DiSPressor – a real-time audio compressor implemented in a DSP

Jörgen Anderton - jorand-3@student.ltu.se
Henrik Wikner - henwik-1@student.ltu.se
Abstract

The compressor is one of the most widely used effects in audio signal processing and the compression algorithm is in this project implemented in a DSP card, providing real-time stereo functionality. The nature of real-time applications requires rewriting the compression algorithm to be more suited for the Assembly language in the DSP programming environment. The results show that while making a working compressor is viable, making a usable compressor with a pleasing sound quality requires a lot of additional work.
Index

Introduction ............................................................................................................................... 1
  Project introduction ............................................................................................................... 1
  Compression .......................................................................................................................... 1
  Hardware specifications ....................................................................................................... 2
  Programming environment .................................................................................................. 2
Method ........................................................................................................................................ 3
  Concepts .............................................................................................................................. 3
Implementation ....................................................................................................................... 3
  MATLAB implementation ................................................................................................. 3
  RMS measurement .............................................................................................................. 4
  Lin/log conversion .............................................................................................................. 4
  Threshold ........................................................................................................................... 4
  Static curve algorithm ....................................................................................................... 4
  Log/lin conversion ............................................................................................................. 4
Results ..................................................................................................................................... 5
Discussion ............................................................................................................................ 6
  Implementation problems ................................................................................................. 6
  How does it sound? Why? ................................................................................................. 7
  Further development ....................................................................................................... 7
References ............................................................................................................................. 8
Appendices ........................................................................................................................... 8
  Appendix 1 – main.asm .................................................................................................... 8
  Appendix 2 – processing.asm ......................................................................................... 13
  Appendix 3 – compressor.m ......................................................................................... 22
Introduction

Project introduction

The final project in the course Audio Signal Processing given at Luleå University of Technology has been to implement an algorithm for audio processing in an off-line environment like MATLAB or in a real-time environment. This project consists of implementing a compression algorithm in a DSP card with real-time control of the parameters. The report is a PDF document that includes clickable audio examples in the Results section.

Compression

The audio effect referred to as compression is one of the most widely used tools in a recording studio today. A compressor can be used to create a specific noticeable sound, make a signal more useful without a certain sonic signature and anything there in between.

The basic idea of compression is to reduce the dynamic range of the input signal in a non-linear fashion by setting a threshold level, \( CT \), and only applying compression above this level. The amount of compression applied is basically a ratio, \( CR \), by which the input signal is decreased. \( CR \) is defined as the fraction of input level change to output level change. Because human hearing is described as roughly logarithmic (Wolfe 1998), describing the input and output levels as logarithmic values of the sound pressure will render the ratio as a linear function on logarithmic axes. These two parameters are described in the static curve graph (Figure 1).

\[
CR = \frac{\Delta L_{\text{in}}}{\Delta L_{\text{out}}} = \frac{3}{1}
\]

Further parameters that have an impact on the sound of the compressor are the attack and release times. The former determines how soon the maximum compression ratio is reached after the input signal exceeds the threshold. The latter determines how soon the ratio should reach 1:1 (no

Figure 1: The static curve, \( CT = -30 \text{dB} \), \( CR = 3:1 \).
compression) when the input signal goes below the threshold again. In the theoretical static curve, the attack and release times are both zero.

**Hardware specifications**

The compressor algorithm is in this project to be implemented in a real-time stand-alone device. This raises the issue of specifications of the specific hardware and its limitations and qualities. The device used is the STUD-1 DSP card from Rubico Systems. It utilizes the 160 MHz ADSP-2191 processor from Analog Devices together with 160kB onboard memory. The A/D/D/A conversion is also provided by Analog Devices, namely the AD1885 16 bit converter, integrated in the card. Operation with a sample rate of 44.1 kHz results in \(\frac{160000000}{44100} = 3628\) cycles per sample. This divided by two results in \(1814\) calculations per sample and channel which is our theoretical limit.

The processor is effectively 16 bit fixed point with an intermediate 32-40 bit representation in single calculations. It can represent numbers in two fashions, the default being 1.15 fractional. The other is integer mode and these two can not be used together in a calculation. (See Figure 2a and b.)

The ADSP-2191 can basically only perform the arithmetic operations multiplication and addition/subtraction in units called MAC (Multiply and Accumulate unit) and ALU (Arithmetic Logic Unit), respectively. The possibility of shifting a word \(k\) bits up or down results effectively in division or multiplication by 2. Reading from memory is also required for every operation, as specific registers must be used in conjunction with the MAC and ALU. These operations take one cycle to execute with the option of combining two memory reads and one MAC operation in one cycle. Other operations can be constructed by combining the basic operations, e.g. the function DIVS executes a division in 16 clock cycles.

**Programming environment**

The programming language used to program the ADSP-2191 is Assembly Language. This low level language is in essence directly connected to the basic features of the processor, resulting in very good control of what the program/DSP is doing exactly. The limitation of this low level programming is the lack of ready-made mathematical functions, use of powers or logarithms require workarounds, e.g. look-up tables. Visual DSP++ is a programming environment from Analog Devices especially constructed for DSP programming that was used together with the serial
device programmer PonyProg to write, assemble, link, load and transfer the program to the
STUD-1.

Method

Concepts

Because of the 16.0 fixed point format any compression ratio can not be chosen. Although, if the
ratios are chosen so that they are powers of 2, that is 1:1, 2:1, 4:1, 8:1, 16:1, 32:1 the operand can
be shifted by one bit if the ratio is set to 2:1 and 5 bits if it's set to 32:1. The formula used in this
implementation is taken from (Zölzer 2002) and looks like this (all capitals are dB-values):

\[ F = -CS(X - CT) \]  

where

\[ CS = 1 - \frac{1}{CR} \]  

and CR is the chosen compression ratio. The equation can be rewritten as:

\[ F = \frac{1}{CR} (X - CT) - (X - CT) \]  

which would imply that one can shift the first part with the corresponding amount of bits and just
subtract the second part from that.

Furthermore, to make the implementation work in the Stud-1 and to make conversions to the
logarithmic domain one must use look up tables. This because of the DSP card's incapability of
calculating the logarithmic function. The logarithmic function could be estimated by

\[ 889 \cdot x^{1/2048} \approx \log_{10}(x) \]  

As this equation involves both integers and fractions and besides that there's an exponent
involved. The difficulty in implementing this in the DSP lead to the formula being rewritten as (3).

To translate the linear values from the input to a logarithmic value look up tables are used. One
vector with dB-values from 0 to -99 dB is constructed and also one vector with the corresponding
linear values. This implies a quite poor resolution but on the other hand, a fraction of one decibel
isn't that significant.

Implementation

The code for the compressor was written in an already existing program that only sent the audio
straight through, aka talk through. This simplified the implementation quite much as the only thing
that needed to be implemented was the compressor. A block diagram of the compressor is
included in Figure 3.

MATLAB implementation

A MATLAB implementation (see Appendix 3 for source code) was done to simulate the roundoff
effects that may occur when using fixed point arithmetic. Although, the implementation did not
include look up tables when converting from logarithmic to linear domain and vice versa. This
because it was considered that this should not affect the sound quality, only give less control over
which threshold that is actually used and then also affect which sound level in the input signal that is actually going over the threshold.

**RMS measurement**

The mean of the absolute values of the left and right channels is taken and translated to a dB-value. The envelope of the signal (the RMS-value) is computed by a first order low pass filter (Zölzer 2002, see Equations 5-7). The variable \( c \) is dependent only on \( f_c \) and \( f_s \), and is taken from an existing vector, its position depending on the chosen \( f_c \) in the compressor interface.

\[
x_{\text{peak}}(n) = x(n) \cdot x(n)
\]

\[
x_{\text{ap}}(n) = c \cdot x_{\text{peak}}(n) + x_{\text{peak}}(n-1) - c \cdot x_{\text{ap}}(n-1)
\]

\[
x_{\text{rms}}(n) = 0.5 \cdot [x_{\text{ap}}(n) + x_{\text{peak}}(n)]
\]

**Lin/log conversion**

The envelope sample of the input is subtracted from one value at a time in the linear vector and for each turn in the loop the pointer to the logarithmic vector is moved one step ahead. When the sign of the difference changes the pointer in the logarithmic vector points to the right position and the corresponding dB-value to the input is now known.

**Threshold**

This dB-value is compared to a threshold and the result of this comparison makes the decision whether the input signal should be compressed or not. If it's below the threshold the final gain factor is set to 1 - 2\(^{-15}\) and then multiplied with the input signal, which in practice means no compression. If the input signal is above the threshold, then the signal should be compressed and the signal's dB-value is fed to the compression algorithm.

**Static curve algorithm**

Using Equation 3, the logarithmic value of the gain factor is calculated.

**Log/lin conversion**

Now the gain factor is calculated and it's logarithmic value is known. Since the input signal is between -1 and 2\(^{-15}\) the gain factor has to be in that region too so the logarithmic value is translated back to linear domain in the same way as before. Subtracting the logarithmic value from each value in the logarithmic vector at a time and checking when the sign changes yields the linear gain factor. This value is multiplied with the input signals both channels.

Figure 3: Block diagram of the compressor.
Results

The most evident results of compression is listening to the sound files but there is also a graphical representation. In Figure 4, an uncompressed and compressed drum loop is shown in the time domain.

![Graphs showing uncompressed and compressed audio](image)

*Figure 4a and b: Uncompressed and compressed audio, CT = -35 dB. Observe the different amplitude scales.*

The use of different cut off frequencies in the RMS measurement stage also renders graphs that illustrate the difference in sound. When $f_c = 10\text{Hz}$, the sluggish response of the compressor never lets the transient of the signal through, resulting in a controlled sound with very little attack. As the cut off frequency increases, a larger part of the transient is output giving more attack to the sound. However, the release time also decreases, resulting in very quick volume changes. This phenomenon is known as *pumping* and is often used with drum beats in electronic music to create a rhythmic effect that works well together with the bass drum beat and bass guitar/synthesizer. At $f_c = 800\text{Hz}$, this effect is very evident, resulting in a crushing sound. (See Figure 5.)
Discussion

Implementation problems

When programming a DSP-card problems occur. Problems encountered in this project are described in this section. Using the Assembly language is always causing problems. There are rules for which instructions that may be used as the last instruction in a loop. The rules are not that clear though so the troubleshooting took very long time in this project. More concrete, one may not do an addition as the last instruction and writing it to the result register. This causes an error worse than one can imagine. And therefore it takes about four days to resolve the problem. Of course this problem created more problems. The goals of implementing a limiter, expander and noise gate and also implementing an attack/release-function could not be achieved and the sound quality of the compressed signal leaves more to wish and the sound could maybe be better with an attack and release function.

A few tricks used while programming can be mentioned. Because of the DSP-card's limitations when it comes to division the ratios were chosen to even powers of two which resulted in that division wasn't necessary anymore.

Of course, the rewriting of the equation used for the static curve gain factor is crucial for this project. If that isn't made the whole implementation would be much harder.
Also, when low pass filtering the input signal to get the RMS-value, the standard equation for that is that first the signal is all-pass filtered and then adding 1 to the all-pass filtered signal and then multiplying with one half. If instead the constant of one half is used all the way from the beginning, overflow is avoided.

Another very useful trick is to use the instruction set reference and hardware reference while programming as this is shown to be very helpful.

**How does it sound? Why?**

The compressor sounds as it should, compressing the signal more with a higher ratio and/or a lower threshold. Unfortunately, it adds an unnatural, ringing sound to the compressed output that we haven’t been able to explain. The lack of attack and release time could be a problem but the MATLAB implementation of the algorithm does not add this ringing sound. Another possibility is that the integer resolution of the dB-values is not sufficient for our needs, as the resolution is reduced to 100/65535 (100 steps of one dB while in integer mode), compared to full 16 bit resolution (65535 steps) in fractional mode. However, a MATLAB experiment where the compression parameters where quantized to simulate the DSP did not result in the ringing sound.

**Further development**

The obvious development of the project is to find an explanation to why the ringing sound is introduced in the real-time implementation. Including control over attack and release time would also be required for the compressor to be really useful.

**References**


Appendices

Appendix 1 – main.asm

Source code for Assembly file main.asm:

```assembly
main.asm

#include "def2191_stud-1.h"
#include "lcd_macro.h"
#define                STATE_MAX               0x0002
#define                STATE_MIN               0x0000
#define              VALUE_MAX               0x0064
#define              VALUE_MIN               0x0000
#define                TALKTHROUGH             0x0000
#define                COMPRESSOR              0x0001

GLOBAL  EXTERNAL DECLARATIONS

// Global symbols
.global        Start;
.global       PUSH1;
.global       CW1;
.global       CCW1;
.global       CW2;
.global       CCW2;

// External symbols
.extern        Change_PLL_Multiplier;
.extern        Codec_Reset;
.extern        Program_SPORT0_Registers;
.extern        Program_DMA_Controller;
.extern        A01885_Codec_Initialization;
.extern        RX_Status;
.extern        Initialize_LCD;
.extern        Poll_Rotary_Encoder1;
.extern        Poll_Rotary_Encoder2;
.extern        Check_Rotary_Encoder1_Push;
.extern        Wait_10ms;

// New symbols for the project
.global        log;
.global        lin;
.global        threshold;
.global        k;
.global        c_coeff;
.global        rotary_1;
.global        rotary_2;
.global        push_1;

.global        output2_text;
.global        talk_text;
.global        threshold_text;
.global        ratio_text;
.global        thresholdvalue_text;
.global        ratiovalue_text;
.global        cutoff_text;
.global        cutoffvalue_text;

DM DATA

.SECTION/dm data0:
```
.var  push_1       = 0x0000;
.var  rotary_1     = 0x0000;
.var  rotary_2     = 0x0001;
.var  writelcd    = 0x1000;  //Make sure the program writes to the LCD the first interrupt

//PRI_DAG circular buffers
.var  log[100]     = "log_hex.dat";  //PRI I0
.var  lin[100]     = "lin.dat";     //PRI I1
.var  threshold[50] = "threshold_hex.dat";  //PRI I2
.var  k[7]         = "k_hex.dat";  //PRI I4
.var  c_coeff[5]   = "c_coeff_new.dat";  //PRI I5

//SEC_DAG circular buffers
.var  thresholdvalue_text[100] = '0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49'; //SEC I0
.var  ratiovalue_ ex[4] = '1 2 4 8 16 32 oo'; //SEC I1
.var  cutoff_text[6]  = 'LP Cutoff 10 Hz';  //SEC I2
.var  ratio_text[16]  = 'Ratio 1:1';  //SEC I7
.var  cutoffvalue_text[15] = '10 20 30 50 80 150 300 600'; //SEC I5
.var  output2_text[16] = 'Output2 '; //SEC I4
.var  compressor_text[16] = ' DiSPressor * ';  //SEC I3

*******************************************************************************
/*PM DATA*/
*******************************************************************************
.SECTION/.pm program0

// ----- Program execution starts here...! -------
Start:
call Change_PLL_Multiplier; // Change the multiplier of the processor.

External clock is only 16 MHz.
call Codec_Reset; // Reset the codec
   call Program_SPORT0_Registers; // Initialize SPORT0 for codec communications
call Program_DMA_Controllers; // Start Serial Port 0 tx and rx DMA Transfers
call AD1885_Codec_Initialization; // Initialize & program AD1885
   // Set direction of flagpins
IOPG = General Purpose IO;
nop;
ar = 0x0FC8B;
   IO(DIRS) = AR;
call Initialize_LCD; // Initialize the LCD
call Wait_10ms; // Wait 10 milliseconds...
   // Turn off all LEDs on STUD-1 board
AR = 0x0000B;
   IO(FLAGC) = AR;
   // Clear RX_Status flag indicating incoming RX data is audio data and can be processed
ax0 = 0x0000;
dm(RX_Status) = ax0;
   //New initializations for this project
   call init_circ_buffer;
   Output_LCD_Token(compressor_text,16,1,1);

// ----- Go into infinite program loop after initialization... ------
wait_forever:
   idle;
   CALL Check_write;
   // Check if Rotary Encoder 1 was pushed...
   CALL Check_Rotary_Encoder1_Push;
   // Check if Rotary Encoder 1 was turned
   CALL Poll_Rotary_Encoder1;
   // Check if Rotary Encoder 2 was turned
   CALL Poll_Rotary_Encoder2;
   jump wait_forever;

PUSH1:
\[\text{CALL Set_write; }\]
\[\text{AY0 = DM(push_1); }\]
\[\text{AX0 = COMPRESSOR; } \quad //0x0001\]
\[\text{AR = AX0 XOR AY0; }\]
\[\text{AX1 = TALKTHROUGH; }\]
\[\text{DM(push_1) = AX1; }\]
\[\text{IF EQ RTS; }\]
\[\text{AX0 = TALKTHROUGH; } \quad //0x0000\]
\[\text{AR = AX0 XOR AY0; }\]
\[\text{AX1 = COMPRESSOR; }\]
\[\text{DM(push_1) = AX1; }\]
\[\text{RTS; }\]

CW1:
\[\text{//If the DSP is in talk through, disregard any movement of Rotary Encoder 1 }\]
\[\text{AX0 = DM(push_1); }\]
\[\text{AY0 = TALKTHROUGH; }\]
\[\text{AR = AX0 XOR AY0; }\]
\[\text{IF EQ RTS; }\]
\[\text{CALL Set_write; }\]
\[\text{//If the Rotary Encoder 1 is in its maximum position, disregard any movement }\]
\[\text{AX0 = DM(rotary_1); }\]
\[\text{AR = AX0 XOR AY0; }\]
\[\text{IF EQ RTS; }\]
\[\text{//Increase the value of rotary_1 by 1 }\]
\[\text{AX0 = DM(rotary_1); }\]
\[\text{AY0 = 0x0001; }\]
\[\text{AR = AX0 + AYO; }\]
\[\text{DM(rotary_1) = AR; }\]
\[\text{RTS; }\]

CCW1:
\[\text{//If the DSP is in talk through, disregard any movement of Rotary Encoder 1 }\]
\[\text{AX0 = DM(push_1); }\]
\[\text{AY0 = TALKTHROUGH; }\]
\[\text{AR = AX0 XOR AY0; }\]
\[\text{IF EQ RTS; }\]
\[\text{CALL Set_write; }\]
\[\text{//If the Rotary Encoder 1 is in its minimum position, disregard any movement }\]
\[\text{AX0 = DM(rotary_1); }\]
\[\text{AR = AX0 XOR AY0; }\]
\[\text{IF EQ RTS; }\]
\[\text{//Decrease the value of rotary_1 by 1 }\]
\[\text{AX0 = DM(rotary_1); }\]
\[\text{AY0 = 0x0001; }\]
\[\text{AR = AX0 - AYO; }\]
\[\text{DM(rotary_1) = AR; }\]
\[\text{RTS; }\]

CW2:
\[\text{CALL Set_write; }\]
\[\text{AX0 = DM(rotary_2); }\]
\[\text{AY0 = 0x0001; }\]
\[\text{AR = AX0 + AYO; }\]
\[\text{DM(rotary_2) = AR; }\]
\[\text{RTS; }\]
CCW2:  
CALL Set_write;

AX0 = DM(rotary_2);
AY0 = 0x0001;
AR = AX0 - AYO;

DM(rotary_2) = AR;
RTS;

Output_LCD:  
// Write the appropriate text to the second row of the LCD
Output_LCD_Token(output2_text,16,2,1);
RTS;

Check_write:  
AX0 = DM(writelcd);
AY0 = 0x1000;
AR = AX0 XOR AYO;

IF EQ CALL Output_LCD;
AX0 = 0x0000;
DM(writelcd) = AX0;
RTS;

Set_write:  
AX0 = 0x1000;
DM(writelcd) = AX0;
RTS;

init_circ_buffer:  
//  log buffer -> I0
DMPG1 = Page(log);
I0 = log;
L0 = length(log);
M0 = 1;
AX0 = log;
Reg(B0) = AX0;

//  lin buffer -> I1
DMPG1 = Page(lin);
I1 = lin;
L1 = length(lin);
M1 = 1;
AX0 = lin;
Reg(B1) = AX0;

//  threshold buffer -> I2
DMPG1 = Page(threshold);
I2 = threshold;
L2 = length(threshold);
M2 = 1;
AX0 = threshold;
Reg(B2) = AX0;

//  k buffer -> I4
DMPG2 = Page(k);
I4 = k;
L4 = length(k);
M4 = 1;
AX0 = k;
Reg(B4) = AX0;

//  c_coeff buffer -> I5
DMPG2 = Page(c_coeff);
I5 = c_coeff;
L5 = length(c_coeff);
M5 = 1;
AX0 = c_coeff;
Reg(B5) = AX0;
///
SEC_DAG: text buffers

ENA SEC_DAG;

DMPG1 = Page(thresholdvalue_text);
I0 = thresholdvalue_text;
L0 = length(thresholdvalue_text);
M0 = 1;
AX0 = thresholdvalue_text;
Reg(B0) = AX0;

DMPG1 = Page(ratiovalue_text);
I1 = ratiovalue_text;
L1 = length(ratiovalue_text);
M1 = 1;
AX0 = ratiovalue_text;
Reg(B1) = AX0;

DMPG1 = Page(cutoff_text);
I2 = cutoff_text;
L2 = length(cutoff_text);
M2 = 1;
AX0 = cutoff_text;
Reg(B2) = AX0;

DMPG1 = Page(talk_text);
I3 = talk_text;
L3 = length(talk_text);
M3 = 1;
AX0 = talk_text;
Reg(B3) = AX0;

DMPG2 = Page(output2_text);
I4 = output2_text;
L4 = length(output2_text);
M4 = 1;
AX0 = output2_text;
Reg(B4) = AX0;

DMPG2 = Page(cutoffvalue_text);
I5 = cutoffvalue_text;
L5 = length(cutoffvalue_text);
M5 = 1;
AX0 = cutoffvalue_text;
Reg(B5) = AX0;

DMPG2 = Page(threshold_text);
I6 = threshold_text;
L6 = length(threshold_text);
M6 = 1;
AX0 = threshold_text;
Reg(B6) = AX0;

DMPG2 = Page(ratio_text);
I7 = ratio_text;
L7 = length(ratio_text);
M7 = 1;
AX0 = ratio_text;
Reg(B7) = AX0;

DIS SEC_DAG;
RTS;
Appendix 2 – processing.asm

Source code for Assembly file processing.asm

/********************************************************************************
Processing.asm
********************************************************************************/
#include "def2191_stud-1.h"

CONSTANT & MACRO DEFINITIONS

/* AD1885 TDM Timeslot Definitions */
#define                TAG_PHASE               0
#define                COMMAND_ADDRESS_SLOT    1
#define                COMMAND_DATA_SLOT       2
#define                STATUS_ADDRESS_SLOT     1
#define                STATUS_DATA_SLOT        2
#define                LEFT                    3
#define                RIGHT                   4

/* Left and Right ADC valid Bits used for testing of valid audio data in current TDM frame */
#define                M_Left_ADC              12
#define                M_Right_ADC             11
#define                THRESHOLD               0x0000
#define                RATIO                   0x0001
#define                CUTOFF                  0x0002
#define                TALKTHROUGH             0x0000
#define                COMPRESSOR              0x0001
#define                ROTARY_DECR             0x0000
#define                ROTARY_NEUTRAL  0x0001
#define                ROTARY_INCR             0x0002
#define                ONE                             0x0001
#define                ZERO                    0x0000

GLOBAL & EXTERNAL DECLARATIONS

.global                Process_AD1885_Audio_Samples;
.global                IOPG_TMP;
.extern                tx_buf;
.extern                rx_buf;
.extern                log;
.extern                lin;
.extern                threshold;
.extern                k;
.extern                c_coeff;
.extern                rotary_1;
.extern                rotary_2;
.extern                push_1;
.extern                output2_text;
.extern                talk_text;
.extern                threshold_text;
.extern                ratio_text;
.extern                thresholdvalue_text;
.extern                ratiovalue_text;
.extern                cutoff_text;            //SEC I2
.extern                cutoffvalue_text;       //SEC I5

/*****************************/
/* DM data */
/*****************************/

SMS044 – DiSPressor – a real-time audio compressor implemented in a DSP 13
// Input samples
.var Left_Channel;
.var Right_Channel;

// Output samples
.var Left_Channel_Out;
.var Right_Channel_Out;
.var IOPG_TMP;

// new symbols for the project
.var x_peak_n = 0; // position 0 = present sample, position 1 = previous sample
.var x_peak_n_1 = 0; // position 0 = present sample, position 1 = previous sample
.var x_ap_half_n = 0; // position 0 = present sample, position 1 = previous sample
.var x_ap_half_n_1 = 0; // position 0 = present sample, position 1 = previous sample
.var x_rms = 0; // position 0 = present sample, position 1 = previous sample
.var lp_const = 0x4000; // constant for lp filtering, = 0.5
.var X_CT;
.var CT_X;
.var iopg_temp_j;
.var temp;
.var temp2;
.var singlespace = ' ';
.var f_gain = 0;
.var g_gain_n = 0;
.var g_gain_n_1 = 0;
.var attack_max;
.var release_max;
.var AT;
.var RT;
.var a_cntr = 0;
.var r_cntr = 0;

//**************************************************************************/
/* PM DATA
/**************************************************************************/

//**************************************************************************/
Process_AD1885_Audio_Samples:
ax0 = 0x8000; /* Clear all AC97 link Audio Output Frame slots */
dm((tx_buf + TAG_PHASE) = ax0; /* and set Valid Frame bit in SLOT '0' TAG phase */
ax0 = 0;
dm((tx_buf + COMMAND_ADDRESS_SLOT) = ax0;
dm((tx_buf + COMMAND_DATA_SLOT) = ax0;
dm((tx_buf + LEFT) = ax0;
dm((tx_buf + RIGHT) = ax0;

Check_ADCs_For_Valid_Data:
ax0 = dm((tx_buf + TAG_PHASE); /* Get ADC valid bits from tag phase slot*/
ax1 = 0x1800; /* Mask other bits in tag */
ar = ax0 and ax1;

Set_TX_Slot_Valid_Bits:
ay1 = dm((tx_buf + TAG_PHASE); /* Frame/Addr/Data valid bits */
ar = ar or ay1; /* Set TX valid bits based on Recieve TAG info */
dm((tx_buf + TAG_PHASE) = ar;

Check_AD1885_ADC_Left:
AR = TSTBIT M_Left_ADC of ax0; /* Check Left ADC valid bit */
IF EQ JUMP Check_AD1885_ADC_Right; /* If valid data then save ADC sample */
ax1 = dm((tx_buf + LEFT)); /* Get AD1885 Left channel input sample */
dm(Left_Channel) = ax1; /* Save to data holder for processing */

Check_AD1885_ADC_Right:
AR = TSTBIT M_Right_ADC of ax0; /* Check Right ADC valid bit */
IF EQ JUMP Valid_Frame; /* If valid data then save ADC sample */
ax1 = dm((tx_buf + RIGHT)); /* Get AD1885 Right channel input sample */
dm(Right_Channel) = ax1; /* Save to data holder for processing */
/*******************************************************************************
* *** Insert DSP Algorithms Here ***
* 
* Input L/R Data Streams  = DM(Left_Channel) DM(Right_Channel)
* Output L/R Results  = DM(Left_Channel_Out) DM(Right_Channel_Out)
*
*******************************************************************************/

// Output to tx_buf is set first to give the DMA transfers enough time.
Playback_Audio_Data:

AR = DM(Right_Channel_Out);
DM(App+RIGHT) = AR;       // ...output Right data

AR = DM(Left_Channel_Out);
DM(tx_buf+LEFT) = AR;    // ...output Left data

// ***** Process input samples here... *****

//Talkthrough or compression, bypass
AX0 = DM(push_1);
AY0 = TALKTHROUGH;
AR = AX0 XOR AY0;
IF EQ jump talkthrough;
AY0 = COMPRESSOR;
AR = AX0 XOR AY0;
IF EQ jump choose_parameter;

choose_parameter:
    //Threshold or Ratio
AX0 = DM(rotary_1);
AY0 = THRESHOLD;
AR = AX0 XOR AY0;
IF EQ jump change_threshold;
AY0 = RATIO;
AR = AX0 XOR AY0;
IF EQ jump change_ratio;
AY0 = CUTOFF;
AR = AX0 XOR AY0;
IF EQ jump change_cutoff;

// ***** End of sample processing... *****

/* ...house keeping prior to RTI */
Valid_Frame:

ay1=3;                     // Clear RX Interrupts
io(SPDR_IRQ)=ay1;
AY1 = DM(IOPG_TMP);
IOPG = ay1;
DIS SEC_REG;               // Disable Secondary Registers
RTI;                      //End of program

talkthrough:
    //Update LCD with talkthrough_text
ENA SEC_DAG;

CNTR = 16;
nop;
nop;
DO talk_loop2 UNTIL CE;
    AX0 = DM(I3+M4);
    talk_loop2: DM(I4+M4) = AX0;
DIS SEC_DAG;
// ****** Talk Through ******
AR = DM(Right_Channel);
DM(Right_Channel_Out) = AR;

AR = DM(Left_Channel);
DM(Left_Channel_Out) = AR;
jump Valid_Frame;

change_threshold:
AX0 = DM(rotary_2);
AY0 = ROTARY_DECR;
AR = AX0 XOR AY0;
IF EQ jump threshold_decr;
AY0 = ROTARY_INCR;
AR = AX0 XOR AY0;
IF EQ jump threshold_incr;

update_threshold:
AX1 = ROTARY_NEUTRAL;
DM(rotary_2) = AX1; //reset the rotary_2 value

//Update LCD with threshold_text
ENA SEC_DAG;
CNTR = 16;
nop;
nop;
DO thres_text_loop UNTIL CE;
   AX0 = DM(I6=M6);
   thres_text_loop: DM(I4=M4) = AX0;
DIS SEC_DAG;
jump compression;

change_ratio:
AX0 = DM(rotary_2);
AY0 = ROTARY_DECR;
AR = AX0 XOR AY0;
IF EQ jump ratio_decr;
AY0 = ROTARY_INCR;
AR = AX0 XOR AY0;
IF EQ jump ratio_incr;

update_ratio:
//Update LCD with ratio_text
AX1 = ROTARY_NEUTRAL;
DM(rotary_2) = AX1; //reset the rotary_2 value
ENA SEC_DAG;
CNTR = 16;
nop;
nop;
DO ratio_text_loop UNTIL CE;
   AX0 = DM(I7=M7);
   ratio_text_loop: DM(I4=M4) = AX0;
DIS SEC_DAG;
jump compression;

change_cutoff:
AX0 = DM(rotary_2);
AY0 = ROTARY_DECR;
AR = AX0 XOR AY0;
IF EQ jump cutoff_decr;
AY0 = ROTARY_INCR;
AR = AX0 XOR AY0;
IF EQ jump cutoff_incr;
update_cutoff:
   //Update LCD with cutoff_text
   AX1 = ROTARY_NEUTRAL;
   DM(rotary_2) = AX1;       //reset the rotary_2 value
   ENA SEC_DAG;
   CNTR = 16;
   nop;
   nop;
   DO cutoff_text_loop UNTIL CE;
      AX0 = DM(I2+M2);
   cutoff_text_loop: DM(I4+M4) = AX0;
   DIS SEC_DAG;
   jump compression;
compression:
   //******compression algorithms*****
   //Peak measurement
   //update vectors
   AR = DM(x_peak_n);        //x_peak(n) -> x_peak{n-1}
   DM(x_peak_n_1) = AR;
   AR = DM(x_ap_half_n);     //x_ap(n) -> x_ap(n-1)
   DM(x_ap_half_n_1) = AR;
   AX0 = DM(Right_Channel);
   AR = ABS AX0;
   MX0 = DM(lp_const);
   MY0 = AR;
   MR = MX0 * MY0 (SS);
   AY0 = MR1;
   AX0 = DM(Left_Channel);
   AR = ABS AX0;
   MX0 = DM(lp_const);
   MY0 = AR;
   MR = MX0 * MY0 (SS);
   AY0 = MR1;
   AR = AX0 + AY0;
   MX0 = AR;
   MY0 = AR;
   MR = MX0 * MY0 (SS);      //x_peak of input mean
   DM(x_peak_n) = MR1;
   //Allpass filtering
   MX0 = DM(lp_const);
   MY0 = DM(x_peak_n);
   MR = MX0 * MY0 (SS);      //0.5 * x_peak(n)
   MX0 = DM(I5+M0);
   MY0 = MR1;
   MR = MX0 * MY0 (SS);      //0.5 * c_coeff * x_peak(n)
   AX0 = MR1;
   MX0 = DM(lp_const);
   MY0 = DM(x_peak_n_1);
   MR = MX0 * MY0 (SS);      //0.5 * x_peak(n-1)
   AY0 = MR1;
   AR = AX0 + AY0;           //0.5 * (c_coeff * x_peak(n) + x_peak(n-1)),
   MX0 = DM(I5+M0);
   MY0 = DM(x_ap_half_n_1);
   MR = MX0 * MY0 (SS);      //c_coeff * 0.5 * x_ap(n-1)
   AX0 = AR;
   AY0 = MR1;
   AR = AX0 - AY0;           //0.5*x_ap(n) = 0.5*(c*x_peak(n) - c*x_ap(n-1) + x_peak(n-1))

SMS044 – DiSPressor – a real-time audio compressor implemented in a DSP  17
DM(x_ap_half_n) = AR;  //0.5*x_ap(n)
MX0 = DM(lp_const);   
MY0 = DM(x_peak_n);   
MR = MX0 * MY0 (SS);  //MR = 0.5*x_peak(n)

AX0 = DM(x_ap_half_n);  
AY0 = AR;               
AR = AX0 + AY0;         

DM(x_rms) = AR;         //x_rms = 0.5*x_ap(n) + 0.5*x_peak(n)

//Translate x_rms left to a logarithmic value (dB), funkar
IO = log;   
II = lin;   
CNTR = 100;  

DO linlog UNTIL CE;
   IF LT jump pop_linlog;  //compare to find corresponding position of x_rms
      MODIFY(I0+=1);     //Step through log vector
      AY0 = DM(I1+=1);  //Step through lin vector
      AR = AX0 - AY0;   //AR = x_rms-lin[i]
   ENDIF;
linlog:

static_curve:          //enables integer mode
   ENA M_MODE;
   AX0 = DM(I0+=0);    // = X
   AY0 = DM(I2+=0);    // = CT
   AR = AX0 - AY0;     //compare with threshold
   DM(X_CT) = AR;      // = X-CT
   IF GT jump above_threshold;  //if RMS energy of X - CT > 0 (-10-(-30)=20>0)
below_threshold:
   //LED för att se om över (LED 8 oxo) eller under threshold (LED 2 xoo)
   AX1 = IOPG;
   DM(iopg_temp_j) = AX1;  //saving memory address in IOPG
   IOPG = General_Purpose_IO;
   AX0 = 2;
   IO(Flags) = AX0;
   AX0 = 8;
   IO(FlagC) = AX0;
   AX1 = DM(iopg_temp_j);
   IOPG = AX1;           //recalling memory address to IOPG

   DIS M_MODE;          //disables integer mode
   MX0 = 0x7FFF;        //set the gain factor to -1
   jump output;

above_threshold:      //funkar
   //LED för att se om över (LED 8 oxo) eller under threshold (LED 2 xoo)
   AX1 = IOPG;
   DM(iopg_temp_j) = AX1;  //saving memory address in IOPG
   IOPG = General_Purpose_IO;
   AX0 = 8;
   IO(Flags) = AX0;
   AX0 = 2;
   IO(FlagC) = AX0;
   AX1 = DM(iopg_temp_j);  //recalling memory address to IOPG
   IOPG = AX1;

   AX0 = 0x0000;
   AY0 = DM(X_CT);
   AR = AX0 - AY0;     //AR = 0-(X-CT)
DM(CT_X) = AR; //AX0 = 0-(X-CT)
AR = DM(I4=0);   
SE = AR;         //Sets the shift factor, ratio
SI = DM(X_CT);   //Sets the number to be shifted
SR = ASHIFT SI (Hi); //Perform the shift operation
DM(temp) = SR1;
AX0 = DM(CT_X);
AY0 = SR1;       //AR = -(X-CT) + 2^(-k)(X-CT)
AR = AX0 + AY0;  //This is the static curve gain factor F in dB.
DM(temp2) = AR;
AX0 = AR;
I0 = log;
I1 = lin;
CNTR = 100;
nop;
nop;
DO loglin UNTIL CE; IF EQ jump pop_loglin;  
IF AX0 = AY0, then we've found the corresponding "linear" value
MODIFY(I1+=1);
AY0 = DM(I1+=1);
AR = AX0 - AY0;  //check the difference between F and current value in log.
loglin:

set_lin_gain:
DIS M_MODE;
MODIFY(I1+=-1); //set I1 to correct position
MX0 = DM(I1+=0); //set gain factor f
jump output;

output:
MY0 = DM(Ieft_Channel);
MR = MX0 * MY0 (SS);
DM(Ieft_Channel_Out) = MR1;
MY0 = DM(Right_Channel);
MR = MX0 * MY0 (SS);
DM(Right_Channel_Out) = MR1;

//reset buffers to position nr 1 for next sample
I0 = log;
I1 = lin;
jump Valid_Frame;

RTS;
threshold_decr:
MODIFY(I2+=1); //Modify value buffer threshold_value
ENA SEC_DAG;
MODIFY(I2+=11); //Modify threshold_text buffer position to insert threshold
MODIFY(I1+=2); //Modify thresholdvalue_text buffer to correct position
AX0 = DM(I0+=1);
DM(I0+=1) = AX0;
AX0 = DM(I0+=1);
DM(I0+=1) = AX0;
I6 = threshold_text;  //reset to first position in vector
MODIFY(I0+=-2);   //Modify to correct position
DIS SEC_DAG;
jump update_threshold;
threshold_incr:
MODIFY(I2+:1);

ENA SEC_DAG;

MODIFY(I6+:11); //Modify threshold_text buffer position to insert threshold
MODIFY(I6+:2); //Modify threshold_value_text buffer to correct position
AX0 = DM(I0+:1);
DN(I6+:1) = AX0;
AX0 = DM(I0+:1);
DN(I6+:1) = AX0;
I6 = threshold_text; //reset to first position in vector
MODIFY(I6+:2); //Modify to correct position

DIS SEC_DAG;
jump update_threshold;

ratio_decr:
MODIFY(I4+:1);

ENA SEC_DAG;

MODIFY(I7+:11); //Modify ratio_text buffer position to insert threshold value
MODIFY(I7+:2); //Modify ratio_value_text buffer to correct position
AX0 = DM(I1+:1);
DN(I7+:1) = AX0;
AX0 = DM(I1+:1);
DN(I7+:1) = AX0;
I7 = ratio_text; //reset to first position in vector
MODIFY(I7+:2); //Modify to correct position

DIS SEC_DAG;
jump update_ratio;

ratio_incr:
MODIFY(I4+:1);

ENA SEC_DAG;

MODIFY(I7+:11); //Modify ratio_text buffer position to insert threshold value
MODIFY(I7+:2); //Modify ratio_value_text buffer to correct position
AX0 = DM(I1+:1);
DN(I7+:1) = AX0;
AX0 = DM(I1+:1);
DN(I7+:1) = AX0;
I7 = ratio_text; //reset to first position in vector
MODIFY(I7+:2); //Modify to correct position

DIS SEC_DAG;
jump update_ratio;

cutoff_decr:
MODIFY(I5+:1); //decrease AT

ENA SEC_DAG;

MODIFY(I2+:10); //Modify cutoff_text buffer position to insert threshold value
MODIFY(I5+:3); //Modify cutoff_value_text buffer to correct position
AX0 = DM(I5+:1);
DN(I2+:1) = AX0;
AX0 = DM(I5+:1);
DN(I2+:1) = AX0;
AX0 = DM(I5+:1);
DN(I2+:1) = AX0;
I2 = cutoff_text;  //reset to first position in vector
MODIFY(I5+=-3);     //Modify to correct position

DIS SEC_DAG;
jump update_cutoff;

cutoff_incr:
MODIFY(I5+=+1);   //decrease AT
ENA SEC_DAG;

MODIFY(I2+=+10);  //Modify cutoff_text buffer position to insert threshold value
MODIFY(I5+=+3);   //Modify cutoffvalue_text buffer to correct position
AX0 = DM(I5+=1);
DN(I2+=1) = AX0;
AX0 = DM(I5+=1);
DN(I2+=1) = AX0;

I2 = cutoff_text;  //reset to first position in vector
MODIFY(I5+=-3);     //Modify to correct position

DIS SEC_DAG;
jump update_cutoff;

pop_linlog:
    POP LOOP;
    jump static_curve;

pop_loglin:
    POP LOOP;
    jump set_lin_gain;
Appendix 3 – compressor.m

Source code for MATLAB file compressor.m:

```matlab
x = wavread('drumloop_mono.wav');
x = x/max(x);
x = x*(1-2^-15);
x = (2^-15)*(ceil(x*2^15));

log = [-99:1:0];
lin = 10.^((log/20));
lin = lin*(1-2^-15);
lin = (2^-15)*(ceil(lin*2^15));
threshold = [0:-1:-49];
k = [0:-1:-5 -15];

fs = 44100;                                 %Sampling frequency in Hz
fc = 10;
c_freq(1:5) = 0;
c_freq(1) = (tan((pi*fc)/fs)-1)/(tan((pi*fc)/fs)+1);
f0 = 30;
c_freq(2) = (tan((pi*fc)/fs)-1)/(tan((pi*fc)/fs)+1);
f0 = 50;
c_freq(3) = (tan((pi*fc)/fs)-1)/(tan((pi*fc)/fs)+1);
f0 = 80;
c_freq(4) = (tan((pi*fc)/fs)-1)/(tan((pi*fc)/fs)+1);
f0 = 150;
c_freq(5) = (tan((pi*fc)/fs)-1)/(tan((pi*fc)/fs)+1);
c_coeff = c_freq;
c_coeff = (2^-15)*(ceil(c_coeff*2^15));

%% new symbols for the project
x_peak_left = zeros(length(x),1);   %position 0 = present sample, position 1 = previous sample
x_peak_right(1:2) = 0;            %position 0 = present sample, position 1 = previous sample
x_ap_left = zeros(length(x),1);     %position 0 = present sample, position 1 = previous sample
x_ap_right = zeros(length(x),1);    %position 0 = present sample, position 1 = previous sample
x_rms_left = zeros(length(x),1);
x_rms_right = 0;
diff = 0;
lp_const = 0.5;     %constant for lp filtering, = 0.5
temp = 0;
minus = -1;

%%******compression algorithm*****
%%Peak measurement
for n = 2:length(x)
    x_peak_left(n) = x(n)*x(n);
x_peak_left(n) = (2^-15)*(ceil(x_peak_left(n)*2^15));

%Allpass filtering
x_ap_left_1 = c_coeff(2) * x_peak_left(n);
x_ap_left_2 = x_ap_left_1 + x_peak_left(n-1);
x_ap_left_3 = c_coeff(2) * x_ap_left(n-1);
x_ap_left_4 = (2^-15)*(ceil(x_ap_left_3*2^15));

x_ap_left(n) = x_ap_left_4 - x_ap_left_3;
x_ap_left(n) = (2^-15)*(ceil(x_ap_left(n)*2^15));

%Lowpass filtering
x_rms_left_1 = lp_const * x_ap_left(n);
x_rms_left_2 = x_rms_left_1 + x_peak_left(n);
x_rms_left_3 = (2^-15)*(ceil(x_rms_left_2*2^15));

x_rms_left(n) = x_rms_left_3 + x_rms_left_2;
x_rms_left(n) = (2^-15)*(ceil(x_rms_left(n)*2^15));
end
```

SMS044 – DiSPressor – a real-time audio compressor implemented in a DSP
figure;
plot(x_rms_left)
X_rms_left = ceil(20*log10(x_rms_left));
F = zeros(length(x),1);
for n = 1:length(X_rms_left)
    if X_rms_left(n) < threshold(thres_k);
        F(n) = 0;
    end
    if X_rms_left(n) > threshold(thres_k)
        F(n) = minus * (X_rms_left(n) - threshold(thres_k)) + floor(2^(k(5))*(X_rms_left(n)-threshold(thres_k)));
    end
end
figure;
plot(X_rms_left)
f = 10.^(F/20);
f = (2^-15)*(ceil(f*2^15));
y = f.*x; %static curve gain factor
y = (2^-15)*(ceil(y*2^15));