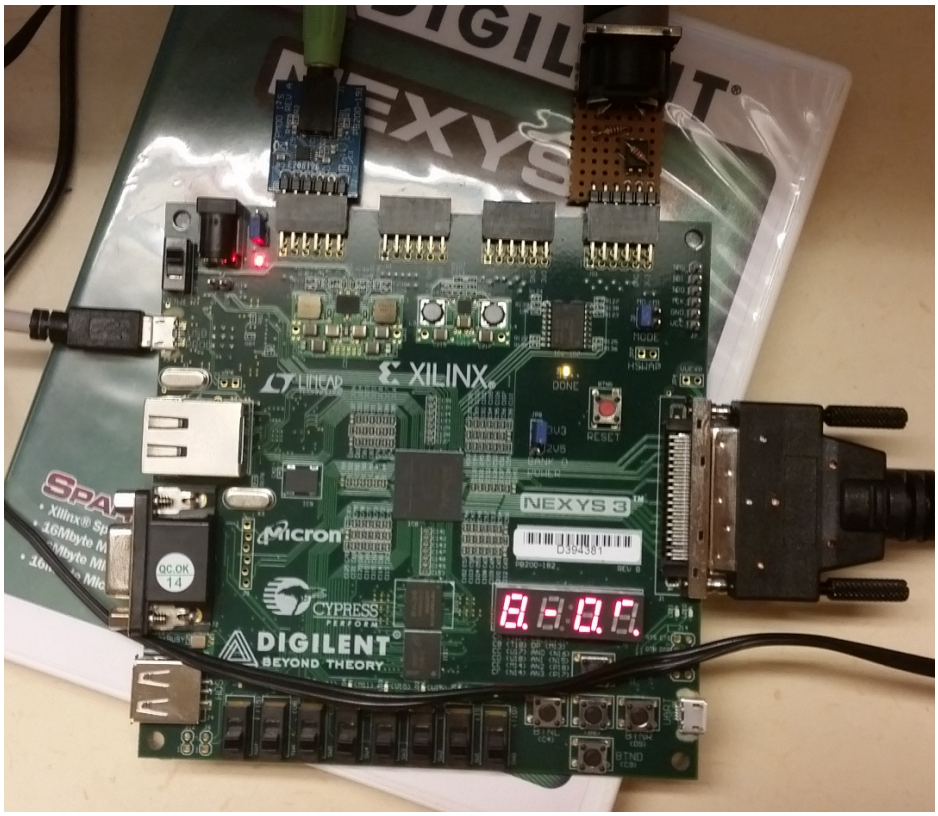
Teknisk rapport

Subtraktiv synth



Version 1.00

Grupp 34
Joacim Stålberg joast229
Mattias Ulmstedt matul773
Viktor Ringdahl vikri500

Innehållsförteckning

1. Inledning

Detta dokument utgör den tekniska rapporten för konstruktionen av en subtraktiv synth, vilket innebär att ljudet som kommer från apparaten går igenom ett filter som ändrar signalen för att skapa diverse speciella ljud. För att spela ljud trycker man givetvis tangenter på keyboardet. Inställningar till synthen görs även de från keyboardet och en LCD-skärm används för att visa dessa inställningar för användaren.

2. Apparaten

Figur 1: Synthen

Denna apparat, se figur 1, är alltså en synth som tar emot midi-meddelanden och spelar upp syntetiserat ljud som beror på vissa parametrar. Inställningarna ändras genom att röra på de reglage som finns på keyboardet: *modulationshjul*, *volym* och *pitch bend-hjul*. De parametrar som går att modifiera är *resonans, frekvens, filtertyp* och *vågform*.

Keyboardets modulationshjul ändrar filtrets frekvens, volymreglaget ändrar resonansen medan pitchbend-hjulet kan dras helt upp eller ner för att ändra vågform eller filtertyp, beroende på om noten C6 hålls nedtryckt eller inte.

I apparaten finns en pipelinad CPU för att utföra diverse operationer med eget *instruction set*. Till detta instruction set finns även en *kompilator* som utvecklats så att man kan skriva programkoden i språket assembly.

3. Teori

3.1 Subtraktiv synth

En synth[1] är ett musikinstrument som genererar en ljudvåg elektroniskt, antingen analogt eller digitalt. Det finns flertalet olika metoder för att generera detta ljud; *frekvensmodulation*, *additiv syntetisering*, *fasdistortionssyntetisering*, *vektormodulation*, *spektralsyntetisering* för att nämna några. Den vanligaste typen är dock *subtraktiv syntetisering*, vilket är metoden som denna synth använder för att generera ljudet.

Metoden är i grunden relativt enkel, genereringen utgår ifrån en vågform, vanligen en sinus-, fyrkants-, triangel- eller sågtandsvåg. Utöver dessa kan mer exotiska vågformer finnas för att komplementera ljudet. Dessa vågformer spelas upp av synthens *oscillatorer*, minst en, men flertalet kan användas för att få ett mer intressant resultat. Det enda som i detta steg brukar finnas tillgängligt att modifiera är uppspelningshastigheten (dvs transponering) och amplituden.

Hur oscillatorns vågformer genereras är också en sak som varierar mellan olika synthar. För denna implementation genererades vågformer m.h.a Matlab (se appendix D) som sedan sparades in i minnet (s.k. *wavetable*) och varje nots frekvens avgör hur fort uppstegningen i tabellen sker enligt formeln:

$$Vågformssteg = \frac{frekvens * tabelllängd}{samplingsfrevkvens} * steg$$

Där steg ökas med ett för varje sampel. Multiplikationen och divisionen framför steg är en faktor som är konstant för varje enskild not, så dessa genererades i förhand, även dessa m.h.a Matlab (appendix D).

Efter det mixas oscillatorerna ihop och skickas vidare till ett eller flera filter. Det filtrerade ljudet kan sedan ytterligare behandlas om så önskas med *effekter*, vanliga sådana är till exempel eko, rumsklang och *chorus*.

Utöver detta är det vanligt att ha ett flertal *moduleringskällor* tillgängliga, som möjliggör ändringar av parametrar automatiskt. Vanligen *envelopes* (kurvor) och *LFOs* (periodiskt upprepande ändringar).

För att begränsa projektets omfattning har denna synth en oscillator per nedtryckt tangent och 12 s.k. röster, *voices*, det vill säga 12 tangenters *polyfoni* á en oscillator styck. Synthen utnyttjar endast ett filter för tillfället, men detta filter designades så att flertalet filter enkelt kan köras utifrån denna. Inga effekter används heller, då dessa i vissa fall kan vara ett helt projekt i sig!

3.2 SVF-filtret

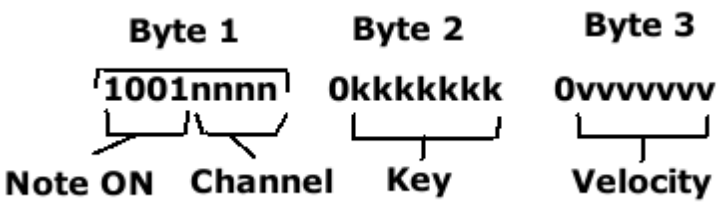
SVF, State Variable Filter, är ett välanvänt filter i synth-sammanhang då de fyra vanligaste filtertyperna direkt finns tillgängliga i en enda krets, *lågpass-*, *högpass*, *bandpass-* och *bandavvisningsfilter*. Ett SVF-filter avtar med 12dB/oktav, dvs ett *tvåpolsfilter*. Filtret kan

sedan implementeras antingen analogt eller digitalt, och i det digitala fallet endast ersätta de analoga komponenterna med de digitala motsvarigheterna till dessa.

3.3 MIDI protokollet

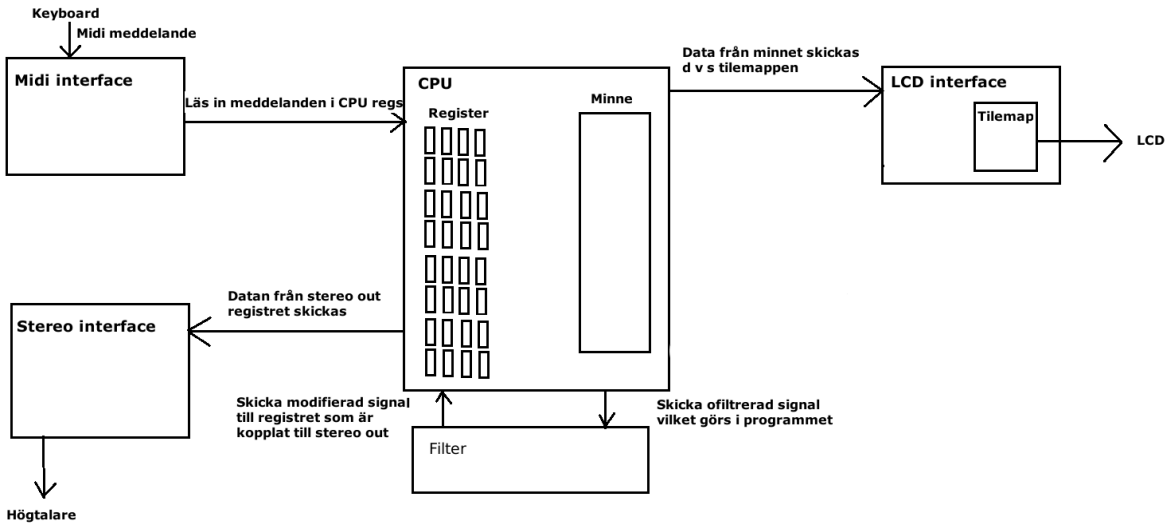
MIDI-meddelanden[2] består (vanligtvis) av 3 bytes varav den första byten beskriver typ av meddelande, s.k. *event*, och *kanalen* som meddelandet skickas på. Kanalen är för denna implementation oväsentlig då synthen lyssnar på alla kanaler, och således är det endast meddelande-typen som är intressant. De två efterföljande bytesen innehåller information specifikt till vilken typ av meddelande det gäller. De meddelandena som synthen behöver lyssna på är *Note Off*, *Note On, Pitch Bend* och *Control Change*. Note On och Note Off följer samma uppbyggnad för de två databytes:en (dvs Byte 2 och Byte 3), först vilken tangent sedan hur hårt tangenten är nedtryckt, *velocity*. Control change har vilken parameter som ändrades i första databyten och det nya värdet i andra databyten. Pitch bend har första byten upptagen av MSByte och andra av LSByte.

Det visade sig i efterhand att keyboardet inte skickade Note Off events utan att de istället skickades ett Note On event men med velocity 0. Ett Note On meddelande ser ut enligt figur 2.



Figur 2: Visar hur Note On meddelandet ser ut

4. Hårdvaran



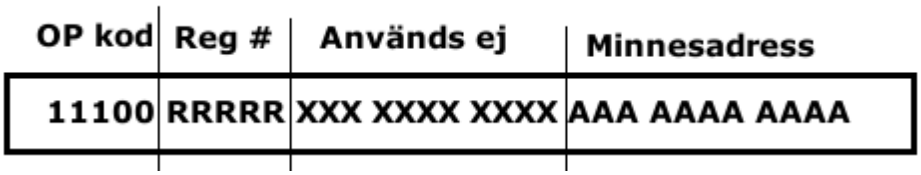
Figur 3: Figuren visar hur de olika delarna interagerar

Figur 3 visar en överblick av alla de olika komponenterna synthen är uppbyggd av och hur de olika komponenterna kommunicerar med CPU:n genom register. Beskrivningen nedan

hänvisar då och då till de olika delarna i CPU:n vid namn, dessa kan finnas i appendix E figur 1.

4.1 CPU

CPU:n är modellerad efter den *pipelinade* datorn som beskrevs under föreläsningarna[3]. Det är en 4-stegs pipelinad CPU med *dataforwarding* och *stall* logik samt ett eget instruction set. En lista med alla instruktioner återfinns i appendix B. Det enda generella för alla instruktionerna är att de första 5 bitarna består av OP-koden. Resten av bitarnas funktion varierar från operation till operation, figur 4 visar hur t.ex. instruktionen för LOAD.a ser ut.



Figur 4: Visar 32-bitars ord för CPU:n

Instruktioner består av 32 bitars ord vilket ger vissa begränsningar. Bitarna utnyttjas så att 2^5 st olika instruktioner, 2^5 st generella register och 2^{11} st minnesplatser möjliggörs. För ALU-operationer används instruktionerna ALU.c och ALU.r som har en speciell OP kod som är en förlängning av OP-koden för att få plats med tillräckligt många instruktioner och ge plats för eventuella utökningar. Mer om dessa instruktioner återfinns i appendix B. CPU:n går dessutom med en klockfrekvens av 100MHz. Schematisk bild av CPU:n finns i appendix E, schema 1.

4.1.1 Pipeline

Pipelinen är i 4 steg och består främst av registerna IR1, IR2, IR3 och IR4. Instruktioner från programminnet laddas in i IR1 varje klockcykel och från IR1 till IR2 o.s.v. tills instruktionen når IR4. I klockcykeln efter att instruktionen lästs in i IR4 slängs instruktionen bort. Varje pipelinesteg består av lite mer än dessa register, som exempel tillhör IR2, B2 och A2-registerna steg 2 i pipelinen. Samma mönster följs i resterande steg i pipelinen.

4.1.2 Programminnet, programräknaren och stalling

Programminnet är som ovan nämnt 32-bitar brett med 512 platser tillgängliga att skriva programkod. Programräknaren pekar hela tiden på nästa instruktion som ska laddas in i IR1, då instruktioner hela tiden förhämtas. Programräknaren stegar hela tiden uppåt i programminnet så länge ett hopp ej inträffar, med undantag när *stall* sker.

IR1 läser hela tiden in värdet från programminnet om inte IR2 laddar från minnet och instruktionen som flyttats in i IR1 är en instruktion som använder det register som laddas från IR2 instruktionen, då hålls istället IR1 kvar och en NOP instruktion skickas ut från IR1 istället, en CPU stall sker.

4.1.3 Hopp

Om IR2 innehåller en hoppinstruktion beräknas nya programräknaren i PC2, nästa klockpuls blir programräknaren detta värde, om ett hopp faktiskt ska ske (dvs flaggor matchar om hoppet är ett villkorligt sådant).

4.1.4 Registerområdet

Här återfinns alla 14 generella register samt de interna register som CPU:n använder för att kommunicera med de andra delarna av synthen. Ett generellt register är 16 bitar. Här finns dessutom statusregistret inbyggt, med ALU flaggorna zero, negative, carry och overflow. Dessutom finns flaggor för inbyggda timrar och midi-meddelande inkommet flaggan som nollställs vid läsning och finns i samma register som ALU flaggorna. Detta område ser till att rätt data läses in i register A2 och B2 (se figur 1 appendix E) vid nästa klockcykel beroende på vilken instruktion som ligger i IR1. Dessutom läses data in till register från MUX:en efter Z4 och D4 registren i nästa klockcykel beroende på vilken instruktion som ligger i IR4.

Det finns även speciella register som används som interna timers som kan laddas in med ett värde, räknar ned 1 varje klockcykel och sätter tillhörande flagga i statusregistret då den räknat ned till 0. En special-timer är “kort-timer 1” som alltid startar om då den når 0.

SVF-filtret har även sina register här, dvs ingång, frekvens, resonans, fördröjning 1, fördröjning 2, så har även ljudutgången som hämtar det nya värdet från ljudutgångsregistret, *Audio Out*.

Övriga speciella register är de 3 register som skrivs till av MIDI gränssnittet och sätter tillhörande flagga vid skrivning.

4.1.5 Dataforwarding

Dataforwarding består av alla de muxarna som sitter innan ALU:n och den interna logiken för den. Denna ser till att om instruktioner har databeroende så forwardas data till ALU:n direkt från register D3 eller D4/Z4-muxen istället för att komma från det databeroende registret, detta för att spara dyrbara klockcykler som skulle gå åt för att vänta på att värdena skulle hamna i registret annars. Det som sker principiellt är att om en instruktion som berör ett register följs av en instruktion som berör (tex läsning) det register inom två klockcykler så kan inte värdet hämtas från registret för att det tar tre klockcykler innan det “rätta värdet” laddas in i registret så datan hämtas direkt från pipelinen via muxar och logik. Denna dataforwarding-koll måste göras för alla instruktioner som ändrar i register och följs av instruktioner som läser från det berörda registret.

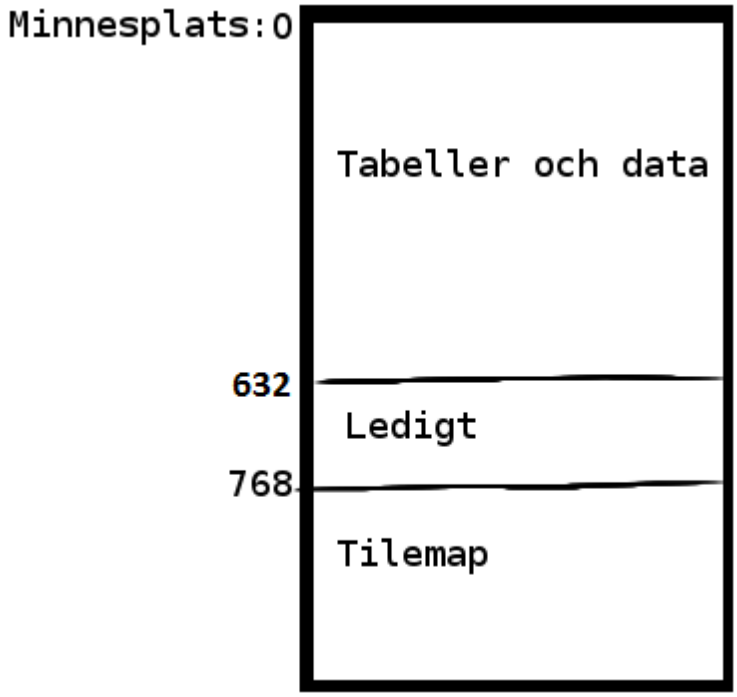
4.1.6 ALU

ALU:n är beräkningsenheten. De direkta instruktionerna i pipelinen läses av och beroende på instruktionen skickas en intern instruktion till ALU:n. Denna interna instruktion består av den förlängda delen av OP koden för vanliga ALU instruktioner såsom ALU.c eller vid andra fall såsom STORE.c så används en intern instruktion som låter värdet (i detta exempel adressen som det skall lagras till i minnet) gå rakt genom ALU:n. ALU flaggor styrs givetvis från ALU och för hur dessa sätts specifikt så hänvisas läsaren till appendix B.

4.1.7 Dataminnet

Minnet är 16x1024 bitar och är uppdelat i olika områden. Generellt sett så är det uppdelat i 2 delar, första delen består av tabeller av data för noter, vågormer och voices medan den andra delen består av *tilemap*-minnet som alltså mappar vilka *tiles* som ska visas var på

LCD-skärmen. Mer om tilemap och tiles under rubriken LCD-skärm. Figur 5 visar hur minnet är uppdelat med index.



Figur 5: Dataminnet

All läsning och skrivning som CPU:n gör till minnet sker synkront men samtidigt sker asynkron avläsning av tilemappen för att skicka data till LCD-gränssnittet. Specifikt så är första delen av minnet implementerat som blockRAM på nexys-kortet medan tilemap-delen inte är det.

Data som skall läsas in i minnet läggs i register Z3 medan adressen som skall sparas till eller läsas från alltid läggs in i register D3. Vid läsning så läggs sedan den lästa datan i Z4. Se schema 1 i appendix E för skiss över dessa register.

4.2 MIDI input

Information om vilka knappar som trycks in på keyboardet skickas seriellt (UART) med meddelanden på 10 bitar styck. Ett meddelande inleds alltid med startbit 0 och avslutas med stoppbit 1, och däremellan kommer en MIDI-byte på 8 bitar. När ett helt meddelande har lästs in skickas en puls till en del av datorn som sparar in byte:n i ett av tre register (Mreg1-3, schema 4 appendix E) eftersom att ett meddelande består av 3 bytes.

Då keyboardet fungerar på så sätt att om en liknande tangent trycks ner flera gånger i snabb följd så skickas bara byte 2 (key info) och byte 3 (velocity), så måste datorn hålla reda på det senaste meddelandet (första byten) tills ett ny meddelandetyp inkommit; så att bytes:en hela tiden sparas in i rätt register.

När hela meddelandet har lästs in i registerna undersöks om det är ett av de meddelanden som är intressanta, och om så är fallet sätts en flagga i statusregistret.

4.3 Stereo output

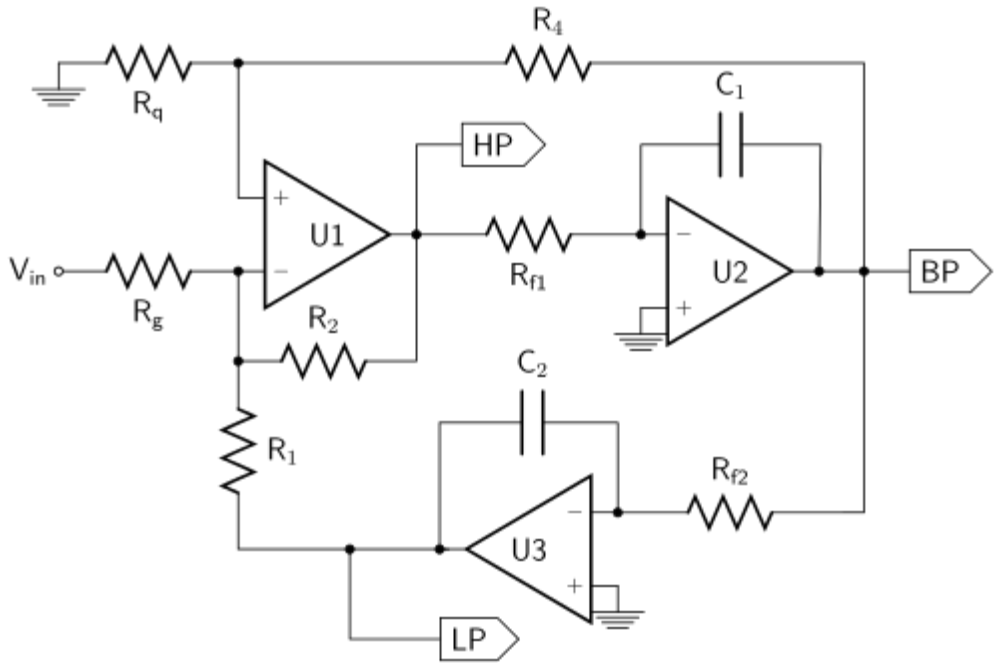
Ljudutgången är en PmodI2S som det seriellt skiftas ut ljudsampels på 16 bitar i 48,828 kHz. Den drivs av tre klockor (MCLK, LRCK och SCLK) som alla genereras av datorn. Varje gång SCLK går hög skiftas en bit ut, och när LRCK är hög respektive låg skiftas samplet för vänstra respektive högra kanalen ut. Eftersom att denna synth enbart opererar på en kanal (dvs mono) så skickas samma sampel ut på båda kanalerna. Detta innebär att SCLK går exakt 32 klockcykler per LRCK, och LRCKs frekvens är 48,828 kHz.

När ett helt sampel är utskiftat läses nästa sampel in från ett internt register i datorn. Detta händer kontinuerligt när synthen är på, oavsett om det är något ljud som ska spelas upp, och om inget ljud ska spelas upp är registret helt enkelt 0.

4.5 Filtret

Filtret som implementeras är som nämnt i teoriavsnittet ett SVF-filter. Implementeringen följer relativt rakt av från hur SVF-filtret ser ut, se figur 6 för den analoga motsvarigheten. En version med de analoga komponenterna utbytta till de digitala motsvarigheterna utnyttjades dock för att inte uppfinna hjulet igen. Figur 1 från National Instruments *"Implementation of Digital Filters as Part of Custom Synthesizer with NI SPEEDY 33"* [4], där ett SVF-filter implementeras på ett *DSP-kort*, användes som utgångspunkt för att designa filtret i VHDL.

Filtrets olika parametrar sparas i register för att kunna utnyttja synthens enda filter för att simulera att flera filter existerar, då ett enskilt filter endast behöver köras en gång per sampel, och tar fyra klockpulser för en genomkörning. De register filtret använder är ingång, fördröjning 1, fördröjning 2, frekvens, resonans och utgång. När ingången laddas så körs filtret en gång, så detta register måste alltid vara det register som laddas sist, efter de andra registren.



Figur 6: Analogt SVF-filter

4.6 LCD-skärm

En schematisk bild över LCD-gränssnittet finns i appendix E, schema 2. LCD-skärmen klockas på 10 MHz, så data skickas då alltså med denna klockfrekvens, Data Enable signalen och 24-bitar färgdata i RGB-form. För att uppnå detta så används en x- och en y-räknare som räknar koordinaterna på skärmen, dessa används sedan till att se till att rätt data skickas till LCD-skärmen vid rätt klockpuls. De 6 högre bitarna av x- och y-räknarna används för att peka rätt plats i tilemappen som i sin tur pekar ut i tileminnet vilken slags tile som skall ritas ut. Varje tilemap består av 5 bitar (detta ger plats för 32 tiles) och vilken tile som ligger på en viss plats i tilemappen sparas parvis i dataminnet. T.ex. plats 768 i minnet pekar alltså ut tile 1 och tile 2. Eftersom en plats i tilemappen består av 16 bitar är de sparade som i exemplet så att tile 1 ligger i byte 1 och tile 2 i byte 2 o.s.v. Som nämnt under underrubriken Minne så är tilemappen en del av minnet som CPU:n kan bearbeta men tileminnet är statiskt och består av 32 tiles.

De lägre 4 bitarna från räknarna skickas till tileminnet genom GK1 och GK2 för att internt i tilen välja ut rätt pixel data. Tiles består alltså av 16x16 pixlar styck, skärmen består av 480 pixlar i x-led, 272 pixlar i y-led vilket ger 480/16 = 30 tiles i x-led och 272/16 = 17 tiles i y-led. Detta innebär att varje pixel innehåller information om vilken färg som skall visas och eftersom det finns två olika värden en bit kan anta så innebär det att 2 olika färger kan användas som i detta fall är vit och svart.

Rätt pixel data skickas sedan vidare till GK3 som skickar vidare till LCD:n, x- och y-räknarna räknar högre än antalet pixlar i x- respektive y-led så att som exempel om x-räknaren går över 480 så skickas ingen färgdata från GK3 i blockschemat.

Det viktigaste med att räknarna räknar över antalet pixlar (innan de slår om till 0) på skärmen är att GK4 som skickar DE till LCD:n ser till att DE går låg efter en hel rad har skickats (dvs då x-räknaren räknar över 480) till LCD:n en viss tid enligt tillverkarens specifikationer. För dessa specifikationer hävisas läsaren till referens 2 i referenslistan. Att nämna är att bakgrundsljuset till LCD:n drivs med en klocka på 50 kHz som motsvarar full ljusstyrka. Dessutom så kräver LCD:n en speciellt startup sequence som det också står närmare om i manualen för skärmen[5].

5. Programkod

5.1 Mainloop

Det första som händer när programmet startar är en del setup som till exempel uppstart av timer:n för att skicka ut nästa sampel samt nollställning av diverse minnesplatser. Därefter går programmet in i en oändlig mainloop varifrån olika flaggor i statusregistret avläses, och hopp görs när det är dags. Dessa hopp fungerar som subrutiner, vilket innebär att när det som ska beräknas är färdigt så hoppar programmet tillbaks till raden efter där hoppet utfördes ifrån.

5.2 MIDI-input

Varje gång en viss flagga i statusregistret går hög så hoppar programmet in i denna del som avläser vilket meddelande som kommit in och utför olika operationer beroende på meddelandet.

Det första som kollas är om det är ett Control Change eller Pitch Bend-meddelande, vilket innebär att någon parameter till synthen ska modifieras. Om så är fallet hoppar programmet till koden som hanterar det aktuella meddelandet, och sparar in rätt värden i minnet/register.

Om det inte handlar om någon inställning så betyder det att det är ett Note On-meddelande vilket innebär att en not antingen ska börja eller sluta spelas (undantag om noten är C6 vilket innebär att vilken vågform som används ska ändras). Om så är fallet kollas först om velocity är noll eller inte, och om den är noll utförs ett hopp till delen som tar hand om att sluta spela notes. Där utförs en iteration över minnesplatserna för var noter kan vara sparade och om samma not som finns i det aktuella MIDI-meddelandet hittas så nollställs den minnesplatsen.

Om velocity inte är noll innebär det att en ny not ska sparas in i minnet, och för att hitta en ledig plats itereras även här över de minnesplatser där noter kan sparas, och om en ledig plats hittas så sparas information om noten in där. Om ingen av platserna är ledig betyder det att max antal noter redan spelas (en konstant i koden) och meddelandet ignoreras.

5.3 Ljudutskickning

Programmet hoppar hit med 48,828 kHz vilket är samma hastighet som ljudet skickas ut med från ljudkretsen (alltså den hastigheten nästa ljudsampel behöver beräknas i).

En iteration utförs över alla notplatser i minnet och om noten inte är noll betyder det att den spelas, och med hjälp av en konstant för varje not som finns i minnet räknas det ut vilket steg (0 - 63) av den nuvarande vågformen som ska skickas ut. Alla delsampels från noterna som spelas adderas ihop, och skickas sedan genom filtret och vidare till ljudutgången PmodI2S.

För att hämta rätt konstant från minnet används notens nummer som offset, vilket är väldigt smidigt då alla noternas konstanter ligger på rad i minnet (C0 - B5).

5.4 Uppdatering av skärmen

Skärmen uppdateras ungefär varje 6.5 ms. Det är fyra parametrar som kan ändras och därför orsakar att bilden på skärmen skall ändras. Först kollas vilken vågform som skall skrivas ut genom att kolla registret som sparar nuvarande vågform och sedan skrivs motsvarande vågnamn ut genom att ändra i tilemappen. Liknande sker för utskrivning av filtertyp.

Data om frekvens och resonans finns i dataminnet och hämtas alltså därifrån för att sedan maska ut de 8 högsta bitarna och lägga dessa i tilemappen. Detta fungerar väldigt smidigt tack vare att karaktärerna för 0 - 9 och A - E ligger uppradade från plats 0 i tileminnet.

6. Kompilator

6.1 Användning

För att underlätta programmeringen av datorn används en enkel kompilator skriven i Python. Det finns inbyggt stöd för diverse olika saker så som konstanter, kommentarer, tolkning av tal av olika baser med mer (se appendix C).

För att kompilera dokument med kod i måste först Python startas och programmet importeras med:

```
>> from compile import *
```

Sedan kompileras filen med:

```
>> comp_file("fil1.txt", "fil2.txt", …, "filn.txt" )
```

6.2 XML

Kompilatorn använder sig av *XML* vilket gör det väldigt enkelt att lägga till fler instruktioner till kompilatorn. Det enda som behöver göras är att lägga in ytterligare ett element i filen instructions.xml.

En instruktion följer följande format (exempel på instruktion med 3 argument):

```
<instr name="LOAD.O">
      <OP>11110</OP>
      <DEST>
            <LENGTH>5</LENGTH>
      </DEST>
      <SRC>
            <LENGTH>11</LENGTH>
      </SRC>
      <OFFSET>
            <LENGTH>11</LENGTH>
      </OFFSET>
</instr>
```

7. Slutsatser

Projektet blev något större än vad som var planerat, eller snarare att de problem som uppstod tog upp så mycket tid som det gjorde. Ett exempel är fel som inte uppstod i simulering men dök upp i verkligheten. Dock har utvecklingsprocessen av synthen varit väldigt lärorik, med många nya utmaningar som har lösts; allt från små knep i VHDL till generering av ljud.

8. Bortprioriterad funktionalitet

En sak valdes bort från synthen som från början var planerad att finnas med, att istället för att ändra inställningar från keyboardet utnyttja skärmens touch. Att detta valdes bort var helt enkelt på grund av tidsbrist när andra saker istället strulade. Att det blev just touch som valdes bort var att det mest var en häftig gimmick, men inte väsentlig för att få synthen till de krav som annars ställts upp, då projektets grundtanke var att skapa en synth med parametrar som går att modifiera.

Projektet hade medvetet en stor omfattning, och således fanns det en backup plan med i designspecifikationen om det visade sig att omfattningen blev allt för stor, dock var inte just touch en sak som stod med där, men efter rådfrågning med handledare valdes denna bort istället för en annars sämre fungerande synth.

9. Möjliga utökningar

Det finns flertalet utökningar som relativt enkelt kan läggas till utan att behöva ändra någon VHDL-kod och endast skriva programkod. Ett urval av dessa är:

- Flera oscillatorer per röst
- Flera filter
- Kunna skicka de olika oscillatorerna till olika filter
- Moduleringskällor
- Effekter

10. Referenser

1. Richard Boulanger(red.), Victor Lazzarini(red.). 2000. *The audio programming book*, (ISBN: 978-0262014465)
2. Summary of MIDI Message http://www.midi.org/techspecs/midimessages.php (Hämtad 2015-05-25)
3. Föreläsningsunderlag TSEA83 Pipeline 2015 http://www.isy.liu.se/edu/kurs/TSEA83/tex/lec_pipe.pdf (Hämtad 2015-05-25)
4. Implementation of Digital Filters http://www.ni.com/white-paper/3476/en/ (Hämtad 2015-05-25)
5. Digilent User manual VmodTFT http://www.digilentinc.com/Data/Products/VMOD-TFT/VmodTFT_rm.pdf (Hämtad 2015-05-25)

11. Bildreferenser

1. Figur 6: State Variable Filter.svg under CC BY-SA 4.0 licens http://en.wikipedia.org/wiki/File:State_variable_filter.svg

Appendix

A. VHDL-kod

VHDL-kod bifogas. Koden är uppdelad i flertalet mappar, där varje mapp innehåller de filer som är specifika för en “komponent”, så som CPU:n, filtret osv.

B. CPU instruction set

STATUSFLAGGOR:
Zero, Negative, Carry, Overflow, Long Timer finished counting, Short Timer 1 finished counting, Short Timer 2 finished counting, Midi Ready

14 allmänna regs.
1 statusregister
16 extra regs.
Adresser 11 bitar.

Generellt gäller att:
Destination skrivs direkt efter OP koden, source i slutet

--- Instruktioner ---
NOP
00000 XXX XXXX XXXX XXXX XXXX XXXX XXXX

TRAP
00001 XXX XXXX XXXX XXXX XXXX XXXX XXXX

STORE.c
11000 AAA AAAA AAAA DDDD DDDD DDDD DDDD

STORE.r
11001 AAA AAAA AAAA XXXX XXXX XXXR RRRR

STORE.o
11010 AAAA AAAA AAA CCCC CCCC CCCR RRRR

STORE.or
11011 AAAA AAAA AAA OOOO O XXXX XX R RRRR
(O = Register with offset data)
LOAD.a
11100 RRRR RXXX XXXX XXXX AAA AAAA AAAA

LOAD.c
11101 RRRR RXX XXXX DDDD DDDD DDDD DDDD

LOAD.o
11110 RRRR RCCC CCCC CCCC AAA AAAA AAAA

LOAD.or
11111 RRRR R OOOO O XXXX XX AAA AAAA AAAA

MOVE

```
00100 RRRR RXX XXXX XXXX XXXX XXXR RRRR

-----------------------------------------

BRA
01000 XXXX XXXX XXXX XXXX AAA AAAA AAAA

BRA.OR
01001 XXX XXXX XXXX XXXX XXXX XXXR RRRR

BRA.O
01010 XXX XXXX XXXX XXXX XXOO OOOO OOOO

BCC
10000 SSSS XXXX XXXX XXXX AAA AAAA AAAA
(S = status flag)

BCC.OR
10001 SSSS XXX XXXX XXXX XXXX XXXR RRRR

BCC.O
10010 SSSS XXX XXXX XXXX XXOO OOOO OOOO

BNCC
10100 SSSS XXXX XXXX XXXX AAA AAAA AAAA
(S = status flag)

BNCC.OR
10101 SSSS XXX XXXX XXXX XXXX XXXR RRRR

BNCC.O
10110 SSSS XXX XXXX XXXX XXOO OOOO OOOO


---------- ALU ------------------------


ALU.r
00101 OOOOO RRRR RX XXXX XXXX XXXR RRRR

ALU.c
00110 OOOOO RRRR RX DDDD DDDD DDDD DDDD


O = Operation
IF = Ingen förändring
```

`00000: do nothing`
Påverkade statusflaggor:
Z : IF
N : IF
C : IF
O : IF

`00001: ADDU unsigned`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : IF
C : 1 om additionen orsakar carry, annars 0
O : IF

`00010: ADD signed`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : IF
O : 1 om två positiva tal resulterar i ett tal med signbit 1(eller vice versa med 2 negativa tal), annars 0

`00011: SUBU unsigned`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om source < destination dvs indikerar att man använd operationen felaktigt, annars 0
C : 1 om additionen orsakar carry, annars 0
O : IF

`00100: SUB signed`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : IF
O : 1 om (s-d) < 0 om s > 0 och d < 0 eller (s-d) > 0 om s < 0 och d > 0 annars 0

`00101: MUL signed(fixed point)`
Påverkade statusflaggor:
Z : 1 om den övre hälften av bitar alla är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : IF
O : IF

`00110: BSR bitshift right`

Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : =sista biten som shiftades ut(när det gäller de fallet då man skiftar ut 0 bitar 0-ställs carryn)
O : IF

`00111: BSL bitshift left`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : =sista biten som shiftades ut(när det gäller de fallet då man skiftar ut 0 bitar 0-ställs carryn)
O : IF

`01000: AND`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : IF
O : IF

`01001: OR`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : IF
O : IF

`01010: XOR`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : IF
O : IF

`01011: NOT`
Påverkade statusflaggor:
Z : 1 om resultatet är 0 annars 0
N : 1 om resultatets sign bit är satt, annars 0
C : IF
O : IF

`01100: CMPU(unsigned)`
Påverkade statusflaggor:

Z : 1 om x-y=0 annars 0
N : 1 om x < y annars 0
C : 1 om carry genereras av x-y
O : IF

`01101: CMP(signed)`
Påverkade statusflaggor:
Z : 1 om x-y=0 annars 0
N : 1 om x < y annars 0
C : IF
O : set overflow if x-y results in overflow

`01111: BITTEST`
#0-15(tar in unsigned)
Påverkade statusflaggor:
Z : 1 om biten är 0, 0 annars eller om talet som skickas in är >=REG_WIDTH
N : IF
C : IF
O : IF

`10000: ADDX addera utan att ändra statusflaggor(unsigned)`
Påverkade statusflaggor:
Z : IF
N : IF
C : IF
O : IF

`11111: reserverad för att låta leftIn gå igenom`
Påverkade statusflaggor:
Z : IF
N : IF
C : IF
O : IF

C. Assembly syntax

```
########################################
#                                      #
#     Syntax för compiler till synth   #
#                                      #
########################################

INSTR ARG1, ARG2...
Det är ICKE case-sensitive

Man kan skriva "ARG1,,,,,,,,   ,,,,,,,,,,ARG2"


################
# Kommentarer  #
################

Kommentarer skrivs med "instruction ; kommentar goes here" (utan "") och
måste (currently) skrivas EFTER en rad kod (Man kan alltså inte ha en
rad med bara en kommentar)
Ex:
MOVE $A4, 1001 ; Exempelkommentar


################
#     Hopp     #
################

För att skapa konstant för hopp (skapar konstant vars värde blir rad#)
&jumphere

För att hoppa:
BRA #jumphere

Eftersom att det är konstanter måste man se upp med att ha vanliga
konstanter med samma namn.


################
#    Talbas    #
################

$F    -- Hex
```

```
%1111 -- Bin
15    -- Dec

Instruktioner med 2 argument:
INSTR DEST, SRC

Instruktioner med 3 argument:
INSTR DEST, SRC, OFFSET


################
#  Konstanter  #
################

För att skapa konstanter:
CONSTANT name, value

För att använda konstanter:
INSTR, #konstant1, #konstant2

Fördefinerade konstanter:

   # Status flags
   "SR_Z":      "0"
   "SR_N":      "1"
   "SR_C":      "2"
   "SR_O":      "3"
   "SR_LT1":    "4"
   "SR_ST1":    "5"
   "SR_ST2":    "6"
   "SR_MIDI":   "7"

   # General registers
   "R_G0":      "0"
   "R_G1":      "1"
   "R_G2":      "2"
   "R_G3":      "3"
   "R_G4":      "4"
   "R_G5":      "5"
   "R_G6":      "6"
   "R_G7":      "7"
   "R_G8":      "8"
   "R_G9":      "9"
   "R_G10":    "10"
   "R_G11":    "11"
   "R_G12":    "12"
```

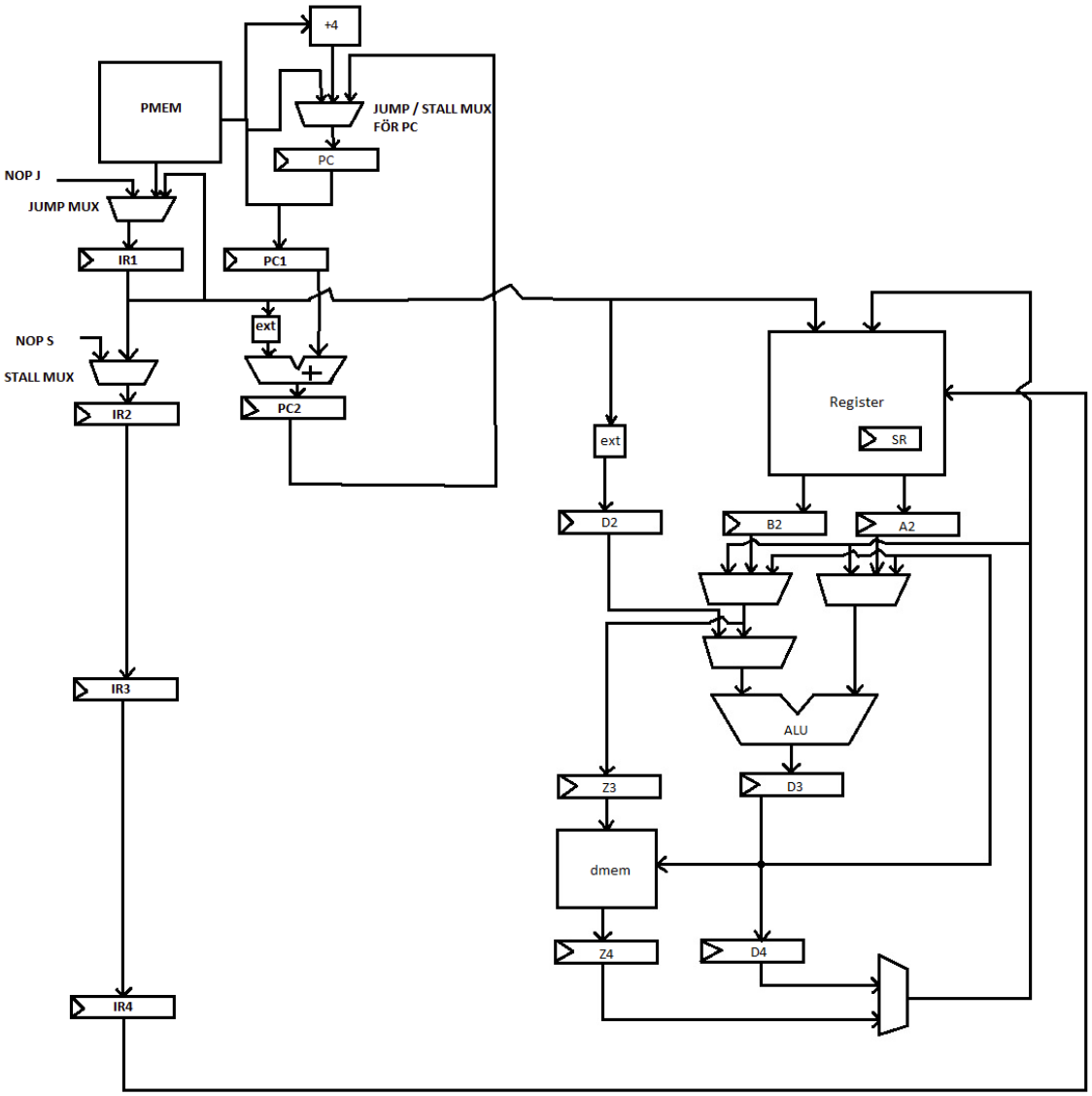
```
"R_G13":    "13"

"R_SR":     "14"
"R_LT1_L":  "16"
"R_LT1_H":  "17"
"R_ST1":    "18"
"R_ST2":    "19"
"R_TOUCHX": "20"
"R_TOUCHY": "21"
"SVF_IN":   "22"
"SVF_D1":   "23"
"SVF_D2":   "24"
"SVF_OUT":  "25"
"SVF_F":    "26"
"SVF_Q":    "27"
"SVF_SETUP":"28"
"R_MREG12": "29"
"R_MREG3":  "30"
"R_AUDIO":  "31"
```

D. Matlab script

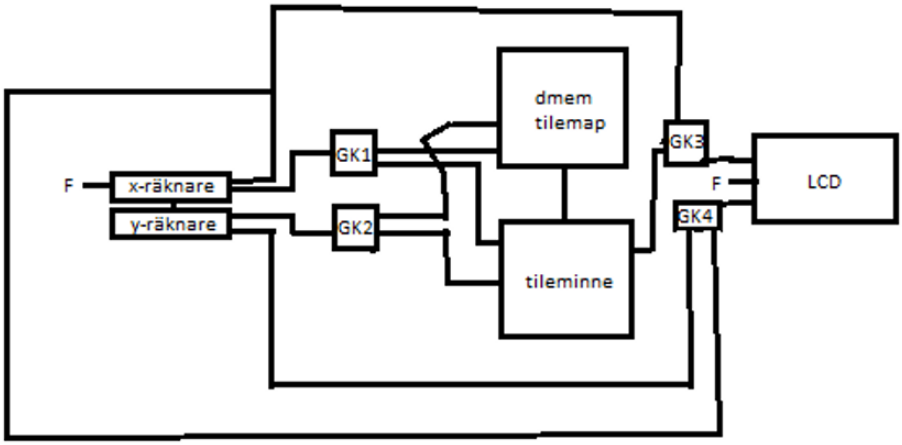
```
%% Wavetable gen - setup
N = 64;
n = 1 : N;
MAX = 2^16 - 128;

%% Generate
square = floor((MAX .* (n > N/2)) - MAX/2);
sinoid = floor(MAX/2 .* sin(2*pi*n / N));
saw = floor((MAX .* (N - n) / N) - MAX/2);
tri = (2 * abs(saw)) - MAX/2;

%% Print
fprintf('-- %d step square wave\n', N);
for i = 1 : N
      fprintf('("%016d"), \n',
str2num(dec2bin(typecast(int16(square(i)),'uint16'))));
end
fprintf('-- %d step sin wave\n', N);
for i = 1 : N
      fprintf('("%016d"), \n',
str2num(dec2bin(typecast(int16(sinoid(i)),'uint16'))));
end
fprintf('-- %d step saw wave\n', N);
for i = 1 : N
      fprintf('("%016d"), \n',
str2num(dec2bin(typecast(int16(saw(i)),'uint16'))));
end

fprintf('-- %d step tri wave\n', N);
for i = 1 : N
      fprintf('("%016d"), \n',
str2num(dec2bin(typecast(int16(tri(i)),'uint16'))));
end

%% Frequency constants
% Not completely accurate, but sufficient for our need
freqs = [16.35, 17.32, 18.35, 19.45, 20.60, 21.83, 23.12, 24.50, 25.96,
27.50, 29.14, 30.87];
freq = (1:7*12);
for k = 0:6
      for j = 1:12
      freq(j+12*k) = freqs(j) * 2^k;
      end
end
```

```
res = (freq * 64) / 48828;

for r = 1 : length(res)
   a = fi(res(r), 0, 15, 13);
   fprintf('("0%s"),\n', a.bin);
end
```

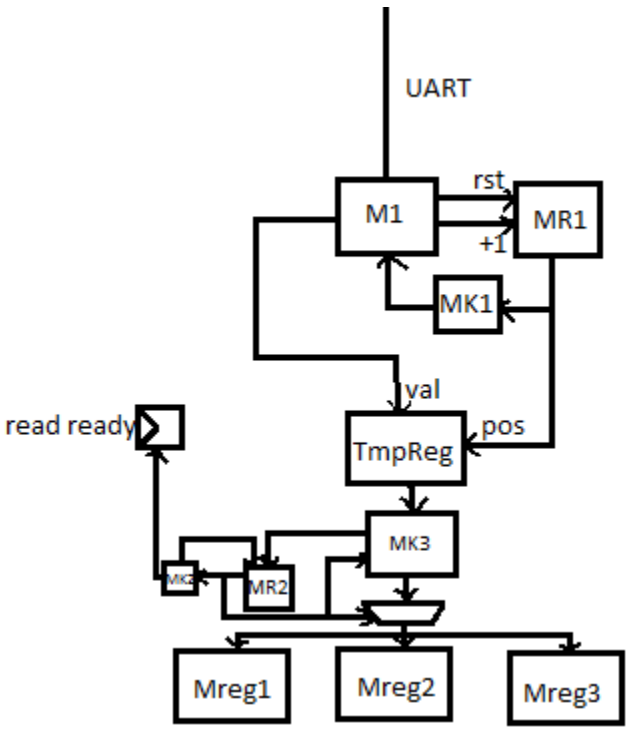
E. Scheman



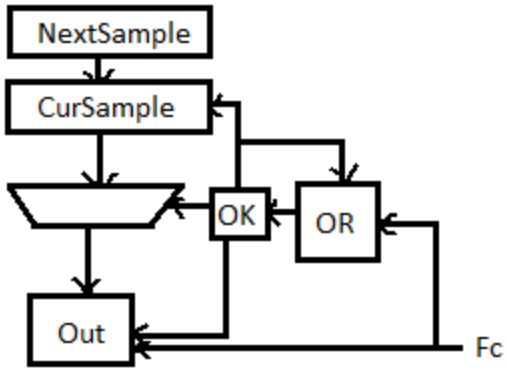
Schema 1: Schematisk bild över CPU:n.



Schema 2: Blockschema över LCD-interface.



Schema 3: Blockschema för inläsning av keyboard.



Schema 4: Blockschema för hanteringen av ljud-output.