

# Oscilloskop Projekt

Andreas Skåning, Joakim Butenko, Jonas Beck Jensen

Mads Rudolph, Sigurd Hestbech Christiansen

Juni 2025

Projektgruppe 4



Andreas  
s241123



Joakim  
s245946



Jonas  
s240324

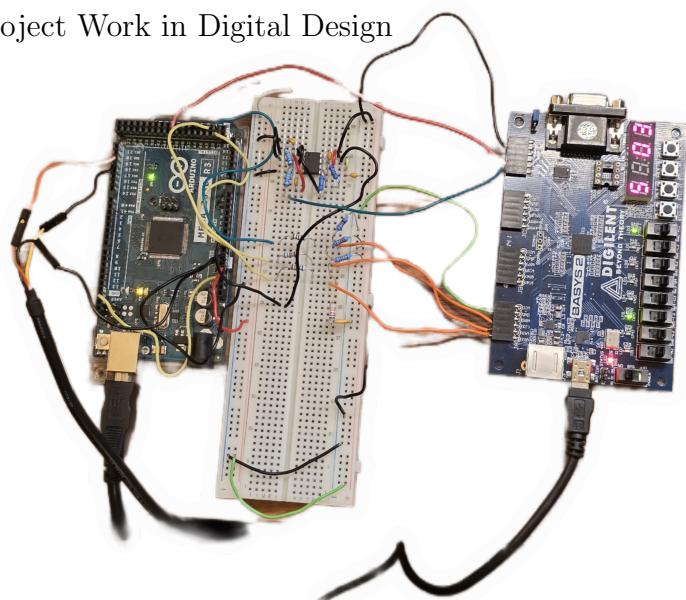


Mads  
s246132



Sigurd  
s245534

30082 - Project Work in Digital Design



**DTU Elektro**  
Diplomingeniør, Elektro

# Indholdsfortegnelse

<b>1 Indledning</b>	<b>4</b>
1.1 Baggrund . . . . .	4
1.2 Opgaven . . . . .	4
1.3 Kravspecifikation . . . . .	5
<b>2 Design</b>	<b>6</b>
2.1 Overordnet opsætning . . . . .	6
2.2 Oscilloskop . . . . .	7
2.2.1 Sampling af det analoge inputsignal . . . . .	7
2.2.2 Lagring af ADC-værdier i databuffer . . . . .	7
2.2.3 Kommunikation mellem LabVIEW og MCU . . . . .	7
2.2.4 Kommunikation fra MCU til FPGA . . . . .	8
2.2.5 Kodeopbygning . . . . .	8
2.2.6 Processer og interrupts . . . . .	9
2.2.7 Tidsrelationer mellem sampling, buffer og kommunikation . . . . .	10
2.3 Filterdesign . . . . .	11
2.4 Signalgenerator . . . . .	12
2.4.1 Signalgeneratorens grundlæggende funktion og ændringer . . . . .	12
2.4.2 Skifteregister og timing . . . . .	13
2.4.3 Protokolblok . . . . .	14
2.4.4 SigGendatapath . . . . .	15
<b>3 Implementering</b>	<b>17</b>
3.1 MCU-implementering . . . . .	17
3.2 Signalgenerator . . . . .	19
<b>4 Test</b>	<b>21</b>
4.1 Analyse af kravopfyldelse – MCU del . . . . .	21
4.2 Visualisering af signalformer . . . . .	24
4.3 Signalgenerator . . . . .	25
4.3.1 Fysiske tests . . . . .	25
4.3.2 Simuleringer . . . . .	29
4.4 Gennemgang af kravspisifikationerne . . . . .	32
<b>5 Bilag</b>	<b>33</b>
<b>Appendix A - Globals og funktioner fra source koden</b>	<b>33</b>
<b>Appendix B — Beregning af lavpasfilter</b>	<b>35</b>
5.1 VHDL-kode . . . . .	36
5.1.1 SigGenTop . . . . .	36
5.1.2 SigGenSPIControl . . . . .	38
5.1.3 ProtokolBlok . . . . .	39
5.1.4 std_8bit_reg . . . . .	42
5.1.5 SkifteReg . . . . .	43
5.1.6 TimingComponent . . . . .	44
5.1.7 SigGenDataPath . . . . .	45
5.1.8 SevenSeg5 . . . . .	47
5.1.9 DivClk . . . . .	49
5.1.10 DispMux . . . . .	50
5.1.11 BTNdb . . . . .	51
5.1.12 siggentop.ucf . . . . .	52
5.2 C-kode . . . . .	53
5.2.1 main.c . . . . .	53
5.2.2 UART1_comm.c . . . . .	55

5.2.3	SPI_comm.c . . . . .	59
5.2.4	ADC.c . . . . .	60
5.2.5	uart.c . . . . .	61
5.2.6	SPI.c . . . . .	62
5.2.7	UART1_comm.h . . . . .	63
5.2.8	SPI_comm.h . . . . .	63
5.2.9	ADC.h . . . . .	63
5.2.10	uart.h . . . . .	64
5.2.11	SPI.h . . . . .	64
	<b>Kildekode og GitHub</b>	<b>64</b>

# 1 Indledning

## 1.1 Baggrund

Denne rapport laves i forbindelse med 3-ugers kurset 30082 - Projektarbejde i Digital Design. Produktet består af 2 hoveddele, som naturligt lægger op til at gruppen inddeltes i 2 hold, som hver især arbejde på en del, som løbende gennem arbejdet testes sammen. Indelingen er sket således at 3 gruppemedlemmer (*Andreas, Jonas og Joakim*) arbejder på FPGA-enheten, og 2 medlemmer (*Mads og Sigurd*) arbejder med MCU-enheten.

Kurset ligger i naturlig forlængelse af semesterets tidligere kurser: Digital Teknik, og Data teknik og Programmering, hvor der blevet stiftet bekendsskab til arbejdet med både FPGA og MCU. FPGA-enheten er en Basys 2, som programmeres i VHDL, mens MCU-enheten er en microprocessor af typen Atmega2560, som kodes i C.

## 1.2 Opgaven

Opgavens formål var at gruppen skulle udarbejde et oscilloskop, som kan måle elektrisk spænding. Denne spænding kommer fra FPGA-enheten, der fungerer som signalgenerator. Den målte spænding skal derefter kunne vises ved hjælp af programmet "LabVIEW" på en computerskærm, der forbindes med systemet. Signalet som dannes i signalgeneratoren, skal kunne konfigureres fra computerenheden, således at Amplituden, Formen og Frekvensen kan varieres direkte derfra.

### 1.3 Kravspecifikation

Oscilloskop		Verifikation
ADC konvertering	Den analoge spænding fra signalgeneratoren skal måles fra 0 til 3.3 V med en oplosning på 8 bit.	Info*
Dataintegritet	Oscilloskopet skal ved alle nedenstående indstillinger kunne køre med kontinuert ubrudte målinger	Test* og Analyse
Parametre	Oscilloskopets samplerate og Record length skal kunne indstilles fra Labview programmet.	Test*
Samplerate min	Oscilloskopet skal kunne køre ned til 10 sps.	Test*
Samplerate max	Oscilloskopet skal kunne køre op til 5.000 sps.	Test*
	Oscilloskopet skal kunne køre op til 10.000 sps.	Test og Analyse
Record length min	Oscilloskopet skal kunne køre med ned til 10 ADC målinger i hver pakke.	Test*
	Den minimale tilladelige record length skal tage højde for sampleraten.	Test og Analyse
Record length max	Oscilloskopet skal kunne køre med op til 1000 ADC målinger i hver pakke for alle samplerater.	Test/ Analyse*
RS-232 baudrate	RS232 forbindelsen skal køre med en baud rate på 115.2 kbaud	Info*
RS-232 håndtering	Modtagelse af data fra LabView programmet skal foregå ved interrupt. Transmission kan foregå ved polling eller interrupt.	Info*
Signalgenerator		
PWM filter	Der skal designes et lav-pas filter der på passende vis udglatter de digitale PWM pulser.	Test/ Analyse*
Parametre	Signalgeneratorens signalform (SHAPE) , amplitude (AMPL) og frekvens (FREQ) skal kunne indstilles fra Labview programmet.	Test*
	SHAPE, AMPL og FREQ kan gøres synligt på syv segment displayet.	Test
Sinus signal	Der kan implementeres en look-up tabel i VHDL koden der gør det muligt at signalgeneratoren kan lave et sinus-formet signal	Test
SPI baudrate	SPI forbindelsen skal køre med en baudrate på 500 kbaud	Info*
SPI håndtering	To-vejs SPI kommunikation kan implementeres f.eks. med acknowledge handshake	Info
SPI protokol	Der skal vælges og implementeres en robust protokol til at overføre SHAPE, AMPL og FREQ	Analyse*
SPI test	Der skal ved test demonstreres en sikker forbindelse ved modtagelse. Denne test kan laves som et separat projekt med moduler fra det endelige oscilloskop projekt.	Test*

Verifikation skrevet med rødt og med asterisk (\*) er obligatorisk mens verifikation skrevet med grøn er ønskeligt. "Test" betyder at der i rapporten som minimum bliver skrevet at der er

## 2 Design

### 2.1 Overordnet opsætning

Opsætningen af projektet vil overordnet kunne beskrives med nedst  ende diagram. Det indeholder de prim  re logikblokke:

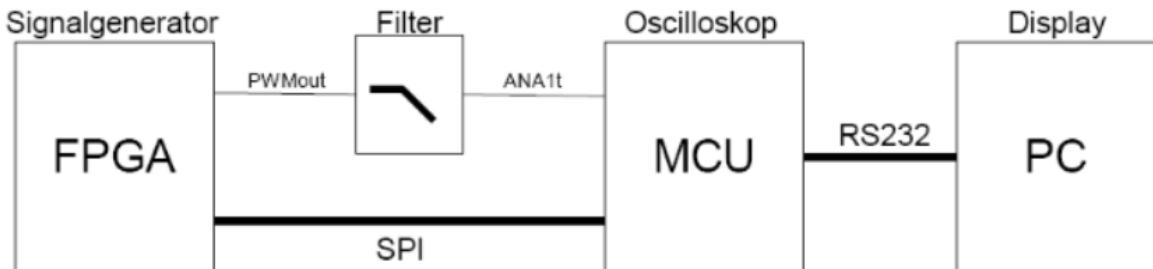


Figure 1: Blokdiagram over systemets ops  tning

Systemdesignet best  r, som illustreret i diagrammet, af f  lgende hovedkomponenter: en signalgenerator implementeret p   en FPGA, et oscilloskop bygget omkring MCU-enheden, samt en PC med LabView som grafisk brugergr  nseflade. Kommunikation mellem PC og MCU foreg  r via UART, og forbindelsen mellem MCU'en og Signalgeneratoren sker ved SPI kommunikation, hvor der i projektet er blevet udarbejdet en protokol, som dataen sendes med.

Brugeren interagerer med systemet gennem LabVIEW, hvor en r  kke parametre for det   nskede signal kan justeres, herunder:

- Amplitude: signalets sp  ndingsniveau
- Form (Shape): Signaltypen, konstant, sinus, savtak, firkant eller trekant.
- Frekvens: den   nskede frekvens for PWM-signalet

N  r brugeren   ndrer en parameter i LabVIEW, sendes kommandoen via UART til MCU'en, som fortolker signalbeskrivelsen og kommunikerer videre med FPGA'en via SPI. FPGA'en fungerer som signalgenerator og genererer et PWM-signal baseret p   de modtagede styresignaler. Dette PWM-signal filtreres via et lavpasfilter, s   det omdannes til et analogt signal, der kan m  les af MCU'en ADC-enhed.

Den m  lte v  rdi sendes derefter retur til PC'en, hvor signalet vises i realtid i LabVIEW. For en mere detaljeret forklaring af signalbehandlingen og filterets virkem  de henvises til det dedikerede afsnit om filteret.

## 2.2 Oscilloskop

MCU'en, *ATmega2560*, fungerer som hjernen i oscilloskopet. I dette underkapitel forklares, hvordan mikrocontrolleren styrer og koordinerer de centrale funktioner i systemet. Gennem de følgende underafsnit vil vi beskrive de vigtigste komponenter og processer, der tilsammen udgør oscilloskopets funktionalitet:

- Sampling af det analoge inputsignal
- Lagring af ADC-værdier i databuffer
- Kommunikation mellem LabVIEW og MCU
- Kommunikation fra MCU til FPGA
- Kodeopbygning
- Processer og interrupts
- Tidsrelationer mellem sampling, buffer og kommunikation

### 2.2.1 Sampling af det analoge inputsignal

I vores design modtager MCU'en den analoge spænding fra lavpasfilteret via den analoge indgang A0, som er forbundet til ADC0. ADC0 er én af flere indbyggede analog-til-digital konvertere (ADC'er) i *ATmega2560*-mikrocontrolleren.

ADC'en er konfigureret til at benytte auto-trigger-funktionen, hvilket muliggør sampling med en fast frekvens, som specificeres via LabVIEW. Auto-triggeren aktiveres, når Timer1 sætter sit compare match-interrupt flag. Ved at ændre den værdi, som Timer1 tæller op til, kan vi justere samplingfrekvensen. Dette gør det muligt at tilpasse oscilloskopets tidsopløsning og tidsbase direkte fra user interface i LabVIEW.

### 2.2.2 Lagring af ADC-værdier i databuffer

De digitaliserede ADC-værdier lagres i et `uint16_t` array, som fungerer som databuffer med kapacitet til 1000 individuelle 16-bit heltalsværdier. Selve lagringen foregår i interrupt service-rutinen for Timer1 Compare Match (`TIMER1_COMP_vect`).

I denne ISR (Interrupt Service Routine) anvendes to korte `if`-betingelser: Den første indsætter den aktuelle ADC-værdi i databufferen på det aktuelle indeks og inkrementerer indekset. Den anden betingelse nulstiller indekset i tilfælde af fejl eller sætter et flag, når bufferen er fuld og klar til afsendelse.

Ved at udføre lagringen direkte i interrupt-rutinen sikres det, at data opsamles og gemmes med minimal forsinkelse, hvilket er afgørende for at opretholde en stabil og præcis sampling.

### 2.2.3 Kommunikation mellem LabVIEW og MCU

Til kommunikationen mellem MCU og LabVIEW anvendes en UART-forbindelse, hvor data udveksles i form af pakker struktureret efter den protokol, der er defineret i LabviewOscilloscope-programmet. Protokollen gør brug af en synkroniseringsssekvens, længdeangivelse, datatype, datafelt og en checksum, hvilket sikrer, at datapakker kan genkendes og valideres korrekt.

MCU'en er designet til at kunne modtage tre typer pakker fra LabVIEW, afhængigt af hvilken fane brugerne interagerer med i programmet. Disse inkluderer knapkommandoer fra Generator-tabben, konfigurationspakker til ændring af samplingparametre og en separat knap til at igangsætte systemtests. Ved at følge denne protokol kan MCU'en fortolke brugerens input og videresende de relevante indstillinger til signalgeneratoren.

Når ADC'en på MCU'en har samlet et sæt målinger, returneres disse til LabVIEW i en datapakke med samme overordnede struktur. Dette muliggør en kontinuerlig opdatering af signalvisningen i oscilloskopet.

For at sikre, at datastrømmen fra ADC'en til LabVIEW kan opretholdes uden afbrydelser, har vi valgt at implementere et triple-buffer system. Det består af tre separate buffere, som skifter rolle undervejs i samplingprocessen. Denne løsning muliggør, at nye samples kan indsamles samtidig med, at tidligere samples sendes videre, uden at de to processer blokerer for hinanden. Det er særligt vigtigt ved høje samplingrater, hvor det ellers ville være svært at undgå datatab.

Denne arkitektur gør det muligt at opretholde kontinuerlig, realtids dataoverførsel fra MCU'en til LabVIEW, og den er samtidig skalerbar til forskellige samplingbehov og bufferstørrelser.

#### 2.2.4 Kommunikation fra MCU til FPGA

MCU'en kommunikerer med FPGA'en via SPI, hvor MCU'en fungerer som master og FPGA'en som slave. Til denne kommunikation har vi designet en simpel protokol, der sikrer pålidelig og entydig overførsel af kommandoer og data mellem enhederne.

Formålet med protokollen er at strukturere kommunikationen, så både timing og datastruktur overholdes, og fejl undgås. Protokollens opbygning og implementering vil blive uddybet i implementeringsafsnittet.

## 2.2.5 Kodeopbygning

Vi har illustreret opbygningen af MCU'ens source-kode med et strukturdiagram. Diagrammet viser, hvordan de forskellige softwaremoduler interagerer, og hvordan hardwarelaget abstraheres gennem et hardware abstraction layer (HAL). Diagrammet kan ses i figur 2.

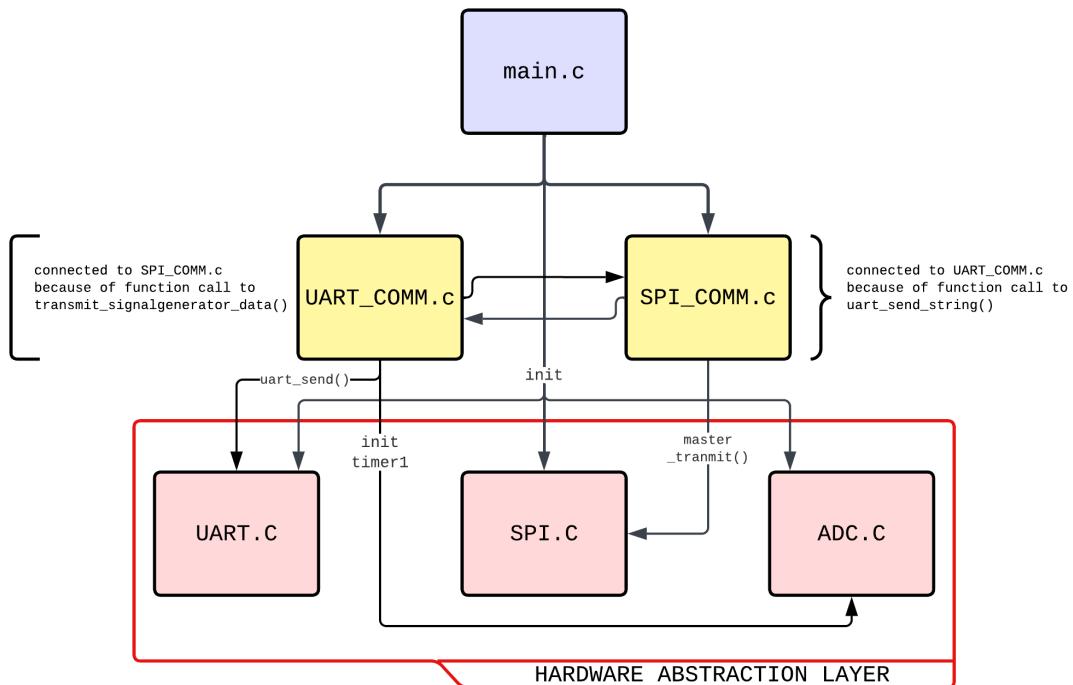


Figure 2: Proces diagram af oscilloskop

## 2.2.6 Prossesser og interrups

Vi har illustreret processerne som foregår i MCU'en, med et process diagram.

Diagrammet kan ses i figur 3.

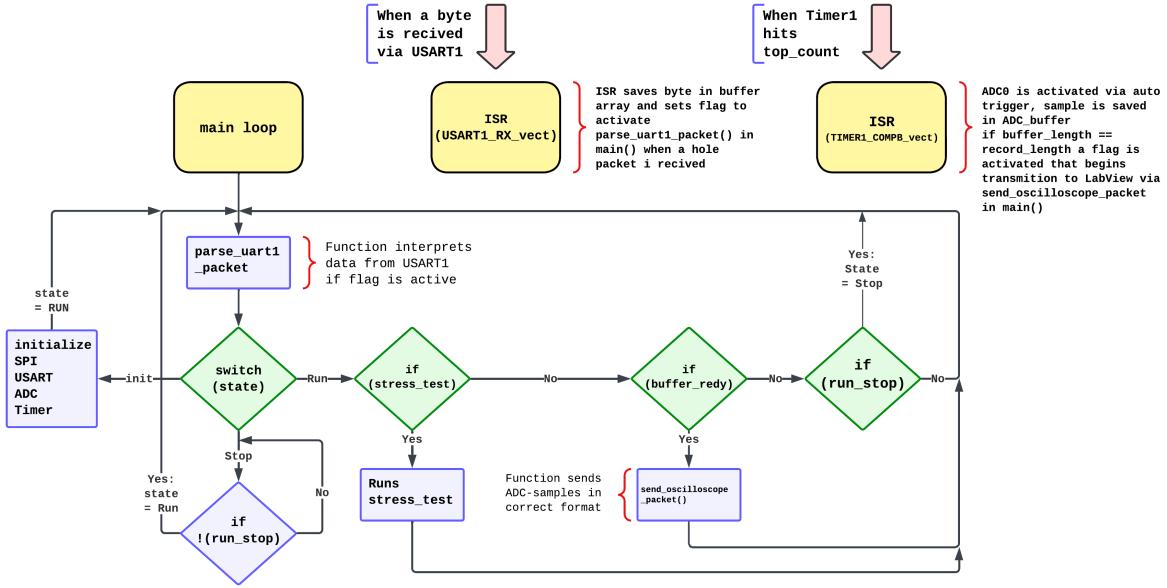


Figure 3: Proces diagram af oscilloskop

Her er en liste med forklaringer til processerne i figur 3:

- **Main loop**

Den centrale løkke, der kontinuerligt overvåger systemets tilstand og reagerer på flags sat af interrups.

- **Init-proces**

Udføres ved første gennemløb af main loop og initialiserer USART0, USART1, SPI, Timer1 og ADC0. Denne proces udføres kun én gang.

- **ISR (USART1\_RX\_vect)**

Interrupt service-rutine, der aktiveres, når en byte modtages via USART1. Den modtagede byte gemmes i et buffer-array. Når en komplet datapakke er modtaget, sættes et flag, som aktiverer videre behandling i main loop.

- **ISR (TIMER1\_COMPB\_vect)**

Interrupt service-rutine, der aktiveres, når Timer1 når sin topværdi. ADC0 starter sampling via auto-trigger, og værdien gemmes i et buffer-array. Når antallet af samples svarer til record length, sættes et flag, som aktiverer dataoverførsel i main loop.

- **parse\_uart1\_packet()**

Funktion, der aktiveres, når flaget fra USART1\_RX\_vect er sat. Funktionen fortolker den modtagede datapakke og udfører relevante handlinger baseret på pakkens indhold.

- **send\_oscilloscope\_packet()**

Funktion, der aktiveres, når flaget fra TIMER1\_COMPB\_vect er sat. Funktionen sender data fra ADC-bufferen til LabVIEW via USART1.

- **Stop-tilstand**

Når brugeren trykker på pause i LabVIEW, skifter systemet til denne tilstand. Her afventer systemet, at brugeren trykker på start igen, hvorefter det vender tilbage til Run-tilstanden.

- **Stress test**

Hvis `stress_test`-flaget er sat, sendes 10.827 fulde instruktionssæt til FPGA'en. FPGA'en har et tællerregister, der optæller modtagede bytes, hvilket bruges til at verificere stabiliteten af den serielle kommunikation.

### 2.2.7 Tidsrelationer mellem sampling, buffer og kommunikation

Baudraten fungerer som den begrænsende faktor, for hvor høj en sample-rate vores oscilloskop kan benytte. Dette skyldes, at ADC-bufferen potentielt kan blive fyldt hurtigere, end UART-kommunikationen kan nå at overføre data. Derfor er det nødvendigt at begrænse ADC'ens sample-rate for at undgå buffer-overflow.

Vores baudrate er 115200 *bit/s*, hvilket svarer til 11520 *byte/s*, da hver byte overføres med 10 bit (8 data + start + stop). Den maksimale sample-rate, som kan opretholdes kontinuerligt, afhænger af både baudrate og *record length* (størrelsen af databufferen).

Vi definerer følgende størrelser:

$$\mathbf{RL} = \text{Record length (antal samples)}, \mathbf{BR} = \text{Byte rate (11520 byte/s)}, \mathbf{SR} = \text{ADC sample-rate (samples/s)}$$

For at sikre kontinuerlig drift uden tab af data, skal tiden det tager at fyldde bufferen med samples være lig med (*for at finde max-værdi*) eller større end tiden det tager at sende hele pakken over UART. Dette kan udtrykkes med følgende ligninger:

- **Tid det tager at sende en pakke:**

$$t_{\text{send}} = \frac{7 + RL}{BR}$$

- **Tid det tager at fyldde bufferen:**

$$t_{\text{sample}} = \frac{RL}{SR}$$

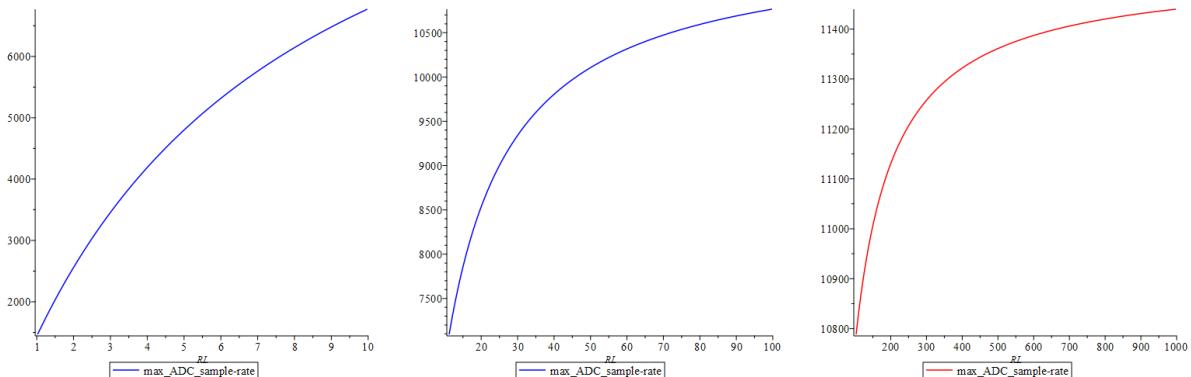
#### Sammenhæng mellem sample rate og record length

Ved kontinuerlig drift uden datatab gælder følgende udtryk for den maksimale sample-rate:

$$SR_{\max} = \frac{BR \cdot RL}{7 + RL}$$

Denne sammenhæng viser, at jo større *record length*, desto højere sample-rate kan opnås, da overheadet fra de 7 kontrolbytes bliver relativt mindre. Denne formel finder den teoretiske max sample frekvens, i virkeligheden er det fornuftigt at indstille sample frekvensen lidt lavere end den teoretiske max.

Vi har plottet funktionen for  $SR_{\max}$  i tre intervaller for RL



## 2.3 Filterdesign

For at kunne måle det genererede signal korrekt anvender systemet et analogt lavpasfilter mellem signalgeneratorens PWM-udgang og MCU'ens ADC-indgang. PWM-signalen svinger mellem 0 og 3.7 V og skal filtreres, så det fremstår som en jævn analog spænding proportional med pulsbredden. For mere detaljeret gennemgang af udregninger af Q-værdi og komponentvalg henvises til [section 5](#).

Filteret er designet som et aktivt 4. ordens lavpasfilter baseret på Sallen-Key topologi. Det består af to kaskadekoblede 2. ordens lavpasfiltre, hvilket samlet giver en stejl dæmpning på -80 dB/decade efter grænsefrekvensen. Målet er at undertrykke PWM'ens højfrekvente komponenter effektivt, så kun den ønskede grundbølge forbliver.

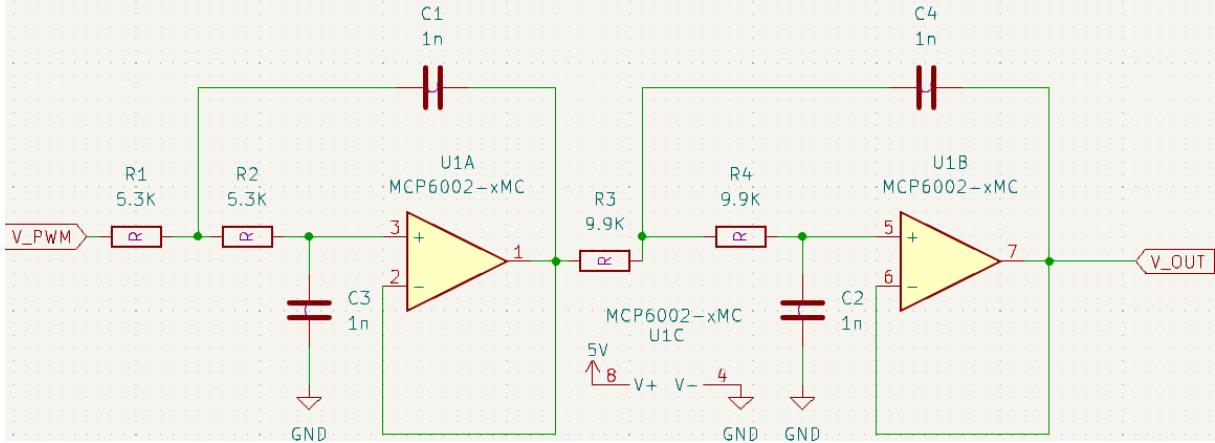


Figure 5: 4. ordens aktivt lavpasfilter med Sallen-Key topologi

Filteret anvender MCP6002 opamps, som fungerer ved enkeltforsyning på 5 V. Alle kondensatorer er 1 nF, og modstandene er valgt for at opnå passende grænsefrekvensen( $f_c$ ):

- Første trin:  $R = 5.3 \text{ k}\Omega$ ,  $C = 1 \text{ nF} \rightarrow f_c \approx 30.0 \text{ kHz}$
- Andet trin:  $R = 9.9 \text{ k}\Omega$ ,  $C = 1 \text{ nF} \rightarrow f_c \approx 16.1 \text{ kHz}$

### Filtertest ved 9.4 kHz sinus

For at verificere lavpasfilterets funktion blev der genereret et sinusformet PWM-signal med en ønsket frekvens på ca. 9.4 kHz via LabVIEW. De følgende oscilloskopmålinger viser både det rå PWM-signal, det filtrerede analogsignal samt en samlet visning af begge signaler.



Figure 6: Oscilloskopmålinger før og efter lavpasfilter ved genereret sinusformet PWM-signal

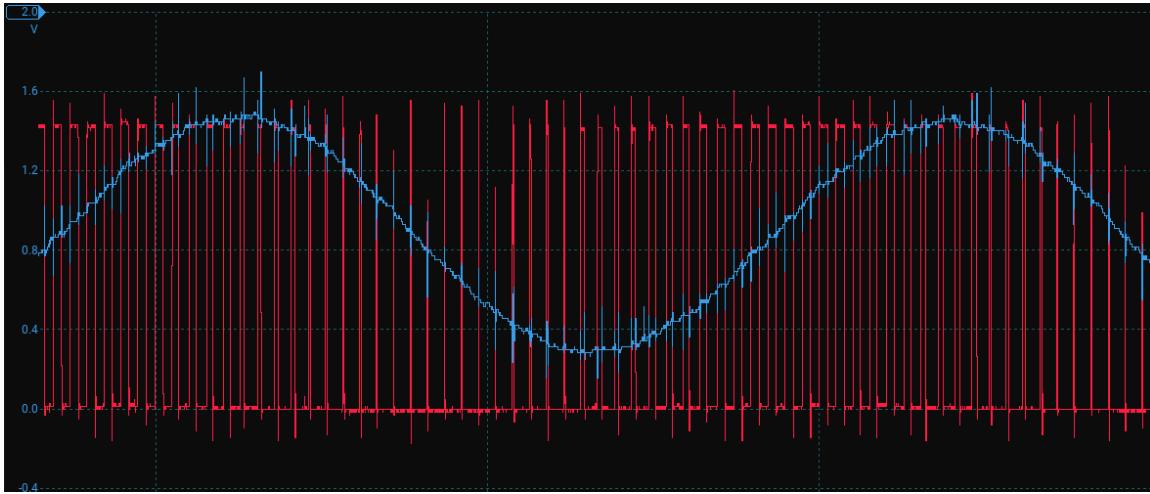


Figure 7: Samtidig visning af PWM-signal (rød) og det filtrerede analogsignal (blå)

Som det ses, glatter filteret effektivt de højfrekvente komponenter i PWM'en og producerer en ren analog sinusform. Dette bekræfter, at det designede lavpasfilter fungerer som ønsket og muliggør korrekt ADC-sampling i MCU'en.

## 2.4 Signalgenerator

### 2.4.1 Signalgeneratorens grundlæggende funktion og ændringer

FPGA'en er designet til at styre en signalgenerator, hvor brugeren kan navigere mellem indstillinger for signalform (Shape), amplitude og frekvens. I den oprindelige opsætning blev nogle funktioner styret direkte via fysiske knapper på signalgeneratorens frontpanel. I vores version er disse flyttet til computegrænsefladen i LabVIEW, for at samle styringen ét sted og gøre betjeningen mere intuitiv.

Det betyder, at både reset-funktionen (tidligere BTN3) og start/stop-funktionen (BTN2) nu håndteres via LabVIEW, mens FPGA'en kun bevarer en enkelt knap (BTN1). Denne bruges til at skifte mellem visning af de tre registre – Shape, Amp og Freq – så brugeren fortsat kan følge med i de aktuelle værdier via displayet.

Ændringerne er gennemført for at reducere kompleksiteten i brugerfladen på selve hardwareenheden og sikre, at alle relevante indstillinger og kontroller er tilgængelige centralt fra PC'en.

#### 2.4.2 Skifteredister og timing

For at FPGA'en kan agere SPI-slave, har vi implementeret et skifteredister, der indlæser data fra MCU'en via **MOSI**. Når **SSNOT** (Slave Select, aktiv lav) bliver trukket lav af MCU'en, begynder skifteredistret at skifte den aktuelle værdi på **MOSI** ind ved hver opadgående flanke på **SCLK**.

For at synkronisere overførslen og signalere, hvornår en komplet byte er modtaget, har vi implementeret en timing-komponent.

**Timingkomponenten** fungerer ved at bruge signalet **SSnot** til generere en puls på **DataReady**, når **SSnot** går fra lav til høj. Dette opnås ved at gemme den tidligere værdi af **SSnot** og sammenligne den med den nuværende. Med en inverter og en AND-port dannes følgende logiske udtryk:

$$\text{DataReady} = \sim \text{SSnot}_{\text{old}} \& \text{SSnot}$$

Dette sikrer, at **DataReady** kun bliver høj i det clockcyklus, hvor **SSnot** skifter fra lav til høj – altså ved afslutningen af en SPI-overførsel. Pulsen bruges derefter til at signalere, at en komplet byte er modtaget og klar til behandling.

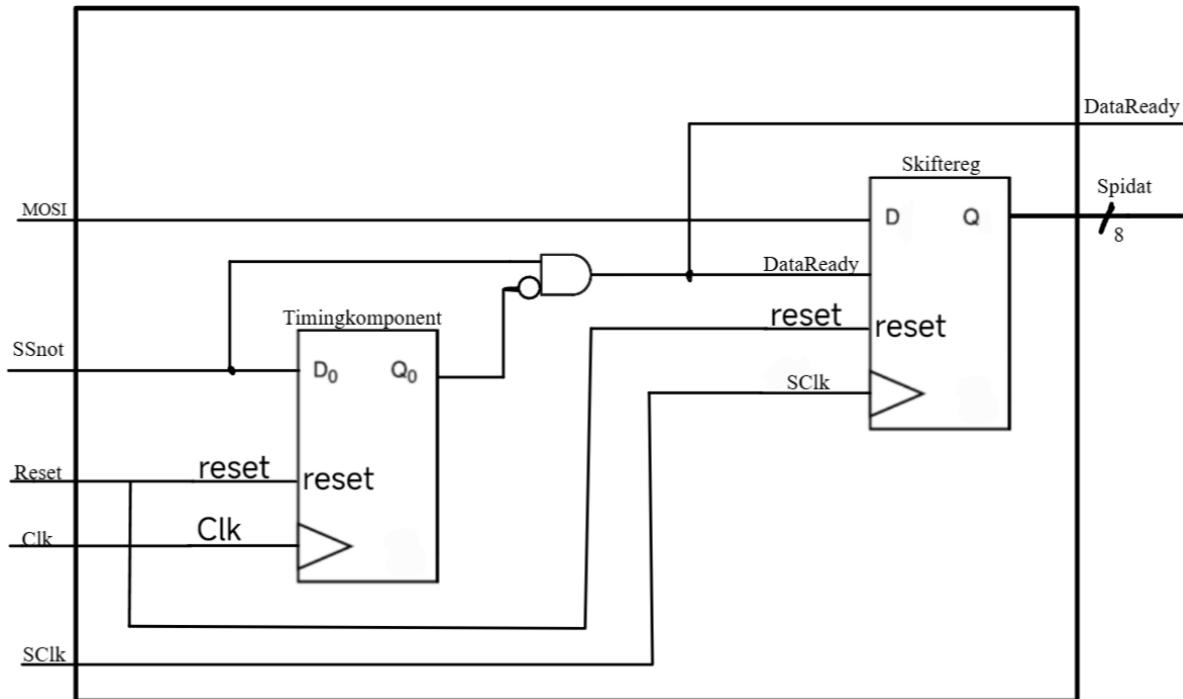


Figure 8: Illustration af Skifteredister og Timingkomponent

### 2.4.3 Protokolblok

Protokolblokken er designet til at håndtere de data, der modtages via SPI-forbindelsen. Den er implementeret som en tilstandsmaskine (FSM) med følgende tilstande: **IDLE**, **ADDRS**, **DataS**, **CheckSumEnS**, **CheckSumS**, **AmpS**, **FreqS** og **ShapeS**.

FSM'en starter i **IDLE**-tilstanden, hvor den afventer modtagelse af en synkroniseringsbyte (01010101) via SPI. Når denne byte er modtaget, deaktiveres **SigEN** for at forhindre, at systemet sender ugyldige data videre under modtagelsen.

I **ADDRS**-tilstanden gemmes adressen i **ADDR**-registeret, som bruges til at bestemme, hvilken type data der modtages. Derefter opdateres Data-registeret i **DataS**-tilstanden. Tilstanden **CheckSumEnS** er indført som en separat tilstand for at sikre korrekt timing mellem modtagelse af data og opdatering af **CheckSum**-registeret. Ved at adskille aktiveringstenget af registeret fra selve valideringen i **CheckSumS**, sikres det, at dataen har haft tid til at blive korrekt latched ind, før den sammenlignes med den beregnede **checksum**. Dette designvalg øger robustheden og forudsigteligheden i FSM'en sekventielle forløb.

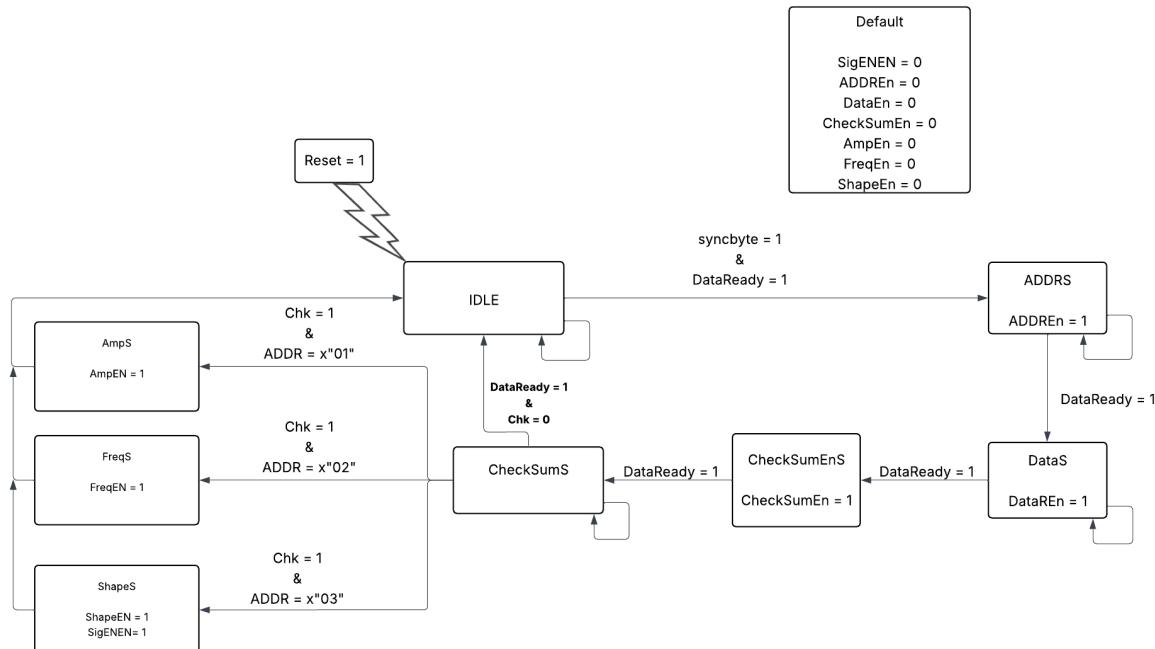


Figure 9: Tilstands diagram

I **CheckSumS**-tilstanden sammenlignes den modtagne checksum med en beregnet checksum i FPGA'en. Hvis de stemmer overens ( $\text{Chk} = '1'$ ), fortsætter FSM'en til en af de tre datatilstande afhængigt af adressen: **AmpS**, **FreqS** eller **ShapeS**. I **ShapeS**-tilstanden aktiveres også **SigENEN**, hvilket signalerer, at systemet er klar til at sende data videre. Dette er valgt, da LabVIEW altid sender data i en fast sekvens: Amplitude → Frekvens → Shape.

Derudover indeholder protokolblokken en stresstest af SPI-kommunikationen som kan ses beskrevet og illustreret nede i test afsnittet.

Denne mekanisme gør det muligt at verificere, at alle datapakker er modtaget korrekt, og at checksummen har været valid for hver pakke.

#### 2.4.4 SigGendatapath

Denne blok er den del af signalgeneratoren som står for at danne selve det udgående PWM-signal. Designet er udleveret af vejlederne, og derefter konfigureret til at imødekomme den ønskede funktionalitet.

Komponenten kan danne nogen forskellige signalformer. Disse er følgende:

- Konstant(DC)
- Firkant
- Savtak
- Sinus
- Trekant

Hvilken type signal der produceres afhænger af et 3-bit styresignal benævnt "Shape". De forskellige former, er tildelt disse værdier i styresignalet:

Shape værdi	Signalform
000	Konstant
001	Firkant
010	Savtak
011	Sinus
100	Trekant

Table 1: Signalformer baseret på Shape-værdi

De øvrige indgange til komponenten er følgende:

- Clk (Systemets clock)
- Reset
- SigEN (aktiveringssignal)
- Ampl: Amplitude (8-bit)
- Freq: Frekvensparameter (8-bit)

Komponentens eneste output kaldes PWMOut

### Intern opbygning

#### PWM-generator

PWM-signalet dannes ud fra en intern 9-bit tæller (PWMCntvar), der inkrementeres med et fast trin (PWMinc) på hver clockcyklus. Når tælleren overstiger maksimumsværdien (111111111), nulstilles den til nul. Tællerværdien bruges til 2 primære formål:

1. At bestemme PWM duty cycle, via en sammenligning med den ønskede amplitude (SigAmpl)
2. At generere et "wrap-signal" benævnt "PWMWrap", der indikerer at en PWM periode er afsluttet.  
Dette signal er også vigtigt da det bruges til at opdatere signalets værdi

### **Frekvensstyring**

Frekvensparametren (Freq) behandles i en simpel dekoder (FreqDec), som genererer et 12-bit tal (FreqCnt). Denne værdi bruges til at opdatere en signalfase-tæller (SigCnt). Denne tæller øges kun, når PWMwrap er høj – altså én gang pr. PWM-periode – hvilket betyder, at signalets ”form” udvikler sig langsommere, jo lavere Freq er.

### **Signalgenerering**

Afhængigt af den valgte Shape genereres en 8-bit signalværdi (Sig), som beskriver det ønskede signaludtryk. F.eks. anvendes en lookup-table (SinusLUT) til sinusformen, mens firkant og savtak danner direkte ud fra SigCnt. Denne signalværdi multipliceres derefter med Ampl via en binær multiplikationsproces, og resultatet reduceres til 7 bit (SigAmpl), som bruges som PWM-sammenligningsværdi.

### **PWM-komparator**

PWM-signalet (PWM) sættes høj, når PWM-tælleren (**PWMcnt**) er mindre end eller lig med SigAmpl. Ellers er det lavt. Det endelige output (**PWMOut**) er PWM, gated med **SigEN**, så signalet kun er aktivt, når systemet er aktiveret.

### **Trekantssignal**

Trekantsignalet er udviklet af gruppen selv og implementeret som en VHDL-process. Signalformen danner ud fra tællerens **SigCnt**, der fungerer som faseindikator for signalforløbet. Når tælleren er i første halvdel af sin periode (dvs. mindre end x"800"), bruges et udsnit af bit 10 `downto` 3 direkte som signalværdi. Dette udgør den stigende del af trekantsignalet.

Vores trekantsignal minder meget om det udleverede savtak-signal. Forskellen er, at vi i trekantsignalet har behov for både at tælle op og ned, hvilket i praksis fordobler antallet af punkter på grafen og dermed forlænger perioden. For at undgå at halvere frekvensen har vi derfor ændret det bitspænd, vi bruger i **SigCnt**, fra (11 `downto` 4) til (10 `downto` 3). Denne ændring sikrer, at vi bevarer samme frekvens som vi har fået fra SPI-kommunikationen, selvom tælleren nu dækker en dobbelt sekvens (op og ned).

Når **SigCnt** overstiger x"800", vendes forløbet ved at invertere bitværdien (`not SigCnt(10 downto 3)`), hvilket skaber den faldende del. På den måde danner en symmetrisk trekantbølge, hvor signalet først stiger lineært og derefter falder i samme tempo.

Ved at udvælge og bruge et midterstykke af **SigCnt**-tælleren (8 bit fra en 12-bit tæller), sikres det, at signalets frekvens er passende, og at oplosningen passer til systemets PWM-niveau.

Denne metode sikrer også, at hele den tilgængelige amplitude i signalet udnyttes uden skaleringstab, og at signalet har et lineært og periodisk forløb – hvilket er karakteristisk for et ideelt trekantsignal.

## 3 Implementering

I dette implementerings afsnit fokuserer vi på den praktiske realisering af oscilloskopets software. Gennem en samlet beskrivelse af sampling, bufferstyring og kommunikation med LabVIEW og FPGA'en opnås en teknisk forståelse af systemets funktion og samspil mellem modulerne.

### 3.1 MCU-implementering

Firmware på MCU'en er struktureret i separate moduler, som håndterer sampling, bufferstyring og kommunikation med både LabVIEW og FPGA'en. Systemet er drevet af interrupts og styret gennem en hovedløkke, der reagerer på flag sat af afbrydelser.

#### Sampling og triple-buffer-system

Sampling udføres af ADC0 på den analoge indgang A0 og triges af Timer1 Compare Match B-interruptet. I interrupt service-rutinen TIMER1\_COMPB\_vect læses ADC-værdien (8-bit, ADCH) og gemmes i `active_buffer`. Når bufferen er fuld (dvs. `sample_index == record_length`) og der ikke pågår dataoverførsel, roteres bufferne: `active_buffer` bliver til `send_buffer`, `standby_buffer` bliver aktiv, og den tidligere `send_buffer` går i standby. Rotationen udføres mens interrupts er midlertidigt slæt fra (`cli()` og `sei()`), hvilket sikrer, at sampling ikke foregår samtidig med buffer-ombytningen. Denne triple-buffer arkitektur sørger, at der kan opsamles data der kan sendes til LabVIEW uden datatab.

#### UART1-modtagelse og parsing

MCU'en modtager pakker fra LabVIEW via UART1. Hver byte modtages i interruptet USART1\_RX\_vect og gemmes i en RX-buffer. Når den modtagede længde matcher længdefeltet i pakken, sættes et flag, Vi har illustreret ISR(USART1\_RX\_vect) i et flowchart (figur 10).

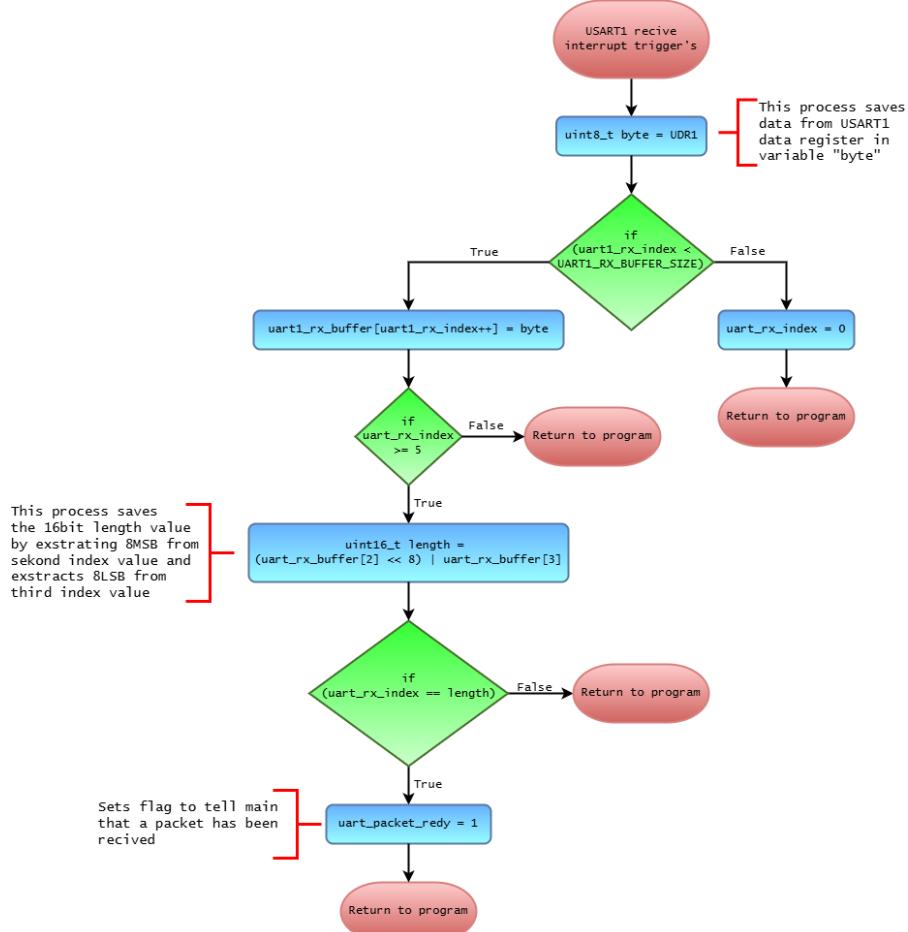


Figure 10: Flowchart af ISR(USART1\_RX\_vect)

Efter `buffer_uart_ready` flaget er sat kalder hovedprogrammet `parse_uart1_packet()` for at analysere pakken. Parseren håndterer følgende pakker:

- **Type 0x01 (BTN):** Repræsenterer knaptryk i Generator-tabben i LabVIEW. MCU'en opdaterer en aktiv parameter (amplitude, frekvens eller signalform) eller skifter fokusfelt. Når *Enter* trykkes, sendes de aktuelle parametre til FPGA'en via SPI. Samtidig sendes en statuspakke tilbage til LabVIEW, så GUI'en kan opdateres.
- **Type 0x02 (SEND):** Indeholder ny `record_length` og sample rate. MCU'en beregner en ny TOP-værdi for Timer1 ud fra baudraten og konfigurerer timeren til at matche den ønskede samplingfrekvens.
- **Type 0x04 (START):** Triggles fra Bode-plot-tabben. Når denne modtages, sættes et flag, og MCU'en igangsætter en stresstest, hvor 10.827 SPI-pakker sendes til FPGA'en i høj hastighed.

### Afsendelse af ADC-data til LabVIEW

Når en buffer er fyldt med samples, og `buffer_ready` er sat, kaldes `send_oscilloscope_packet()` i main-loop'en. Funktionen pakker dataen i henhold til LabVIEW-protokollen (sync, length, type, data, checksum) og sender pakken byte-for-byte via UART1. Under transmission deaktiveres sampling midlertidigt for at undgå, at buffere ændres midt i afsendelsen. Når pakken er sendt, genaktivieres sampling og buffere roteres igen.

### Statusopdatering til LabVIEW GUI

Efter enhver parameterændring – f.eks. som resultat af et knaptryk – sendes en statuspakke tilbage til LabVIEW via funktionen `send_generator_packet()`. Denne pakke indeholder de aktuelle værdier for amplitude, frekvens og form, så GUI'en altid afspejler systemets faktiske tilstand.

### SPI-kommunikation fra MCU til FPGA

SPI benyttes som envejskommunikation til at overføre de nyeste signalparametere til FPGA'en. Funktionen `transmit_signalgenerator_data()` samler tre SPI-pakker (én for hver parameter) bestående af sync-byte (0x55), adresse (1-3), databyte og en checksum beregnet som XOR af de tre første bytes. Disse sendes sekventielt med `master_transmit()`, som håndterer SPI-dataregistrering og toggler SS-linjen for at indramme hver overførsel. Transmissionen initieres typisk ved et *Enter*-tryk i LabVIEW, hvilket indikerer at en ny konfiguration er klar til brug.

### Sekvensdiagram

Følgende sekvensdiagram viser dataflowet mellem centrale funktioner i MCU'en, fra sampling og buffer-rotation til kommunikation med LabVIEW og signalgeneratoren.

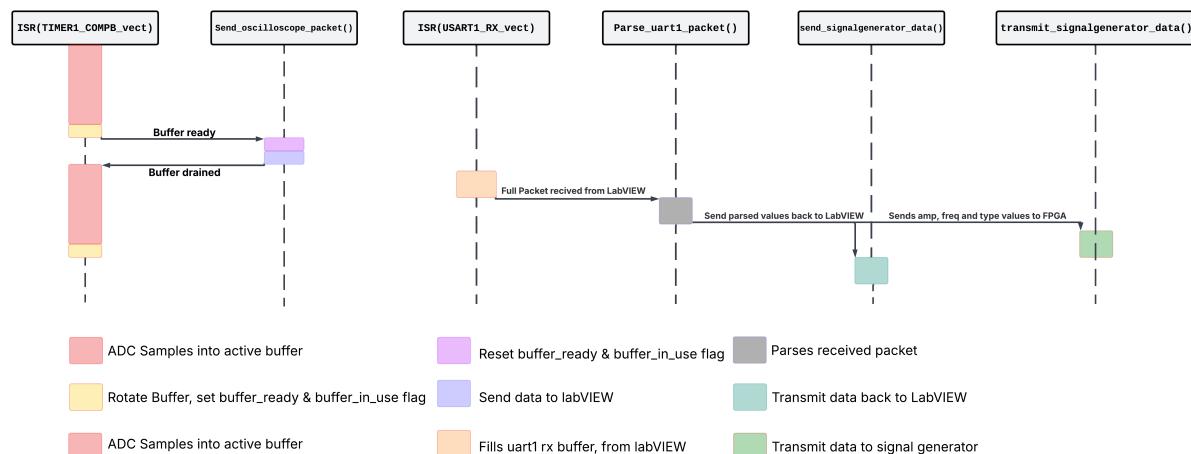


Figure 11: Sekvensdiagram: Dataflow mellem LabVIEW, MCU og Signalgenerator

### 3.2 Signalgenerator

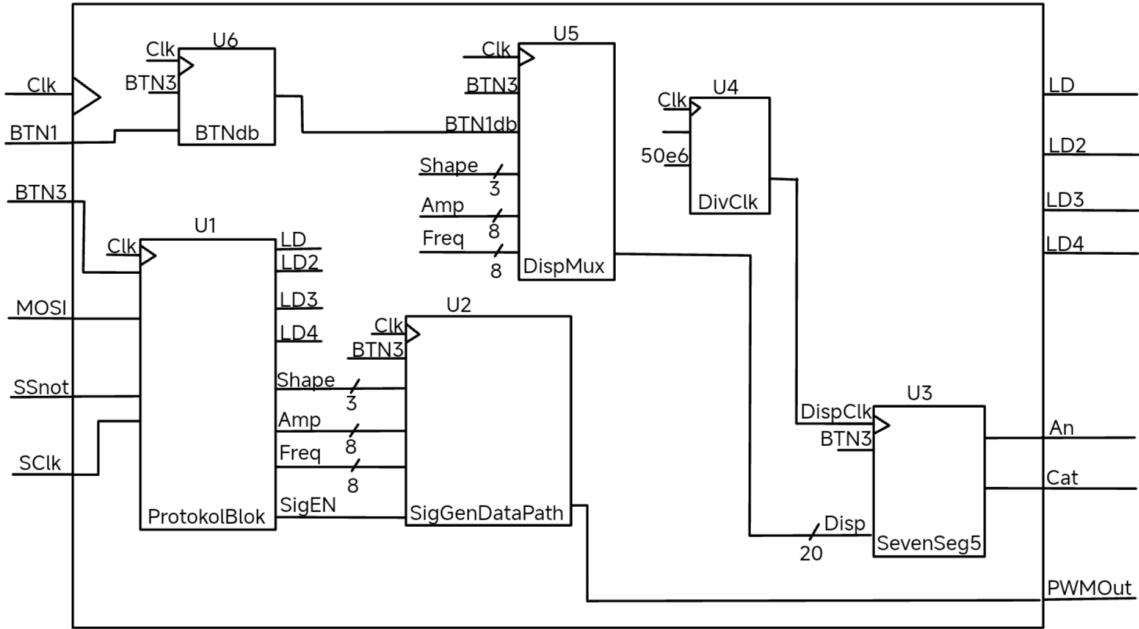


Figure 12: Blokdiagram af signalgeneratoren

Diagrammet viser den overordnede struktur for signalgeneratoren og illustrerer, hvordan de enkelte komponenter interagerer. Systemet modtager signaler fra en mikrocontrolleren via **BTN3** (reset), **MOSI**, **SSnot** og **SClk**. Disse signaler håndteres af **ProtokolBlok** (**U1**), som tolker SPI-kommunikationen og overfører de modtagne data til **SigGenDataPath** (**U2**). Denne datapath genererer det ønskede signal baseret på tre parametre: **Shape**, **Amplitude** og **Frekvens**. Det genererede signal sendes derefter ud via **PWMOut**, som kan forbides til et lavpasfilter for at opnå et glat analogt signal.

Samtidig sendes de modtagne signalparametre videre til **DispMux** (**U5**), som vælger, hvilke data der vises for brugeren. **SevenSeg5** (**U3**) står for at vise data på et 7-segment display. For at displayet opdateres i et passende tempo, anvendes **DivClk** (**U4**) til at nedskalere systemets clock-signal. Det reducerede kloksignal (DispClk) sikrer, at displayet ikke opdateres for hurtigt til at kunne læses.

Knappen **BTN1** føres gennem et debounce-modul (**BTNDb**, **U6**), der stabiliserer signalet, før det bruges til at skifte visningen mellem frequency, shape og amplitude på displayet. **BTN3** derimod debounces ikke, da det antages, at signalet er stabilt, da det kommer direkte fra mikrocontrolleren. Signalet fungerer som reset-signal, og er derfor forbundet til alle komponenter.

Derudover vises LED'er (**LD**, **LD2–LD4**) som indikationer: én for at vise, om signalgeneratoren er aktiv (**SigEN**), og de øvrige for at vise hvor mange pakker der er blevet modtaget korrekt.

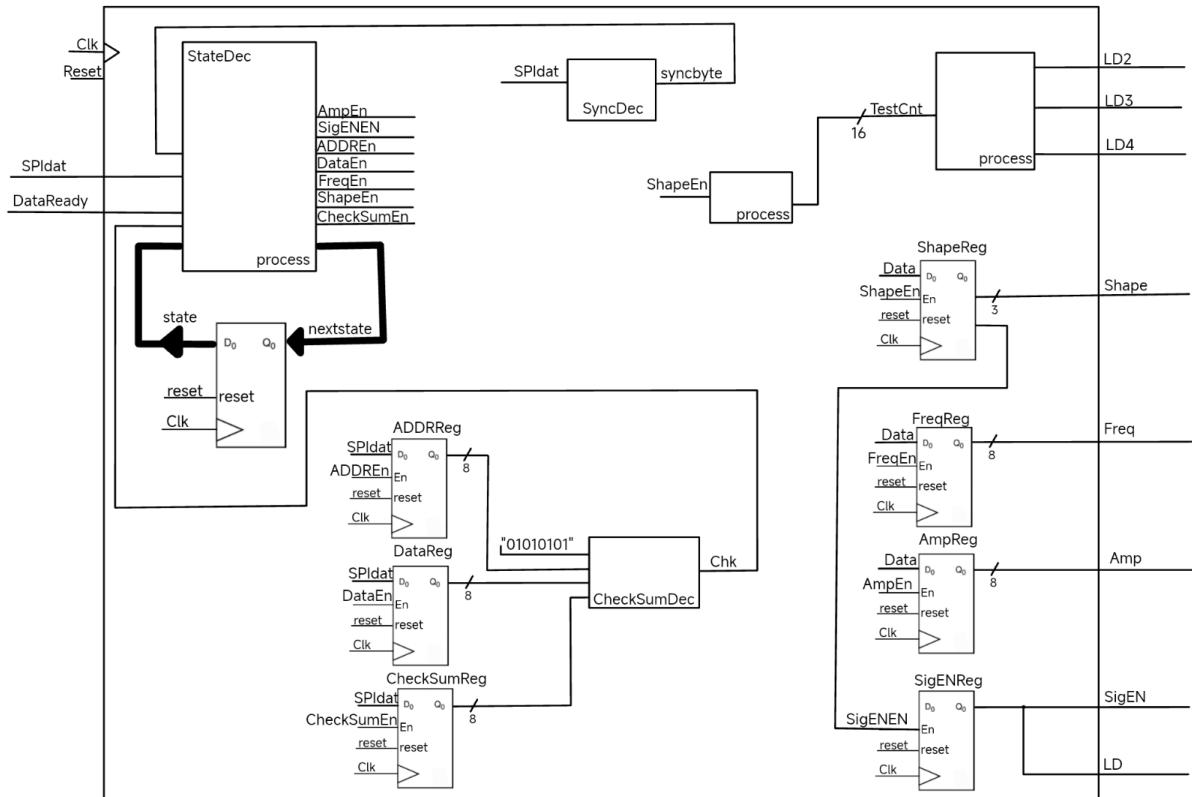


Figure 13: Funktionsdiagram af ProtokolBlok

Ovenstående diagram viser den logiske opbygning af Protokolblokken. Som tidligere nævnt skal den håndtere SPI-kommunikationen mellem den eksterne master enhed, og signalgeneratorens interne registre, som ses i diagrammet. Protokolblokken modtager data fra ”**SPIdat**” og bruger en synkron tilstandsmaskine, benævnt ”**StateDec**” på diagrammet til at styre protokollen. For at data modtagelse sættes i gang, skal der modtages en korrekt prædefineret ”**syncbyte**” som markerer starten på en dataoverførsel. Derefter gemmes dataen henholdsvis først i et adresseregister ”**ADDRReg**” og et dataregister ”**DataReg**”. Derudover bruges et register ”**CheckSumReg**” også til at gemme masterens Checksum. Ud fra den modtagne sync-byte, adresse-byte og data-byte, udregnes en checksum i signalgeneratoren også. Denne sammenlignes med masterens checksum inde i processen benævnt ”**CheckSumDec**”. Processeher vil danne signalet ”**Chk**” som bruges i tilstandsmaskinen. Kort sagt vil **Chk** blive sat højt når de 2 Checksum-værdier matcher, hvilket vil betyde at masteren og slaven har gennemført en korrekt kommunikation. Tilstandsmaskinen vil ud fra adresseregisteret, sende det relevante enable signal til det register som data-registeret skal overføre sin data til.

## 4 Test

### 4.1 Analyse af kravopfyldelse – MCU del

#### Samplerate: Maksimal frekvens

Et vigtigt krav til systemet er, at det skal kunne understøtte høje samplerater, op til 10.000 samples per sekund (sps), uden at miste data eller opleve ustabilitet i overførslen til LabVIEW.

For at teste dette blev `record_length` sat til 100 og `sample_rate` konfigureret til 10.000 sps i LabVIEW. MCU'en modtog konfigurationen korrekt og rapporterede følgende:

```
==== Oscilloscope Configuration ====
Record Length: 100 samples
Max Sample Rate: 10228.04 sps
```

Derefter blev signalet visualiseret i LabVIEW med disse parametre. Kurven fremstår komplet, stabil og kontinuerlig med en jævn fordeling af 100 samples. Det indikerer, at alle samples blev modtaget og behandlet korrekt, uden overbelastning af UART-kommunikationen:

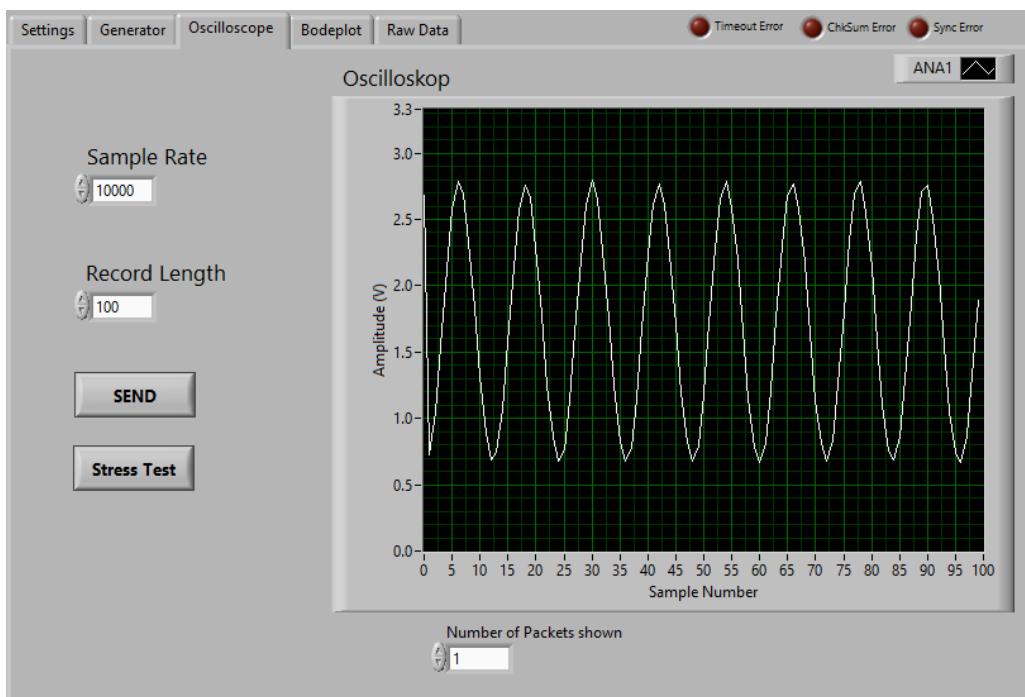


Figure 14: Oscilloskopvisning ved 10.000 sps og 100 samples per pakke, ved 9.4 kHz

Testen viser, at systemet lever op til kravet om at kunne måle og transmittere ved meget høje samplerater. Timer1 blev konfigureret automatisk til at matche den maksimale tilladte sample rate, beregnet ud fra UARTens baudrate. Denne værdi findes ud fra sammenhængen vist i udtryk 2.2.7, hvor systemet dynamisk justerer TOP-værdien, så sampling foregår præcis ved grænsen af, hvad UART'ens Baudrate tillader.

### Record Length: Maksimal pakkestørrelse

Oscilloskopet understøtter datapakker med op til 1000 samples, hvilket udgør den maksimale `record_length` tilladt af systemet. Denne grænse er valgt for at balancere mellem opløsning og overførselstid samt for at undgå buffer-overløb på både sender og modtager. Testen blev udført med et lavfrekvent signal (24 Hz) og en `record_length` på 1000 samples. Dette tilledt en detaljeret visualisering af hele signalperioden i én pakke.

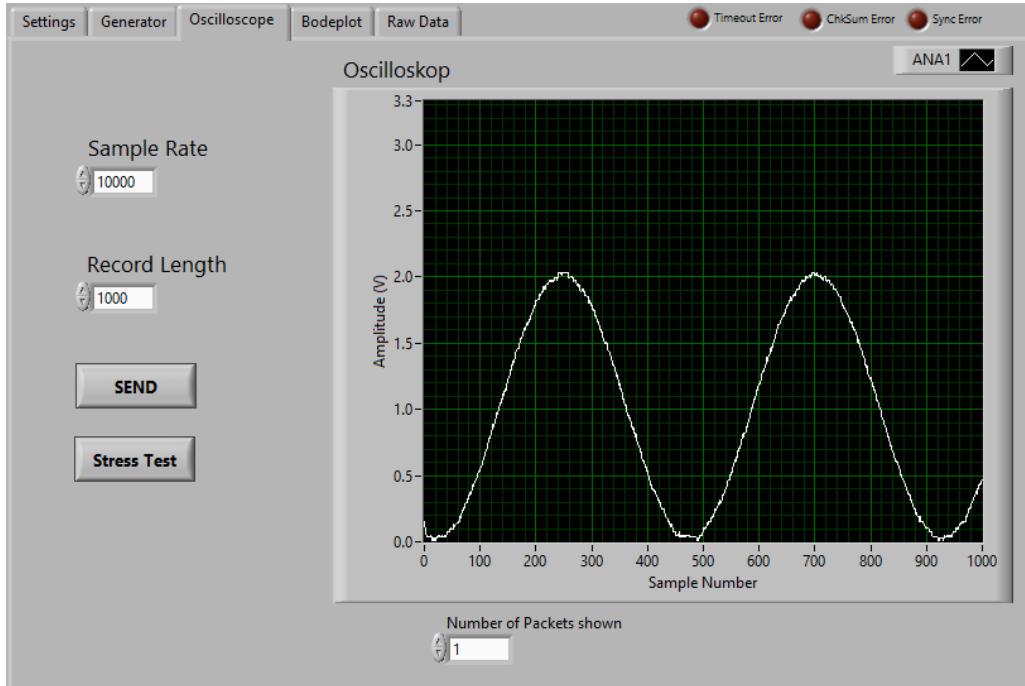


Figure 15: Oscilloskopvisning i LabVIEW med 1000 samples i én datapakke

Resultatet demonstrerer, at systemet korrekt håndterer den maksimale pakke uden datatab eller visningsfejl. Triple-buffer systemet sikrer, at både sampling og afsendelse kan fortsætte uafbrudt, selv ved store pakkestørrelser.

## Ekstra funktionalitet: Trekantsignal

Som en del af de udvidelser, der blev undersøgt i projektet, blev en ny signalform, trekantsignal, implementeret i signalgeneratoren. LabVIEW-interfacet blev desuden udvidet, så brugeren kan vælge "Triangle" som signalform sammen med de oprindelige muligheder: konstant, firkant, savtak og sinus. Trekantsignalet aktiveres ved at sætte parameteren **Shape = 4**, som sendes fra MCU'en til FPGA'en via den eksisterende SPI-protokol. Signalet genereres herefter ved hjælp af en process i FPGA'en, der danner trekant-signalet.

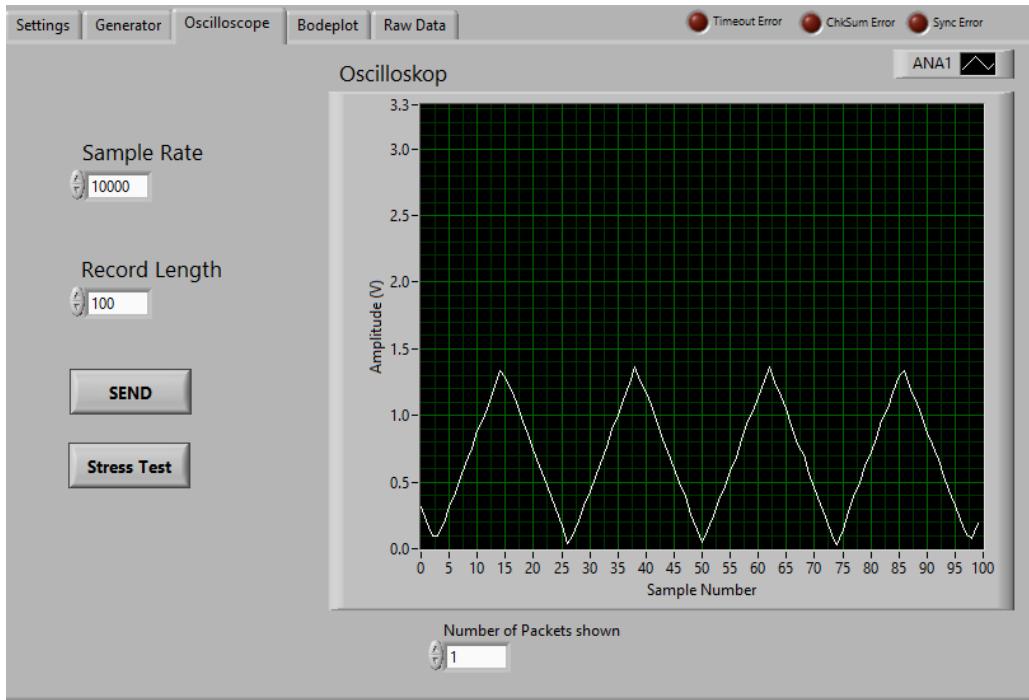
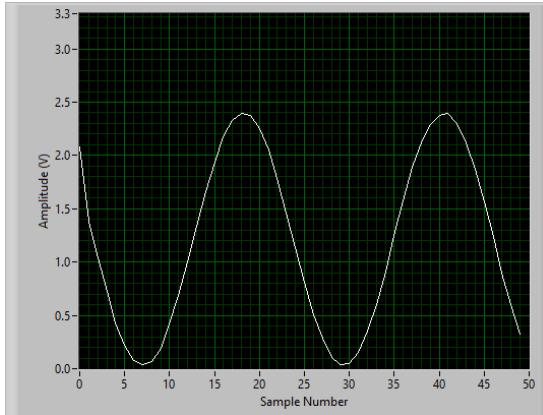


Figure 16: Trekant-signal genereret af signalgeneratoren, ved 432Hz og amplitude 1.3V

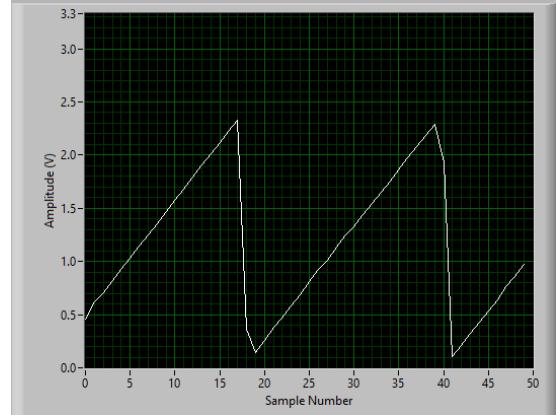
Målingen viser, at trekantsignalet blev genereret korrekt og transmitteret uden fejl, hvilket bekræfter at udvidelsen er implementeret og funktionel.

## 4.2 Visualisering af signalformer

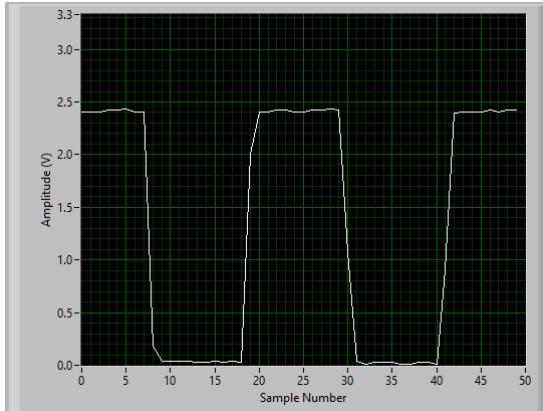
For at demonstrere de fem forskellige signalformer, som systemet kan generere, er der herunder indsat billeder fra LabVIEW, hvor det filtrerede signal vises i realtid. Alle signaler er målt med en sample rate på 10 kHz og **record length** = 50, med en amplitude svarende til ca. 2.4 V og en frekvens på 432 Hz.



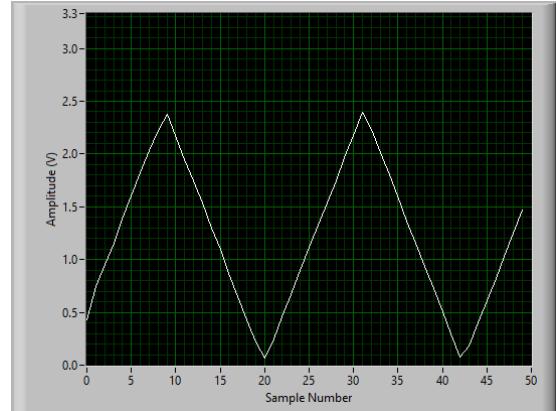
(a) Sinusformet signal



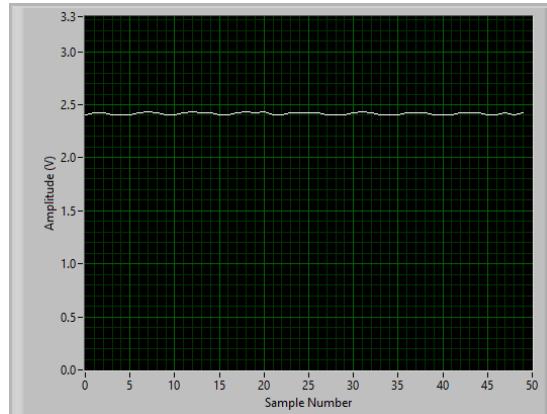
(b) Savtak



(c) Firkantsignal



(d) Trekantsignal



(e) Konstant DC-niveau

Figure 17: Visualisering af de fem mulige signalformer genereret af signalgeneratoren

## 4.3 Signalgenerator

### 4.3.1 Fysiske tests

#### Display Menu

Nedenfor kan vores menu, som man kan bladre igennem ved at trykke på BTN1, ses:



(a) Shape vist på display



(b) Amp vist på display



(c) Freq vist på display

Figure 18: Display Menu

## SPI - Test

Inden vi påbegyndte udarbejdelsen af protokolblokken, udførte vi en test for at verificere, at SPI-kommunikationen fungerede korrekt. Formålet var at sikre, at en byte kunne modtages via SPI uden fejl.

Testen blev udført ved at opsætte et skifteredister, som ved hver opadgående flanke på **SClk**, right-shifter værdien fra **MOSI** ind i registeret. Når **SSnot** igen trækkes høj, holdes værdien i skifteredistret, og indholdet vises direkte på **LED7** til **LED0**.

Nedenfor ses en visualisering af testen, hvor vi har overført bitmønstret "01010101" fra MCU'en til FPGA'en. Da dette mønster vises korrekt på LED'erne, kan vi konkludere, at SPI-forbindelsen mellem de to enheder fungerer som forventet.

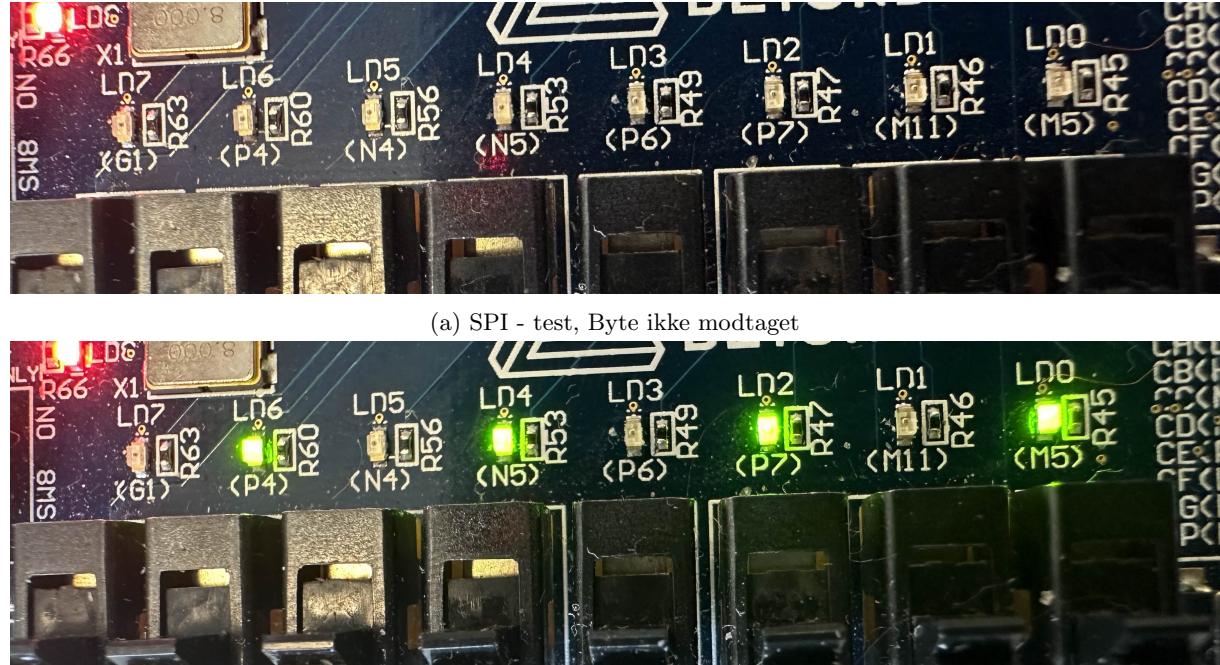


Figure 19: SPI-test på FPGA: visning af LED før og efter korrekt modtagelse

### **Stress test**

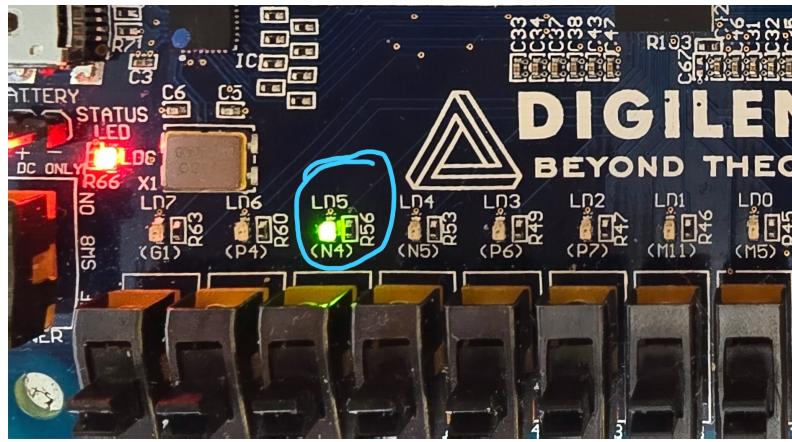
Vi har lavet en stresstest af SPI-kommunikationen. Denne er implementeret via en tæller (**TestCnt**), som inkrementeres hver gang FSM'en når ShapeS-tilstanden. For at FSM'en kan nå denne tilstand, skal checksummen være korrekt, hvilket betyder, at den forudgående SPI-overførsel er sket uden fejl.

Stresstesten aktiveres fra brugergrænsefladen i LabVIEW ved at trykke på knappen **Start Stress Test** i Oscilloscope-tabben. Dette sender en datapakke med type 0x04 via UART1 til MCU'en. Når denne pakke modtages, sættes et internt flag (**run\_stress\_test\_flag**), som aktiverer en sekvens i main-loop'en. Herfra udsender MCU'en præcis **10.827 SPI-kommandoer** til FPGA'en, hver struktureret som en gyldig instruktionspakke med korrekt checksums. På denne måde fungerer MCU'en som testgenerator for FPGA'ens modtagerlogik.

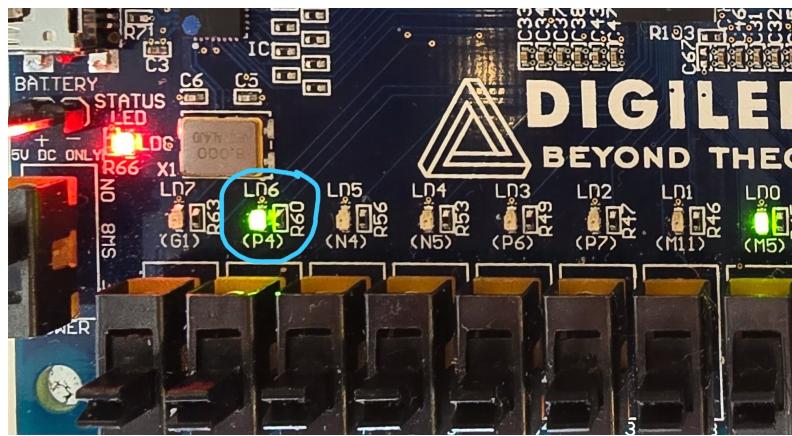
En separat proces i FPGA'en overvåger tælleren (**TestCnt**) og styrer tre LED'er baseret på dens værdi:

- Hvis tælleren er under 10.827, tændes en LED (indikator for få pakker)
- Hvis tælleren er lig med 10.827, tændes en anden LED (indikator for korrekt antal pakker)
- Hvis tælleren er over 10.827, tændes en tredje LED (indikator for mange pakker)

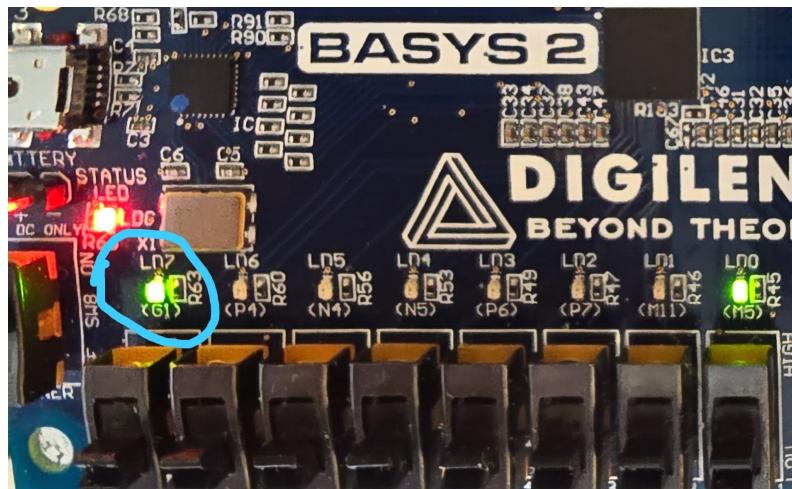
Visualisering af stress test kan ses på næste side:



(a) Under 10.827 instruktionssæt



(b) Præcis 10.827 instruktionssæt



(c) Over 10.827 instruktionssæt

Figure 20: Stress test med forskellige mængder instruktionssæt

#### 4.3.2 Simuleringer

##### Skiftered og Timingkomponent

Timingdiagrammet illustrerer skifteredregistrets funktion. Det ses, at data bliver right-shiftet ind via **MOSI** ved hver clockflanke, mens **SSnot** er lav. Når **SSnot** efterfølgende går høj, sættes **DataReady**-signalet høj i én clock-periode. Dette indikerer, at en komplet byte er modtaget og klar til videre behandling.

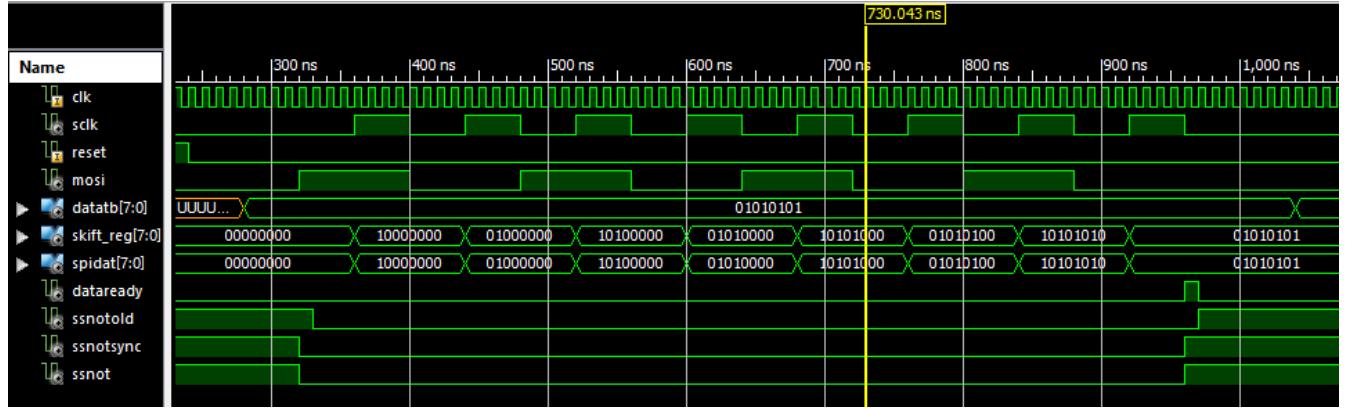


Figure 21: Simulation af Skiftered og Timingkomponent

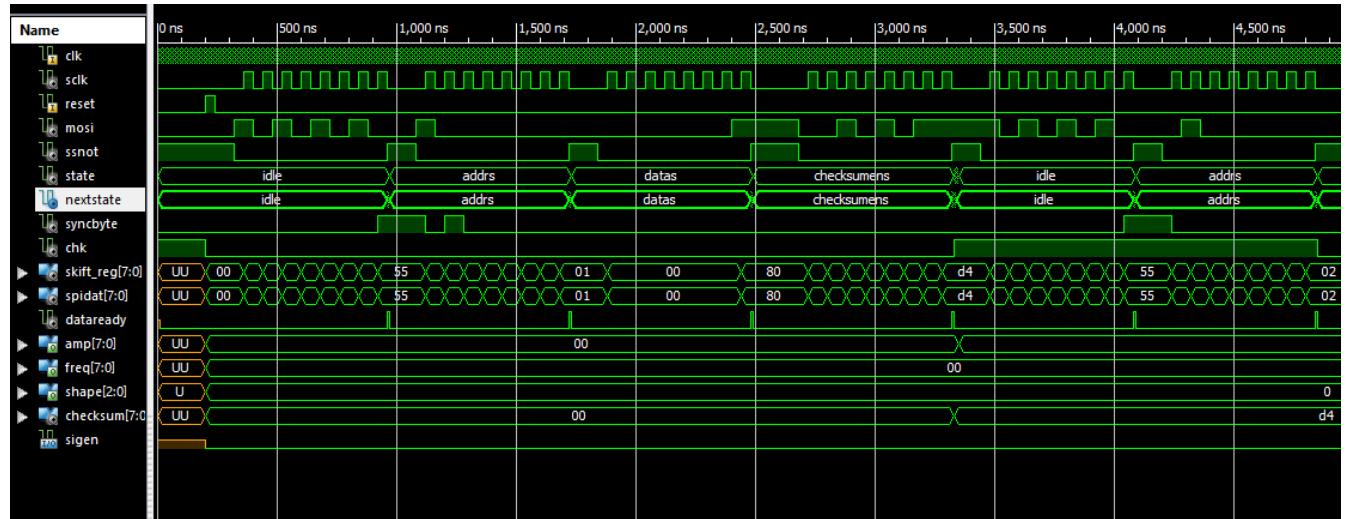
## Protokolblok Sim

Timingdiagrammet illustrerer **Protokolblokkens** funktion. Det ses tydeligt, at når **SSnot** bliver trukket lav, igangsættes en række processer, som resulterer i, at der læses én byte ad gangen. Dette gentages fire gange for at fuldende et datasæt, dvs. enten opdaterer Freq, Shape eller Amp.

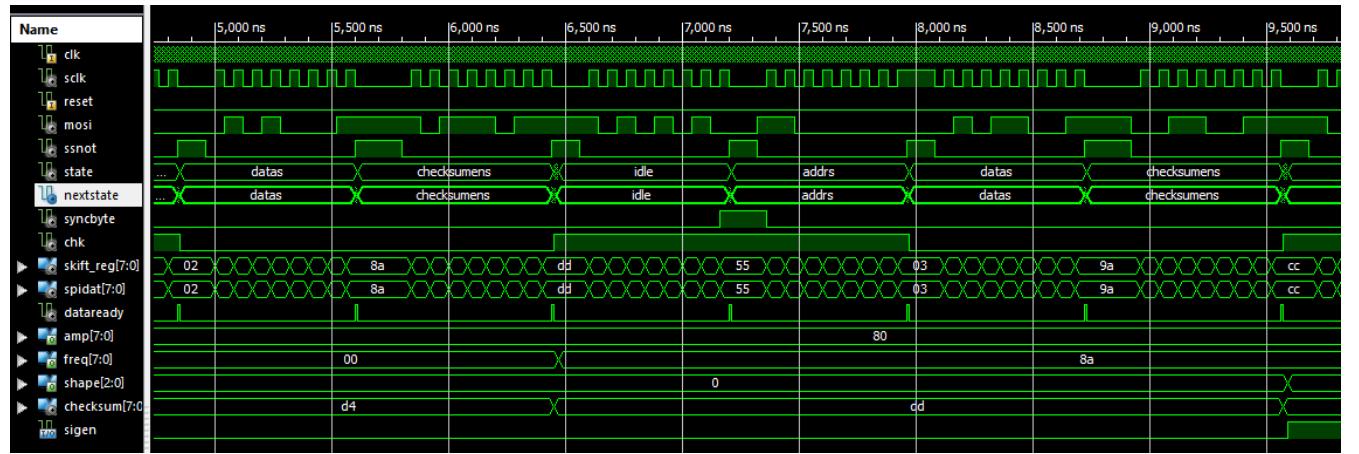
De fire bytes er: **Syncbyte**, **Adressebyte**, **Databyte** og til sidst **Checksum**. Hvis **Syncbyte** modtages korrekt, sættes signalet **syncbyte = '1'**. Tilsvarende, hvis den beregnede **Checksum** stemmer overens med den modtagne, sættes **chk = '1'**.

Når **syncbyte** sættes, forventer **Protokolblokken** kommunikation, og når **chk** sættes, ved vi, at der er modtaget en komplet og gyldig instruktion.

Som det fremgår af diagrammet, følges denne sekvens. Når den er afsluttet, indlæses **Databyte** skiftevis i **Amp**, **Freq** og **Shape**, dette giver os i alt 12 bytes, hvorefter **SigEN** sættes høj for at aktivere signalgeneratoren. Dette signal forbliver højt, indtil der igen modtages et syncbyte. Dette sikrer at den ønskede data bliver sendt videre, til der ønskes at opdatere en eller flere parametre.



(a) Simulation af Protokolblok del 1



(b) Simulation af Protokolblok del 2

Figure 22: Simulation af Protokolblok i to dele

## Konklusion

Projektet har resulteret i et fuldt fungerende oscilloskopsystem bestående af en signalgenerator implementeret på FPGA og et oscilloskop bygget op omkring en ATmega2560-mikrocontroller. Gennem en veldesignet SPI-protokol sikres stabil og fejlfri kommunikation mellem de to enheder, hvilket blev bekræftet via stresstest.

Systemet muliggør reeltidsopsamling og overførsel af signaldatal til LabVIEW, hvor en brugervenlig grænseflade giver kontrol over parametre som record length og signalform. Sample rate fastsættes automatisk ud fra record length og baudrate for at sikre optimal drift uden datatab, hvilket betyder, at brugeren ikke selv har fuld bestemmelse over samplingfrekvensen. Kravet om justérbar sample rate og test ved laveste frekvens (f.eks. 10 sps) er dermed ikke opfyldt i denne version, men har været muligt i tidlige versioner. Valget er truffet for at sikre maksimal robusthed og performance i den endelige løsning.

Lavpasfilterets funktion er verificeret eksperimentelt og sikrer, at PWM-signalen fra FPGA'en konverteres til en ren analog spænding, som ADC'en i mikrokontrolleren kan måle pålideligt. Filteret tillader tydelig visualisering af genererede signalformer, herunder en nyimplementeret trekantsignalform, som blev tilføjet som en udvidelse af signalgeneratorens funktionalitet.

Kravene til både maks. sample rate og maks. record length er opfyldt med succes, og brugen af triple buffering og automatisk baudrate-justering har gjort systemet robust og skalerbart. Samlet set opfylder løsningen langt de fleste krav i kravspecifikationen.

## 4.4 Gennemgang af kravspisifikationerne

	Krav	Oscilloskop	Verifikation
✓	ADC konvertering	Den analoge spænding fra signalgeneratoren skal måles fra 0 til 3.3 V med en oplosning på 8 bit.	Info*
✓	Dataintegritet	Oscilloskopet skal ved alle nedenstående indstillinger kunne køre med kontinueret ubrugte målinger.	Test* og Analyse*
✓	Parametre	Oscilloskopets samplerate og Record length skal kunne indstilles fra LabVIEW programmet.	Test*
X	Samplerate min	Oscilloskopet skal kunne køre ned til 10 sps.	Test*
✓	Samplerate max	Oscilloskopet skal kunne køre op til 5.000 sps.	Test* og Analyse
✓	Record length min	Oscilloskopet skal kunne køre med ned til 10 ADC-målinger i hver pakke. Den minimale tilladelige record length skal tage højde for sampleraten.	Test* og Analyse
✓	Record length max	Oscilloskopet skal kunne køre op til 1000 ADC-målinger i hver pakke for alle samplerater.	Test*/Analyse*
✓	RS-232 baudrate	RS232-forbindelsen skal kunne køre med en baudrate på mindst 115,2 kbaud.	Info*
✓	RS-232 håndtering	Modtagelse af data fra LabVIEW-programmet skal foregå via interrupt. Transmission kan foregå ved polling eller interrupt.	Info*
<b>Signalgenerator</b>			
✓	PWM filter	Der skal designes et lavpasfilter, der passende udglatter de digitale PWM-pulser.	Test*/Analyse*
✓	Parametre	Signalgeneratorens signalform (SHAPE), amplitude (AMPL) og frekvens (FREQ) skal indstilles fra LabVIEW-programmet. SHAPE, AMPL og FREQ kan gøres synlige på syvsegment-display.	Test*
✓	Sinus signal	Der kan implementeres en look-up tabel i VHDL koden der gør det muligt at signalgeneratoren kan lave et sinus-formet signal	Test
✓	SPI baudrate	SPI-forbindelsen skal køre med en baudrate på 500 kbaud eller mere.	Info*
X	SPI håndtering	To-vejs SPI-kommunikation kan implementeres, f.eks. med acknowledge handshake.	Info
✓	SPI protokol	Der skal vælges og implementeres en robust protokol til at overføre SHAPE, AMPL og FREQ.	Analyse*
✓	SPI test	Der skal ved test demonstreres en sikker forbindelse ved modtagelse. Testen kan udføres som et separat projekt.	Test*

Table 2: Oversigt over krav, verifikation og opfyldelse. Både nuværende og tidligere versioner

## 5 Bilag

### Appendix A - Globals og funktioner fra source koden

UART1\_comm.c

Globals	Beskrivelse
# define MAX_RECORD_LENGTH 1000	Definere den maksimale størrelse af record length
extern volatile uint16_t record_length;	Giver adgang til variablen record_length fra main
extern volatile uint16_t current_timer1_top;	Giver adgang til variablen current_timer1_top fra main
static uint8_t selected_param = 0;	Til at holde styr på hvilken knap er trykket i LabVIEW
volatile uint8_t shape = 0;	Til at gemme shape, 0 er default
volatile uint8_t amplitude = 128;	Til at gemme amplitude, 128 er default
volatile uint8_t frequency = 100;	Til at gemme frequency, 100 er default
volatile uint8_t run_stop_flag = 0;	Flag til at skifte mellem RUN og STOP tilstande
float sample_rate = 0;	Til at gemme sample-rate for ADC, 0 er default
#define UART1_RX_BUFFER_SIZE 128	Definere bufferstørrelsen for modtaget byte fra UART1
volatile uint8_t uart1_rx_buffer[UART1_RX_BUFFER_SIZE];	uint8_t array til at holde byte modtaget fra UART1
volatile uint8_t uart1_rx_index = 0;	Benyttes i funktion til at modtage byte fra UART1
volatile uint8_t uart1_packet_ready = 0;	Flag der sættes når en hel pakke er modtaget fra UART1
Funktioner	
ISR(USART1_RX_vect)	Service rutine som gemmer byte fra UART1 i bufferen
void send_oscilloscope_packet(uint8_t *samples, uint16_t length)	Sender data fra ADC'en til LabVIEW via rette protokol
void send_generator_packet(uint8_t active, uint8_t shape, uint8_t ampl, uint8_t freq)	Opdaterer LabVIEW gui
void parse_uart1_packet()	analysere pakker modtaget fra LabVIEW og udfører funktioner alt efter pakkens indhold

Table 3: UART1\_comm.c funktioner og Globals

## SPI\_comm.c

Globals	Beskrivelse
<b>Funktioner</b>	
<code>void spi_stess_test_10000_packets()</code>	Genstarter FPGA for at cleare registre, derefter sendes 10000 byte
<code>void transmit_singalgenerator_data(uint16_t amp, uint8_t freq, uint8_t shape)</code>	Sender 3 datapakker med amplitude, frekvens og shape til FPGA

Table 4: SPI\_comm.c funktioner og Globals

## HAL moduler

### ADC.c

Globals	Beskrivelse
<b>Funktioner</b>	
<code>void init_ADC_kanal0()</code>	initialisere ADC0 i MCU'en
<code>void init_timer1(int top_value)</code>	initialisere timer1 i MCU'en
<code>ISR(ADC_vect)</code>	har ingen funktion

Table 5: ADC.c funktioner og Globals

### SPI.c

Globals	Beskrivelse
<b>Funktioner</b>	
<code>void master_transmit(uint8_t data)</code>	Sender byte som master via SPI
<code>void master_init()</code>	Initialisere MCU'en som master
<code>unsigned char slave_reciver(unsigned char data)</code>	Bruges ikke i programmet
<code>void slave_init()</code>	Bruges ikke i programmet

Table 6: SPI.c funktioner og Globals

### uart.c

Globals	Beskrivelse
<b>Funktioner</b>	
<code>void uart_init(unsigned int ubrr)</code>	Initialisere uart0 i MCU'en
<code>void uart_send(char data)</code>	Sender en byte via uart0
<code>void uart_send_string(const char *str)</code>	Sender en streng bestående af flere byte via uart0
<code>void uart1_init(unsigned int ubrr)</code>	Initialisere uart1 i MCU'en
<code>void uart1_send(char data)</code>	Sender en byte via uart1

Table 7: uart.c funktioner og Globals

## Appendix B — Beregning af lavpasfilter

### Teori: Sallen-Key lavpasfilter

Et Sallen-Key lavpasfilter er et aktivt analogt filter, som anvender en op-amp i ikke-inverterende konfiguration med tilbagekobling via modstande og kondensatorer. Fordelen ved dette filter er dets præcise kontrol over både grænsefrekvens og Q-værdi (dæmpningskarakteristik), hvilket gør det velegnet til signalbehandling.

### Formler for 2. ordens Sallen-Key lavpasfilter

- Cutoff-frekvens:

$$f_c = \frac{1}{2\pi\sqrt{R_1C_1R_2C_2}}$$

- Q-værdi:

$$Q = \frac{\sqrt{R_1C_1R_2C_2}}{C_2(R_1 + R_2)}$$

Hvis man vælger  $R_1 = R_2 = R$  og  $C_1 = C_2 = C$ , forenkles formlerne til:

$$f_c = \frac{1}{2\pi RC}, \quad Q = \frac{1}{2}$$

### Beregning: Butterworth 2. ordens filter $f_c = 16$ kHz og $f_c = 30$ kHz

Ved ønsket cutoff-frekvens  $f_c = 16$  kHz og kondensatorer  $C_1 = C_2 = 1$  nF:

$$R = \frac{1}{2\pi f_c C} = \frac{1}{2\pi \cdot 16000 \cdot 1 \cdot 10^{-9}} \approx 9947 \Omega$$

Ved ønsket cutoff-frekvens  $f_c = 30$  kHz og kondensatorer  $C_1 = C_2 = 1$  nF:

$$R = \frac{1}{2\pi f_c C} = \frac{1}{2\pi \cdot 30000 \cdot 1 \cdot 10^{-9}} \approx 5305 \Omega$$

### Udvalgte komponentkombinationer

- Filter 1:

- $R_1 = 5.305$  k $\Omega$ ,  $R_2 = 5.305$  k $\Omega$
- $C_1 = 1$  nF,  $C_2 = 1$  nF

- Filter 2:

- $R_1 = 9.947$  k $\Omega$ ,  $R_2 = 9.947$  k $\Omega$
- $C_1 = 1$  nF,  $C_2 = 1$  nF

## 5.1 VHDL-kode

### 5.1.1 SigGenTop

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity SigGenTop is
7     Port ( BTN3 : in std_logic;
8             BTN1 : in std_logic;
9                 Clk : in std_logic;
10            MOSI : in std_logic;
11            SClk : in std_logic;
12            SSnot : in std_logic;
13            An : out std_logic_vector(3 downto 0);
14            Cat : out std_logic_vector(7 downto 0);
15            LD : out std_logic;
16            LD2 : out std_logic;
17            LD3 : out std_logic;
18            LD4 : out std_logic;
19            PWMOut : inout std_logic);
20 end SigGenTop;
21
22 architecture Behavioral of SigGenTop is
23 signal DispClk, SigEN, BTN1db, LED, LED2, LED3, LED4: std_logic;
24 signal Disp: std_logic_vector(19 downto 0);
25 signal Amp, Freq : std_logic_vector(7 downto 0);
26 signal Shape : std_logic_vector(2 downto 0);
27
28 begin
29
30 U1: entity WORK.SigGenSPIControl
31     port map (CLK => Clk,
32               Reset => BTN3,
33               SClk => SClk,
34               MOSI => MOSI,
35               SSnot => SSnot,
36               Shape => Shape,
37               Amp => Amp,
38               Freq => Freq,
39               SigEN => SigEN,
40               LD2 => LED2,
41               LD3 => LED3,
42               LD4 => LED4,
43               LD => LED);
44
45
46 U2: entity WORK.SigGenDataPath generic map (PWMinc => "0000001")
47     port map(Reset => BTN3, Clk => Clk, Shape => Shape(2 downto 0), Ampl => Amp, Freq =>
48               Freq, SigEN => SigEN, PWMOut => PWMOut);
49
50
51 U3: entity WORK.SevenSeg5
52     port map(Reset => BTN3, Clk => DispClk, Data => Disp, An => An, Cat => Cat);
53
54
55 U4: entity WORK.DivClk
56     port map(Reset => BTN3, Clk => Clk, TimeP => 50e3, Clk1 => DispClk);
57
58
59 LD <= LED; --Run signal
60 LD2 <= LED2;
61 LD3 <= LED3;
62 LD4 <= LED4;
63 U5: entity WORK.DispMux
64     port map(
65               Shape => Shape,
66               Amp => Amp,
67               Freq => Freq,
68               Clk => Clk,
```

```
69      Reset => BTN3,
70      BTN1 => BTN1db,
71      Disp => Disp);
72
73
74 U6: entity WORK.BTNdb
75     port map(
76         Reset => BTN3,
77         Clk => Clk,
78         BTNin => BTN1,
79         BTNout => BTN1db
80     );
81
82
83
84
85 end Behavioral;
```

Listing 1: SigGenTop

### 5.1.2 SigGenSPIControl

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity SigGenSPIControl is
5     Port ( CLK : in STD_LOGIC;
6             Reset : in STD_LOGIC;
7             SClk : in STD_LOGIC;
8             MOSI : in STD_LOGIC;
9             SSnot : in STD_LOGIC;
10            Shape : out STD_LOGIC_VECTOR (2 downto 0);
11            Amp : out STD_LOGIC_VECTOR (7 downto 0);
12            Freq : out STD_LOGIC_VECTOR (7 downto 0);
13            LD2: out STD_LOGIC;
14            LD3: out STD_LOGIC;
15            LD4: out STD_LOGIC;
16            LD: out STD_LOGIC;
17            SigEN : inout STD_LOGIC);
18            -- inout da den bruges som et internt signal ogsaa
19
20 end SigGenSPIControl;
21
22 architecture Behavioral of SigGenSPIControl is
23
24 signal SPIdat: STD_LOGIC_VECTOR (7 downto 0);
25 signal DataReady: STD_LOGIC;
26
27
28 begin
29
30    -- Clock Divider to generate display-friendly clock
31    ProtokolBlok: entity work.ProtokolBlok
32        port map (
33            Reset      => Reset,
34            Clk       => Clk,
35            Amp       => Amp,
36            Freq      => Freq,
37            Shape     => Shape,
38            DataReady => DataReady,
39            SPIdat   => SPIdat,
40            SigEN    => SigEN,
41            LD2      => LD2,
42            LD3      => LD3,
43            LD4      => LD4,
44            LD       => LD
45        );
46
47    --7-Segment Display Controller
48    SkifteReg: entity work.Skifte_reg_til_Parallel
49        port map (
50            Reset => Reset,
51            SPIdat => SPIdat,
52            MOSI  => MOSI,
53            SClk  => SClk
54        );
55
56    TimingComponent: entity work.TimingComponent
57        port map (
58            Reset => Reset,
59            Clk  => Clk,
60            DataReady => DataReady,
61            SSnot => SSnot
62        );
63
64 end Behavioral;

```

Listing 2: SigGenSPIControl

### 5.1.3 ProtokolBlok

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5
6
7 entity ProtokolBlok is
8     Port ( CLK : in STD_LOGIC;
9             Reset : in STD_LOGIC;
10            DataReady : in STD_LOGIC;
11            SPIdat: in STD_LOGIC_VECTOR (7 downto 0);
12            Shape: out STD_LOGIC_VECTOR (2 downto 0);
13            Amp: out STD_LOGIC_VECTOR (7 downto 0);
14            LD: out STD_LOGIC;
15            LD2: out STD_LOGIC;
16            LD3: out STD_LOGIC;
17            LD4: out STD_LOGIC;
18            Freq: out STD_LOGIC_VECTOR (7 downto 0);
19            SigEN: inout STD_LOGIC);
20 end ProtokolBlok;
21
22 architecture Behavioral of ProtokolBlok is
23
24 type Statetype is (IDLE, ADDRS, DataS, CheckSumEnS, CheckSumS, AmpS, ShapeS, FreqS);
25
26 signal state, nextstate : Statetype;
27 signal DataEN, ADDREN, AmpEN, FreqEN, ShapeEN, ShapeEN_prev, CheckSumEN, Chk, syncbyte,
   SigENEN, UnderFlag, OverFlag, LigeFlag: STD_LOGIC;
28 signal CheckSum, ADDR, Data: STD_LOGIC_VECTOR (7 downto 0);
29 signal TestCnt: STD_LOGIC_VECTOR(15 downto 0):= "0000000000000000";
30 -- Dette er til test af robust kommunikation
31
32 begin
33
34 SyncDec: syncbyte <= '1' when SPIdat = "01010101" else '0';
35 --Byte som indikerer start af datapakke
36
37 CheckSumDec: Chk <= '1' when Checksum = ((01010101 xor ADDR) xor Data) else '0';
38 -- Kontrol signal som indikere at data er modtaget korrekt
39
40 Statereg: process(CLK, Reset)
41 begin
42     if Reset = '1' then
43         state <= IDLE; -- Reset state to IDLE
44         TestCnt <= "0000000000000000";
45     elsif CLK'event and CLK = '1' then
46         state <= nextstate; -- Update state on clock edge
47         if ShapeEN = '1' and ShapeEN_prev = '0' then
48             -- puls som sikrer vi taeller en op paa antal shapes modtaget korrekt
49             TestCnt <= TestCnt + "0000000000000001";
50             -- Taeller til robusthed
51         end if;
52         ShapeEN_prev <= ShapeEN;
53
54     end if;
55 end process;
56
57
58 -- Next state logic and register enable control
59 StateDec: process (DataReady, state, Chk, syncbyte, ADDR)
60 begin
61
62 -- Default values
63 DataEn <= '0';
64 ADDREN <= '0';
65 CheckSumEn <= '0';
66 AmpEn <= '0';
67 ShapeEn <= '0';
68 FreqEn <= '0';
69 SigENEN <= '0';
70

```

```

71 --State machine
72
73 case state is
74
75 when IDLE =>
76   if syncbyte = '1' and DataReady = '1' then
77     -- check spidat(sync) stemmer
78     SigENEN <='0';
79     nextstate <= ADDRS;
80   else
81     nextstate <= IDLE;
82   end if;
83
84 when ADDRS =>
85   if DataReady = '1' then
86     ADDREN <= '1';
87     nextstate <= DataS;
88   else
89     nextstate <= ADDRS;
90   end if;
91
92
93 when DataS =>
94   if DataReady = '1' then
95     DataEN <= '1';
96     nextstate <= CheckSumEnS;
97   else
98     nextstate <= DataS;
99   end if;
100
101 when CheckSumEnS =>
102   if DataReady = '1' then
103     CheckSumEn <= '1';
104     nextstate <= CheckSumS;
105   else
106     nextstate <= CheckSumEnS;
107   end if;
108
109 when CheckSumS => -- Styring af hvilken data, der skal indlaeses
110   if Chk = '1' and ADDR = "00000001" then
111     nextstate <= AmpS;
112
113   elsif Chk = '1' and ADDR = "00000010" then
114     nextstate <= FreqS;
115   elsif Chk = '1' and ADDR = "00000011" then
116     nextstate <= ShapeS;
117   else
118     nextstate <= IDLE;
119   end if;
120
121
122 when AmpS =>
123   AmpEN <= '1';
124   nextstate <= IDLE;
125
126
127 when FreqS =>
128   FreqEN <= '1';
129   nextstate <= IDLE;
130
131
132 when ShapeS =>
133   ShapeEN <= '1';
134   SigENEN <= '1'; -- Enable signal til Signalenable
135   nextstate <= IDLE;
136 end case;
137
138
139 end process;
140
141
142 Process(TestCnt) -- Kontrol af kommunikations robusthed.
143 begin

```

```

144 if TestCnt < "0010101001001011" then -- led lys hvis under 10827 shapes
145 UnderFlag <= '1';
146 OverFlag <= '0';
147 LigeFlag <= '0';
148 elsif TestCnt = "0010101001001011" then -- led lyser hvis over 10827 shapes
149 OverFlag <= '0';
150 UnderFlag <= '0';
151 LigeFlag <= '1';
152 elsif TestCnt > "0010101001001011" then -- led lyser hvis over 10827 shapes
153 OverFlag <= '1';
154 LigeFlag <= '0';
155 UnderFlag <= '0';
156 else
157     UnderFlag <= '0';
158     OverFlag <= '0';
159     LigeFlag <= '0';
160 end if;
161 end process;
162
163 LD <= SigEn;
164 LD2 <= UnderFlag;
165 LD3 <= OverFlag;
166 LD4 <= LigeFlag;
167 ADDRReg: entity work.std_8bit_reg
168 port map (
169     Reset => Reset,
170     Clk => Clk,
171     Enable => ADDREn,
172     Data_in => SPIdat,
173     Data_out => ADDR
174 );
175
176 DataReg: entity work.std_8bit_reg
177 port map (
178     Reset => Reset,
179     Clk => Clk,
180     Enable => DataEN,
181     Data_in => SPIdat,
182     Data_out => Data
183 );
184
185 CheckSumReg: entity work.std_8bit_reg
186 port map (
187     Reset => Reset,
188     Clk => Clk,
189     Enable => CheckSumEn,
190     Data_in => SPIdat,
191     Data_out => CheckSum
192 );
193
194
195
196 ShapeReg: entity work.std_3bit_reg
197 port map (
198     Reset => Reset,
199     Clk => Clk,
200     Enable => ShapeEn,
201     Data_in => Data (2 downto 0),
202     Data_out => Shape
203 );
204
205
206 AmpReg: entity work.std_8bit_reg
207 port map (
208     Reset => Reset,
209     Clk => Clk,
210     Enable => AmpEn,
211     Data_in => Data,
212     Data_out => Amp
213 );
214
215
216 FreqReg: entity work.std_8bit_reg

```

```

217  port map (
218      Reset => Reset,
219      Clk => Clk,
220      Enable => FreqEn,
221      Data_in => Data,
222      Data_out => Freq
223  );
224
225 SigENReg : entity work.std_1bit_reg
226  port map (
227      Reset => Reset,
228      Clk => Clk,
229      Enable => SigENEN,
230      Data_in => SigENEN,
231      Data_out => SigEN
232  );
233
234 end Behavioral;

```

Listing 3: ProtokolBlok

#### 5.1.4 std\_8bit\_reg

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity std_8bit_reg is
5     Port ( reset : in STD_LOGIC;
6             clk : in STD_LOGIC;
7             enable : in STD_LOGIC;
8             data_in : in STD_LOGIC_VECTOR (7 downto 0);
9             data_out : out STD_LOGIC_VECTOR (7 downto 0));
10 end std_8bit_reg;
11
12 architecture Behavioral of std_8bit_reg is
13 begin
14     process (reset, clk)
15     begin
16         if reset = '1' then
17             data_out <= (others => '0'); -- Reset the register to 0
18         elsif rising_edge(clk) then
19             if enable = '1' then
20                 data_out <= data_in; -- Load data_in into data_out on clock edge
21             end if;
22         end if;
23     end process;
24 end Behavioral;

```

Listing 4: std\_8bit\_reg

### 5.1.5 SkifteReg

```
1 use IEEE.STD_LOGIC_1164.ALL;
2 use IEEE.STD_LOGIC_unsigned.all;
3
4
5 entity Skifte_reg_til_Parallel is
6     Port ( MOSI : in STD_LOGIC;
7             RESET : in STD_LOGIC;
8             SPIdat : out STD_LOGIC_VECTOR (7 downto 0);
9             SClk : in STD_LOGIC);
10
11 end Skifte_reg_til_Parallel;
12
13 architecture Behavioral of Skifte_reg_til_Parallel is
14     signal skift_reg : std_logic_vector (7 downto 0); -- Register hvor MOSI data gemmes.
15
16 begin
17
18     process(SClk, RESET)
19     begin
20         if RESET = '1' then
21             skift_reg <= "00000000"; -- Register tommes ved reset.
22         elsif rising_edge(SClk) then
23             skift_reg <= MOSI & skift_reg(7 downto 1);
24             -- Skifte register, laeser MOSI bit.
25         end if;
26     end process;
27
28     SPIdat <= skift_reg; -- Parallel indl    sing mellem skifteregister og SPIdat
29
30
31 end Behavioral;
```

Listing 5: SkifteReg

### 5.1.6 TimingComponent

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 -- Component som synkroniserer Masterenhedens SSnot signal med den interne
4 -- clock i FPGA-enheden
5 entity TimingComponent is
6     Port ( SSNot: in STD_LOGIC;
7             Clk : in STD_LOGIC;
8             Reset : in STD_LOGIC;
9             DataReady: out STD_LOGIC);
10 end TimingComponent;
11
12 architecture Behavioral of TimingComponent is
13
14 signal SSnotOld: STD_LOGIC;
15 signal SSnotSync: STD_LOGIC;
16
17 begin
18
19
20 -- SyncReg
21 process(Reset, Clk)
22 begin
23     if reset = '1' then
24         SSnotSync <= '1';
25     elsif rising_edge(Clk) then
26         -- SSnot signal fra Master synkroniseres med intern clock i FPGA
27         SSnotSync <= SSnot;
28
29     end if;
30
31 end process;
32
33 -- DelReg
34 process(Reset, Clk)
35 begin
36
37     if Reset = '1' then
38         SSnotOld <= '1'; -- Nulstiller tidligere SSnot værdi
39     elsif rising_edge(Clk) then
40         SSnotOld <= SSnotSync; -- Gemmer forrige SSNOT værdi
41
42     end if;
43
44 end process;
45
46
47 DataReady <= SSNOTSync and (not SSnotOld);
48     -- Flanke detektion - Dataready = 1, når SSNOT går fra 0 til 1
49
50 end Behavioral;
```

Listing 6: TimingComponent

### 5.1.7 SigGenDataPath

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5 -- Denne komponent staar for at generere vores PWM signal
6 entity SigGenDatapath is
7 generic( PWMinc : std_logic_vector(6 downto 0) := "0000001" );
8 -- sat markant ned
9 Port ( Reset : in std_logic;
10        Clk : in std_logic;
11        SigEN : in std_logic;
12        Shape : in std_logic_vector(2 downto 0);
13        Ampl : in std_logic_vector(7 downto 0);
14        Freq : in std_logic_vector(7 downto 0);
15        PWMOut : out std_logic);
16 end SigGenDatapath;
17
18 architecture Behavioral of SigGenDatapath is
19
20 signal SigCnt, nSigCnt, FreqCnt: std_logic_vector(11 downto 0);
21 signal Sig, SigSquare, SigSaw, SigSinus, SigTri : std_logic_vector(7 downto 0);
22 signal SigAmpl: std_logic_vector(6 downto 0);
23 signal PWMcnt: std_logic_vector(6 downto 0) := "0000000";
24 signal PWM, PWMwrap: std_logic;
25
26 begin
27
28 FreqDec: FreqCnt <= "00" & Freq(7 downto 6) & Freq(5 downto 4) & '0' & Freq(3 downto 2) &
29     '0' & Freq(1 downto 0);
30
31 FreqAdd: nSigCnt <= SigCnt + FreqCnt;
32
33 SigReg: process (Reset, PWMwrap, Clk)
34 begin
35   if Reset = '1' then SigCnt <= X"000";
36   elsif Clk'event and Clk = '1' then
37     if PWMwrap = '1' then
38       SigCnt <= nSigCnt;
39     end if;
40   end if;
41 end process;
42
43 SinusDec : entity WORK.SinusLUT PORT MAP (clka => Clk, addra => SigCnt, douta => SigSinus
44 );
45
45 PWMcount: process(Reset, Clk)
46 variable PWMcntvar: std_logic_vector(8 downto 0);
47 begin
48   if Reset = '1' then PWMcntvar := "000000000";
49   elsif Clk'event and Clk = '1' then
50     PWMcntvar := PWMcntvar + PWMinc;
51     if PWMcntvar > "11111111" then
52       --udviddet med 2 bit for at snke signalet
53       PWMcntvar := "000000000";
54     end if;
55   end if;
56   if PWMcntvar = "000000000" then
57     PWMwrap <= '1';
58   else
59     PWMwrap <= '0';
60   end if;
61   PWMcnt <= PWMcntvar(6 downto 0);
62   -- PWMcnt er kun 7 bit da den skal tælle hurtigere og følge clock direkte
63 end process;
64
65 SquareDec: SigSquare <= "00000000" when SigCnt < X"800" else "11111111";
66
67 SawDec: SigSaw <= SigCnt(11 downto 4);
68
69 Tridec: process(SigCnt)

```

```

70 begin
71   if Sigcnt < x"800" then
72     SigTri <= SigCnt (10 downto 3);
73   elsif SigCnt >= x"800" then
74     SigTri <= not SigCnt(10 downto 3);
75   end if;
76 end process;
77
78
79 SigMux: Sig <= X"FF" when Shape = "000" else
80           SigSquare when Shape = "001" else
81           SigSaw when Shape = "010" else
82           SigSinus when Shape = "011" else
83           SigTri when Shape = "100";
84
85
86 AmplDec: process(Ampl, Sig)
87 variable MulA, MulB, MulC: std_logic_vector(15 downto 0);
88 --variable MulC: std_logic_vector(6 downto 0);
89 begin
90   MulB := X"00" & Sig;
91   MulA := X"00" & Ampl;
92   MulC := X"0000";
93   for j in 0 to 15 loop
94     if MulA(j) = '1' then
95       MulC := MulC + MulB;
96     end if;
97     MulB := MulB(14 downto 0) & '0';
98   end loop;
99   -- MulC := MulC + 1;
100  SigAmpl <= MulC(15 downto 9);
101  -- SigAmpl <= "1111111";
102 end process;
103
104
105
106 PWMcomp: PWM <= '1' when PWMcnt <= SigAmpl else '0';
107
108 PWMon: PWMon <= PWM when SigEn = '1' else '0';
109
110 end Behavioral;

```

Listing 7: SigGenDataPath

### 5.1.8 SevenSeg5

```

1 ----- Driver to sevensegment display -----
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4 use IEEE.STD_LOGIC_ARITH.ALL;
5 use IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7 entity SevenSeg5 is
8     Port ( Reset,Clk: in std_logic;
9             Data : in std_logic_vector (19 downto 0); -- Binary data
10            cat : out std_logic_vector(7 downto 0); -- Common cathodes
11            an : out std_logic_vector(3 downto 0)); -- Common Anodes
12 end SevenSeg5;
13
14 architecture SevenSeg_arch of SevenSeg5 is
15 signal DispCount: integer range 0 to 3;
16 signal DataN: std_logic_vector (3 downto 0);
17 signal CatInt, CatData, CatSign: std_logic_vector (7 downto 0);
18 signal AnInt: std_logic_vector (3 downto 0);
19
20 begin
21
22
23 DispCountReg: process(Reset, Clk)
24 begin
25     if Reset = '1' then
26         DispCount <= 0;
27     elsif Clk'event and Clk = '1' then
28         if DispCount = 3
29             then DispCount <= 0;
30         else DispCount <= DispCount + 1; end if;
31     end if;
32 end process DispCountReg;
33
34 DispCountDec: process(DispCount, Data)
35 begin
36     case DispCount is
37         when 0 =>
38             AnInt <= "1110";          -- Display 1 activated
39             DataN <= Data(3 downto 0);
40         when 1 =>
41             AnInt <= "1101";          -- Display 1 activated
42             DataN <= Data(7 downto 4);
43         when 2 =>
44             AnInt <= "1011";          -- Display 1 activated
45             DataN <= Data(11 downto 8);
46         when others =>
47             AnInt <= "0111";          -- Display 1 activated
48             DataN <= Data(15 downto 12);
49     end case;
50 end process DispCountDec;
51
52 with DataN select -- Activate segments acc. to Data
53     CatData <= "11000000" when "0000", -- 0
54             "11111001" when "0001", -- 1
55             "10100100" when "0010", -- 2
56             "10110000" when "0011", -- 3
57             "10011001" when "0100", -- 4
58             "10010010" when "0101", -- 5
59             "10000010" when "0110", -- 6
60             "11111000" when "0111", -- 7
61             "10000000" when "1000", -- 8
62             "10011000" when "1001", -- 9
63             "10001000" when "1010", -- A
64             "10000011" when "1011", -- b
65             "11000110" when "1100", -- C
66             "10100001" when "1101", -- d
67             "10000110" when "1110", -- E
68             "10001110" when "1111", -- F
69             "11111111" when others; -- blank
70
71 with DataN select

```

```

72  CatSign <= "11111111" when "0000", -- Blank
73  "10101111" when "0001", -- "r"
74  "11100011" when "0010", -- "u"
75  "10101011" when "0011", -- "n"
76  "10011001" when "0100", -- 4
77  "10010010" when "0101", -- 5
78  "10000010" when "0110", -- 6
79  "11111000" when "0111", -- 7
80  "10000000" when "1000", -- 8
81  "10011000" when "1001", -- 9
82  "10001000" when "1010", -- A
83  "10000011" when "1011", -- b
84  "11000110" when "1100", -- C
85  "10100001" when "1101", -- d
86  "10000110" when "1110", -- E
87  "10001110" when "1111", -- F
88  "11111111" when others; -- blank
89
90 CatInt <= CatData when Data(DispCount+16) = '0' else CatSign;
91
92 process(Reset, Clk)
93 begin
94  if Reset = '1' then Cat <= "00000000"; An <= "0000";
95  elsif Clk'event and Clk = '1' then
96    Cat <= CatInt;
97    An <= AnInt;
98  end if;
99 end process;
100
101 end SevenSeg_arch;

```

Listing 8: SevenSeg5

### 5.1.9 DivClk

```

1 ----- Clock divider -----
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4 use IEEE.STD_LOGIC_ARITH.ALL;
5 use IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7 entity DivClk is
8     port ( Reset: in STD_LOGIC;      -- Global Reset (BTN1)
9             Clk: in STD_LOGIC;        -- Master Clock (50 MHz)
10            TimeP: in integer;      -- Time periode of the divided clock (50e6)
11            Clk1: out STD_LOGIC);   -- Divided clock1 (1 Hz)
12 end DivClk;
13
14 architecture DivClk_arch of DivClk is
15 --constant MaxCnt1: integer:= 14;
16 signal Cnt1: integer range 0 to 25000000;  -- 24 bit counter
17 signal Clear1: STD_LOGIC;
18 signal Clk1_D: STD_LOGIC;
19
20 begin
21
22     -- T-register with enable and async.reset
23     Div1Reg: process(clk,Reset)
24     begin
25         if Reset = '1' then Clk1_D <= '0';          -- async. reset
26         elsif (clk'event and clk ='1') then
27             if Clear1= '1' then                      -- enable
28                 Clk1_D <= not Clk1_D;
29             end if;
30         end if;
31     end process Div1Reg;
32
33     Div1Dec: process(Cnt1, TimeP) -- Kombinatorisk
34     begin
35         Clear1 <= '0';
36         if Cnt1 = TimeP/2 then
37             Clear1 <= '1';
38         end if;
39     end process Div1Dec;
40
41     -- 24 bit up-counter with clear and async. reset
42     Div1Cnt:process(clk,Reset)
43     begin
44         if Reset = '1' then Cnt1 <= 1;           -- async. reset
45         elsif (clk'event and clk ='1') then
46             if Clear1 = '1' then Cnt1 <= 1;    -- clear
47             else Cnt1 <= Cnt1 + 1; end if;
48         end if;
49     end process Div1Cnt;
50
51     Clk1 <= Clk1_D;
52
53 end DivClk_arch;

```

Listing 9: DivClk

### 5.1.10 DispMux

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_unsigned.all;
4
5
6 entity DispMux is
7     Port ( Shape : in STD_LOGIC_VECTOR (2 downto 0);
8             Amp : in STD_LOGIC_VECTOR (7 downto 0);
9             Freq : in STD_LOGIC_VECTOR (7 downto 0);
10            Reset : in std_logic;
11            Clk : in std_logic;
12            BTN1 : in std_logic;
13            Disp : out STD_LOGIC_VECTOR (19 downto 0));
14 end DispMux;
15
16 architecture Behavioral of DispMux is
17 signal DispSel : std_logic_vector(1 downto 0);
18
19
20 begin
21
22 process(BTN1, Reset, Clk)
23 begin
24     if Reset = '1' then
25         DispSel <= "00";
26     elsif rising_edge(Clk) then
27         if BTN1 = '1' then
28             case DispSel is
29                 when "00" => DispSel <= "01";
30                 when "01" => DispSel <= "10";
31                 when "10" => DispSel <= "00";
32                 when others => DispSel <= "00";
33             end case;
34         end if;
35     end if;
36 end process;
37
38
39
40 process(DispSel, Shape, Amp, Freq)
41 begin
42
43     if DispSel = "00" then
44         Disp <= x"450" & "00000" & Shape; -- 20 bits i alt
45
46
47     elsif DispSel = "01" then
48
49         Disp <= x"4A0" & Amp;
50
51     elsif DispSel = "10" then
52
53         Disp <= x"4F0" & Freq;
54
55     end if;
56 end process;
57
58 end Behavioral;

```

Listing 10: DispMux

### 5.1.11 BTNdb

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6
7
8 entity BTNdb is
9  port( Reset, Clk: in std_logic;
10        BTNin: in std_logic;
11        BTNout: out std_logic);
12 end BTNdb;
13
14 architecture Behavioral of BTNdb is
15
16 constant CountMax: std_logic_vector(11 downto 0) := X"FOO";
17
18 type StateType is (BTNup, wBTNup, BTNdown, wBTNdown);
19 signal State, nState: StateType;
20 signal count: std_logic_vector(11 downto 0);
21 signal CountEN, CountClr: std_logic;
22
23 begin
24
25 StateReg: process (Reset, Clk)
26 begin
27  if Reset = '1' then State <= BTNup;
28  elsif Clk'event and Clk = '1' then
29   State <= nState;
30  end if;
31 end process;
32
33 StateDec: process (state, BTNin, Count)
34 begin
35  CountEN <= '0';
36  CountClr <= '0';
37  BTNout <= '0';
38  nState <= BTNup;
39  case state is
40  when BTNup =>
41   CountClr <= '1';
42   if BTNin = '1' then
43    nState <= wBTNup;
44   else
45    nState <= BTNup;
46   end if;
47
48  when wBTNup =>
49  CountEN <= '1';
50  if Count = CountMax then
51   BTNout <= '1';
52   nState <= BTNdown;
53  else
54   nState <= wBTNup;
55  end if;
56
57  when BTNdown =>
58  CountClr <= '1';
59  if BTNin = '0' then
60   nState <= wBTNdown;
61  else
62   nState <= BTNdown;
63  end if;
64
65  when wBTNdown =>
66  CountEN <= '1';
67  if Count = CountMax then
68   nState <= BTNup;
69  else
70   nState <= wBTNdown;
71  end if;
```

```

72      end case;
73  end process;
75
76 CountReg: process(Reset, Clk)
77 begin
78   if Reset = '1' then Count <= X"000";
79   elsif Clk'event and Clk = '1' then
80     if CountClr = '1' then
81       Count <= X"000";
82     elsif CountEN = '1' then
83       Count <= Count + '1';
84     end if;
85   end if;
86 end process;
87
88 end Behavioral;

```

Listing 11: BTNdb

### 5.1.12 siggentop.ucf

```

1 NET "PWMOut" LOC = C12;
2
3 NET "LD" LOC = M5;
4 NET "LD2" LOC = N4;
5 NET "LD3" LOC = G1;
6 NET "LD4" LOC = P4;
7
8 #BTN signals
9 NET "BTN3" LOC = A13;
10 NET "BTN1" LOC = C11;
11 #Assignment of CLK
12 NET "Clk" LOC = B8;
13
14 #Assignment of 7-seg cathode
15 NET "cat[7]" LOC = N13;
16 NET "cat[6]" LOC = M12;
17 NET "cat[5]" LOC = L13;
18 NET "cat[4]" LOC = P12;
19 NET "cat[3]" LOC = N11;
20 NET "cat[2]" LOC = N14;
21 NET "cat[1]" LOC = H12;
22 NET "cat[0]" LOC = L14;
23
24 #Assignment of anode
25 NET "an[0]" LOC = F12;
26 NET "an[1]" LOC = J12;
27 NET "an[2]" LOC = M13;
28 NET "an[3]" LOC = K14;
29
30
31 # Vores UCF
32 NET "MOSI" LOC = B2;
33 NET "SClk" LOC = A3;
34 NET "SClk" CLOCK_DEDICATED_ROUTE = FALSE;
35 NET "SSnot" LOC = J3;

```

Listing 12: siggentop.ucf

## 5.2 C-kode

### 5.2.1 main.c

```
1 // ## Program developed by Mads Rodulph and Sigurd Hestbech Christiansen ##
2 // ## Last edited: 20-6-2025 ##
3 // ## Throughout the code, page references are marked as S.xxx, referring to the
4 //      Atmega2560 datasheet. ##
5
6 #ifndef F_CPU
7 #define F_CPU 16000000UL // Define CPU frequency
8 #endif
9
10 #include <avr/io.h>           // Include I/O functions
11 #include <stdio.h>             // Include sprintf
12 #include <stdbool.h>           // Include boolean support
13 #include <avr/interrupt.h>     // Include interrupt handling
14 #include <stdint.h>             // Include fixed width integer types
15 #include <util/delay.h>         // Includes delay function
16 #include "ADC.h"                // Include ADC initialization functions
17 #include "SPI.h"                // Include SPI init functions
18 #include "UART.h"               // Includes uart init functions
19 #include "UART1_comm.h"          // Includes uart1 communication functions
20 #include "SPI_comm.h"            // Include SPI communication functions
21
22 // Declare parse_uart1_packet prototype
23 void parse_uart1_packet(void);
24
25 // Define maximum buffer size for dynamic record length
26 #define MAX_RECORD_LENGTH 1000
27 volatile uint8_t buffer_a[MAX_RECORD_LENGTH]; // triple buffer A
28 volatile uint8_t buffer_b[MAX_RECORD_LENGTH]; // triple buffer B
29 volatile uint8_t buffer_c[MAX_RECORD_LENGTH]; // triple buffer C
30 volatile uint8_t *active_buffer = buffer_a; // buffer we write into
31 volatile uint8_t *send_buffer = buffer_b; // buffer we send from
32 volatile uint8_t *standby_buffer = buffer_c; // next available buffer
33 volatile bool buffer_in_use = false; // indicates if buffer is locked for
34 // transmission
35 volatile uint16_t current_timer1_top = 200; // ADC sampling rate control (OCR1A value)
36 volatile uint16_t record_length = 100; // number of samples per transmission
37
38 // ADC control flags
39 volatile uint16_t sample_index = 0; // current sample index into active buffer
40 volatile bool buffer_ready = false; // signals that a full buffer is ready to send
41
42 // BTN/SW state
43 volatile uint8_t btn = 0;
44 volatile uint8_t sw = 0;
45 volatile uint8_t run_stress_test_flag = 0; // triggered by LabVIEW START (BTN3)
46
47 // Enumeration for program states
48 enum states
49 {
50     state_init,
51     state_Run,
52     state_Stop,
53     state_SPITest
54 };
55 static enum states state = state_init;
56
57 // Interrupt Service Routine (ISR) for TIMER1 Compare Match B (see ADC.c for more info)
58 ISR(TIMER1_COMPB_vect)
59 {
60     if (sample_index < record_length)
61     {
62         active_buffer[sample_index++] = ADCH;
63     }
64     else if (!buffer_in_use)
65     {
66         cli();
67         buffer_ready = true;
68         buffer_in_use = true;
69     }
70 }
```

```

68     // rotate buffers: standby      send      active
69     uint8_t *temp = (uint8_t *)standby_buffer;
70     standby_buffer = send_buffer;
71     send_buffer = active_buffer;
72     active_buffer = temp;
73
74     sample_index = 0;
75     sei();
76 }
77 // if buffer is still in use, skip storing until main clears it
78 }
79
80 int main(void)
81 {
82     while (1)
83     {
84         parse_uart1_packet(); // check for UART1 commands from LabVIEW
85
86         switch (state)
87         {
88             case state_init:
89
90                 init_ADC_kanal0();                                // configure ADC0 with auto-
91                 trigger via Timer1
92                 init_timer1(current_timer1_top);                // configure Timer1 for ADC
93                 sampling
94                 master_init();                                // initialize SPI as master
95                 uart_init(16);                               // init UART0 (115200 baud)
96                 uart1_init(16);                             // init UART1 (115200 baud, for
97                 LabVIEW)
98                 uart_send_string("System initialized.\r\n"); // confirm system boot
99                 DDRD |= (1 << PD7);                      // configure PD7 as output (FPGA
100                reset)
101                PORTD &= ~(1 << PD7);                     // ensure reset line is low
102
103            sei();           // enable global interrupts
104            state = state_Run; // move to Run state
105            break;
106
107            case state_Run:
108
109                if (run_stress_test_flag)
110                {
111                    run_stress_test_flag = 0;
112                    state = state_SPITest;
113                }
114
115                if (buffer_ready)
116                {
117                    // prevent race condition during buffer flag update
118                    cli();
119                    buffer_ready = false;
120                    buffer_in_use = false;
121                    sei();
122
123                    // send data to LabVIEW
124                    send_oscilloscope_packet((uint8_t *)send_buffer, record_length);
125                    _delay_ms(9); // wait for UART to complete before sending debug text
126
127                    uart_send_string("\rSample: ");
128                    char buf[16];
129                    sprintf(buf, "%02X      ", send_buffer[0]);
130                    uart_send_string(buf);
131                }
132
133                if (run_stop_flag)
134                {
135                    state = state_Stop;
136                }
137                break;
138
139            case state_Stop:
140                if (!(run_stop_flag))
141
```

```

137     {
138         state = state_Run; // resume sampling when unpause
139     }
140     break;
141
142     case state_SPITest:
143         spi_stress_test_10000_packets(); // run SPI communication stress test
144         state = state_Run;
145     break;
146
147     default:
148         break;
149 }
150
151 }
```

Listing 13: main.c

### 5.2.2 UART1.comm.c

```

1 // ## Program developed by Mads Rodulph and Sigurd Hestbech Christiansen ##
2 // ## Last edited: 21-6-2025 ##
3 // ## This file contains UART1 communication functions ##
4
5 #include <avr/io.h>
6 #include <avr/interrupt.h>
7 #include <string.h>
8 #include <util/delay.h>
9 #include <stdio.h>
10 #include "UART1_comm.h"
11 #include "UART.h"
12 #include "ADC.h"
13 #include "SPI.h"
14 #include "SPI_comm.h"
15
16 #define MAX_RECORD_LENGTH 1000           // max samples allowed per oscilloscope
17   packet
17 extern volatile uint16_t record_length;    // current active record length
18 extern volatile uint16_t current_timer1_top; // timer TOP value for sampling rate
19 static uint8_t selected_param = 0;          // user-selected parameter: 0 = shape, 1 =
19   amplitude, 2 = frequency
20
21 volatile uint8_t shape = 0;                // waveform shape: 0-3 (const, square, saw, sine)
22 volatile uint8_t amplitude = 128;          // waveform amplitude (0-255)
23 volatile uint8_t frequency = 100;           // waveform frequency (custom encoded)
24 volatile uint8_t run_stop_flag = 0;         // flag toggled by BTN2 (pause/resume)
25 float sample_rate = 0;                    // calculated sample rate in Hz
26
27 #define UART1_RX_BUFFER_SIZE 128
28 volatile uint8_t uart1_rx_buffer[UART1_RX_BUFFER_SIZE]; // circular RX buffer
29 volatile uint8_t uart1_rx_index = 0;           // current write index in RX
29   buffer
30 volatile uint8_t uart1_packet_ready = 0;       // set when complete packet is
30   received
31
32 // UART1 RX interrupt      assembles bytes into packet buffer and checks length field
33 ISR(USART1_RX_vect)
34 {
35     uint8_t byte = UDR1;
36
37     if (uart1_rx_index < UART1_RX_BUFFER_SIZE)
38     {
39         uart1_rx_buffer[uart1_rx_index++] = byte;
40
41         // once header is complete, check if full packet has arrived
42         if (uart1_rx_index >= 5)
43         {
44             uint16_t length = (uart1_rx_buffer[2] << 8) | uart1_rx_buffer[3];
45             if (uart1_rx_index == length)
46                 uart1_packet_ready = 1; // entire packet received
47         }
48     }
}
```

```

49     else
50     {
51         uart1_rx_index = 0; // reset buffer if overflow occurs
52     }
53 }
54
55 // Send a full oscilloscope data packet to LabVIEW
56 void send_oscilloscope_packet(uint8_t *samples, uint16_t length)
{
57     TIMSK1 &= ~(1 << OCIE1B); // Disable Timer1 Compare Match B interrupt, with the
58     // purpose of disableing ADC auto-trigger
59
60     uart1_send(0x55); // sync byte 1
61     uart1_send(0xAA); // sync byte 2
62
63     uint16_t payload_length = 2 + 2 + 1 + length + 2; // full packet size including
64     // header and checksum
65     uart1_send((payload_length >> 8) & 0xFF); // length high byte
66     uart1_send(payload_length & 0xFF); // length low byte
67     uart1_send(0x02); // packet type: OSCILLOSCOPE (0x02)
68
69     for (uint16_t i = 0; i < length; i++)
70         uart1_send(samples[i]); // send all sample data (1 byte each)
71
72     uart1_send(0x00); // checksum byte 1
73     uart1_send(0x00); // checksum byte 2
74
75     while (!(UCSR1A & (1 << TXC1)))
76         ; // wait for final byte to be fully shifted out
77     UCSR1A |= (1 << TXC1); // clear transmit complete flag
78
79     TIMSK1 |= (1 << OCIE1B); // re-enable ADC auto-trigger
80 }
81
82 // Send current generator configuration back to LabVIEW
83 void send_generator_packet(uint8_t active, uint8_t shape, uint8_t ampl, uint8_t freq)
{
84     uart1_send(0x55); // sync
85     uart1_send(0xAA);
86     uart1_send(0x00);
87     uart1_send(0x0B); // total length = 11 bytes
88     uart1_send(0x01); // packet type: GENERATOR (0x01)
89     uart1_send(active); // LED select (0 = shape, 1 = amp, 2 = freq)
90     uart1_send(shape); // waveform shape
91     uart1_send(ampl); // amplitude
92     uart1_send(freq); // frequency
93     uart1_send(0x00); // checksum
94     uart1_send(0x00);
95 }
96
97 // Parse incoming LabVIEW packet and respond accordingly
98 void parse_uart1_packet()
{
99     if (!uart1_packet_ready)
100        return;
101    uart1_packet_ready = 0;
102
103    if (uart1_rx_buffer[0] != 0x55 || uart1_rx_buffer[1] != 0xAA)
104    {
105        uart1_rx_index = 0; // invalid sync header
106        return;
107    }
108
109    uint16_t length = (uart1_rx_buffer[2] << 8) | uart1_rx_buffer[3];
110    if (length > UART1_RX_BUFFER_SIZE)
111    {
112        uart1_rx_index = 0; // reject oversized packet
113        return;
114    }
115
116    uint8_t type = uart1_rx_buffer[4];
117
118    switch (type)
119

```

```

120 {
121 case 0x01:
122 {
123     uint8_t btn = uart1_rx_buffer[5];
124     uint8_t sw = uart1_rx_buffer[6];
125
126     char msg[64];
127     sprintf(msg, "BTN: %d, SW: %d\r\n", btn, sw);
128     uart_send_string(msg);
129
130     switch (btn)
131     {
132     case 0x00: // SEND pressed
133         uart_send_string("BTN0 pressed: Send current values\r\n");
134         if (selected_param == 0 && sw <= 4)
135             shape = sw;
136         else if (selected_param == 1 && sw <= 255)
137             amplitude = sw;
138         else if (selected_param == 2 && sw <= 255)
139             frequency = sw;
140         else
141             uart_send_string("error: invalid parameter value\r\n");
142
143         transmit_singalgenerator_data(amplitude, frequency, shape);
144         uart_send_string("Current values sent to FPGA\r\n");
145         break;
146
147     case 0x01: // PARAM SELECT pressed
148         selected_param = (selected_param + 1) % 3;
149         uart_send_string("BTN1 pressed: Cycle parameter\r\n");
150         break;
151
152     case 0x02: // RUN/PAUSE pressed
153         run_stop_flag = !run_stop_flag;
154         uart_send_string(run_stop_flag ? "Paused\r\n" : "Running\r\n");
155         break;
156
157     case 0x03: // RESET pressed
158         uart_send_string("BTN3 pressed: Reset\r\n");
159         PORTD |= (1 << PD7);
160         _delay_ms(10);
161         PORTD &= ~(1 << PD7); // pulse reset to FPGA
162         selected_param = 0;
163         shape = 0;
164         amplitude = 128;
165         frequency = 1;
166         transmit_singalgenerator_data(amplitude, frequency, shape);
167         uart_send_string("Reset values sent to FPGA\r\n");
168         break;
169
170     default:
171         uart_send_string("Unknown BTN value\r\n");
172         break;
173     }
174
175     send_generator_packet(selected_param, shape, amplitude, frequency);
176     break;
177 }
178
179 case 0x02:
180 {
181     uint16_t record_len = (uart1_rx_buffer[7] << 8) | uart1_rx_buffer[8];
182     if (record_len > 0 && record_len <= MAX_RECORD_LENGTH)
183     {
184         record_length = record_len;
185         // set optimal sample rate based on record length
186         sample_rate = ((11520.0f * record_length) / (7.0f + record_length)) * 0.95f;
187         // calculate sample rate
188         current_timer1_top = F_CPU / (8.0f * sample_rate);
189         init_timer1(current_timer1_top); // reconfigure Timer1 for new sample rate
190     }
191
192     char msg[64];

```

```

192     uart_send_string("\r\n== Oscilloscope Configuration ==\r\n");
193
194     snprintf(msg, sizeof(msg), "Record Length: %u samples\r\n", record_length);
195     uart_send_string(msg);
196
197     snprintf(msg, sizeof(msg), "Max Sample Rate: %.2f sps\r\n", sample_rate);
198     uart_send_string(msg);
199
200     snprintf(msg, sizeof(msg), "Timer1 TOP: %lu\r\n", (unsigned long)
201             current_timer1_top);
202     uart_send_string(msg);
203
204     break;
205 }
206
207 case 0x04: // START packet received (used for SPI stress test)
208     uart_send_string("START received: triggering SPI stress test\r\n");
209     extern volatile uint8_t run_stress_test_flag;
210     run_stress_test_flag = 1;
211     break;
212
213 default:
214     uart_send_string("Unknown type\r\n");
215     break;
216 }
217
218 uart1_rx_index = 0; // reset buffer for next packet
219 }
```

Listing 14: UART1.comm.c

### 5.2.3 SPI\_comm.c

```

1 // ## Program developed by Mads Rodulph and Sigurd Hestbech Christiansen ##
2 // ## Last edited: 21-6-2025 ##
3 // ## This file contains SPI communication functions ##
4
5 #include <avr/io.h>
6 #include <stdio.h>
7 #include "uart.h"
8 #include <util/delay.h>
9 #include "SPI_comm.h"
10 #include "SPI.h"
11
12 void spi_stress_test_10000_packets()
13 {
14     // send message to terminal before reset
15     uart_send_string("Resetting FPGA before SPI stress test...\r\n");
16
17     // toggle PD7 high then low to reset FPGA
18     PORTD |= (1 << PD7);
19     _delay_ms(10);
20     PORTD &= ~(1 << PD7);
21
22     // notify that reset is complete
23     uart_send_string("FPGA reset complete. Starting SPI stress test (10,827 packets)...\\r
24         \\n");
25
26     uint16_t packet_count = 0;
27     uint8_t val;
28
29     for (uint16_t i = 0; i < 10827; i++)
30     {
31         val = i % 256;
32         transmit_signalgenerator_data(val, val, val); // send packet with repeating value
33         packet_count += 1;
34
35         if ((i % 500) == 0)
36         {
37             // print status update every 500 packets
38             char buf[48];
39             sprintf(buf, "Packets sent: %u\r\\n", packet_count);
40             uart_send_string(buf);
41         }
42
43     // send final message to terminal
44     char final_buf[64];
45     sprintf(final_buf, "Stress test complete. Sent %u packets.\r\\n", packet_count);
46     uart_send_string(final_buf);
47 }
48
49 void transmit_signalgenerator_data(uint8_t amp, uint8_t freq, uint8_t shape)
50 {
51     unsigned int checksum = 0;
52
53     for (uint8_t adress = 1; adress < 4; adress++)
54     {
55         switch (adress)
56         {
57             case 1:
58                 checksum = adress ^ 0x55 ^ amp;
59
60                 master_transmit(0x55);      // sync byte
61                 master_transmit(adress);   // address byte (1 = amplitude)
62                 master_transmit(amp);     // amplitude value
63                 master_transmit(checksum); // checksum byte
64                 break;
65
66             case 2:
67                 checksum = adress ^ 0x55 ^ freq;
68
69                 master_transmit(0x55);      // sync byte
70                 master_transmit(adress);   // address byte (2 = frequency)

```

```

71     master_transmit(freq);      // frequency value
72     master_transmit(checksum); // checksum byte
73     break;
74
75     case 3:
76         checksum = adress ^ 0x55 ^ shape;
77
78         master_transmit(0x55);      // sync byte
79         master_transmit(adress);   // address byte (3 = shape)
80         master_transmit(shape);   // shape value
81         master_transmit(checksum); // checksum byte
82         break;
83
84     default:
85         break;
86     }
87 }
88 }
```

Listing 15: SPI\_comm.c

#### 5.2.4 ADC.c

```

1 // ## Program developed by Mads Rodulph and Sigurd Hestbech Christiansen ##
2 // ## Last edited: 21-6-2025 ##
3 // ## This file contains ADC and autotrigger initialization functions ##
4 // ## References to datasheet pages are marked as S.xxx for Atmega2560 ##
5
6 #include "ADC.h"
7 #include <avr/io.h>
8 #include <avr/interrupt.h>
9
10 // Initialize ADC channel 0 (AO) with auto-trigger from Timer1 Compare Match B
11 void init_ADC_kanal0()
12 {
13     ADMUX = (1 << ADLAR); // Use external voltage reffrence on AREF pin, left-adjust
14     result (8-bit in ADCH) (S.289-290 & S.294)
15     // MUX [3:0] = 0000      ADC0 selected (S.290 Table 26-4)
16
17     // information on ADCSRA at S.292
18     ADCSRA = (1 << ADEN) | // Enable ADC
19                 (1 << ADATE) | // Auto Trigger Enable
20                 (1 << ADIE) | // Enable ADC interrupt (ISR will clear interrupt flag)
21                 (1 << ADSC) | // Start first conversion
22                 (1 << ADPS2); // Prescaler = 16 (ADC clock = 1MHz @ 16MHz(F_CPU))
23
24     ADCSRB = (1 << ADTS2) | (1 << ADTS0); // Auto Trigger Source = Timer1 Compare Match B
25     (ADTS [2:0] = 101)
26
27     DIDR0 = (1 << ADCOD); // Disable digital input on ADC0 (S.295)
28 }
29
30 // ADC ISR to acknowledge interrupt only
31 ISR(ADC_vect)
32 {
33     // Empty ISR to clear interrupt flag (sampling is handled in TIMER1_COMPB_vect)
34 }
35
36 // Initialize Timer1 to trigger ADC at given sampling rate
37 void init_timer1(int top_value)
38 {
39     TCCR1A = 0; // Clear Timer1 control A      avoid PWM
40     TCNT1 = 0; // Reset Timer1 counter
41
42     TCCR1B |= (1 << WGM12); // CTC mode (TOP = OCR1A)      S.148, S.161
43     TCCR1B |= (1 << CS11); // Prescaler = 8      S.161
44
45     TIMSK1 |= (1 << OCIE1B); // Enable Timer1 Compare Match B interrupt      S.166
46
47     OCR1A = top_value; // Sets TOP value for sampling frequency
48                 // Sampling rate = F_CPU / (8 * top_value)
49     OCR1B = top_value; // Used to trigger ADC
```

48 }

Listing 16: ADC.c

### 5.2.5 uart.c

```

1 // ## Program developed by Mads Rodulph and Sigurd Hestbech Christiansen ##
2 // ## Last edited: 21-6-2025 ##
3 // ## This file contains UART initialization and transmission functions ##
4 // ## References to datasheet pages are marked as S.xxx for Atmega2560 ##
5
6 #include <avr/io.h>
7 #include <avr/interrupt.h>
8 #include <string.h>
9 #include <util/delay.h>
10 #include <stdio.h>
11 #include "UART.h"
12
13 void uart_init(unsigned int ubrr)
14 {
15     UBRROH = (unsigned char)(ubrr >> 8);                      // USART0 Baud rate register
16     high byte (S.412)
17     UBRROL = (unsigned char)ubrr;                                // USART0 Baud rate register
18     low byte (S.412)
19     UCSROA |= (1 << U2X0);                                     // Enable double speed (S.223)
20     UCSROB = (1 << RXENO) | (1 << TXENO) | (1 << RXCIE0); // Enable RX, TX, and RX
21     interrupt (S.224)
22     UCSROC = (1 << UCSZ01) | (1 << UCSZ00);                  // 8-bit data format (S.226)
23     // No parity (UPM00 & UPM01 = 0), 1 stopbit (USBS0 = 0), async mode (UMSEL00 &
24     // UMSEL01 = 0) (S.225-226)
25     // One complete data cycle is 1 start-bit followed by 8 data-bit's followed by 1 stop
26     -bit.
27     // USART0 used for debugging via USB serial terminal
28 }
29
30 void uart_send(char data)
31 {
32     while (!(UCSROA & (1 << UDRE0)))
33         ;                                // Wait until data register is empty
34     UDR0 = data; // Load data into register for transmission
35 }
36
37 void uart_send_string(const char *str)
38 {
39     while (*str)
40         uart_send(*str++); // Send each character in the string
41 }
42
43 void uart1_init(unsigned int ubrr)
44 {
45     UBRR1H = (unsigned char)(ubrr >> 8);
46     UBRR1L = (unsigned char)ubrr;
47     UCSR1A |= (1 << U2X1); // Double speed mode
48     UCSR1B = (1 << RXEN1) | (1 << TXEN1) | (1 << RXCIE1);
49     UCSR1C = (1 << UCSZ11) | (1 << UCSZ10); // 8-bit data format
50     // USART1 used for LabVIEW communication over RS232
51 }
52
53 void uart1_send(char data)
54 {
55     while !(UCSR1A & (1 << UDRE1))
56         ;                                // Wait until data register is empty
57     UDR1 = data; // Load data into register for transmission
58 }
```

Listing 17: uart.c

## 5.2.6 SPI.c

```

1 // ## Program developed by Mads Rodulph and Sigurd Hestbech Christiansen ##
2 // ## Last edited: 21-6-2025 ##
3 // ## This file contains SPI initialization and transmission functions ##
4 // ## References to datasheet pages are marked as S.xxx for Atmega2560 ##
5
6 #include "SPI.h"
7 #include <avr/io.h>
8 #include <stdio.h>
9 #include "uart.h"
10 #include <util/delay.h>
11
12 // Send a byte via SPI as master
13 void master_transmit(uint8_t data)
14 {
15     PORTB &= ~(1 << PBO); // Set SS low to activate slave
16     SPDR = data;           // Load data into SPI Data Register
17
18     while (!(SPSR & (1 << SPIF)))
19         ; // Wait for transmission to complete
20     PORTB |= (1 << PBO); // Set SS high to deactivate slave
21     // Transmission complete, slave deselected
22 }
23
24 // Initialize SPI in master mode
25 void master_init()
26 {
27     DDRB |= (1 << PBO) | (1 << PB1) | (1 << PB2); // Set SS, MOSI, and SCK as output, on
28     // ArduinoMega2560(MOSI(D51), SCK(52), SS(D53))
29     DDRB &= ~(1 << PB3); // Set MISO as input, on
28     // ArduinoMega2560(MISO(D50))
29     PORTB |= (1 << PBO); // Set SS high (inactive)
30
31     SPCR |= (1 << SPE) | (1 << MSTR) | (1 << DORD); // Enable SPI in master mode, (DORD =
32     // 1) makes so LSB transmits first
33     SPCR |= (1 << SPI2X) | (1 << SPR1); // Set SCK clock rate too fck/32 (S
34     .203), 500kHz with F_CPU = 16 MHz(500 kilo baud)
35     SPCR &= ~((1 << CPOL) | (1 << CPHA)); // Sample on rising edge, transmit on
36     // falling (S.200 202 )
37     // SPI mode = 0, LSB first, double speed, clock polarity and phase = 0
38 }
39
40 // Receive a byte via SPI as slave
41 unsigned char slave_reciver(unsigned char data)
42 {
43     SPDR = data; // Send dummy data to initiate clock
44
45     while (!(SPSR & (1 << SPIF)))
46         ; // Wait for transmission to complete
47
48     return SPDR; // Return received data
49     // Data received from SPI master
50 }
51
52 // Initialize SPI in slave mode
53 void slave_init()
54 {
55     DDRB |= (1 << DDB3); // Set MISO as output
56     SPCR |= (1 << SPE) | (1 << DORD); // Enable SPI in slave mode, LSB first
57 }
```

Listing 18: SPI.c

### 5.2.7 UART1.comm.h

```
1 #ifndef UART1_COMM_H
2 #define UART1_COMM_H
3
4 #include <stdbool.h> // Include boolean support
5 #include <stdint.h>
6
7 void send_oscilloscope_packet(uint8_t *samples, uint16_t length);
8
9 extern volatile uint8_t uart1_rx_buffer[];
10 extern volatile uint8_t uart1_rx_index;
11 extern volatile uint8_t uart1_packet_ready;
12 extern volatile uint8_t shape;
13 extern volatile uint8_t amplitude;
14 extern volatile uint8_t frequency;
15 extern volatile uint8_t run_stop_flag;
16
17#endif
```

Listing 19: UART1.comm.h

### 5.2.8 SPI.comm.h

```
1 #ifndef SPI_COMM_H
2 #define SPI_COMM_H
3
4 #include <avr/io.h>
5
6 // SPI function declarations
7 void spi_stress_test_10000_packets(void);
8 void transmit_singalgenerator_data(uint8_t amp, uint8_t freq, uint8_t shape);
9
10#endif // SPI_H
```

Listing 20: SPI.comm.h

### 5.2.9 ADC.h

```
1 #ifndef ADC_H
2 #define ADC_H
3
4 // Initializes ADC on channel 0 (A0) with Timer1 Compare Match B trigger
5 void init_ADC_kanal0(void);
6
7 // Initializes Timer1 to trigger ADC sampling at desired rate
8 void init_timer1(int top_value);
9
10#endif
```

Listing 21: ADC.h

### 5.2.10 uart.h

```
1 #ifndef UART_H
2 #define UART_H
3 #include <stdbool.h> // Include boolean support
4 #include <stdint.h>
5
6 void uart_init(unsigned int ubrr);
7 void uart_send(char data);
8 void uart_send_string(const char *str);
9
10 void uart1_init(unsigned int ubrr);
11 void uart1_send(char data);
12 void send_oscilloscope_packet(uint8_t *samples, uint16_t length);
13
14 extern volatile uint8_t uart1_rx_buffer[];
15 extern volatile uint8_t uart1_rx_index;
16 extern volatile uint8_t uart1_packet_ready;
17 extern volatile uint8_t shape;
18 extern volatile uint8_t amplitude;
19 extern volatile uint8_t frequency;
20 extern volatile uint8_t run_stop_flag;
21
22#endif
```

Listing 22: uart.h

### 5.2.11 SPI.h

```
1 #ifndef SPI_H
2 #define SPI_H
3
4 #include <avr/io.h>
5
6 // SPI function declarations
7 void master_init(void);
8 void slave_init(void);
9 void master_transmit(uint8_t data);
10 unsigned char slave_reciver(unsigned char data);
11
12#endif // SPI_H
```

Listing 23: SPI.h

## — Kildekode og GitHub

Projektets kildekode og dokumentation er tilgængelig i det tilhørende GitHub-repository:

- [https://github.com/MadsRudolph/Oscilloscope\\_Project](https://github.com/MadsRudolph/Oscilloscope_Project)

Repositoriet indeholder:

- Firmware-kode til ATmega2560
- FPGA VHDL-filer
- Filterberegninger og kicad simuleringer