ELEC 490/492/498/499 Final Report

Discontinuity Handling In Single-Bit Audio Applications

Submitted by:

Group #25

Moorthy, Theepan

Nasr, Amr

Nasr, Sherif

Faculty Supervisor:

Dr. Sudharsanan

Executive Summary

Currently a new audio CD format is emerging in the market, known as the Super Audio CD format (SACD), patented by Sony – Philips corporations. This new audio processing technology so far has been implemented (i.e. SACD processors) by Sony and Philips only. However, there exists a demand for other companies to produce SACD processors, hence our project to design a specific functionality of an SACD processor.

The SACD format uses a new technology for recording and reproducing music digitally known as Direct Stream Digital (DSD, trademarked by Sony/Philips). DSD technology is radically different relative to the existing Pulse Code Modulation (PCM) technology for recording and reproducing music in that it functions entirely in a one-bit domain. The problem with designing an SACD processor is that presently no established standard methods of handling one-bit stream audio signals exist, as they do for the PCM domain.

Brand new processing methodologies are required to handle even basic features such as muting, pausing, and skipping music tracks. Our project specifically resolves the problem of pausing and un-pausing a DSD music signal smoothly (i.e. without any audible clicks even at high volume levels). More technically stated, this project deals with distortion free handling of a user-imposed discontinuity in a DSD stream. Our solution to this problem consisted of developing an actual hardware implementation that will handle the pause/un-pause functionality, based upon current literature and research papers that document theories on manipulating one-bit signals. More specifically, the hardware is a FPGA platform based implementation.

Evaluation of the designed hardware was analyzed through simulation waveform tests to verify functionality. Specifically the original DSD waveform was compared to the final DSD waveform output from our system to verify, even after our pause/un-pause functionality, that the signal remained distortion free. The waveforms proved that this was the case, and our design is set to be downloaded to an actual FPGA chip as future work.

TABLE OF CONTENTS

EXECUTIVE SUMMARY……………………………………………………….2

TABLE OF CONTENTS…………………………………………………….3

1. INTRODUCTION……………………………………………………...4
1.1 Purpose…………………………………………………………………...4
1.2 Background and Objectives……………………………………………4
1.3 Overview of Project Work……………………………………………..6
2. BACKGROUND AND MOTIVATION………………………………..6
2.1 General……………………………………………………………………6
2.2 Interface and/or Performance Specifications…………………………..7
3. DESIGN AND PRODUCTION APPROACH………………………….8
3.1 Design Requirements…………………………………………………..8
3.2 Division of Labor………………………………………………………10
3.3 Design Methods………………………………………………………..11
4. TESTING, EVALUATION AND RESULTS…………………………..13
4.1 Testing using Matlab…………………………………………………..13
4.2 Testing of the Accumulator in Altera………………………………….15
4.3 Evaluation in Altera……………………………………………………16
4.4 Software Simulation Results…………………………………………...18
5. CONCLUSION…………………………………………………………19
6. REFERENCES………………………………………………………….21
APPENDIX A Matlab Files …………………………………………………22
APPENDIX B Altera File..……………………………………………………27

1. INTRODUCTION

1.1 Purpose

The objective of this report is to deliver a summary of the work and results of the ELEC 490/492/498 project committed to by Group #25. This report is primarily directed towards ELEC 498's course instructors, our Faculty Supervisor Dr. Sudharsanan, as well as future ELEC 490/492/498 project groups.

1.2 Background and Objectives

Today leading audio companies are striving for a new technology triggered by the idea of having an audio CD with DVD audio quality (i.e. premixed settings, surround sound, etc…). There exist two main camps of companies within the industry each battling to get to that goal via two opposing routes. One of the camps still bases its research on the Pulse Code Modulation scheme (PCM), which is what we still listen to today (audio signals are sampled at 44.1 kHz). Their research and development simply consists of increasing the sampling rate (88.2 kHz) and therefore increasing the number of bits representing the analog signal, thus representing the analog signal more accurately (16, 32, 64, etc…). The other camp, on the other hand, consisting of Sony – Philips, decided to come up with a revolutionary scheme to better *map out* the analog audio signal: the Direct Stream Digital. Their work and code still remains a trade secret today aiming to incite a third party to compete in order for the new technology to be marketable, which motivated our group to derive a general solution irrelevant of Sony-Philips' encoding methodologies. The analog signal is represented in a 1-bit domain by taking the acceleration of the slope of the analog signal (the 2nd derivative of the signal). More specifically a *ramp up* of the analog signal would result in a series of *ones* (the number of ones is proportional to the speed at which the signal is changing), and a *ramp down* would result in a series of *zeros*, leaving us with the trivial zero voltage input analog signal represented by a perfect oscillation of *one-zero*. This scheme solves the problem of PCM, which consists of

basically loss of information due to sampling and quantizing of the signal. Is there a price to pay for using this new scheme? The answer is *Yes* and it will be explained why shortly. An example of DSD sampling is shown in Figure1.1 below.

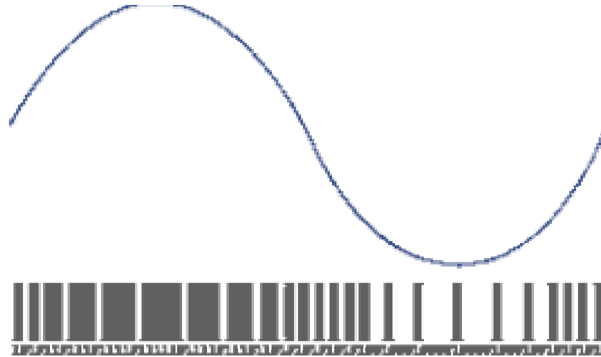


Figure 1.1 DSD sampling[1]

The goal of this project is to handle user-introduced discontinuities in a single bit stream audio signal without any audible distortion, specifically Sony's DSD audio signal. User-introduced discontinuities are caused by user-desired functionalities such as pausing and un-pausing music. Traditionally discontinuities have been easily handled in the PCM domain by adjusting signal gain levels appropriately (i.e. fading the musical signal in and out at discontinuity points). Gain level adjustments require arithmetic operations to be performed, and arithmetic operations intrinsically require a multi-bit framework. This poses no problems in the PCM domain since PCM itself is a multi-bit signal, but the question is raised when working in a 1-bit domain. How can you represent the aspect of gain or volume with merely one bit?

The underlying solution in this project is to perform the required arithmetic operations using multi-bit processes and then re-convert a multi-bit signal back into its original single bit DSD form while striving for the lowest levels of original DSD signal distortion. The final aim of this project is to produce a working product that is physically capable of performing the pause and un-pause functionality with a DSD audio signal. This product due to its required real-time nature will be based primarily on hardware aspects.

[1] http://www.superaudio-cd.com

1.3 Overview of Project work

The final goal of our project was not met because the hardware platform that we intended to use was not available on time. However the group managed to come up with a fully working software model on the Altera hardware platform.

2. BACKGROUNG AND MOTIVATION

2.1 General

The main problem in audio signal processing is distortion whether it is due to user-imposed discontinuities or due to the quantization of the signal. Using the 1-bit domain processing has solved the latter problem. The former however remains the main obstacle to matching the input and output signals with minimal noise levels.

More specifically in our project, an implementation of the pause/un-pause functionality is required. In more technical words, when a user-induced pause is applied, the input is faded out, and then faded in only when the user un-pauses the input signal, which theoretically gets rid of that audible click. However, fading in/out the signal, or in other terms cross-mixing it with a mute signal introduces more noise into the circuit. To accomplish a smooth switching between the multi-bit altered signal and the 1-bit original DSD, an accumulator acting as a memory element restoring the changes in the signal followed by a digital-to-digital Sigma Delta Modulator were used. The latter firstly helped to convert the multi-bit digital signal into single bit, and secondly to filter out the noise beyond the audible frequencies (refer to section 3 of this document for detailed operation).

To better understand the interaction between the different blocks of our circuit (refer to section 3 of this document), it was seen fit by the group members as well as the faculty advisor to start with a simulation using Matlab. Different components were implemented

and tested individually, using the graphical modules as well as custom made functions written in C-code, before integrating the circuit in its entirety.

2.2 Interface and/or Performance Specifications

The final product will be a fully programmed Xilinx FPGA development board. The board, on the front-end side, will be connected to a Personal Computer (PC) via an onboard Ethernet interface and a USB port (other peripheral connections required for the actual programming of the FPGA chip will also be present). At the back-end side, the board will be connected to speakers that have analog volume control capabilities. Please see Figure 2.1 for an Operational System Diagram.

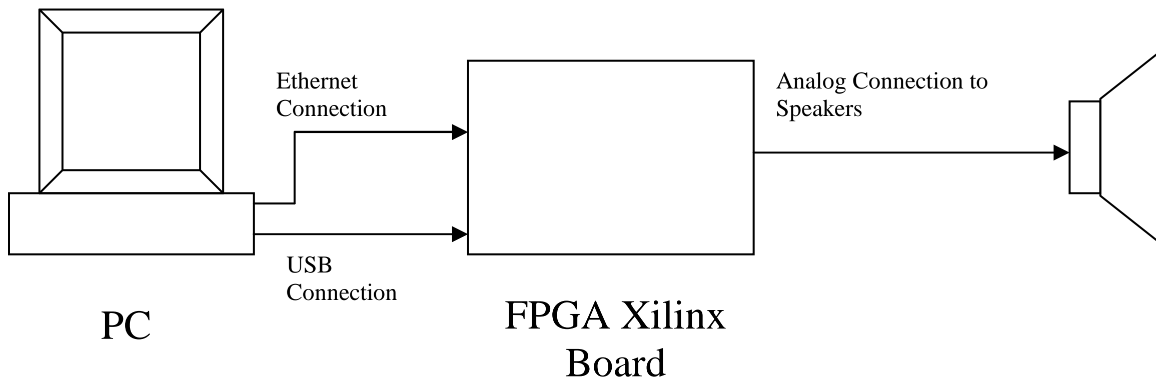


Figure2.1 Operational System Diagram

The board will take in a DSD music signal from the PC in real-time. Up until a pause is requested by the user, the music signal will simply be passed through the device via Route A without any significant alternations/processing- with the exception of it lastly being converted to an analog signal for the speakers (Please see Figure 3.1 for indications of signal path lines a and m). This outputted analog signal would be amplified by the external speakers and played at audible volumes so that distortion free music can be heard while users pause and un-pause. When the user wishes to pause the music, the output line to the speakers will be switched to disconnect the speakers from signal path Route A and connect them to a second route of the incoming DSD signal (Route M).

Then the required processing will be performed by the components in Route B. Through this processing for the pause and appropriate switching mechanisms the demonstrational playback music will be outputted as a non-clicking analog signal for a smooth fade out of the music. A certain time later when the user wishes to un-pause the music- again the components in signal path Rout B of the device will accomplish a smooth fade in of music. Then finally a switch from Route M to Route A will be made to return to normal music playback mode. The primary input interface between the PC, which generates the DSD music signal, and our device is the Ethernet connection.

3. DESIGN AND PRODUCTION

3.1 Design Requirements

The following system block diagram illustrates the key components of our audio processing system:

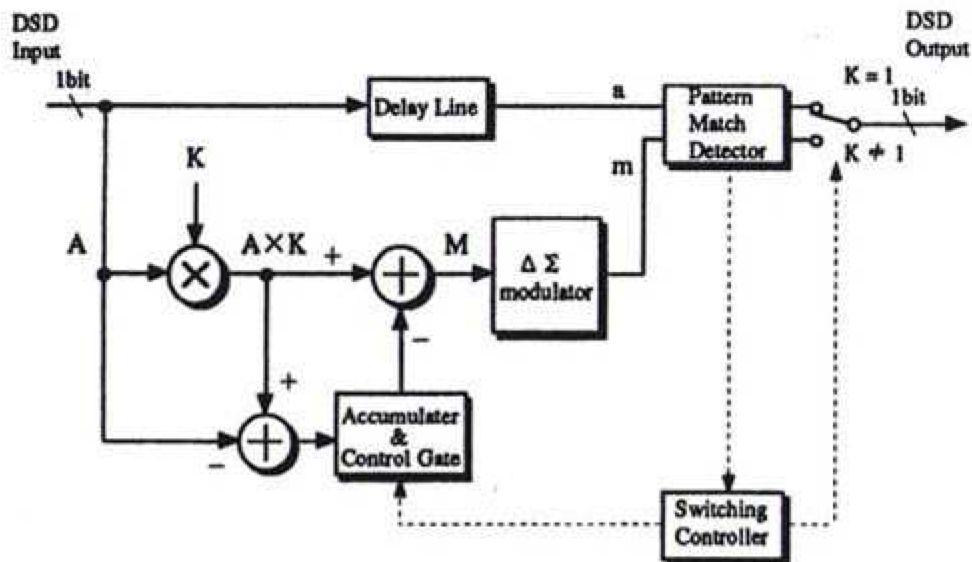


Figure 3.1. Early operational diagram[2]

The hardware is internally divided into the following operational components:

[2] http://www.airjohn.com/dsd/dsd.html

Mixer, Accumulator, Σ-Δ modulator, Delay Line, Pattern Match Detector, and Switching Control Logic Unit. In the first term initial project development consisted of designing each of these components in Matlab's Simulink according to their individual functionality requirements as listed below.

- The Mixer is required to fade in/out the original DSD signal when desired and mix in an internally generated mute signal (a perfect oscillation between –1 and 1) when silence is required. This is done by multiplying the input signal by a reduction factor K, while simultaneously increasing gain on the mute signal by another factor, both signals are now outputted as multiple-bit signals. The modified DSD signal is then compared with the original signal by performing add/subtract operations and this information is passed on to the Accumulator as input.
- The Accumulator takes the compared signal mentioned above, which has become a multiple-bit signal as well and stores it, or more specifically stores the opposite of what was done to the original DSD signal in terms of gain controls.
- The Σ-Δ modulator is used to reconvert a parallel multiple-bit signal back into a single-bit serial DSD signal. Here the Σ-Δ modulator will take in the modified multiple-bit signal after subtracting all kinds of noise (via the output of the Accumulator), and reconvert it to a DSD signal. There are different orders to the Σ-Δ modulator. The higher the order is, the better the resolution for the multiple-to-one bit conversion is. For our implementation a 4th order Σ-Δ modulator was utilized.
- The Delay Line has as its input the original DSD input, and as its output the delayed version of that original input. This delay will be relative to the "time" needed for the signal to be faded in/out by the Mixer, plus the time needed for the Accumulator to re-inject the correction values after the Mixer has finished fading in and out, and plus the time that it then takes for the Σ-Δ modulator to do the multi-bit to single-bit conversion.

- The Pattern Match Detector is a two-input block with a flag to the Switching Control Logic Unit as output. It constantly checks if the original and modified DSD signals are matching in real time, and sets a 'Match Flag' accordingly.
- The Switching Control Logic Unit's primary role is to determine the most optimal times to make a switch from the audio out line being connected to the original DSD signal line to it being connected to the output of the Σ-Δ modulator during the pause/un-pause cycle and then back again. Its subsidiary role also includes the activation and deactivation of the Mixer block, through the assertion and desertion of an internal pause signal. The internal pause signal (controlled by the Switching CLU itself) has to be different from the master pause signal (controlled by external pause/un-pause requests by users themselves) since fading in and out by the Mixer can not be started the instant a user request is received. Whenever an intentional discontinuity is performed (pause/mute/skip), the Switching Controller first waits until it has determined via the Pattern Match Detector's output that a distortion free switch can be made to our system's processing line, makes that switch, and then only at this point will it assert the internal pause signal which activates the Mixer component. In a similar backwards fashion it accomplishes the switch back to the original DSD line when the user's un-pause request is received. Please note that only the Mixer component requires input from the Switching Controller, all other logic blocks continually run in real-time upon system power up independent of any control from the Switching Controller.

3.2 Division of Labor

The operational system was divided into two main parts. The first part is the Σ-Δ modulator, which will be handled by one group member, since it constitutes one third of the entirety of the design. The second main part consists of the remainder of the components described in 3.1, which can be subdivided into two smaller parts between the two other group members. One logical grouping of work for the 2nd group member consists of the Mixer & Accumulator, while the second logical grouping of work for the 3rd member consists of the Switching Controller and the Pattern Match Detector. These

groupings were chosen such that an adequate understanding of one of the components within a grouping will eventually help in implementing the other components in the grouping.

3.3 Design Methods

In Matlab's Simulink a combination of available signal blocks in Simulink's library and custom built blocks using C in Simulink's user defined S-Blocks were used to compile the initial system model. Please refer to Appendix A for a schematic diagram of the Matlab model, and the C code (which are listed in the following pages of the same Appendix section) that makes up the custom blocks. Please note that not all of the components that were listed in Section 3.1 were modeled in Matlab. Specifically the Pattern Match detector and the Switching Control Logic Unit were purposely chosen to be excluded in the Matlab modeling phase since these components were strictly more real-time based and could be more easily implemented in the FPGA design environment.

Referring to the Matlab schematic it can be seen that the Mixer component is compromised of the following sub-blocks: K_DSD, K_Mute, Mute Signal Generator, Counter, Product of DSD & K_DSD, Product of MUTE & K_Mute. The K_DSD and K_MUTE sub-blocks were modelled using C. They both manipulate a K factor variable according to the input Pause Control Signal sent by the CLU. The K_DSD block upon the assertion of the pause signal will take the initial value of 1 for the K factor gradually down to 0. As can be seen in the C code for the K_DSD, K factor variable reduction is done in minute increments of 0.000709. The time period chosen for each reduction decrement is 10 DSD bits or 3.5 microseconds. This 3.5 microsecond reduction interval combined with the specific reduction decrement of 0.000709 results in the K factor being brought down to 0 from 1 in roughly 5 milliseconds. And 5 milliseconds was the desired design requirement for the periods of fading out and fading in. Similarly the K_MUTE block manipulates the K mute factor variable in exactly the opposite manner, where the

initial value of 0 is brought up to 1 in increments of 0.000709. Timing for these two K factor manipulation blocks is kept externally through the use of a counter block. This counter block counts up to 10 DSD bits and then rolls over back down to 0 again. The value of 10 generated by the counter is used by the two K factor blocks to execute the next decrement/increment. The MUTE Signal Generator block itself serves as the clock for the Counter block. This is feasible since the MUTE Signal itself is a perfect oscillation between 1 and –1, and is set to have a frequency exactly equal to the period of two DSD bits. Therefore this serves as a convenient method for keeping the entire system in sync with the actual DSD bit stream. Finally the two K factor variables are multiplied with the two DSD and MUTE signals that they are required to manipulate via the two 'Product of' blocks respectively. Once the single-bit DSD and MUTE signals pass through these two blocks respectively, they will have now been converted to multi-bit signals, which have gain information contained within them.

The Accumulator was designed in a single complete custom block using C to accomplish its full functionality. The accumulator internally consists of a level variable and DelSig Input variable. The level variable is used to accumulate the offset differences to the DSD signal in a cyclic manner (i.e. rolling the magnitude over from 1 to –1 or rolling under from –1 to 1, so that at any give time the absolute magnitude of error is kept within the required range of –1 to 1). The DelSigInput variable is used to actually apply the error correction value of plus or minus 0.177154195e-3 as output. This value is derived from the requirement that even if the level variable comes to rest at a maximum magnitude error of –1 or 1, this error should be restored to 0 within a maximum operation time of 2 milliseconds. If the level variable rests at values under magnitudes of –1 and 1, the accumulator will thus complete the error correction process faster than the 2 millisecond maximum requirement.

The Σ-Δ modulator was designed entirely using basic operational blocks available in Simulink's library, with the exception of the Quantizer sub-block which had required custom C code. It was designed to meet the requirement of a 4th order modulator, as can be seen in the Matlab Schematic.

Once the Matlab model of the system had been completed, and the functionality of each block tested individually and together as a complete system, implementation work of the system on an FPGA platform was commenced in the 2nd term. Altera's development environment was used to develop schematic design entry via graphic blocks, which contained the custom logic functionality coded in VHDL. It was not too difficult to convert the previously completed C code from the Matlab environment to VHDL code in Altera's environment. However, a few minor details required special attention. For example initialization of variables could not be done simply through code as was the case with C in Matlab. In VHDL since these variables represented actual physical signals they had to be externally initialized using separate initialization pulse blocks which would be active during power-up of the system and then remain stable and inactive while the system was running.

The complete graphic and VHDL design files are provided in Appendix B, please refer to this index for the specific implementation details of each logic block.

4. TESTING, EVALUATION, AND RESULTS

4.1 Testing using Matlab

As mentioned in our design approach of this project, Matlab software was used to simulate the functionality of our operational system. Specifically, illustrative models of the different components of the circuit as well as real-time models were implemented so as to give a better understanding of their behavior. This provided us with more insight as to what type of waveform patterns to expect at the output, and helped us in implementing our final architectural design.

In our Matlab design, there were three major components that needed to be tested in order for the group to move on to the Altera software implementation: the Mixer, the Accumulator, and the Sigma-Delta Modulator.

The mixer was meant to multiply the DSD music signal by a factor K, thus decreasing its gain from one to zero, or more technically stated, perform the fading out and fading in of

the DSD music signal. Since there is no such thing as “silence” in a 1-bit system, a mute signal was simultaneously multiplied by a factor pulling its gain from zero to one. This cross-fading of the music signal is triggered by the assertion of the pause signal and is shown in Fig.1 of Appendix A. On this figure, we can see three different waveforms: that of the DSD signal, the mute signal, and the pause signal, from top to bottom respectively. It is clear that when the pause signal is asserted, the mute signal, which was initially at zero, starts increasing until it reaches one; conversely, the value of the DSD signal, which was initially at one, starts decreasing until it reaches zero. Furthermore, when the pause signal is de-asserted, the opposite behavior happens: the DSD goes from zero back to one whereas the mute signal goes from one back to zero. Clearly, these test waveforms proved the proper functionality of our Mixer. Moreover, since the representation of gain, volume, or multiplication is unconceivable in a 1-bit domain, the Mixer converted the 1-bit DSD music signal to a multi-bit signal.

As for the Sigma-Delta Modulator (SDM), its purpose was to bring back the multi-bit signal obtained from the mixer to the 1-bit domain, as well as filtering the noise beyond the audible frequencies. Getting a replica of the 1-bit input signal would denote not only the proper behavior of the SDM but the proper functioning of the entire circuit since the latter component is connected to the output of the Accumulator. The model used for this component was illustrative, therefore not functioning in real-time, because this required creating a random DSD signal and over-sampling it by a factor of 64 (process that could not be done in Matlab), then inputting it into the SDM. Thus, testing of the filtering of the noise could not be done. However, the output of the SDM was tested with our DSD input being the mute signal because of its trivial nature. The mute input signal and the resulting output waveform of the SDM are shown in Fig.2 of Appendix A. The mute signal is given by the top waveform, whereas the bottom one shows the corresponding SDM output. We can see that the output of the SDM is an exact replica of the 1-bit DSD input with the exception of the width of the pulse. This broadened pulse accounts for the time delay introduced by the 4 orders of the SDM. Therefore, proper functioning of the SDM was proved.

The Accumulator, however, could not be tested using Matlab. The main purpose of the Accumulator was to store the changes experienced by the DSD signal and account for the

errors introduced by the mixing operations, and thus this process could not be demonstrated using Matlab. Therefore, thorough testing of the Accumulator was done in Altera using waveforms simulations. This is discussed in the next subsection.

4.2 Testing of the Accumulator in Altera

The accumulator does not provide any input to the circuit unless the pause is asserted and then de-asserted; that is, when the pause is asserted the accumulator starts storing the changes experienced by the signal, and only when the pause signal is de-asserted that the accumulator starts to inject the opposite of these changes into the SDM. Part of our simulation results, which were done using Altera, consisted of a simulation of the correction of the accumulator, which is shown in Fig.4 of Appendix B. If we pay attention to the “kdsd” and “kmute” signals we notice that the former reaches a value of 10000 (representing a ‘1’) and the latter reaches a value of zero which denotes that pause signal was de-asserted and the gains of the DSD signal and the mute signal are back to one and zero respectively. In the meantime, the value of “accMAG” was still increasing, hence storing changes into the accumulator, but one clock cycle after the de-assertion of the pause signal the accumulator starts decreasing until it reached a value of zero. These constant decrements of the accumulator level correspond to the “injection” of the opposite changes into the SDM. And therefore, the Accumulator is tested and demonstrated proper behavior.

During the development and design of the project, MATLAB will be used to simulate the functionality of our operational system. This will provide us with more insight into what type of signals/waveforms to expect as output, and will aid us in implementing our final hardware design.

In our final testing phase, a PC (capable of sending DSD signals) will be connected to the FPGA board, which will in turn be connected to speakers (with analog signal amplification control). Firstly, our FPGA circuitry will be left out of the loop and a pause

and un-pause will be performed from the PC- for which very obvious distortion noises should be heard. The same experiment will then be performed again with our FPGA circuitry active between the PC and speakers. This time a listener should hear a smooth fade out and fade in of the music signal without any distortion noises.

Time permitting, other functionalities of our operational system could be exploited such as muting or skipping a track without any distortion noise. Taking it a step further, we could implement a more interesting functionality: mixing two tracks, which basically is the muting functionality done with a “non mute signal”.

4.3 Evaluation in Altera

Without a FPGA board, the only other alternative solution to provide a working model was to have a software implementation of the circuit, as opposed to a hardware implementation. For this purpose, Altera software was used, and since it is a completely different environment, some difficulties arose as to how to translate our now-working Matlab model to an Altera model. For instance, a number representation system had to be chosen. Firstly, the IEEE Floating point number representation was considered but discarded because of the computational complexities thereby introduced. Consequently, a signed magnitude number representation was chosen where the numbers were scaled up by a factor of 10,000 in order to avoid the ambiguities introduced by decimal numbers, and were represented by 15 bits. This significantly simplified the functionality of the mixer as well as the accumulator. However, in the SDM block, the remainder of the divisions had to be rounded up or down according to its most significant bit. A “zero” means that the magnitude of the decimal is less than 0.5 and remainder would be rounded up, and the opposite is true for a one. If we were to round up, then a one would be added to the result of the division. Please refer to Appendix B for the SDM schematic. Only the top hierarchy was shown for neatness and clarity of the report.

Another problem that was faced in Altera was the speed of operation of the software itself. Specifically, the components in Matlab were modeled in real-time; however, in Altera, the cross-fading of the DSD input signal and the mute signal, for example, would require approximately 15,000 clock cycles to complete, which is equivalent to 45 minutes

of simulation time in Altera, and only after the cross-fading is finished that the accumulator starts injecting back the errors. This is the main reason why a Xilinx FPGA board was needed since it operated at a much faster speed.

The Altera model was divided into two main components: the SDM and the rest of the circuit. The SDM was composed of 4 orders, each order having the same logic blocks, namely adders, subtractors, dividers, unit delays, and a quantizer at the end. Therefore, it was more practical to implement this component using the Graphic Design Files (*.gdf) provided by the Altera environment. The rest of the circuit, on the other hand, required more specific and block-oriented computations, and hence was designed in VHDL code. The use of VHDL, however, resulted in some difficulties of its own. More specifically, having chosen the signed magnitude number representation, computations as simple as an addition or a subtraction could not be performed, since the only way to recognize a positive or a negative number was to check the Most Significant Bit (MSB) of the number. A solution to this problem was to represent all the variables in the circuit by their MSB as well as their magnitude, which was given by the 14 remaining bits, and treat them separately.

A schematic of the entire Altera model is given in Fig.1 of Appendix B. This figure shows all the components that were done in Matlab as well as new components, such as the Pattern Match Detector, the Control Logic Unit (CLU), and the Delay Line.

The purpose of the delay line is to account for the delay introduced by the circuit under a pause assertion. The delay introduced by the SDM was found to be equivalent to 6 clock cycles, and the delay of the combination of the Mixer and Accumulator was found to be equivalent to 2 clock cycles. Therefore, an 8-bit shift register was designed for the delay line, each bit accounting for one clock cycle.

As for the pattern match detector, a match of 10 bits between the processed signal and the original signal was sufficient for proper functionality.

The CLU commands the switching between the original signal and the processed signal only after having a consecutive 10-bit match between the two signals. Moreover, this logic unit has as one of its inputs, a Master-pause, which represents the physical pausing and un-pausing of the music by the user. A point worth mentioning is how the Master-pause affects our electronic circuitry. For reference, fig.2 of Appendix B shows the

activation of the internal pause signal resulting from the activation of the Master-pause. The variable “pcount” from the figure denotes the count inside the pattern match detector. From this figure, we can see that as the Master-pause is asserted at 470ms, pcount start increasing from zero, and when pcount reaches the value of 10, that means that 10 consecutive bits are matching and the CLU, in turn, asserts the internal pause signal. An analog situation to this one is the pause deactivation, which is shown in fig.3 of Appendix B. When the Master-pause signal is de-asserted, this results in the de-assertion of the internal pause signal only one clock cycle later. In this case, the CLU does not have to wait for a 10-bit match, because the switching back to the original circuit occurs only when the output of the SDM is matched to the original signal, and therefore does not depend on the de-assertion of the internal pause signal.

Waveform simulation results done in Altera are discussed in the next subsection. These simulation results represent component functionalities, which are put to test, denoting proper functionality of the model.

4.4 Software Simulation Results

Initially, in the presence of a Xilinx FPGA board, our demonstration of a working model would have consisted of inputting a DSD signal from a PC to the FPGA which would have been in turn connected to amplified speakers, and the listener should have heard a smooth fade out and fade in of the music signal without any distortion noises.

However, in our case, proper functioning of our model consists of proving that every component works as expected and thus obtaining a match between the output of the SDM, which is also the output of the entire circuitry, and the original DSD input signal.

The first simulation results deals with the behavior of the accumulator. Referring to Fig.4 of Appendix B, we see that both the Master-pause and internal pause signal are de-asserted, meaning that the DSD input gain is increasing to 1 and the mute signal is decreasing to zero, which in turn orders the accumulator to inject back the errors experienced by the signal into the SDM. These injections are repeated every clock cycle until the accumulator level goes back to zero. As for the value of the injections, inserting too many injections in a short period of time would result in a distorted signal, and

similarly inserting a few injections in a long period of time would result in an altered signal; so, the optimal period of time found was 2 msec which corresponds to a value of 2 in our number representation, and which is denoted by the variable “delMAG” on the same figure.

Moreover, as discussed in the previous subsection and shown in Fig.2 of Appendix B, the internal pause is asserted after establishing a match between the original signal and the processed signal, in which case the gains of the DSD input signal and the mute signal start to decrease to zero and increase to one respectively. Furthermore, altering the gains of these two signals orders the accumulator to start storing these changes as shown in Fig.2.

On a final note, having demonstrated the proper interactions between all the components in the circuitry, a match between the final DSD signal and the original DSD signal should be proved. Referring to Fig.5 of Appendix B, we see that the accumulator level is brought down to zero and the error injections to the SDM are stopped as expected. In the meantime, the pattern match detector is checking for a 10 bit match between the final DSD signal, which corresponds to the output of the SDM, and the original DSD signal. Thus, when a value of pcount equal to 10 is reached, a steady match is obtained, and the flag in the CLU is asserted allowing proper switching between the processed signal and the original signal. Therefore, proper working of our model is demonstrated.

It should be stated, however, that we were limited as to the randomness of the input DSD signal since, in Altera, the DSD waveform could only be done manually. Thus, future work would consist of downloading our Altera model onto a FPGA board, namely Xilinx (because of its operating speed), as well as connecting the board to a PC generating a random DSD signal on the one end, and speakers (with analog signal amplification control) on the other end, and hence demonstrate a distortion-free pausing and un-pausing of the DSD music signal.

5. CONCLUSION

Currently, there are not many devices that can handle discontinuities in the DSD domain. Therefore this project provides an FPGA downloadable implementation for a working

device that could handle a pause – unpause functionality when listening to a DSD music signal. In order to accomplish this, it was required as a first step to convert the single-bit DSD signal into a multi-bit signal which could hold gain level (volume) information, and then using a Σ-Δ modulator reconvert the multi-bit signal back to its original single-bit DSD domain in a distortion free manner. The circuitry required to accomplish this task consisted of devices such as a mixer, an accumulator, a pattern match detector, as well as a control logic unit. Consequently, a Matlab model was required in order to create algorithms for these devices and provide a simulation of the entire system. This Matlab simulation was followed by a hardware implementation in Altera's development environment using primarily VHDL code and graphical schematic blocks. The functionality of the implementation was verified to be correct and functioning through waveform results. As future work, the design files need to be downloaded into an actual FPGA chip to build a physical device, which would run our designed system.

6. REFERENCES

1. DSD – Thesis, European Media Master of Arts, Arjan Van Asselt, Summer 2000
   http://www.airjohn.com/dsd/dsd.html
2. Care and Feeding of the One Bit Digital to Analog Converter:
   http://www.ee.washington.edu/conselec/CE/kuhn/onebit/primer.htm
3. The Xilinx website:
   http://www.xilinx.com
4. SONY CXD2753R reference manual
5. SACD Sony webpage:
   http://www.superaudio-cd.com

APPENDIX A – MATLAB FILES

Matlab System Schematic

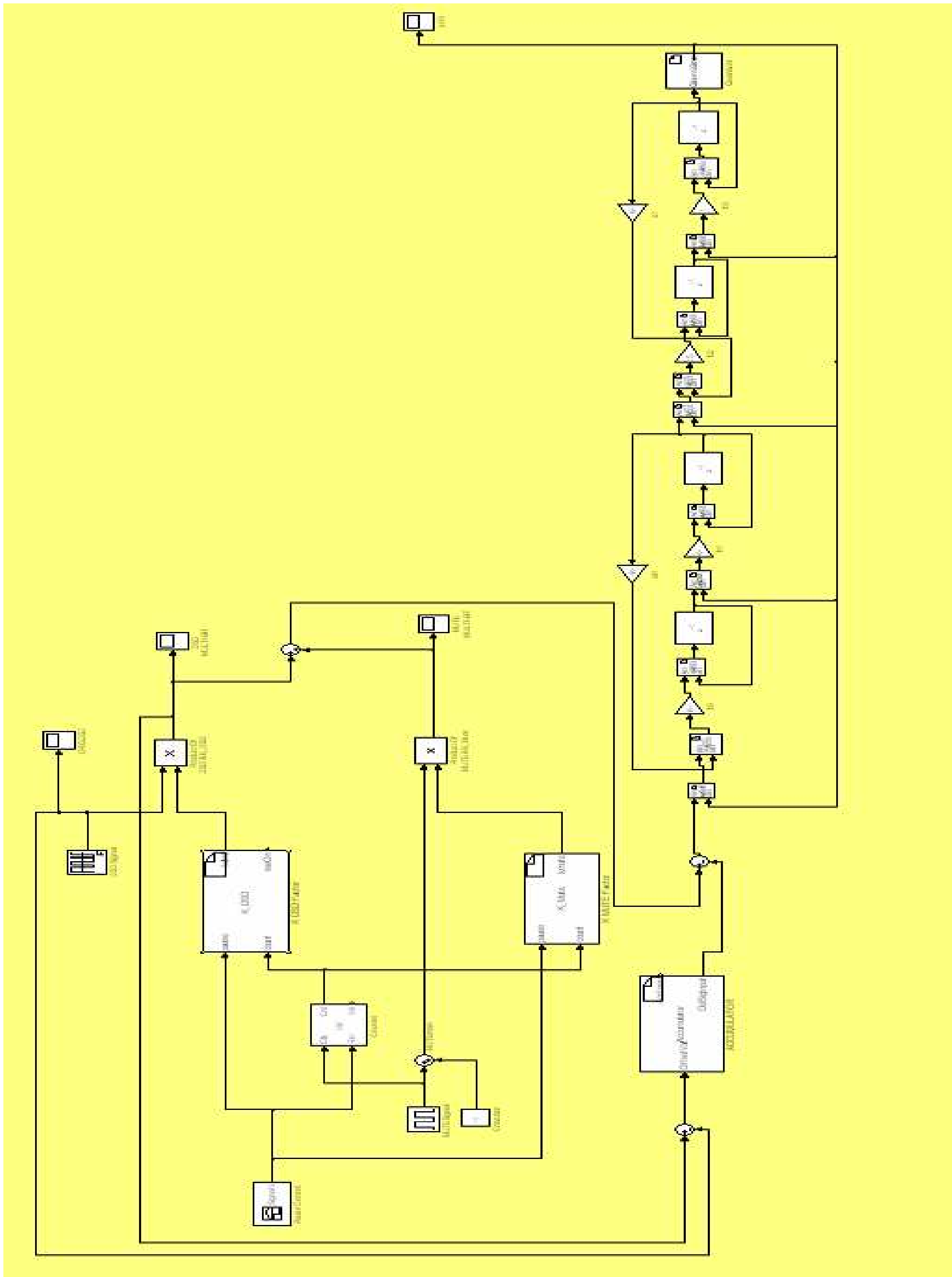


Fig.1. Matlab Schematic

Matlab C code for Custom Blocks

K_Mute Code:

```
if (*pause == 0) {
    if (*kmute < 0.000709) *kmute = 0;
    else if (*count == 0)*kmute = *kmute - 0.000709;
}/* ends if */

if (*pause == 1) {
    if (*kmute > (1-0.000709)) *kmute = 1;
    else if (*count == 0)*kmute = *kmute + 0.000709;
}/* ends if */
```

K_DSD Code:

```
*swiOn = *swiOn + *pause;

if (*swiOn == 0) *kdsd = 1;

if (*swiOn >= 1) {
    if (*pause == 0) {
      if (*kdsd > (1-0.000709)) *kdsd = 1;
      else if (*count == 0)*kdsd = *kdsd + 0.000709;
    }/* ends rising if */
    if (*pause == 1) {
      if (*kdsd < 0.000709) *kdsd = 0;
      else if (*count == 0)*kdsd = *kdsd - 0.000709;
    }/* ends falling if */
      *swiOn = 1;
}/* ends swiOn=1 if */
```

Accumulator Code:

```
*DelSigInput = 0;

*AccLevel = *AccLevel + *OffsetVal;
if (*AccLevel < -1) *AccLevel = 1 + (*AccLevel + 1);
if (*AccLevel > 1)  *AccLevel = -1 + (*AccLevel - 1);

if ( (*OffsetVal == 0) && (*AccLevel != 0) ){
        if (*AccLevel > 0){
                if (*AccLevel < 0.177154195e-3) *AccLevel = 0;
                else {
                        *AccLevel = *AccLevel - 0.177154195e-3;
                        *DelSigInput = 0.177154195e-3;
                }/* ends ELSE */
        }/* ends IF for AccLevel decrementation */
        if (*AccLevel < 0){
                if (*AccLevel > (0 - 0.177154195e-3) )*AccLevel = 0;
                else {
                        *AccLevel = *AccLevel + 0.177154195e-3;
                        *DelSigInput = -0.177154195e-3;
                }/* ends ELSE */
        }/* ends IF for AccLevel incrementation */
}/* ends If for AccLevel Reduction */
```

Matlab Waveforms:

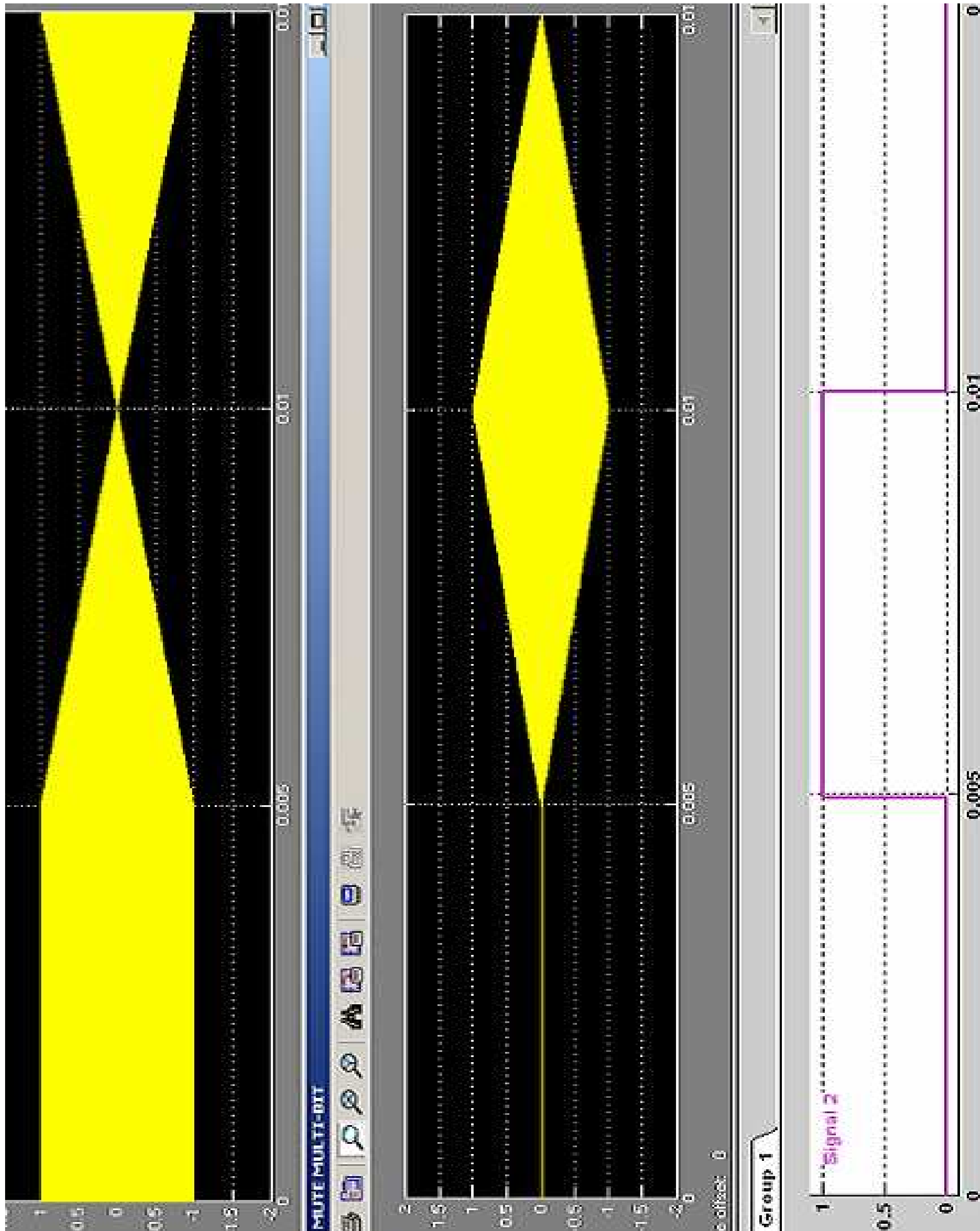


Fig.2. Cross-fading of DSD input signal and mute signal with respect to pause signal

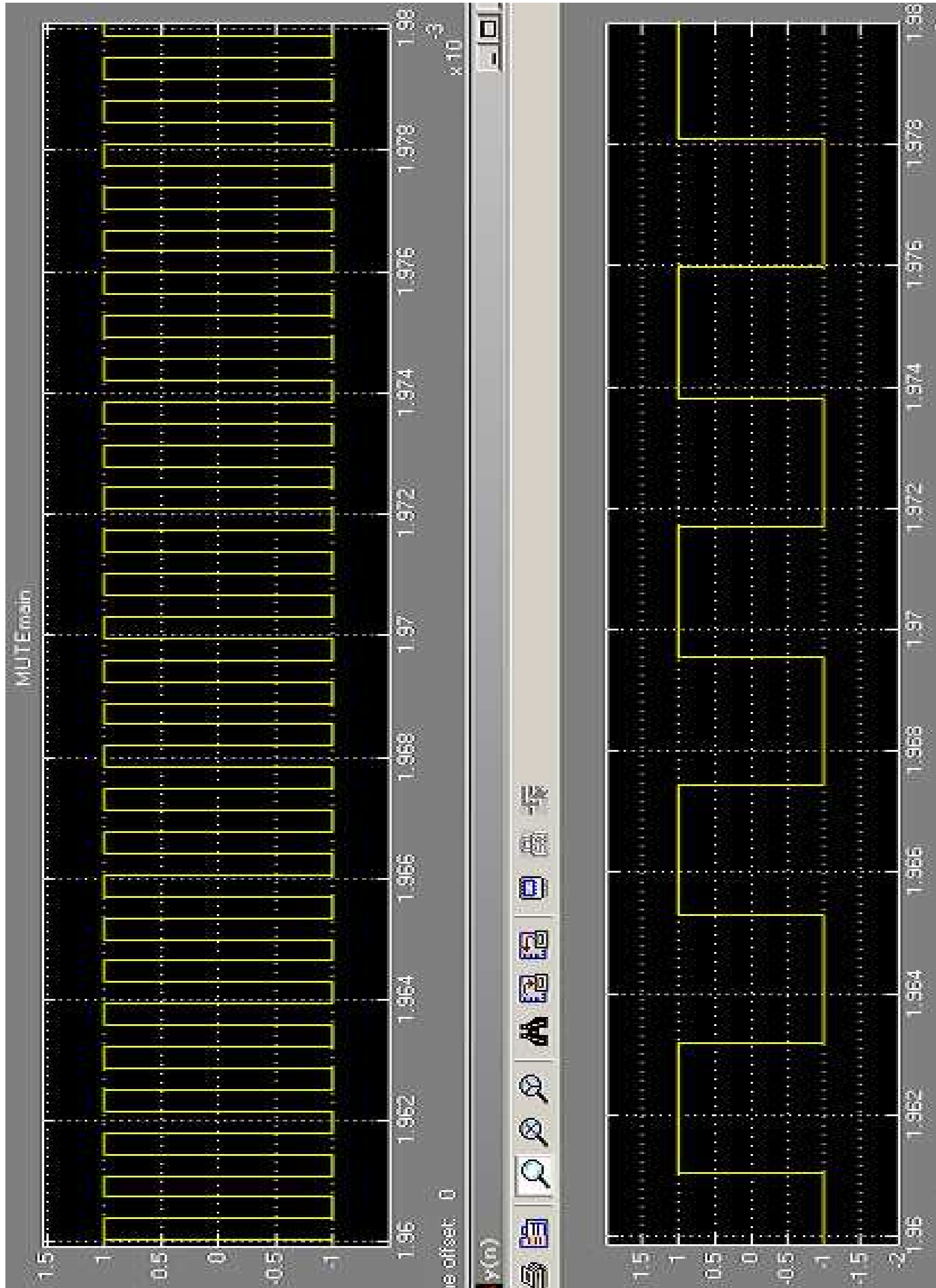


Fig.3. Mute signal as input and corresponding SDM output

APPENDIX B

Altera System Schematic

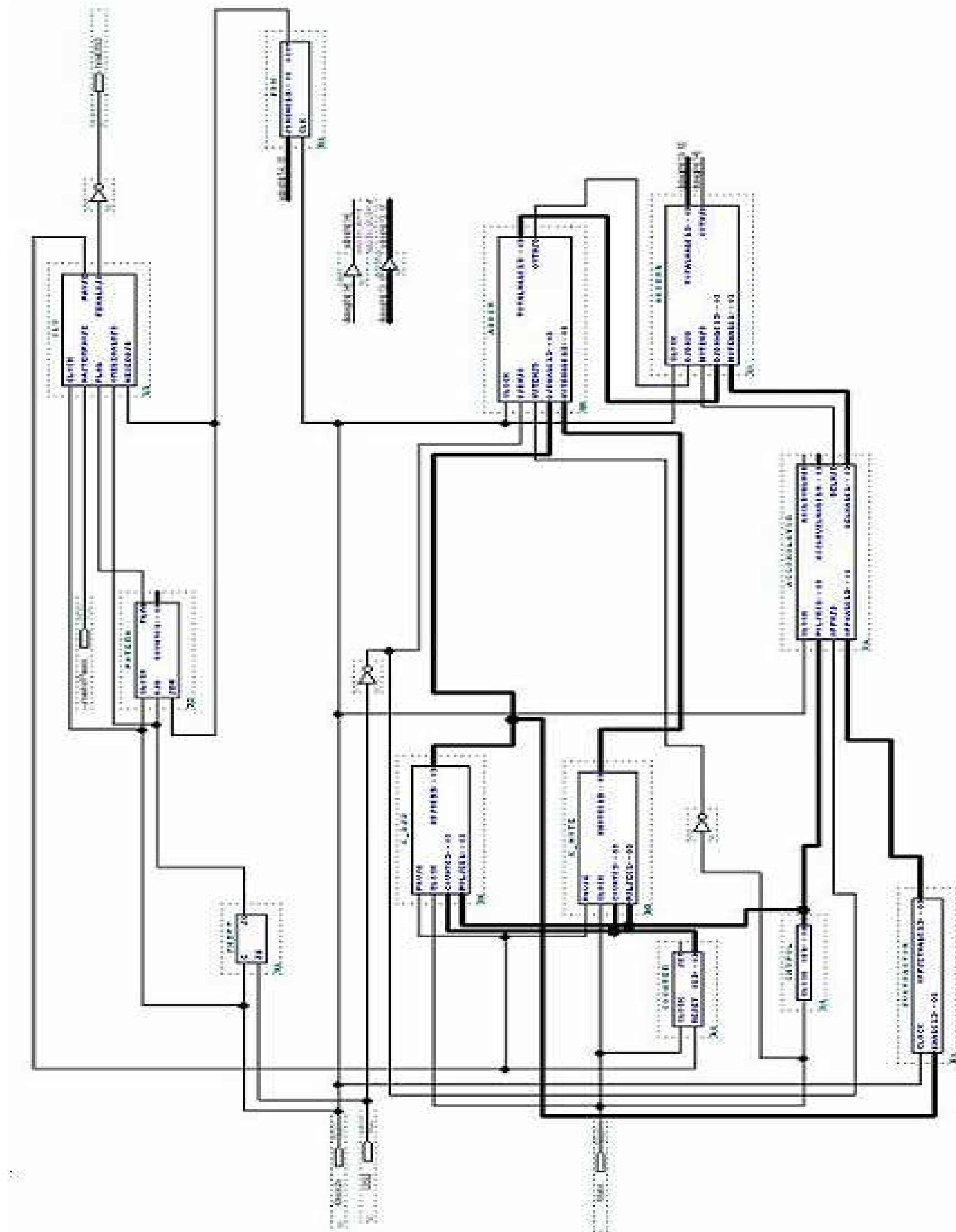
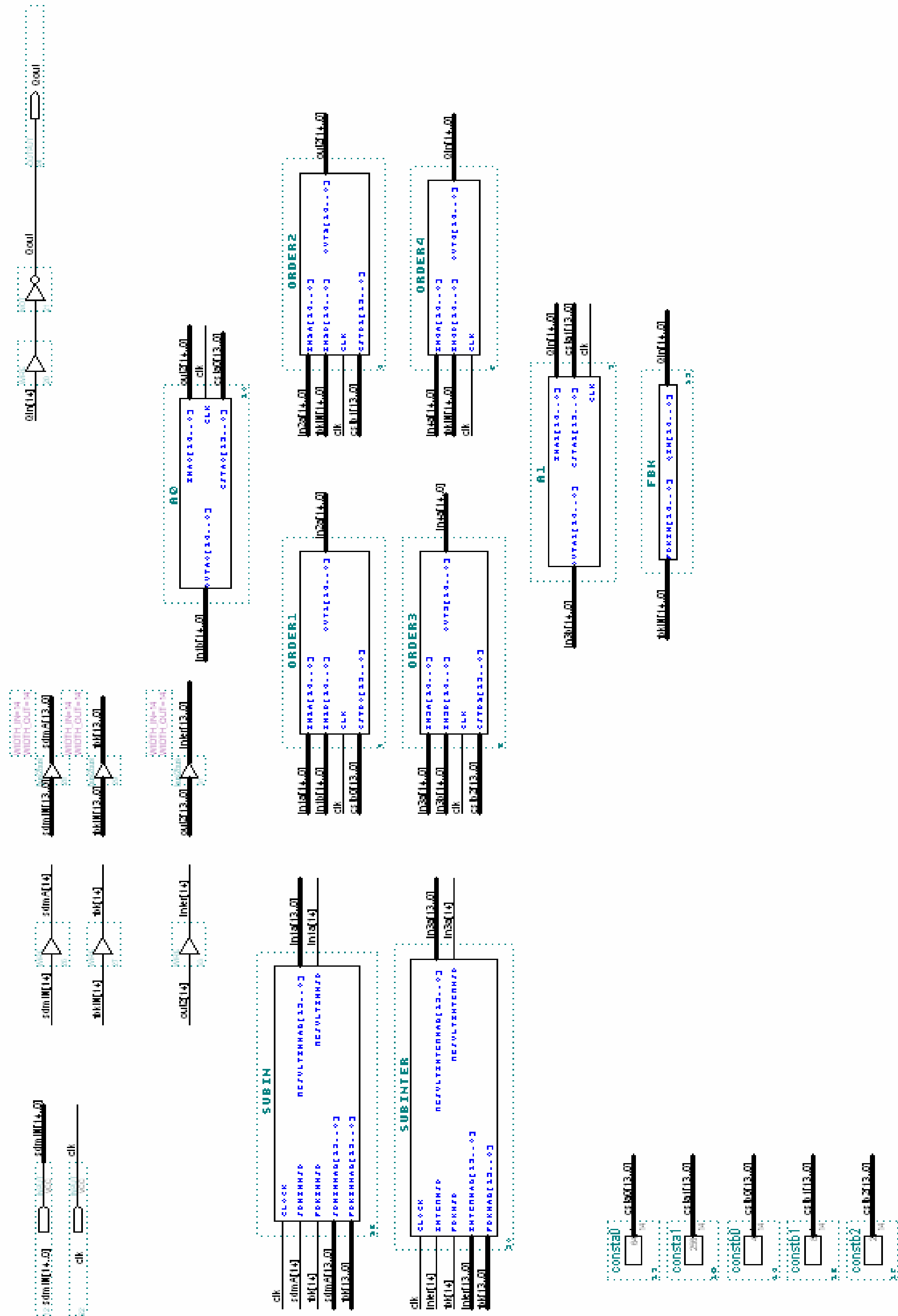


Fig.1. Altera Schematic

Altera SDM Schematic



<u>Altera VHDL files by Component</u>

<u>K DSD Block:</u>

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY K_DSD IS
    PORT ( Pause, Clock : IN STD_LOGIC;
        Count : IN INTEGER RANGE 0 TO 8;
        Pulse : IN INTEGER RANGE 0 TO 2;
        kdsd : BUFFER INTEGER RANGE 0 TO 10000);
END K_DSD;

ARCHITECTURE Behavior OF K_DSD IS
BEGIN
    PROCESS (Pause, Count, Pulse, Clock)
    BEGIN
        IF Pulse=1 THEN kdsd<=10000;

        ELSIF rising_edge(Clock) THEN

            IF (Pause='0' AND Pulse=2) THEN
                IF kdsd>9993 THEN kdsd<=10000;
                ELSIF Count=0 THEN kdsd<=(kdsd+7);
                END IF;

            ELSIF (Pause='1' AND Pulse=2) THEN
                IF kdsd<7 THEN kdsd<=0;
                ELSIF Count=0 THEN kdsd<=(kdsd-7);
                END IF;
            END IF;
        END IF;
    END PROCESS;
END Behavior;
```

<u>K MUTE Block:</u>

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY K_Mute IS
    PORT ( Pause, Clock : IN STD_LOGIC;
        Count : IN INTEGER RANGE 0 TO 8;
        Pulse : IN INTEGER RANGE 0 TO 2;
```

```
                kmute : BUFFER INTEGER RANGE 0 TO 10000);
END K_Mute;

ARCHITECTURE Behavior OF K_Mute IS
BEGIN
        PROCESS (Pause, Count, Pulse, Clock)
        BEGIN
                IF Pulse=1 THEN kmute<=0;

                ELSIF rising_edge(Clock) THEN

                        IF (Pause='0' AND Pulse=2) THEN
                                IF kmute<7 THEN kmute<=0;
                                ELSIF Count=0 THEN kmute<=(kmute-7);
                                END IF;

                        ELSIF (Pause='1' AND Pulse=2) THEN
                                IF kmute>9993 THEN kmute<=10000;
                                ELSIF Count=0 THEN kmute<=(kmute+7);
                                END IF;
                        END IF;
                END IF;
        END PROCESS;
END Behavior;
```

CNTPUL Block (Initialization Pulse):

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY cntPul IS
        PORT ( Clock : IN     STD_LOGIC;
                Q                       : BUFFER INTEGER RANGE 0 TO 2 );
END cntPul;

ARCHITECTURE Behavior OF cntPul IS
BEGIN
        PROCESS (Clock)
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Q=2 THEN Q<=2;
                        ELSE Q<=(Q+1);
                        END IF;
                END IF;
        END PROCESS;
END Behavior;
```

Counter Block:

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY counter IS
    PORT ( Clock, Reset : IN      STD_LOGIC;
           Set                    : BUFFER STD_LOGIC;
           Q                      : BUFFER INTEGER RANGE 0 TO 8 );
END counter;

ARCHITECTURE Behavior OF counter IS
BEGIN
    PROCESS (Reset, Clock)
    BEGIN
        IF Reset='0' THEN Set<='1';
        END IF;
        IF (Reset='1' AND Set='1') THEN Q<=0; Set<='0';
        ELSIF rising_edge(Clock) THEN
            IF Q=8 THEN Q<=0;
            ELSE Q<=(Q+2);
            END IF;
        END IF;
    END PROCESS;
END Behavior;
```

Subtractor Block:

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY subtractor IS
    PORT ( Clock : IN STD_LOGIC;
           KMag : IN INTEGER RANGE 0 TO 10000;
           offSetMag : BUFFER INTEGER RANGE 0 TO 10000 );
END subtractor;

ARCHITECTURE Behavior OF subtractor IS

BEGIN
    PROCESS (Clock)
    BEGIN

    IF rising_edge(Clock) THEN
```

```
                offSetMag<=(10000 - KMag);

        END IF;
        END PROCESS;
END Behavior;
```

Accumulator Block:

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY ACCUMULATOR IS
        PORT ( Clock : IN STD_LOGIC;
                 Pulse : IN INTEGER RANGE 0 TO 2;
                 offMSB : STD_LOGIC;
                 offMAG : IN INTEGER RANGE 0 TO 10000;
                 accLevelMSB : BUFFER STD_LOGIC;
                 accLevelMAG: BUFFER INTEGER RANGE 0 TO 20000;
                 delMSB : OUT STD_LOGIC;
                 delMAG : BUFFER INTEGER RANGE 0 to 2 );
END ACCUMULATOR;

ARCHITECTURE Behavior OF ACCUMULATOR IS
BEGIN
        PROCESS (Clock)
        BEGIN

                IF rising_edge(Clock) THEN

                        IF (Pulse=1) THEN
                                accLevelMSB<='0';
                                accLevelMAG<=0;

                        ELSIF (Pulse=2) THEN

                                delMSB<='0';
                                delMAG<=0;

                                IF (accLevelMSB=offMSB) THEN
                                        accLevelMAG<= accLevelMAG + offMAG;
                                ELSIF (accLevelMAG>=offMAG) THEN
                                        accLevelMAG<= accLevelMAG - offMAG;
                                ELSIF (offMAG>accLevelMAG) THEN
                                        accLevelMAG<= offMAG - accLevelMAG;
                                        accLevelMSB<=offMSB;
                                END IF;
```

```
                    IF (accLevelMAG>10000) THEN
                        accLevelMAG<= (20000 - accLevelMAG);
                        IF (accLevelMSB='1') THEN accLevelMSB<='0';
                        ELSE accLevelMSB<='1';
                        END IF;
                    END IF;

                    IF (offMAG=0 AND accLevelMAG/=0) THEN
                        delMAG<=2;
                        IF (accLevelMAG<2) THEN accLevelMAG<=0;
                        ELSE accLevelMAG<=accLevelMAG - 2;
                        END IF;

                        IF (accLevelMSB='0') THEN delMSB<='1';
                        END IF;
                    END IF;

                END IF;
            END IF;
        END PROCESS;
END Behavior;
```

Adder Block:

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder IS
    PORT ( Clock : IN STD_LOGIC;
            dsdMSB : IN STD_LOGIC;
            muteMSB : IN STD_LOGIC;
            dsdMag  : IN INTEGER RANGE 0 TO 10000;
            muteMag : IN INTEGER RANGE 0 TO 10000;
            totalMag : BUFFER INTEGER RANGE 0 TO 10000;
            outMSB : OUT STD_LOGIC );
END adder;

ARCHITECTURE Behavior OF adder IS

BEGIN

    PROCESS (Clock)
    BEGIN
```

```
        IF rising_edge(Clock) THEN
                IF ( dsdMSB=muteMSB ) THEN
                        totalMag<=(dsdMag + muteMag);
                        outMSB<=dsdMSB;

                ELSIF (dsdMag >= muteMag) THEN
                        totalMag<= (dsdMag - muteMag);
                        outMSB<=dsdMSB;

                ELSIF (muteMag >= dsdMag) THEN
                        totalMag<= (muteMag - dsdMag);
                        outMSB<=muteMSB;
                END IF;

        END IF;
        END PROCESS;

END Behavior;
```

Adder2 Block:

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder2 IS
        PORT ( Clock : IN STD_LOGIC;
                dsdMSB : IN STD_LOGIC;
                muteMSB : IN STD_LOGIC;
                dsdMag  : IN INTEGER RANGE 0 TO 10000;
                muteMag : IN INTEGER RANGE 0 TO 2;
                totalMag : BUFFER INTEGER RANGE 0 TO 10000;
                outMSB : OUT STD_LOGIC );
END adder2;

ARCHITECTURE Behavior OF adder2 IS

BEGIN

        PROCESS (Clock)
        BEGIN

        IF rising_edge(Clock) THEN
                IF ( dsdMSB=muteMSB ) THEN
                        totalMag<=(dsdMag + muteMag);
                        outMSB<=dsdMSB;
```

```
                        ELSIF (dsdMag >= muteMag) THEN
                                totalMag<= (dsdMag - muteMag);
                                outMSB<=dsdMSB;

                        ELSIF (muteMag >= dsdMag) THEN
                                totalMag<= (muteMag - dsdMag);
                                outMSB<=muteMSB;
                        END IF;

        END IF;
        END PROCESS;

END Behavior;
```

Shifter Block (Delay Line):

```
library ieee;
use ieee.std_logic_1164.all;
entity shift is
  port(C, SI : in std_logic;
       SO : out std_logic);
end shift;
architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
      begin
        if (C'event and C='1') then
          for i in 0 to 6 loop
            tmp(i+1) <= tmp(i);
          end loop;
          tmp(0) <= SI;
        end if;
    end process;
    SO <= tmp(7);
end archi;
```

Pattern Match Detector Block:

```
LIBRARY      ieee ;
USE ieee.std_logic_1164.all ;

ENTITY patern IS
```

```
    PORT ( Clock, DSD, SDM : IN STD_LOGIC;
           flag : OUT STD_LOGIC;
           count : BUFFER INTEGER RANGE 0 TO 10 );
END patern;

ARCHITECTURE Behavior OF patern IS

BEGIN

    PROCESS (Clock)
    BEGIN

    IF rising_edge(Clock) THEN

        IF (DSD=SDM) THEN
            IF (count=10) THEN flag<='1';
            ELSE count<=count+1;
            END IF;
        ELSE
            count<=0;
            flag<='0';
        END IF;

    END IF;
    END PROCESS;

END Behavior;
```

Switching Control Logic Unit Block:

```
LIBRARY     ieee ;
USE ieee.std_logic_1164.all ;

ENTITY clu IS
    PORT ( Clock, masterPause, flag, originalDSD, mixedDSD : IN STD_LOGIC;
           pause, finalDSD : OUT STD_LOGIC );
END clu;

ARCHITECTURE Behavior OF clu IS

BEGIN

    PROCESS (Clock)
```

```
	BEGIN

	IF rising_edge(Clock) THEN

		IF (masterPause='0') THEN
			pause<='0';
			IF (flag='1') THEN finalDSD<=originalDSD;
			END IF;
		ELSE
			IF (flag='1') THEN
				finalDSD<=mixedDSD;
				pause<='1';
			END IF;
		END IF;

	END IF;
	END PROCESS;

END Behavior;
```

Altera Simulation Results:

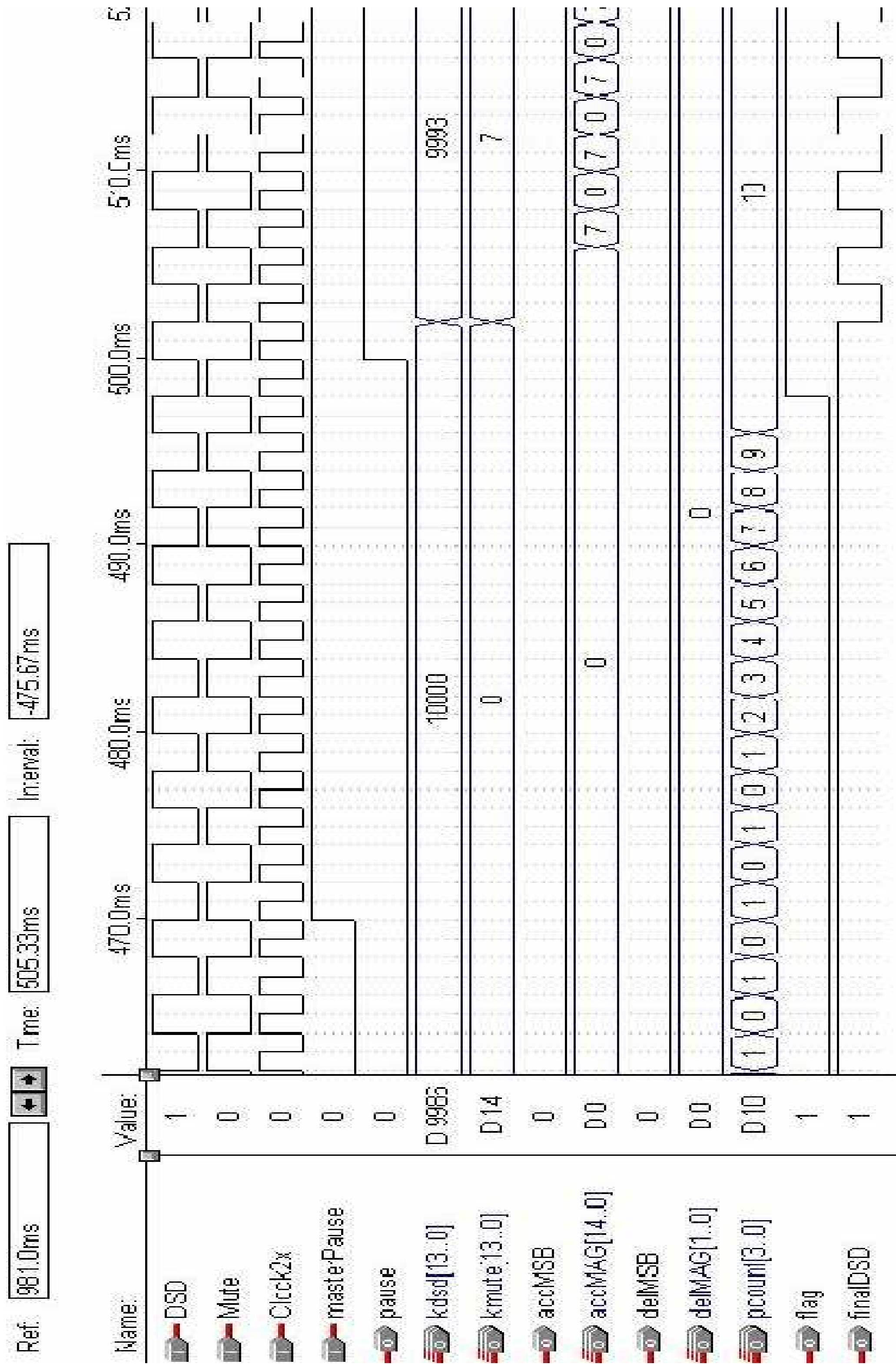


Fig.2. Master-pause and pause activation

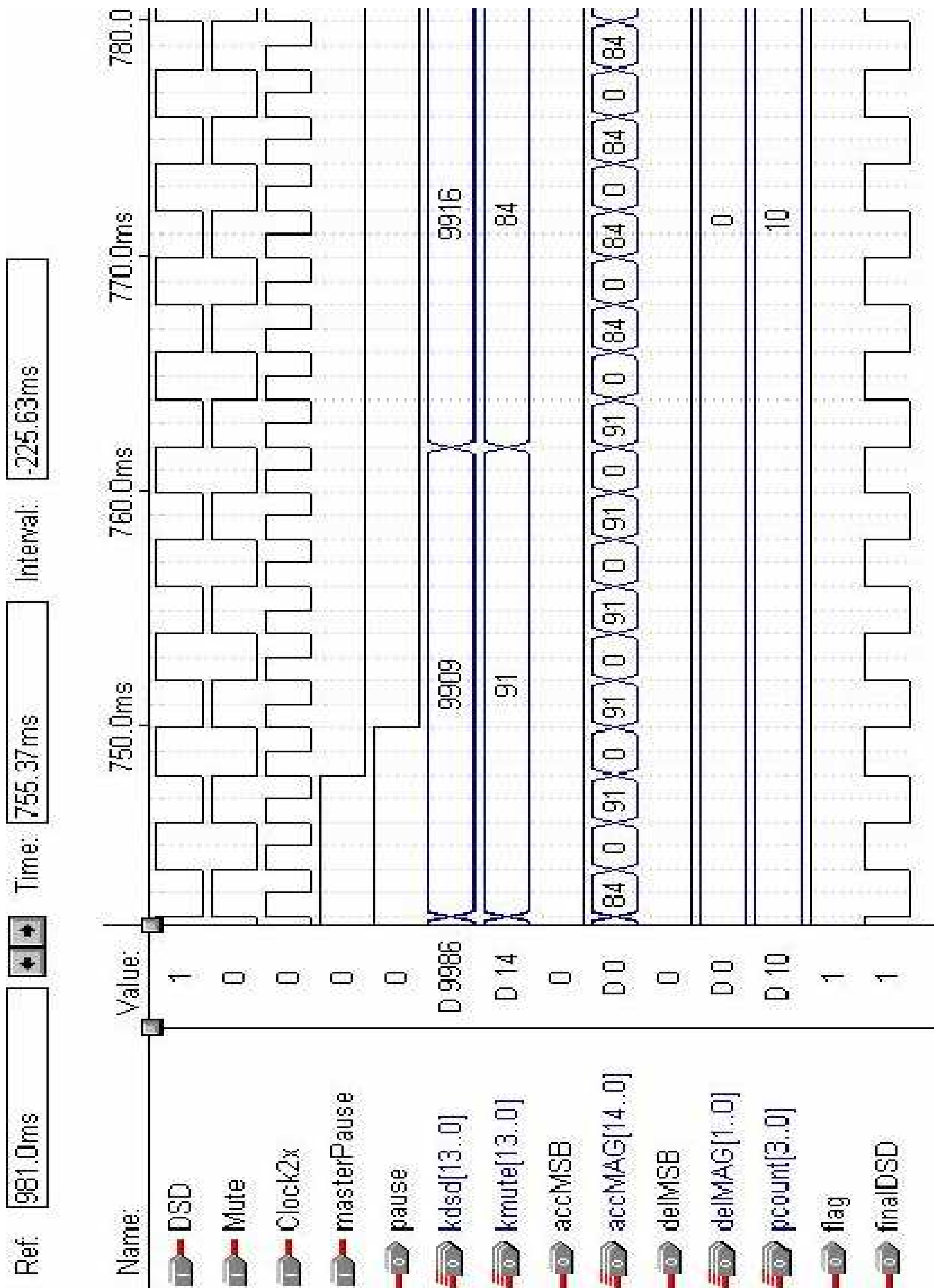


Fig.3. Pause deactivation

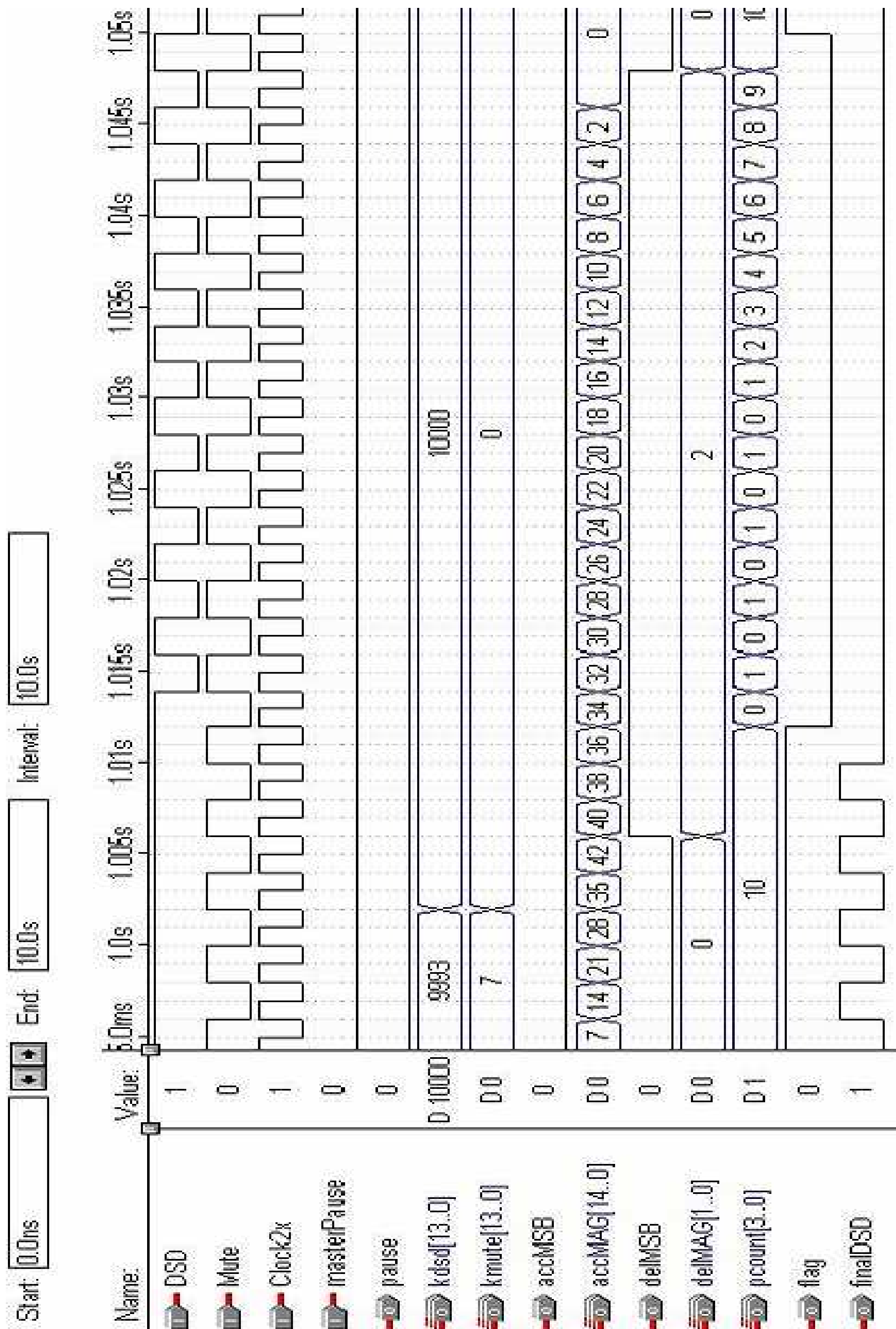


Fig.4. Accumulator correction

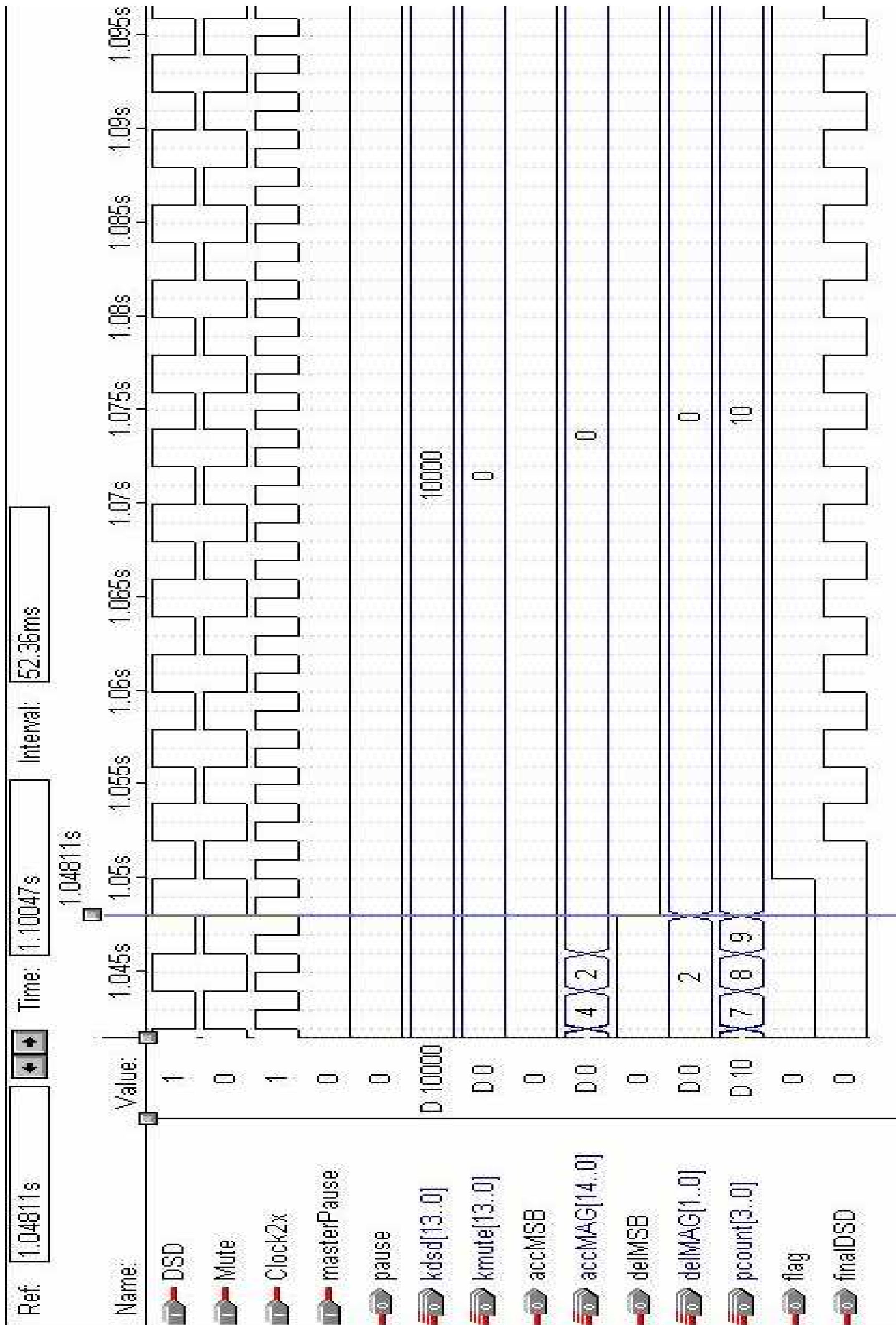


Fig.5. Steady match