# IMPLEMENTING DIGITAL AUDIO EFFECTS USING A HARDWARE/SOFTWARE CO-DESIGN APPROACH

*Markus Pfaff, David Malzner, Johannes Seifert, Johannes Traxler, Horst Weber, Gerhard Wiendl*

FH-OÖ/Hagenberg, Dept. HSSE
Softwarepark 11, A-4232 Hagenberg/Austria

## ABSTRACT

Digital realtime audio effects as of today are realized in software in almost all cases. The hardware platforms used for this purpose reach from multi purpose processors like the Intel Pentium class over embedded processors (e.g. the ARM family) to specialized DSP.

The upcoming technology of complete systems on a single programmable chip contrasts such a software centric solution, because it combines software and hardware via some co-design methodology and makes for a promising alternative for the future of realtime audio. Such systems are able to combine the vast amount of computing power provided by dedicated hardware with the flexibility offered by software in a way the designer is free to influence.

While the main realization vehicles for these systems – FPGAs – were already promising but unfortunately offered limited possibilities a decade ago [1] they have made rapid progress over the years being one of the product classes that drive the silicon technology of tomorrow.

We describe an example for such a realtime digital effects system which was developed using a hardware/software co-design method. While digital realtime audio processing takes place in low latency dedicated hardware units the control and routing of audio streams is done by software running on a 32 bit NIOS II softcore processor. Implementation of the hardware units is done using a DSP centric methodology for raising the abstraction level of VHDL descriptions while still making use of standard of the shelf FPGA synthesis tools. The physical implementation of the complete system uses a rapid prototyping board tailored for communications and audio applications based on an Altera Cyclone II FPGA.

## 1. INTRODUCTION

Software running on a DSP or a common CPU is the prevalent vehicle of digital real-time audio effects implementation today. Realization of such effects in dedicated hardware has some appealing advantages especially in low latency and high reliability applications [2]. Little flexibility and a much more complicated design process than software does offer are the other side of the coin. These severe drawbacks have prevented dedicated hardware design from gaining ground in the digital audio effects realm at least in its consumer and semiprofessional floors.

Using the arithmetic package described in [3] which provides a fractional data type for fixed point digital data processing the abstraction level in terms of data handling and arithmetic expressions raised a lot over what is possible using the integer types typically encountered in such descriptions: signed and unsigned. The work described in [4] gave proof of concept for the general usefulness of the fractional package for digital signal processing as it is done in applications typically found in the communications industry. Such applications seldom make use of the large amount of dynamic parameters that many audio applications demand. Flexibility of a description simply is no issue in this case.

The desire to broaden the application area of our approach led to the decision to implement effects from the audio domain in hardware using a rapid prototyping board. The project DAFX [5] was launched at the University of Applied Sciences of Upper Austria at Hagenberg in October 2006 which aimed the evaluation the feasibility of a combined hardware/software approach for targeting the audio effects application field. Parameterization of the hardware audio effect modules is done through software running on a 32 bit softcore processor which is implemented together with the effects on an FPGA as a re-programmable System-on-Chip. Hardware and software subsystems together built a complete hardware/software co-design. The software controlled processor core also controls the data streams connecting the different hardware effect units.

We will point out advantages as well as disadvantages found while realizing a digital audio system this way. We start by describing the rapid prototyping system used as the realization platform. The second part of the paper deals with the used components and the basic setup for effect development followed by the description of the implementation and design funcional simulation based verification of effects.

## 2. RAPID PROTOTYPING BOARD SANDBOXX

The platform chosen for the realization of the system described in this paper is the rapid prototyping board *Sand-*

*boxX* which has been developed at the University of Applied Sciences of Upper Austria at Hagenberg and is used for educational and research purpose.

The board shown in Fig. 2 is built around an *Altera Cyclone II* FPGA (*EP2C35F*). Peripherals (see Fig. 1) of the FPGA are 16 megabytes of SDRAM, a *Texas Instruments* audio codec (*TLV320AIC23B*), a MIDI Interface, a PS/2 interface and a programmable clock IC (*ICS307*). Further hardware units such as a PCI Bus connector are available on the board, but were not used in the project. The board powers up with a default clock frequency of 25 MHz. Because the system uses a clock frequency of 48 MHz, we needed to reprogram the clock frequency. This is done on startup by software on the NIOS II processor. For configuration an SPI interface is used.

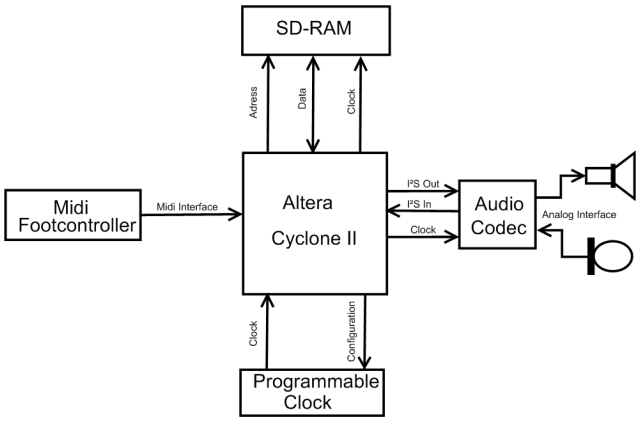


Figure 1: Hardware for the audio effects

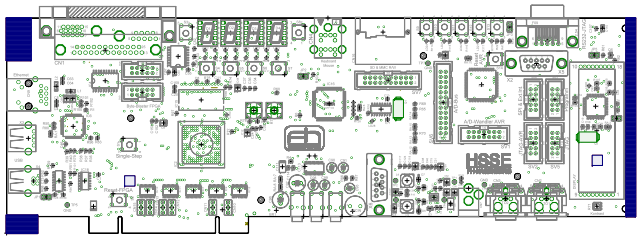

Figure 2: Silkscreen of the rapid prototyping board SandboxX.

The audio codec can be configured and is capable of transferring audio data in several different ways. For our application we used the $I^2S$ interface to transfer audio data and the SPI interface to configure the codec. The audio codec is used in slave mode, so that the clock for the codec has to be generated with a hardware frequency divider implemented on the FPGA. This makes it easier to keep the system synchronous. The serial $I^2S$ audio data is converted to a 24 bit parallel signal by a hardware unit. A valid bit indicates when new data arrives. Sending audio data to the audio codec works the other way round making use of a parallel to serial converter unit.

The analog audio data is pre-amplified and routed to audio connectors, so that the SandboxX can be used standalone for creating audio effects. A disadvantage of the prototyping board with regard to the analog signal quality is the low-cost power supply by an USB port. This keeps cost down, but results in a higher noise floor of the supply creeping into the audio signal domain.

The SDRAM is connected via an address / data bus with 32 data lines. Because the SDRAM is placed beside the FPGA and we use a relatively high clock frequency for the memory (48 MHz system clock), we had to shift the clock's phase with an integrated PLL of the FPGA to meet timing requirements.

The MIDI and PS/2 interfaces are connected to the FPGA via appropriate level conversion and an opto-coupler. The MIDI interface is used to connect the MIDI foot controller described in section 3.5. A PS2-compatible mouse was used during the debugging phase to change the bandpass centerfrequency of the *WahWah* effect (see section 4.5).

The configuration data for the FPGA is automatically loaded from an SPI Flash ROM on the board by a programmer implemented in a separate CPLD.

## 3. SYSTEM DESIGN

The aim of our work is to implement audio effects as dedicated hardware units without losing system flexibility. The system had to have the capability to be configured and parameterized easily. Our proposed solution is the use of a softcore processor leading to a hardware/software co-design as proposed in [6] which is implemented on a single chip. We used the *NIOS II* softcore processor supplied by the FPGA vendor *Altera* to run the software part of the system. The effects are implemented as dedicated hardware units to do the signal processing while the processor core and thus the software controls signal routing.

The used prototyping platform SandboxX offers only a single SDRAM chip with direct interconnect to the FPGA. This kind of memory needs a special controller in order to be accessed in a correct way. In our system the SDRAM is connected to the bus system of the softcore processor via the SDRAM controller provided by the Altera NIOS II development system. While the SDRAM cannot be directly accessed by the audio effect units which might use quite large amounts of memory (especially delay based effects) the use of software to control the audio data streams through the SDRAM posed no performance problems. The processor just has to forward data so that little processing power is needed.

For communication between the *NIOS II* processor and the audio effects we used general purpose IOs instead of

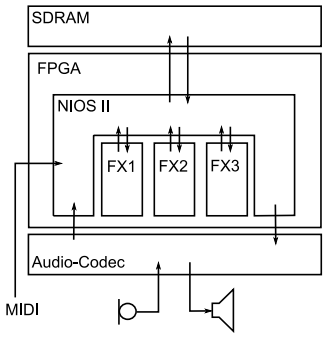


Figure 3: Hardware/Software system

integration with the processor system bus (Fig. 5). This has the advantage that the softcore processor could be easily exchanged and the interface needs less implementation effort. This approach also offers an easy access of interrupt sources. The drawback with respect to a direct system bus connection is the smaller peak performance that can be gained. In our system the throughput was by far high enough to prevent dropping of audio date in any case. In future applications the processor's system bus (*Altera Avalon Bus*) could be used together with some kind of audio stream switching matrix implemented as a dedicated hardware unit to make the system even more independent of software performance irregularities.

The software manages and controls the system to keep the trade-off between performance and flexibility. A *MIDI* control unit (e.g. foot switch) can be connected to the system. With such a device one can choose and control different effects in realtime.

The processor is the only instance that has access to the SDRAM. An effect can give the processor a request of writing data to it's effect memory or read data from it. The used hardware-software interaction scheme is depicted in Fig. 4.

The measurement of the processor usage at the memory intensive delay effect results in 33% usage for the data transport (audio codec, SDRAM) and 66% usage for the main loop.

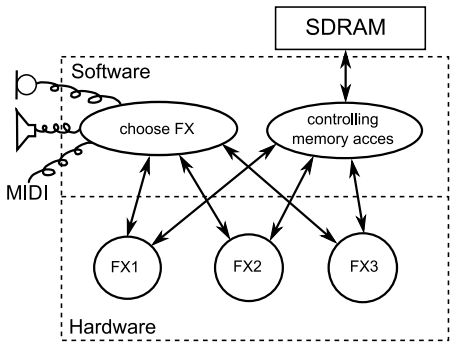


Figure 4: Hardware-software interaction scheme.

### 3.1. Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) are on the verge of revolutionizing digital signal processing in the manner that programmable digital processors (PDSPs) did nearly two decades ago [7, 8]. Many front-end digital processing algorithms, such as FFTs, FIR or IIR filters, to name just a few, previously built with ASICs or PDSPs, can now be replaced by FPGAs. Modern FPGA families provide DSP support with multipliers and fast-carry chains that are used to implement DSP algorithms at high speed, with low overhead and low costs [2].

### 3.2. Embedded Processor

The reason why we used the NIOS II softcore processor is the good support for custom processor system design offered by the Altera Quartus II tool set. We were able to build a specific system exactly tailored to our purpose with all the peripherals needed. They can be configured to adopt specific user needs. An example are the ports used for audio data, because they can be generated with the exact bit width delivered by the audio codec. Further, we can add as many SPI and UART interface units as we need for our peripherals (MIDI, programmable clock generator, audio codec). Detailed information concerning functionality of NIOS II peripherals is contained in [9].

The processor core was generated with the "standard" settings (NIOS II/s), featured with instruction cache, branch prediction, a hardware multiplier as well as a hardware divider. The debugging interface was generated with configuration "level 2". This means that two hardware breakpoints and data triggers are supported and debugging of the system via JTAG-interface is possible. For the hardware MIDI interface an UART module from the Altera SOPC Builder development tool was integrated into the processor system.

The system clock of the processor and all peripherals is set to 48 MHz and the internal RAM size implemented in FPGA internal RAM blocks is set to 15 KByte.

### 3.3. Software Development

The software for the NIOS II is written in C. It's main tasks are the parameterization of the hardware effects in reaction to the incoming MIDI data. Besides that the software is controlling the audio data flow from one effect to an other effectively acting as an audio routing matrix and also configures and initializes the audio codec. Altera offers a software development environment based on Eclipse for the NIOS II family. This environment includes a (fee free) GCC compiler optimized for the NIOS II instruction set. The SOPC Builder tool also generates processor specified libraries. Such libraries can be included into the development environment and provide the function of a hardware

abstraction layer.

Although possible, an operating system is currently not used.

### 3.4. Interface of Effects Units

The interface between the control unit and a single hardware effect unit plays an important role in the overall system design. The user should be provided with maximum flexibility when using the audio effects, which of course includes the way in which audio signal is routed through the effects. There are several implementation strategies, which fall into consideration:

- **Static effects-line:** This attempt leans against the technique used with the common guitar gadget boxes. It means, that there is a fixed order how the effects are joined together and the user just can switch on or off.
- **Multiplexing structure:** An expanded structure of the static effects-line could be a huge multiplexing matrix, so that the effects order is more flexible.
- **PIO:** In order to gain flexibility some software may be required. This branch stands for the easy way of connecting hardware effects to the softcore processor, namely through simple PIOs (Parallel Input/Output).
- **Avalon Bus:** This method is the advanced strategy of PIOs. We used the NIOS II softcore, which main bus system is an *Avalon Bus*. Each hardware effect may provide an Avalon Bus-connection, which is faster than PIOs.

In a software DSP system the routing of the data stream is done in a similar way the data processing is done. Because dedicated hardware units are used for audio processing the routing has to be done by a special unit that acts as a central switch node connecting all processing nodes.

Here the flexibility of the embedded NIOS II softcore comes into play. SOPC-Builder allows any number of port peripheral units needed. Consequentially each individual effect got it's own data and control ports which are directly connected to the NIOS II giving software full control of the routing of audio data. Audio inputs as well as outputs of all processing units are also connected directly to the softcore so that the user is able to pass the audio data through different effects in different order (see Fig. 3).

Some effects may require buffer memory also. Unfortunately it's neither affordable to spend each effect it's own SD-RAM nor was it possible to do so on our prototyping board. The single external SD-RAM (16 MB) is managed by the NIOS II via a SD-RAM controller peripheral. If an effect needs to buffer data, it can use the RAM-ports as shown in Fig. 5. The RAM address is a relative one, the softcore converts it and forwards the data physically to the SD-RAM.

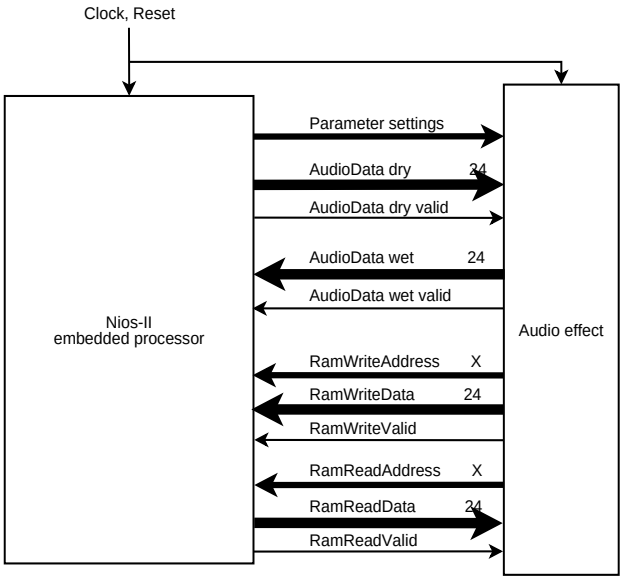


Figure 5: Interface between NIOS-II softcore and effects.

### 3.5. MIDI Control Interface

The human control interface is a MIDI foot controller. There are two different types of pedals on the foot controller: ten foot switches, which are used to switch through effects and two expression pedals, which are used to parameterize the active effect. The software on the NIOS II is sensitive on the used MIDI control sequences. If a valid command is received, the parameters for the chosen effect are calculated and transmitted from the processor to the hardware effects block. Changing effects disposes a recalculation of the parameters.

## 4. EFFECTS

### 4.1. Chorus

The Chorus effect simulates playing various instruments simultaneously. When more musicians play instruments simultaneously, they will not play exactly synchronously. A fixed and a variable time difference between the instruments exists. The chorus effect does the same thing. It adds an audio signal several times to itself where each instance of the audio signal (i.e. the summands) is delayed by some amount in time. The delayed signal is generated by adding a fixed delay in the range of 15 – 20 milliseconds and a sweeping delay of 4 – 8 milliseconds (see Fig. 6). In our implementation, the sweeping delay has the waveform of a triangle and

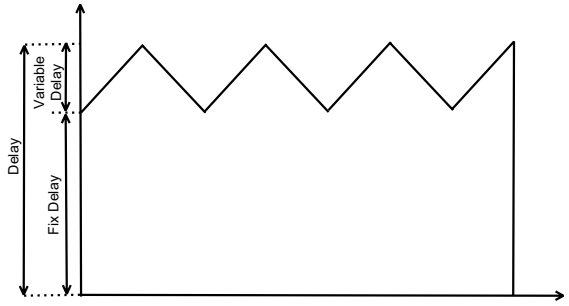


Figure 6: Delay of Chorus

a frequency of 1 – 5 Hz. Another common used waveform for the sweeping delay in the chorus effect is a sine wave. The advantage of a sine wave is that it is a very smooth function and creates a harmonic sound. The sweeping frequency is a parameter which can be changed while the system is in use. As an option there could be also used a logarithmic or sine waveform for the low frequency oscillator to change the variable delay.

### 4.2. Delay

Depending on delay time the effect has different names because of the particular character of the resulting sound. If the delay is in the range between 10 and 25 ms, we will hear a quick repetition named slapback or doubling. If the delay is greater than 50 ms we will hear an echo [10]. One can use a single hardware unit for both of these delay based effects while the splitting into two different effects can be done in software.

Address calculation for the RAM used as audio buffer is implemented in hardware in the manner of a ringbuffer. Via the parameter *delaytime* the size of this ringbuffer can be controlled. The output is calculated in a feedback loop. This means the output is computed by the input sample and a former output value read from the buffer. The output values are scaled with the parameter *level* and written to the buffer memory.

### 4.3. Echo Cancellation

An emerging topic in audio signal processing is echo cancellation. Especially the extensive use of Voice over IP, hands-free kits and conference calls enhanced the publicity and the technology of canceling interfering reflections. This problem occurs due to the fact that the spherical radiation of loudspeakers is reflected by objects. Since acoustic waves are expanding at an almost fixed speed, different time shifted reflections, depending on the place where they are measured, are created. Consequently, rooms have different acoustic characteristics, which means that e.g. a small room with reflecting walls is totally different to a big room with walls that are reflecting hardly anything.

The major problem in conjunction with these reflections arises due to the use of acoustic transmission. Because data is sent in packages there has to be a buffering at first, additionally time delays occur during the transmission. Such a delay could be a packet loss or a busy resource. Fig. 7 illustrates the problem. At first participant A is sending data, so there is a delay caused by buffering at point A as well as the transmission delay to point B. Accordingly the data is played at point B, reflected, recorded, buffered again and sent back to point A. Consequently the sender gets the interfering reflections with the sum of all delays, which is just annoying in the slightest case but can also render the phone useless.

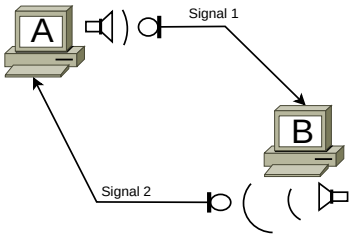


Figure 7: Delayed audio transmission.

A solution of this problem can be filtering of the reflections which are created on the side of each participant. The basic solution of this problem was already solved 50 years ago by *Norbert Wiener*. He found a way to calculate the coefficients of a filter for filtering just a specific signal. Therefore the inverse autocorrelation of the received signal, as well as the cross correlation between the received signal and the sent signal is needed:

$$h_{opt} = R_{xx}^{-1} r_{xs}^{(\lambda)} \tag{1}$$

Now it would basically be possible to measure the autocorrelation and the cross correlation in advance, for calculating the optimal impulse response to create an echo canceler. But since the reflections are different for every place and even differ when moving an object, the calculations have to be done during filtering. This requires the use of an adaptive filter.

Adaptive filters (see Fig. 8) are calculating the impulse response iteratively by minimizing the error signal. They start with an assumption for the impulse response. With this assumption the first convolution is calculated, then the result is used for generating an error signal that can be used to update the filters coefficients again. Thus, the impulse response is converging to the optimum Wiener solution with each iteration. An algorithm which is commonly used for this purpose is the NLMS, the normalized least mean square algorithm. Therefore the variable $\mu$ is used for the step-size

parameter to control the attended adaptation time as well as the residual error. The normalization of this equation is done by the use of the squared Euclidean norm. Haykin [11] describes the effect as lowering the adaption for large signals and increasing the adaptation for small signals. This produces a lower noise which stabilizes the calculation and increases the adaptation time. Since for our simulations and implementations we are using a range of values between -1 and +1, the division has to be exchanged with a multiplication to achieve the same effect. Consequently it can be

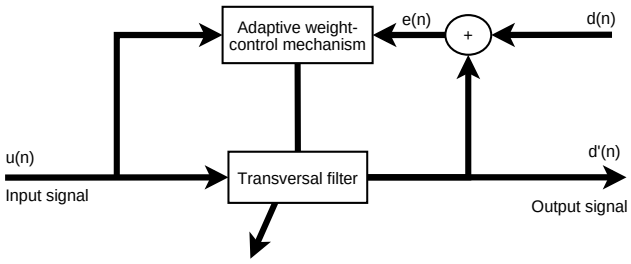


Figure 8: Adaptive filter

summarized that the calculation effort takes about $M$ multiplications and $M$ additions for the convolution of the FIR filter and additionally $3M$ multiplications and $M$ additions for the recursive algorithm.

$$\widehat{w}(n+1) = \widehat{w}(n) + \frac{\tilde{\mu}}{||u(n)||^2} u(n) e^*(n) \qquad (2)$$

To achieve a good performance it should be considered that the length of the filter has to be at least as long as the reflection takes to be recorded. Assuming a delay path of 10 meters it takes about 30 ms until the delay will be recognized. This leads to a minimal order of

$$M = 30\,\mathrm{ms} \times 44117 \frac{\mathrm{Values}}{\mathrm{s}} = 1300$$

for using audio quality sample rates. Such a high order goes along with very high demands for the processing speed, since it takes about $6M$=7800 calculations per value, or $340M$ calculations per second, for using CD quality sample rate. Based on the fact that the used IC is working with a clock of 48 MHz it can be assumed that

$$steps = \frac{48MHz}{44117Hz} = 1088$$

steps are needed to calculate the adaptive filter between two arriving audio values. This already shows that the implementation could be a challenge, because the proportion between calculations and time is 7:1. So in order to solve this problem the parallel advantages of the FPGA have to be used. This can be done by dividing the filter into several parts, calculating them separately at the same time and combining them finally together. This possible realization differs from a digital signal processor in that way, that the maximum number of fragmentation is much higher and that the unused hardware parts are not affected at all. The used IC offers for example 35 embedded 18 bit multiplier. Furthermore a VHDL simulation shows that in order to realize 350M calculations in a second, the filter has to be divided in 5 different parts which would just just 15% of the available multiplier. And while the resources of a DSP would have been depleted by this filter, the implementation on a FPGA would still have a lot of calculation power left.

### 4.4. Flanger

The flanger is a delay based effect like the echo, so the architecture of the delay effect can be used as a starting point. The difference is that in the echo effect a linear addressing scheme is used while the flanger effect uses a delay time that varies at a low frequency. The various frequencies differ with steps by $\frac{1}{44{,}1kHz}$, thus no fractional interpolation is needed. We create a sinusoidal oscillation via a direct digital synthesis (*DDS*) unit. DDS is a technique using a look-up table to store the values of a function which should be generated. This oscillation is added to the linear addressing used for the ringbuffer as shown in Fig. 9. The frequency can be controlled by a parameter.

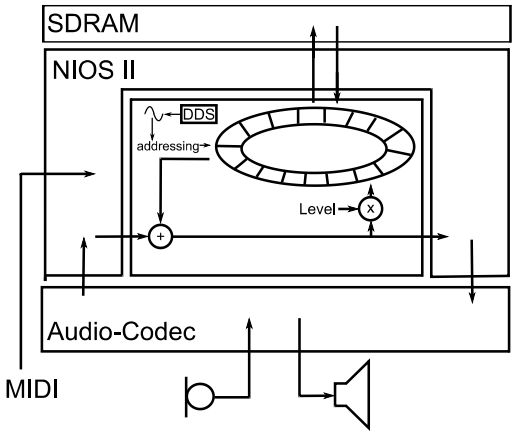


Figure 9: Flanger effect architecture.

### 4.5. WahWah

The WahWah effect describes a time-varying bandpass filter. In this case the concept "second-order allpass" as found in [10, p. 41] was realized. The idea is to create an allpass filter with phase shifting by 360. The filtered signal has to be *subtracted* from the original signal. When this elementary operation is done an allpass behavior results instead of a bandpass (BP) behavior. Analogue to this fact, we can *add*

the original signal to the filtered signal so that the result is a band reject (BR) filter as seen in Fig. 10.

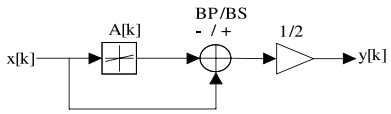


Figure 10: Blockdiagram of the allpass method

#### *4.5.1. System Behavior*

The cut-off frequency is the frequency at which the phase is shifted by 180. When we consider the complete system including subtraction, exactly the same point represents the currently center frequency of the bandpass. The frequency band for the complete phaseshift (-360) is the bandwidth of the bandpass (see Fig. 11).

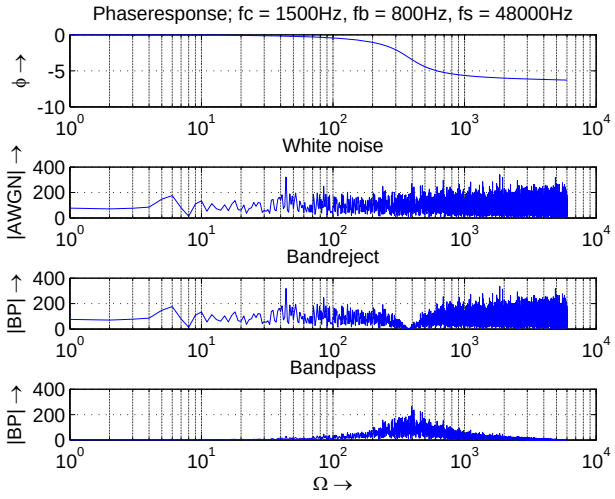


Figure 11: Demonstration of the allpass-method

Because the used filter is an IIR type of second order, there are just 6 filter-coefficients. In addition there is the special character of an allpass, which means that the infinite filtercoefficients are mirrored to the finite filtercoefficients (equation 5). Also the first infinite filtercoefficient has always the value 1. Accordingly we just need to calculate 2 coefficients. Parameter $c$ describes the bandwidth of the bandpass, whereas parameter $d$ describes the center frequency.

$$c = \frac{tan(\pi f_b/f_s) - 1}{tan(\pi f_b/f_s) + 1} \tag{3}$$

$$d = -cos\left(2\pi \frac{f_c}{f_s}\right) \tag{4}$$

$$h_{Allpass}(z) = \frac{-c + d(1-c)\cdot z^{-1} + z^{-2}}{1 + d(1-c)\cdot z^{-1} - c\cdot z^{-2}} \tag{5}$$

One last problem which needs to be handled is the time-variance. The user of the effects-processor should be able to change the center frequency of the WahWah bandpass in realtime. The filter coefficients therefore need to be calculated depending on the desired center frequency. This operation would require the calculation of the cosine (see equation 4). A compromise was to pre-calculate a few possible coefficients and store them in a lookup table. Because these lookup tables don't need to be changed during system uptime, we used the FPGA's Block-RAM as ROM for this matter.

#### *4.5.2. Hardware Effort*

The WahWah effect requires the implementation of a second-order IIR filter. Therefore each order is represented by a VHDL process including forward and recursive branch. 4 filter coefficients out of 6 are not equal to one, therefore the filter needs 4 multiplications per sample. The Cyclone II FPGA includes 35 multipliers with a bitwidth of 18 bits. In order to gain maximum speed the multiplications are done in parallel which means, that 4 multipliers are used.

As said above the pre-calculated filter coefficients need to be saved. For this matter the FPGA's internal memory blocks are used. Considering the fact, that allpass-filter coefficients are mirrored (see [10]) and that the bandwidth is a constant there is just one filter coefficient left, which need to be saved. This single coefficient represents the center frequency and was pre-calculated for a frequency range of 200Hz to 2kHz considering, that in the end 1024 24-Bit values are ready to use. This results in 18432 bits of required memory space. The hardware unit itself uses about 400 logic elements containing 250 registers. The maximum achievable clock frequency is 60 MHz.

## 5. ADVANTAGES, DISADVANTAGES TO A DIGITAL SIGNAL PROCESSOR

The main advantage in implementing audio effects with as dedicated hardware is the very high throughput and low latency. Dedicated hardware also avoids the sometimes unpredictable behavior of a software based implementation. Because all the hardware blocks are working in parallel, we can implement complex designs and as long as we have enough free space on the chip, computing power is guaranteed. This offers us the possibility to have a huge range of effects working in parallel, because each effect works independently from the others. If we want to implement our

effects "the traditional way" by using a digital signal processor, we sooner or later will reach the calculation power limit.

Some applications can be implemented easier in software. In our case it was the control of the effects with the MIDI pedal. Therefore we decided to use also a softcore processor for managing the effects. Managing in this point means to parameterize them. Implementing also the audio effects in software (e.g. on a DSP) would have been easier than in hardware. There are currently more example-implementations available and the design process is faster in software. Also testing would have been easier, because it takes a lot of time to build a test environment in VHDL for hardware effects.

One large disadvantage of FPGAs is left to be mentioned. The point is the multiplication. Common FPGAs seldom provide DSP elements with more than 20 bits width. But digital signal processors mainly calculate with an accuracy of 32 bits. Some DSPs (e.g. *SHARC* from *Analog Devices*) even provides a 40 bit accuracy when multiplication is done. For high-end audio the bitwidth is crucial and must not be neglected. When we want to use the same broad bitwidth on FPGAs it possible to cascade the DSP elements which will in turn increase the propagation delays through combinatorics and makes the system working slower.

Additionally there are a number of well established libraries for DSP processors available, which ease and fasten the development. But on the contrary there are not that many IPs available which serve our specific needs. Furthermore, the advantage of these libraries is that they are available for free and thus have been used, tested and improved by a huge community.

## 6. CONCLUSIONS

The hardware/software co-design method described in this paper provides a practical alternative to the software centric systems dominating the market today. While dedicated hardware units used for audio realtime processing offer optimum performance in terms of latency and throughput the use of an associated software unit can still take care of the parameterization of the system and provides the flexibility a user is accustomed to.

The hardware/software co-design approach we have chosen has proved to be a practical way for the realization of even complex audio realtime effects units. Still there's lots of work left to be done. Varying bit widths throughout the course of the audio processing assembly line would be easily implementable using dedicated hardware. Among the topics we would like to address in future work are the implementation of digital audio interface standards (e.g. MADI) with special emphasis on an efficient solution for the all-pervasive problem of synchronization and the implementation of audio compression algorithms such as FLAC in dedicated hardware.

## 7. REFERENCES

[1] Klaus ten Hagen, *Abstrakte Modellierung digitaler Schaltungen*, Springer-Verlag, Berlin, Heidelberg, NewYork, 1995.

[2] Uwe Meyer-Bäse, *Digital Signal Processing with Field Programmable Gate Arrays*, Signals and Communication Technology. Springer-Verlag, Berlin, Heidelberg, NewYork, 2 edition, 2007, ISBN-13 978-3540211198.

[3] Wolfgang Pauli, Markus Pfaff, and Stefan Reichör, "Dsp in dedicated hardware: Raising value abstraction for fixed point implementation," in *International Symposium on Signals, Systems, and Electronics ISSSE 04*, Linz, Austria, August 2004, University of Linz, ISBN 3-9501491-3-9.

[4] Mario Huemer, Michael Lunglmayr, and Markus Pfaff, "A lecture course series: From concept engineering to implementation of signal processing algorithms with FPGAS," in *Proceedings of the 13th European Signal Processing Conference EUSIPCO 2005*, Antalya, September 2005, Istanbul Technical University.

[5] D. Malzner, J. Seifert, J. Traxler, H. Weber, and G. Wiendl, "Dafx project homepage," http://www.dafx-hsse.info, 2006.

[6] G. Truhlar, T. Pühringer, G. Schedelberger, M. Pfaff, and J. Langer, "Hardware/software co-design of a realtime-rendering architecture for embedded systems," in *Austrochip 2004*, Villach, Austria, October 2004, Technikum Kärnten.

[7] Brian Dipert, "FPGAs DiSPlay their processing prowess," *EDN Magazine*, , no. 22, pp. 61–68, October 2002, Avaiable online from http://www.edn.com.

[8] Nick Tredennick, "The death of DSP," http://www.ttivanguard.com/dublin/dspdealth.pdf, August 2000.

[9] Altera, *NIOS II Processor Handbook*.

[10] Udo Zölzer, Ed., *Digital Audio Effects*, John Wiley & Sons, Inc., New York, 1 edition, 2002, ISBN-13 978-0471490784.

[11] Simon Haykin, *Adaptive Filter Theory*, Prentice Hall, New Jersey, 2002, 0-13-090126-1.