"lav. så snart en SRAM-operation er påbegyndt vil "ready"sættes lavt, hvilket medføre at "bootmem"også sættes lavt. Processen kan ses i kodeeksempel 1.9

Kodeeksempel 1.9: Timingkontrol mellem picoBlaze og SRAM-interface

```
1 sram_operation: process(ready,input_ready,output_ready) -- control of timing between picoblaze
      and sram - might be unneccesary
2 begin
3               if (ready = '1' and input_ready = '1') then
4                       bootmem <= '1';
5               else
6                       bootmem <= '0';
7               end if;
8 end process sram_operation;
```

måske et moduldiagran med fine ledninger?

Programering af picoBlaze

I sig selv er en picoBlaze ikke meget værd. For at få picoBlaze'en til at udføre det ønskede arbejde er det nødvendgt først at programmere den. picoBlaze programmeres som nævnt tidligere i sproget assembly.

Programmets funktion er ret simpel. picoBlaze'en skal tage imod en byte data fra en adresse på SPI-flash'en og gemme det i et register, og derefter sende denne byte ud til SRAM'ene. Derefter skal både SRAM'ens og SPI-flashens adresse "tælle op"sådan at den næste byte i lydfilen bliver læst fra Flash-hukkomelsen, og bliver sendt til den næste adresse på SRAM'ene.

Xilinx har lavet et reference design til kommunikation med flash hukommelsen vha. SPI kaldet KC705. I dette reference design findes "N25Q128_SPI_routines", som er assembly rutiner der kan kaldes af en picoBlaze.

herp derp call_spi_bye hurr durr gem data i s2, miii miii miii s7,s8,s9 adresser

Jeg vil lige prøve et lille magitrick på mandag inden jeg skriver om main–funktionen.

Adresserne er som beskrevet tidligere benævnt med tre bytes der gemmes i tre registre. For at tælle en op på adressen indtil hver byte er sendt, er adressen nødt til at behandles som et langt 24-bits tal. Til dette anvendes følgende funktion der kan ses i eksempel **??**.

Kodeeksempel 1.10: Optælling af adresser

```
1  plusbyte:
2          COMPARE S4, FF
3          JUMP Z, pluspage;
4          ADD s4, 01;
5          ADD S7, 01;
6          JUMP main
7  pluspage:
8          LOAD S4, 00;
9          LOAD S7, 00;
10         COMPARE S5, 36
11         JUMP Z, plussector
12         ADD S5, 01
13         ADD S8, 01
14         JUMP main
15 plussector:
16         LOAD S5, 00;
17         LOAD S8, 00;
18         COMPARE S6, 04
19         JUMP Z, idle;
20         ADD S6, 01
21         ADD S9, 01
22         JUMP main
```

Efter en byte er sendt tjekkes der først om den laveste byte har sin maksimale værdi 255. hvis den er mindre end dette lægges der 1 til registret, og programmet hopper tilbage til main-funktionen. hvis den laveste byte er 255 betyder det at den skal "overflow"e op i næste register. I dette tilfælde hopper programmet til næste del af funktionen, der først nulstiller den laveste byte og derefter igen tjekker om den mellemste byte har noget sin maksimale værdi, for at se hvorvidt registret skal nulstille og hoppe videre til højeste byte, eller lægge 1 til mellemste byte og hoppe tilbage til main-funktionen. Den højeste byte, der svarer til sektore på Flash-memory, kun skal tælles op når enden på en lydfil er nået, da starten på hver lydfil ligger på starten af en sektor. Hver lydfil består af 14.046 bytes, som beskrevet i 1.1.1. I hexadecimal har 14.046 værdien 0x36DE. For at holde programmet simpelt tælles der op til 0x36FF, hvilet medfører at der sendes 33 bytes for

meget[7]. Dette vil medføre at der sendes $4 \cdot 33 = 132$ bytes for meget. Dette antal bytes er meget lille i forhold til de omkring 56000 bytes der sendes i alt.

Den højeste byte bliver behandlet på samme måde. Når højeste byte har noget sin maksimale værdi, må alle bytes i lydfilerne være læst og sendt. Dermed er programmet færdig og hopper til linjen "idle", der er endestationen for denne picoBlaze's program.

Kodeeksempel 1.11: Afslutning af Assembly-programmet

```
1 idle:
2         OUTPUT sE, BOOT
3 idle2:
4         JUMP idle2
```

I registeret "sE"er der konstant gemt hexværdien 0x01. Når denne outputtes på "boot-porten fortælles resten af trommemaskinen at picoBlaze'en har gemt alle lydfiler i SRAM'ene. Herefter nås linjen "idle2", der for picoBlaze'en til at køre i tomgang til evig tid[8] I et reelt produkt ville man måske nærmere lave en funktion der kan "slukke"for picoBlaze'en ved at sætte picoBlaze'ens "sleep-signal permanent højt for at minimere trommemaskinens strømforbrug. Dog er dette ikke fokuset i dette projekt og derfor er den nemmeste løsning til deativering af picoBlaze'en valgt.

ved ikke lige hvor det her passer ind —> Reference designet til VHDL koden, er umiddelbart ikke brugbar, da visse dele af modulet ikke fungere på en Spartan-6. <- er det her nødvendigt at have med? hvis ja, hvorfor kan den ikke anvendes? Derfor programmere vi istedet selv de 4 ben hvad menes der med at programmere ben?, som skal bruges som interface mellem flash og picoBlaze, på samme måde reference designet umiddelbart selv ligger optil, således assembly-rutinerne kun skal ændres minimalt.

1.1.5 Fra lydfil til lydGuf

Ikke helt sikker på hvad der skal stå her men forestillede mig at det skulle være hvordan: lydfilerne blev til -> skrives fra arduino med LUT -> FLASH (picoBlaze) -> SRAM -> KNUD -> DAC -> LYD

arduino som fil loader

Som tidligere skrevet bruges arduinoen on board eller ej, som en form for "lydfil loader". Dette gøres ved at forbinde den, vha. FPGA'en, direkte til flash hukommelsen. Lydfilerne skrives så på specifikke adresser, så de let kan findes senere. Lydfilerne befinder sig på arduino'en som et LUT (Look Up Table), for hver sample. Når lydfilerne er skrevet over til flash hukommelsen skrivebeskyttes de, så det ikke er muligt at slette lyd filerne ved et uheld.

Her efter en mere detaljeret beskrivelse af: adresser, lydfilerne (LUT/HEX), hvordan de skrivebeskyttes - beskriv vitale funktioner evt. med små bider af kode

Fra flash hukommelse til SRAM

Når lydfilerne er lagt over på flash hukommelsen bruges picoBlaze processoren, til at føre dem over til SRAM'ene. Dette gøres ved at læse på de før bestemte specifikke adresser, disse læses

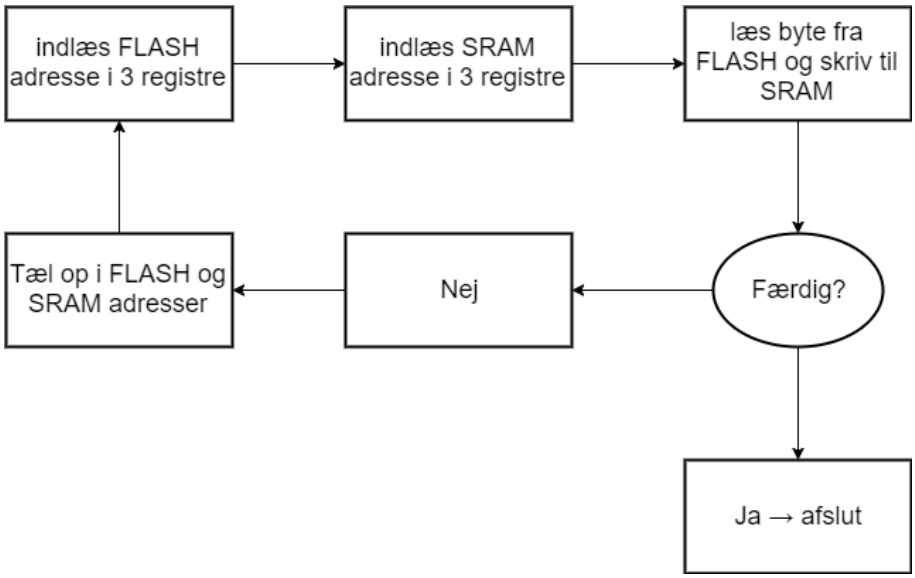
[7] $FF - DE = 255 - 222 = 33$
[8] I virkeligheden er det kun indtil FPGA'en nulstilles.

vha. et stykke assembly kode som kalder de rutiner som er lavet af Xilinx. Disse rutiner gør det forholdsvis let, at læse data fra flash hukommelsen og over i et bestemt register på picoBlaze processoren. Dette register kan så skrives over til SRAM'ene.

koden detaljer eller eksempel??? Når picoBlaze processoren skal læse filer fra flash hukommelsen skal der bruges en 24-bit adresse, dette opnåes ved at sætte indlæse 8-bit i tre register, herefter kaldes der på en læse-rutine. Denne skrive en læse-instruktion til flash hukommelsen, hvorefter den bruger de tre register til at bestemme den ønskede adresse, så læses den byte som ligger på adressen over i et fjerde register.

Denne byte skrives så over i SRAM'ene. Dette gøres ved at give SRAM-interfacet en 21-bit adresse vha. tre registre ligesom tidligere, samtidig med byten, alt dette gøres parallelt.

vis et super crisp kode eksempel eller pseudokode/flowchart



Figur 1.10: Flowchart af opstartssekvensen - Er den gu' nok?

1.1.6 Design af D/A-konverter

Rasmus skal huske at skrive at der ikke er fokus på DAC i afgrænsning

For at trommemaskinen skal kunne afspille de valgte waw-filer i en højtaler er det nødvendigt at omdanne waw-filerne til et analogt signal. Da der anvendes lydfiler i waw-formatet er det ikke nødtvendigt med nogen form for dekodning af filerne, og der kan blot nøjes med en D/A-konverter. Da dette projekt ikke lægger specielt meget op til analog elektronik, og da der ønskes en forhold realistisk men hurtig gengivelse af de digitale lydfiler, vælges der at anvende en allerede eksisterende D/A-konverter i lydmodulet.

Da lydfilerne er samplet med en ordlængde på 16 bits, og da det er lydfiler der arbejdes med, falder det logiske valg på D/A-konverteren TDA1541A [4], givet de D/A-konvertere der er til rådighed. En TDA1541A var tidligere en ofte anvendt D/A-konverter i digitalt Hi-Fi-udstyr og egner sig derfor godt til dette projekt.

TDA1541A'en har tre måder at operere på. Fælles for dem alle er at det digitale input læses serielt, og at afspilningsfrekvensen bliver styret af en bitclock "BCK". Derudover er det altid

muligt at afspille to kanaler (stereo), i en bitdybde på max 16 bit. D/A-konverterens måde at operere på styres ved hjælp af et enkelt ben på IC'en "OB/TWC", ved at sætte det til hhv. 5 V, 0 V eller -5 V.

Første mulighed er at styre starten på et nyt "ord", ved at benytte et ben som "latch enable". "Latch enable"sættes høj i en clockcyclus for at markere starten på et nyt "ord". I denne tilstand sendes hver stereokanal på to separate ben. Dog kan D/A-konverteren i denne tilstand kun behandle data representeret som forskudt binær[9].

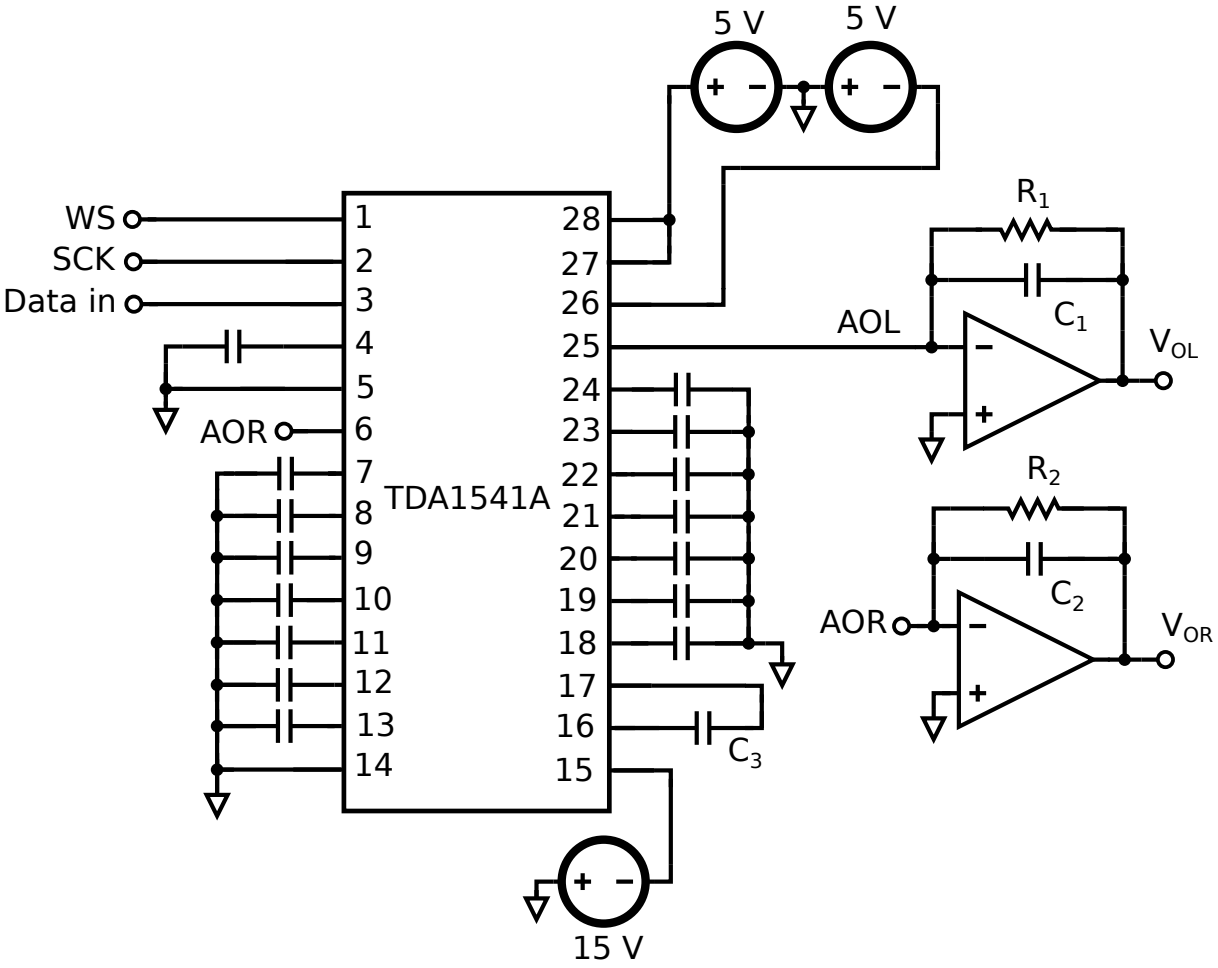
Anden og tredje mulighed er ens på nær hvordan input-data representeres binært (enten forskudt eller i to's kompliment). I disse tilstande bliver input-data læst på enkelt ben, styret med et "word select-signal. "Word select"sættes lav når data sendes på venstre kanal og høj når data sendes på højre. Med data representeret som to's kompliment, er D/A-konverterens input ækvivalent med kommunikationsstandarden I²S.[10] Denne standard er beskrevet nærmere i afsnit **??**.

Da WAV-filer i forvejen er i to's kompliment, vælges D/A-konverteren til at styres med "word select"i to's komplimet, d.v.s. I²S.

For at anvende D/A-konverteren i denne tilstand skal TDA1541Aen opkobles som vist på figur 1.11 jævnfør databladet [4]

[9] En måde at representere binære tal på, hvor 00...0 svarer til den største negative værdi.

[10] Her svarer bitclocken "BCK"til det der kaldes "Serial Clock"SCK jævnfør I²S-standardan beskrevet i afsnit **??**.



Figur 1.11: TDA1541A opkoblet til I²S-format

Da de fleste ben på IC'en enten går til DC eller er afkoblet, vil de individuelle ben ikke beskrives i dybden.

På begge udgange er der, som det kan ses på figur 1.11, placeret et aktivt førsteordens lavpasfilter, til at fjerne frekvenser over det hørbare frekvensområde. Modstandene R_1 og R_2 er ii databladet specficeret til 1,8 kΩ og kondensatorne C_1 og C_2 er specificeret til 2,2 nF.

Dette giver lavpasfiltret en knækfrekvens på:

$$f_c = \frac{1}{2\pi R_1 C_1} = \frac{1}{2\pi \cdot 1,8\ \text{k}\Omega \cdot 2,2\ \text{nF}} = 40,2\ \text{kHz} \tag{1.1.1}$$

Dette virker meget rimeligt da 40 kHz anses for at ligge tilpas meget over 20 kHz til at filtret ikke har en markant effekt på det ønskede frekvensområde da forstærkningen vil være faldet 3 dB ved knækfrekvensen.

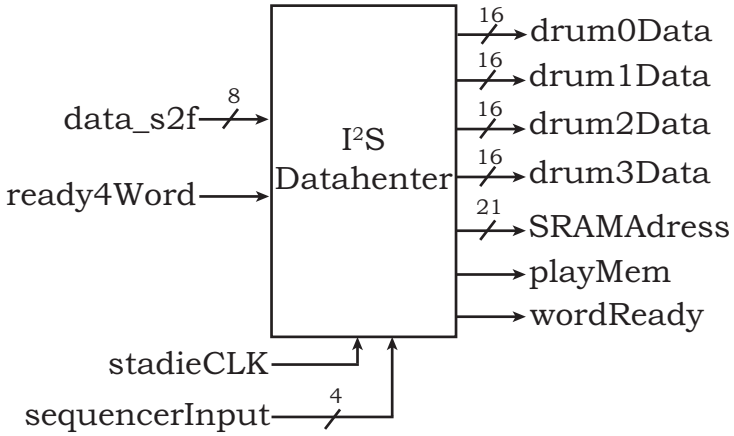
Kondensatoren C_3 er specificeret til 470 pF og de 15 resterende afkoblingskondensatore er sat til 100 nF.

1.1.7 Kobling mellem SRAM og D/A-konverter

kunne eventuelt komme efter i2s-modul Med valget af D/A-konverter medfører dette at udgangstrinet skal overholde I²S-standarden[11], og at der skal leveres trommesamples til D/A-konverteren. Dette deles op i to moduler, hvor det ene modul skal hente data, og det andet skal konvertere det så det stemmer overens med I²S.

Datahentning fra SRAM

De fire valgte trommelyde ligger på Papilioens SRAM, men fordi SRAM'et kun kan gemme én byte pr. adresse[12], er det nødvendigt at opdele hvert trommesample i to. Derudover indeholder trommemaskinen fire trommelyde, så der benyttes fire buffers til midlertidig opbevaring af samples. Disse hedder drumXData, hvor X er et tal mellem 0 og 3. På figur 1.12 ses moduldiagrammet for hentning af data fra SRAM'et.



Figur 1.12: Moduldiagram af tilstandsmaskinen der henter data til I²S-modulet. eventuelt fjerne playMem

Dette modul er fremstillet som en endelig tilstandsmaskine. Til hver af de fire trommelyde er der oprettet to konstanter og én variable. De to konstanter er RAM-adressen hvor trommelyden starter og ender, og variablen indikerer lokationen i trommelyden. Disse hedder drumXStartAddr (trommestart), drumXEndAddr (trommeslut) og drumXCurrntAddr (nuværende adresse). Her er "X"et tal mellem 0 og 3 der afgør hvilken tromme der er tale om. Derudover har tilstandsmaskinen fire signaler der afgør om deres respektive tromme er "tændt"(1) eller "slukket"(0). Disse signaler bliver fremadrettet kaldes drumOn(X), hvor X igen kan være mellem 0 og tre 3, hvilket angiver hvilken tromme der er tale om.

Efter at have blivet fyldt op med data fra picoBlaze'en, er der ingen grund til at overskrive SRAM'ene, og de vil dermed virke som ROM På grund af dette kan SRAM-interfacets kontrolsignaler abstraheres yderligere. Det ønskes at kunne læse SRAM'ens data udelukkende ved at sætte den ønskede adresse. Dette gøres ved at forbinde signalet "ready"og signalet "playmem"hvilket foresager at SRAM-interfacet konstant vil påbegynde en læseoperation på den nuværende adresse, hver gang en operation er fuldført. SRAM-interfacet vil dermed køre i en konstant løkke på tre clockcucklusser igennem stadierne "idle->"read_1->"read_2"[13]. Når en SRAM'ens adresse bliver ændret vil adressens indhold stå på SRAM-interfacets output "data_s2f"tre clockcuclusser efter adressen sættes.

[11] Se afsnit **??** for dybere information.
[12] hvilket almindeligvis er tilfældet
[13] Se figur 1.6

Tilstandsmaskinen tager imod fire input fra sequenceren. Disse inputs bruges til at afgøre om trommerne skal afspilles. Hvis én eller flere af disse signaler er logisk høje, vil deres tilsvarende drumOn(X) sættes til logisk 1. Samtidig med dette bliver den nuværende trommeadresse sat til trommens startadresse (drumXCurrntAddr := drumXStartAddr). Dette medvirker at selv hvis en tromme bliver afspillet, vil det være muligt at genstarte afspilningen. Dette kunne for eksempel være tilfældet med tromning i realtid hvis der trommes hurtigt nok, eller hvis der trommes i realtid på "næsten"samme tid som et input fra stepsequenceren.

Tilstandsmaskinen er også drevet af clockstimuli, som er tilpas langsom i forhold til tilstandsmaskinen der styre SRAM'ene. Da denne er op til tre clockcyklusser om en operation, skifter qwer stadie hver fjerde clockcyklus, således at det med sikkerhed vides at not qwer er færdig med sine operationer. Hvis ikke dette overholdes kunne qwer risikere at få forkert data til de fire trommer da den rigtige byte der skal hentes ikke står på SRAM-interfacets output før der er gået tre clockcyclusser. Clockken der driver datahentnings-modulet kaldes stadieCLK, og den har en periodetid på $\frac{4}{32\,\mathrm{MHz}} = 125\,\mathrm{ns}$

Denne tilstandsmaskine og I²S-modulet sikkerhedsmæssigt koblet sammen med to signaler, hvilket beskrives dybere en underafsnit 1.1.7. De to signaler kaldes ready4Word og wordReady, hvor ready4Word er drevet af I²S-modulet, og wordReady er drevet af datahenter-modulet. Med mindre at ready4Word er 1 kan modulet ikke finde ny data, hvilket tjekkes hver gang der forekommer en stigende kant på stadieCLK.

Når der hentes data fra SRAM'ene er det nødvendigt at hente data fra to adresser hver gang der skal bruges et sample fra én enkelt tromme. Med dette skal der i princippet læses fra 8 RAM-adresser hvis der skal bruges samples til alle fire tromme, men da der skal samme operationer til et læse én byte, som der skal til for at læse otte, er tilstandsmaskinen delt op i fire hovedoperationer. I kronologisk rækkefølge er disse: lave tilstandsdiagram

1. At tjekke om den respektive tromme er tændt. Hvis trommen er tændt sættes RAM-adresse lig med den nuværende trommeadresse. Hvis ikke trommen er tændt fyldes nuværende trommesamplebuffer med 0'er, og der hoppes ned for at tjekke næste tromme.
2. Fylde den nuværende trommesamplebuffer med den mest betydningsfyle byte med data fra RAM'ene og tælle den respektive drumXCurrntAddr én op.
3. Sætte RAM-adressen til den nuværende trommeadresse for den nuværende tromme.
4. Fylde den mindst betydningsfylde byte med data fra RAM'ene, og tælle den nuværende trommeadresse op. Derudover tjekkes der om trommens slutadresse er nået. Hvis dette er tilfældet sættes drumOn(X) til 0.

Efter disse stadier er gennemført for hver af de fire trommer, sættes signalet, wordReady, til 1 hvorved I²S-modulet ved at der er data klar. noget halløj med at tælle adresser op og fylde buffer. Rasmus forklare om RAMmaskinen. skal lige høre hvad du tænker for jeg synes ikke det passer så godt her. V.H. Rasmus

Kravet for denne tilstandsmaskine er at den skal kunne levere 4 trommesamples inden I²S-modulet løber tør for data. Da der vælges at afspille samme sample på højre og venstre kanal, kan den maksimale tidsgrænse udregnes som:

$$T_{samplerate} = \frac{1}{50\,\mathrm{kHz}} = 200\,\mu\mathrm{s} \tag{1.1.2}$$

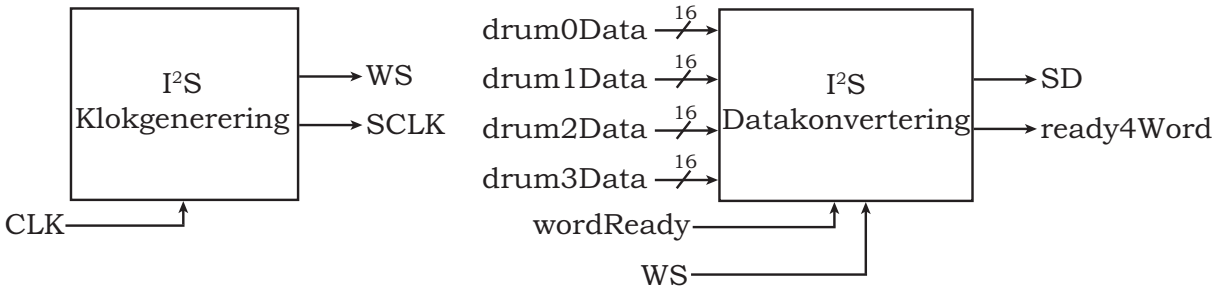
At skulle finde data til én tromme tager fire stadier, hvor ét stadie gennemløbes hvert stadieCLK-cyklus. Da der i princippet skal hentes data til fire trommer svare dette til 16 stadier. Derudover benyttes der endnu et stadie til at signalere at alt dataen er klar. Den maksimale tid det vil tage en finde data udregnes til:

$$T_{qwer} = 17 \cdot 125\,\text{ns} = 2{,}125\,\mu\text{s} \tag{1.1.3}$$

Da T_{qwer} er en del mindre end $T_{samplerate}$ vurderes implementeringen ikke at være i tidsnød.

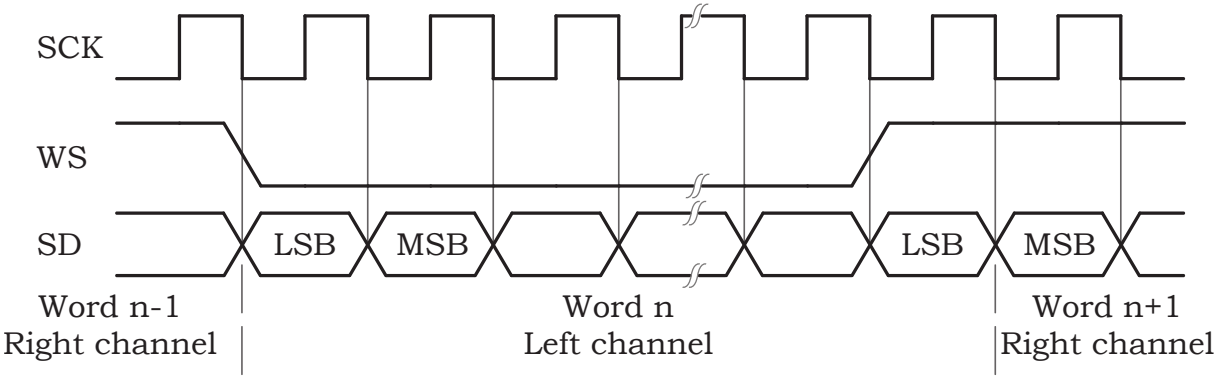
Design af I²S-modul

Selve I²S-modulet består af to undermoduler som kan ses på figur 1.13. Den ene af disse moduler har til opgave at denne de nødvendige kloksignaler for at kunne kommunikere i overensstemmelse med I²S-standarden, og den anden skal behandle data fra de fire trommesamples således at det bliver til ét sample.



Figur 1.13: Moduldiagram for I²S-modul med in- og outputsignaler

På figur 1.14 genses opbygningen af I²S-standarden.



Figur 1.14: Opbygning af I²S-standard

For at overholde standarden er det nødvendigt at fremstille de to kloksignaler, SCK(Serial Clock) og WS(Word Select).

På en periode af WS-signalet sendes der ét ord på begge kanaler. Dette medfører at WS skal have samme frekvens som trommelydene er samplet med for at trommelydene afspilles korrekt. Igennem en WS-periode skal der sendes et 16-bits ord på både venstre og højre kanal. hvilket giver SCK en frekvens på

$$f_{SCK} = 2 \cdot 16 \cdot f_{WS} = 2 \cdot 16 \cdot 44{,}1\text{kHz} = 1{,}41\ \text{MHz} \tag{1.1.4}$$

Tidsperioden for dette klok-signal vil derfor være:

$$T_{SCK} = \frac{1}{f_{SCK}} = \frac{1}{44,1\text{kHz}} = 708,62 \text{ ns} \tag{1.1.5}$$

Clock-signalet fra Papilioens oscilator har en frekvens på 32 MHz, hvilket giver den en tidsperiode på $\frac{1}{32\text{MHz}} = 31{,}25$ ns.

Da forholdet mellem disse to clock-signaler giver $\frac{708{,}62 \text{ ns}}{31{,}25\text{ns}} = 22,67$ er det ikke muligt at danne den ønskede SCK ud fra et antal klok-cyklusser af klok-signalet fra Papilioen.

Derfor er der valgt at resample trommelydene med en samplingsfrekvens på 50 kHz. Dette giver SCK en frekvens på

$$f_{BCK} = 2 \cdot 16 \cdot 50\text{kHz} = 1{,}6\,\text{MHz} \tag{1.1.6}$$

hvilket giver en tidsperiode på

$$T_{BCK} = \frac{1}{1{,}6\,\text{MHz}} = 625\text{ns} \tag{1.1.7}$$

Det nye forhold bliver derved

$$\frac{625 \text{ ns}}{31,25 \text{ ns}} = 20 \tag{1.1.8}$$

Faktoren mellem disse to klok-signaler er 20, hvilket betyder at den eksisterende occilator skal geares ned med en faktor 20. Hver gang der optræder en stigende kant på occilatorsignalet lægges der én til en variable. Når denne variable opnår en predefineret værdi nulstilles den og SCK-signalet inverteres, hvilket medfører at den predefineret værdi skal være 10, fordi I²S-standarden læser bits på stigende signalkanter så SCK-signalet skal inverteres to gange. Samtidig benyttes en anden variable til at holde styr på hvornår der skal skiftes kanal. Denne tælles op til 16 gange SCK-signalet, hvorefter WS-signalet inverteres og variablen nulstilles.

Som det ses på figur 1.14 er den første bit i hvert ord den mindst betydningsfulde bit fra det foregående ord. Da dataen bliver leveret parallelt til modulet benyttes der en buffer til midlertidig opbevaring af ordet der afspilles. Inden denne buffer opdateres til et nyt ord gemmes den mindst betydningsfylde bit i en variable så den er klar til skift i WS og derved det nye ord.

I kodeeksemple 1.12 ses strukturen for opdatering af afspilningsdata, for letteregøre forståelsen. Med timingen på plads er der valgt kun at hente ny data når der afspilles på højre lydkanel (WS = 1). Selvom der kun hentes data på højre kanal giver dette en øvre tidsgrænse for datahentning på det halve af $T_{\text{samplerate}}$, hvilket er 100 µs. Dette er markant større end de 2,125 µs det tager I²S-datahenter-modulet at hente data. Fordelen ved kun at hente data på den ene kanal, er at det vides med sikkerhed at det samme sample afspilles på begge kanaler. Hvis hele WS-periode blev brugt til at hente data, vil der med den anvendte implementering være mulighed for at overskrive den fåregående LSB. Der kunne benyttes endnu en buffer til opbevaring af sampledata hvis dette ville undgås, men dette vil forsinke afspilningen af lyd med en WS-periode, og da processen ikke er i tidsnød er denne implementering valgt.

Ud over timingen er der implementeret sikkerhedssignaler der skal sørge for at der kun modtages ny data på de rigtige tidspunkter, således at et sample ikke overskrives under afspilning. Denne sikkerhed består af to signaler der hedder ready4Word og wordReady. Når I2S-modulet er klar til at få ny data, sættes ready4Word til 1, hvilket den gør på hver faldende signalkant på WS. WordReady fungerer som en indikator for hvornår det næste ord til D/A-konverteren skal klargøres. Med mindre at wordReady er 1 kan intet data overskrives. Når wordReady er 1 bliver ready4Word sat til 0, hvilket betyder at der ikke bedes om mere data inden næste overgang til venstre lydkanal (WS = 1).

Kodeeksempel 1.12: Klargørelse af I2S-data lige inden afspilning

```
1  i2sModul: process (wordSelect,wordReady)
2  begin
3          if rising_edge(wordSelect) then
4                  ready4Word     <= '1';
5          end if;
6
7          if falling_edge(wordSelect) then
8                  dataBuffer     <= STD_LOGIC_VECTOR(to_signed(sampleSum,16));
9          end if;
10
11         if wordSelect = '1' then
12                 if wordReady = '1' then
13                         ready4Word <= '0';
14                         sampleSum := to_integer(signed(drum0)) + to_integer(signed(drum1)) +
                               to_integer(signed(drum2)) + to_integer(signed(drum3));
15
16                         if sampleSum > 32767 then    --A 16 bit 2C's range is -32768 to 32767.
17                                 sampleSum := 32767;
18                         elsif sampleSum < -32768 then
19                                 sampleSum := -32768;
20                         end if;
21
22                         previousLSB <= dataBuffer(0);
23                 end if;
24         end if;
25 end process i2sModul;
```

Som sidste led i keden inden D/A-konverteren benyttes der en 16:1 MUX til at konvertere den parallelle data til seriel data. Denne MUX er drevet af lokationen i WS hvilket bliver dannet under klokfremstillingen af denne.

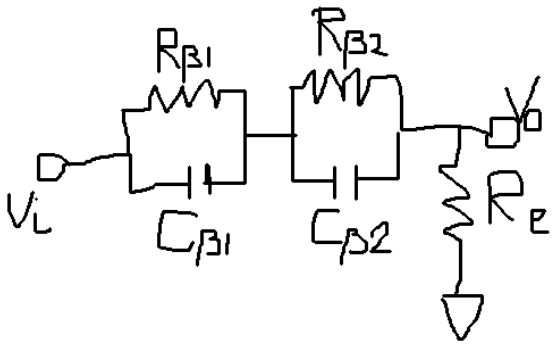
1.1.8 Opsumering af lydmodulet

Det har været en lang og trætsommelig rejse gennem designet af lydmodulet, men nu af rejsen ovre og det er tid til at fortælle hvad vi har lært.

1.2 Design af sequencer

Som nævnt tidligere, er det nødvendigt for den endelige trommemaskine, at kunne holde en rytme og afspille lyde på et bestemt tidspunkt. For at styrer dette er der brug for nogle input og output. Med hensyn til input, skal dette delmodul styres af en række knapper, med forskellige funktioner. For det første skal det være muligt at vælge mellem forskellige sekvenser, som man gerne vil afspille. Derudover skal det være muligt at vælge mellem forskellige trommelyde, da det ellers vil være en forholdsvis kedelig trommemaskine. Yderligere skal der være knapper til at styre hastigheden, som en sekvens afspilles med. Til sidst skal der være knapper for de enkelte dele af sekvensen, så det kan bestemmes hvornår trommelyde skal afspilles.

Måske det ville give mere mening at skrive om knapper i kontrolmodul-afsnittet?



Figur 1.15: Placeholder for blokdiagram

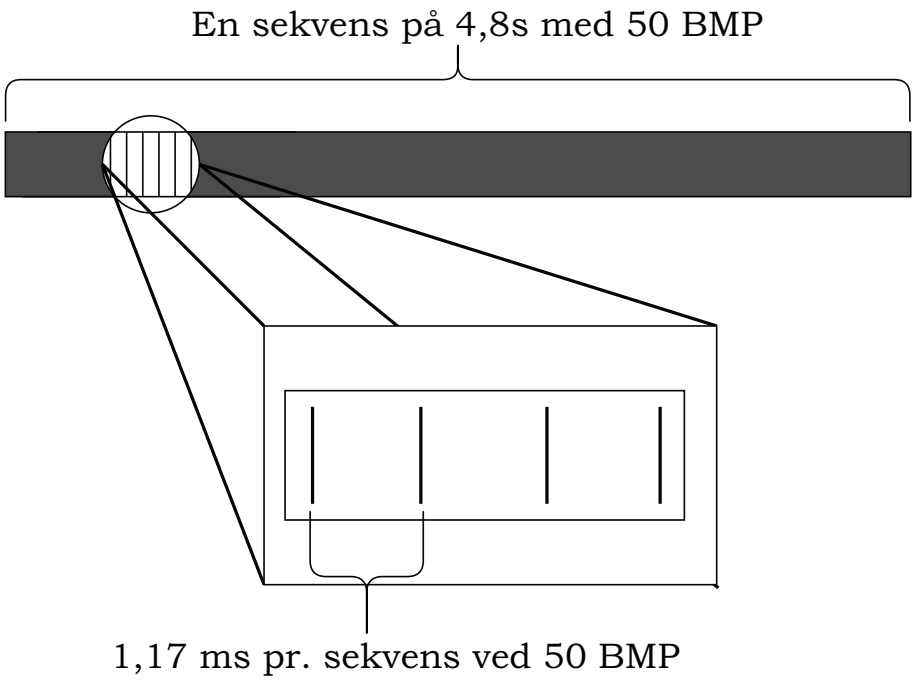
1.2.1 Kontrol af hastighed

Da der sidder en intern clock i Papilio'en bruges denne til at holde tempoet. Denne clock har en frekvens på 32 MHz og skal derfor sænkes i hastighed for at holde rytmen for trommemaskinen. Til at starte med tages der udgangspunkt i 100 BPM. Da det ønskes at sequenceren kan opfange lyd i realtid er der blevet undersøgt hvor præcist et signal skal værre, for at det passer med præcisionen for en trommeslager. Det er her blevet besluttet at 1 ms er passende præcist for dette. Det er derfor nødvendigt at beregne hvor lang tid det tager at afspille en sekvens. Det kan gøres ved at tage den reciprokke at BPM'en og gange med fire med et minut, da der er fire beats på en takt.

$$\frac{60 \cdot 4}{100} = 2{,}4\,\mathrm{s} \tag{1.2.1}$$

Altså vil det tage sequenceren 2,4 sekunder, at gennemgå en hel sekvens, med en BMP på 100. Da det ønskes at trommemaskinen skal kunne gå ned til en hastighed på omkring 50 BPM udføres de følgende beregninger på 4,8 s. For at gøre det nemmere at programmere i VHDL, er der taget udgangspunkt i den nærmeste to'er potens, som i dette tilfælde er 4096. Derved kan præcisionen for et signal men 50 BPM beregnes:

$$\frac{4,8\ \mathrm{s}}{4096} = 1,17\ \mathrm{ms} \tag{1.2.2}$$



Figur 1.16: if (ugly and shit) remake; fin tegning, hvis bare bmp -> bpm

Idé til hvordan man kan skitsere sekvensdelene

Det vil sige at når hastigheden for sequenceren er 50 BPM vil der være 1,17 ms mellem sekvensens dele, og dette antages at være tilstrækkeligt. For 60 BPM er dette krav opfyldt da der er 0,97 ms mellem de enkelte dele i sekvensen, hvilket medføre at højere hastigheder også vil opfylde kravet hertil. Da der bruges den samme opløsning for højere hastigheder, vil dette blot øge præcisionen.

For et få en clock til at passe med de ønskede BPM, laves en tæller, som skifter et signal når der nås til at bestemt tal. Dette signal bruges så som clock i en anden process. Ud fra de tidligere oplysninger kan det nu beregnes hvad der skal tælles til, ved en hastighed på 100 BPM. Dette medføre naturligvis at der skal tælles til det dobbelte for 50 BPM og det halve for 200 BPM. For at beregne hvad der skal tælles til ved 100 BPM, tages følgende formel i brug:

$$\frac{32\ \text{MHz} \cdot 2,4\ \text{s}}{4096} = 18750 \tag{1.2.3}$$

Kodeeksempel 1.13: Clocken der holder rytmen

```
1 if rising_edge(clk) then
2         count := count + 1;
3         -- See if count needs to be reset
4         if count >= speedSignal then
5                 count := 0;
6         end if;
7 end if;
```

Som nævnt tidligere skal dette tal ændres alt efter hvilken hastighed man vil have at trommemaskinen har. For at ændre på dette sammenlignes counteren for klokken med et separat signal. Dette signal kan så øges og sænkes ved brug af to knapper, som derved giver kontrol over hastigheden.

For at opsummere strukturen for sequenceren tages der først udgangspunkt i den interne clock på 32 MHz, som tæller til 18750 hvorefter der stiftes et signal. Dette signal bruges som clock'en for en process, der skal hold styr på realtid, hvis man gerne vil holde rytmen selv. Processen der holder styr på realtid har igen en tæller, til at holde styr på hvor langt i processen man er. Når denne tæller når 4096 vil der være gennemgået en fuld sekvens og tælleren nulstilles. Hvis der ikke ønskes at bruge realtid kan tælleren for realtid stadig tages i brug, men der ses kun på de fire MSB, da disse vil være en 16. del af sekvensen.

1.2.2 Datastruktur

Sequenceren har også til opgave at holde styr på de enkelte sekvenser for de forskellige trommelyde. For at holde styr på dette skal der brugen en række input fra kontrolmodulet. For det første skal der bruges et input der fortæller hvilken sekvens og hvilken tromme der er valgt. Derudover skal der bruges input fra de knapper, som holder styr på realtid og de knapper, som holder styr på trommelydene ved sekstendedele af en sekvens. For at tage højde for alle disse input er der brug for en datastruktur som set i eksemplet herunder:

Kodeeksempel 1.14: Data struktur

```
1 TYPE seqData is array (4095 downto 0) of STD_LOGIC;
2 TYPE drum is array (3 downto 0) of seqData;
3 TYPE seq is array (3 downto 0) of drum;
4 shared variable data : seq;
5 -- Setting a position in data as high
6 data(1)(2)(1024) := '1';
```

Hvis der spilles i realtid skal der bruges et input, der fortæller hvornår, der bliver trykket på en trommeknap, og et tidspunkt hvor der bliver trykket på knappen på. Dette styres med den tidligere nævnte clock. Da clock'en er designet til at styre en counter, som går op til 4096, bruges denne counter til at bestemme hvilken plads i dataarrayet det skal tilgås. Da der bruges fire knapper, en for hver tromme, styre de tilhørende signaler hvilken tromme man tilgår. Og ligesom tidligere vælges sekvensen med sit eget signal.

Figur 1.17: if (ugly and shit) remake;

Idé til hvordan man kan skitsere data array'et, men jeg er ikke sikker på om der er fire tromme for hver sekvens, eller om sekvenserne "låner"trommerne af hinanden?

1.3 Brugerinterface

H3ll0 guy5 4nd w3lc0m3 t0 u53r int3rf4c3 d3scripti0n.txt - 4k4 intr0dukti0n

Brugerinterfacet bliver delt op i x: step-sequencer indikator, step-sequencer knapper, takt indikator, tromme knapper, sequence behandling, mode select og BMP kontrol.

Step-sequencer indikator er en række af 16 LED'er, som lyser op når man trykker på tilsvarende step-sequencer knap. Disse LED'er har til formål at vise brugeren, når man har valgt at afspille en sample, på en specifik takt.

Step-sequencer knapper består af en række af 16 tryk-knapper, som sammen med step-sequencer LED'erne, bruges til at vælge og se hvornår man vælger at spille en sample på et specifikt tidspunkt i de 16 dele af en takt.

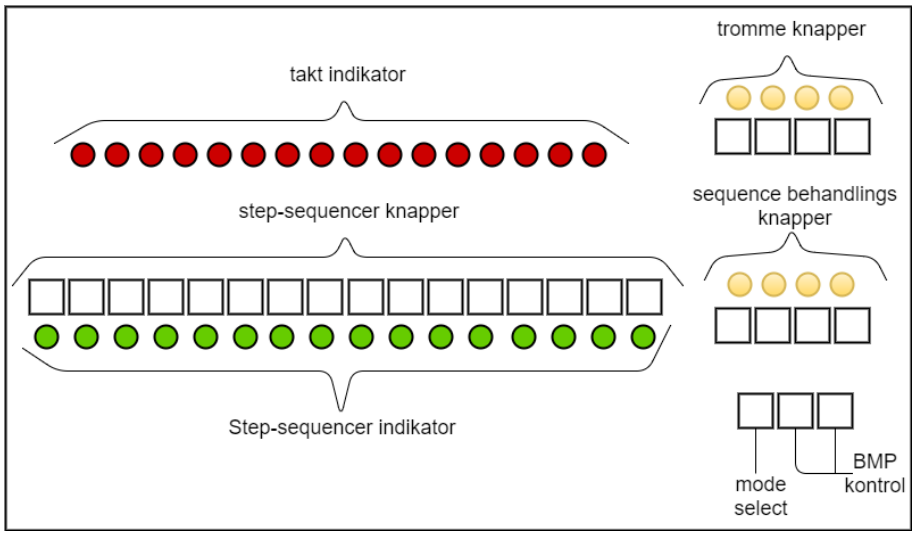
Takt-indikator består af 16 LED'er som tænder og slukker en efter en, som viser i hvilken af de 16 dele af en takt man er på. Altså vil lyd eksempelvis spille når der er lys i både den specifikke takt indikator LED samt den tilsvarende step-sequencer LED.

Tromme-knapper er fire tryk-knapper, som skal bruges at fortælle sequenceren hvilken tromme man gerne vil opdatere signalet for. Hvis der vælges en ny tromme skal signalet opdateres, så det valgte signal kan ses og ændres.

Sequence behandling består af fire tryk-knapper, som skal gøre det muligt at skifte mellem forskellige sekvenser. På samme måde som tromme-knapperne skal der kunne vælges mellem forskellige sekvenser, så der muligt at opdatere disse signaler.

Slet-knaptidligere mode knap har til formål at slette dataen for en bestemt sekvens og tromme.

BMP kontrol består af to tryk-knapper, som kan skrue op eller ned for BMP i et interval mellem 50 og 200.



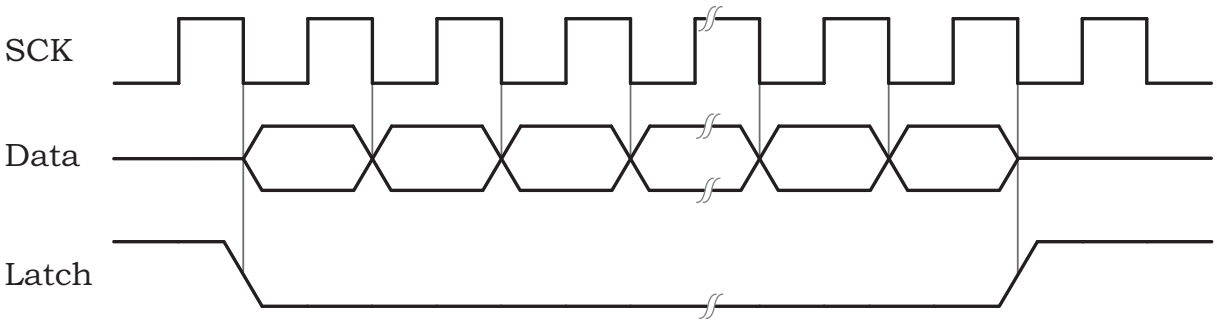
Figur 1.18: Umiddelbar billede af brugerinterfacet - high ress pl0x

1.3.1 ved ikke hvad dette afsnit skal hedde men det handler om teknikken bag bit-boardet

De mange knapper og LED'er vil kræve mange I/O, hvis de skulle forbindes direkte til FPGA'en. For at spare på I/O's bruges der shift-register. Med disse er det muligt at sende sekventielt input til shift-registerne, for derefter at skrive parallelt ud til hvor mange ben den nu engang har. Ligeledes gør shift-registerne også det muligt at læse parallelt fra mange ben, som så kan sendes sekventielt til FPGA'en. I dette projekt bruges der 8-bit shift register. Der bruges tre inputs til at styre dem, en clk, en parallel-load "latch"og et data ben.

Der skrives til og læses fra shift-registerne på samme måde. Når latch benet er højt, vil shift-registerne huske på deres nuværende tilstand. Når lachen sættes til lav, vil det derefter være muligt at læse eller skrive til dem igen.

Shift-registerne til LED'erne vil, når latch benet er lavt, have mulighed for at styre om LED'erne skal være tændte eller ej. Når latch benet er højt, vil shift-registerne sætte de forbundne LED'er høj eller lav, alt efter hvad der er skrevet forinden. Denne telstand vil blive husket til lachen ændres igen. Det data der bestemmer hvorvidt en LED skal være tændt eller ej, sendes til databenet. Data bliver så skrevet til shift-registerne, under overgangen fra lav til høj på clk benet. Samme proces foregår når der skal læses, men her skal databenet bruges som input til FPGA'en.



Figur 1.19: Umiddelbar billede af signal til shift-registerne - if (ugly and shit) remake;

Som tidligere nævnt består brugerinterfacet af knapper og LED'er. Step-sequence knapperne og alle LED'er er forbundet vha. shift-register. Til takt-indikatoren bruges der to 8-bits 74HC595 shift-registre i daisy-chain. Dette betyder at man sender et signal til det første shift-register i rækken og derefter bliver de overflødige bit sendt videre til den næste. Her bruges der samme clk og lach for hele daisy-chain'en. De resterende LED'er til indikation af sequence behandling og tromme knapper, består ligeledes af tre shift-register i daisy-chain, dermed skal der bruges hhv. 16-bit eller 24-bit når der skrives til dem. De 16 sequence knapper bruger to 8-bits HEF4021B shift-register, på samme måde som ved LED'erne, hvor der læses fra dem i stedet.

De resterende knapper (tromme, sequence knapper, delete og BMP kontrol) er forbundet direkte til FPGA'ens I/O, da der var overvejelser omkring deres funktionalitet. Dette medførte at de eventuelt skulle være brugt til tidsfølsomme processor og derfor ikke bør styres via shift-registre, da disse tager lidt længere tid at opdatere. Det har senere vist sig, at kommunikationen med shift-registerene er tilstrækkelig hurtig. Men da der er I/O nok, er det valgt ikke, at implementere yderligere shift-registere.

1.4 Kontrol Modul

Title might change For at kontrollere hvilke sekvenser og trommer der skal spille på de forskellige tidspunkter skal der holdes styr på diverse input fra knapper. Disse input kommer, som tidligere beskrevet, fra shift-registre eller er koblet direkte til FPGA'ens I/O's.

1.5 Brugerinterface

H3ll0 guy5 4nd w3lc0m3 t0 u53r int3rf4c3 d3scripti0n.txt - 4k4 intr0dukti0n

Brugerinterfacet bliver delt op i x: step-sequencer indikator, step-sequencer knapper, takt indikator, tromme knapper, sequence behandling, mode select og BMP kontrol.

Step-sequencer indikator er en række af 16 LED'er, som lyser op når man trykker på tilsvarende step-sequencer knap. Disse LED'er har til formål at vise brugeren, når man har valgt at afspille en sample, på en specifik takt.

Step-sequencer knapper består af en række af 16 tryk-knapper, som sammen med step-sequencer LED'erne, bruges til at vælge og se hvornår man vælger at spille en sample på et specifikt tidspunkt i de 16 dele af en takt.

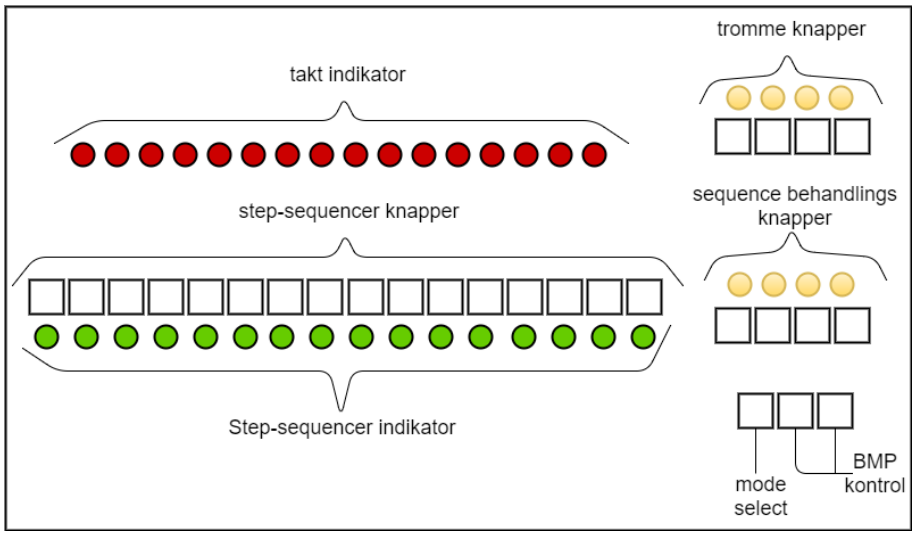
Takt-indikator består af 16 LED'er som tænder og slukker en efter en, som viser i hvilken af de 16 dele af en takt man er på. Altså vil lyd eksempelvis spille når der er lys i både den specifikke takt indikator LED samt den tilsvarende step-sequencer LED.

Tromme-knapper er fire tryk-knapper, som skal bruges at fortælle sequenceren hvilken tromme man gerne vil opdatere signalet for. Hvis der vælges en ny tromme skal signalet opdateres, så det valgte signal kan ses og ændres.

Sequence behandling består af fire tryk-knapper, som skal gøre det muligt at skifte mellem forskellige sekvenser. På samme måde som tromme-knapperne skal der kunne vælges mellem forskellige sekvenser, så der muligt at opdatere disse signaler.

Slet-knaptidligere mode knap har til formål at slette dataen for en bestemt sekvens og tromme.

BMP kontrol består af to tryk-knapper, som kan skrue op eller ned for BMP i et interval mellem 50 og 200.



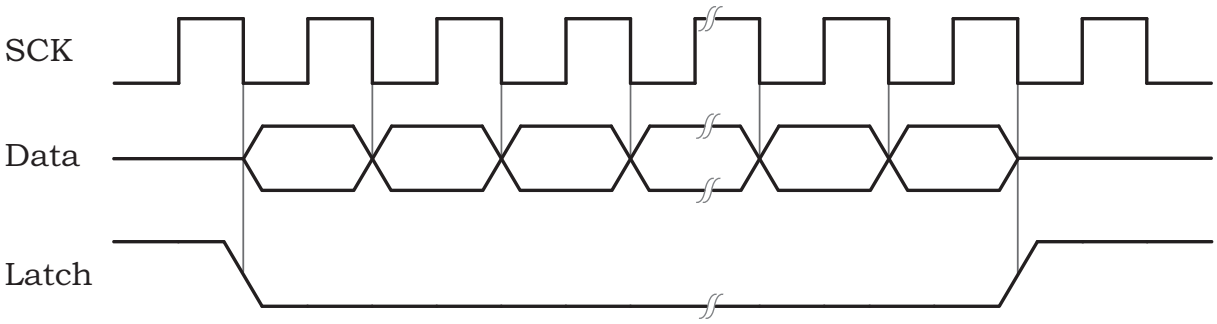
Figur 1.20: Umiddelbar billede af brugerinterfacet - high ress pl0x

1.5.1 ved ikke hvad dette afsnit skal hedde men det handler om teknikken bag bit-boardet

De mange knapper og LED'er vil kræve mange I/O, hvis de skulle forbindes direkte til FPGA'en. For at spare på I/O's bruges der shift-register. Med disse er det muligt at sende sekventielt input til shift-registerne, for derefter at skrive parallelt ud til hvor mange ben den nu engang har. Ligeledes gør shift-registerne også det muligt at læse parallelt fra mange ben, som så kan sendes sekventielt til FPGA'en. I dette projekt bruges der 8-bit shift register. Der bruges tre inputs til at styre dem, en clk, en parallel-load "latch"og et data ben.

Der skrives til og læses fra shift-registerne på samme måde. Når latch benet er højt, vil shift-registerne huske på deres nuværende tilstand. Når lachen sættes til lav, vil det derefter være muligt at læse eller skrive til dem igen.

Shift-registerne til LED'erne vil, når latch benet er lavt, have mulighed for at styre om LED'erne skal være tændte eller ej. Når latch benet er højt, vil shift-registerne sætte de forbundne LED'er høj eller lav, alt efter hvad der er skrevet forinden. Denne telstand vil blive husket til lachen ændres igen. Det data der bestemmer hvorvidt en LED skal være tændt eller ej, sendes til databenet. Data bliver så skrevet til shift-registerne, under overgangen fra lav til høj på clk benet. Samme proces foregår når der skal læses, men her skal databenet bruges som input til FPGA'en.



Figur 1.21: Umiddelbar billede af signal til shift-registerne - if (ugly and shit) remake;

Som tidligere nævnt består brugerinterfacet af knapper og LED'er. Step-sequence knapperne og alle LED'er er forbundet vha. shift-register. Til takt-indikatoren bruges der to 8-bits 74HC595 shift-registre i daisy-chain. Dette betyder at man sender et signal til det første shift-register i rækken og derefter bliver de overflødige bit sendt videre til den næste. Her bruges der samme clk og lach for hele daisy-chain'en. De resterende LED'er til indikation af sequence behandling og tromme knapper, består ligeledes af tre shift-register i daisy-chain, dermed skal der bruges hhv. 16-bit eller 24-bit når der skrives til dem. De 16 sequence knapper bruger to 8-bits HEF4021B shift-register, på samme måde som ved LED'erne, hvor der læses fra dem i stedet.

De resterende knapper (tromme, sequence knapper, delete og BMP kontrol) er forbundet direkte til FPGA'ens I/O, da der var overvejelser omkring deres funktionalitet. Dette medførte at de eventuelt skulle være brugt til tidsfølsomme processor og derfor ikke bør styres via shift-registre, da disse tager lidt længere tid at opdatere. Det har senere vist sig, at kommunikationen med shift-registerene er tilstrækkelig hurtig. Men da der er I/O nok, er det valgt ikke, at implementere yderligere shift-registere.

Ordliste

BPM Beats per minute. generelt andvent måleenhed for musiktempo. Antallet af taktslag per minut. Eksemplelvis antallet af fjerdedele på et minut i taktarten $\frac{4}{4}$.. *se* takt

D/A-konverter Elektronisk enhed der har til formål at omdanne et digitalt signal til et analogt.. 19

FPGA Field Programable Logic Array. Integreret kredsløb af konfigurærbar logic, bygget af logic-blokke eller "celler" . 2

I²S Inter-IC Sound. Seriel bus til interfacing mellem digitalt lydudstyr. . 20

KCPSM6 Processer til en picoBlaze lavet specifikt til Spartan-6 og lign. FPGA'er. *se* picoBlaze, 9

picoBlaze 8-bits soft mikrocontroller designet til FPGA'er fra Xilinx. *se* FPGA, i

RAM Random Access Memory. hukommelse der både kan læses fra og skrives til. 2

ROM Read only memory. Hukommelse der kun kan læses fra.. *se* RAM, 22

SRAM Static Random Access Memory. Statisk RAM. . *se* RAM, 3

TDA1541A 16 bits stereo D/A-konverter designet til brug i Hi-Fi-aparater. Dette er D/A-konverteren anvendt i dette projekt.. *se* D/A-konverter, 19

WAV / **Wave** Waveform Audio File Format. Filformat til lyd, der foruden en header består af ukomprimeret data.. 20

Bibliografi

[1] Ken Chapman. *PicoBlaze for Spartan-6, Virtex-6, 7-series, Zynq and UltraScale Devices (KCPSM6)*. Xilinx. 2014.

[2] Pong P. Chu. *FPGA programming by VHDL examples*. Wiley, 2008.

[3] ISSI. *IS61WV2048ALL IS61/64WV2048BLL 2M X 8 HIGH-SPEED CMOS STATIC RAM*. Set d. 1/4/17. 2014.

[4] Philips. *TDA1541A datasheet*. Sidst set: 14/04/2017. 1991. URL: `http://pdf.datasheetcatalog.com/datasheet/philips/TDA1541A.pdf`.

[5] wikipedia. *A typical hardware setup using two shift registers to form an inter-chip circular buffer*. Sidst set: 26/4/2017. URL: `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#/media/File:SPI_8-bit_circular_transfer.svg`.

Test af D/A-konverter og I2S 2

2.1 Formål

Formålet med dette forsøg er at undersøge hvorvidt Papilio DUOen kan kommunikere med den valgt D/A-konverter via I2S, og om D/A-konverteren kan repræsentere alle værdier i et 16 bits ord som en spænding.

2.2 Udstyr

På tabel 2.1 ses en tabel over det anvendte forsøgsudstyr.

Instrument	Model	AAU-nummer
FPGA	Papilio-duo 2MB	
Logik analysator	Digilent Analog Discovery 2	2179-04
D/A-konverter	TD1541A	
Strømforsygning	HAMEG HM7042	B1-101-N-7 B1-101-O-6

Tabel 2.1: Tabel over anvendt forsøgsudstyr.

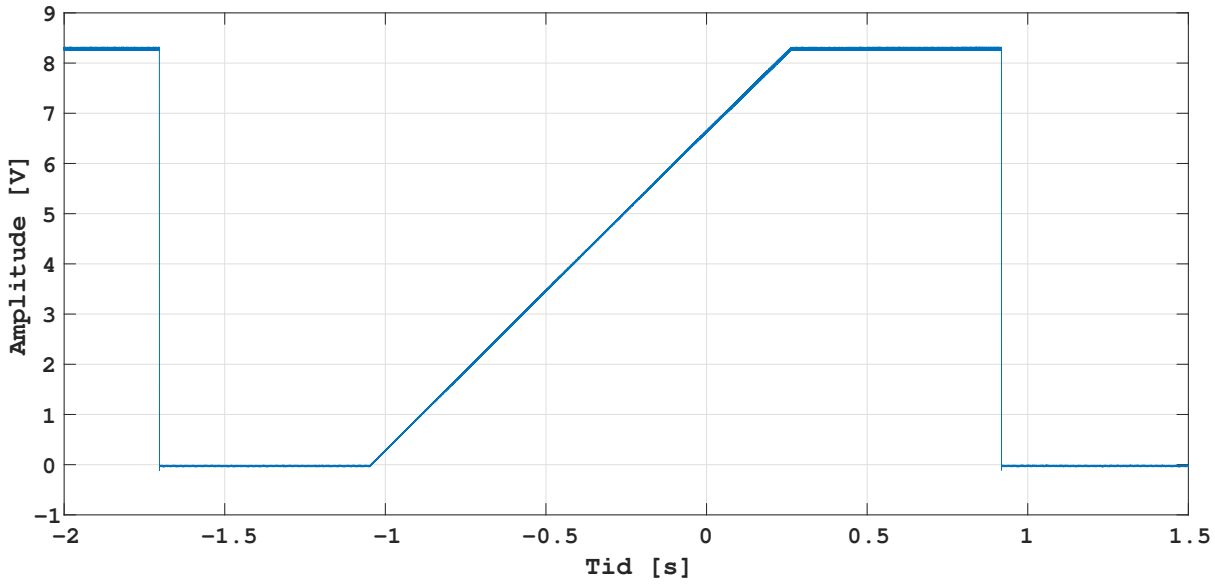
2.3 Metode

Der er kodet et VHDL-modul, som skal kommunikere med D/A-konverteren via I2S. Dataet der sendes til D/A-konverteren, skal falde inden for grænserne for et 16 bits ord i 2's kompliment. I decimaltal svarer dette til, at den sendte serielle data skal gå fra -32768 til 32767. Der er i kravspecifikation vedtaget, at outputtet skal gå i klipning hvis værdien overskriver disse grænser. For at teste hvorvidt dette overholdes, tælles en variable op fra -33000 til 33000 hver gang der skiftes fra venstre til højre lydkanal. Denne øvre og nedre grænse er valgt, da den ligger uden for et 16 bits ord i 2's kompliment, hvilket vil medfører over- eller underflow. Hver gang der er blevet sendt en specifik værdi på hver kanal, vil den næste værdi være én størrer. Når variablen når sin øvre grænse på 30000, sættes den igen til -30000.

D/A-konverteren opkobles jævnfor dens datablad[4], hvor dens input bliver danne af Papilioen. I databladet ses det at D/A-konverterens output skal direkte i en operationsforstærker med predefineret tilbagekobling. Forstærkningsgraden, og derved den maksimale udgangsspænding, er derfor ukendt. Der benyttes intet DC-ofset i operationsforstærkeren, så negative spændinger forventes ikke. Operationsforstærkeren medtages som en del af D/A-konverteren, og dette udgangssignal måles og gemmes af logikanalysatoren. Hvis outputdatavariablen overskider grænserne for et 16 bits ord forventes det at ouputtet klipper, og hvis variablen ligger inden for grænsen forvæntes en lineær stignen i D/A-konverterens output.

2.4 Resultater

På figur 2.1 ses forsøgets måledata. Der ses her at udgangsspændingen klipper ved 0 V og lige over 8 V. Det ses også at stignen i spændingen er lineær mellem den øvre og nedre grænse for klipning.



Figur 2.1: Måledata over spændingsniveauerne fra D/A-konverteren.

2.5 Konklusion

Det konkluderes at Papilioen kan kommunikere med D/A-konverteren via I2S, og at alle værdier for et 16 bit ord kan repræsenteres som en spænding. Hvis ikke de alle var repræsenteret, ville stigningen på figur 2.1 have små ujævnhedere hvor værdien ikke kunne repræsenteres. Det konkluderes også at klipning håndteres som forventet.