Vermilya, Daniher, Victoroff
December 2012
Design and (attempted) Implementation of an Arbitrary Transfer Function Engine

Introduction

We set out to design and build a digital device capable of realizing a wide range of continuous functions for a variety of applications. Basic signal processing - filters with fewer than ten poles - crops up time and time again in daily engineering. In small embedded devices, almost every real-world input will require conditioning, from switch debounce to sensor smoothing. In larger systems, digital filter chains are used for a variety of control and processing applications, from extracting useful information from datastreams, to shaping signals such that they're more compatible with other building blocks and systems. We settled on performing digital audio manipulation using the XMOS microprocessor, accepting S/PDIF, processing it with a multitap FIR filter, and outputting the new stream as S/PDIF.

Our Design

We wanted our system to be runtime configurable, such that the exact transfer function would be input immediately before or during our signal was streaming. With this in mind, we selected the XTAG2 USB device made by XMOS - it's cheap, offered enough I/O, and most importantly had a 500MHz CPU with USB hardware broken out. This hardware, coupled with XMOS's interesting concurrency model, meant that we could have one thread expose a USB bulk transfer endpoint and receive FIR tap coefficients from a host, one thread receive S/PDIF via a 3v3 signal over a RCA connector, one thread run a many-tap FIR filter, and one thread transmit S/PDIF via a red LED. We got all of these pieces functional except for the S/PDIF transmit module due to its weird clocking requirements.

XMOS S/PDIF

The XMOS S/PDIF library comes equipped with two modules, one intended to receive raw S/PDIF byte code, and the other intended to transmit it. The documentation on these modules is scant at best, but after examining the source code we were able to discern proper use and behavior to some degree.

The receive function is relatively straightforward. The input to the SPDIF Recieve function is a 4 bit buffered port alongside a streaming channel, and a clock. This function places a stream of 32 bit words into the given streaming channel and this function should then be split into right and left channels. In the standard S/PDIF format the 4 LSBs indicate whether the word is intended for the left or right channel, if the 4 LSBs are equal to 5 it is a sign to send the value to the right channel. Additionally, the 4 MSBs are not useful information and so the 4 LSBs and MSBs are both removed and the remainder is placed into the appropriate channel.

The transmit function is significantly less straightforward and it was the implementation of the transmission alongside the receiving that our group was ultimately unable to accomplish largely due to the nearly non-existent documentation. Firstly the channel required for the transmission module is *not* a streaming channel, which means that it cannot actually be the same channel used to receive. Therefore it is required to instantiate a non-streaming channel in the sorting of the streaming channel into right and left and to write the results of the bit shifting

into that non-streaming channel.

The actual data transmission and conversion into S/PDIF format is largely done through a large lookup table that converts the non-streaming channel into S/PDIF format, but before this is actually transmitted through the XMOS chip the port must be properly configured which requires both a standard clockblock and a masterClockPort which is an input port dedicated to acting as a regular clock, which for some odd reason seemed to require an external source. One of the major problems here was that the required clock rates for the transmission seemed to be very specific when the source code was viewed, but the specifics were not actually documented anywhere outside of the source code. The example code for transmission was long complex and used a separate library that seemed to have workarounds for some of these problems, but which was not simple to implement, nor was it well suited to our actual goal.

FIR Filter Coefficients

In order to calculate the coefficients for an arbitrary multitap FIR filter, we created a series of python functions that generates coefficients for the taps. The functions are based off of a C implementation but have the advantage of being easily linked to python based USB communication and expanding the open source codebase for FIR filters. The functions allow for the implementation of low pass, high pass, band pass, and band stop filters with any desired cutoff frequency or frequencies. The actual form of the filter to implement can be Gaussian, Hamming, Hann or Blackman, with Hamming as the default. The number of coefficients, and thus the corresponding number of taps can also be specified.

Serial Communication

XMOS FIR Implementation

We chose to use the xcore sc_dsp_filters functions created by Mikael Bohman for the implementation of an FIR filter on the XMOS chip itself. Because we were not able to successfully integrate S/PDIF receiving and transmission, we did not actually implement this, but we did however explore these functions thoroughly. In order build the filter on XMOS, we would use the tap coefficients received through serial communication create an array. We would call fir.xc with our new array as the coefficients array, the number of coefficients as the desired number of taps, the data received through the S/PDIF thread as the samples, and an empty array to initialize the statetable. As new samples were received, they would be filtered and transmitted to the S/PDIF transmitting thread which would be used to play the filtered sound on speakers.

PortAudio Exploration

We additionally used an external library in C called portaudio which gave us direct access to the data being provided to our computer speakers and using a simple algorithmic implementation of digital high pass and low pass filters nested in for loops we defined a number of case statements that allowed for the implementation of these simple filter algorithms with a semi-arbitrary cutoff frequency and a semi-arbitrary number of passes. The particular implementation involved a reduction in overall amplitude which was adjusted for, using a constant multiplier of 1.2 at each filter pass, which increased the noise present, but generally not past unacceptable levels.