Project Report
Realtime DSP Sound Effects
Embedded System Design - Advanced Course
EDA385

Stefan Grandlund et07sg8@student.lth.se
Anders Skoog et08as9@student.lth.se
Fredrik Stolt et08fs1@student.lth.se

October 26, 2012

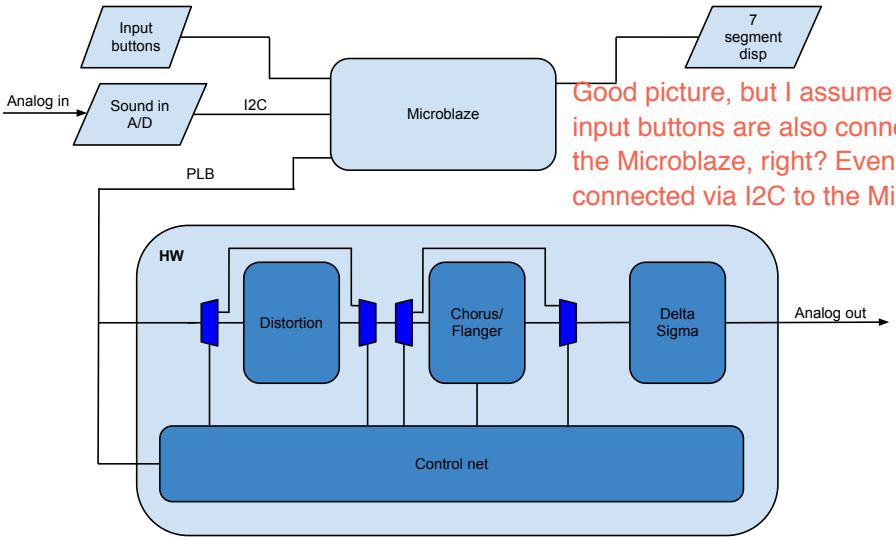
1 Abstract

This report describes how five real time sound effects were implemented on a FPGA with the help of digital signal processing methods. The architecture consists of one Microblaze CPU which handles the user interface and three of the sound effects and some custom hardware that handles the other two effects and the DA conversion.

Contents

2 Project overview

This project implements sound effect on a Nexys 3 FPGA board. All this is done in real time with analog input and output. Five different sound effects are available: echo, bit crush, pitch shift, distortion and chorus/flanger. To turn on different effects the switches on the board are used. To supply parameters for the effects the directional buttons are used together with the seven segment display.



Good picture, but I assume the 7 seg and input buttons are also connected via PLB to the Microblaze, right? Even Sound in is not connected via I2C to the Microblaze.

Figure 1: Block diagram of code.

Figure 1 shows an overview of how it is implemented on the FPGA. A microblaze processor is used to run the software part. The software is responsible for communication with the ADC, polling the buttons/switches and updating the seven segment display. The processing of the signal for echo, bit crush and pitch shift is also done in software. After the software is done processing a sample it sends it on to the hardware via the PLB bus. When the user make changes the software also sends control data about the effects to the hardware via PLB.

The hardware then apply the effects for distortion and chorus/flanger before it converts the signal to a stream of zeros and ones with a delta sigma converter. The signal is lastly amplified and filtered to an analog output with the PmodAMP1 module.

The software is run on microblaze CPU instantiated from XPS. Also instantiated from XPS is the seven segment controller,GPIO ports for the LEDs and buttons, I2C bus interface and PLB interface. The hardware blocks shown in figure 1 is custom written for this project.

How the FPGA resources are divided between the different ip-cores can be seen in table 1.

IP-core	Flip Flops used	LUTs used
The Whole system	7314	5576
Xps iic	385	470
Custom Hardware	3666	1638
Microblaze	3263	3468

Table 1: FPGA slice and LUT allocation

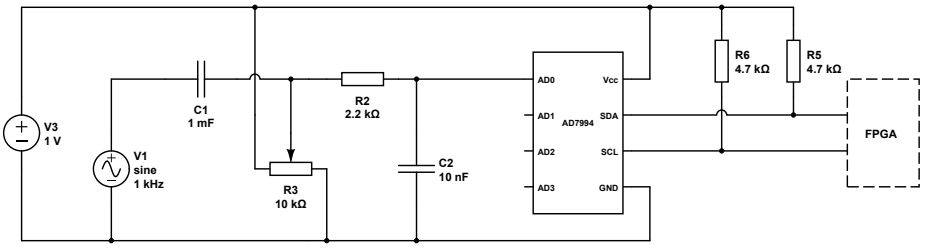


Figure 2: Analog input circuitry for ADC.

3 ADC

3.1 Digital hardware and I2C interface

The AD-converter for this project was supplied on a module from Digilent called PmodAD2. The AD-converter chip, from Analog Devices called AD7994, uses an I2C-bus for transferring the data to the FPGA.

The ADC has an maximum sample rate of 3.4 MHz and a guaranteed 12 bit sample resolution up to 1.7 MHz. To reach those high sample rates with I2C using it in high speed mode (3.4 Mbit/s) is needed. Unfortunately the I2C ip-core available for the microblaze only supports Fast mode (400 Kbit/s) which, if you add the overhead of writing and reading to the I2C ip-core, gives us a sample rate of only ~ 8 KHz. According to the Nyquist sampling theorem that gives the ability to sample signals up to ~ 4 KHz.

Since a sample rate of 8 KHz gave bad sound quality and high speed mode was not available a decision to try and overclock the I2C was made. After some trial and error a clock speed of 1 MHz was chosen for the I2C-bus and the timer interrupt in software controlling the sample speed was set to give us a sample speed of 16666 Hz. The reason for choosing that sample frequency is that it gives a good balance between the memory required in the effects and the audio quality.

3.2 Analog Hardware

Since the ADC used in the project works between 0 V and 3.3 V and the sound signal usually is represented with a signal between -1 V and 1 V a bias stage is needed and a low pass filter to avoid aliasing when sampling as seen in figure 2.

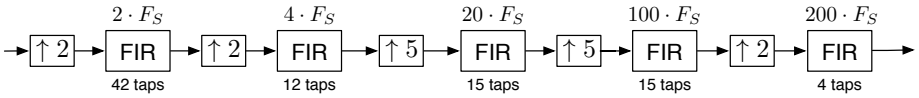


Figure 3: Upsampling of signal for DAC.

3.3 Software

The software controlling the ADC has two parts, a setup part and a sampling part. The setup part consist of writing a value to a register in the ADC controlling which of the input channel to use. The sample part consist of a timer interrupt that is set to fire at the desired sample rate. When the timer interrupt is fired the software writes the address of the AD-converter together with the write bit to the I2C ip-core. The AD-converter responds by sending back two bits containing which channel is being read and the sample value. The data is saved in a byte array, to be able to use the sample data saved in the array, the sample is masked out and concatenated into a 16 bit variable and the DC biasing added by the analog circuit is removed. The sample can then be processed by the different hardware and software sound effects.

4 DAC

As output the module PmodAMP1 from Digilent was used. It amplifies the signal from the FPGA board and outputs it to a headphone jack. It also has a built in band-pass filter for the signal with high pass cutoff frequency of 150 Hz and low pass frequency of 8 kHz which makes it suitable for a PWM (pulse width modulated) signal. The PWM signal is digital so the output is only high and low voltage but at a higher frequency then the original signal. It contains the original signal plus a lot of higher frequency noise that is all ideally filtered away. The simplest approach to generate a PWM signal is to compare the signal to a counter which isn't good enough when using twelve bits so instead we used a Delta Sigma modulator.

4.1 Delta Sigma Converter

The Delta Sigma Converter can be used for both analog to digital and for digital to analog conversion. When using it for digital analog conversion it takes a low frequency and high resolution signal as input and outputs a high frequency and low resolution signal. Here we have 12-bit input and 1-bit output. The input frequency is the sampling frequency and the output frequency is 200 times higher, where 200 is called the oversampling ratio(OSR).

The first stage is to upsample the signal by 200 retaining its resolution as showed in figure 3. For correct upsampling the signal has to be filtered afterwards by a lowpass filter to avoid aliasing. It is possible to do the upsampling in just one stage but that is computationally inefficient because the demands on the single filter which would need a very narrow transition band would make it huge. Also notice that the filters operate at different frequency, that is how often it has to produce a new output, one single filter would have to operate at

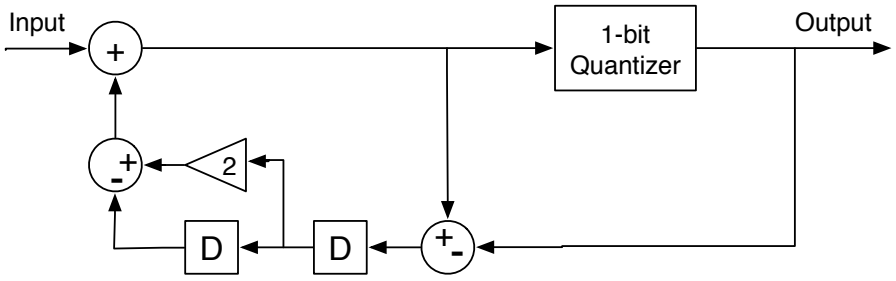


Figure 4: Delta sigma modulator.

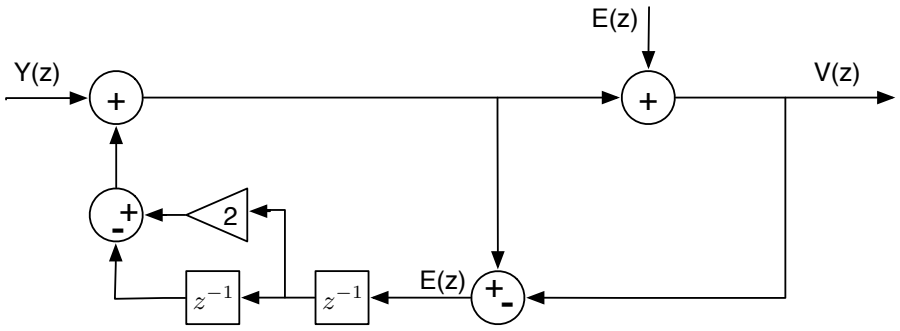


Figure 5: Delta sigma modulator with z-transform.

$200 \cdot F_S$. When using severeal upsampling stages the demand for the subsequent filters transition band is lowered since the signal band gets smaller which explains why the second filter uses only 12 taps compared to the first filter which uses 42 taps.

After the signals has been upsampled it is fed into the delta sigma modulator shown in figure 4. The quantizer transforms it into a 1-bit signal at the output. But it also feeds back the signal with the result that the noise is shaped to higher frequencies but leaving the signal unchanged. If the quantizer is modelled as adding quantization noise E(z) as in figure 5 and using the z-transform you get the result in equation 1. Here the input Y(z) is unchanged at the output while the quantization noise E(z) is shaped by $(1-z^{-1})^2$ which grows for higher frequencies. Since it is to the power of 2 it is called a second order modulator and the noise will grow in frequency by 40dB/decade.

$$V(z) = E(z) + Y(z) + (2z^{-1} - z^{-2}) \cdot E(z) = Y(z) + (1 - z^{-1})^2 \cdot E(z) \quad (1)$$

4.2 Implementation

The converter is implemented completely in hardware because of the computational complexity of the filters and high speed of operation. I takes a new sample every 5000 clock cycle and with the OSR of 200 that means it has to

output a sample every 25 clock cycle. That gives enough clock cycles for time-multiplexing.

4.2.1 Upsampling

$$y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k) \tag{2}$$

A FIR filter is calculated as in equation 2 where y is the filtered output, x the input, h the filter coefficients and M is the number of taps. The sum is implemented in hardware by one multiplier and adder connected in a multiply-accumulate configuration which means that it takes M clock cycles to calculate the output. To calculate y the M latest samples are needed which are stored in D-flip flop shift registers. Using circular shift registers makes it easy to access all samples by shifting one whole round. The coefficients h is stored in a ROM in the hardware, the values are calculated with Matlab and uses 16-bit word length.

Thanks to the fact that upsampling introduces zeros it reduces the size of the samples that have to be stored and multiply-accumulate operations that have to be performed. So for the first filter in figure 3 every second sample is zero so only 21 values have to be stored and only 21 multiplications have to be performed but observe that the ROM size has to be the same. In the same way the third filter with 15 taps only has to store 3 values and takes 3 clock cycles to compute.

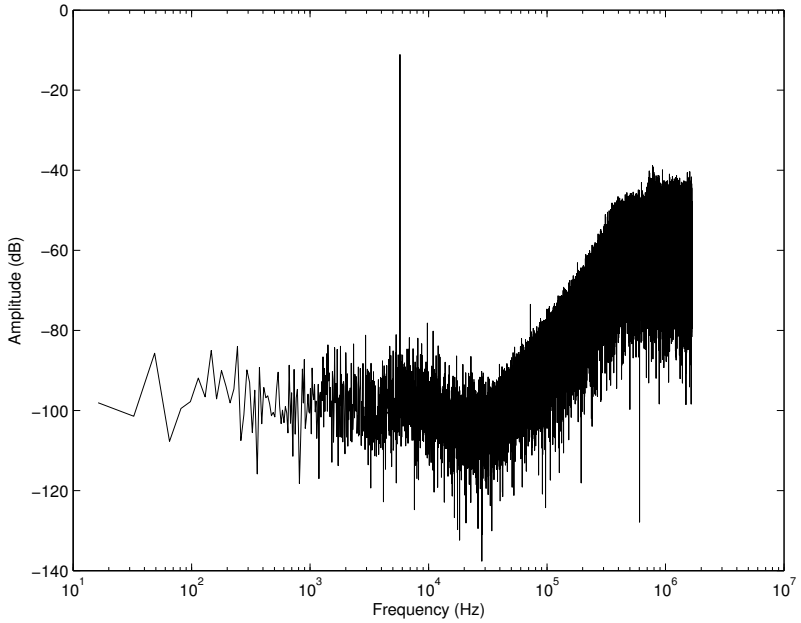
Performing all the computations for a sample in the configuration in figure 3 then takes $(42/2) \cdot 2 + (12/2) \cdot 4 + (15/5) \cdot 20 + (15/5) \cdot 100 + (4/2) \cdot 200 = 826$ clock cycles which means all can be time-multiplexed onto the same multiply-accumulate with good margin. A new sample has to be calculated every 25 clock cycle which means that filter 5 has to be run every 25 clock cycle. That is possible to achieve by giving filter 5 the highest priority and since the longest filter, filter 1, takes only 21 clock cycles to compute means that it can be scheduled without pre emption which simplifies a lot.

The scheduling is done so that filter 5 has the highest priority followed by filter 4 and so on. Basically it is Rate Monotonic Scheduling without using pre-emption. For every filter there is a counter that counts the number of cycles to the next time it should have an output ready and sets a ready flag every times it start again from zero. That is used by the scheduler which is just a simple state in a state machine that picks the filter with highest priority that is ready. The scheduler thus takes one clock cycle to run. All this means that the DAC will expect a new sample every 5000 clock cycles to function correctly and it is not possible to change the sampling frequency without redesigning the converter.

This is a finely tuned solution, and you seem to have put a lot of thought into getting the right rates here.

4.2.2 Modulator

The modulator part of the converter uses three additions and since it has 25 clock cycles to compute this part is also time multiplexed using only one adder. The quantizer is implemented by looking at the sign of the signal. The feedback from the quantizer is two constants for +1/-1 in the right word length. The multiplication by two is just a shift. Lastly a negator is needed to perform subtraction with the adder.



Nice illustrative graph. Good job obtaining this from the simulation.

Figure 6: Spectrum from delta sigma modulator, from VHDL-simulation.

The word length used in modulator is not the same as for the signal which is 12-bits. The reason for that is that the internal signals in the modulator can easily grow bigger than that of the maximum signal value. Simulations showed that the internal signals rarely exceeded ± 4 (the audio signal is between -1 and 1) so the modulators uses 2 extra bits. Even though it would not happen often an overflow is quite severe in a delta sigma modulator therefore the adder is designed so it saturates at the max/min value instead of overflowing.

4.3 Comparison with PWM

In figure 6 is the spectrum from the output of the delta signal when given a pure sine as input. The spectrum is from simulation of the VHDL-code ans is logarithmic on both axes. The noise is clearly shaped towards higher frequencies with the expected 40 dB/decade. At lower frequencies the noise is dominated by rounding errors in the filter which falls in the signal band and is therefore not shaped.

In figure 7 is instead a spectrum of an ordinary PWM implemented with a counter. The signal is at the highest peak. Compared to the delta-sigma the noise is concentrated at lower frequencies. Note that the second highest peak is at 20.3 kHz ($83.3 \cdot 10^6/2^{12}$ Hz) which is the frequency of the counter and which is only partly filtered out by the output filter and can be heard at the output. To get rid of that a faster counter is needed but that would also mean fewer bits of the signal were used.

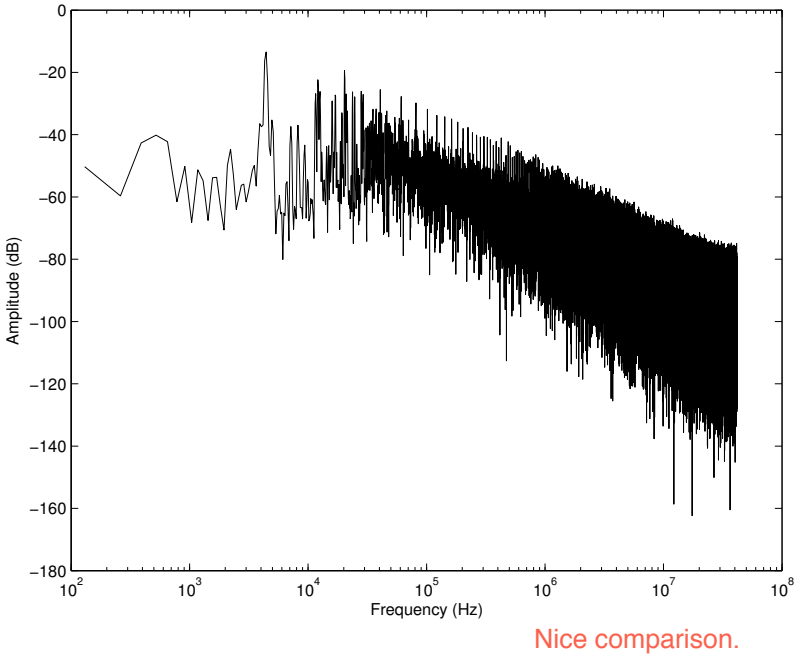


Figure 7: Spectrum from PWM simulaton in matlab.

5 Software

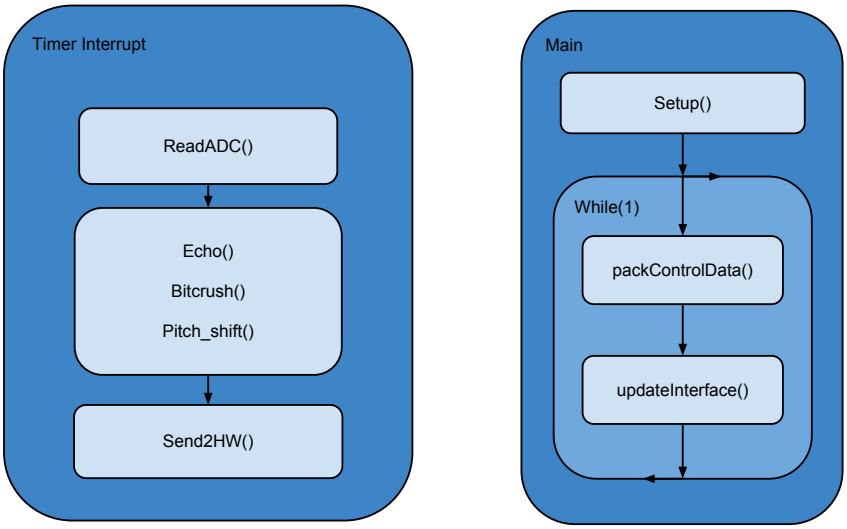


Figure 8: Split between main loop and timer interrupt

The software is divided into two main parts, the periodic tasks and the low priority tasks as seen in figure 8. The periodic task are handled inside a function that is activated with a timer interrupt. The tasks in the timer interrupt handler are the tasks that have to done before the next sample arrives, in our case requesting a sample, sending the sample through the different software sound effect algorithms and sending the sample to the custom hardware. In the main loop the tasks that aren't as timing critical such as updating the user interface, reading buttons and switches and sending control data to the hardware are placed. The main loop will be run with the *extra* clock cycles left between the timer interrupts.

This software splitting of tasks works as long as the tasks inside the timer interrupt don't take up too many cpu cycles so there is some cycles *left* for the main loop to run between the interrupts. If one would want to expand this project it might be beneficial to look into running a real time kernel to get optimal performance.

The microblaze is configured with all available BRAM which is 64 kB. Actually used a little bit more than half. As it is now a sample of 12 bits is stored in 16 bit integer and over 3000 samples are stored mostly for the echo effect. So the memory usage could be optimized a bit if that was a restriction.

6 Sound effects

6.1 Echo

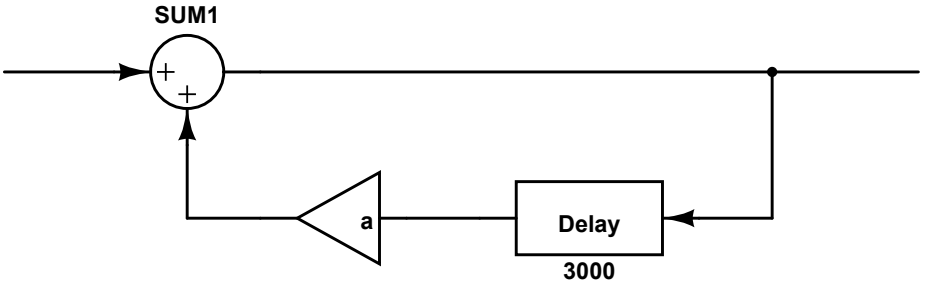


Figure 9: Echo block diagram

The echo sound effect is produced by using a delayed feedback loop according to the equation 3 also seen as a block diagram in figure 9.

$$Y = X + a \cdot Y \cdot z^{-1} \tag{3}$$

By changing the the length of the delay, the echo can be simulated to have to travel different lengths before bouncing back to the listener. This gives the ability to simulate different sizes of "rooms". For example a log delay might give the illusion of that the music is playing in a big arena while a small delay gives the illusion of being in a small confined space such as in the shower.

The other variable that can be changed is the feedback, a in figure 9. By choosing a feedback close to one, but not equal or bigger than one since that

will make the system unstable, it will give the illusion of the room having hard walls giving lot of echos while choosing a low feedback give the feeling of having a room with a lot of fabric damping the echos.

6.2 Bitcrush

$$\begin{aligned} &1.\ 11101011 \rightarrow 00001110 \\ &2.\ 00001110 \rightarrow 11100000 \end{aligned} \tag{4}$$

Bitcrush works by lowering the resolution of the signal and removing parts of the signal which amplitude is lower than a certain level. This is done by first doing a bitshift of the sample to the right and then shifting the sample the same amount back to the left, as seen in figure 4. Depending on how many times the sample is shifted different levels of bitcrush can be achieved. Bitshift introduces a lot of distortion and sounds best when used together with a single instrument such as a electric guitar.

6.3 Pitch Shift

6.3.1 Concept

With the pitch shift effect it is possible to increase the pitch by one octave or decrease it by one octave. A pitch shift is done by changing the frequency of the sound. Pitching up one octave doubles the frequency and pitching down one octave means halving the frequency. Note that there is a difference between a frequency shift where every frequency is shifted a constant amount in frequency. When pitch shifting you retain the dynamics of the sound because every frequency is instead shifted by *multiplying* it with a constant.

This multiplying each frequency with a constant can be done by upsampling/downsampling the signal. For pitch up you use downsampling with a factor 2 (multiply spectrum by 2) and for pitch down use upsampling with a factor 2 (multiply spectrum by 1/2) instead. The only problem is by downsampling/upsampling you change the length of the signal. When downsampling the length is halved so first it has to be made twice as long and for upsampling it gets twice as long and has to be halved beforehand. How this changing of the signal is done determines the quality of the pitch shift.

For making the signal longer the idea is to repeat segments of the signal. To make it repeat segments so that it resembles the original signal it tries to take segments that are a number of whole periods. To detect a period it looks for when the signal transitions from a negative to positive value. A period is then between two such transitions. Figure 10 shows an example of pitching up a signal. Making the signal shorter that is needed for pitch down is done by by finding a segment containing two periods and then only using the first part of the segment.

6.3.2 Implementation

The pitch shift is implemented in software. Doubling and halving of the signal is both done with the same circular buffer that stores 256 values. The size of the buffer is a power of two so that modulo of indices can be performed with a simple bitwise add operation. The doubling and the halving each has a pointer

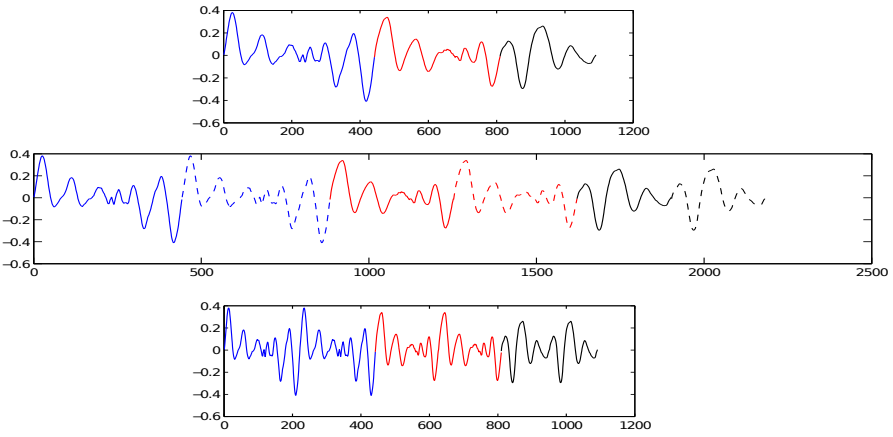


Figure 10: First plot is original signal, second plot is the signal twice as long by repeating segments, third plot is finally downsample the second to get pitched up signal.

Good visual explanation.

into this buffer that it samples from. For the doubling of the signal it takes two samples out of the buffer for every new sample. It then throw away one of the values when it downsamples. For the halving of the signal it takes a new sample from the buffer for every second new sample. For every second time that it do not take a new sample it upsamples by giving a zero instead.

The doubling starts taking its value in the middle of the buffer and since it samples from the buffer faster than it fills up the pointer moves to the beginning of the buffer. It keeps doing this until it encounters the latest transition which means that it should start repeating the segment. The repeating is simply done by doubling back, that is moving the pointer backwards the number of samples in the current segment. When it is done repeating the segment it is automatically back at the middle of the buffer.

The halving instead starts at the newest sample in the buffer. Since it samples from the buffer slower than it fills up the pointer will move into the buffer towards older samples. When it discovers that two whole periods have entered the buffer the pointer moves to the beginning again and thus skipping half of the segment.

For both methods if it does not encounter transitions so that the pointer reaches the end of the buffer that is no problem since it is a circular buffer. For doubling it will just repeat the whole buffer and for halving it will start again from the beginning. Both effects run simultaneously and can be mixed. As parameters you supply how much you want of each effect and how much of the original signal.

6.3.3 Filters

When doing upsampling and downsampling correctly the signal has to be filtered to avoid aliasing. The first implementation was done in hardware and contained a 42 tap FIR filter low pass filter that did this. Unfortunately it had to be moved

to software and that was not feasible any longer. Therefore the filter for pitch up and downsampling was skipped after Matlab simulations showed there were no significant penalty in sound quality by doing this. For pitch down and the upsampling a fourth order IIR-filter is instead used. Here the filter is more important because it also smooths out the boundary between segments which do not fit as well as for pitch up where the boundaries is always at a negative to positive transition. A IIR filter is used because it needs fewer multiplications and since we are in software there is no penalty for using the longer word length needed in a IIR filter.

6.4 Distortion

The Distortion effect simulates the clipping distortion that occurs when an amplifier is driven to hard. This means that when the input signal rises above a set level the output stays the same. This is illustrated in figure 11. In this example the signal is clipped at 75% of the maximum value. The main use for distortion as a effect is with a single instrument, often the guitar, in rock'n'roll style music. The distortion increases the number of harmonic overtones in the signal as well as increase the sustain of the notes played. The parameters that can be set by the user is the threshold for the clipping. Then the signal is amplified with a value set by the user.

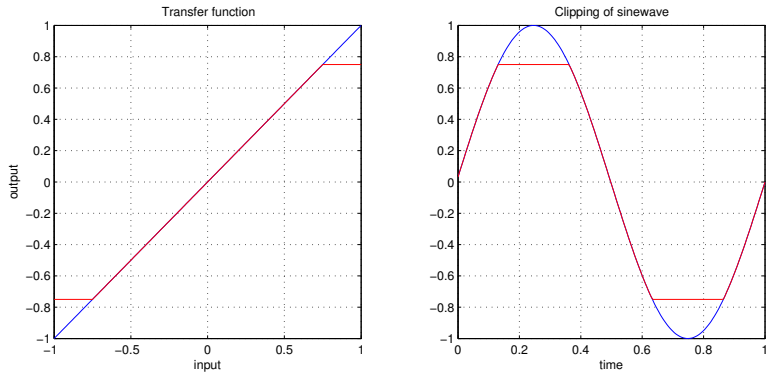


Figure 11: Distortion of a signal. The red signal is the distorted signal and the blue signal is the unclipped signal

6.5 Chorus/Flanger

The chorus and flanger effect are very similar and works as a single block in hardware. The effect emulates the effect that occurs when two notes are played at almost the same frequency. The effect can be described by the formula in equation 5 that describes how two sine waves mix together.

$$\sin(2\pi \cdot f_1 \cdot t) + \sin(2\pi \cdot f_2 \cdot t) = 2 \cdot \cos(2\pi \cdot \frac{f_1 - f_2}{2} \cdot t) \cdot \sin(2\pi \cdot \frac{f_1 + f_2}{2} \cdot t) \quad (5)$$

$$
\begin{aligned}
y(n) &= x(n) + \alpha \cdot x(n - \beta(n)) \\
\beta(n) &= R \cdot (1 - \gamma \cdot \sin(\omega_0 \cdot n))
\end{aligned}
\tag{6}
$$

The effect can approximately be achieved by using the formula in equation 6 where α is the intensity of the effect. R is the centerpoint of the sweep. γ is the width of the sweep. ω_0 is the speed of the sweep.

This equation was realized by implementing a shift register with addresses and then the address was swept using a sine wave, see section below. The block diagram of the effect can be seen in figure 12.

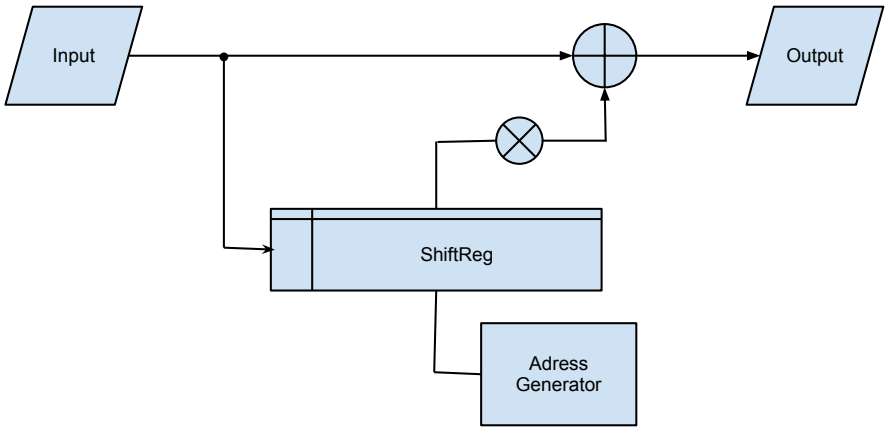


Figure 12: Block diagram of the chorus/flanger effect

The shift register stores 256 samples, which enables a maximum delay of 15 ms. The width of the address sweep, the speed of the sweep as well as the intensity of the effect where accessible by the user through the seven segment display. The choice of whether the effect should act as a chorus or a flanger is selected by one of the switches for effect selection.

The main difference between a chorus and a flanger is the center of the delay, R. In flanger mode, the sweep is between 0.1 ms and 7 ms, whereas the chorus effect sweeps between 8 ms and 15 ms.

An example of the effect on a sine wave can be seen in figure 13.

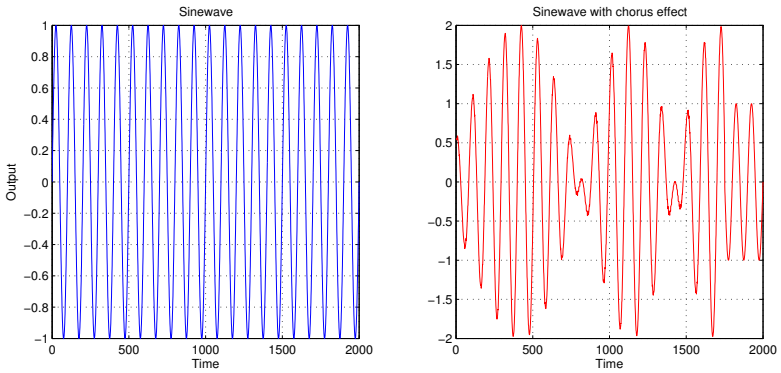
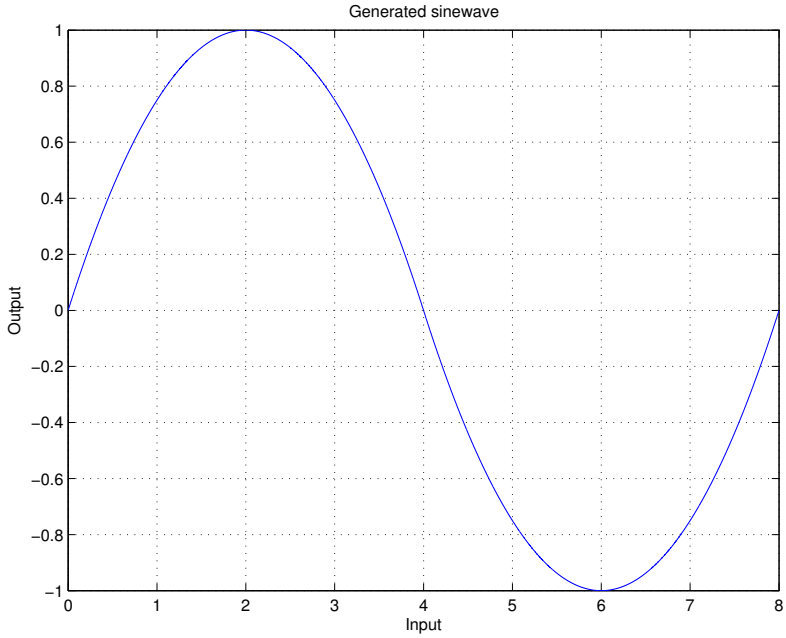


Figure 13: The effect of chorus on a sinewave

6.5.1 Sine generation

This effect needs some kind of oscillating component in order to function. A simple triangular wave would have worked, but a sine wave will produce a more smooth and natural sounding effect. An easy way to simulate one period is with a third order polynomial . If a half period is simulated a second order polynomial is sufficient. However, a true sinusoidal signal with a period of 2π is hard to realize in hardware. This can be solved with a modified sine wave with a period of 8. This solves two problems, first it makes the calculations of the wave easy, since all the coefficient becomes a power of 2. Second, since the period is an even number, the period can be halved and then the other half is simply inverted. The algorithm for the modified sine wave can be seen in equation 7 and the waveform can be seen in figure 14.

$$f(n) = \begin{cases} x \cdot (1 - 0.25 \cdot x) & \text{if } 0 \leq x < 4 \\ -(x \cdot (1 - 0.25 \cdot x)) & \text{if } 4 \leq x < 8 \end{cases} \tag{7}$$



A nice concise but explanatory description of all the effects you implemented.

Figure 14: The simulated sine wave with period 8

7 Manual

7.1 Connections

There are two pieces of external hardware that has to be connected to the FPGA-board in order to get the project up and running, the PmodAD2 (ADC) and the PmodAMP1 (amplifier and filter ciruit).

The PmodAMP1 is connected directly to the Pmod headers JA1-JA6. The PmodAD2 is connected to the Pmod JA4-JA6 but since the PmodAD2 is lacking the pullup resistors on the I2C lines (see 8.1) required to get the communication working two external 4.7 kΩ resistors have to be added to JA4 and JA5 that pulls up the lines to Vcc.

7.2 User interface

The user interface includes the switches, the buttons and the seven segment display on the FPGA. The switches handles the effect selection and the buttons and seven segment display are used to tweak the parameters of the effects.

7.2.1 Effect selection

The switches enable och disable the individual effects. There is also a switch to bypass all the effects and one to switch the chorus/flanger effect between its two modes. The 8th and last switch isn't used since there is no more effects in the

design, but in a future implementation it could be used to control new effects or additional parameters in existing effects.

The order of the switches are as follows, from left to right:

- Bypass on/off
- Echo on/off
- Bitcrush on/off
- Pitch shift on/off
- Distortion on/off
- Chorus/flanger on/off
- Off: flanger On : chorus
- Not used

When a switch i activated, a green diode will lit up to give a visual confirmation that the switch has been activated. This is illustrated in figure 15, where the echo, pitch shift, distortion and flanger is active.

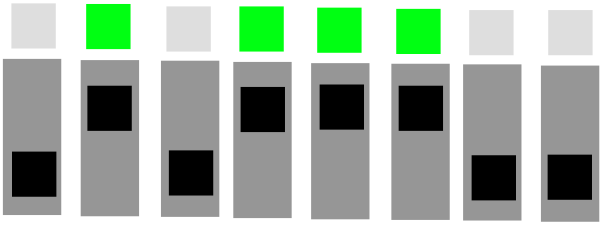


Figure 15: Switches to enable effects

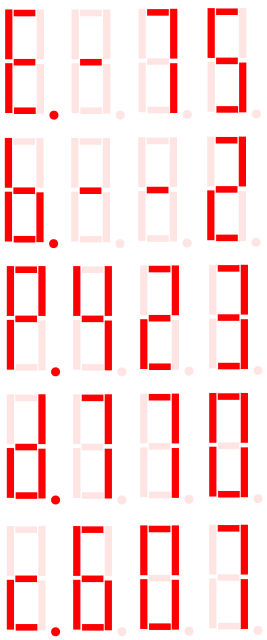
7.2.2 Parameter settings

The parameters of the different effects is controlled by the buttons on the FPGA. The seven segment display shows the values of the current effect that is being modified. the left and right buttons change the cursors position while the up and down buttons switches between the different effects or the level of the parameters. The cursor is visualized by the red dot in the display. The parameters can be set between 0 (minimum) and 7 (maximum). An example can be seen in figure 16. Any place for parameters that isn't used are represented by a '-', and the cursor simply ignores them when navigating the display.

Figure 16: The seven segment dislay. Current effect is Echo, the cursor is at its leftmost position, the feedback is set to 7(max) and the delay time is set to 5

Here is a complete list of all the effects and their parameters that the user can access. For more information on each effect, see their respective sections in this report. For a graphic view of all effects and parameters, see figure 17. The values in figure 17 are examples of how to setup the effect as a starting point.

- Echo
  - feedback
  - delay time
- Bitcrush
  - crush
- Pitch shift
  - Volume pitch up
  - Volume original signal
  - Volume pitch down
- Distortion
  - Clip level
  - Volume out
  - Clean volume
- Chorus/flanger
  - Width of sweep
  - Speed of sweep
  - Intensity of effect



Nice description.

Figure 17: The different views of the seven segment display

8 Problems

8.1 I2C ADC

A problem we had with the AD-converter was that the PCB containing the ADC chip made by digilent, PmodAD2, was lacking the pullup resistors on the SDA and SDL lines of the I2C bus, that according to the I2C standard should be 4.7 kΩ. We solved the problem by plugging in the PmodAD2 to a breadboard and adding two external resistors to the bus-lines, shown in figure 2 as R5 and R6.

Another problem we had was that the was that the I2C ip-core provided i XPS did not support I2C in high speed mode which meant that the I2C communication was limited to a speed of 400 Kbit/s, which is too low when sampling music. To solve the problem we took a brute force approach and overclocked the I2C bus to 800Kbit/s by raising the clock frequency in the ip-core to 1 MHz instead of the 400 KHz specified in the I2C specification.

8.2 Memory access speed

When experimenting with where to put the software code we realised that putting the code in microRAM instead of the BRAM would give us a access time that was an order of four magnitude longer. We solved the problem by adding more BRAM to the Microblaze and running the code from there.

8.3 Helicopter sound

When first connecting the delta sigma converter we noticed something in the background that can best be described as the rotors from a helicopter. When trying with a pure sine wave it was more obvious and we measured that there was a glitch occurring with a frequency of about 13 Hz. This was due to the delta sigma not being in complete sync so that the hardware tried to supply a value to the converter when it was not ready and thus ignored it. The error then propagated through the filters giving the distinct sound. This was remedied by adding sync signals and if needed repeat a sample to keep them in sync.

9 Conclusions

Implementing echo in hardware would likely require more memory, right? In fact it would probably be ok to use slower off-chip memory.

The final result was quite close to what we imagined it would be like when we first started the project. The quality of the sound and effects are quite good. After testing the full system, one thing we would like to change is the ordering of the effects. Most notably the echo should be applied at the end after all the other effects. That would require redesigning since echo is done in software which means the sample would have to be sent to the the hardware, apply effects, send back to software to add echo and sending it back to hardware to output it.

After having some problem interfacing with the ADC we have learned that you have to be very meticulous and critical when reading the documentation. You cannot always trust what they write but have to look something up yourself. And we would do it again we would probably try to use another ADC to avoid the slow I2C bus.

A thing that was helpful to us that before we started to implement everything we first modelled all effects in Matlab. It is very fast way to find out what works and do not work and for example how much delay that is needed for the echo effect. It is also good to verify that it works when it is finished. For the more complex delta sigma converter it was first modelled so it worked in Simulink which can later be used for debugging. And of course Matlab was invaluable when designing filters.

Since much of the time as spent debugging we have improved our skills at that. Especially since it isn't as easy on a FPGA as normal software. For example for debugging the helicopter sound we used an oscilloscope looking at the output and managed to recreate the effect by using Matlab/Simulink to isolate the error.

Final result video: http://youtu.be/7s-slHC7MQ0

Nice with an online video.

10 Contributions

- Stefan Granlund: designing and implementing hardware effects for chorus/flanger and distortion, designing top file connecting hardware parts together including control data, co-designing user interface
- Anders Skoog: researching and connecting the ADC, writing main software structure including effects for bit crush and echo, co-designing user interface and implementing it
- Fredrik Stolt: designing and implementing delta sigma converter in hardware, choosing algorithm for pitch shift and implementing it, adapting hardware for connecting to PLB bus

Debugging and testing was done by all three in cooperation.

11 Hardware and Software code

Download here:
https://docs.google.com/open?id=0B4lW24npVpTodk5COEVUUk83ZlE

A very nice project and good report. A few references here to various effects for instance would help. Otherwise not much to improve. Well done.