



CS-476
REAL-TIME EMBEDDED SYSTEMS

Mini-project: Audio volume control

Elias De Smijter
SCIPER : 366670

Erik Wilhelm Widlund Møllergård
SCIPER : 361948

June 4, 2023

■ Contents

Introduction	2
1 System description	3
1.1 Audio subsystem	3
1.2 Signal processing subsystem	4
1.2.1 Custom signal processing modules	5
2 System usage	8
2.1 Recording and playback	8
2.2 Debugging/troubleshooting	11
3 Profiling	13
3.1 Audio subsystem profiling	13
3.2 SP subsystem profiling	13
4 Annex: Source code	15
4.1 Custom instruction	15
4.2 Hardware Accelerator	15
4.3 Audio subsystem C program	20
4.4 Signal processing subsystem C program	29

■ Introduction

This is the report for the mini-project of Real-Time Embedded Systems (CS-476). The goal is to develop a dual-core, synchronized embedded system that records audio as input, changes the volume either up or down based on the user's choice, and then plays it back as output. This is accomplished using a dual-core system, where the tasks are divided between two NIOS II processors and various other components. One subsystem is tasked with managing the recording and playback of audio, while the other is tasked with performing the volume shift. The two communicate with each other using hardware mailboxes, to ensure synchronous operation.

Three different methods of shifting the volume have been implemented, for profiling purposes. A software implementation in C code, a custom instruction for the processor, and a dedicated Direct Memory Access component that reads data directly from the RAM, performs the volume shift operation, and then stores it directly on RAM again.

■ 1. System description

The system used in this mini-project is roughly based on the dual-core system implemented by the authors in Lab 3 of the same course. At the base is a DE1-SOC board. Two subsystems are implemented, each based around a NIOS II processor. These are capable of completely independent, concurrent operation. They also have access to a set of shared resources and mailboxes that enable synchronization of key tasks between the subsystems. Other shared resources include a PLL and SDRAM controller, needed for accessing the RAM available to the FPGA part of the DE1-SOC board. An overview of the system, with each subsystem compartmentalized, can be seen in figure 1, with shared resources and physical hardware on the DE1-SOC board as white blocks, and subsystems in color.

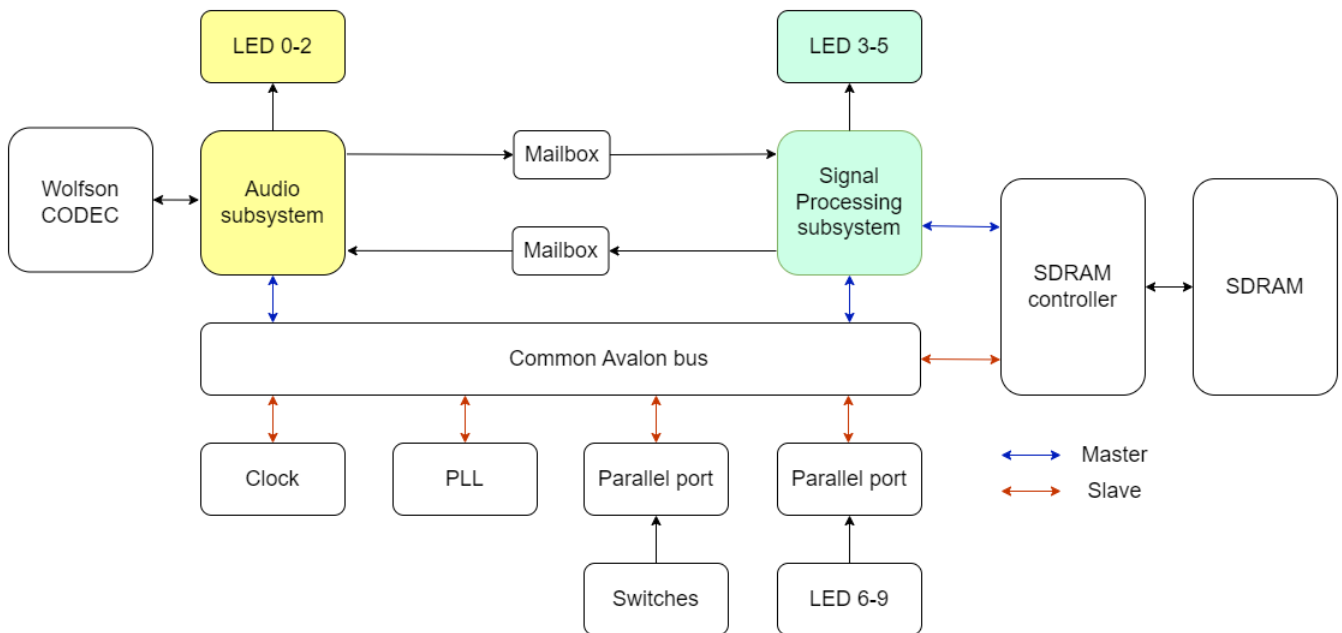


Figure 1: Overall system

Further, tasks are started by the user by way of the switches on the DE1-SOC board. These are shared between the subsystems. Turning a switch that corresponds to a task sends an interrupt to the Audio subsystem, which also serves as the primary subsystem for task initiation. A mail is then sent to the Signal Processing subsystem, informing it about which task has been chosen. Now, both subsystems may start the task at hand, each performing its own part of it. While only the Audio subsystem registers the task starting interrupts, the switches are nonetheless shared as some of them are used to choose certain parameters, which must be available to the Signal Processing subsystem.

1.1 Audio subsystem

The audio subsystem is tasked with recording sound, storing it in memory, reading it from memory, and playing it. The components and inter-connectivity of the subsystem can be seen in figure 2. An Onchip memory is used to store the subsystem's program. A JTAG module handles communication with a computer for such tasks as loading software, printing messages, etc. For the profiling in section 3, a Performance counter is included. A parallel port connects to some LEDs on the DE1-SOC board, and was mainly used for early debugging.

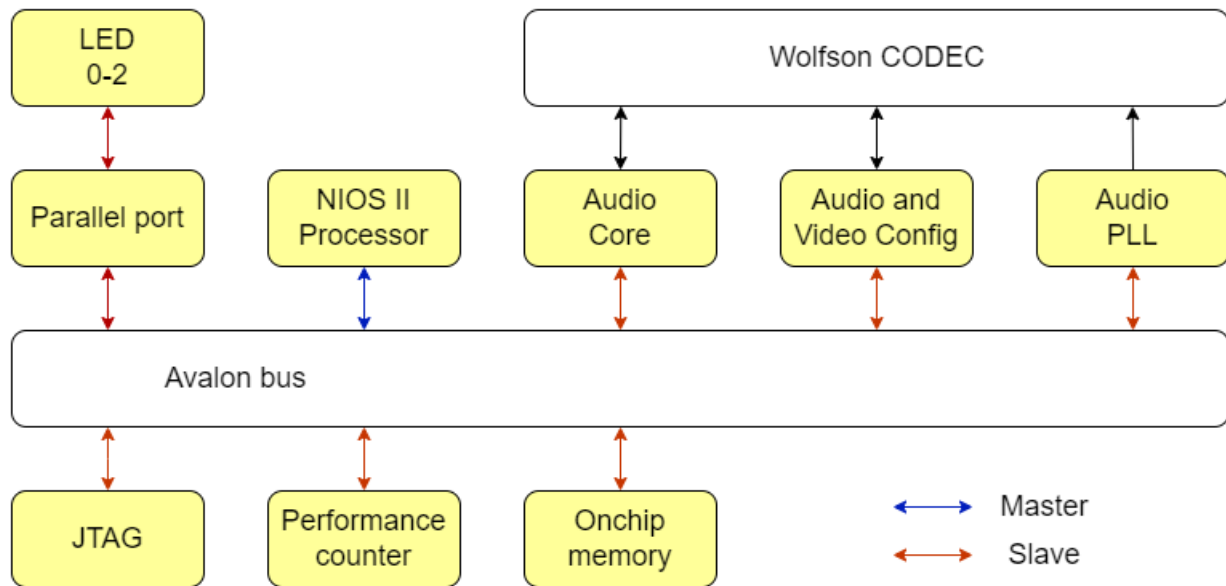


Figure 2: Audio subsystem

The main functionality of the audio subsystem is granted by the three IPs that manage the audio hardware on the DE1-SOC board: a WM8731 Portable Internet Audio CODEC from Wolfson Microelectronics, hereafter referred to as the Wolfson CODEC. These three IPs are the Audio Controller, the Audio and Video Config, and the Audio PLL. These handle the use of the Wolfson CODEC, each playing a crucial role in enabling the capture and playback of sound.

The Audio Controller is the main IP for these tasks, as it provides an interface in C code for handling audio. The component translates C code instructions to instructions for the physical audio hardware on the DE1-SOC board. Essentially, the Audio Core contains two pairs of FIFO blocks, one pair for recording and one pair for playback. They are in pairs due to the Left and Right audio channels: one FIFO in each pair for each channel. When properly configured, the recording FIFOs will contain recorded audio data, and data placed in the playback FIFOs will be played by the Wolfson CODEC.

The Audio and Video Config IP is tasked with the setup of the Wolfson CODEC. At startup, it ensures that the Wolfson CODEC is properly configured and ready to perform the tasks specified by the Audio Controller. Specifically, it configures the Wolfson CODEC with the Audio In and Out paths, the data format, the bit length and the sampling rate to be used. In this project, Audio In is set to Microphone to ADC, and Audio Out to DAC output. This is due to the digital storage and signal processing that takes place, which require the conversion between analog and digital representation. Data format is set to Left Justified, and bit length to 32. The sampling rate is set to 48 kHz, a typical rate for audio sampling.

Finally, the Audio PLL provides a clock signal to the Wolfson CODEC, with a frequency that depends on the settings chosen in the Audio and Video Config. For the chosen sampling rate, the Wolfson CODEC requires a clock signal of either 12.288 MHz or 18.432 MHz. The Audio PLL provides this signal: in this case the 12.288 MHz option is chosen.

1.2 Signal processing subsystem

The task of the Signal Processing subsystem, hereafter referred to as the SP subsystem, is to load data from RAM, perform a signal processing operation on the data, and store the data in a different

location on RAM. For this project, the chosen signal processing operations are to increase and decrease the volume of stored audio data. The subsystem's components and inter-connectivity can be seen in figure 3. Some general components include an on-chip memory, a JTAG module, and some PIO's, all of which have the same tasks as described for the Audio subsystem.

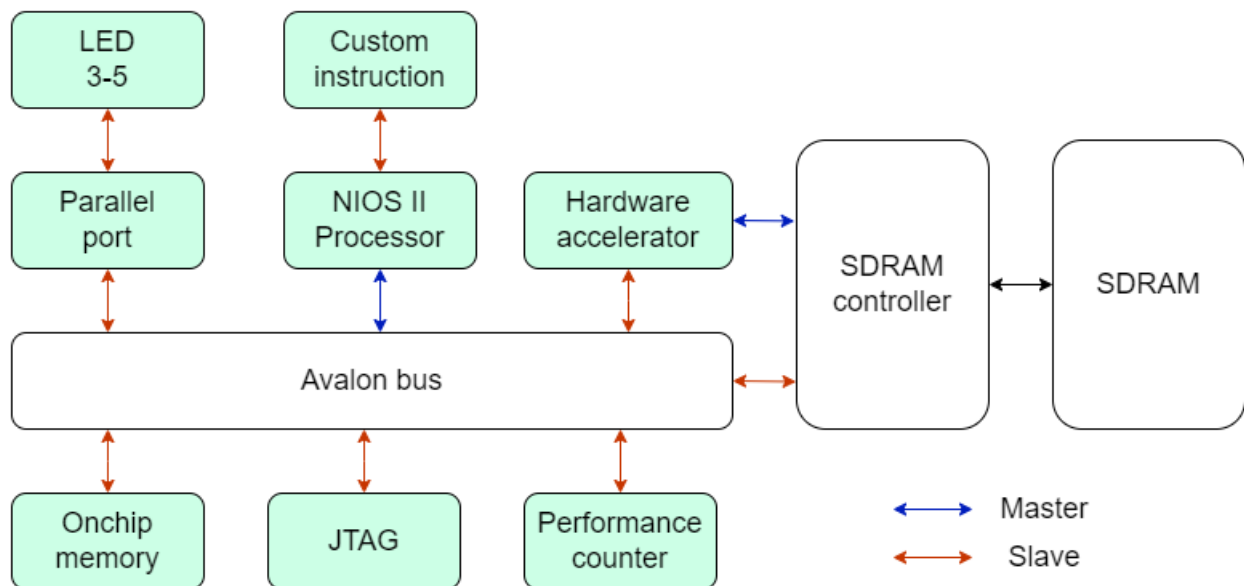


Figure 3: Signal Processing subsystem

The significant components of the SP subsystem are, of course, concerned with signal processing. As the main focus of the project isn't on performing any especially advanced signal processing operation, but rather to run a synchronized dual-core system, the implemented operations have been chosen for their simplicity and straightforward demonstrability. They are to increase and decrease the volume of stored audio. Due to the data format produced by the Audio subsystem, these operations are quite simple: a left-shift of one step doubles the volume, and a right-shift halves it. The only consideration that needs to be taken is to exclude the MSB from the shift, as the data is stored in the Signed format.

1.2.1 Custom signal processing modules

Two separate hardware methods for these operations have been implemented, for enabling a comparison of performance, see section 3. The first is a custom instruction for the NIOS II processor, a quite simple and straightforward way to perform the actual operation in just one clock cycle. The code for the custom instruction can be seen in section 4.1. However, it requires the processor to read the data from RAM before, and store it again after. Since typical usage will entail performing the operation tens of thousands of times, 48 thousand per second of audio, this means that the processor will be busy for a considerable duration.

To free up the processor after initializing the operation on a block of data, the second implementation is a dedicated Hardware Accelerator. While the implemented system in this project has no problems at all to keep up with the rate of audio being captured, see section 3, similar applications may see a Signal Processing subsystem tasked with many different processing tasks, more complex tasks, etc. For such implementations, freeing up the processor as much as possible is a desirable goal.

The hardware accelerator is essentially a DMA module encapsulated in a finite state machine. It

takes in starting addresses for where to load and store the data block, the length of the data block, and which of the two possible operations should be performed on it. Once it has these parameters, it performs all the steps of loading, performing the operation, and storing, all by itself. As it loops over the data in the block until the entire block has been processed, the processor is freed up for other tasks meanwhile. The code for the Hardware Accelerator can be seen in section 4.2.

Included in the code is a debugging process, which outputs some key signals, enabling them to be read by a logic analyzer for debugging, timing measuring and other purposes. Figure 4 shows this output for the capture of a single snippet. Figure 5 shows the capture of 4 snippets; a 1 second recording cycle. Combined, these figures can be seen as an alternate view of the same result as in section 3: the hardware accelerator performs its task in a small fraction of the time taken by the recording process. For more detail on the signals displayed, please refer to the code in section 4.2.

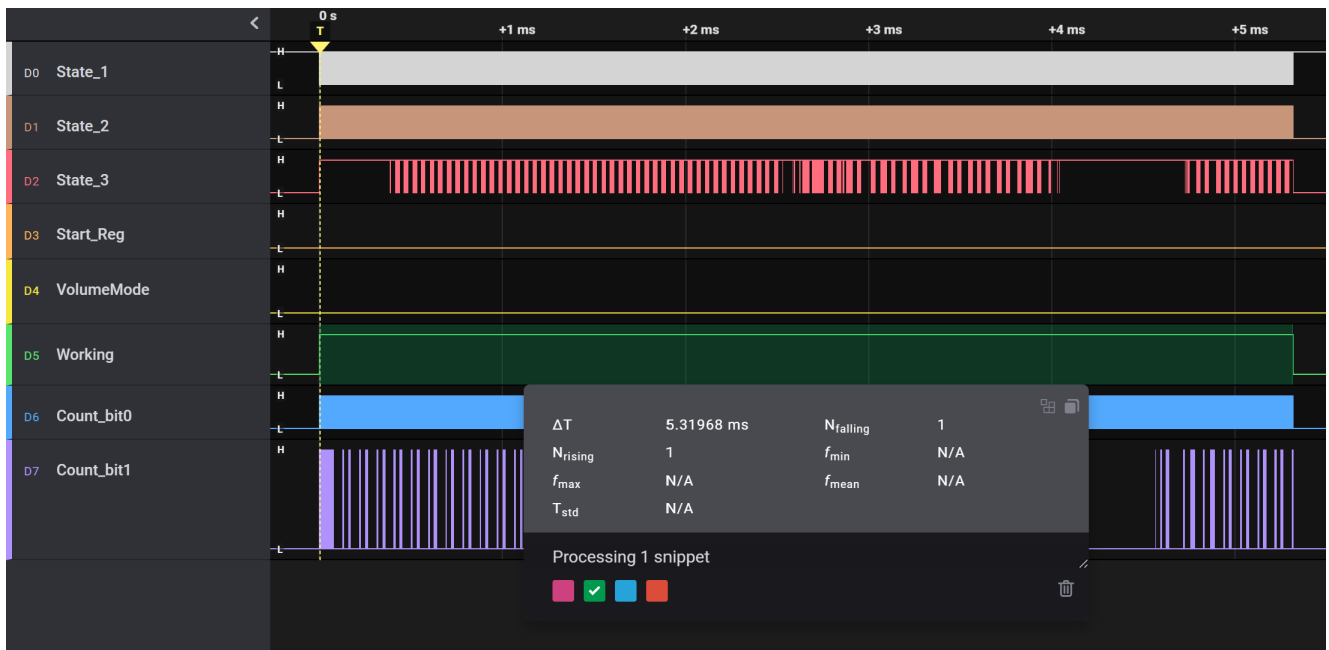


Figure 4: Logic analyzer view of one snippet being processed

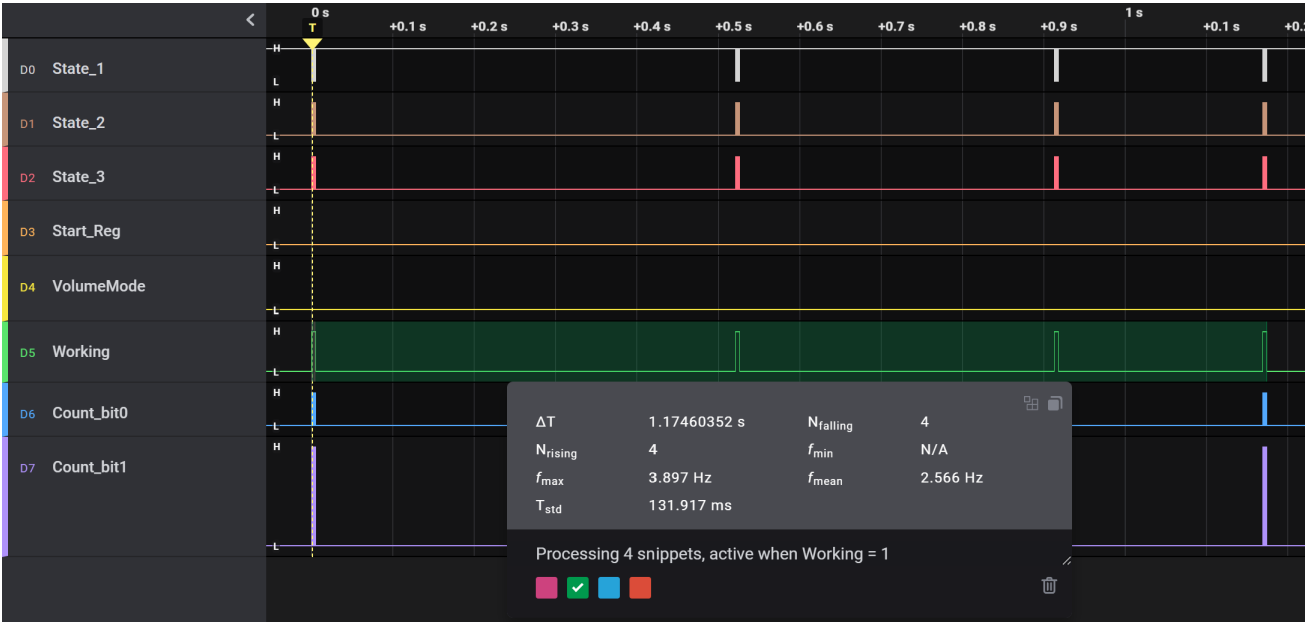


Figure 5: Logic analyzer view of a full recording cycle

■ 2. System usage

The functionality of the system is divided into two parts: actions aimed at the user, and debugging/troubleshooting actions. The former are described in section 2.1, and the latter in section 2.2.

2.1 Recording and playback

Three main options are available to the user: record audio, play audio, and toggle operation type. For these to be performed correctly by the dual-core system, synchronization is required. With the aid of the two mailboxes, each subsystem is capable of communication with the other, enabling synchronous operation by proper usage in the C code.

For recording and processing audio, the user should begin by selecting the desired volume operation by setting the 8th switch on the DE1-SOC board (SW7) to 1 for increasing the volume, or 0 for decreasing. Then, the user may start recording by flicking the first switch (SW0) **from 0 to 1**. This triggers an interrupt sent to both subsystems, initializing the recording-and-processing operation. The workflow of the operation can be seen in figure 6. Of special note are the blocks where mail is sent, and the AND blocks they lead to. Each such structure corresponds to a synchronization between the two processors. Note as well that signals sent by the mailboxes are the only lines that cross from one subsystem to another.

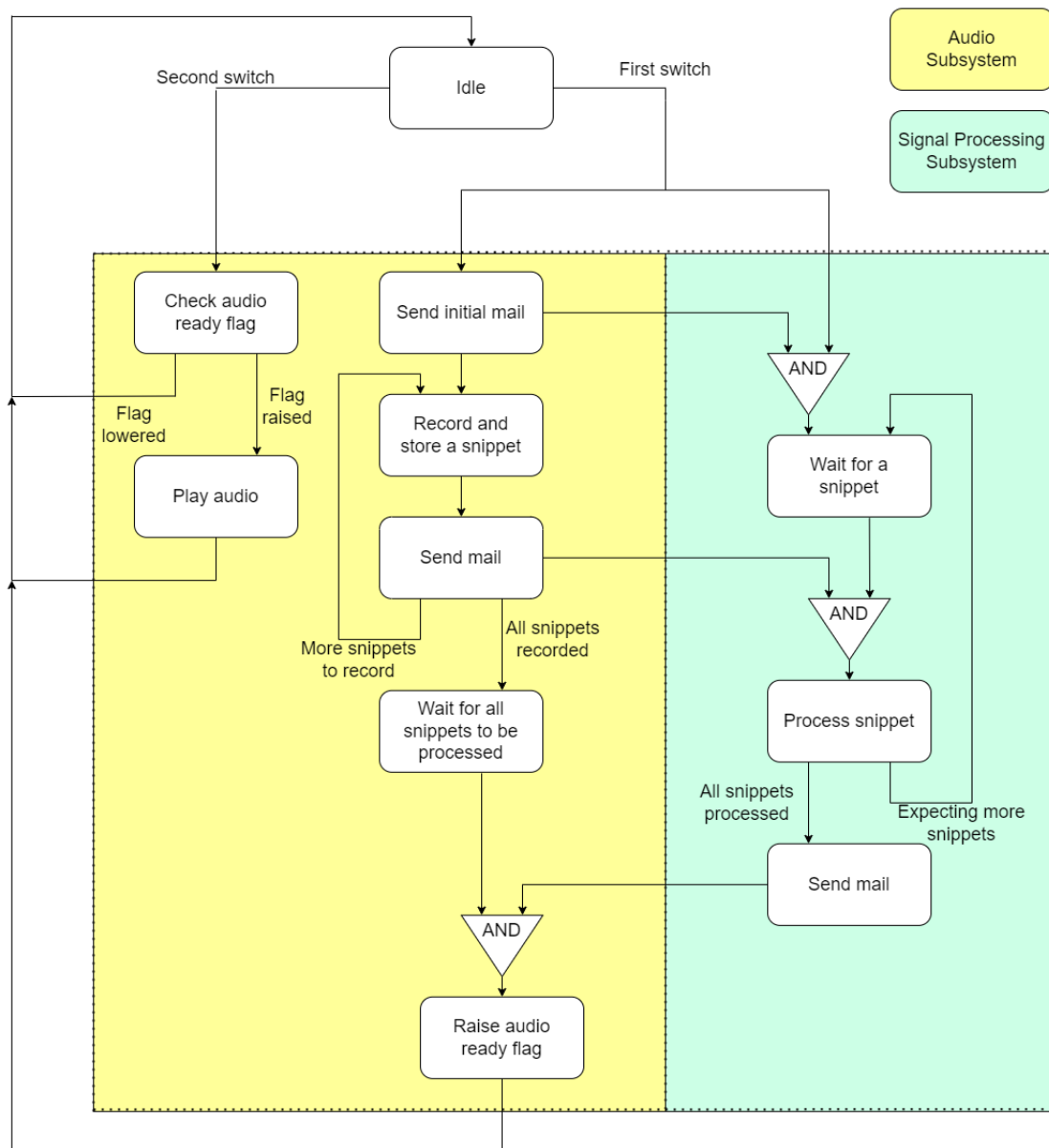


Figure 6: Record, process and playback workflow

The audio to be recorded is divided in blocks, each 0.25 seconds in length, which corresponds to 12 000 32-bit words due to the 48 kHz sampling rate. Such blocks are hereafter referred to as **snippets**. To start with, the Audio subsystem sends an initializing mail to the SP subsystem, containing the starting address of where in the RAM it will store the recording, and the amount of snippets it will be divided into. Then, it begins recording audio captured by the microphone, and storing it. Each time an entire snippet has been stored, a mail is sent to the SP subsystem.

The SP subsystem starts the operation in an idle mode, waiting for the mail with the starting address and the amount of snippets to expect. With this information, it knows where to start and how much data to expect, in total. This information is used to calculate where the storage of processed data should begin, see figure 7 for further details. In this way, the Signal Processing subsystem may begin processing and storing snippets before all recording is complete, without risking placing processed data in an area of the RAM where the Audio subsystem will store unprocessed data.

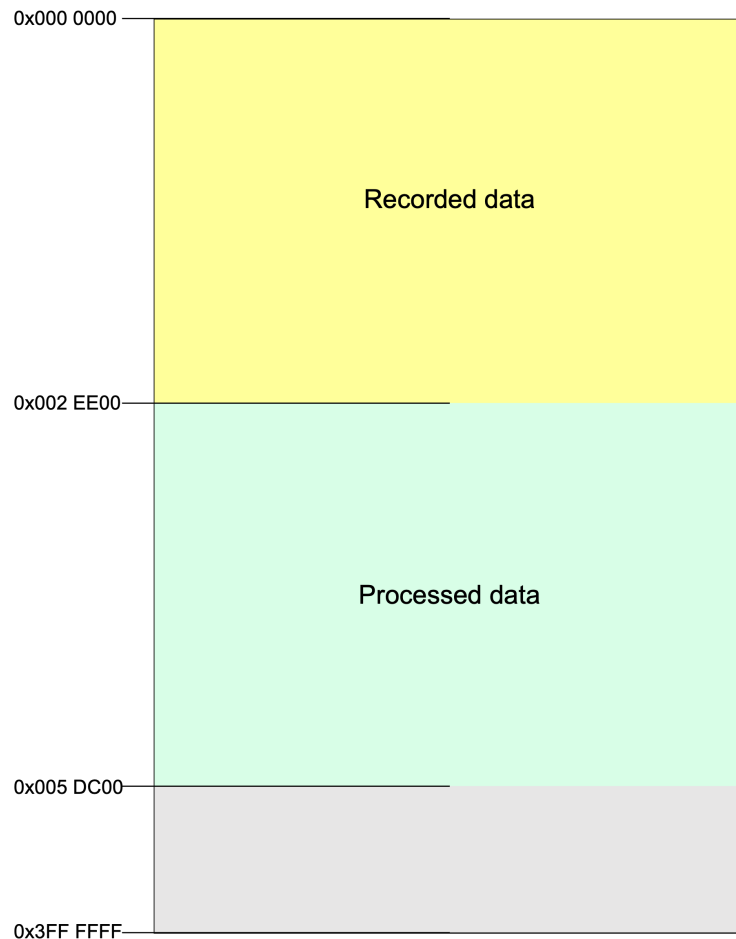
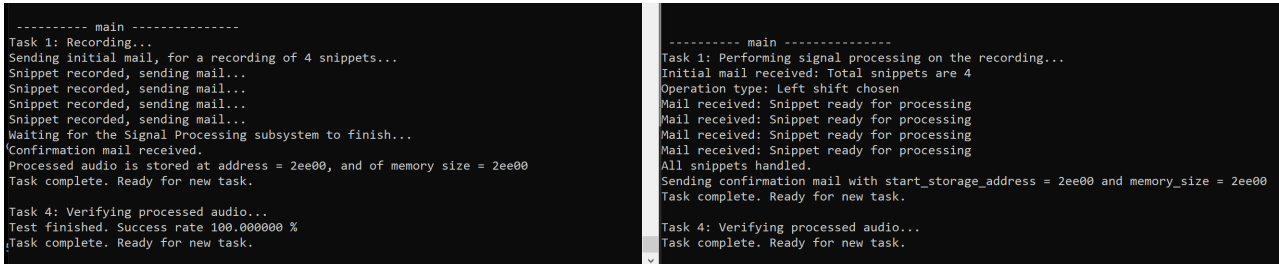


Figure 7: RAM usage map

After receiving the initial mail, the SP subsystem enters another waiting state, in which it remains until a mail is received telling it that a snippet has been stored in RAM by the Audio subsystem. Once a snippet has been recorded and stored, the SP subsystem starts working on it, loading each word, performing the chosen signal processing operation on the word, and storing it in RAM. Then, the SP subsystem waits for the next snippet.

Once the last snippet has been recorded and stored, the Audio subsystem enters a waiting state, while the SP subsystem finishes its work. After handling the last snippet, the SP subsystem sends a mail to the Audio subsystem, with the starting address to where the processed recording is stored in RAM, and the memory size. As the Audio subsystem receives this final mail, it raises a flag signifying that the entire process is complete, and the entire system enters an idle state from which it can either playback the modified audio clip, by flicking the second switch (SW1) **from 0 to 1**, or record-and-process a new one.

When using the system, the terminal outputs are as in figure 8 for recording. The figure also shows the printout of the verification function, see section 2.2. The left side is the Audio subsystem and the right is the SP subsystem. As the playing function merely prints a line saying "Playing...", beyond actually playing the audio, no figure is included.



```

----- main -----
Task 1: Recording...
Sending initial mail, for a recording of 4 snippets...
Snippet recorded, sending mail...
Snippet recorded, sending mail...
Snippet recorded, sending mail...
Snippet recorded, sending mail...
Waiting for the Signal Processing subsystem to finish...
Confirmation mail received.
Processed audio is stored at address = 2ee00, and of memory size = 2ee00
Task complete. Ready for new task.

Task 4: Verifying processed audio...
Test finished. Success rate 100.000000 %
Task complete. Ready for new task.

```

```

----- main -----
Task 1: Performing signal processing on the recording...
Initial mail received: Total snippets are 4
Operation type: Left shift chosen
Mail received: Snippet ready for processing
Mail received: Snippet ready for processing
Mail received: Snippet ready for processing
Mail received: Snippet ready for processing
All snippets handled.
Sending confirmation mail with start_storage_address = 2ee00 and memory_size = 2ee00
Task complete. Ready for new task.

Task 4: Verifying processed audio...
Task complete. Ready for new task.

```

Figure 8: Recording and verification printouts

2.2 Debugging/troubleshooting

As shown in figure 8, there is a function to check the processed data to verify whether the signal processing operation was carried out successfully. It loops over all words in a recording, loading both the recorded and the modified words from RAM. Then, the recorded word is subjected to the C code version of the signal processing operation. Finally, they are compared. If they are not the same, an error is recorded in a counter. After testing all the words, the percentage of correctly processed words is calculated, and printed. As the figure demonstrates, this number is consistently a 100 percent success rate.

Some functions for checking the system status, resetting components, and general debugging have also been prepared. The user may read various statuses and registers of the Audio components, and also reset each of the Audio devices, see figure 9. For detailed information of what the various register values signify, please see the datasheets of each component.

```

Task 5: Checking audio device status...
Checking the settings of the Audio and Video Config device...
The status register is 30102
The Ready bit is 1
The Acknowledge bit is 0
Checking the settings of the Audio Core device...
The status register is 0
the FIFOspace registers are 7f808080 and 5a8c00
The ldada register is 5ed800
The rdata register is = 291500
Task complete. Ready for new task.

Task 6: Reset Audio Core...
Resetting Audio core...
Task complete. Ready for new task.

Task 7: Reset Audio and Video Config device...
Resetting Audio and Video Config device...
Successfully reset!
Task complete. Ready for new task.

Task 5: Checking audio device status...
Checking the settings of the Audio and Video Config device...
The status register is 30102
The Ready bit is 1
The Acknowledge bit is 0
Checking the settings of the Audio Core device...
The status register is 0
the FIFOspace registers are 7f808080 and 87fc00
The ldada register is 830f00
The rdata register is = 479f00
Task complete. Ready for new task.

```

Figure 9: Status reading and reset of audio devices

Of note in the figure is that the only values that change after both components being reset are the

FIFO space registers, and the ldata and rdata registers. These do not concern any settings, but rather describe how much data is in each FIFO block and the value of the most recently added words. As the audio CODEC is constantly recording (ie. putting data in these FIFO blocks), those values are expected to change over time. The other registers and values, that represent settings, stay the same. This indicates that the configuration works reliably.

■ 3. Profiling

For measuring the performance of the 3 different signal processing implementations, a performance counter is used on the SP subsystem. In the audio subsystem a performance analysis is also carried out to see the overhead on the recording. For each implementation, a measurement was made when recording one second of sound, and processing the recording. This means that in total, 3 seconds of audio should be recorded. The results can be seen in figure 10.

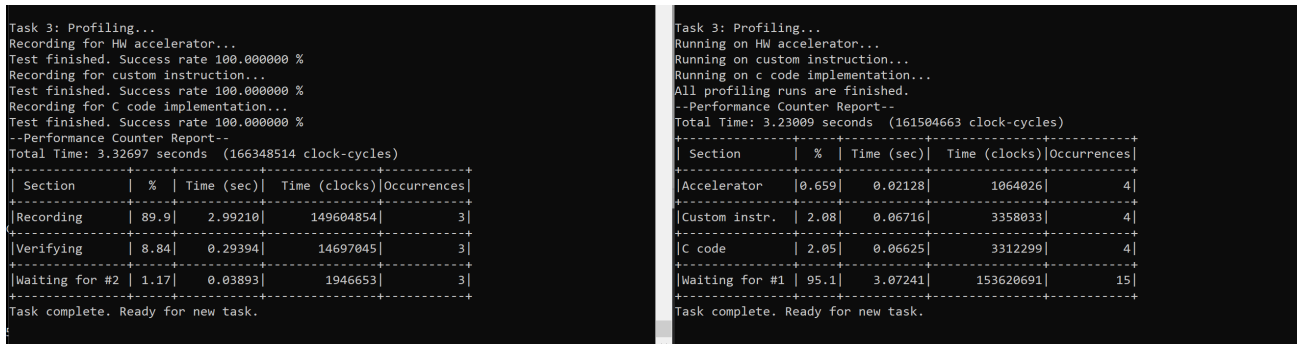


Figure 10: Profiling results

3.1 Audio subsystem profiling

From the view of the audio subsystem, one task gets carried out three times when performing the profiling task. First, a second of audio gets recorded ('Recording' in figure 10). Then, a mail is sent to the SP subsystem and the audio subsystem waits for a mail back ('Waiting for #2'). Finally the result from the SP subsystem gets compared with a C implementation of the functionality to make sure the operation was carried out successfully ('Verifying').

As expected, the actual recording of the sound takes by far the longest time: 3 seconds in total or 90% of the total execution time. The second most time is taken up by the verification (almost 9%). This function takes a while because it has to load each recorded sample from the RAM memory, perform a shift operation on it and a comparison in C. Waiting for the mail from the SP subsystem takes the least amount of time (around 1%). This last amount of time can be influenced by the size of a snippet. At this moment, the SP subsystem is processing the samples fast enough for it to be finished before the next sample arrives. The time the audio subsystem spends waiting for the mail from the SP subsystem thus corresponds to the time it takes for the SP subsystem to process the last snippet. Theoretically, this time could go down when the snippet size is decreased but in reality this adds extra overhead.

Another possible side effect is that at a certain point the SP subsystem will not be able to finish a snippet before a new one arrives so it will begin facing a backlog. The time processing a snippet will become longer relative to the snippet size because the set-up time of the function will become more and more significant in the overall function-time.

3.2 SP subsystem profiling

For the SP subsystem every possible implementation ('Accelerator', 'Custom instruction' or 'C code') occurs four times because the one second recording consists of four snippets. Every implementation

is finished way quicker than the 0.25 seconds time interval that is between each snippet (which can also be seen on figure 5). This is why 95% of the time in the SP subsystem is spent waiting for a mail from the audio subsystem containing the information for a new snippet.

When comparing the three different implementations, the first thing standing out is that the hardware accelerator is way quicker (approximately 3 times quicker) than the two other methods. This is because our algorithm needs a lot of memory accesses. Each sample needs to be read from memory and again stored in memory after the processing operation. The DMA component of the accelerator speeds this part up significantly. The second take-away from the profiling results is that the custom instruction provides no speedup compared to the C code. This is because the only difference between these two implementations is the actual operation on the signal itself. The operation in this project is a simple shift of the data so the C code for this as efficient as an implementation in VHDL.

■ 4. Annex: Source code

4.1 Custom instruction

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SigProcOperation is
  port(
    Param      : in std_logic_vector(31 downto 0);
    InData     : in std_logic_vector(31 downto 0);
    OutData    : out std_logic_vector(31 downto 0)
  );
end SigProcOperation;

architecture design of SigProcOperation is
  begin
    -- Choose volume up or down
    with Param(0) select OutData <=
      -- Increase the volume
      std_logic_vector(shift_left(signed(InData), 1)) when '1',
      -- Decrease the volume
      std_logic_vector(shift_right(signed(InData), 1)) when '0',
      InData when others;
  end design;
```

4.2 Hardware Accelerator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity HW_accelerator is
  port (
    -- Clock & reset interface
    Clk      : in std_logic;
    nReset   : in std_logic;

    -- Avalon master interface
    AM_address      : out std_logic_vector (31 downto 0);
    AM_byteenable   : out std_logic_vector (3 downto 0);
    AM_write        : out std_logic;
    AM_writedata    : out std_logic_vector (31 downto 0);
    AM_read         : out std_logic;
    AM_readdata     : in std_logic_vector (31 downto 0);
    AM_waitrequest  : in std_logic;

    -- Avalon slave interface
    AS_address      : in std_logic_vector (2 downto 0);
    AS_read         : in std_logic;
    AS_readdata     : out std_logic_vector (31 downto 0);
```



```

AS_write      : in std_logic;
AS_writedata  : in std_logic_vector (31 downto 0);

-- Debugging interface
HW_Debug      : out std_logic_vector (7 downto 0)
);
end HW_accelerator;

architecture design of HW_accelerator is

-- Define possible states
type all_states is (Idle, Setup, Init_Read, Finish_Read, Operation, Init_Write, Finish_Write);
signal State : all_states := Idle;

-- Define AS registers
signal StartAddrRead_Reg  : std_logic_vector(31 downto 0);
signal StartAddrWrite_Reg : std_logic_vector(31 downto 0);
signal WordCount_Reg      : std_logic_vector(31 downto 0);
signal VolumeMode_Reg     : std_logic;
signal Working_Reg        : std_logic;
signal Start_Reg          : std_logic;

-- Address / Register
-- 00 / Start Address for Reading
-- 01 / Start Address for Writing
-- 10 / Word count to process
-- 11 / Status (bit 0 = VolumeMode_Reg. high -> volume up, bit 0 low -> volume down
--          bit 1 = Working_Reg. high -> working, bit 1 low -> Idle/Ready for new request)

-- Define internal registers
signal iCurrentAddrRead   : unsigned(31 downto 0);
signal iCurrentAddrWrite  : unsigned(31 downto 0);
signal iWordsLeftCount    : unsigned(31 downto 0);
signal iInData            : std_logic_vector(31 downto 0);
signal iOutData           : std_logic_vector(31 downto 0);

begin

-- Debug process
-- Drives HW_Debug
Debug_Proc: process(Clk, nReset)
begin
    if rising_edge(Clk) then
        -- Always show as much as possible of iWordsLeftCount on pins 7 downto 6
        HW_Debug(7 downto 6) <= std_logic_vector(iWordsLeftCount(1 downto 0));
        -- Always show Working_Reg on pin 5
        HW_Debug(5) <= Working_Reg;
        -- Always show VolumeMode_Reg on pin 4
        HW_Debug(4) <= VolumeMode_Reg;
        -- Always show Start_Reg on pin 3
        HW_Debug(3) <= Start_Reg;
        -- Show the current state on pins 2 downto 0
        case State is

```

```

when Idle =>
    HW_Debug(2) <= '0';
    HW_Debug(1) <= '0';
    HW_Debug(0) <= '1';
when Setup =>
    HW_Debug(2) <= '0';
    HW_Debug(1) <= '1';
    HW_Debug(0) <= '0';
when Init_Read =>
    HW_Debug(2) <= '0';
    HW_Debug(1) <= '1';
    HW_Debug(0) <= '1';
when Finish_Read =>
    HW_Debug(2) <= '1';
    HW_Debug(1) <= '0';
    HW_Debug(0) <= '0';
when Operation =>
    HW_Debug(2) <= '1';
    HW_Debug(1) <= '0';
    HW_Debug(0) <= '1';
when Init_Write =>
    HW_Debug(2) <= '1';
    HW_Debug(1) <= '1';
    HW_Debug(0) <= '0';
when Finish_Write =>
    HW_Debug(2) <= '1';
    HW_Debug(1) <= '1';
    HW_Debug(0) <= '1';
end case;
end if;
end process Debug_Proc;

-- Avalon slave write process
-- Drives the AS registers
AS_WriteProc: process(Clk, nReset)
begin
    if nReset = '0' then
        StartAddrRead_Reg <= (others => '0');
        StartAddrWrite_Reg <= (others => '0');
        WordCount_Reg <= (others => '0');
        VolumeMode_Reg <= '0';
        Start_Reg <= '0';
    elsif rising_edge(Clk) then
        Start_Reg <= '0';
        if AS_write = '1' then
            case AS_address is
                when "000" =>
                    StartAddrRead_Reg <= AS_writedata;
                when "001" =>
                    StartAddrWrite_Reg <= AS_writedata;
                when "010" =>
                    WordCount_Reg <= AS_writedata;
                when "011" =>

```

```

        VolumeMode_Reg <= AS_writedata(0);
    when "100" =>
        Start_Reg <= '1';
    when others => null;
    end case;
end if;
end if;
end process AS_WriteProc;

-- Avalon slave read process
-- Drives AS_readdata
AS_ReadProc: process(Clk)
begin
    if rising_edge(Clk) then
        if AS_read = '1' then
            case AS_address is
                when "000" =>
                    AS_readdata <= StartAddrRead_Reg;
                when "001" =>
                    AS_readdata <= StartAddrWrite_Reg;
                when "010" =>
                    AS_readdata <= WordCount_Reg;
                when "011" =>
                    AS_readdata(1 downto 0) <= Working_Reg & VolumeMode_Reg;
                    AS_readdata(31 downto 2) <= (others => '0');
                when others => null;
            end case;
        end if;
    end if;
end process AS_ReadProc;

-- Main finite state machine and Avalon master interface process
-- Drives State, Avalon master interface, Working_Reg, and internal register
StateProc: process(Clk, nReset)
begin
    if nReset = '0' then
        State <= Idle;
        iCurrentAddrRead <= (others => '0');
        iCurrentAddrWrite <= (others => '0');
        iWordsLeftCount <= (others => '0');
        iInData <= (others => '0');
        iOutData <= (others => '0');
        Working_Reg <= '0';
    elsif rising_edge(Clk) then
        case State is
            when Idle =>
                AM_address <= (others => '0');
                AM_byteenable <= (others => '0');
                AM_write <= '0';
                AM_writedata <= (others => '0');
                AM_read <= '0';
                Working_Reg <= '0';
                if Start_Reg = '1' then

```

```

    State <= Setup;
    Working_Reg <= '1';
end if;
when Setup =>
    iCurrentAddrRead <= unsigned(StartAddrRead_Reg);
    iCurrentAddrWrite <= unsigned(StartAddrWrite_Reg);
    iWordsLeftCount <= unsigned(WordCount_Reg);
    State <= Init_Read;
when Init_Read =>
    AM_write <= '0';
    AM_read <= '1';
    AM_address <= std_logic_vector(iCurrentAddrRead);
    AM_byteenable <= (others => '1');
    State <= Finish_Read;
when Finish_Read =>
    if AM_waitrequest = '0' then
        iInData <= AM_readdata;
        AM_read <= '0';
        State <= Operation;
    end if;
when Operation =>
    case VolumeMode_Reg is
        when '0' =>
            iOutData <= std_logic_vector(shift_right(signed(iInData), 1));
        when '1' =>
            iOutData <= std_logic_vector(shift_left(signed(iInData), 1));
        when others =>
            iOutData <= iInData;
    end case;
    State <= Init_Write;
when Init_Write =>
    AM_write <= '1';
    AM_read <= '0';
    AM_address <= std_logic_vector(iCurrentAddrWrite);
    AM_byteenable <= (others => '1');
    AM_writedata <= iOutData;
    State <= Finish_Write;
    iWordsLeftCount <= iWordsLeftCount - 1;
when Finish_Write =>
    if AM_waitrequest = '0' then
        AM_write <= '0';
        if iWordsLeftCount > 0 then
            iCurrentAddrRead <= iCurrentAddrRead + 4;
            iCurrentAddrWrite <= iCurrentAddrWrite + 4;
            State <= Init_Read;
        else
            Working_Reg <= '0';
            State <= Idle;
        end if;
    end if;
end case;
end if;
end process StateProc;

```

```
end design;
```

4.3 Audio subsystem C program

```
/* Audio system main file
 * To monitor terminal in powershell:
 * nios2-terminal --device 2 --instance 0

 * To download code to board
 * nios2-download -g cpu_0_proj.elf --device 2 --instance 0
 */

#include <stdio.h>
#include "system.h"
#include "io.h"
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/alt_irq.h>
#include "altera_avalon_mailbox_simple.h"
#include <altera_avalon_performance_counter.h>
#include "altera_up_avalon_audio.h"
#include "altera_up_avalon_audio_and_video_config.h"

// PIO definitions for the interrupt handling on the switches
#define PIO_IntrSwitch_Data 0
#define PIO_IntrSwitch_IRQEN 4*2
#define PIO_IntrSwitch_IRQFLAG 4*3

// Audio Core registers
#define AudioControlReg 0
#define AudioFifoSpaceReg 4
#define AudioLData 8
#define AudioRData 12

// Initiate audio_ready_flag
volatile int audio_ready_flag = 0;
// Playback memory location storage array
volatile alt_u32 ready_audio[2];

// Message storage arrays
volatile alt_u32 mail_send[2];
volatile alt_u32 mail_receive[2];

// Volatile variables to prevent overzealous optimization at compilation
volatile int choice = 0;
volatile alt_u32 dataword = 0;
volatile float error_rate;
volatile float success_rate;
```

```

// Setup audio core
alt_up_audio_dev * audio_dev;

// ----- Function declarations -----
// Mailbox functions
void send_callback(void* report, int status);
void send_mail(alt_u32 start_addr, alt_u32 storage_length);
void receive_callback(void* message);
void receive_mail();
// Audio setup functions
void setup_audio(alt_up_audio_dev * audio_dev);
void check_config_status();
void reset_config_device();
void check_core_status();
void reset_core();
// Audio usage functions
void record_process(int seconds, int test, int run_profiling);
void record_audio_snippet(int start_addr_ptr, int record_words);
void play_audio(int start_addr_ptr, int memory_size);
int test_audio_processing(alt_u32 start_recorded, alt_u32 start_processed, alt_u32 memspace);
// Interrupt and task choosing functions
void choose_task(int task);
void isr_switches(void* context);
void setup_switch_interrupts(uint8_t chosen_switches);
// Misc functions
int min(int a, int b);
// ----- End of function declarations -----

// ----- Function definitions -----

// Mailbox functions

void send_callback(void* report, int status)
{
    if(!status)
    {
        alt_printf("Sending completed, contents are %x and %x \n", mail_send[0], mail_send[1]);
    }
    else
    {
        alt_printf("Sending error.\n");
    }
}

void send_mail(alt_u32 start_addr, alt_u32 storage_length)
{
    // Load mailbox
    altera_avalon_mailbox_dev* mailbox_AudioToSigProc = altera_avalon_mailbox_open("/dev/mailbox");
    // Create mail "envelope"
    alt_u32 mail_send[2] = {start_addr, storage_length};
    // Send message
    altera_avalon_mailbox_send(mailbox_AudioToSigProc, mail_send, 0, POLL);
    // Close mailbox

```

```
    altera_avalon_mailbox_close(mailbox_AudioToSigProc);
}

void receive_callback(void* message)
{
    if(message != NULL)
    {
        alt_printf("Receiving completed, contents are %x and %x \n", mail_receive[0], mail_receive[1]);
    }
    else
    {
        alt_printf("Receiving error.\n");
    }
}

void receive_mail()
{
    // Open mailbox
    altera_avalon_mailbox_dev* mailbox_SigProcToAudio = altera_avalon_mailbox_open("/dev/mailbox");
    // Wait for mail confirming the processing is complete
    altera_avalon_mailbox_retrieve_poll(mailbox_SigProcToAudio, mail_receive, 0);
    alt_dcache_flush_all();
}

// Audio setup functions

void setup_audio(alt_up_audio_dev * audio_dev)
{
    // open the Audio port
    audio_dev = alt_up_audio_open_dev ("/dev/audio_0");
    if ( audio_dev == NULL)
        alt_printf("Error: could not open audio device \n");
    else
        alt_printf("Opened audio device \n");
}

void check_config_status()
{
    alt_printf("Checking the settings of the Audio and Video Config device... \n");
    alt_up_av_config_dev* config_device = alt_up_av_config_open_dev(AUDIO_AND_VIDEO_CONFIG_0_NAME);
    alt_u32 status = IORD_32DIRECT(AUDIO_AND_VIDEO_CONFIG_0_BASE, 4);
    alt_printf("The status register is %x \n", status);
    int ready = alt_up_av_config_read_ready(config_device);
    alt_printf("The Ready bit is %x \n", ready);
    int ackbit = alt_up_av_config_read_acknowledge(config_device);
    alt_printf("The Acknowledge bit is %x \n", ackbit);
}

void reset_config_device()
{
    alt_up_av_config_dev* config_device = alt_up_av_config_open_dev(AUDIO_AND_VIDEO_CONFIG_0_NAME);
    alt_printf("Resetting Audio and Video Config device... \n");
    int reset = alt_up_av_config_reset(config_device);
}
```

```
if(reset == 0)
{
    alt_printf("Successfully reset! \n");
}
else
{
    alt_printf("Reset failed, returned %x \n", reset);
}
}

void check_core_status()
{
    alt_printf("Checking the settings of the Audio Core device... \n");
    alt_u32 statusreg = IORD_32DIRECT(AUDIO_0_BASE, 0);
    alt_printf("The status register is %x \n", statusreg);
    alt_u32 fiforeg = IORD_32DIRECT(AUDIO_0_BASE, 4);
    alt_u32 fiforeg2 = IORD_32DIRECT(AUDIO_0_BASE, 8);
    alt_printf("the FIFOspace registers are %x and %x \n", fiforeg, fiforeg2);
    alt_u32 ldata = IORD_32DIRECT(AUDIO_0_BASE, 8);
    alt_printf("The ldada register is %x \n", ldata);
    alt_u32 rdata = IORD_32DIRECT(AUDIO_0_BASE, 12);
    alt_printf("The rdata register is = %x \n", rdata);
}

void reset_core()
{
    alt_printf("Resetting Audio core... \n");
    alt_up_audio_reset_audio_core(audio_dev);
}

// Audio usage functions

void record_process(int seconds, int test, int run_profiling)
{
    if(run_profiling == 1)
    {
        // Start profiling for the time spent recording
        PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE, 1);
    }
    // Lower audio_ready_flag
    audio_ready_flag = 0;
    // Divide the desired recording time into quarter-second snippets
    int snippets = seconds*4;
    int snippet_counter = 0;
    // 48 kHz sampling frequency => 12k samples per snippet
    int words_per_snippet = 12000;
    unsigned int start_address = SDRAM_CONTROLLER_2_BASE;
    // Send mail to Signal Processing subsystem, letting it know a recording process has started
    // stored and how many snippets it will contain
    if(run_profiling != 1)
    {
        alt_printf("Sending initial mail, for a recording of %x snippets... \n", snippets);
    }
}
```



```
send_mail(start_address, snippets);
unsigned int current_address;
while(snippet_counter < snippets)
{
    current_address = start_address + snippet_counter*words_per_snippet*4;
    // Record
    record_audio_snippet(current_address, words_per_snippet);
    if(run_profiling != 1)
    {
        alt_printf("Snippet recorded, sending mail... \n");
    }
    // Send mail with snippet info
    send_mail(current_address, words_per_snippet);

    snippet_counter = snippet_counter + 1;
}
if(run_profiling == 1)
{
    // Stop profiling for the time spent recording
    PERF_END(PERFORMANCE_COUNTER_0_BASE, 1);
    // Start profiling for the time spent waiting for the other subsystem
    PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE, 3);
}
else
{
    alt_printf("Waiting for the Signal Processing subsystem to finish... \n");
}
receive_mail();
if(run_profiling == 1)
{
    // Stop profiling for the time spent waiting for the other subsystem
    PERF_END(PERFORMANCE_COUNTER_0_BASE, 3);
}
// Store address and data size of the processed audio
ready_audio[0] = mail_receive[0];
ready_audio[1] = mail_receive[1];
// Raise flag, enabling playback of the processed audio
audio_ready_flag = 1;
if(run_profiling != 1)
{
    alt_printf("Confirmation mail received.\nProcessed audio is stored at address = %x, and of
}
// Test the work
if(test == 1)
{
    if(run_profiling == 1)
    {
        // Start profiling for the time spent verifying the results
        PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE, 2);
    }
    test_audio_processing(SDRAM_CONTROLLER_2_BASE, ready_audio[0], ready_audio[1]);
    if(run_profiling == 1)
    {
```

```
// Stop profiling for the time spent verifying the results
PERF_END(PERFORMANCE_COUNTER_0_BASE, 2);
}
}
}

void record_audio_snippet(int start_addr_ptr, int record_words)
{
    int words_recorded = 0;
    int words_in_fifo = 0;
    int current_words = 0;
    int i;
    while( words_recorded < record_words)
    {
        words_in_fifo = (IORD_32DIRECT(AUDIO_0_BASE, AudioFifoSpaceReg) & 0xff00) >> 8;
        current_words = min(words_in_fifo, record_words - words_recorded);
        i = 0;
        while(i < current_words)
        {
            dataword = IORD_32DIRECT(AUDIO_0_BASE, AudioLData);
            IOWR_32DIRECT(start_addr_ptr, (words_recorded + i)*4, dataword);
            i ++;
        }
        words_recorded += current_words;
    }
}

void play_audio(int start_addr_ptr, int memory_size)
{
    int play_words = memory_size/4;
    int words_played = 0;
    alt_32 current_address = start_addr_ptr;
    int free_space = 0;
    int words_to_buffer;
    int i;
    alt_u32 resultword;
    while( words_played < play_words)
    {
        // Check how much space is available
        free_space = (IORD_32DIRECT(AUDIO_0_BASE, AudioFifoSpaceReg) & 0xff000000) >> 24;
        words_to_buffer = min(free_space, play_words - words_played);
        i = 0;
        while(i < words_to_buffer)
        {
            current_address += 4;
            resultword = IORD_32DIRECT(SDRAM_CONTROLLER_2_BASE, current_address);
            IOWR_32DIRECT(AUDIO_0_BASE, AudioLData, resultword);
            IOWR_32DIRECT(AUDIO_0_BASE, AudioRData, resultword);
            i ++;
        }
        words_played += words_to_buffer;
    }
}
```

```

int test_audio_processing(alt_u32 start_recorded, alt_u32 start_processed, alt_u32 memspace)
{
    int wordcount = memspace/4;
    int words_tested = 0;
    int faults_found = 0;
    alt_u32 test1;
    alt_u32 test2;
    int op_type;
    if((IORD_32DIRECT(PIO_2_BASE, 0) & 0x80) == 0)
    {
        op_type = 0;
    }
    else
    {
        op_type = 1;
    }
    while(words_tested < wordcount)
    {
        // Load both
        test1 = IORD_32DIRECT(start_recorded, words_tested*4);
        test1 = (signed)test1;
        test2 = IORD_32DIRECT(start_processed, words_tested*4);
        // Test
        if(op_type == 0)
        {
            test1 = (test1 & 0x80000000)+((test1 & 0xffffffff)>>1);
        }
        else if (op_type == 1)
        {
            test1 = (test1 << 1);
        }
        if((int)test1 != (int)test2)
        {
            faults_found += 1;
        }
        // Increment
        words_tested ++;
    }
    error_rate = (float)faults_found/(float)words_tested;
    success_rate = 100*(1 - error_rate);
    printf("Test finished. Success rate %f %% \n", success_rate);
    return faults_found;
}

//Interrupt and task choosing functions

void choose_task(int task)
{
    if(task == 1)
    {
        alt_printf("Task 1: Recording... \n");
        // First switch (SW0) Record audio for a second, without testing, without profiling, using

```

```
    record_process(1, 0, 0);
}
else if(task == 2)
{
    alt_printf("Task 2: Playing... \n");
    // Second switch (SW1): Playback processed audio if ready
    if(audio_ready_flag)
    {
        play_audio((int)ready_audio[0], (int)ready_audio[1]);
    }
    else
    {
        alt_printf("Error: No audio is ready! Please record first. \n");
    }
}
else if(task == 4)
{
    alt_printf("Task 3: Profiling... \n");
    // Third switch (SW2): record thrice, once for each method on the Signal Processing side,
    // Start overall performance counter
    PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
    PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);
    // Record for HW accelerator
    alt_printf("Recording for HW accelerator... \n");
    record_process(1, 1, 1);
    // Record for custom instruction
    alt_printf("Recording for custom instruction... \n");
    record_process(1, 1, 1);
    // Record for c code implementation
    alt_printf("Recording for C code implementation... \n");
    record_process(1, 1, 1);
    // Stop overall performance counter and print results
    PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);
    perf_print_formatted_report(PERFORMANCE_COUNTER_0_BASE, alt_get_cpu_freq(), 3, "Recording")
}
else if(task == 8)
{
    alt_printf("Task 4: Verifying processed audio... \n");
    // Fourth switch (SW3): Test audio by immediately playing back what is recorded
    if(audio_ready_flag)
    {
        test_audio_processing(SDRAM_CONTROLLER_2_BASE, ready_audio[0], ready_audio[1]);
    }
    else
    {
        alt_printf("Error: No audio is ready! Please record first. \n");
    }
}
else if(task == 16)
{
    alt_printf("Task 5: Checking audio device status... \n");
    check_config_status();
    check_core_status();
}
```

```
}
else if(task == 32)
{
    alt_printf("Task 6: Reset Audio Core... \n");
    reset_core();
}
else if(task == 64)
{
    alt_printf("Task 7: Reset Audio and Video Config device... \n");
    reset_config_device();
}
alt_printf("Task complete. Ready for new task. \n\n", task);
}

void isr_switches(void* context)
{
    // Read interrupt source
    uint8_t pinvals = (IORD_8DIRECT(PIO_2_BASE, PIO_IntrSwitch_IRQFLAG) & 0x7f);
    // Save choice
    choice = (int)pinvals;
    // Send mail to the other CPU, letting it know that the flag can be lowered
    send_mail(choice, 0);
    // Clear the interrupt flag
    IOWR_8DIRECT(PIO_2_BASE, PIO_IntrSwitch_IRQFLAG, pinvals);
}

void setup_switch_interrupts(uint8_t chosen_switches)
{
    // Setup interrupts on the chosen switches
    IOWR_8DIRECT(PIO_2_BASE, PIO_IntrSwitch_IRQEN, chosen_switches);
    alt_irq_register(PIO_2_IRQ, NULL, isr_switches);
    return;
}

// Misc functions
int min(int a, int b)
{
    return (a > b) ? b : a;
}

// ----- Main function -----

int main()
{
    setup_audio(audio_dev);
    // Setup interrupts on the first 6 switches
    setup_switch_interrupts(0x7f);
    alt_printf("\n \n \n \n ----- main ----- \n");

    // Wait for switches
    while(1)
    {
        // Poll choice once every millisecond
```

```

    usleep(1000);
    if(choice != 0)
    {
        choose_task(choice);
        choice = 0x0;
    }
};

    return 0;
}

```

4.4 Signal processing subsystem C program

```

/* Signal processing system main file
 * To monitor terminal in powershell:
 * nios2-terminal --device 2 --instance 1

 * To download code to board
 * nios2-download -g cpu_1_proj.elf --device 2 --instance 1
 */

#include <stdio.h>
#include "system.h"
#include "io.h"
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <sys/alt_irq.h>
#include "altera_avalon_mailbox_simple.h"
#include <altera_avalon_performance_counter.h>

// PIO definitions for the interrupt handling on the switches
#define PIO_IntrSwitch_Data 0
#define PIO_IntrSwitch_IRQEN 4*2
#define PIO_IntrSwitch_IRQFLAG 4*3

// HW Accelerator address offset definitions for writing
#define HW_ACC_STARTREADADDR 0
#define HW_ACC_STARTWRITEADDR 4
#define HW_ACC_WORDCOUNT 8
#define HW_ACC_VOLUMEMODE 12
#define HW_ACC_START 16
// HW Accelerator address offset definitions for reading
#define HW_ACC_STATUS 12

// Message storage arrays
volatile alt_u32 mail_send[2];
volatile alt_u32 mail_receive[2];

// Task choice variable
volatile int choice = 0;

// ----- Function declarations -----

```

```

// Mailbox functions
void send_callback(void* report, int status);
void send_mail(alt_u32 start_addr, alt_u32 storage_length);
void receive_callback(void* message);
void receive_mail();
// Interrupt and task choice functions
void choose_task(int task);
// HW Accelerator functions
void HW_Accelerator_start(alt_u32 start_addr, alt_u32 store_addr, alt_u32 word_count, int op_t
void HW_Accelerator_wait_for_ready();
// Comparison methods
void c_function(alt_u32 start_addr, alt_u32 store_addr, alt_u32 word_count, int op_type);
void custom_function(alt_u32 start_addr, alt_u32 store_addr, alt_u32 word_count, int op_type);
// Main function for running the subsystem task
void signal_processing_function(int choice, int run_profiling);
// ----- End of function declarations -----

// ----- Function definitions -----

// Mailbox functions

void send_callback(void* report, int status)
{
    if(!status)
    {
        alt_printf("Sending completed, contents are %x and %x \n", mail_send[0], mail_send[1]);
    }
    else
    {
        alt_printf("Sending error.\n");
    }
}

void send_mail(alt_u32 start_addr, alt_u32 storage_length)
{
    //alt_printf("----- send_mail ----- \n");
    // Load mailbox
    altera_avalon_mailbox_dev* mailbox_SigProcToAudio = altera_avalon_mailbox_open("/dev/mailbox
    // Create mail "envelope"
    alt_u32 mail_send[2] = {start_addr, storage_length};
    // Send message
    altera_avalon_mailbox_send(mailbox_SigProcToAudio, mail_send, 0, POLL);
    // Close mailbox
    altera_avalon_mailbox_close(mailbox_SigProcToAudio);
}

void receive_callback(void* message)
{
    if(message != NULL)
    {
        alt_printf("Receiving completed, contents are %x and %x \n", mail_receive[0], mail_receive
    }
    else

```

```
{
    alt_printf("Receiving error.\n");
}

}

void receive_mail()
{
    // Load mailbox
    altera_avalon_mailbox_dev* mailbox_AudioToSigProc = altera_avalon_mailbox_open("/dev/mailbox");
    altera_avalon_mailbox_retrieve_poll(mailbox_AudioToSigProc, mail_receive, 0);
    alt_dcache_flush_all();
}

// Interrupt and task choosing functions

void choose_task(int task)
{
    if(task == 1)
    {
        alt_printf("Task 1: Performing signal processing on the recording... \n");
        // First switch (SW0)
        signal_processing_function(1, 0);
    }
    else if(task == 2)
    {
        alt_printf("Task 2: Playing... \n");
        // Second switch (SW1)
        // Play audio, so do nothing here...
    }
    else if(task == 4)
    {
        alt_printf("Task 3: Profiling... \n");
        // Third switch (SW2): run with profiling
        // Start overall performance counter
        PERF_RESET(PERFORMANCE_COUNTER_1_BASE);
        PERF_START_MEASURING(PERFORMANCE_COUNTER_1_BASE);
        alt_printf("Running on HW accelerator... \n");
        // Run on HW accelerator with profiling
        signal_processing_function(1, 1);
        alt_printf("Running on custom instruction... \n");
        // Run on custom instruction with profiling
        signal_processing_function(2, 1);
        alt_printf("Running on c code implementation... \n");
        // Run on c code implementation with profiling
        signal_processing_function(3, 1);
        alt_printf("All profiling runs are finished. \n");
        // Stop overall performance counter and print results
        PERF_STOP_MEASURING(PERFORMANCE_COUNTER_1_BASE);
        perf_print_formatted_report(PERFORMANCE_COUNTER_1_BASE, alt_get_cpu_freq(), 4, "Accelerato
    }
    else if(task == 8)
    {
        alt_printf("Task 4: Verifying processed audio... \n");
    }
}
```



```

    // Fourth switch:
    // Audio test, so do nothing here
}
else if(task == 16)
{
    alt_printf("Task 5: Checking audio device status... \n");
    // Audio check, so do nothing here
}
else if(task == 32)
{
    alt_printf("Task 6: Reset Audio Core... \n");
    // Audio component reset, so do nothing here
}
else if(task == 64)
{
    alt_printf("Task 7: Reset Audio and Video Config device... \n");
    // Audio component reset, so do nothing here
}
alt_printf("Task complete. Ready for new task. \n\n", task);
}

// Hardware accelerator functions

void HW_Accelerator_start(alt_u32 start_addr, alt_u32 store_addr, alt_u32 word_count, int op_type)
{
    // Wait for accelerator to be available
    HW_Accelerator_wait_for_ready();
    //alt_printf("Sending arguments to accelerator \n");
    // Send starting address for reading
    IOWR_32DIRECT(HW_ACCELERATOR_0_BASE, HW_ACC_STARTREADADDR, start_addr);
    // Send starting address for writing
    IOWR_32DIRECT(HW_ACCELERATOR_0_BASE, HW_ACC_STARTWRITEADDR, store_addr);
    // Send word count
    IOWR_32DIRECT(HW_ACCELERATOR_0_BASE, HW_ACC_WORDCOUNT, word_count);
    // Send volume mode
    IOWR_32DIRECT(HW_ACCELERATOR_0_BASE, HW_ACC_VOLUMEMODE, (alt_u32)op_type);
    // Run Hardware accelerator
    IOWR_32DIRECT(HW_ACCELERATOR_0_BASE, HW_ACC_START, 0x1);
    //alt_printf("Accelerator has been started \n");
}

void HW_Accelerator_wait_for_ready()
{
    // Check if the accelerator is already working, and wait until it's done
    int working = 0;
    alt_u32 status;
    status = IORD_32DIRECT(HW_ACCELERATOR_0_BASE, HW_ACC_STATUS);
    // Shift out the 'VolumeMode' bit, leaving only zeros and the 'Working' bit
    working = (int)(status >> 1);
    while(working != 0)
    {
        // Wait for the accelerator to be available
        status = IORD_32DIRECT(HW_ACCELERATOR_0_BASE, 12);
    }
}

```

```
// Shift out the 'VolumeMode' bit, leaving only zeros and the 'Working' bit
working = (int)(status >> 1);
}
}

// Comparison methods

void c_function(alt_u32 start_addr, alt_u32 store_addr, alt_u32 word_count, int op_type)
{
    alt_u32 data = 0;
    if(op_type == 0)
    {
        while(word_count > 0)
        {
            // Load data
            data = IORD_32DIRECT(start_addr, 0);
            // Perform operation
            data = (signed)data;
            // Shift right
            data = (data & 0x80000000)+((data & 0xffffffff)>>1);
            // Store data
            IOWR_32DIRECT(SDRAM_CONTROLLER_2_BASE, (alt_u32)store_addr, (alt_u32)data);
            // Update variables
            start_addr += 4;
            store_addr += 4;
            word_count --;
        }
    }
    else
    {
        while(word_count > 0)
        {
            // Load data
            data = IORD_32DIRECT(start_addr, 0);
            // Perform operation
            data = (signed)data;
            // Shift left
            data = (data << 1);
            // Store data
            IOWR_32DIRECT(SDRAM_CONTROLLER_2_BASE, (alt_u32)store_addr, (alt_u32)data);
            // Update variables
            start_addr += 4;
            store_addr += 4;
            word_count --;
        }
    }
}

void custom_function(alt_u32 start_addr, alt_u32 store_addr, alt_u32 word_count, int op_type)
{
    alt_u32 data = 0;
    while(word_count > 0)
    {
```

```

    // Load data
    data = IORD_32DIRECT(start_addr, 0);
    // Perform operation
    data = (signed)data;
    // Use custom instruction
    data = ALT_CI_SIGPROCOPERATION_0(data, op_type);
    // Store data
    IOWR_32DIRECT(SDRAM_CONTROLLER_2_BASE, (alt_u32)store_addr, (alt_u32)data);
    // Update variables
    start_addr += 4;
    store_addr += 4;
    word_count --;
}
}

// ----- Overall subsystem function -----
void signal_processing_function(int choice, int run_profiling)
{
    int op_type;
    // Receive initial mail, containing starting address and amount of snippets
    if(run_profiling == 1)
    {
        // Start profiling for the time spent in waiting for the other subsystem
        PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE, 4);
    }
    receive_mail();
    if(run_profiling == 1)
    {
        // Stop profiling for the time spent in waiting for the other subsystem
        PERF_END(PERFORMANCE_COUNTER_1_BASE, 4);
    }
    int total_snippets = mail_receive[1];
    if(run_profiling != 1)
    {
        alt_printf("Initial mail received: Total snippets are %x \n", total_snippets);
    }
    // 48 kHz sampling frequency => 12k samples per snippet
    int words_per_snippet = 12000;
    int handled_snippets = 0;
    int memory_size = total_snippets*words_per_snippet*4;
    int start_storage_address = mail_receive[0] + memory_size;
    // Check if the volume shift switch is set to up or down
    if((IORD_32DIRECT(PIO_2_BASE, 0) & 0x80) == 0)
    {
        op_type = 0;
        if(run_profiling != 1)
        {
            alt_printf("Operation type: Right shift chosen \n");
        }
    }
    else
    {
        op_type = 1;
    }
}

```

```
if(run_profiling != 1)
{
    alt_printf("Operation type: Left shift chosen \n");
}
}
while(handled_snippets < total_snippets)
{
    // Wait for mail, signifying that a snippet has been recorded and is ready to process
    if(run_profiling == 1)
    {
        // Start profiling for the time spent in waiting for the other subsystem
        PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE, 4);
    }
    receive_mail();
    if(run_profiling == 1)
    {
        // Stop profiling for the time spent in waiting for the other subsystem
        PERF_END(PERFORMANCE_COUNTER_1_BASE, 4);
    }
    if(run_profiling != 1)
    {
        alt_printf("Mail received: Snippet ready for processing \n");
    }
    // Prepare parameters
    int snippet_starting_address = mail_receive[0];
    int snippet_storage_address = start_storage_address + handled_snippets*words_per_snippet*4;
    int snippet_word_count = mail_receive[1];
    if(choice == 1)
    {
        if(run_profiling == 1)
        {
            // Start profiling for the current choice
            PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE, choice);
        }
        HW_Accelerator_start(snippet_starting_address, snippet_storage_address, snippet_word_count);
        if(run_profiling == 1)
        {
            // Wait for the accelerator to finish
            HW_Accelerator_wait_for_ready();
            // Stop profiling for the current choice
            PERF_END(PERFORMANCE_COUNTER_1_BASE, choice);
        }
        handled_snippets = handled_snippets + 1;
    }
    else if(choice == 2)
    {
        // call comparison function for custom instruction
        if(run_profiling == 1)
        {
            // Start profiling for the current choice
            PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE, choice);
        }
        custom_function(snippet_starting_address, snippet_storage_address, snippet_word_count, o
```

```

    if(run_profiling == 1)
    {
        // Stop profiling for the current choice
        PERF_END(PERFORMANCE_COUNTER_1_BASE, choice);
    }
    handled_snippets = handled_snippets + 1;
}
else if(choice == 3)
{
    // call comparison function for C code implementation
    if(run_profiling == 1)
    {
        // Start profiling for the current choice
        PERF_BEGIN(PERFORMANCE_COUNTER_1_BASE, choice);
    }
    c_function(snippet_starting_address, snippet_storage_address, snippet_word_count, op_type);
    if(run_profiling == 1)
    {
        // Stop profiling for the current choice
        PERF_END(PERFORMANCE_COUNTER_1_BASE, choice);
    }
    handled_snippets = handled_snippets + 1;
}
}
if(run_profiling != 1)
{
    alt_printf("All snippets handled. \nSending confirmation mail with start_storage_address = ");
}
send_mail(start_storage_address, memory_size);
}

// ----- Main function -----

int main()
{
    alt_printf("\n \n \n \n ----- main ----- \n");

    // Wait for mail from the Audio subsystem, with task info
    while(1)
    {
        // Use mailbox to start tasks on this subsystem
        receive_mail();
        choice = (int)mail_receive[0];
        if(choice != 0)
        {
            choose_task(choice);
            choice = 0x0;
        }
    };
    return 0;
}

```