UNIVERSITETET I AGDER

Prosjektrapport i ELE-112

Fakultet for teknologi- og realfag, Grimstad

Tittel: ELE-112 Prosjekt vår 2019	**Antall sider:** 14 **Tilgjenglighet:** Åpen
Forfattere: Mohammed Alrawi, Simen Mathias Risnes, Fredrik Kullerud	**Semester:** Vår 2019
Emneord: Akselerometer ADXL362, Nexys 4 DDR, 7-segment, Vivado, SPI.	

Abstrakt:

I denne rapporten skal vi forklare hvordan vi gikk fram for å hente informasjon fra et ADXL 362 akselerometer på et Nexys 4DDR kort.

For å løse denne oppgaven har vi brukt Vivado v2018.3 til koding. Vi startet med å lage blokkdiagram for Del 1 og Del 2 av oppgaven for å få en oversikt over hva vi kunne og hva vi måtte lære mer om. Deretter laget vi og testet komponentene hver for seg, før vi koblet dem sammen.

Sluttresultatet til produktet er slik vi hadde planlagt fra starten av og har funksjonaliteten vi ønsker. Vi leser verdiene fra akselerometeret og får vist det på 7-segment form.

Telefon: +47 37 23 30 00	Jon Lilletuns vei 9, 4879 Grimstad	Telefax: +47 38 14 10 01

UNIVERSITETET I AGDER

ELE-112 PROSJEKT 2019

Laget av:
Mohammed Alrawi
Simen Mathias Risnes
Fredrik Kullerud

Lærere:
Geir Jevne
Ragnhild Veimo Larsen

26.04.19, Grimstad

Forord

Denne rapporten handler om et skoleprosjekt på UiA Grimstad, i faget ELE-112. Oppgaven går ut på å ta målinger fra et akselerometer, og vise verdiene på 7-segment display. Denne rapporten er skrevet av 3 studenter som går elektronikk, vi ønsker å takke alle som har hjulpet oss på veien, da spesielt til lærere på elektronikk og elever som går andre års trinn som har gjort denne oppgaven før.

Innhold

Innledning

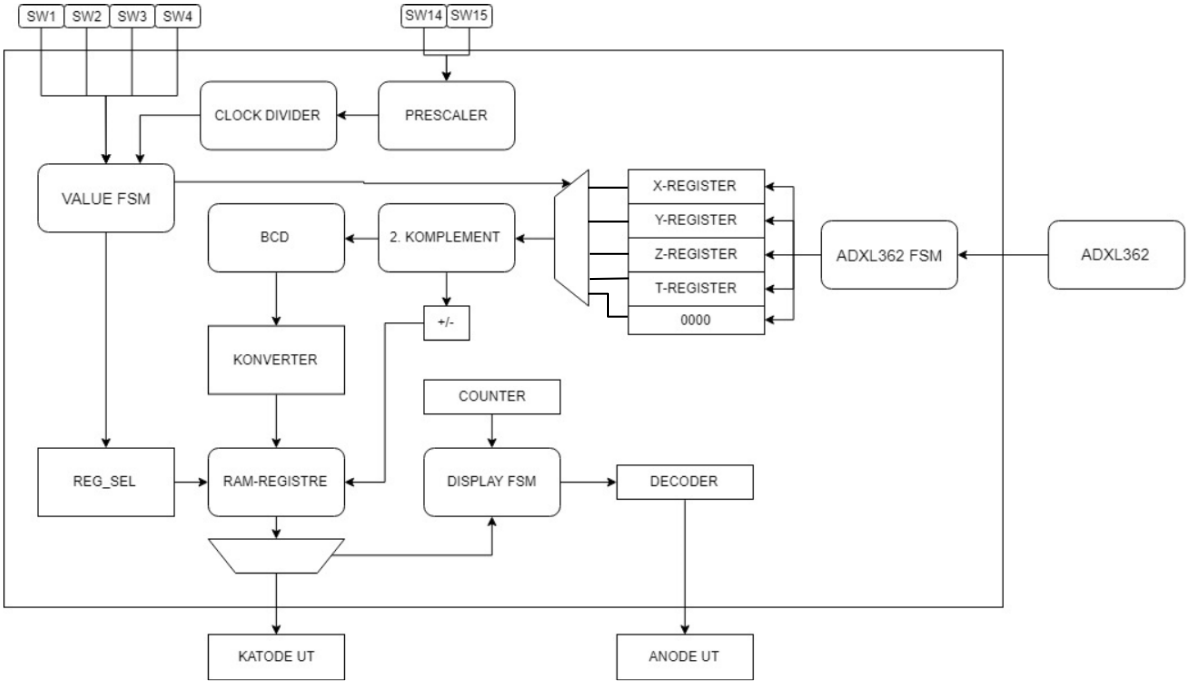
Prosjektoppgaven gikk ut på å hente data fra et ADXL 362 akselerometer og vise dette på syv segment displayet på et Nexys 4 DDR(A 10) kort, gjennom å kommunisere med bruk av SPI. I denne rapporten skal vi forklare hvordan vi kom fram til løsningen og hvilke hindre vi møtte på underveis. Vi skal også forklare kort om hvordan de essensielle komponentene vi har laget og brukt fungerer.

Metode

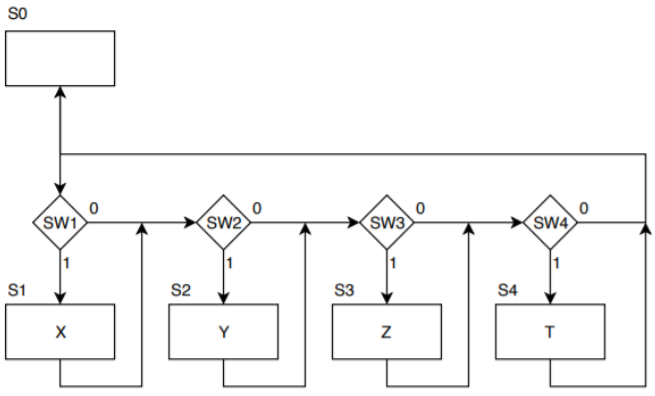
Vi brukte programmet Vivado v2018.3 til koding, i tillegg til Nexys A10(4 DDR) kortet og Digilent Inc. sin nettside.

Når vi skulle angripe problemet startet vi med å lage ett blokk diagram for å få oversikt over hva som måtte lages og for å få en oversikt over hvilke egenskaper komponentene måtte ha for å fungere sammen. Etter å ha gjort dette ble det veldig greit å fordele oppgaver innad i gruppen i tillegg ble vi enige om en estimert tidsplan. Deretter satt vi i gang med å kode komponentene vi trengte. Når vi hadde kodet ferdig lagde vi testbench til hver enkelt komponent for å se om de fungerte slik vi ønsket. Etter noe mindre forbedringer og feilrettinger på komponentene satte vi alle komponentene sammen til det ferdige produktet i del 1. Etter dette ble det også behov for noen forbedringer, og vi var etter hvert klare for å sette del 1 og del 2, som inneholder lab 5, en tilstandsmaskin og akselerometeret, sammen. Vi har vært fornøyde med fremgangsmåten vi valgte, det har vært lett å ha god oversikt over hva som må gjøres og hva vi har manglet. Å ha et bra blokkdiagram har vært til god hjelp både under koding og under feilsøking når det trengtes.

Blokk Diagram



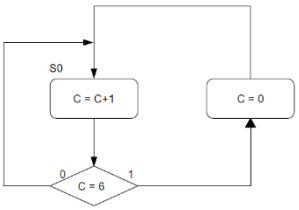
Value FSM



Denne komponenten blir styrt av SW1(X-register), SW2(Y-register), SW3(Z-register) og SW4(T-register), og går på klokken som kommer ut av Clock Divider komponenten. Switchene styrer hvilke register som skal vises. Er flere av switchene høye samtidig, veksler den mellom hvilket register som skal vises, hver gang klokken går høy. Måten den styrer dette på er ved å sende et tre bits signal til en 5 til 1 MUX, som bestemmer hvilket register som skal sendes gjennom. Er alle switchene lave vil MUXen sende gjennom et register der alle bitene er satt til 0.

Display FSM

Denne komponenten looper gjennom de seks 7-segment displayene som skal vises, ved hjelp av en teller, ettersom den bare kan vise en av de seks om gangen. Telleren gir ut ett 3 bits signal som styrer MUXen til ram registrene og blir sendt til en 3- til 8-decoder som styrer anodene.



ADXL362 Control

Vi bruker SPI-Interface (SPI-Lab 5) til å kommunisere med ADXL362. Ved initialiseringstid tilbakestilles ADXL362, og konfigurerer deretter sine interne registre. Etter å ha konfigurert sine interne registre, vil akselerasjonen bli lest på de tre aksene som følges av temperaturdataene: Et sett med 8 databytes er lest: XDATA_L, XDATA_H, YDATA_L, YDATA_H, ZDATA_L, ZDATA_H, TEMP_L og TEMP_H. Lesingen gjøres kontinuerlig og et gjennomsnitt er laget av 16 lesninger.

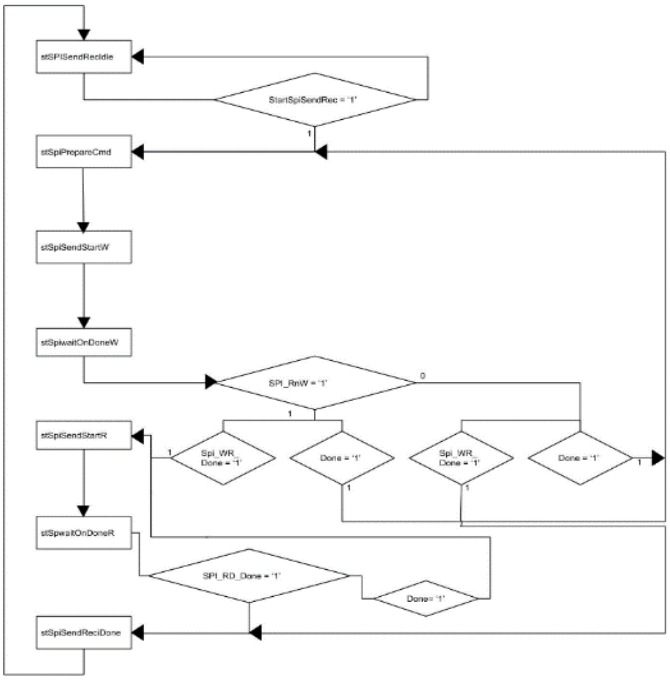
Antallet lesninger for gjennomsnittet som blir laget, skal være potensialet av (2) og bestemmes av NUM_READS_AVG (= 16) parameteren, som er konstant 16. UPDATE_FREQUENCY_HZ-parameteren setter en teller til en periode på 1 / UPDATE_FREQUENCY_HZ (ca. 1ms). Tilstandsmaskinen venter denne perioden før den starter et antall NUM_READS_AVG ganger data lese prosedyre. Før vi leser et sett med data, leser tilstandsmaskinen og sjekker statusregisteret for å se når nye data er tilgjengelige på ADXL362. Derfor er den reelle "prøve"-tiden avhengig av ADXL362 ODR (Output Data Rate) frekvens, satt i Filter Control Register, adresse 0x2C, i dette prosjektet satt som standard til 200Hz.

Modulen består av tre tilstandsmaskiner:

SPI send & receive FSM

Denne tilstandsmaskinen oppretter en handshake-transaksjon med SPI-Interface (SPI-If-lab 5), bruker den Start og Done-signalene til:

- Å Sende antall byte angitt av Cnt_Bytes_Sent-telleren (3 når du konfigurerer en ADXL362 intern registrer, 2 når du sender en lesekommando). Bytesene som skal sendes via SPI-Interface, er hentet fra kommandoregister, Cmd_Reg.



- Å Motta antall byte spesifisert av Cnt_Bytes_Rec-telleren (8 når du leser akselerasjon og temperaturdata, 1 når du leser statusregisteret), når lesing kreves (SPI_RnW = 1) Akselerasjons- og temperaturdata lagres i dataregisteret Data_Reg.

Signaler	
6 - Shift_Cmd_Reg	5 - EN_Shift_Data_Reg
4 - SPI_SendRec_Done	3 - Start
(2 downto 0) - Dummy data	

Tilstand	MSB (6)	(5)	(4)	(3)	(2)	(1)	LSB (0)
stSPISendRecIdle	0	0	0	0	0	0	0
stSpiPrepareCmd	1	0	0	0	0	0	1
stSpiSendStartw	0	0	0	1	0	1	1
stSpiWaitOnDoneW	0	0	0	0	1	1	1
stSpiSendStartR	0	0	0	1	1	1	0
stSpiWaitOnDoneR	0	1	0	0	1	0	0
stSpiSendRecDone	0	0	1	0	1	0	1

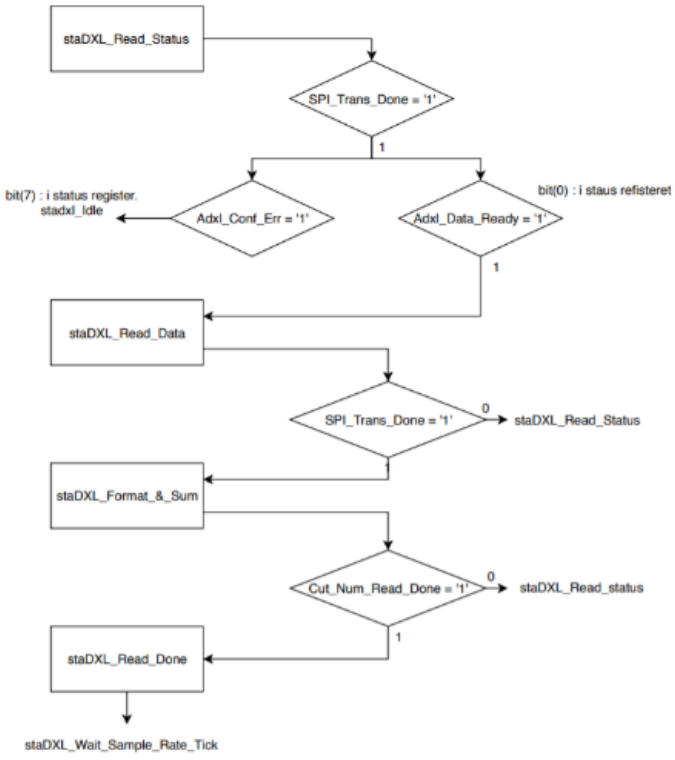
ADXL Control FSM

Denne tilstandsmaskinen styrer den tidligere beskrevne SPI Transaction tilstandsmaskinen, også ved å bruke "handshake" med StartSpiTr (write) og SPI_Trans_Done (read) signaler:

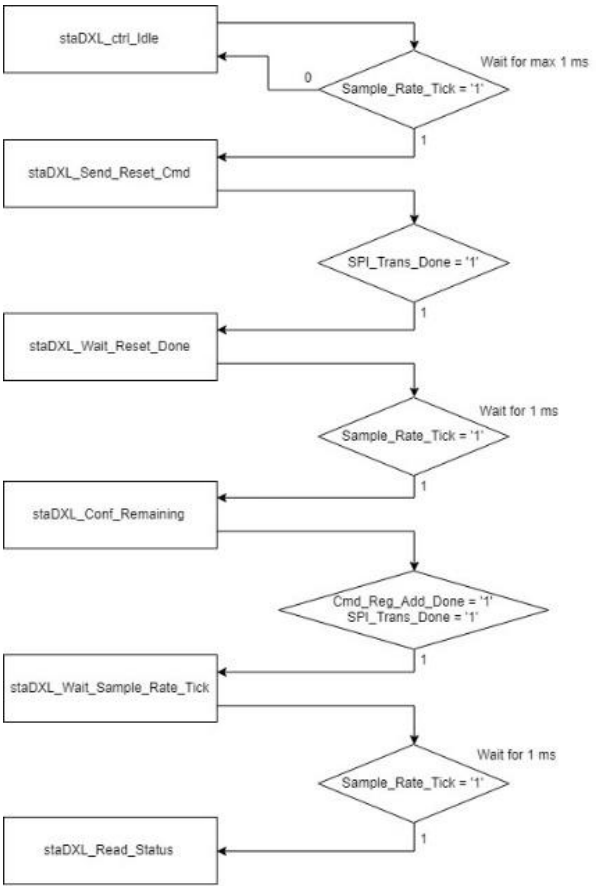
1. Først vil Cmd_Reg bli lastet med Reset-kommandoen fra Cmd_Reg_Data ROM og tilstandsmaskinen starter SPI Transaction tilstandsmaskinen for å sende Reset-kommandoen til ADXL362-akselerometeret.
2. Tilstandsmaskinen venter på en tidsperiode(ca. 1 ms) og sender deretter de gjenværende konfigurasjonsregisterdataene fra Cmd_Reg_Data til ADXL362 akselerometeret.

```
Cmd_Reg_Data : rom_type := ( X"1F", X"52",  -- Soft Reset Register Address and Reset Command
                             X"1F", X"00",  -- Soft Reset Register Address, clear Command
                             X"2D", X"02",  -- Power Control Register, Enable Measure Command
                             X"2C", X"14"   -- Filter Control Register, 2g range, 1/4 Bandwidth, 200HZ Output Data Rate
                           );
```

3. Etter at du har konfigurert ADXL362, venter tilstandsmaskinen i en tidsperiode som er lik 1 / UPDATE_FREQUENCY_HZ(ca. 1 ms) før den begynner å lese
4. Etter at tiden har gått, leser tilstandsmaskinen Statusregisteret, (bit(0)) og blir høy når nye data er tilgjengelige, og når det er nye data tilgjengelige, leser X-, Y- og Z-akselerasjonsdataene, etterfulgt av temperaturdata. Et antall lesninger tilsvarende NUM_READS_AVG(16) utføres, dvs. trinn 4 gjentas NUM_READS_AVG(16) 16 ganger.



5. Dataene leses i gjennomsnitt i ACCEL_X_SUM-, ACCEL_Y_SUM-, ACCEL_Z_SUM- og ACCEL_TMP_SUM-registrene. NUM_READS_AVG er potensialet av 2, og for å gjøre gjennomsnittet lettere ved å fjerne de minst signifikante bitene(her tok vi 4 LSB, siden det var 16 lesninger). Etter en rekke lesninger tilsvarende NUM_READS_AVG er ferdig, oppdaterer tilstandsmaskin utdataene ACCEL_X, ACCEL_Y, ACCEL_Z og ACCEL_TMP, lagret i 12-bits toers komplement format og signaler til utgangen ved å aktivere for en klokkeperiode Data_Ready-signalet. Deretter fortsetter tilstandsmaskinen til trinn 3 i en uendelig sløyfe Tilstandsmaskinen starter bare på nytt fra trinn 1 når FPGA er omkonfigurert eller Reset-signalet er aktivert.

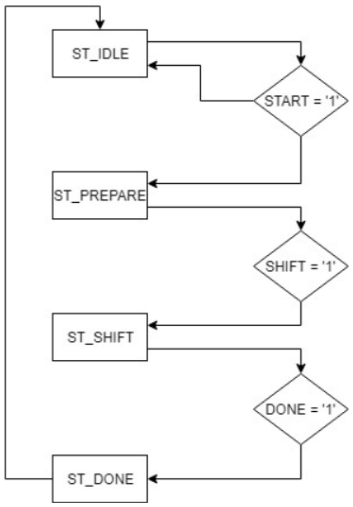


Dataene i ACCEL_X, ACCEL_Y,ACCEL_Z og ACCEL_TMP-registrene blir sendt til registrene som ligger før 5-to-1 MUX i Del 1. En FSM som bruker Switchene (3 til 0), styrer hvilken akse (X,Y,Z og T) skal vises på 7-segmentet, hvis ingen av switchene er på så vises det nuller. Denne FSM'en kjøres på en Clock-divider(fra 1 Hz til 0.5, 0.25 og 0.125 Hz) som blir styrt av Switchene 14 og 15 De dataene som slippes ut fra MUXen går gjennom 3 hoved prosesser:

- Fra 2's komplement form til unsigned binært: I den komponenten i tillegg til konverteringen, blir det sendt et signal til (plussminus- komponent) for å vise fortegnet på 7-segmentet.
- Deretter blir dataene utsendt til Double dabbel komponenten, det binære tallet blir gjort om til BCD-form.
- Etter at dataene er omgjort til BCD, blir de sendt ut til BCD-til-7-segment-form komponent. Der blir de gjort om til 7-segment form.
- Dataene i 7-segment-form blir lagret i RAM-registerne som er koblet til 6-to-1 MUX. MUX'en blir styrt av en FSM som teller fra 0 til 5, den sender utgangen av telleren til både 6-to-1 MUX'en og 3-to-8 decoderen.
- 3-to-8 decoderen er koblet til anodene i 7-segmentet Displayet. Utgangen av 6-to-1 MUX'en er koblet til katodene.

SPI transaction FSM

Denne tilstandsmaskinen styrer den tidligere beskrevne SPI Send / Recieve Control tilstandsmaskin, bruker "handshake" med StartSpiSendRec (write) og SPI_SendRec_Done (read) signaler:



- Forbereder og laster kommandoregisteret, Cmd_Reg med den passende kommando stringen (konfigurer de bestemte registrene, lese data eller lese status).
- Laster Cnt_Bytes_Sent og Cnt_Bytes_Rec med antall byte som skal sendes og / eller mottas.
- Aktiverer StartSpiSendRec for å starte SPI Send / Receive Control tilstandsmaskin og venter på svaret ved å lese SPI_SendRec_Done-signalet.

Mellom hver SPI-transaksjon (Register write eller Register read) må SS-signalet deaktiveres i minst (10 us) før en ny kommando er utsendt.

SPI Interface (LAB 5)

Denne modulen representerer en SPI-kontroller. Kontrolleren overfører 8 bits data, MSB først. Data er lest på den positive kanten av SCLK og også stabil på den positive kanten av SCLK. Når det ikke er dataoverføring, SCLK = 0, derfor CPOL = 0, CPHA = 0. Dataoverføring starter ved å aktivere "Start" -signalet, og når dataoverføringen er ferdig, vil "Done" -signalet er aktivert for en SYSCLK-periode. Modulen kan også overføre flere byte hvis "HOLD_SS" -signalet er aktivt. I dette tilfellet SS ikke heves mellom byte overføringer og en ny overføring kan begynne hvis "Start" -signalet er aktivert. Denne funksjonen brukes sammen med ADXL 362 akselerometer eller en hvilken som helst enhet som trenger flere byteoverføringer for å sende kommandoer og /eller lese data

BCD

Det er en algoritme for å konverter fra binære tall til BCD (binary-coded decimal), den algoritmen er kjent også som “shift-and-add-3” algoritme. Algoritmen repeteres n (antall bits) ganger. På hver repetisjon skiftes en bit til venstre. Men før venstre skift er gjort, økes et BCD-nummer større enn 4 med 3. Økningen sikrer at en verdi på 5, økt og forskjøvet til venstre, blir 16, og dermed korrekt "carrying" inn i det neste BCD-sifferet(hvert BCD-siffer består av 4 bits).

I hovedsak opererer algoritmen ved å fordoble BCD-verdien til venstre for hver repetisjon og legge til enten en eller null i henhold til det originale bitmønsteret. Venstre skift utfører begge oppgavene samtidig. Hvis et tall er fem eller høyere, legges tre til for å sikre verdien "carrying" i base 10.

```
0000 0000 0000   11110011   Initialization
0000 0000 0001   11100110   Shift
0000 0000 0011   11001100   Shift
0000 0000 0111   10011000   Shift
0000 0000 1010   10011000   Add 3 to ONES, since it was 7
0000 0001 0101   00110000   Shift
0000 0001 1000   00110000   Add 3 to ONES, since it was 5
0000 0011 0000   01100000   Shift
0000 0110 0000   11000000   Shift
0000 1001 0000   11000000   Add 3 to TENS, since it was 6
0001 0010 0001   10000000   Shift
0010 0100 0011   00000000   Shift
   2    4    3
      BCD
```

Dobble-algoritmen, utført til verdi $(230)_{10}$, ser slik ut, figur høyre side:

Toers komplement

Toers komplement komponenten sjekker om verdien vi får fra akselerometeret er et positivt eller negativt tall på toers komplement form. Dersom det er et negativt tall på toers komplement endres tallet til et positivt binært tall før det sendes videre og sender ut et ett bits signal som er 1 når tallet er negativt og 0 dersom det er positivt.

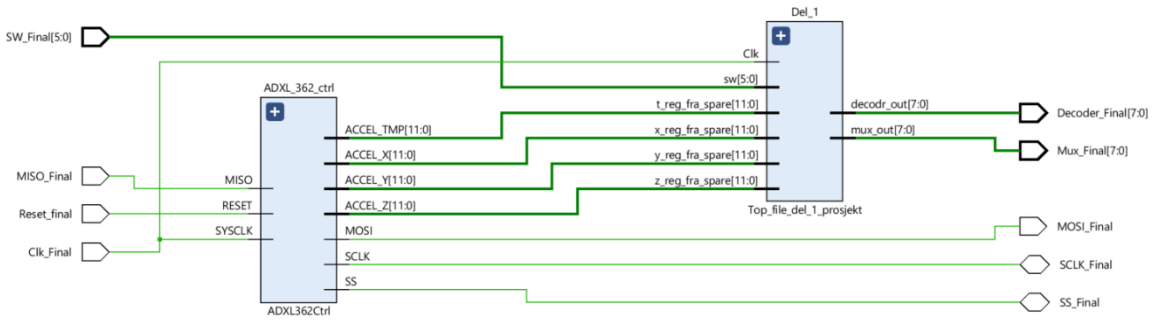
Clock divider & Prescaler¨

Prescaleren tar inn en klokke på 100 MHz og sender ut en klokke på 1 Hz, deretter går klokka inn i en Clock Divider som blir styrt av SW14 og SW15. Ut fra hvordan man styrer SW14 og SW15 får man ut en klokke på 1 Hz, 0.5 Hz, 0.25 Hz og 0.125 Hz. Denne sendes inn til Tilstandsmaskin 1 og styrer hvor lenge den skal stå i hver tilstand.

Reg_sel, pluss/minus, konverter

Reg_sel tar inn signalet som brukes som styresignal fra Value FSM til 5 til 1 MUXen. Den konverterer dette til et 8 bits signal som sendes til syv segmentet for å vise hvilket register man viser data fra. Pluss minus tar in 1 bits signalet ut fra Toers komplement komponenten, denne komponenten endrer dette til et 8 bits signal som gjør at vi kan vise om tallet er negativt eller ikke på syv segmentet. 4 til 8 bit konvertert tar inn signalene fra Double Dabble og konverterer disse 4 bits signalene til 8 bits signaler slik at vi kan vise tall fra 0 til 9 på syv segmentet.

VIVADO Schematic



Vivado Schematic:
Vår løsning består av 2 deler, en del (Del_1) som styrer at verdiene vi får blir vist på syv segmentet og en del (ADXL_362_ctrl) som henter data fra ADXL 362 akselerometeret.

Testing

Testingen vår har hovedsakelig gått ut på å teste hver enkel komponent før vi begynte å sette de sammen. For å teste alle komponentene, lagde vi en testbench til hver enkelt komponent. Etter vi hadde satt komponentene sammen fant vi ut at vi ikke fikk noen fornuftige sifre ut på 7-segment displayet og fant ut av dette var fordi LSB og MSB kom i feil rekkefølge. Vi endret da deklareringen til signalet fra (0 to 7) til (7 downto 0). Etter å ha gjort dette fikk vi fortsatt like rare tall ut på displayet, og måtte ha feil andre steder også. Det var da vi skjønte at det måtte være noe med rekkefølgen vi valgte katodene. Vi måtte da endre rekkefølgen i Nexys filen også.

Når alt så bra ut i del 1 av prosjektet prøvde vi det på kortet igjen og alt så ut til å fungere som det skulle. Når vi fikk til dette portmap'et vi del 2 inn og prøvde å teste det på kortet igjen, men vi fikk bare lest verdier fra 2 registre. Vi lagde da en testbench for hele prosjektet for å prøve å finne feilen i simulasjonen. Vi fant da ut at feilen lå i tilstandsmaskinen inne i labb 5, endringen vi måtte gjøre var å resette akselerometeret og så enable «measure command» og så konfigurere «filter control» registret slik at rangen er ±2g «default», ¼ bandwidth, 200Hz «output data rate». Så leser vi status registret, bit nummer 0, hvis den er 1 har det kommet ny data som skal leses. Vi tar gjennomsnittet av 16 lesinger av hvert register før vi sender det til ram-registrene i del 1.

Diskusjon

Her skal vi komme litt innom hvordan planen og utføringen vår har gått som det skulle. Vi startet, som nevnt, med planleggingen med å lage et blokkdiagram for problemet. Produksjonen av de aller fleste komponentene i del 1 gikk som planlagt, og fungerte som de skulle hver for seg, mens i del 2 brukte vi ca. 20 timer (det var estimert 10 timer). I del 1 av problemstillingen gikk det aller meste ganske knirke fritt, selv om det selvfølgelig trengtes noen mindre finjusteringer på noen av komponentene.

Det var i del 2 vi møtte på de største utfordringene, vi måtte inn i lab 5 og endre ganske mye for at den skulle være kompatibel med komponentene våre i del 1. Når vi hadde fått løst disse problemene fikk vi enda større problemer med tilstandsmaskinen vår som skulle kjøre lab 5 og lese verdiene ut fra akselerometeret. Vi fikk rett og slett den aldri til å fungere, men det var det noen som hadde klart tidligere. Vi fant en tilstandsmaskin på nettet som fikk det til å kjøre, så vi bestemte oss får å bruke den. Nå så alt ut til å fungere som det skulle men det var enda en feil, To av bryterne våre var defekte slik at de alltid var på, slik at verdiene bryterne styrte alltid ble vist. Dette ble løst med å bytte brytere.

Konklusjon

Vi har lært under arbeidet at SPI var en veldig god måte å kommunisere mellom master og slave. Når man bruker ett shift-register for data overføring slipper man å bruke så mange porter og kommunikasjonen går fortsatt fort.

Sluttresultatet til produktet er slik vi hadde planlagt fra starten av og har funksjonaliteten vi ønsker. Vi leser verdiene fra akselerometeret og får vist det på 7-segment form. For å få vist temperaturen i celsius måtte vi dele rådverdien ut fra akselerometeret med 0.065. Vi har lagt til litt funksjonalitet underveis for å forbedre resultatet, som å ta snittet av 16 målinger før vi sender det videre. Vi vil si at produktet fortsatt har litt forbedringspotensial, for å ta noe så blinker 7- segment displayet litt.

Råd om å starte med å lage blokk diagram for så å dele oppgaven opp i mindre deler har vært til stor hjelp. Etter vi var ferdige med blokk diagrammet kom vi veldig godt i gang med å lage alle komponentene til del 1 og vi fikk denne delen til å fungere relativt greit grunnet godt grunnarbeid.

Arbeidet for å få produktet ferdig har vært godt, vi kunne nok brukt tiden litt bedre og vært mer effektive, men produktet har blitt ferdig til deadline og vi er fornøyde med produktet vi har levert.

Kilder og Vedlegg

StackExchange, *How to divide 50MHz down to 2Hz in VHDL on Xilinx FPGA*. Hentet 14.03.2019. https://electronics.stackexchange.com/questions/61422/how-to-divide-50mhz-down-to-2hz-in-vhdl-on-xilinx-fpga

Wikipedia, *Double Dabble*. Hentet 14.03.2019. https://en.wikipedia.org/wiki/Double_dabble#VHDL_implementation,

DIGILENT, *Nexys 4 DDR Advanced I/O Demo (Built-In Self-Test)*. Hentet 12.04.2019. https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-user-demo/start

Vedlegg:

Timeliste:

Dato	Tid brukt
Fre. 01.03.2019	3t 30min
Man. 04.03.2019	2t 30min
Fre. 08.03.2019	3t 30min
Tors. 14.03.2019	3t 30min
Man. 19.03.2019	2t 00min
Fre. 22.03.2019	4t 00min
Man. 25.03.2019	4t 00min
Fre. 29.03.2019	6t 00min
Man. 01.04.2019	4t 00min
Man. 08.04.2019	5t 00min
Fre. 12.04.2019	4t 00min
Tirs. 23.04.2019	6t 00min
Ons. 24.04.2019	6t 00min
Tors. 25.04.2019	6t 00min
Fre. 26.04.2019	4t 00min
Totalt tidsbruk	66t 00min

Vi brukte en del mer tid på prosjektet enn det vi først hadde beregnet. Dette var fordi prosjektet var mer oppfangende og krevende enn det vi først trodde. Vi kom godt i gang og fulgte tidsplanen bra i starten, men når vi kom lengre med prosjektet og begynte med delene vi ikke var helt sikre på hvordan vi skulle løse ble det brukt mye tid på å finne løsninger. Etterhvert som vi møtte på feilmeldinger ble det brukt mer tid på dette enn det vi hadde tatt forbehold om. Det vi lærte av dette er at neste gang vi skal jobbe med et større prosjekt, må vi beregne mer tid til delene vi ikke er helt sikre på hvordan vi skal løse og feilmeldinger. Tidene er per gruppemedlem.