1 Projektbeskrivelse

1.1 Arkitektur

Efter selve projektet var blevet udtænkt, begyndte nedbrydningen af det således, at der kunne fordeles opgaver til de enkelte gruppemedlemmer.

Den oprindelige idé gik på, at der skulle sidde et komponent i FPGA'en, skrevet i VHDL, som ville tage imod en frekvensværdi og derefter generere de enkelte værdier til ram-modulerne. Dette blev omsat således, at værdierne blev beregnet fra C-kode og derefter sendt enkeltvis til ram-modulerne.

1.1.1 VHDL-kode

Hernæst blev ram-modulernes interface defineret, således andre komponenter kunne udtænkes. Dette indvolder de to komponenter, *TransportProtocol* og *PlaySound*, der hhv. modtager og gemmer dataen fra C-koden og afspiller hvad der ligger i ram-modulerne.

- *TransportProtocol*: skulle modtage data igennem *Avalon Memory-Mapped*-interfacet, som kan omsættes til funktioner til C-kode. Derudover skulle det kunne skrive data til hver af de to *RamAccess*-moduler, der hver i sær vil kunne indeholde én frekvens.

  Således ville *TransportProtokol* holdes styr på, hvor mange samples hver frekvens benyttede, hvilket ram-modul der ikke blev afspillet fra og hvilken adresse i modulet dataen skulle gemmes på.

- *PlaySound*: Når data for en frekvens var gemt, skulle komponenten *PlaySound* have besked herom således, at den vil spille den nyeste data fra det korrekte ram-modul. Når der læses data, er adressen der skal læses fra ens for alle ram-modulerne, således at de alle sender hver sin data til *PlaySound*.

  For at forenkle *PlaySound*'s interface er der kun ét data-input og ikke et per ram-modul. Ved at sætte en multiplexer mellem komponenten og alle ram-modulerne, kan et chipselect afgøre hvilket input der ønskes.

  Hver data fra ram-modulerne bliver herefter sendt videre igennem *Avalon Streaming*-interfacet som `ast_source_data`.

1.1.2 C-kode

I forbindelse med designet af C koden har vi så vidt muligt forsøgt at opdele koden i funktioner og filer således det er nemt at overskue. Samtidig har vi beskrevet de forskellige funktioner således det er ligetil at gå til dem.

I vores projekt har vi følgende grupperinger:

- Main: I denne fil ligger main-funktionen der er med til at kalde alle de andre funktioner på baggrund af hvad brugeren gør. I funktionen er der en initieringsfase, hvor bl.a. hardware-komponenter sættes op. Herefter kører programmet i en while-loop som hele tiden tjekker op på enten tastetur eller knapper, behandler de inputs og sender data til displays og hukommelsesmodulet.

- RAM Sound: Gør det muligt at skrive ned til hukommelsesmodulet. Det er meningen at man bruger den ene funktion til at angive hvor mange samples der efterfølgende kommer og den anden til at sende samples en efter en.

- Keyboard: Har som formål at indhente og bringe information om de seneste tasteturtryk. I forbindelse med opdatering af seneste tastetryk gives et array hvori de seneste taster er og som skal gives med igen senere for at få det korrekte billede. I filen er der også en funktion som bruges til at konvertere de HEX-tal tasteturet giver om til ASCII-karakterer.

- Keys: Denne opdaterer simpelt bare et array der angiver de knapper, der pt. er trykket ned. Idet knapperne hovedsagligt er med som test er denne del forsøgt nogenlunde simpel.

- 7-seg: Gør det muligt at skrive et tal ud på de fire 4 7-segments displays på DE2-boardet. På displayet vil det blive vist som BCD-tal og ikke HEX-tal, som det ellers ville pr. standard.

- LCD: Her er det muligt at skrive ned til LCD-displayets to linjer. Når man skriver ned til en af linjerne bliver den først ryddet, således der kan skrives på en frisk.

1.2 Detaljeret beskrivelse

1.2.1 Bruger input

Som bruger har du to muligheder for at give input til systemet. Et PS/2-tastetur har vi valgt som det primære input, men har også implementeret således man kan give input fra knapperne og derfor nemmere teste systemet.

1.2.1.1 PS/2 Tastetur

Fra tasteturet kan man trykke på en knap og derved kunne spille en tone. PS/2 interfacet er som sådan lige til at gå til. Enheden (mus eller tastetur) sender en række hexidecimale værdier der indikerer hvad brugeren gør. Fx bliver der sendt '1C' når tasten 'A' trykkes ned og sekvensen 'F0,1C' bliver sendt når 'A' løftes igen. Således kan man hele tiden holde styr på, hvilke taster der er trykket ned (så længe man ved hvad de forskellige hex-værdier svarer til). Som det også antydes bliver 'F0' sendt afsted når en tast løftes; på den måde ved vi, at når 'F0' modtages er den næste værdi svarende til tasten der er løftet.

Selve "driverne"til PS/2-tasteturet kunne vi godt selv have skrevet, men det vurderede vi som værende ikke-relevant at lægge vægt på og derfor har vi valgt at bruge Alteras egen implementation af interfacet til PS/2-porten. På den måde er vi også blevet udfordret ved, at vi ikke kun bruger materiale udleveret i undervisningen. Selve modulet, PS2 Controller, tilføjes SOPC'en som alle de andre og relevante pins (clock og data) forbindes.

For at få data fra tasteturet bliver vi ved med at kalde en funktion indtil den indikerer at der ikke er mere i tasteturets buffer (ved at returnere -1):

Kodeudsnit 1: Opsamling af data

```
1 PS2_dev = alt_up_ps2_open_dev (PS2_PORT_NAME);
  while(alt_up_ps2_read_data_byte (PS2_dev, &PS2_data) == 0)
3 {
    //Use PS2_data for something.
5 }
```

Langt størstedelen af arbejdet omkring PS/2-tasteturet har, foruden det at forstå interfacet, været at implementere softwaren i C der gør, at vi kan bruge tasteturet på en fornuftig måde. Forbindelsen ned til tasteturet er lavet på baggrund af de eksempler Altera selv har lagt med Quartus. Udover det skulle der bl.a. konverteres fra "tastetur-hex"til "ASCII-hex"og da der ikke er nogenlunde logisk sammenhæng mellem de to har vi

valgt bare at implementere det vha. en switch-case.

Derudover har der været et designvalg ift. om vi ønsker udelukkende bare at bruge en buffer for tasteturtryk eller om vi også ønskede et interrupt. Sidstnævte har vi valgt fra, idet interrupts hurtigt kan give nogle problemer ift. hvordan ting eksekveres og vi har vurderet at vi hellere ville have en stabil funktionalitet frem for en hurtig.

Problemet ved at vælge buffer-metoden kan være at man glemmer at tømme den jævnligt, men så længe man er opmærksom på dette, burde der ikke ske noget.

Bemærk desuden at vi har valgt at afgrænse vores projekt til at vi kun understøtter tasterne 0-9 og A-G, da flere taster som sådan ikke ville give mere værdi, men bare ville tage længere tid at implementere.

1.2.1.2 Knapper

Af test-grunde har vi valgt at tre af knapperne også kan bruges som input. Vha. af switch 0 (SW0) bestemmes om der ønskes at bruges PS/2-tastetur eller de tre "test-knapper".

Knapperne kan bruges på mere eller mindre samme vis om tasteturet, men med mindre funktionalitet. På den måde undgås at man skal sætte tasteturet til hvis det ikke er det man ønsker at afprøve. De to input er helt adskilt, hvilket vil sige at keys ikke påvirker tasteturet når det benyttes og omvendt.

Hertzene der spilles, når knapperne er trykket ned bestemmes ud fra deres binære værdi. Fx giver et tryk på 1 og 2 samtidig 011 = 3. Dette ganges med 200 for at få en fornuftig frekvens og således spænder tasterne over 200-1400 Hz.

1.2.2 Visuel visning

1.2.2.1 7-segments display

7-segments displayet er klart den nemmeste måde at få nogenlunde brugbar information ud til brugeren. I vores tilfælde spilles der toner og vi har derfor skønnet det som oplagt at bruge display'et til at vise hvilken frekvens der spilles ved lige nu - fx 440 Hz.

I undervisningen er der udleveret to VHDL-filer der gør det muligt at lave en komponent til vores SOPC således, at vi kan skrive værdier dertil vha. MM-bussen. Ved at skrive dertil skrives der dog pr. standard de hexadecimale værdier ud og idet normale mennesker ikke vil kunne forstå det har vi valgt i vores C-kode at konvertere vores frekvenser til en værdi der, når den bliver sendt til 7-segments displayet, vises som en decimal frekvens.

1.2.2.2 LCD-skærm

LCD-skærmen er en mulighed til at få flere informationer ud til brugeren. Der er to linjer, som gør det muligt at skrive en "hel del". I vores tilfælde har vi valgt på første linje at indikere hvilket input der pt. benyttes. Hvis der bruges PS/2-tastetur står der "Keyboard press"og bruges knapperne står der "Keys pressed". På samme vis står der for de to input hvilke taster der pt. er trykke ned: For knapperne står angivet om 1, 2 og/eller 3 er trykket ned og for tasteturet står der hvilke taster som er trykket ned (i rækkefølgen de blev trykket ned).

For at skrive til skærmen skal der simpelt bare åbnes en strøm som skrives til. Dette gør vi således hele tiden at have den sidst nye info omkring hvilke taster der er trykket ned.

1.2.2.3 Afspilning af lyd

For at kunne afspille lyden sender hukommelsesmodulet de forskellige samples via ST-bussen til et komponent der sender dem videre ud på DAC'en. Dette komponent, ST2IIS, er lavet meget på baggrund af et komponent vi fik givet i undervisningen. Komponentet indeholder ikke længere nogen source (da det er hukommelses-modulet) og samtidig er skåret ind til benet således overflødig VHDL-kode så vidt muligt er fjernet. Forskellen derudover er at vi i stedet for ADCLRCK nu venter på at DACLRCK er klar til at modtage signaler (går fra lav til høj eller høj til lav). Samtidig har vi også implementeret modulet således at der nu spilles ud på begge kanaler (left/right). I praksis betyder det at der sendes en sample (bestående af 24 bits) til DAC'en både når DACLRCK er høj (højre kanal) og lav (venstre kanal). Det oprindelige modul havde en statemaskine til at holde styr på denne proces og dette har vi valgt at bibeholde for overskueligheden.

Vi har bevidst valgt ikke at lave en test-bench til ST2IIS for at begrænse projektet en smule. Samtidig er det originale modul, blevet testet i forbindelse med undervisningen, så til trods for vi har modificeret det, valgte vi at prioritere det at lave testbench'en til hukommelsesmodulet frem for til ST2IIS-modulet.

1.2.3 Beregning af sinuskurver

For at få en nogenlunde pæn tone valgte vi at implementere C-koden således den skulle sende samples svarende til en sinus-kurve ned til hukommelses-modulet. For at gøre det har vi valgt at inkludere math-biblioteket, hvori funktionen sin() lå. Vi fandt dog ret

hurtigt ud af at det ville tage utrolig lang tid at beregne alle samples i en sinuskurve og derfor blev i nødt til at optimere lidt på det. I stedet for at beregne en fuld svingning beregnede vi i stedet en kvart og derefter vendte og drejede den således vi kunne forme en fuld og derved rundt regnet spare en fjerdedel af tiden.

For at gøre det nemmere os selv valgte vi så dog også at skære lidt af den fulde svingning ved at finde et tal tæt derpå, som går op i fire, således svingningen er nem at dele op i fire lige store dele. Dette giver naturligvis ikke altid en lige så præcis hertz, men en tilnærmelsesvis god nok.

For at finde antallet af samples vi skulle bruge udførte vi følgende udregning. Bemærk hvorledes vi sørger for, at samlet antal af samples går op i fire.

Kodeudsnit 2: sample-udregning

```
1 quarter_samples = 48000/current_hertz/4;
  full_samples = quarter_samples*4;
```

Herefter kan vi beregne de fire vha. en række løkker og kendskabet til en sinuskurves udformning. Den første fjerdedel beregnes som beskrevet herunder. Implementeringen af de samlede fire dele kan ses i appendix **??**.

De enkelte samples beregnes således:

$$\left(1 + sin\left(\frac{sample\_iterator * 2 * \pi}{full\_samples}\right)\right) * HALF\_MAX\_CODEC\_SIZE$$

Hvor:

- sample_iterator tæller fra 0 til $quarter\_samples - 1$
- $HALF\_MAX\_CODEC\_SIZE = \frac{2^{24} - 1}{2}$

Et eksempel på en svinging er vist i appendix **??**, hvor en frekvens på 1 kHz vises over 48 samples. Af grafen kan det observeres at vores data følger den reelle sinuskurve nogenlunde, men ikke 100% idet vi har en mindre regnefejl i vores kode. Dette fandt vi desværre først ud af sent i projektet og har derfor valgt ikke at arbejde mere på det frem for at få færdiggjort vigtigere dele. Samtidig kan vi også se af appendix **??** at punkterne ved lavere frekvenser giver en mere jævn sinuskurve.

1.2.4 Begrænsning frekvens

Idet vi havde valgt at begrænse antallet af pladser i de to ram-blokke til 256 var antallet af hertz også naturligt begrænset. Idet CODEC'et kører ved en frekvens på 48 KHz er

den maksimale frekvens vi kan spille ved:

$$\frac{48.000 \text{ Hz}}{256} = 187{,}5 \text{ Hz}$$

Vores laveste frekvens var dog på 200 Hz fordi det passede bedst ind i vores system.

1.2.4.1 Forslag til optimering af udregninger

For at vi nemt kunne afprøve vores system med en lang række frekvenser og samtidig forsimple vores design valgte vi at bruge sinus-funktionen i C-koden og kun at implementere to RAM-blokke. At vi så senere fandt ud af at der ikke var plads til mere end to RAM-blokke (med 256 pladser i hver) er en helt anden sag.

Det er dog ingen hemmelighed at især sinus-funktionen er meget langsom når man tager i betragtning at man normalt ville forvente der ikke ville gå særlig meget tid mellem tastetryk og at en tone begynder at spille. Det eneste vi specielt har gjort for at optimere denne proces er at vi valgte at dele sinuskurven op i fire dele. Ønskede vi at optimere endnu mere kunne vi have gjort diverse andre ting.

Hovedsageligt ville det være en fordel at man ikke skulle beregne alle samples hele tiden. Fx kunne man starte ud med at beregne alle ønskede frekvenser og gemme dem i nogle arrays før selv hovedsekvensen går i gang. På samme måde kunne frekvenserne også være beregnet "manuelt"og ligge i en fil der loades ind.

En af de helt klart bedste optimeringer ville i stedet være hvis der havde været flere buffere hvori frekvenser så skrives ned til fra en start. Derefter skal man fra C-koden fortælle hvilken RAM-blok der skal spilles fra i stedet for at skulle skrive alt ned til dem hver eneste gang. I princippet kunne dette virke lidt som en look-up-table

1.3 Problematikker og deres løsninger

Herunder beskrives nogle af de problemstillinger, som der er mødt i udviklingsprocessen.

1.3.1 Forskellige clock-domæner

For at læse fra ram-modulerne og sende dette ud gennem ST-bussen, bruges der to forskellige clock-domæner. Dette er en udfordring, omend let at overkomme, så medfører den komplikationer. På udklip 3 ses, hvordan `data` overføres fra et clock-domæne til et andet.

Kodeudsnit 3: Clock domæne

```
signal data1 : std_logic_vector (dataSize-1 downto 0);
```

```
2 ...
  ramRead: process(ast_clk, reset_n)
4 begin
    ...
6   data1 <= data;
    ast_source_data <= data1(23 downto 0);
8   ...
```

1.3.2 SOPC-builder

SOPC-builderen, der skal generere generere alle komponenterne, brokkede sig med følgende besked:
Fitter requires 2103 LABs to implement the project, but the device contains only 2076 LABs. Problemet har muligvis dækket over, at vi bruger for mange logiske elementer i vores design, for da vi reducerede *RamAccess*'s `ramSize`-variabel fra 2048 til 256, forsvandt fejlen.

1.3.3 PS/2 Keyboard

I forbindelse med implementeringen af PS/2-tasteturet løb vi ind i et mindre problem idet vi fandt frem til at PS/2-tasteturet kun kan holde styr på et vis antal tastetryk ad gangen og at dette antal er lidt "uforudsigeligt". I starten troede vi at det var os der havde implementeret det forkert, men efter en kort google-søgning fandt vi frem til at man på et PS/2-tastetur maks. kan få input fra 5-6 taster ad gangen.

Som en sidebemærkning ift. PS/2-tasteturet så var det i starten også et problem overhovedet at få fat i et, idet de fleste er smidt ud. Vi prøvede i første omgang med et USB-tastetur og en USB-til-PS/2 konverter, men dette gav ingen data overhovedet. Til sidst fandt vi dog et tastetur i en kælder og kunne derved udføre vores projekt som ønsket.

mapning mellem filnavne og betegnelser i diagrammet

1.4 Analyse og optimering

Optimering af projektet har været i fokus, men er ikke vægtet i samme grad, som andre punkter. Dette viser sig i Quartus' analyse af projektet, hvor LE:Register-forholdet ikke er 1:1. Der er totalt 23.272 logiske elementer og 13.549 registre. Langt størstedelen af alle disse registre kommer fra de to *RamAccess*-moduler, som hver i sær har 5911 registre, hvorimod *TransferProtocol* har 59, mens *PlaySound* har 66.

Quartus vil desværre ikke vise hvad fmax er, hvilket er en skam, da dette fortæller hvor høj frekvens systemet maksimalt vil kunne køre med.

I bund og grund ville det heller ikke give alt for meget mening at gå ned og optimere meget i VHDL-koden idet vi har valgt at skrive store dele af vores projekt i C som langt fra er så hurtigt som hvis man gjorde det direkte i VHDL. Herunder er bl.a. også at vi vælger at bruge math-bibliotekets sinus-funktion hver gang en ny frekvens findes frem for at gemme det i en LUT. Se mere herom under 1.2.3

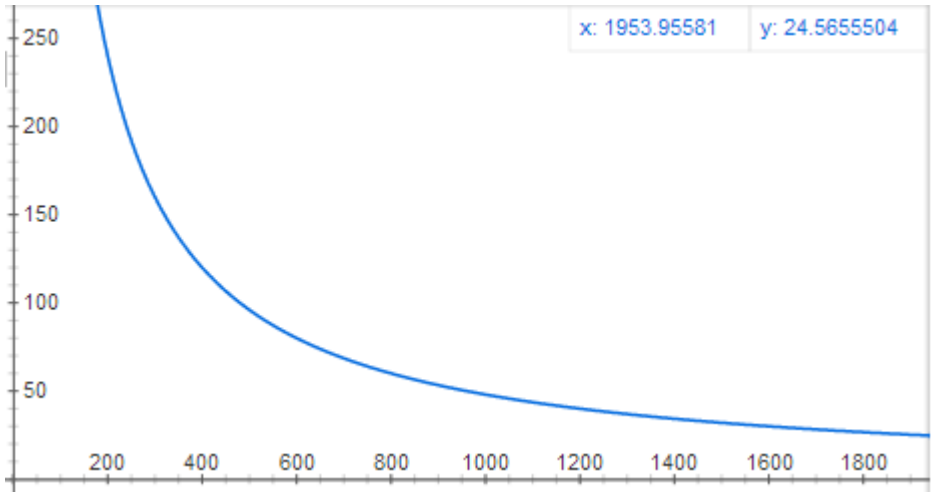
1.5 Test og simulering

For at sikre, at VHDL-koden virker som ønsket, er koden testet med programmet ModelSim 10.1b, hvor hvert komponent er unittestet samt en integreringstest for 2 af komponenterne.

Koden er testet således, at alle linjer er blevet afprøvet, så der ikke er en linje der vil sende underlig data ud. Ligeledes er der testet flere scenarier, hvor forskellige værdier sendes til og fra de forskellige komponenter.

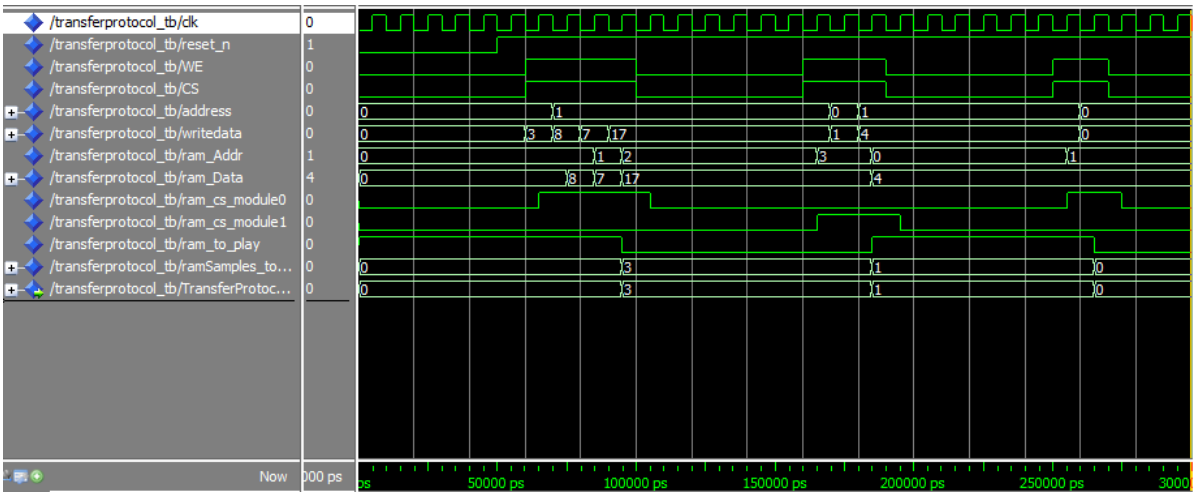
Grundet den mængde af data der sendes til ram-modulerne, for hver frekvens der ønskes afspillet, er der ikke testet nær det antal der sendes over, men blot enkelte, sekventielle værdier, således selve processen er vist funktionel. Hvis man ønskede at teste for, at alle værdier ville blive skrevet korrekt over vil det tage meget lang tid at verificere, bedst illustreret på figur 1.

$$\text{antal samples} = \frac{48000 \text{ Hz}}{\text{ønsket frekvens}}$$



Figur 1: Mængden af data der skal overføres (y-akse) for en frekvens (x-akse)

Første test er af komponentet *TransportProtocol*, der står for overførelsen af data fra microcontrolleren til ram-modulerne. På figur 2 ses et wave-udsnit af *TransportProtocol*'s testbench:



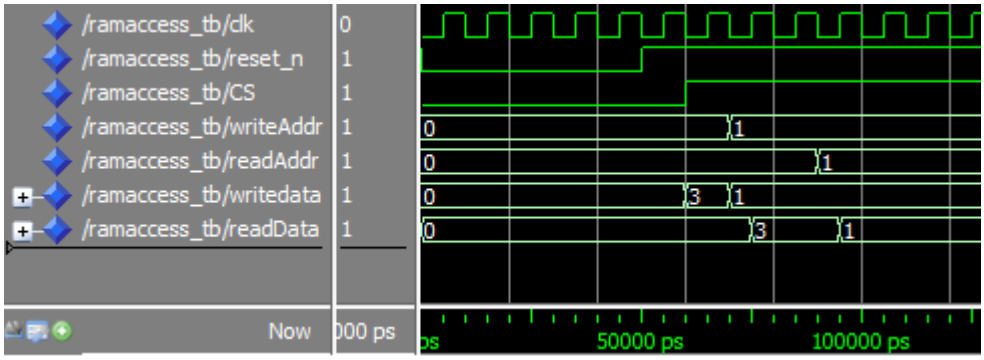
Figur 2: Waveforms for *TransportProtocol*

På figur 2 foretages der først et reset af komponentet (0-50 ns), hvorefter der skrives hvor mange samples der kommer (60-70 ns), værdierne skrives med 10 ns mellemrum (70-100 ns). Ved 95 ns bliver den sidste værdi overført og det angives hvor mange samples der er overført. Alt dette skrives til *ram_cs_module0*.

Efter 160 ns aktiveres *CS* igen og der skrives én ny værdi til ram-modulet. Denne skrives til *ram_cs_module1*, hvor *ram_cs_module0* er deaktiveret.

Der testes ligeledes også når der ikke sendes værdier over – altså hvor der ikke skal afspilles noget.

Komponentet *RamAccess* lagrer de overførte værdier.

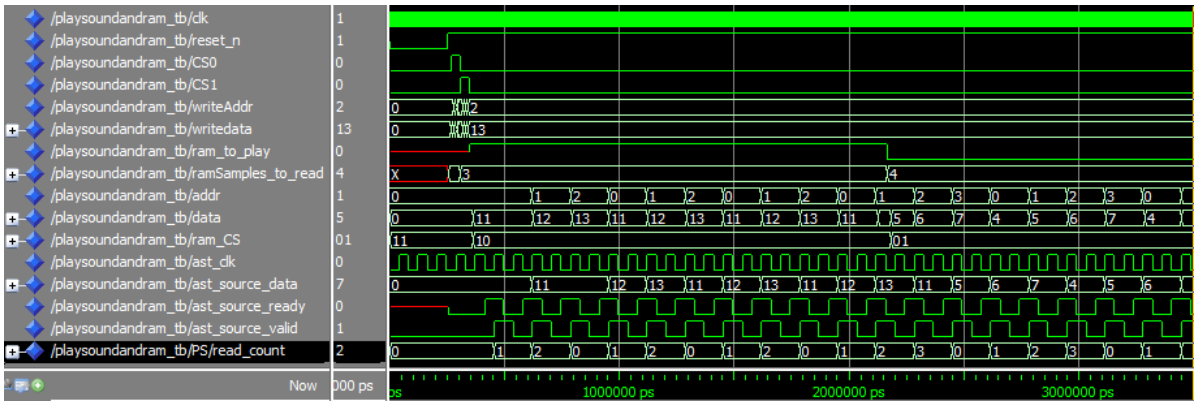


Figur 3: Waveforms for *RamAccess*

På figur 3 foretages der først et reset på 50 ns, chipselect aktiveres og der skrives til

adresse 0 med værdien 3 (60-70 ns). Herefter skrives til adresse 1 med værdien 1 (70-90 ns) og fordi *readAddress* ikke ændres, læses værdien fra adresse 0 ud (75-95 ns). Læsningsprocessen gentages fra 95 ns og frem.
Dette viser, at *RamAccess* umiddelbart virker som det skal.

Komponentet *PlaySound* er en smule mere omfattende og det testes sammen med *RamAccess* for at opnå det bedste resultat.



Figur 4: Waveforms for *Playsound* og *RamAccess*

Den viste sekvens på figur 4 er på 3,5 microsekunder. Først foretages et reset, for at nulstille alle værdier (0-250 ns), hvorefter der skrives nogle dummy-værdier ind i de to ram-moduler (270-350 ns). Herefter vælges læsning fra ram-modul1 (250-620 ns) og værdierne læses og genlæses indtil ram-modul0 vælges (620-2.170 ns). Det nye antal af værdier der skal læses bliver herefter læst ind (2.170-3.500 ns).

Overstående test viser, at der kan læses fra *RamAccess* når der ligger data deri, samt den kun afspiller det den bliver bedt om igen og igen.