Pascal Lüdi, Daniel Hobi

Audio Playback Tasks for RHWOS

Student Thesis SA-2004-12
Winter Term 2003/2004

Tutor: Herbert Walder

Supervisor:
Prof. Dr. L. Thiele

6.2.2004
# Contents

1 Introduction ........................................... 1  
  1.1 XF-Board ........................................... 1  
  1.2 Reconfigurable Hardware Operating System ............... 2  
  1.3 Audio tasks for RHWOS ................................ 2  

2 Audio Driver ........................................... 5  
  2.1 Target Platform ...................................... 5  
  2.2 Purpose ............................................. 5  
  2.3 Requirements ....................................... 5  
  2.4 Interface ........................................... 6  
    2.4.1 Clock and Reset Signal ......................... 6  
    2.4.2 Data Signals .................................. 6  
    2.4.3 Control Signals ................................ 7  
    2.4.4 Chip Signals .................................. 9  
  2.5 Design .............................................. 9  
  2.6 Problems and Bugs .................................. 9  
    2.6.1 Audio Chip Reset ................................ 9  
    2.6.2 Audio Chip Placement ......................... 9  
    2.6.3 ISE Projnav Timing Analysis .................... 10  

3 MP3 Decoder ........................................... 11  
  3.1 Target Platform .................................... 11  
  3.2 Evaluation of Libraries .............................. 11  
  3.3 Porting to MicroBlaze ................................ 12  
  3.4 Bootloader and Compiling ............................ 13  
  3.5 Loading the MP3 Decoder ............................. 14  
  3.6 Starting the MP3 Decoder ............................ 15  
  3.7 Performance ....................................... 16  
  3.8 Problems and Bugs .................................. 17  
    3.8.1 XPS Sub-module Bug ............................. 17  
    3.8.2 Static MAC Address ............................. 17  
    3.8.3 SRAM Data Width ............................... 18  

4 ADPCM Decoder ........................................... 19  
  4.1 PCM Playback Task .................................. 19  
  4.2 ADPCM Algorithm ................................... 19
4.3 Implementation ............................................. 19
4.4 ADPCM Mono Decoder Task ............................... 20
  4.4.1 Function .................................................. 20
  4.4.2 Interfaces ............................................... 20
  4.4.3 Requirements ......................................... 21
4.5 ADPCM Stereo Decoder Task .............................. 22
  4.5.1 Function .................................................. 22
  4.5.2 Interfaces ............................................... 22
  4.5.3 Requirements ......................................... 22
4.6 Software ...................................................... 23

5 Outlook and Acknowledgements ............................ 25
  5.1 Further work and improvements ......................... 25
    5.1.1 MP3 Decoder ......................................... 25
    5.1.2 Bootloader ........................................... 25
  5.2 Acknowledgements ........................................ 25

Bibliography ................................................. 27
Chapter 1

Introduction

1.1 XF-Board

The usage of configurable components as FPGAs or CPLDs in embedded systems not only offers the chance for rapid prototyping, but it makes it possible to execute independent applications in sequential order on the same device to save precious hardware resources.

The XF-Board[1] was especially developed to study the advantages of this approach. Two FPGAs by Xilinx are the main part of the board. While the bigger R-FPGA is used to execute the application tasks, there is a MicroBlaze Soft-CPU instantiated on the C-FPGA, primarily doing the scheduling of the tasks and the dynamic reconfiguration of the R-FPGA.

![Figure 1.1: Schematic of the XF-Board](image)

Figure 1.1: Schematic of the XF-Board
1.2 Reconfigurable Hardware Operating System

The operating system to manage the reconfigurable devices on the XF-Board is currently being implemented as work of diploma theses [2], [3].

Equally to the prototype operating system [5] for the previous hardware platform, there will be several task slots on the R-FPGA, each able to store and execute an application task. As the FPGAs are partially reconfigurable at run-time, it is possible to exchange a single task while keeping the other applications running.

1.3 Audio tasks for RHWOS

Issue of this student thesis was to evaluate and implement audio tasks for the RHWOS running on the XF-Board.

First of all we had to write an Audio Driver for the operating system, so that the application tasks can communicate with the audio codec chip on the XF-Board.

Our primary objective was to decode MP3 in real-time. We started with a pure software implementation for the MicroBlaze Soft-CPU, but unfortunately the computing power of the system did not suffice to decode MP3 streams with bit rates higher than 24 kbps. We had the choice whether to implement the computationally expensive parts or even the whole appli-
cation directly in hardware, or to concentrate on alternative audio tasks. As the MP3 algorithm is very hard to be mapped on hardware due to its irregular control flow and as it was clear that the decoder would not fit into a single task slot, we decided to look for better suited audio applications.

After all we created a PCM task which simply forwards PCM data to the Audio Driver and two versions of an ADPCM decoder.
Chapter 2

Audio Driver

In this chapter you will find details about the Audio Driver and its implementation. First, the Target Hardware and Purpose are evaluated, then detailed information about Properties and the Interface follows. The last sections concern the Design and Problems of the Audio Driver.

2.1 Target Platform

The Audio Codec on the target platform is a AK4563A[7] chip by AKM. It is a 16 bit Codec with two analog stereo inputs and one analog stereo output. The AK4563A also includes a control interface which allows to set various parameters such as Input Programmable Gain Amplifier (IPGA) and Power Management settings. Another feature is the built-in Peak Meter.

2.2 Purpose

The purpose of the Audio Hardware Driver is to provide an interface between the Audio Codec chip and applications which want to play or record audio streams. First, it directly controls the hardwired signals between the R-FPGA and the Audio Codec. Second, it has to offer an easy-to-use interface to which applications (in this case, VHDL modules) requiring audio functionality can connect.

2.3 Requirements

The Audio Driver requires a clock rate of 50 Mhz in order to generate to lower clock rates needed by the audio chip. On our target FPGA (XC2V3000 Virtex-II), device utilization was quite low. On this particular device, the Audio Driver requires 86 out of 14336 Slices, 80 out of 28672 flip flops, 1 out of 12 DCMs and 11 IOBs which correspond to the connections to the audio chip.

5
2.4 Interface

The interface signals can be roughly divided in four categories:

- **Clock and Reset Signals**

- **Data Signals** are used to exchange audio sample data

- **Control Signals** are used to change driver settings and get status information

- **Chip Signals** are connected to the Audio Codec Chip pins

The following sections describe these signals in detail.

2.4.1 Clock and Reset Signal

\[
\text{CLKINxCI} : \text{in std_logic};
\]

The Audio Driver requires an input clock frequency of 50 MHz in order to operate correctly.

\[
\text{RSTxRI} : \text{in std_logic};
\]

The reset input is high active. Please note: It is important to reset the Audio Driver at least once to ensure proper functionality.

2.4.2 Data Signals

The main purpose of the data signals is connecting one or more synchronous, 16 bit wide fifo queues. The queues can either be generated automatically by the CORE Generator which is part of the Xilinx Integrated Software Environment (ISE) or a custom design can be used. All data signals are MSB-first (most significant bit first), 2’s complement format.

\[
\text{Play FIFO Interface}
\]

- PlayReadEnablexSO : \text{out std_logic};
- PlayReadDataxDI : \text{in std_logic_vector(15 downto 0)};
- PlayEmptyxSI : \text{in std_logic};

These signals are used to connect a fifo queue which contains audio samples to be played. The strobe output \text{PlayReadEnablexSO} indicates a request for the next fifo entry at the data input \text{PlayReadDataxDI}. Each data frame is expected to be valid at the latest approximately 500 clock cycles after the strobe signal was high, and is required to be stable until the next strobe. \text{PlayEmptyxSI} input is used to indicate the lack of data in which case the Audio Driver interrupts playing.
2.4 Interface

−− Record (Mic1/Line Input) FIFO Interface
Record0WriteEnablexSO : out std_logic;
Record0WriteDataxDO : out std_logic_vector(15 downto 0);

−− Record (Mic2 Input) FIFO Interface
Record1WriteEnablexSO : out std_logic;
Record1WriteDataxDO : out std_logic_vector(15 downto 0)

If the Audio Driver is used to record audio data on one or both channels, the appropriate fifo queues are connected using these signals. Output values on Record0WriteDataxDO and Record1WriteDataxDO are valid as soon as Record0WriteEnablexSO and Record1WriteEnablexSO respectively are going high. They should be latched not later than after approximately 500 clock cycles.

2.4.3 Control Signals

MutexSI : in std_logic; −− Mute (0: off / 1: on)

This binary input is used to mute and unmute the player part of the Audio Driver (data is still read in at the regular rate).

PlayStereoxSI : in std_logic; −− Stereo Mode (0: mono / 1: stereo)
Record0StereoxSI : in std_logic;
Record1StereoxSI : in std_logic;

These signals control whether the Audio Codec is operating in mono or stereo mode. It is possible to configure the player part as well as both recording parts independently.

TestTonexSI : in std_logic; −− Play Test Tone (0: off / 1: on)

This allows a basic function test by generating a simple test tone.

InputSelectxSI : in std_logic; −− (0: Mic inputs / 1: Line input)
InputLevelxDI : in std_logic_vector(6 downto 0);
― IPGA Level (range 00H – 60H, 60H = 1100000b)
― (00H = mute, 28H = 0101000b = default)

These signals are used to configure both analog inputs of the Audio Codec. InputSelectxSI being set to 0 means that both analog inputs are active and a sensitive IPGA gain table is used. This mode is intended if one or two microphones are connected to the audio chip. Otherwise (InputSelectxSI set to 1) only the first input channel is active and a less sensitive IPGA gain table is used. This is the appropriate setting if a low impedance input is being used such as the output of a PC sound card. If you intend to use the first input channel while InputSelectxSI is set to 0, please have a look at section 2.6.2

InputLevelxDI allows a fine-grained setting of the input sensitivity of both audio input channels as well as both stereo channels if stereo mode is
used. The input gain set by InputLevelxDI depends on the setting of Input-SelectxSI as described above. The default value of InputLevelxDI should be “0101000b” in binary format. Setting this value to “0000000b” allows muting of both input channels. Others values of both input gain tables can be found at table 2.1. Please note that the maximum value is 1100000b, thus not the entire possible range of InputLevelxDI is valid.

<table>
<thead>
<tr>
<th>DATA</th>
<th>GAIN(dB)</th>
<th>Step</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIC</td>
<td>LINE</td>
<td></td>
</tr>
<tr>
<td>60H</td>
<td>+28.0</td>
<td>+6.0</td>
<td></td>
</tr>
<tr>
<td>5FH</td>
<td>+27.5</td>
<td>+5.5</td>
<td></td>
</tr>
<tr>
<td>5EH</td>
<td>+27.0</td>
<td>+5.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5dB</td>
<td></td>
<td>73</td>
</tr>
<tr>
<td>28H</td>
<td>+0.0</td>
<td>-22.0</td>
<td></td>
</tr>
<tr>
<td>27H</td>
<td>-0.5</td>
<td>-22.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19H</td>
<td>-7.5</td>
<td>-29.5</td>
</tr>
<tr>
<td></td>
<td>18H</td>
<td>-8.0</td>
<td>-30.0</td>
</tr>
<tr>
<td></td>
<td>17H</td>
<td>-9.0</td>
<td>-31.0</td>
</tr>
<tr>
<td></td>
<td>16H</td>
<td>-10.0</td>
<td>-32.0</td>
</tr>
<tr>
<td></td>
<td>11H</td>
<td>-15.0</td>
<td>-37.0</td>
</tr>
<tr>
<td></td>
<td>10H</td>
<td>-16.0</td>
<td>-38.0</td>
</tr>
<tr>
<td></td>
<td>0FH</td>
<td>-18.0</td>
<td>-40.0</td>
</tr>
<tr>
<td></td>
<td>0EH</td>
<td>-20.0</td>
<td>-42.0</td>
</tr>
<tr>
<td></td>
<td>05H</td>
<td>-38.0</td>
<td>-60.0</td>
</tr>
<tr>
<td></td>
<td>04H</td>
<td>-40.0</td>
<td>-62.0</td>
</tr>
<tr>
<td></td>
<td>03H</td>
<td>-44.0</td>
<td>-66.0</td>
</tr>
<tr>
<td></td>
<td>02H</td>
<td>-48.0</td>
<td>-70.0</td>
</tr>
<tr>
<td></td>
<td>01H</td>
<td>-52.0</td>
<td>-74.0</td>
</tr>
<tr>
<td>00H</td>
<td>MUTE</td>
<td>MUTE</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Input Gain Setting

PeakLeftxDO : out std_logic_vector(7 downto 0); -- Peak Level Left Channel (Mic1/Line Input)
PeakRightxDO : out std_logic_vector(7 downto 0); -- Peak Level Right Channel

PeakLeftxDO and PeakRightxDO are used to read out the on-chip peak meter. There is only one peak meter available, which is bound to the first analog audio channel (Microphone 1 or Line input). Although each stereo channel can be accessed separately. The peak meter outputs are a great method to get information about the audio signal without reading out data samples. These outputs are updated every 2273 clock cycles.
2.4.4 Chip Signals

MCkCO : out std_logic;      -- audio master clock
BCkCO : out std_logic;      -- audio bit clock
LRCkCO : out std_logic;     -- audio left/right clock
CCkCO : out std_logic;      -- audio control clock

SxDI : in std_logic_vector(1 downto 0);  -- audio data in
SxDO : out std_logic;        -- audio data out
CxDI : in std_logic;         -- control data in
CxDO : out std_logic;        -- control data out

PDnxSO : out std_logic;     -- power down & reset
CSnxSO : out std_logic;     -- chip select

These signals have to be connected to the corresponding IO pins of the R-FPGA. This is usually done by using a .ucf file (User Constraint File) which can be found on the included CD-ROM. For a detailed description of each Chip signal have a look at [7].

2.5 Design

The top level design is named audiodrv and its components can be classified in three categories. First, there are components responsible for transferring data in and out, these are readplayfifo, writerec0fifo, writerec1fifo, output, input0 and input1. Then there is a controller that changes settings of the Audio Codec according to the interface control signals described in section 2.4.3. The third group of components is dividing the input clock (50Mhz) to several slower clock signals required by the audio chip. The main VHDL processes regarding this task are clock_gen and the DCM (Digital Clock Manager) instance dcm_audio.

2.6 Problems and Bugs

2.6.1 Audio Chip Reset

One has to reset the audio chip at least once to ensure proper operation. This can be done by resetting the Audio Driver (input RSTxRI, see section 2.4.1) prior to audio playback or recording.

2.6.2 Audio Chip Placement

According to schematics in [1], the audio chip pins INTL0 and INTR0, which correspond to the microphone input of the first analog input channel, are not connected to the Audio In 0 jack. This prevents the use of both input channels at the same time, as when the input mode is set to microphone usage (ie. InputSelectxSI is set 0, see section 2.4.3), only the second microphone input is operating correctly.

A possible solution to this problem is to short-circuit the audio chip pins INTL0 with LIN as well as INTR0 with RIN.
2.6.3 ISE Projnav Timing Analysis

Running a timing analysis within Xilinx ISE Project Navigator 6.1.02i reports delay paths of more than 40µs when requiring a clock period of 20ns. This problem is caused by a feature of the timing analysis tool, which is converting a clock period constraint on DCM input clock domains to an additional clock period constraint on the output clock domain. Whenever this resulting constraint expressed in picoseconds is not an integer number, the timing analysis will fail.

This is known to Xilinx and may be fixed in a future version of ISE. A workaround is to specify the clock period constraint in a way that the resulting (automatically generated) constraint expressed in picoseconds is an integer number too. An example constraint for the Audio Driver would be 19.998ns instead of 20ns.
Chapter 3

MP3 Decoder

This chapter presents the steps in designing and implementing the MP3 decoder. First, we explain the evaluation of different MP3 libraries and porting the best suited one to our system. Then, a description of compiling and loading the MP3 decoder follows. At the end of this chapter, some remarks about performance and problems of the decoder are due.

3.1 Target Platform

The target platform which we designed and implemented the MP3 decoder for is a Xilinx XC2V3000 Virtex-II FPGA being part of the XFBoard[1]. The FPGA configuration contains a MicroBlaze Soft CPU[9] by Xilinx and our self-made Audio Driver (see chapter 2). The MicroBlaze CPU is part of the Embedded Development Kit (EDK) which also contains several compiling and linking utilities based on GCC[10].

3.2 Evaluation of Libraries

As a first step, we evaluated different MP3 decoding libraries which are available on the Internet. A quite extensive comparison of MP3 decoders can be found at [8]. Aside from several decoders out of this list we have also considered the official ISO reference decoder. Table 3.1 gives an overview of the evaluated decoders which are all freely available on the Internet.

We then decided to use libmad as it is the only decoder that relies exclusively on integer operations. Since the MicroBlaze Soft CPU does not have a floating point ALU, such operations are executed as software functions which is multiple times slower than integer operations of the same complexity. Thus libmad is a suited library for porting to the MicroBlaze processor. In addition, it does also support every official MPEG audio layer standard (MPEG 1/2 layer 1 to 3) as well as the unofficial MPEG 2.5 standard.
Table 3.1: Evaluated MP3 Decoders

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO 13818-3.2 reference decoder[11]</td>
<td>+ 100% compliant to the ISO standards</td>
</tr>
<tr>
<td></td>
<td>− outdated, hard to read, slow</td>
</tr>
<tr>
<td>lame 3.93.1[12]</td>
<td>+ wide spread encoder</td>
</tr>
<tr>
<td></td>
<td>− relies on floating point operations</td>
</tr>
<tr>
<td>mpg123 0.59r[13]</td>
<td>+ decoding engine used by lame</td>
</tr>
<tr>
<td></td>
<td>+ fast, heavily optimized</td>
</tr>
<tr>
<td></td>
<td>− relies on floating point operations</td>
</tr>
<tr>
<td>libmad 0.15.0b[14]</td>
<td>+ fixed-point (integer) computation only</td>
</tr>
<tr>
<td></td>
<td>+ high accuracy (24-bit PCM output possible)</td>
</tr>
<tr>
<td></td>
<td>+ easy to read, good code structure</td>
</tr>
</tbody>
</table>

3.3 Porting to MicroBlaze

The next step was porting the chosen C code to the MicroBlaze system. The main challenge at this point was the relatively weak C libraries available for the MicroBlaze and the amount of Block RAM (BRAM) memory on the FPGA.

Although the libraries provided by Xilinx for the MicroBlaze system contain all functions defined by ANSI C, not all of them are implemented in a reasonable way. Especially the dynamic memory management functions such as `malloc()` and `free()` are not suitable for porting libmad. As these functions allocate memory but do not release it (`free()` does nothing), the system is running out of memory after a short while. Therefore the main task in porting libmad to MicroBlaze was changing the memory management such that it uses only static allocation.

After the changes to the memory allocating and releasing parts of libmad, we had a rough overview about the memory demands of the MP3 decoder. That amount was approximately 200 kB without considering buffer space for the MP3 data itself. As this is three times the internal memory available to the MicroBlaze processor on the chosen FPGA, we either had to decrease the memory requirement or to swap out instruction and data parts to external SRAM or SDRAM memory.

Even after removing the entire MPEG layer 1 and 2 decoding functionality the remaining layer 3 (MP3) decoding engine would have needed more than 64 kB memory. Thus the only option left was utilizing additional memory. Fortunately, there was already a so-called Bootloader present, which grants the possibility to first configure the FPGA and then load additional data into the external SRAM banks. This Bootloader was originally written by Samuel Nobs during his diploma thesis[2]. To fit our need we had to modify his Bootloader which is explained in the next section.
3.4 Bootloader and Compiling

The underlying concept of the Bootloader is to first configure the FPGA with the MicroBlaze system and a small set of instructions. As these instructions get executed, they load additional instruction and data segments from a peripheral device (like a PC) into SRAM or SDRAM banks. These external memory banks have a huge capacity compared to the FPGA internal Block RAM cells.

![Sectional layout of an object or executable file](image)

As you can see in figure 3.1, a regular executable consists of various memory sections. With the help of a Bootloader, we can now not only store parts of the `.text` section (instructions) in the SRAM, we can also swap out parts of the other sections. Although there is a restriction regarding the data types of swapped out sections. Due to the hardware structure of the XFBoard[1], only data types with a size of a multiply of 2 may be written to SRAM (see section 3.8.3 for details).

Because of this restriction, not every section of the MP3 decoder can be stored in SRAM, which would not be a good idea anyway, since access to external SRAM memory is about three times slower than accessing internal Block RAM memory. Thus, a partitioning is necessary to assign each part of every section a memory location.

Figure 3.2 shows the final memory assignment. The left side corresponds to sections stored in the internal Block RAM memory while the right side corresponds to section loaded into the external SRAM memory.

The `.text` section in BRAM contains the Bootloader instructions plus its required libraries. The same applies for read-only data in `rodata`. The remaining data section in BRAM consist of both Bootloader and decoder
Figure 3.2: Modified sectional layout

On the righthand side, there are \texttt{.opb_*} sections loaded into SRAM. The following data is assigned to these sections: decoder instructions (\texttt{.opb\_text}), read-only tables required by various decoding functions (\texttt{.opb\_rodata}), read-write data (\texttt{.opb\_data} and \texttt{.opb\_bss}) and a buffer for incoming MP3 data (\texttt{.opb\_buffer}).

This partitioning can be done by providing a special linker script to the compiling tools. Such a linker script can be found on the included CD-ROM. Developers using Xilinx Platform Studio can make use of this file by specifying the script path in \texttt{Options\rightarrow Compiler Options\rightarrow Details\rightarrow Linker} (example entry: \texttt{-T etc/combined\_linker\_script}).

Since the Bootloader is using a standard serial connection to transfer data, one has to convert the executable file (usually named \texttt{executable.elf}) after compiling. The file format used to transfer the SRAM parts of the executable is called srec, a format based on ASCII, human-readable characters only. During the conversion, splitting the SRAM data from the rest is suggested. Both processes are automatically run by using the included Makefile (\texttt{program\_make}). The resulting srec file is then called \texttt{program\_srec.txt}.

### 3.5 Loading the MP3 Decoder

After compiling and splitting the executable, we are ready to load the MP3 decoder. Since the Bootloader is using a serial connection to transfer data, one first has to start a terminal application capable of communicating over
a serial interface. Then, after loading the generated FPGA configuration bitstream (usually `implementation/download.bit`), a Bootloader menu is presented (see figure 3.3). The important options are 0 - load new code and 3 - start execution of code. The next step is loading the SRAM data by selecting 0 - load new code and make your terminal send the srec file as ASCII text. Usually GUI terminals like Hyperterminal by Microsoft offer such an option (in Hyperterminal it is located at Transfer→Send Text File...).

When the srec data has been sent, the loading process is finished and the MP3 decoder can be started by selecting 3 - start execution of code at the Bootloader menu.

![Bootloader Menu](image)

### Figure 3.3: Bootloader Menu

**3.6 Starting the MP3 Decoder**

While running, the decoder requests MP3 data through the Ethernet interface. This is done by sending an UDP data packet containing only the string “NEXT” as UDP payload. This request is sent to IP 192.168.1.1 at port 7648. Additionally, the destination Ethernet MAC address is also static, because we were not able to successfully operate IP address resolving via ARP (see section 3.8.2).
Then, the decoder expects an UDP response packet containing MP3 data only. This simple data exchange protocol is compatible to the *Streaming Server* written by Silvan Wegmann during his diploma thesis[4]. A copy of the server can be found on the included CD-ROM. The handling is straight-forward (see figure 3.4): First, a file to be shared is selected via the *Browse* . . button, then the server may be started by pressing *Start Stream*. Available options are the used packet payload size (*Buffersize:*), the port number on which the server is listening (*Port:*), and the options to loop the selected file until stopped (*Repeat* checkbox).

![Streaming Server](image)

**Figure 3.4: Streaming Server**

### 3.7 Performance

Since the implemented MP3 decoder does support every available MPEG audio layer specification, we were able to test various audio streams of different quality to measure the real-time performance of the MP3 software decoder.

The results are quite modest. Although the network part is able to handle very high quality streams, the MicroBlaze clocked at 50 Mhz is slowing down the decoding process. At least, the system can decode MP3 audio streams with the following properties:

- *MPEG2 layer 3, mono, 22kHz, 8kbps*
3.8 Problems and Bugs

- MPEG2 layer 3, mono, 22kHz, 16kbps
- MPEG2 layer 3, mono, 22kHz, 24kbps (if compiler optimization level 2 is turned on)

By doing a basic profiling, we were able to identify the most computing power demanding functions (see table 3.2). After this analysis, we decided to be content with the MP3 decoder in its current state and not to implement data-flow orientated functions as hardware modules (see section 1.3).

<table>
<thead>
<tr>
<th>function name</th>
<th>processor time used total time used by subfunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>mad_frame_decode</td>
<td>~40%</td>
</tr>
<tr>
<td>-III_huffdecode</td>
<td>~8% (varying)</td>
</tr>
<tr>
<td>-III_reorder</td>
<td>~8%</td>
</tr>
<tr>
<td>-III_imdct</td>
<td>~24%</td>
</tr>
<tr>
<td>synth_full</td>
<td>~30%</td>
</tr>
<tr>
<td>-dct32</td>
<td>~15%</td>
</tr>
<tr>
<td>audio_output</td>
<td>~30%</td>
</tr>
</tbody>
</table>

Table 3.2: Profile results

3.8 Problems and Bugs

3.8.1 XPS Sub-module Bug

In order to design a project where the MicroBlaze system is a sub-module of a higher level design, Xilinx Platform Studio 6.1.02i offers the option to create a template VHDL file of the top-module. This option can be turned on by selecting This is a sub-module in my design in Options→Project Options→Hierarchy and Flow→Design Hierarchy.

The template file (named hdl/system_stub.vhd by default) is incorrect if one specifies any “downto” ports in Project→Add/Edit Cores…→Ports. In particular, this happened when we have defined the range of a port to be [3:0] (contrary to the usual setting [0:3]). In that case, XPS is generating a correct port definition for both the top-module and the sub-module containing the MicroBlaze, but it does not create valid internal signals, which interconnect the sub-module and the top-module.

To be precise, it defines port signals as std_logic_vector(3 downto 0) and corresponding internal signals as std_logic_vector(0 to 3).

3.8.2 Static MAC Address

The underlying network driver used in this project was originally written by Marco Kuster during his student thesis[6]. After upgrading Xilinx Embedded
Development Kit (EDK) from version 3.2 to 6.1, we were facing various problems. Although we were able to solve most of them, the ARP resolving mechanism still does not operate correctly.

Therefore, not only the Streaming Server’s IP address is static, but also its MAC address.

3.8.3 SRAM Data Width

As the C-FPGA on the XF-Board[1] shares four Write Enable signals with the connected SRAM chips, it is possible to write data at a granularity of 8 bits. On the other hand, only two Write Enable signals are available to the R-FPGA, allowing a granularity of 16 bits.

Hence, the MicroBlaze system (running on the R-FPGA) cannot write single byte data types to SRAM addresses, while it can perfectly read single byte types from SRAM. Because of this restriction, we have swapped out only integer data types to read-write section in SRAM (see section 3.4).
Chapter 4

ADPCM Decoder

In this chapter we describe the audio tasks that will be adapted to run on the RHWOS. As the operating system is still in a development stage, the task environment is emulated using a MicroBlaze Soft CPU and IP Cores for the Ethernet receiver and the input FIFO.

We started with a simplistic PCM playback task and implemented an ADPCM decoder for compressed audio streams in mono and stereo quality.

4.1 PCM Playback Task

The PCM Playback Task gets a 16-bit PCM data stream via Ethernet and forwards it to the Audio Driver. It is designed to play stereo files only, but can easily be changed to play mono files.

This task was mainly written to verify the emulation environment.

4.2 ADPCM Algorithm

The Adaptive Differential Pulse Code Modulation is a rather simple algorithm to compress 16-bit PCM data down to 3, 4 or 5 bit.

ADPCM does not encode the sound sample itself but its difference to the following sample. At high sample frequencies this differences become small, allowing to encode them with less than 16 bits without a severe loss of quality. Adaptive means that the sample differences are encoded by using index table functions whose input parameters are adapted to the current sample difference.

As the algorithm is not free of quality losses, it slightly decreases the SNR by introducing some quantization noise.

4.3 Implementation

We decided to implement a 4-bit version of ADPCM which reaches a stable compression rate of 4:1 compared to PCM. The input data is expected to be sampled with 44.1 kHz. Due to the high sample rate the quality loss of the decoder is guaranteed to remain small.
The VHDL code of the decoder is derived from the C source code of an IMA ADPCM decoder written by Jack Jansen[15]

### 4.4 ADPCM Mono Decoder Task

#### 4.4.1 Function

The ADPCM Mono Decoder Task gets a 4-bit encoded stream from the Streaming Server application by Silvan Wegmann[4] via the Ethernet receiver and pipes the decoded 16-bit PCM output to the Audio Driver.

If the input buffer is signalling that it is not empty, the decoder reads in a sample from the input queue. At the moment this is a generic 16-bit FIFO to adapt to the task emulation, but actually only the 4 lowest bits are processed. The ADPCM sample is decoded in six sequential steps and the output is written to the output buffer if it is not full, else the decoder is waiting until there is enough space in the output queue. Afterwards it jumps to idle state if there is no further data to process or it starts reading the next sample if there are any. See the state diagram in fig. 4.1.

#### 4.4.2 Interfaces

**Clock and Reset signal**

\[ CLKxCI : \text{in std\_logic}; \]

\[ CLKxCI \] is the clock signal of the ADPCM decoder.

\[ RSTxRI : \text{in std\_logic}; \]

\[ RSTxRI \] is the active high reset signal. It leads the decoder to its idle state, if the signal is set to logic ‘1’.

**Control signals**

\[ \text{InputReadEnablexSO} : \text{out std\_logic}; \]

\[ \text{InputReadEnablexSO} \] is set to logic ‘1’ if the decoder wants to read from the input queue. The actual reading operation happens in the next cycle after the setting of the read enable signal.

\[ \text{InputEmptyxSI} : \text{in std\_logic}; \]

\[ \text{InputEmptyxSI} \] has to be logic ‘1’ if the input queue is empty and logic ‘0’ otherwise. The decoder checks the input empty signal to decide whether to leave its idle state or not.

\[ \text{OutputWriteEnablexSO} : \text{out std\_logic}; \]

\[ \text{OutputWriteEnablexSO} \] is set to logic ‘1’ if the decoder wants to write the current value of \[ \text{OutputWriteDataxDO} \] to the output queue. The output FIFO has to transfer the PCM output data not later than 5 cycles after the write enable signal has been set.
4.4 ADPCM Mono Decoder Task

OutputFullxSI : in std_logic;

*OutputFullxSI* has to be logic ‘1’ if the output queue is full and logic ‘0’ otherwise. The decoder waits for this signal to become logic ‘0’ until it proceeds with the next input sample.

**Data signals**

InputReadDataxDI : in std_logic_vector(15 downto 0);

*InputReadDataxDI* holds the ADPCM input data to be processed next. Only the 4 lowest bits are processed, the rest of the signal is not regarded.

OutputWriteDataxDO : out std_logic_vector(15 downto 0);

*OutputWriteDataxDO* holds the 16-bit PCM output signal and is valid from the setting of the output write enable for at least 5 clock cycles.

### 4.4.3 Requirements

The ADPCM Mono Decoder can be run with a clock rate ranging from about 360 kHz up to 150 MHz.

There is a need for 225 out of 14336 slices and 121 out of 28672 flip flops on the target device (*Xilinx Virtex-II XC2V3000*) for the decoder only, respectively 443 slices and 331 flip flops for the whole task emulation.

There are no hard requirements for the input queue. It is no problem if the FIFO is not quick enough to present the data at the next cycle after the request as it is expected by the decoder. The input data is buffered, so if the FIFO latency is longer than one cycle, the decoder is always one step in behind but still working correctly.

![State diagram of ADPCM Mono Decoder](image)

**Figure 4.1:** State diagram of ADPCM Mono Decoder
4.5 ADPCM Stereo Decoder Task

4.5.1 Function

The ADPCM Stereo Decoder Task gets 8-bit encoded stereo samples via the Ethernet receiver and produces two 16-bit PCM samples for the Audio Driver.

The stereo decoder works similarly to the mono decoder. There is also a generic 16-bit input FIFO, but now the lowest 8 bits are used. The ADPCM sample for the left and the right channel are read in simultaneously. The decoder first processes the sample for one channel and writes the respective PCM data to the output FIFO, afterwards it handles the other channel in the same manner. New data is read in after every second pass only. See the state diagram in fig. 4.2.

4.5.2 Interfaces

The interfaces are identical to the interfaces of the ADPCM Mono Decoder.

4.5.3 Requirements

The ADPCM Stereo Decoder can be run with a clock rate ranging from about 670 kHz up to 150 MHz.

There is a need for 249 out of 14336 slices and 179 out of 28672 flip flops on the target device for the decoder only, respectively 462 slices and 389 flip flops for the whole task emulation.

The requirements for the input FIFO are the same as that for the mono decoder.

![Figure 4.2: State diagram of ADPCM Stereo Decoder](image-url)
4.6 Software

Based on the C source code by Jack Jansen\cite{15} we built some software tools to convert sound files on a personal computer. The source code and the executables of the tools can be found on the CD-ROM.

    adpcm_encoder.exe is a simple console application to encode a PCM input file to an ADPCM output file. If the input file is in stereo PCM format, the encoder must be invoked with the `-stereo` option and will produce an ADPCM stereo file. Otherwise the input is considered to be in mono format and the output also will be a mono file.

usage: adpcm_encoder \textit{inputfile} \textit{outputfile} [\textit{-stereo}]

    stereo2mono.exe extracts one channel of an ADPCM stereo file. The channel can be chosen by invoking the program with the `-left` or `-right` option.

usage: stereo2mono \textit{inputfile} \textit{outputfile} [-\textit{left} | -\textit{right}]

    adpcm_decoder.exe is an ADPCM to PCM converter using the same algorithm as the hardware decoder.
Chapter 5

Outlook and
Acknowledgements

In this short chapter we give some hints how the audio playback tasks could be refined and want to thank the people who supported us in our work.

5.1 Further work and improvements

5.1.1 MP3 Decoder

The computationally expensive parts of the MP3 source code could be implemented directly in hardware, creating some sort of co-processors for the MicroBlaze. Especially the transformation functions as the Discrete Cosine Transformation (DCT) or the Inverse Modified DCT are candidates to get accelerated. This would allow to decode streams with higher bit rates, anyway the task still wouldn’t fit into a task slot.

One could also map the whole decoder into hardware, but this is hard work as the MP3 decoder is dominated by irregular control flow.

5.1.2 Bootloader

The Bootloader could use the Ethernet interface to transmit program files instead of sending via the slow serial port.

5.2 Acknowledgements

We want to thank our advisor Herbert Walder for introducing us to the field of reconfigurable computing and supporting us in our work. We also want to thank the other students in G69 for sharing their knowledge and helping us with minor problems.

Last but not least we want to thank Prof. Dr. L. Thiele and the Computer Engineering and Networks Laboratory for allowing this student thesis.
Bibliography


http://www.mp3-tech.org/programmer/sources/dist10.tgz

http://lame.sourceforge.net/

http://www.mpg123.de/

http://www.underbit.com/products/mad/


Please note that the URLs noted below the reference entries have been valid at the time writing this document. They may be outdated in the meantime and therefore point to nowhere.