Human Seeing Aid Development

Peter Oakham & Simon So

April 12, 2004

Digital Hardware - ECE532
Division of Engineering Science
University of Toronto

Contents

1 Preface **2**

1.1 Background 2

1.2 Running Your Application 2

1.3 Hardware Cores 4

2 Introduction **5**

2.1 Goals 5

2.2 Design Components 5

2.3 The Moving Target 7

3 The Audio Component **7**

3.1 AC97 core 7

3.2 LM4549 9

3.3 AC97 Core 11

3.4 AC97 Controller 12

3.5 Tone Generator 12

4 The Video Component **12**

4.1 Implementing Video Fetch and Store 13

4.2 YCrCb to RGB0 13

4.3 Single Frame Variation 14

4.4 Driver Code 14

5 Video to Audio Processing **14**

5.1 C Implementation - Colour Averages . . . 14
5.2 Core Ideas . . . 15
5.2.1 Colour Amplitude Frequency Matching . . . 15
5.2.2 Fourier Domain Analysis . . . 15
5.2.3 Stereo Fourier Signal . . . 15
5.2.4 Beyond . . . 16

6 The Design Trees **16**

6.1 Audio . . . 16
6.2 Video . . . 16

7 Outcome **16**

8 Conclusion **17**

A Additional Video Information - Chipsets **18**

B VHDL Code **18**

B.1 Video Component . . . 18
B.2 Reference Video Code . . . 25
B.3 Audio Code . . . 25

C C code **60**

D Simulations **63**

E Specifications Sheets **63**

1 Preface

Our project utilizes the sound and video hardware of the Xilinx MicroBlaze^{TM} processor version 1.0 and Multimedia board. This report will describe first our project overview, the detail description of each component and finally our accomplishments. But before we get into the details, well give some background information of the system we are using

1.1 Background

The MicroBlaze^{TM} processor is implemented onto the chipset Xilinx Virtex II. Firstly, we like to explain some of the basic terminology that are used throughout the report to avoid confusion

- Physical Device the ASIC or FPGA that is off-chip on the Multimedia board, i.e., the external memory, RS232
- Hardware a component that is on-chip, there are default components and user defined components
- Software C program that is written, compiled, run by the processor.

The MicroBlaze^{TM} processor is what is known as a soft processor: meaning that functionality can be added and deleted. The flexibility of having a soft processor is that you can build your system based on your specific applications, thus no extra real estate is wasted because you dont need that feature. What does it mean by functionality can changed? A processor has certain functions, i.e., it may have a UART which communicates with the serial hardware RS232[1] , if one doesnt need to use the serial port, then he/she can choose not to incorporate the UART into the processor. There are many components that can be incorporate into the processor depending on ones need, and these components are known as cores.

Note that the addition or subtraction of functionality can be changed statically. The cores inside the processor is determined and programmed into the FPGA. Once the FPGA is programmed, it will contain those specific cores. During operation, these cores cannot be changed. Think of cores as constants in a C program, you can change it before complication, but once its compiled, they cannot be changed during run-time.

1.2 Running Your Application

There are two ways you can implement a designs, in software or in hardware. To implement a design in hardware required the knowledge of VHDL or Verilog, currently, Xilinx provides support

[1]RS232 an IC (hardware) that communicates with the serial port of the computer

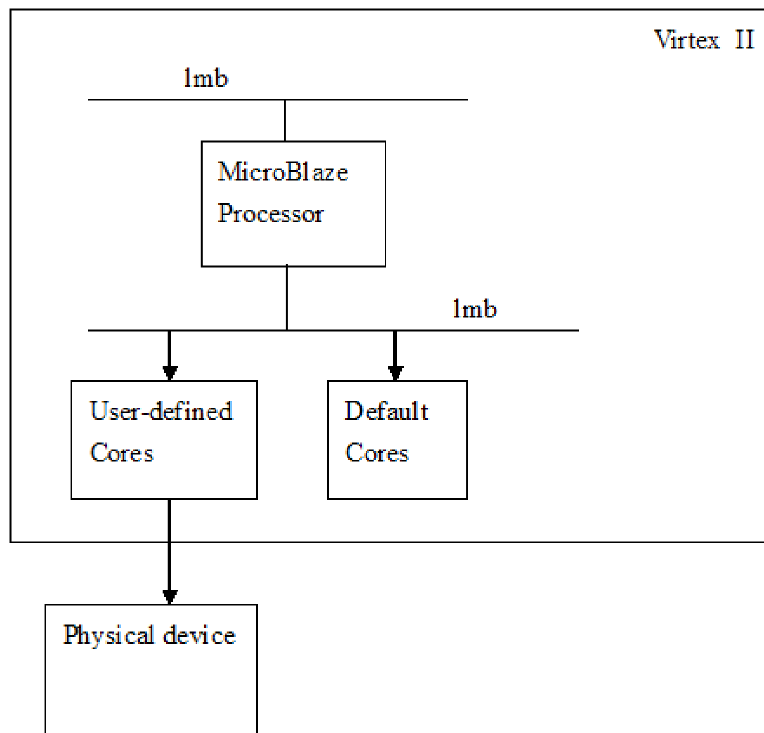


Figure 1: diagram to show MicroBlaze processor and H/W, like the one on the mid-term

for VHDL for us, so we recommend you to use VHDL. Building designs in hardware has advantage of faster data processing, and there are certain things that must be design in hardware. Let me give you some examples, say you would like to implement a multiplier[2], you can create a hardware multiplier using VHDL as you would have done in your Digital course. Then you will incorporate this multiplier as a core in your processor. Now you have a multiplier that will do multiplication in real-time[3]. An example of something needs to be hardware would be a UART, because the UART needs to communicate with the RS232. And so it needs to process data in real-time and in synchronous with the clock; hence it must be implement in hardware.

Software implementation of a design refers to assembly codes. You dont have to write assemble code to get assembly, the Xilinx tools provides a C compiler which generates the assembly code. And so if you were to create a multiplier for instance like our previous example, you could write the code as follows:

```
int a, b, c;

int main() {
c = a * b; // multiply a and b
return c;
}
```

It is important to understand, although it is much easier to implement multiplication function in software, the multiplication does not perform as quickly as it would have in hardware. Although multiplication is relatively simple operation and the difference is not that noticeable, operations such as FFT is more advantageous done in hardware than software.

1.3 Hardware Cores

Building cores is one of the most challenging and pain-stacking tasks of a design. Fortunately, Xilinx has many standard components that can aid your designs, for example, from simple flip flops and shift registers to complex multipliers and FIFOs. If you have the Xilinx ISE installed, you will find documentation about the standard library in following directory:

```
<Drive:>\<installed folder name>\doc\usenglish\books\docs\lib\
```

There are also other intellectual properties provided by Xilinx designer which you can download and use. You can find them in www.xilinx.com, in the section Product and Services, the link IP

[2]A multiplier core is a basic primitive and can be implement by CORE Generator

[3]Real-time refers to process data as soon as becomes available. For instance, a AND gate or a flip-flop process data in real-time, as soon as two inputs are change, the output changes or the clock changes, the output changes respectively. Software on the other hand does not process data in real-time, each operation takes certain amount of clock cycle to process.

Center under the heading Design Resources. These are usually more complex cores that provide specific application. I recommend that you look over it before starting any design; you might find cores that exist which can give you what you need.

2 Introduction

2.1 Goals

The objective of our project is to use video as input and convert the signal into an audio signal. The conversion ideally should be done in hardware using a fourier space to smooth out and match different time domains. To implement such a complex function, we decided to create our design in the standard design hierarchy:

- Software - abstract level, written in C, calls on drivers, and process audio to video
- Driver - written in C, provides an interface to the hardware VHDL cores
- Hardware - written in VHDL, provides access to external chipsets and process data

The cores implemented are the video core, zbt external memory controller, and audio core.

2.2 Design Components

Video Core The video core's purpose is to take in one frame of video from the off-board video decoder, convert it from a YCrCb signal into an RGB signal, and finally store the frame in the ZBT memory. The information from the video input also needs to be separated into pixel data versus spacing and blanking information; only the pixel data is stored. The video data when stored also needs to be easily accessible and readily useful for video processing.

ZBT Core The external memory core interfaces the FPGA to the external memory. The data is received from the video core is written and stored in the external memory. These data will later be processed and convert to an audio signal.

Audio Core The audio core contains two major components, a tone generator and an ac97 core. The tone generator creates tones. A tone is basically a square wave of a particular frequency. In order to play that tone, the signal needs to be sample and convert into a 16-bit value per sample. The 16-bit value is then sent to the ac97 core. The ac97 core deals with the timing and data format that is required to interface the ac97 compliant CODEC used is National Semiconductor LM4549. The CODEC converts the digital signal to analog and outputs to speakers.

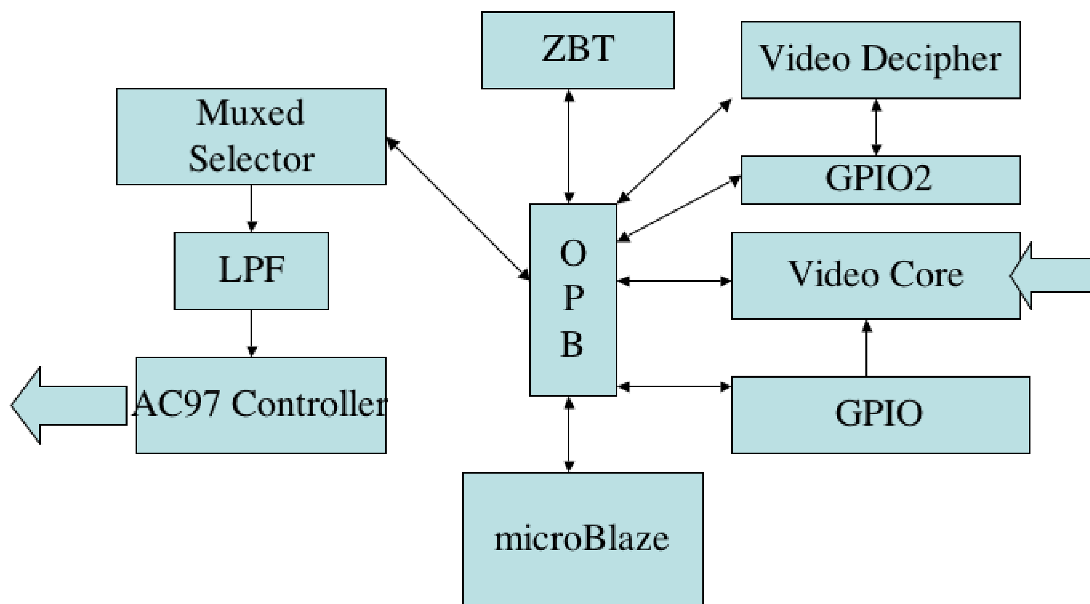


Figure 2: The Ultimate Design Goal's Block Diagram

The core is foundation of the design, though it cannot function by itself. Drivers are written to control functions of the cores. All cores contain a block that interfaces with the OPB bus. And so drivers can give instructions to the cores for certain functions, i.e. initialize the LM4549.

The software layer is the highest of the hierarchy. It uses function provided by the drivers. The layer contains the main of the program, it initialize the video and sound hardware. After that, it takes the video data and processes it into audio data. The value is passed to the audio core and changes the tone and amplitude of the signal.

2.3 The Moving Target

Originally the goal was to have almost all features implemented in hardware. As time drew on however it became increasingly obvious that there was insufficient time to implement everything in Hardware. The end result is that a lot of functions that were schedule to be created in hardware were either moved up to the software level or they were downgraded slightly. The end result is shown in figure 3

A Note About Time Usage Lastly a quick note should be made about effective time usage. In retrospect a lot of the development of this project should have been done as a team instead of different persons pioneering different efforts.

3 The Audio Component

The function of the audio core is to generate different tones and output the sound to speakers or headphones. The audio consist of two parts: a tone generator and a AC97 core. This section describes the detail of the two components.

3.1 AC97 core

The AC97 core deals with the interface from the MicroBlaze to the LM4549 physical device. We first briefly describe the input data and output data format from LM4549.

Four pins are designated to interface witht LM4549, and they are SYNC, BIT_CLK, SDATA_IN, and SDATA_OUT. This is only a quick reference guide to control the LM4549, for further detail, please refer to the LM4549 data sheet in appendix E. We will denote our terminology the same as the data sheet. SDATA_OUT is referred to signal output from the controller and input to the LM4549 as drawn in the figure below.

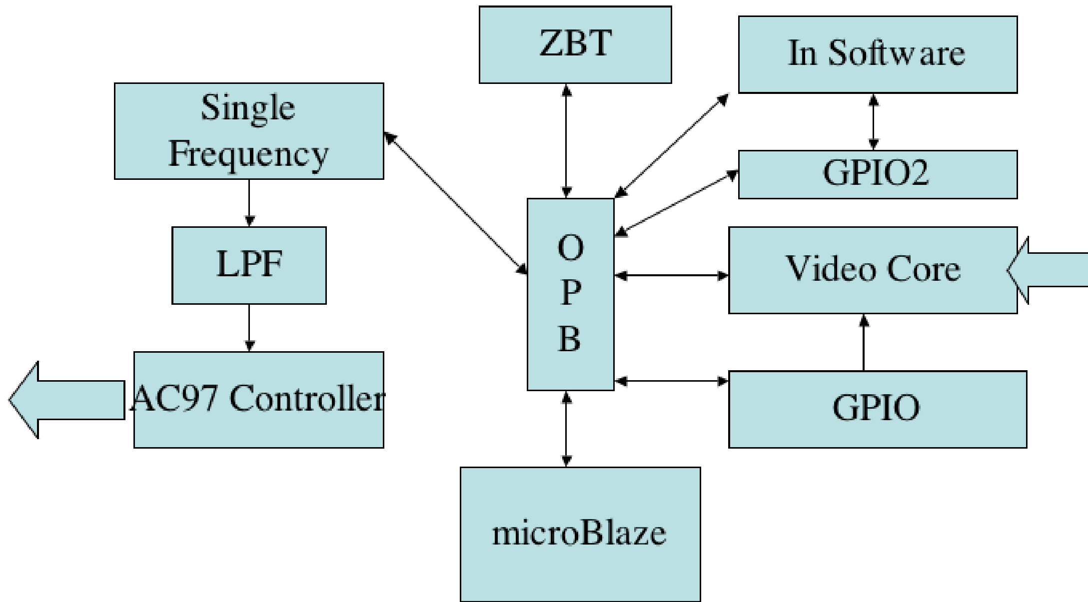


Figure 3: The Current Implementation of the Design

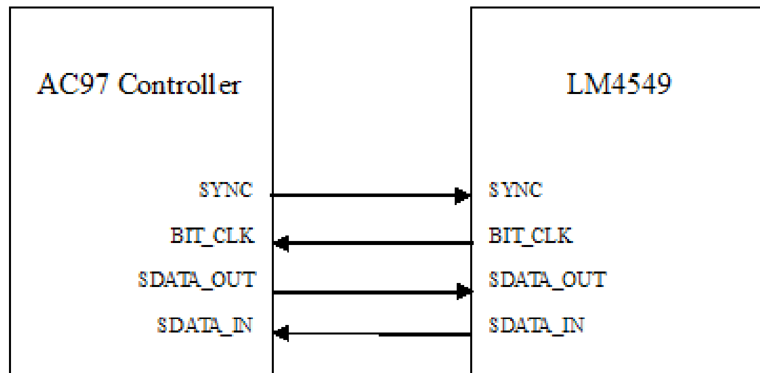


Figure 4: Controller and LM4549

3.2 LM4549

The LM4549 samples the audio signal with a certain sample rate specified by the user at initialization. We will discuss briefly on the initialization and some basic register for LM4549 at the end this section.

Output Frame SDATA_OUT, Controller Output to LM4549 Input Each sample of the data is stored in a Frame. A Frame is made up of 256 bits and is divided into thirteen Slots (0-12). In each Slot, the MSB is send first. The beginning of each frame is marked by a rising edge of the sync signal. Different slot is designated to contain different types of information. Table 1 summarizes the functionality of each slot.

Table 1: Slot Functionality Summary

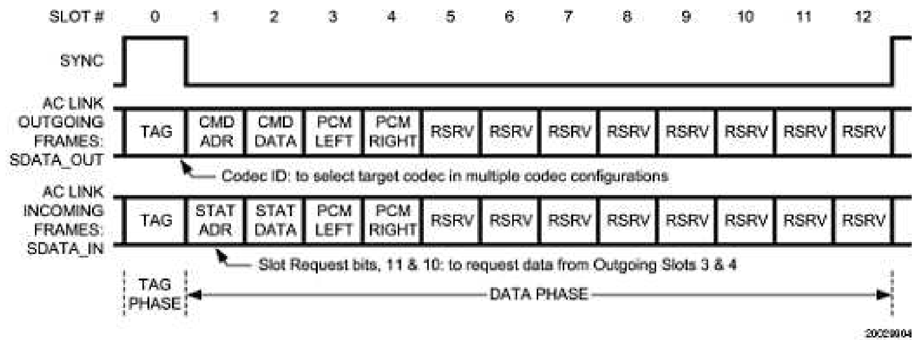
Slot No	Slot Name	Information	Number of bits per slot
0	Tag phase		16
1	Control address	The address of register for read or write	20
2	Register value	The value of the to be written or read from	20
3-4	PCM DAC data (L/R channel)	18 bit value of stream data	20
5-12	Reserved	Padded with zeroes	20

The LM4549 samples the bits on the negative edge of the BIT_CLK signal. To start sending data, drive the SYNC signal high. LM4549 will detect the SYNC signal on the next negative edge of the BIT_CLK, and trigger the LM4549 to expect the first bit (the most significant bit of slot 0) on the next negative edge of BIT_CLK as shown in the Figure below. As you can see, the first bit received is actually the second bit relative to the rising edge of the SYNC signal. The SYNC signal will stay high for 16 BIT_CLK cycle and then go low until the end of the frame, the SYNC signal marks Slot #0, which is also call the Tag Phase of the Frame.

SDATA_OUT: Slot 0 Tag Phase
The reason that Slot 0 is known as the Tag Phase is because it labels which slot contains valid data and needs to be registered by the LM4549. The table below outlines the function of each bit.

SDATA_OUT: Slot 1 Read/Write, Control Address Slot 1 indicates the address of the register which the controller would like to read or write to. The MSB of (bit 19) controls whether it is a Read or Write operation and (18 to 12) identities the address of the register. If a read operation of an address is requested, the value of the register will output in Slot 2 of the subsequent frame.

SDATA_OUT: Slot 2 Control Data Slot 2 is contains data to be written into the registers



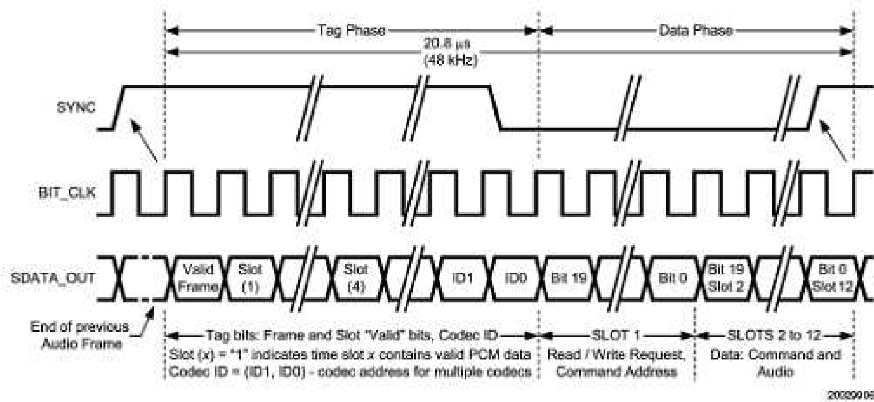


Figure 5: SYNC, Bit_CLK, and SDATA_OUT waveform

Table 2: default

Bit	Description	Comment
15	Valid Frame	1 = Frame contains valid data
14	Control register address	1 = Valid Control Address in slot 1 (Primary Codec only)
13	Control register data	1 = Valid Control Register Data in Slot 2 (Primary Codec ony)
12	Left DAC data in Slot 3	1 = Valid PCM Data in Slot 3
11	Right DAC data in Slot 4	1 = Valid PCM Data in Slot 4
10:2	Not used	All 0s
1:0	Codec ID (ID1, ID0)	The codec ID is used in a multi-codec system to identify the target Secondary codec for the Control Register address and/or data sent in the Output frame

Table 3: default

Bit	Description	Comment
19	Read/Write	1 = Read 0 = Write
18:12	Register Address	Identifies the Status/Command register for read/write
11:0	Reserved	All 0s

Table 4: default

Bit	Description	Comment
19:4	Control Register Write Data	Value of the Register, zeroes if operation is read
3:0	Reserved	All 0s

SDATA_OUT: Slot 3 & 4 Playback Left or Right Channels Slot 3 and 4 contains data for the channel, since the CODEC is only 18-bit resolution, the 18 MSB should be used.

Table 5: default

Bit	Description	Comment
19:0	PCM DAC Data (Left/Right Channels)	Slots used to stream data to DACs for all Primary or Secondary modes, set unused bits to 0s

3.3 AC97 Core

The core is driven by the BIT_CLK generated by the LM4549. The core is divided into 3 major sections: generates the SYNC signal, create SDATA_OUT, and handling SDATA_IN.

SYNC Signal Generation The SYNC signal goes high as soon as the start_Sync signal goes high. After 16 clock cycles, it will generate a reset_Sync which will pull the SYNC signal low. To keep track of the slot number, a 16-bit shift register and a 4-bit shift register is designated to do so. The reason to use two shift register is so that we can count 16 bit for slot 0 and 20 bit for the remaining slots. The 16-bit shift register starts counting when either start_Sync goes high or output from 4-bit shift register, delay_4 goes high. The signal delay_4 goes high when slot_end goes high for slot 1 to 12. Finally, a 13-bit shift register is used, enable only by signal slot_end to keep track to start/end of a frame. Refer to the simulation in the Appendix D.

Create SDATA_OUT To shift data out, we used three processes: a counter that counts the slot number, a mux to setup what data to shift into the signal vector new_data_out according to

the slot number, and a serial data shift register to shift data from new_data_out one by one. Refer to the simulation in the Appendix D.

Handling SDATA_IN To handle incoming data, we setup process which is controlled by the bits of slot 0.

3.4 AC97 Controller

The AC97 Controller instantiates the AC97 Core and a FIFO, it also contains a decoder which interface with the OPB bus. The decoder uses the bit 31 to 8 of the address of the OPB bus to determine whether the processor is trying to access this core. Bit 7 to 0 is reserved for instructions used in the core. The data output of the AC97 core is hooked up to the FIFO directly to stored sound data.

3.5 Tone Generator

The tone generator is basically a clock divider. It uses the BIT_CLK as a reference. Refer to Appendix for reference.

4 The Video Component

The video component is designed to save a single frame of video when asked to by the microblaze. The key portion however is that the signal is in one format, YCrCb, while ideally the signal should be saved as RGB0. The full code from the video component is included in appendix B.1. The component itself could be divided into several different components. These portions are not entirely obvious from examining the code. In fact in examining the code one of the problems that presents itself is the order in which the code was written. The subsections included are in the order the code was written.

To understand the code a bit of understanding about the nature of the digital video signal is necessary. When the signal is transmitting pixels they arrive in pairs Y,Cr Y,Cb. Each Y represent information about the brightness of individual pixel, while the Cr and Cb components give information about the colour components of the pixels. For any given pixel Cr and Cb information are used to determine the pixels quality but information about the Cr and the Cb are only given every second pixel. The reasoning behind this is the eye is more sensitive to brightness information then colour information and bandwith is at a premium.

4.1 Implementing Video Fetch and Store

This component was designed initially by a different group to perform the task of reading in the video data, parcelling it up and delivering it to the ZBT RAM. Although a similar one was in development by this group an already working component was to good to pass up. The code has since been modified to include the properties of modifying the data and taking in only single frames instead of video streams. Key signals in this code include the "newdata" signal which decides when the signal to the OPB bus, the pixel data is both valid and should be updated. The TRS signal is used as a marker within the video signal and is trickled down to the newdata signal and others when it is determined what is being marked. A TRS marker could be simply part of an end of a line or it could be a beginning of a frame marker.

Whenever the newdata signal goes high two pixels are moved onto the pixel bus "pixeldata".The first 8 bits of the data are for the Y (luminance) component of the pixel and the next 8 bits give the local Cr value. The next 16 bits start with the Y (luminance) component of the second pixel and then follows with the local Cb value. This pixel bus is later transfered onto the the OPB bus so long as the data is valid which is then transferred to the ZBT RAM.

The CPU makes use of this core in signalling the core when to stop capturing or to continue. This is done using the vector port, inflags, which is connected to the GPIO making for a very simple interface. This is expanded in However this is improved later on in section 4.3 in order to implement in hardware single frame capture.

4.2 YCrCb to RGB0

The format presented above in many ways allows for easy conversion to RGB0. Before being transferred to the Pixel data bus the data can be delayed for under one video clock cycle or 29 nanoseconds. Within this time a set of process' are implemented that generate the RGB data. The YCrCb HDL code that describes this transition is generally only mathematical relations. The only interesting component is the MULT18X18 multiplier, which is a Vertex II primitive. It was implemented in hopes of producing faster VHDL code and allowing the calculations to be performed within a clock period. The mathematically the transformation between colour spaces is:

$$R = 1.164(Y - 16) + 1.596(Cr - 128) \tag{1}$$
$$G = 1.164(Y' - 16)(0.813)(Cr - 128)0.392(Cb - 128) \tag{2}$$
$$B = 1.164(Y' - 16) + 2.017(Cb - 128) \tag{3}$$

This is implemented easily by removing half of the horizontal pixels. This implementation instead of approximating the Cr and Cb values only evaluates every second pixel and uses the next the Cb

value.[4] The result is RGB data at 8 bits a piece which is then padded with zeros to create a 32 bit word or an RGB0 pixel.

4.3 Single Frame Variation

So now the code captures the data needed however the implementation constantly captures the video data as it arrives and the project only requires the ability to capture specific frames. In order to do this the "inflags" port vector is expanded to include a secondary two bit signal. This signal now can be reset over the GPIO. A counter takes note of when the first piece of video information has been captured. When this has occurred twice logically one frame has been captured and hence it shuts down this logic block. It is then reset by writing zeroes to the two highest bits of the three bit GPIO.

4.4 Driver Code

The driver code for the video is actually relatively simple. It consists solely of writing the correct bits to the GPIO to get the video core to capture a single frame. It also has an off function which guarantees that the video core is shut off. Since the video core operates as a master on the OPB it should be noted that stopping it while it is writing a frame may be difficult. The drivers are included in appendix C.

5 Video to Audio Processing

The function of converting the video from the visual space into the audio space in a final implementation must be done in hardware. This is the portion however that received the least attention due to the amount of work required to simply get the video and audio cores working. The core was therefore rendered simply in C code instead. An example of this code which would need to be integrated into the audio code is included in appencdix C

5.1 C Implementation - Colour Averages

The C implementation uses the simplest implementation and hence is easily implementable in software as it is not highly processor intensive. The c-code may be able to run in less time then it take to process a single frame into the ZBT buffer. The algorithm used simply runs an average of all the RGB values in a single row. The resulting numbers represent the respective intensities of the the colours in the image. These value could then be used either to select from pre-determined audio cues or to create an appropriate frequency amplitude hybrid matching the RGB space onto

[4] You may be noticing that in fact the image will be compress horizontal resulting in a distrotion. Although this can be easily fixed, for this project there should be more then enough information in the remaining image that dropping half of the horizontal pixels shouldn't affect anything.

three frequencies. The first is a simple proof of concept while the later provides a more intense image. It can be imagined that the a bright white image would be a loud chord of three distinct tones, while a pre-dominantly dull yellow image would be a less intense combination of two tones.

5.2 Core Ideas

Core ideas[5] presents some of the ideas about possible hardware implementations of video to audio conversion. Most of these ideas would need to ideally be implemented without using a ZBT address space and instead be implemented almost entirely in hardware attached directly to the output of the video inputs core. In order to have enough freely available memory it may be necessary to reduce the size of the image, which is easily accomplished by either skipping pixels or entire lines.

5.2.1 Colour Amplitude Frequency Matching

This would be a simple method of doing what is done in C code currently. If this is the chosen method it could be done with possibly very few lines of VHDL. This implementation only requires the contribution of the strength of each colour, which could be added to a running average. Hence it could be used to build a very simple design that did not require external memory.

5.2.2 Fourier Domain Analysis

This method entails moving the video spectrum into a Fourier domain and then transforming it back by changing the sampling frequency into one appropriate to the audio domain. This already has the difficulty of sampling frequency matching it would be difficult to figure out which signals to reject outright and which to keep. For instance some signals may be to unpleasant to listen to. This should actually be a fairly easy method to implement in concept as it would only require a data storage length of at most 320 * 2 * 3 bytes (3 colours * 2 signals - source and its fourier - and 320 pixels) . Also the algorythm could begin to work on the first pixel when it arrived allowing it to process while the signal is being created.

5.2.3 Stereo Fourier Signal

The Fourier method could also be expanded to give a stereo signal. The exact same code which generates the audio signal in the horizontal could then be used to generate a second signal in the vertical. These two could then be used as left and right audio channels the stereo audio effect might be able to yield a larger image perception in the listener.

[5] Pun Intended

5.2.4 Beyond

Further there are many other interesting implementations that could be done. Perhaps sequential frames could be compared and the difference between them would generate the audio. Or rough patterns could attempt to be detected in the signal to true and give the wearer an idea of the surroundings. For instance a strong contrast which would result in a lot of high frequency noise under fourier analysis, could instead be dealt with by detecting the contrast and creating a signal around that mimics that location. Also with considerably more time the 3-D audio codec's developed by places like NASA could be used here to implement a more visual audio.

6 The Design Trees

The design tree is unfortunately forked in this design. Two designers worked seperatly and an audio as well as a video design tree have been created. The hope that one tree would germinate the other and create a third tree that would bloom remains unfulfilled. A description of both existing trees is included below.

6.1 Audio

6.2 Video

The video core has currently makes use of four cores in the pcores directory, and some C-driver code. The C-driver code is written but actually outside of the directory structure; it is included separately. The cores inside the pcores directory include two cores from the ZBT example, one implementation of the snooopy core and the video processing core. The UCF file has been modified appropriately to accommodate all of these changes.

7 Outcome

In general this project has not been overtly successful. The project successfully generated two cores an audio and a video and c code to translate between the two. However, neither the video nor the audio core successfully implements in reality. In fact both cores happen independently developed the same problem of not interfacing with the OPB correctly. Although the there may be different or simple solutions to these problems they still are inherently currently not working. Unfortunately this means it is impossible to determine if the logic actually is working correctly in either core.

The main problem existing with this project is that it was approached from two different angles. The video and the audio were develop by independent developers, when it would have been more

efficient to work together on the same core. Dolling out tasks evenly is not always the best approach.

To finish this implementation a few steps need to be taken. First the audio and the video cores need to be corrected so that they operate within current parameters. Next a core which translates video to audio could be written and implemented. This could follow any of the models outlined in section 5.2, Core Ideas. Further to this the audio and video cores would need to be expanded accordingly.

8 Conclusion

This project is a worthwhile and directly useful one. It could easily be made into one which is directly applicable to several circumstances. However the implementation at this rate of work requires at least another 4 months of work to truly deliver an audio image.

APPENDIX

A Additional Video Information - Chipsets

The Multimedia board contains both a video encoder and a video decoder. Although the video decoder is the focus of this project, the two are similar and of the same make. The video encoder is an ADV7194 and the video decoder is an ADV7185, full schematics for both can be found in PDF format on the web. Both devices use an I2C interface for configuration, however for the the video input (the 7185) generally configuration is unnecessary. When a video source is plugged into the multimedia board the chipset automatically recognizes the signal and its type and begins coding the signal into a digital one. As well the decoder has the property of automatically recognizing whether a signal is connected to the S-Video or Composite video inputs. The digital signal, along with a clock is immediately transmitted to the FPGA on a set of pins listed in multimedia boards info package.

The video encoder on the other hand requires a signal from the I2C interface in order to begin encoding the information for the output. Without this signal the encoder will not produce an output. Also the Video output does not detect where a connector is attached a video output source must be chosen in order for it to work.

B VHDL Code

B.1 Video Component

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

entity vidcap is  --USER-- change entity name
  generic
  (
    C_OPB_AWIDTH         : INTEGER := 32;
    C_OPB_DWIDTH         : INTEGER := 32;
    C_DEV_BURST_ENABLE   : INTEGER := 0;
    C_DEV_MAX_BURST_SIZE : INTEGER := 64;
    C_FAMILY             : string  := "virtex2";
    C_FBADDR             : std_logic_vector(0 to 31) := X"00005000"
  );

  port
```

```
  (
    --Required OPB bus ports, do not add to or delete
    Mn_ABus           : out std_logic_vector(0 to C_OPB_AWIDTH - 1 );
    Mn_DBus           : out std_logic_vector(0 to C_OPB_DWIDTH - 1 );
    Mn_request        : out std_logic;
    Mn_busLock        : out std_logic;
    Mn_select         : out std_logic;
    Mn_RNW            : out std_logic;
    Mn_BE             : out std_logic_vector(0 to C_OPB_DWIDTH/8 - 1 );
    Mn_seqAddr        : out std_logic;
    OPB_Clk           : in  std_logic := '0';
    OPB_Rst           : in  std_logic := '0';
    OPB_DBus          : in  std_logic_vector(0 to C_OPB_DWIDTH - 1 ) := (others => '0');
    OPB_MGrant        : in  std_logic := '0';
    OPB_xferAck       : in  std_logic := '0';
    OPB_errAck        : in  std_logic := '0';
    OPB_retry         : in  std_logic := '0';
    OPB_timeout       : in  std_logic := '0';


    --User ports
    led1              : out std_logic;  -- these our old and not necessary
    led2              : out std_logic;
    YCrCb_in          : in std_logic_vector(9 downto 0);  -- video in
    vid_clk           : in std_logic;   -- video clock in
    inflags           : in std_logic_vector(0 to 2)      -- control flags from ublaze / GPIO
  );
end entity vidcap;

architecture imp of vidcap is
  signal Mn_select_s : std_logic := '0';
  signal Mn_request_s : std_logic := '0';
  signal go             : std_logic_vector (1 downto 0);  -- my control variable

  signal pixeldata        : std_logic_vector (31 downto 0) := X"00000000"; -- what's put on the
  signal vidcount             : std_logic_vector (0 to 31);  -- kinda like pixel number across
 -- signal cnt                : std_logic_vector (0 to 31) := X"00000000";
  signal write_ena            : std_logic;  -- monty's control variable

  signal H_rg                 : std_logic_vector (4 downto 0);
  signal H_rising             : std_logic;
  signal TRS                  : std_logic;
  signal V_falling            : std_logic;
  signal V_rising             : std_logic;
  signal YCrCb_rg1            : std_logic_vector (9 downto 0);
  signal YCrCb_rg2            : std_logic_vector (9 downto 0);
  signal YCrCb_rg3            : std_logic_vector (9 downto 0);
  signal YCrCb_rg4            : std_logic_vector (9 downto 0);
```

```
  signal YCrCb_rg5            : std_logic_vector (9 downto 0);

--Edit by Peter Oakham : Mar. 24th
  signal north                  : std_logic := '1'; -- and of monty and peter's control variabl
  signal change                   : std_logic := '0';  -- indicator of a call from the inflags

  signal Red                    : std_logic_vector (7 downto 0);  --Final RGB values
  signal Blue                   : std_logic_vector (7 downto 0);
  signal Green                  : std_logic_vector (7 downto 0);
  signal Red1                   : std_logic_vector (35 downto 0); -- out of bounds RGB values
  signal Blue1                  : std_logic_vector (35 downto 0);
  signal Green1                 : std_logic_vector (35 downto 0);

-- These constants are used to multiply to obtain RGB
  constant const1: std_logic_vector(17 downto 0) := "000000000100101011"; -- 1.164 = 01.0010100
  constant const2: std_logic_vector(17 downto 0) := "000000000110011000"; -- 1.596 = 01.1001100
  constant const3: std_logic_vector(17 downto 0) := "000000000011010000"; -- 0.813 = 00.1101000
  constant const4: std_logic_vector(17 downto 0) := "000000000001100100"; -- 0.392 = 00.0110010
  constant const5: std_logic_vector(17 downto 0) := "000000001000000100"; -- 2.017 = 10.0000010
  signal P1,P2,P3,P4,P5: std_logic_vector(35 downto 0);  -- products of multipliacation
  constant Maskaraid: std_logic_vector(35 downto 0) := X"FFFFFFC00"; --"11111111111111111111111
  signal ext_y: std_logic_vector(17 downto 0);  --sign extending inputs for multiplier
  signal ext_cr: std_logic_vector(17 downto 0);
  signal ext_cb: std_logic_vector(17 downto 0);
  signal oldvidcount          : std_logic_vector (0 to 31);  -- delayed one vidcount
  signal checkplease           : std_logic_vector (0 to 2);  -- data ready for pixeldata

-- the primitive multiplier
component MULT18X18
port(
  A,B  : in std_logic_vector  (17 downto 0);
  P    : out std_logic_vector (35 downto 0)
);
end component;


--end of Edit
-- control signals enbedded in video
  signal Fo                   : std_logic;
  signal Ho                   : std_logic;
  signal Vo                   : std_logic;

  signal MstReq2              : std_logic;
  signal MstReq3              : std_logic;
  signal NewData              : std_logic;
begin
  led1 <= write_ena;
  --led2 <= cnt(22);
```

```
  write_ena <= inflags(0);

  Mn_busLock <= '0';
  Mn_RNW <= '0';
  Mn_BE <= (others => '1');
  Mn_seqAddr <= '0';

  Mn_request <= Mn_request_s;
  Mn_select <= Mn_select_s;

-- allows a inflags to be called without interference
 process (inflags)
 begin
  change <= '1';
 end process;

-- these are OPB events unaltered from Monty's code
  process (OPB_Clk)
    variable addr : std_logic_vector(0 to C_OPB_AWIDTH - 1);
  begin
    if go(1) = '0' and OPB_Clk'event and OPB_Clk = '1' then
      -- reset signals
      if OPB_Rst = '1' then
        Mn_select_s <= '0';
        MstReq2 <= '0';
      elsif OPB_MGrant = '1' then
        MstReq2 <= '0';
        Mn_select_s <= '1';
      elsif MstReq3 = '0' then
        MstReq2 <= NewData;
      end if;

      if OPB_xferAck = '1' or OPB_timeout = '1' or OPB_retry = '1' then
        Mn_select_s <= '0';
      end if;

      MstReq3 <= NewData;

   end if;
  end process;

  -- the TRS signal see the text
  TRS <= '1' when YCrCb_rg2(9 downto 2) = "00000000"
            and YCrCb_rg3(9 downto 2) = "00000000"
            and YCrCb_rg4(9 downto 2) = "11111111" else '0';

  Mn_request_s <= ((NewData and not MstReq3) or MstReq2) and not Mn_select_s;
```

```
  NewData <= write_ena and not (Ho or vidcount(30) or OPB_Rst);

-- used to allow simple comparison below
  North <=  Mn_select_s AND (NOT go(1)) ;

  Mn_DBus <= pixeldata when ( North = '1') else (others => 'Z'); --considering switching to no
  Mn_ABus <= ("0000000000" & (vidcount(10 to 29) & "00") + C_FBADDR)
             when (North = '1') else (others => 'Z');

  Ho <= H_rg(0) or H_rg(4) ;
  H_rising <= H_rg(0) and not H_rg(1) ;
  V_rising <= ( TRS and YCrCb_rg1(7) ) and not Vo ;
  V_falling <= ( TRS and not YCrCb_rg1(7) ) and Vo ;

-- whenever the video clock goes high
  process (vid_clk)
  begin  -- reset ?
    if  (OPB_Rst = '1' or go(1) = '1' ) then
      YCrCb_rg1 <= (others => '0') ;
      YCrCb_rg2 <= (others => '0') ;
      YCrCb_rg3 <= (others => '0') ;
      YCrCb_rg4 <= (others => '0') ;
      YCrCb_rg5 <= (others => '0') ;

      Fo <= '0' ;
      Vo <= '0' ;
      H_rg <= "00000";
      vidcount <= (others => '0');
    elsif vid_clk'event and vid_clk = '1' then
      YCrCb_rg1 <= YCrCb_in ;   -- move the video data down the pipe
      YCrCb_rg2 <= YCrCb_rg1 ;
      YCrCb_rg3 <= YCrCb_rg2 ;
      YCrCb_rg4 <= YCrCb_rg3 ;
      YCrCb_rg5 <= YCrCb_rg4 ;

      if ( TRS = '1' ) then    -- are we at the begining of a line or not
        Fo <= YCrCb_rg1(8) ;
        Vo <= YCrCb_rg1(7) ;
        H_rg(4 downto 0) <= H_rg(4 downto 1) & YCrCb_rg1(6) ;
      else
        Fo <= Fo ;
        Vo <= Vo ;
        H_rg(4 downto 0) <= H_rg(3 downto 0) & H_rg(0) ;
      end if;

         oldvidcount <= vidcount;

         if V_falling = '1' and Fo = '0' then
```

```
        vidcount <= (others => '0');
      elsif H_rg(4) = '1' and H_rg(3) = '0' then
        vidcount(30 to 31) <= "00";
      elsif Ho = '1' then
        vidcount(30) <= '1';
      else
        vidcount <= vidcount + 1;
      end if;
         if( (oldvidcount ="00000000000000000000000000000000") and (vidcount(0) = '1') ) then
         go <= go + "01";  -- when go reaches two it shuts off the process'
           elsif change = '1' then
           change <= '0'  ; -- reset change
           go <= inflags(2) & inflags(1);  -- reset inflags allowing the process to go ahead
         end if;

         --cnt <= cnt + 1;
    end if;
  end process;

  process (NewData)
  begin

-- when newdata is ready it is sign extended to go into the multiplier
        ext_y  <= "0000000" & (('0' & YCrCb_rg2) - "00001000000");
        ext_cr <= "0000000" & (('0' & YCrCb_rg3) - "01000000000");
        ext_cb <= "0000000" & (('0' & YCrCb_rg5) - "01000000000");

  end process;

-- multiplier is not clocked and for our purposes is assumed to be instantaneous
mult1a: MULT18X18 port map (A => const1, B => ext_y, P => P1);
mult2a: MULT18X18 port map (A => const2, B => ext_cr, P => P2);
mult3a: MULT18X18 port map (A => const3, B => ext_cr, P => P3);
mult4a: MULT18X18 port map (A => const4, B => ext_cb, P => P4);
mult5a: MULT18X18 port map (A => const5, B => ext_cb, P => P5);

-- end process;  -- SoS

-- create RGB signal
  process(P5)
    begin
        Red1 <= P1 + P2;
        Green1 <= P1 -  P3 - P4;
        Blue1 <= P1 +  P5;
  end process;


-- fix out of boundedness
```

```
   process(Red1)
   begin
      if (Red1(35) = '1') then
          Red <= "00000000";
      elsif ((Red1 and maskaraid) > 0) then    -- SoS  instead of != '0', change to > 1
          Red <= "11111111";
      else
          Red <= Red1(9 downto 2);
      end if;
         checkplease <= checkplease + 1;
   end process;

  process(Green1)
   begin
      if (Green1(35) = '1') then
          Green <= "00000000";
      elsif ((Green1 and Maskaraid) > 0) then    -- SoS  instead of != '0', change to > 1
          Green <= "11111111";
      else
          Green <= Green1(9 downto 2);
      end if;
         checkplease <= checkplease + 1;
  end process;

  process(Blue1)
   begin
      if (Blue1(35) = '1') then
          Blue <= "00000000";
      elsif ((Blue1 and maskaraid) > 0) then    -- SoS  instead of != '0', change to > 1
          Blue <= "11111111";
      else
          Blue <= Blue1(9 downto 2);
      end if;
         checkplease <= checkplease + 2;
  end process;


-- queue data to go onto OPB bus.
  Process(checkplease(2))
    begin
    if OPB_Rst = '1' then
                pixeldata <= (others => '0');
    elsif NewData'event and NewData = '1' then
                pixeldata(31 downto 24) <=  Red(7 downto 0);      -- Y'1  -- R
          pixeldata(23 downto 16) <=  Green(7 downto 0);         -- Cr0  -- G
          pixeldata(15 downto 8)  <=  Blue(7 downto 0); --&         -- Y'0  -- B
          pixeldata(7 downto 0)   <=  (others => '0');    -- Cb0  -- 0
        end if;
```

```
      checkplease <= "000";
   end process;

-- End of Edit

end architecture imp;
```

B.2 Reference Video Code

B.3 Audio Code

```
-------------------------------------------------------------------------------
-- $Id: opb_ac97_controller.vhd,v 1.1 2003/07/01 10:42:08 patoh Exp $
-------------------------------------------------------------------------------
-- opb_ac97_controller.vhd
-------------------------------------------------------------------------------
--
--                  ****************************
--                  ** Copyright Xilinx, Inc. **
--                  ** All rights reserved.   **
--                  ****************************
--
-------------------------------------------------------------------------------
-- Filename:        opb_ac97_controller.vhd
--
-- Description:
--
-- VHDL-Standard:   VHDL'93
-------------------------------------------------------------------------------
-- Structure:
--              opb_ac97_controller.vhd
--
-------------------------------------------------------------------------------
-- Author:          goran
-- Edited by: Simon So
-- Revision:        $Revision: 2.1 $
-- Date:            $Date: 2004/04/03 10:42:08 $
--
-- History:
--   goran  2002-01-09    First Version
--   Simon  2004-04-03    Second Version
--
-------------------------------------------------------------------------------
-- Naming Conventions:
--      active low signals:                     "*_n"
```

```
--      clock signals:                          "clk", "clk_div#", "clk_#x"
--      reset signals:                          "rst", "rst_n"
--      generics:                               "C_*"
--      user defined types:                     "*_TYPE"
--      state machine next state:               "*_ns"
--      state machine current state:            "*_cs"
--      combinatorial signals:                  "*_com"
--      pipelined or register delay signals:    "*_d#"
--      counter signals:                        "*cnt*"
--      clock enable signals:                   "*_ce"
--      internal version of output port         "*_i"
--      device pins:                            "*_pin"
--      ports:                                  - Names begin with Uppercase
--      processes:                              "*_PROCESS"
--      component instantiations:               "<ENTITY_>I_<#|FUNC>
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity OPB_AC97_CONTROLLER is
  generic (
    C_OPB_AWIDTH      : integer                     := 32;
    C_OPB_DWIDTH      : integer                     := 32;
    C_BASEADDR        : std_logic_vector(0 to 31) := X"FFFF_8000";
    C_HIGHADDR        : std_logic_vector          := X"FFFF_80FF";
    C_PLAYBACK        : integer                     := 1;
    C_RECORD          : integer                     := 0;
    -- value of 0,1,2,3,4
    -- 0 = No Interrupt
    -- 1 = empty      Nr_Of_Valid_Words = 0
    -- 2 = halfempty  Nr_Of_Valid_Words = 0-7
    -- 3 = halffull   Nr_Of_Valid_Words = 8-16
    -- 4 = full       Nr_Of_Valid_Words = 16
    C_PLAY_INTR_LEVEL : integer                     := 1;
    C_REC_INTR_LEVEL  : integer                     := 4;

    C_FSL_DWIDTH      : integer                     := 32
    );
  port (
    -- Global signals
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;

    -- OPB signals
    OPB_ABus    : in std_logic_vector(0 to 31);
    OPB_BE      : in std_logic_vector(0 to 3);
    OPB_RNW     : in std_logic;
```

```
    OPB_select  : in std_logic;
    OPB_seqAddr : in std_logic;
    OPB_DBus    : in std_logic_vector(0 to 31);

    OPB_AC97_CONTROLLER_DBus    : out std_logic_vector(0 to 31);
    OPB_AC97_CONTROLLER_errAck  : out std_logic;
    OPB_AC97_CONTROLLER_retry   : out std_logic;
    OPB_AC97_CONTROLLER_toutSup : out std_logic;
    OPB_AC97_CONTROLLER_xferAck : out std_logic;

    -- Interrupt signals
    Playback_Interrupt : out std_logic;
    Record_Interrupt   : out std_logic;

    -- CODEC signals
    Bit_Clk   : in  std_logic;
    Sync      : out std_logic;
    SData_Out : out std_logic;
    SData_In  : in  std_logic

    -- PlayBack FSL signals
      --FSL_S_Clk     : out std_logic;
    --FSL_S_Read    : out std_logic;
    --FSL_S_Data    : in  std_logic_vector(0 to C_FSL_DWIDTH-1);
    --FSL_S_Control : in  std_logic;
    --FSL_S_Exists  : in  std_logic
    );

end entity OPB_AC97_CONTROLLER;

--library opb_ac97_controller_v1_00_a;
--use opb_ac97_controller_v1_00_a.all;


--library Common_v1_00_a;
--use Common_v1_00_a.pselect;

library unisim;
use unisim.all;

--library opb_ac97_controller_v1_00_a;
--use opb_ac97_controller_v1_00_a.opb_ac97_core;

architecture IMP of OPB_AC97_CONTROLLER is

  component opb_freq_gen is
    generic (
   C_freq_spec_bit : integer :=16  -- 5 use 5 bit for sim purpose
```

```
    );
    port (
-- signals
Data_out : out std_logic_vector(0 to 15);  -- shift out 16 bit data
Bit_clk : in std_logic; -- bit_clk from the ac97
Enable : in std_logic
     );
  end component opb_freq_gen;

  component opb_ac97_core is
    generic (
      C_PLAYBACK : integer := 1;
      C_RECORD   : integer := 0
      );
    port (
      -- signals belonging to Clk clock region
      Clk   : in std_logic;
      Reset : in std_logic;

      AC97_Reg_Addr         : in  std_logic_vector(0 to 6);
      AC97_Reg_Read         : in  std_logic;
      AC97_Reg_Write_Data   : in  std_logic_vector(0 to 15);
      AC97_Reg_Read_Data    : out std_logic_vector(0 to 15);
      AC97_Reg_Access       : in  std_logic;
      AC97_Got_Request      : out std_logic;
      AC97_Reg_Finished     : out std_logic;
      AC97_Request_Finished : in  std_logic;
      CODEC_RDY             : out std_logic;

      In_Data_FIFO   : in  std_logic_vector(0 to 15);
      In_Data_Exists : in  std_logic;
      in_FIFO_Read   : out std_logic;

      Out_Data_FIFO  : out std_logic_vector(0 to 15);
      Out_FIFO_Full  : in  std_logic;
      Out_FIFO_Write : out std_logic;

      -- signals belonging to Bit_Clk clock region
      Bit_Clk   : in  std_logic;
      Sync      : out std_logic;
      SData_Out : out std_logic;
      SData_In  : in  std_logic);
  end component opb_ac97_core;

  component pselect is
    generic (
      C_AB  : integer;
      C_AW  : integer;
```

```
    C_BAR : std_logic_vector);
  port (
    A      : in  std_logic_vector(0 to C_AW-1);
    AValid : in  std_logic;
    ps     : out std_logic);
end component pselect;

component SRL_FIFO is
  generic (
    C_DATA_BITS : integer;
    C_DEPTH     : integer);
  port (
    Clk         : in  std_logic;
    Reset       : in  std_logic;
    Clear_FIFO  : in  std_logic;
    FIFO_Write  : in  std_logic;
    Data_In     : in  std_logic_vector(0 to C_DATA_BITS-1);
    FIFO_Read   : in  std_logic;
    Data_Out    : out std_logic_vector(0 to C_DATA_BITS-1);
    FIFO_Full   : out std_logic;
    Data_Exists : out std_logic;
    Half_Full   : out std_logic;
    Half_Empty  : out std_logic
    );
end component SRL_FIFO;

component FDRE is
  port (
    Q  : out std_logic;
    C  : in  std_logic;
    CE : in  std_logic;
    D  : in  std_logic;
    R  : in  std_logic);
end component FDRE;

component FDSE is
  port (
    Q  : out std_logic;
    C  : in  std_logic;
    CE : in  std_logic;
    D  : in  std_logic;
    S  : in  std_logic);
end component FDSE;

component FDR is
  port (Q : out std_logic;
        C : in  std_logic;
        D : in  std_logic;
```

```
        R : in  std_logic);
end component FDR;

component FDCE is
  port (
    Q   : out std_logic;
    C   : in  std_logic;
    CE  : in  std_logic;
    D   : in  std_logic;
    CLR : in  std_logic);
end component FDCE;

function Addr_Bits (x, y : std_logic_vector(0 to C_OPB_AWIDTH-1)) return integer is
  variable addr_nor : std_logic_vector(0 to C_OPB_AWIDTH-1);
begin
  addr_nor := x xor y;
  for i in 0 to C_OPB_AWIDTH-1 loop
    if addr_nor(i) = '1' then return i;
    end if;
  end loop;
  return(C_OPB_AWIDTH);
end function Addr_Bits;

constant C_AB : integer := Addr_Bits(C_HIGHADDR, C_BASEADDR);

-- the address to decode from the OPB_BUS is set here
-- we are only interested in range from 27 to 29 (recall that this little endian <0 to 31>)
-- for instance OUT_FIFO_ADR = 001 implies that we need to set in the driver file volatile i
subtype  ADDR_CHK is natural range C_OPB_AWIDTH-5 to C_OPB_AWIDTH-3;
constant IN_FIFO_ADR     : std_logic_vector(0 to 2) := "000";
constant OUT_FIFO_ADR    : std_logic_vector(0 to 2) := "001";
constant FIFO_STATUS_ADR : std_logic_vector(0 to 2) := "010";
constant FIFO_CTRL_ADR   : std_logic_vector(0 to 2) := "011";
constant AC97_CTRL_ADR   : std_logic_vector(0 to 2) := "100";
constant AC97_READ_ADR   : std_logic_vector(0 to 2) := "101";
constant AC97_WRITE_ADR  : std_logic_vector(0 to 2) := "110";

signal opb_ac97_controller_CS : std_logic;

signal opb_ac97_controller_CS_1 : std_logic;  -- Active as long as OPB_AC97_CONTROLLER_CS is
signal opb_ac97_controller_CS_2 : std_logic;  -- Active only 1 clock cycle during an
signal opb_ac97_controller_CS_3 : std_logic;  -- Active only 1 clock cycle during an
                                              -- access

signal xfer_Ack  : std_logic;
signal opb_RNW_1 : std_logic;

signal OPB_AC97_CONTROLLER_Dbus_i : std_logic_vector(0 to 15);
```

```
signal in_FIFO_Write      : std_logic;
signal in_FIFO_Read       : std_logic;
signal in_FIFO_Read_gated : std_logic;
signal in_Data_FIFO       : std_logic_vector(0 to 15);
signal in_FIFO_Full       : std_logic;
signal in_Data_Exists     : std_logic;
signal in_FIFO_Half_Full  : std_logic;
signal in_FIFO_Half_Empty : std_logic;

signal out_FIFO_Write      : std_logic;
signal out_FIFO_Read       : std_logic;
signal out_Data_Read       : std_logic_vector(0 to 15);
signal out_Data_FIFO       : std_logic_vector(0 to 15);
signal out_FIFO_Full       : std_logic;
signal out_Data_Exists     : std_logic;
signal out_FIFO_Half_Full  : std_logic;
signal out_FIFO_Half_Empty : std_logic;

-- Read Only
signal status_Reg : std_logic_vector(7 downto 0);
-- bit 7 '1' if out_FIFO hade a overrun condition
-- bit 6 '1' if in_FIFO hade a underrun condition
-- bit 5 If the CODEC is ready for commands
-- bit 4 Register Access is finished and if it was a read the data
--       is in AC97_Reg_Read register, reading AC97_Reg_Read_Register will clear
--       this bit
-- bit 3 out_FIFO_Data_Present
-- bit 2 out_FIFO_Empty
-- bit 1 in_FIFO_Empty
-- bit 0 in_FIFO_Full

signal out_FIFO_Overrun : std_logic;
signal in_FIFO_Underrun : std_logic;

signal clear_in_fifo  : std_logic;
signal clear_out_fifo : std_logic;

signal in_fifo_interrupt_en   : std_logic;
signal out_fifo_interrupt_en  : std_logic;

signal ac97_Reg_Addr       : std_logic_vector(0 to 6);
signal ac97_Reg_Read       : std_logic;
signal ac97_Reg_Write_Data : std_logic_vector(0 to 15);
signal ac97_Reg_Read_Data  : std_logic_vector(0 to 15);
signal ac97_Reg_Access     : std_logic;
signal ac97_Got_Request    : std_logic;
signal ac97_reg_access_S   : std_logic;
```

```
  signal ac97_Reg_Finished   : std_logic;
  signal ac97_Reg_Finished_i : std_logic;

  signal register_Access_Finished     : std_logic;
  signal register_Access_Finished_Set : std_logic;

  signal codec_rdy : std_logic;

  -- Playback signal
  signal sound_data : std_logic_vector(0 to 15);

begin  -- architecture IMP


  -----------------------------------------------------------------------------
  -- Handling the OPB bus interface
  -----------------------------------------------------------------------------

  -- Do the OPB address decoding, pselect is a decoder that decodes the address according to tl
  -- opb_ac97_controller_CS goes high when the correct address is decoded
  pselect_I : pselect
    generic map (
      C_AB  => C_AB,                    -- [integer]
      C_AW  => C_OPB_AWIDTH,            -- [integer]
      C_BAR => C_BASEADDR)              -- [std_logic_vector]
    port map (
      A      => OPB_ABus,               -- [in  std_logic_vector(0 to C_AW-1)]
      AValid => OPB_select,             -- [in  std_logic]
      ps     => opb_ac97_controller_CS);  -- [out std_logic]

  OPB_AC97_CONTROLLER_errAck  <= '0';
  OPB_AC97_CONTROLLER_retry   <= '0';
  OPB_AC97_CONTROLLER_toutSup <= '0';

  -----------------------------------------------------------------------------
  -- Decoding the OPB control signals
  -- generates a pulse if correct address is decoded to enable the controller
  -----------------------------------------------------------------------------
  opb_ac97_controller_CS_1_DFF : FDR
    port map (
      Q => opb_ac97_controller_CS_1,    -- [out std_logic]
      C => OPB_Clk,                     -- [in  std_logic]
      D => OPB_AC97_CONTROLLER_CS,      -- [in  std_logic]
      R => xfer_Ack);                   -- [in std_logic]

  opb_ac97_controller_CS_2_DFF : process (OPB_Clk, OPB_Rst) is
  begin  -- process opb_ac97_controller_CS_2_DFF
    if OPB_Rst = '1' then               -- asynchronous reset (active high)
```

```
    opb_ac97_controller_CS_2 <= '0';
    opb_ac97_controller_CS_3 <= '0';
    opb_RNW_1                <= '0';
  elsif OPB_Clk'event and OPB_Clk = '1' then  -- rising clock edge
    opb_ac97_controller_CS_2 <= opb_ac97_controller_CS_1
                                and not opb_ac97_controller_CS_2
                                and not opb_ac97_controller_CS_3;
    opb_ac97_controller_CS_3 <= opb_ac97_controller_CS_2;
    opb_RNW_1                <= OPB_RNW;
  end if;
end process opb_ac97_controller_CS_2_DFF;


-------------------------------------------------------------------------------
-- Selects what to read
-------------------------------------------------------------------------------

-- Selects what to read
Read_Mux : process (status_reg, OPB_ABus, out_Data_Read, ac97_Reg_Read_Data) is
begin  -- process Read_Mux
  OPB_AC97_CONTROLLER_Dbus_i <= (others => '0'); -- Reset Dbus_i to '0'
  if (OPB_ABus(ADDR_CHK) = FIFO_STATUS_ADR) then
    OPB_AC97_CONTROLLER_Dbus_i(15-status_reg'length+1 to 15) <= status_reg;    -- Shift the s
  elsif (OPB_ABus(ADDR_CHK) = AC97_READ_ADR) then
    OPB_AC97_CONTROLLER_Dbus_i(0 to 15) <= ac97_Reg_Read_Data;
  else
    OPB_AC97_CONTROLLER_Dbus_i(0 to 15) <= out_Data_Read;
  end if;
end process Read_Mux;

DWIDTH_gt_16 : if (C_OPB_DWIDTH > 16) generate -- use only when opb width > 16
  OPB_AC97_CONTROLLER_Dbus(0 to C_OPB_DWIDTH-17) <= (others => '0');    -- zero padded the fi
end generate DWIDTH_gt_16;

OPB_rdDBus_DFF : for I in C_OPB_DWIDTH-16 to C_OPB_DWIDTH-1 generate
  OPB_rdBus_FDRE : FDRE
    port map (
      Q  => OPB_AC97_CONTROLLER_DBus(I),  -- [out std_logic]
      C  => OPB_Clk,                  -- [in  std_logic]
      CE => opb_ac97_controller_CS_2,     -- [in  std_logic]
      D  => OPB_AC97_CONTROLLER_Dbus_i(I-(C_OPB_DWIDTH-16)),  -- [in  std_logic]
      R  => xfer_Ack);                -- [in std_logic]
end generate OPB_rdDBus_DFF;

-- Generating read and write pulses to the FIFOs
in_FIFO_write <= opb_ac97_controller_CS_2 and (not OPB_RNW_1) when (OPB_ABus(ADDR_CHK) = IN_F
out_FIFO_read <= opb_ac97_controller_CS_2 and OPB_RNW_1       when (OPB_ABus(ADDR_CHK) = OUT_
```

```
clear_in_fifo <= OPB_DBus(31) and opb_ac97_controller_CS_2 and (not OPB_RNW_1) when (OPB_ABus
clear_out_fifo <= OPB_DBus(30) and opb_ac97_controller_CS_2 and (not OPB_RNW_1) when (OPB_ABu

in_fifo_interrupt_en <= OPB_DBus(29) and opb_ac97_controller_CS_2 and (not OPB_RNW_1) when ((
out_fifo_interrupt_en <= OPB_DBus(28) and opb_ac97_controller_CS_2 and (not OPB_RNW_1) when

XFER_Control : process (OPB_Clk, OPB_Rst) is
begin  -- process XFER_Control
  if OPB_Rst = '1' then               -- asynchronous reset (active high)
    xfer_Ack <= '0';
  elsif OPB_Clk'event and OPB_Clk = '1' then  -- rising clock edge
    xfer_Ack <= opb_ac97_controller_CS_2;
  end if;
end process XFER_Control;

OPB_AC97_CONTROLLER_xferAck <= xfer_Ack;

-----------------------------------------------------------------------------
-- Status register
-----------------------------------------------------------------------------
FIFO_Error_Handle: process (OPB_Clk, OPB_Rst) is
begin  -- process FIFO_Error_Handle
  if OPB_Rst = '1' then                 -- asynchronous reset (active high)
    out_FIFO_Overrun <= '0';
    in_FIFO_Underrun <= '0';
  elsif OPB_Clk'event and OPB_Clk = '1' then  -- rising clock edge
    -- Reading FIFO_Status register will clear the error flags
    if (clear_in_fifo = '1') then
      in_FIFO_Underrun <= '0';
    elsif (in_Data_Exists = '0') then --and (in_FIFO_Read = '1')
      in_FIFO_Underrun <= '1';
    end if;
    if (clear_out_fifo = '1') then
      out_FIFO_Overrun <= '0';
    elsif (out_FIFO_Full = '1') and (out_FIFO_Write = '1') and (out_FIFO_read = '0') then
      out_FIFO_Overrun <= '1';
    end if;
  end if;
end process FIFO_Error_Handle;

status_reg(7) <= out_FIFO_Overrun;
status_reg(6) <= in_FIFO_Underrun;
status_reg(5) <= codec_rdy;
status_reg(4) <= register_Access_Finished;
status_reg(3) <= out_Data_Exists;
status_reg(2) <= not(out_Data_Exists);
status_reg(1) <= not(in_Data_Exists);
status_reg(0) <= in_FIFO_Full;
```

```
-----------------------------------------------------------------------------
-- AC97 Register Handling
-----------------------------------------------------------------------------
AC97_Write_Reg_Data : process (OPB_Clk, OPB_Rst) is
begin  -- process AC97_Write_Reg_Data
  if OPB_Rst = '1' then               -- asynchronous reset (active high)
    ac97_reg_write_data <= (others => '0');
  elsif OPB_Clk'event and OPB_Clk = '1' then  -- rising clock edge
    if (opb_ac97_controller_CS_2 = '1') and (OPB_RNW_1 = '0') and (OPB_ABus(ADDR_CHK) = AC97_
      ac97_reg_write_data <= OPB_DBus(C_OPB_DWIDTH-16 to C_OPB_DWIDTH-1);   -- the last 16 bi
    end if;
  end if;
end process AC97_Write_Reg_Data;

AC97_Access_Reg : process (OPB_Clk, OPB_Rst) is
begin  -- process AC97_Access_Reg
  if OPB_Rst = '1' then               -- asynchronous reset (active high)
    ac97_reg_addr     <= (others => '0');
    ac97_reg_read     <= '0';
    ac97_reg_access_S <= '0';
  elsif OPB_Clk'event and OPB_Clk = '1' then  -- rising clock edge
    ac97_reg_access_S <= '0';
    if (opb_ac97_controller_CS_2 = '1') and (OPB_RNW_1 = '0') and (OPB_ABus(ADDR_CHK) = AC97_
      ac97_reg_addr     <= OPB_DBus(C_OPB_DWIDTH-7 to C_OPB_DWIDTH-1);  -- the ac97 has 7 bit
      ac97_reg_read     <= OPB_DBus(C_OPB_DWIDTH-8);
      ac97_reg_access_S <= '1';
    end if;
  end if;
end process AC97_Access_Reg;

ac97_reg_access_FDCE : FDCE
  port map (
    Q   => ac97_reg_access,           -- [out std_logic]
    C   => OPB_Clk,                   -- [in  std_logic]
    CE  => ac97_reg_access_S,         -- [in  std_logic]
    D   => '1',                       -- [in  std_logic]
    CLR => ac97_Got_Request);         -- [in std_logic]

ac97_reg_access_FDSE : FDSE
  port map (
    Q  => register_Access_Finished,       -- [out std_logic]
    C  => OPB_Clk,                        -- [in  std_logic]
    CE => ac97_reg_access_S,              -- [in  std_logic]
    D  => '0',                            -- [in  std_logic]
    S  => register_Access_Finished_Set);  -- [in std_logic]

AC97_Register_SM : process (OPB_Clk, OPB_Rst) is
```

```
  begin  -- process AC97_Register_SM
    if OPB_Rst = '1' then               -- asynchronous reset (active high)
      ac97_Reg_Finished_i          <= '0';
      register_Access_Finished_Set <= '0';
    elsif OPB_Clk'event and OPB_Clk = '1' then  -- rising clock edge
      register_Access_Finished_Set <= '0';
      if (ac97_Reg_Finished = '1' and ac97_Reg_Finished_i = '0') then
        register_Access_Finished_Set <= '1';
      end if;
      ac97_Reg_Finished_i <= ac97_Reg_Finished;
    end if;
  end process AC97_Register_SM;

  -----------------------------------------------------------------------------
  -- Instanciating the FIFOs
  -----------------------------------------------------------------------------

in_FIFO_Read_gated <= in_Data_Exists;  --in_FIFO_Read and
in_Data_Exists <= '1';

--    Using_Playback : if (C_PLAYBACK = 1) generate

--     IN_FIFO : SRL_FIFO
--       generic map (
--         C_DATA_BITS => 16,              -- [integer]
--         C_DEPTH     => 16)              -- [integer]
--       port map (
--         Clk         => OPB_Clk,         -- [in  std_logic]
--         Reset       => OPB_Rst,         -- [in  std_logic]
--         Clear_FIFO  => clear_in_fifo,  -- [in  std_logic]
--         FIFO_Write  => in_FIFO_Write,   -- [in  std_logic]
--         Data_In     => OPB_DBus(C_OPB_DWIDTH-16 to C_OPB_DWIDTH-1),  -- [in  std_logic_vecto
--         FIFO_Read   => in_FIFO_Read_gated,    -- [in  std_logic]
--         Data_Out    => in_Data_FIFO,  -- [out std_logic_vector(0 to C_OPB_DWIDTH-1)]
--        FIFO_Full   => in_FIFO_Full,    -- [out std_logic]
--         Data_Exists => in_Data_Exists,  -- [out std_logic]
--         Half_Full   => in_FIFO_Half_Full,    -- [out std_logic]
--         Half_Empty  => in_FIFO_Half_Empty);  -- [out std_logic]
--   end generate Using_Playback;

  No_Playback : if (C_PLAYBACK = 0) generate
    in_Data_FIFO   <= (others => '0');
    in_FIFO_Full   <= '0';
    in_Data_Exists <= '0';
  end generate No_Playback;

  Using_Recording : if (C_RECORD = 1) generate
```

```
  OUT_FIFO : SRL_FIFO
    generic map (
      C_DATA_BITS => 16,              -- [integer]
      C_DEPTH     => 16)              -- [integer]
    port map (
      Clk         => OPB_Clk,         -- [in  std_logic]
      Reset       => OPB_Rst,         -- [in  std_logic]
      Clear_FIFO  => clear_out_fifo,  -- [in  std_logic]
      FIFO_Write  => out_FIFO_Write,  -- [in  std_logic]
      Data_In     => out_Data_FIFO,  -- [in  std_logic_vector(0 to C_OPB_DWIDTH-1)]
      FIFO_Read   => out_FIFO_Read,   -- [in  std_logic]
      Data_Out    => out_Data_Read,  -- [out std_logic_vector(0 to C_OPB_DWIDTH-1)]
      FIFO_Full   => out_FIFO_Full,   -- [out std_logic]
      Data_Exists => out_Data_Exists,       -- [out std_logic]
      Half_Full   => out_FIFO_Half_Full,    -- [out std_logic]
      Half_Empty  => out_FIFO_Half_Empty);  -- [out std_logic]
end generate Using_Recording;

No_Recording : if (C_RECORD = 0) generate
  out_Data_Read   <= (others => '0');
  out_FIFO_Full   <= '0';
  out_Data_Exists <= '0';
end generate No_Recording;

-----------------------------------------------------------------------------
-- Instanciating the OPB_AC97_CONTROLLER core
-----------------------------------------------------------------------------
--FSL_S_Read <= In_FIFO_Read_gated;

opb_ac97_core_I : opb_ac97_core
  generic map (
    C_PLAYBACK => C_PLAYBACK,
    C_RECORD   => C_RECORD
    )
  port map (
    -- signals belonging to Clk clock region
    Clk   => OPB_Clk,                 -- [in  std_logic]
    Reset => OPB_Rst,                 -- [in  std_logic]

    AC97_Reg_Addr         => ac97_reg_addr,  -- [in  std_logic_vector(0 to 6)]
    AC97_Reg_Read         => ac97_reg_read,  -- [in  std_logic]
    AC97_Reg_Write_Data   => ac97_reg_write_data,  -- [in  std_logic_vector(0 to 15)]
    AC97_Reg_Read_Data    => ac97_reg_read_data,  -- [out std_logic_vector(0 to 15)]
    AC97_Reg_Access       => ac97_reg_access,     -- [in  std_logic]
    AC97_Got_Request      => ac97_got_request,    -- [out  std_logic]
    AC97_Reg_Finished     => ac97_reg_finished,   -- [out std_logic]
    AC97_Request_Finished => register_access_finished,  -- [in std_logic]
    CODEC_RDY             => codec_rdy,      -- [out std_logic]
```

```
      In_Data_FIFO   => sound_data,        --FSL_S_Data(16 to 31),    -- [in  std_logic_vector
      In_Data_Exists => '1',--FSL_S_Exists,    -- [in  std_logic]
      in_FIFO_Read   => in_FIFO_Read,    -- [out std_logic]

      Out_Data_FIFO  => Out_Data_FIFO,   -- [out std_logic_vector(0 to 15)]
      Out_FIFO_Full  => Out_FIFO_Full,   -- [in  std_logic]
      Out_FIFO_Write => Out_FIFO_Write,  -- [out std_logic]

      -- signals belonging to Bit_Clk clock region
      Bit_Clk   => Bit_Clk,             -- [in  std_logic]
      Sync      => Sync,                -- [out std_logic]
      SData_Out => SData_Out,           -- [out std_logic]
      SData_In  => SData_In);           -- [out std_logic]


  -----------------------------------------------------------------------------
  -- Instanciating the tone generator
  -----------------------------------------------------------------------------
  opb_freq_gen_I : opb_freq_gen
    generic map(
   C_freq_spec_bit => 16)  -- 5 use 5 bit for sim purpose
    port map(
-- signals
Data_out => sound_data,-- [out std_logic_vector(0 to 15)]  -- shift out 16 bit data
Bit_clk => Bit_Clk,--[in std_logic] -- bit_clk from the ac97
Enable => '1'); --[in std_logic]

end architecture IMP;

------------------Next -----------------

-------------------------------------------------------------------------------
-- $Id: opb_ac97_core.vhd,v 1.1 2003/07/01 10:42:08 patoh Exp $
-------------------------------------------------------------------------------
-- opb_ac97_core.vhd
-------------------------------------------------------------------------------
--
--                  ***************************
--                  ** Copyright Xilinx, Inc. **
--                  ** All rights reserved.   **
--                  ***************************
--
-------------------------------------------------------------------------------
-- Filename:        opb_ac97_core.vhd
--
-- Description:
--
```

```
-- VHDL-Standard:   VHDL'93
-------------------------------------------------------------------------------
-- Structure:
--              opb_ac97_core.vhd
--
-------------------------------------------------------------------------------
-- Author:          Simon
-- Revision:        $Revision: 1.2 $
-- Date:            $Date: 2004/03/31 18:42:08 $
--
-- History:
--   goran  2002-01-24    First Version
--
-------------------------------------------------------------------------------
-- Naming Conventions:
--      active low signals:                     "*_n"
--      clock signals:                          "clk", "clk_div#", "clk_#x"
--      reset signals:                          "rst", "rst_n"
--      generics:                               "C_*"
--      user defined types:                     "*_TYPE"
--      state machine next state:               "*_ns"
--      state machine current state:            "*_cs"
--      combinatorial signals:                  "*_com"
--      pipelined or register delay signals:    "*_d#"
--      counter signals:                        "*cnt*"
--      clock enable signals:                   "*_ce"
--      internal version of output port         "*_i"
--      device pins:                            "*_pin"
--      ports:                                  - Names begin with Uppercase
--      processes:                              "*_PROCESS"
--      component instantiations:               "<ENTITY_>I_<#|FUNC>
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity opb_ac97_core is
  generic (
    C_PLAYBACK : integer := 1;
    C_RECORD   : integer := 0
    );
  port (
    -- signals belonging to Clk clock region
    Clk   : in std_logic;
    Reset : in std_logic;

    AC97_Reg_Addr         : in  std_logic_vector(0 to 6);   -- AC97 Register address
    AC97_Reg_Read         : in  std_logic; -- AC97 Read or not Write control
    AC97_Reg_Write_Data   : in  std_logic_vector(0 to 15); -- AC97 Register value to be writter
```

```
    AC97_Reg_Read_Data    : out std_logic_vector(0 to 15);  -- AC97 Register Data Read back
    AC97_Reg_Access       : in  std_logic; -- goes high if want to read AC Registers
    AC97_Got_Request      : out std_logic; -- goes high after one bit_clk cycle when AC_Reg_Acc
    AC97_Reg_Finished     : out std_logic;
    AC97_Request_Finished : in  std_logic;
    CODEC_RDY             : out std_logic;

    In_Data_FIFO   : in  std_logic_vector(0 to 15);
    In_Data_Exists : in  std_logic;    -- not used anymore
    in_FIFO_Read   : out std_logic;

    Out_Data_FIFO  : out std_logic_vector(0 to 15);
    Out_FIFO_Full  : in  std_logic;
    Out_FIFO_Write : out std_logic;

    -- signals belonging to Bit_Clk clock region
    Bit_Clk   : in  std_logic;   -- generates by AC97
    Sync      : out std_logic;   -- generates the sync signal so AC97 knows when to receive in
    SData_Out : out std_logic;
    SData_In  : in  std_logic
    );

end entity opb_ac97_core;

library unisim;
use unisim.all;

architecture IMP of opb_ac97_core is

  component SRL16E is
    -- pragma translate_off
    generic (
      INIT : bit_vector := X"0000"
      );
    -- pragma translate_on
    port (
      CE  : in  std_logic;
      D   : in  std_logic;
      Clk : in  std_logic;
      A0  : in  std_logic;
      A1  : in  std_logic;
      A2  : in  std_logic;
      A3  : in  std_logic;
      Q   : out std_logic);
  end component SRL16E;

  component FDRSE is
    port (
```

```
    Q  : out std_logic;
    C  : in  std_logic;
    CE : in  std_logic;
    D  : in  std_logic;
    S  : in  std_logic;
    R  : in  std_logic);
end component FDRSE;

component FDCE is
  port (
    Q   : out std_logic;
    C   : in  std_logic;
    CE  : in  std_logic;
    D   : in  std_logic;
    CLR : in  std_logic);
end component FDCE;

component FD is
  -- pragma translate_off
  generic (
    INIT : bit := '0'
    );
  -- pragma translate_on
  port (
    Q : out std_logic;
    C : in  std_logic;
    D : in  std_logic
    );
end component FD;

signal rst_n    : std_logic;
signal sync_i : std_logic;
signal sync_i_1 : std_logic;

signal start_from_reset : std_logic;
signal start_sync       : std_logic;
signal start_sync_clean : std_logic;
signal reset_sync       : std_logic;

signal new_slot    : std_logic;
signal con_slot    : std_logic;
signal slot_end    : std_logic;
signal slot_end_1  : std_logic;
signal delay_4     : std_logic;
signal last_slot   : std_logic;

signal new_data_out : std_logic_vector(19 downto 0);
signal data_out     : std_logic_vector(19 downto 0);
```

```
  signal data_in      : std_logic_vector(19 downto 0);

  signal data_valid    : std_logic;
  signal got_read_data : std_logic;
  signal got_request   : std_logic;

  signal read_fifo   : std_logic;
  signal read_fifo_1 : std_logic;

  signal slot0 : std_logic_vector(15 downto 0);
  signal slot1 : std_logic_vector(19 downto 0);
  signal slot2 : std_logic_vector(19 downto 0);

  signal valid_Frame           : std_logic;
  signal valid_Control_Addr    : std_logic;
  signal valid_Control_Data    : std_logic;
  signal valid_Playback_Data_L : std_logic;
  signal valid_Playback_Data_R : std_logic;

  signal got_record_data : std_logic;

  signal valid_Record_Data_L : std_logic;
  signal valid_Record_Data_R : std_logic;
  signal fifo_written        : std_logic;
  signal write_fifo          : std_logic;

  signal slot_No : natural range 0 to 5;

  signal Bit_Index : integer;

  signal ac97_Reg_Access_1 : std_logic;
  signal ac97_Reg_Access_2 : std_logic;

  signal ac97_read_access  : std_logic;
  signal ac97_write_access : std_logic;

  signal ac97_Reg_Finished_i   : std_logic;

  signal In_Data_FIFO_i : std_logic_vector(0 to 15);  -- 16 bit data from the FIFO which conta

begin  -- architecture IMP

  -----------------------------------------------------------------------------
  -- Temporary signals for debugging in VHDL Simulator
  -----------------------------------------------------------------------------
    -- pragma translate_off
  Dbg : process (Bit_Clk) is
    variable tmp  : std_logic;
```

```
  variable tmp2 : std_logic;
begin  -- process Dbg
  if Reset = '1' then                 -- asynchronous reset (active high)
    Bit_Index <= 15;
    tmp       := '0';
    tmp2      := '0';
  elsif Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
    if (tmp = '1') then
      Bit_Index <= 15;
    elsif (tmp2 = '1') then
      Bit_Index <= 19;
    else
      Bit_Index <= Bit_Index - 1;
    end if;
    tmp  := start_sync;
    tmp2 := slot_end;
  end if;
end process Dbg;
  -- pragma translate_on

rst_n <= not reset;



In_Data_FIFO_i <= In_Data_FIFO; -- map the in_Data_FIFO port to internal port

-----------------------------------------------------------------------------
-- Handle AC97 register accesses (write)
-----------------------------------------------------------------------------
Reg_Access_Handle : process (Bit_Clk) is
begin  -- process Reg_Access_Handle
  if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
    ac97_Reg_Access_1 <= AC97_Reg_Access;
    ac97_Reg_Access_2 <= ac97_Reg_Access_1;

-- detect a rising edge on AC97_Reg_Access, use only the first pulse
    if (ac97_Reg_Access_1 = '1' and ac97_Reg_Access_2 = '0') then
      valid_Control_Addr <= '1';
      valid_Control_Data <= not AC97_Reg_Read;  -- '1' on writes
      AC97_Got_Request   <= '1';

-- resets the valid_Control_Addr and data, and got_request signal
elsif (valid_Control_Addr and got_request) = '1' then
      valid_Control_Addr <= '0';
      valid_Control_Data <= '0';
      AC97_Got_Request   <= '0';
    end if;
  end if;
```

```
end process Reg_Access_Handle;

-----------------------------------------------------------------------------
-- Setup all the slots at start of frame
-----------------------------------------------------------------------------
Setup_Slot0 : process (Bit_Clk) is
begin  -- process Setup_Slot0
  if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
    if (delay_4 and last_slot) = '1' then  -- Set up data in the last slot starting from 16 b
      slot0(15)         <= valid_Frame;
      slot0(14)         <= valid_Control_Addr;   -- set to one
      slot0(13)         <= valid_Control_Data;
      slot0(12)         <= valid_Playback_Data_L;
      slot0(11)         <= valid_Playback_Data_R;
      got_request       <= valid_Control_Addr;
      ac97_read_access  <= slot0(14) and not slot0(13);
      ac97_write_access <= slot0(14) and slot0(13);
    end if;
  end if;
end process Setup_Slot0;

valid_Frame <= valid_Control_Addr or valid_Playback_Data_L or valid_Playback_Data_R;

-- just tie to the signal to the vector
-- slot 1 = Register Address, use for reading and writing
-- if Read -> Reg value will arrive the next frame from Sdata_in
slot1(19)           <= AC97_Reg_Read;
slot1(18 downto 12) <= AC97_Reg_Addr;
slot1(11 downto 0) <= (others => '0');

-- slot 2 = value of the Address to be written
slot2(19 downto 4) <= AC97_Reg_Write_Data;
slot2( 3 downto 0) <= (others => '0');

-----------------------------------------------------------------------------
-- Generating the Sync signal
-----------------------------------------------------------------------------
Sync_SRL16E : SRL16E
  -- Once start_Sync goes high, it takes 16 cycle of bit_clk, before reset_Sync goes high
  -- this indicates the beginning of the slot0 and end of slot0
  -- pragma translate_off
  generic map (
    INIT => X"0000")                   -- [bit_vector]
  -- pragma translate_on
  port map (
    CE  => '1',                        -- [in  std_logic]
    D   => start_Sync,                 -- [in  std_logic]
    Clk => Bit_Clk,                    -- [in  std_logic]
```

```
      A0  => '1',                     -- [in  std_logic]
      A1  => '1',                     -- [in  std_logic]
      A2  => '1',                     -- [in  std_logic]
      A3  => '1',                     -- [in  std_logic]
      Q   => reset_Sync);             -- [out std_logic]

  Sync_FDRSE : FDRSE
    -- enables the internal Sync signal only when start_Sync goes high
    -- goes low when reset_Sync goes high
    port map (
      Q  => sync_i,                     -- [out std_logic]
      C  => Bit_Clk,                  -- [in  std_logic]
      CE => start_Sync,               -- [in  std_logic]
      D  => '1',                      -- [in  std_logic]
      S  => '0',                      -- [in  std_logic]
      R  => reset_Sync);              -- [in std_logic]

  Sync <= sync_i;

  Shift_Sync_internal : process (Bit_Clk) is
  -- creates a copy of the sync_i signal that is one clock cycle delay
  begin
    if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
      sync_i_1  <= sync_i;
end if;
  end process Shift_Sync_internal;

  -----------------------------------------------------------------------------
  -- Generating a 16 delay followed with continous 20 clock delays
  -----------------------------------------------------------------------------

  new_slot <= slot_end when last_slot = '0' else '0';

  -- counts 4 + 16 bit
  -- generates a delay_4 bit and then enables the 16 bit shift register
  -- slot_end will go '1' after 16 clock cycle from start sync and each 20
  -- clock cycle there after

  Delay_4_SRL16 : SRL16E
    -- 4 bit shift register
    generic map (
      INIT => X"0000")                -- [bit_vector]
    -- pragma translate_on
    port map (
      CE  => '1',                     -- [in  std_logic]
      D   => new_slot,                -- [in  std_logic]
      Clk => Bit_Clk,                 -- [in  std_logic]
      A0  => '1',                     -- [in  std_logic]
```

```
    A1  => '1',                         -- [in  std_logic]
    A2  => '0',                         -- [in  std_logic]
    A3  => '0',                         -- [in  std_logic]
    Q   => delay_4);                    -- [out std_logic]

con_slot <= (start_Sync or delay_4);  -- counts either start_Sync or delay4

Delay_16_SRL16 : SRL16E
  -- 16 bit shift register
  generic map (
    INIT => X"0000")                    -- [bit_vector]
  -- pragma translate_on
  port map (
    CE  => '1',                         -- [in  std_logic]
    D   => con_slot,                    -- [in  std_logic]
    Clk => Bit_Clk,                     -- [in  std_logic]
    A0  => '1',                         -- [in  std_logic]
    A1  => '1',                         -- [in  std_logic]
    A2  => '1',                         -- [in  std_logic]
    A3  => '1',                         -- [in  std_logic]
    Q   => slot_end);                   -- [out std_logic]

-----------------------------------------------------------------------------
-- Count 13 slot_ends to restart a new frame
-----------------------------------------------------------------------------

Slot_count_SRL16 : SRL16E
  -- pragma translate_off
  generic map (
    INIT => X"0000")                    -- [bit_vector]
  -- pragma translate_on
  port map (
    CE  => slot_end,                    -- [in  std_logic]
    D   => sync_i,                        -- [in  std_logic]
    Clk => Bit_Clk,                     -- [in  std_logic]
    A0  => '1',                         -- [in  std_logic]
    A1  => '1',                         -- [in  std_logic]
    A2  => '0',                         -- [in  std_logic]
    A3  => '1',                         -- [in  std_logic]
    Q   => last_slot);                  -- [out std_logic]

-- goes high only when detected last_slot and slot_end goes high
-- At Start up - last_slot = 0, slot_end = 0, and start_from_resest = 1 after first bit_clk :
start_Sync <= (last_slot and slot_end) or not start_from_reset;

-- use for start up to avoid ambigous states
start_from_reset_FD : FD
  port map (
```

```
    Q => start_from_reset,            -- [out std_logic]
    C => Bit_Clk,                     -- [in  std_logic]
    D => '1');                        -- [in std_logic]

-----------------------------------------------------------------------------
-- Handling the SData_Out
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- Controller for the Slot Number
-----------------------------------------------------------------------------
Slot_Cnt_Handle : process (Bit_Clk) is
begin  -- process Data_Out_Handle
  if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
    if (start_sync = '1') then -- Reset Slot number at the beginning of each frame
      slot_No <= 0;
    elsif (slot_end = '1') then -- Everytime it encounters a slot end, increments the counte
      if (slot_No < 4) then  -- only increment up to 4, the rest is useless
        slot_No <= slot_No + 1;
      end if;
    end if;
  end if;
end process Slot_Cnt_Handle;

-----------------------------------------------------------------------------
-- Set up data
-- Place appropriate data into new_data_out depending on which slot it is in
-- The process only generates one clock pulse of new_data_out so data don't overwrite each o
--   (start_sync and slot_end only goes high on bit_clk cycle)
--
-- Data value from OPB (slot#) is set up in the previous slot, and then it gets updated into
--    by detected a slot_end or start_sync (denotes start of a new frame)
--    and so data is shifted out in current slot.
-----------------------------------------------------------------------------
process (start_sync, slot_No, slot_end, slot0, slot1, slot2, In_Data_FIFO_i) is
begin  -- process
 -- reset all data first
  new_data_out <= (others => '0');
  read_fifo    <= '0';

  -- Slot 0, Tag
  if (start_sync = '1') then
    new_data_out(19 downto 4) <= slot0;
    read_fifo                 <= '0';
  -- Slot 1-4
  elsif (slot_end = '1') then
    read_fifo    <= '0';
    case slot_No is  -- slot_No is 1 behind because, when slot_end is detected, the slot_No
```

```
      when 0 => new_data_out(slot1'range) <= slot1;
      when 1 => new_data_out(slot2'range) <= slot2;
      when 2 =>
        if (C_PLAYBACK = 1) then  -- Left channel
          new_data_out(19 downto 4) <= In_Data_FIFO_i;
          read_fifo                 <= slot0(12);   -- use
        end if;
      when 3 =>
        if (C_PLAYBACK = 1) then   -- Right channel
          new_data_out(19 downto 4) <= In_Data_FIFO_i;
          read_fifo                 <= slot0(11);
        end if;
      when others => null;
    end case;
  end if;
end process;


-----------------------------------------------------------------------------
--  obselete, input FIFO no longer in use
--  The code is use in operation with the FSL, so this section becomes obselete
-----------------------------------------------------------------------------
Read_FIFO_DFF: process (Bit_Clk) is
begin  -- process Read_FIFO_DFF
  if Bit_Clk'event and Bit_Clk = '1' then    -- rising clock edge
    read_FIFO_1 <= read_fifo;
  end if;
end process Read_FIFO_DFF;

Reading_the_FIFO : process (Clk, Reset) is
  variable tmp   : std_logic;
  variable tmp_1 : std_logic;
begin  -- process Reading_the_FIFO
  if Reset = '1' then                 -- asynchronous reset (active high)
    in_FIFO_Read <= '0';
    tmp          := '0';
    tmp_1        := '0';
  elsif Clk'event and Clk = '1' then  -- rising clock edge
    in_FIFO_Read <= '0';
    if ((tmp_1 = '0' and tmp = '1')) then
      in_FIFO_Read <= '1';
    end if;
    tmp_1 := tmp;
    tmp   := read_FIFO_1;
  end if;
end process Reading_the_FIFO;

-----------------------------------------------------------------------------
```

```
  -- Shift Data out one by one
  -------------------------------------------------------------------------------
  Data_Out_Handle : process (Bit_Clk) is
  begin  -- process Data_Out_Handle
    if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
      SData_Out <= data_out(19);
      if (start_sync = '1') or (slot_end = '1') then  -- beginning of a frame or beginning of a
        data_out <= New_Data_Out;  -- update the slot to new information
      else
        data_out(19 downto 0) <= data_out(18 downto 0) & '0';  -- Shift information
      end if;
    end if;
  end process Data_Out_Handle;



  -------------------------------------------------------------------------------
  -- Handling data coming in
  -------------------------------------------------------------------------------

  -------------------------------------------------------------------------------
  -- Shift Sdata serially into data_in
  -------------------------------------------------------------------------------
  Shifting_Data_Coming_Back : process (Bit_Clk) is
  begin  -- process Shifting_Data_Coming_Back
    if Bit_Clk'event and Bit_Clk = '0' then  -- falling clock edge
      data_in(19 downto 0) <= data_in(18 downto 0) & SData_In;
    end if;
  end process Shifting_Data_Coming_Back;

  Shift_Slot_end : process (Bit_Clk) is
  begin
    if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
      slot_end_1    <= slot_end;
end if;
  end process Shift_Slot_end;

  -------------------------------------------------------------------------------
  --  Get Slot 0 data
  --  Because the one slot is complete one bit_clk cycle later, data is not register until the
  --    is finished
  -------------------------------------------------------------------------------
  Grabbing_Data_Coming_Back : process (Bit_Clk) is
  begin  -- process Grabbing_Data_Coming_Back
    if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
      if (sync_i_1 = '1' and slot_end_1 = '1') then
        codec_rdy           <= data_in(15); -- bit 15 Codec Ready
```

```
      data_valid          <= data_in(14);  -- Slot 1 data valid
       -- Slot 2 Status valid
      valid_Record_Data_L <= data_in(12);  -- Slot 3 L Audio valid
 valid_Record_Data_R <= data_in(11); -- Slot 4 R Audio valid
    end if;
  end if;
end process Grabbing_Data_Coming_Back;

-----------------------------------------------------------------------------
-- Get slot 1 data
-----------------------------------------------------------------------------
Get_Slot_1_Data : process (Bit_Clk) is
begin  -- process Get_Slot_1_Data
  if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
    if (slot_end_1 = '1' and slot_No = 2) then
      valid_Playback_Data_L <= not data_in(11);
      valid_Playback_Data_R <= not data_in(10);
    end if;
  end if;
end process Get_Slot_1_Data;

-----------------------------------------------------------------------------
-- Get slot 2 data
-----------------------------------------------------------------------------
Get_Reg_Read_Data : process (Bit_Clk) is
begin  -- process Get_Reg_Read_Data
  if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
    -- slot_end_1    <= slot_end;
    got_read_data <= '0';
    if (slot_end_1 = '1' and slot_No = 3 and data_valid = '1') then
      AC97_Reg_Read_Data <= data_in(19 downto 4);
      got_read_data <= '1';
    end if;
  end if;
end process Get_Reg_Read_Data;

-----------------------------------------------------------------------------
-- Get slot 3 and 4 data
-----------------------------------------------------------------------------
Get_Record_Data : process (Bit_Clk) is
begin  -- process Get_Record_Data
  if Bit_Clk'event and Bit_Clk = '1' then  -- rising clock edge
    got_record_data <= '0';
    if (C_RECORD = 1) then
      if (slot_end_1 = '1' and slot_No = 4 and valid_Record_Data_L = '1') then
        Out_Data_FIFO   <= data_in(19 downto 4);  -- output to FSL right away
        got_record_data <= '1';
      elsif (slot_end_1 = '1' and slot_No = 5 and valid_Record_Data_R = '1') then
```

```
        Out_Data_FIFO   <= data_in(19 downto 4);  -- output to FSL right away
        got_record_data <= '1';
      end if;
    end if;
  end if;
end process Get_Record_Data;


-------------------------------------------------------------------------------
-- Handshaking logic with input FIFO
-- PCM DAC data in, goes to a temporary FIFO to buffer the data
-------------------------------------------------------------------------------

Got_Record_Data_DFF : FDCE
  port map (
    Q   => write_fifo,                -- [out std_logic]
    C   => Bit_Clk,                   -- [in  std_logic]
    CE  => Got_Record_Data,           -- [in  std_logic]
    D   => '1',                       -- [in  std_logic]
    CLR => fifo_written);             -- [in std_logic]

Write_FIFO_Handle: process (Clk, Reset) is
  variable tmp   : std_logic;
  variable tmp_1 : std_logic;
begin  -- process Write_FIFO_Handle
  if Reset = '1' then                 -- asynchronous reset (active high)
    fifo_written <= '0';
    tmp   := '0';
    tmp_1 := '0';
  elsif Clk'event and Clk = '1' then  -- rising clock edge
    fifo_written <= '0';
    if ((tmp = '1') and (tmp_1 = '0')) then
      fifo_written <= '1';
    end if;
    tmp_1 := tmp;
    tmp := write_fifo;
  end if;
end process Write_FIFO_Handle;

Out_FIFO_Write <= fifo_written;

ac97_Reg_Finished_i <= (got_read_data and ac97_read_access) or  -- Read operation
                       (start_sync and ac97_write_access);  -- Write operation

Req_Finished_DFF : FDCE
  port map (
    Q   => ac97_Reg_Finished,         -- [out std_logic]
    C   => Bit_Clk,                   -- [in  std_logic]
```

```
      CE  => ac97_Reg_Finished_i,       -- [in  std_logic]
      D   => '1',                       -- [in  std_logic]
      CLR => AC97_Request_Finished);    -- [in std_logic]

end architecture IMP;

-------Next ----

library IEEE;
use IEEE.std_logic_1164.all;

entity opb_freq_gen is
  generic (
   C_freq_spec_bit : integer :=16  -- 5 use 5 bit for sim purpose
  );
  port (
-- signals
Data_out : out std_logic_vector(0 to 15);  -- shift out 16 bit data
Bit_clk : in std_logic; -- bit_clk from the ac97
Enable : in std_logic
);

end entity opb_freq_gen;

library unisim;
use unisim.all;

architecture IMP of opb_freq_gen is

  component down_counter is
   generic (
  C_bit : integer
);
port (
  clock: in std_logic;
  preset: in std_logic;
  count: in std_logic;
  Q: out std_logic_vector(C_bit-1 downto 0);
  preset_value : in std_logic_vector(C_bit-1 downto 0)
);
  end component down_counter;

  component FD is
    -- pragma translate_off
    generic (
      INIT : bit := '0'
      );
    -- pragma translate_on
```

```
    port (
      Q : out std_logic;
      C : in  std_logic;
      D : in  std_logic
      );
  end component FD;


  -- constants for frequencies  -- "11011" for simulation
  constant NOTE_A : std_logic_vector(C_freq_spec_bit-1 downto 0) := X"3688";  -- note_A is 440H

  signal wave_out_i : std_logic := '0';   -- set to '0' as initial condition for simulation pu
  signal rst_i : std_logic := '0';   -- internal reset that resets the counter when it toggles
  signal count_out : std_logic_vector(C_freq_spec_bit-1 downto 0);
  signal start_from_reset : std_logic;
  signal clr : std_logic;


begin   -- architecture

  -----------------------------------------------------------------------------
  -- Output frequency Mux
  -----------------------------------------------------------------------------

  -----------------------------------------------------------------------------
  -- Signal conversion, from 1010 to 18 bit per sample
  -----------------------------------------------------------------------------
  Convert_to_16bit_signal : process (wave_out_i) is
  begin
   if wave_out_i = '1' then
Data_out <= X"2000";
else
Data_out <= X"0008";
end if;
  end process Convert_to_16bit_signal;

  -----------------------------------------------------------------------------
  -- Generate Reset Signal, toggle data_out
  -----------------------------------------------------------------------------

  -- use to initialize signal in case of unstable
  start_from_reset_FD : FD
    port map (
      Q => start_from_reset,            -- [out std_logic]
      C => Bit_Clk,                     -- [in  std_logic]
      D => '1');

  rst_i <= (not start_from_reset) or clr;
```

```
  -- if Q = delay_value, then set clr to high, sample the value at falling edge
  Clear_pulse : process (count_out, Bit_clk) is
  begin
   if Bit_clk'event and Bit_clk = '0' then   -- setup at falling edge
   if (count_out = "00000") then
clr <= '1';
else
clr <= '0';
end if;
end if;
  end process Clear_pulse;

  Toggle_wave_out : process (rst_i, Bit_clk) is
  begin
   if Bit_clk'event and Bit_clk = '1' then   -- rising edge
   if rst_i = '1' then
wave_out_i <= not wave_out_i;
end if;
end if;
  end process Toggle_wave_out;

  -----------------------------------------------------------------------------
  -- instantiation of counter
  -----------------------------------------------------------------------------

  Sound_Freq_Counter : down_counter -- synchronous clear
   generic map (
C_bit => C_freq_spec_bit -- [integer]
)
   port map (
  clock => Bit_clk,-- [in std_logic]
  preset => rst_i,-- [in std_logic]
  count => Enable,-- [in std_logic]
  Q    => count_out,-- [out std_logic_vector(C_bit-1 downto 0)]
  preset_value => NOTE_A); -- [in std_logic_vector(C_bit-1 downto 0)]

end architecture IMP;

-------- Next --------

-------------------------------------------------------------------------------
-- $Id: pselect.vhd,v 1.2 2003/02/07 20:23:35 goran Exp $
-------------------------------------------------------------------------------
-- pselect.vhd - entity/architecture pair
-------------------------------------------------------------------------------
--
--                  ***************************
```

```
--                  ** Copyright Xilinx, Inc. **
--                  ** All rights reserved.   **
--                  ****************************
--
-------------------------------------------------------------------------------
-- Filename:        pselect.vhd
--
-- Description:     Parameterizeable peripheral select (address decode).
--                  AValid qualifier comes in on Carry In at bottom
--                  of carry chain. For version with AValid at top of
--                  carry chain, see pselect_top.vhd.
--
-- VHDL-Standard:   VHDL'93
-------------------------------------------------------------------------------
-- Structure:
--                  pselect.vhd
--
-------------------------------------------------------------------------------
-- Author:          B.L. Tise
-- Revision:        $Revision: 1.2 $
-- Date:            $Date: 2003/02/07 20:23:35 $
--
-- History:
--   BLT            2001-04-10   First Version
--   BLT            2001-04-23   Moved function to this file
--   BLT            2001-05-21   Changed library to MicroBlaze
--   BLT            2001-08-13   Changed pragma to synthesis
--   ALS            2001-10-15   C_BAR is now padded to nearest multiple of 4
--                               to handle lut equations
--   FLO            2002-03-26   Corrected implementation for case where C_AB
--                               is not a multiple of 4 and the C_BAR values
--                               at the pad bits are not '0'.
--                               Removed implementation restriction that
--                               required C_AW = C_BAR'length.
--                               Added assertion to flag invalid generic
--                               combinations.

-- ALS, FLO 2002-04-09 -Implemented XST workaround for the case
-- that C_AB = 0.
-- -Removed remnants of earlier
-- "instantiated-lut" implementation.
-------------------------------------------------------------------------------
-- Naming Conventions:
-- active low signals: "*_n"
-- clock signals: "clk", "clk_div#", "clk_#x"
-- reset signals: "rst", "rst_n"
-- generics: "C_*"
-- user defined types: "*_TYPE"
```

```
-- state machine next state: "*_ns"
-- state machine current state: "*_cs"
-- combinatorial signals: "*_com"
-- pipelined or register delay signals: "*_d#"
-- counter signals: "*cnt*"
-- clock enable signals: "*_ce"
-- internal version of output port "*_i"
-- device pins: "*_pin"
-- ports: - Names begin with Uppercase
-- processes: "*_PROCESS"
-- component instantiations: "<ENTITY_>I_<#|FUNC>
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

library unisim;
use unisim.all;

-----------------------------------------------------------------------------
-- Entity section
-----------------------------------------------------------------------------
-------------------------------------------------------------------------------
-- Definition of Generics:
-- C_AB                                 -- number of address bits to decode
--          C_AW                        -- width of address bus
--          C_BAR                       -- base address of peripheral (peripheral select
--                             is asserted when the C_AB most significant
--                             address bits match the C_AB most significant
--                             C_BAR bits
-- Definition of Ports:
--          A                           -- address input
--          AValid                      -- address qualifier
--          PS                          -- peripheral select
-------------------------------------------------------------------------------

entity pselect is

  generic (
    C_AB  : integer := 9;
    C_AW  : integer := 32;
    C_BAR : std_logic_vector(0 to 31) := X"FFFF_8000"
    );
  port (
    A      : in  std_logic_vector(0 to C_AW-1);
    AValid : in  std_logic;
    ps     : out std_logic
```

```
    );

end entity pselect;

-----------------------------------------------------------------------------
-- Architecture section
-----------------------------------------------------------------------------

architecture imp of pselect is

  component MUXCY is
    port (
      O  : out std_logic;
      CI : in  std_logic;
      DI : in  std_logic;
      S  : in  std_logic
      );
  end component MUXCY;

  attribute INIT : string;

-----------------------------------------------------------------------------
-- Constant Declarations
-----------------------------------------------------------------------------
  constant NUM_LUTS : integer := (C_AB+3)/4;

  -- C_BAR may not be indexed from 0 and may not be ascending;
  -- BAR recasts C_BAR to have these properties.
  constant BAR : std_logic_vector(0 to C_BAR'length-1) := C_BAR;

-----------------------------------------------------------------------------
-- Signal Declarations
-----------------------------------------------------------------------------

--signal lut_out : std_logic_vector(0 to NUM_LUTS-1);
  signal lut_out : std_logic_vector(0 to NUM_LUTS);  -- XST workaround

  signal carry_chain : std_logic_vector(0 to NUM_LUTS);


-------------------------------------------------------------------------------
-- Begin architecture section
-------------------------------------------------------------------------------
begin

--------------------------------------------------------------------------------
-- Check that the passed generics allow for correct implementation.
--------------------------------------------------------------------------------
```

```
-- synthesis translate_off
  assert (C_AB <= C_BAR'length) and (C_AB <= C_AW)
    report "pselect generic error : " &
    "(C_AB     <= C_BAR'length) and (C_AB <= C_AW)" &
    " does not hold."
    severity failure;
-- synthesis translate_on


--------------------------------------------------------------------------------
-- Build the decoder using the fast carry chain.
--------------------------------------------------------------------------------

  carry_chain(0) <= AValid;

  XST_WA : if NUM_LUTS > 0 generate     -- workaround for XST; remove this
                                        -- enclosing generate when fixed
    GEN_DECODE : for i in 0 to NUM_LUTS-1 generate
      signal lut_in : std_logic_vector(3 downto 0);
      signal invert : std_logic_vector(3 downto 0);
    begin
      GEN_LUT_INPUTS : for j in 0 to 3 generate
        -- Generate to assign address bits to LUT4 inputs
        GEN_INPUT : if i < NUM_LUTS-1 or j <= ((C_AB-1) mod 4) generate
          lut_in(j) <= A(i*4+j);
          invert(j) <= not BAR(i*4+j);
        end generate;
        -- Generate to assign one to remaining LUT4, pad, inputs
        GEN_ZEROS : if not(i < NUM_LUTS-1 or j <= ((C_AB-1) mod 4)) generate
          lut_in(j) <= '1';
          invert(j) <= '0';
        end generate;
      end generate;

      --------------------------------------------------------------------------
      -- RTL LUT instantiation
      --------------------------------------------------------------------------
      lut_out(i) <= (lut_in(0) xor invert(0)) and
                    (lut_in(1) xor invert(1)) and
                    (lut_in(2) xor invert(2)) and
                    (lut_in(3) xor invert(3));


      MUXCY_I : MUXCY
        port map (
          O  => carry_chain(i+1),       --[out]
          CI => carry_chain(i),         --[in]
          DI => '0',                    --[in]
```

```
          S  => lut_out(i)              --[in]
          );

    end generate GEN_DECODE;
  end generate XST_WA;

  ps <= carry_chain(NUM_LUTS);          -- assign end of carry chain to output;
  -- if NUM_LUTS=0, then
  -- PS <= carry_chain(0) <= AValid

end imp;

--- Next ----

----------------------------------------------------
-- VHDL code for n-bit counter
-- by Simon So, 04/2004
--
-- this is the behavior description of n-bit counter
-- another way can be used is FSM model.
----------------------------------------------------

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

----------------------------------------------------

entity down_counter is

  generic(
C_bit  : integer
  );

  port(
clock : in std_logic;
preset : in std_logic;
count : in std_logic;   -- enable for counter
Q : out std_logic_vector(C_bit-1 downto 0);    -- output bus
preset_value : in std_logic_vector(C_bit-1 downto 0)
  );

end down_counter;

----------------------------------------------------

architecture Behavioral of down_counter is
```

```
    signal Q_i: std_logic_vector(C_bit-1 downto 0);

begin

    -- behavior describe the counter
    process(clock, count)
    begin
if (clock='1' and clock'event) then  -- rising edge
     if preset = '1' then   -- sync reset
       Q_i <= preset_value;
     elsif count = '1' then
Q_i <= Q_i - 1;
     end if;
end if;
    end process;

    -- concurrent assignment statement
    Q <= Q_i;

end Behavioral;

-----------------------------------------------------
```

C C code

```
#include "xgpio.h"          /* layer 1 GPIO device driver                   */


XGpio Gpio;

/* this would be used to initialize the GPIO */
Status = XGpio_Initialize(&Gpio, XPAR_OPB_GPIO_0_DEVICE_ID);
  if (Status != XST_SUCCESS)
  {
       printf("GPIO initialization error\n\r\r");
 }

 /* This function just tells the video core to grab one frame*/
void TakeFrame()
{
XGpio_DiscreteWrite(&Gpio, 1);
}
/* This function ensures that the video core is turned off*/
void VideoOff()
```

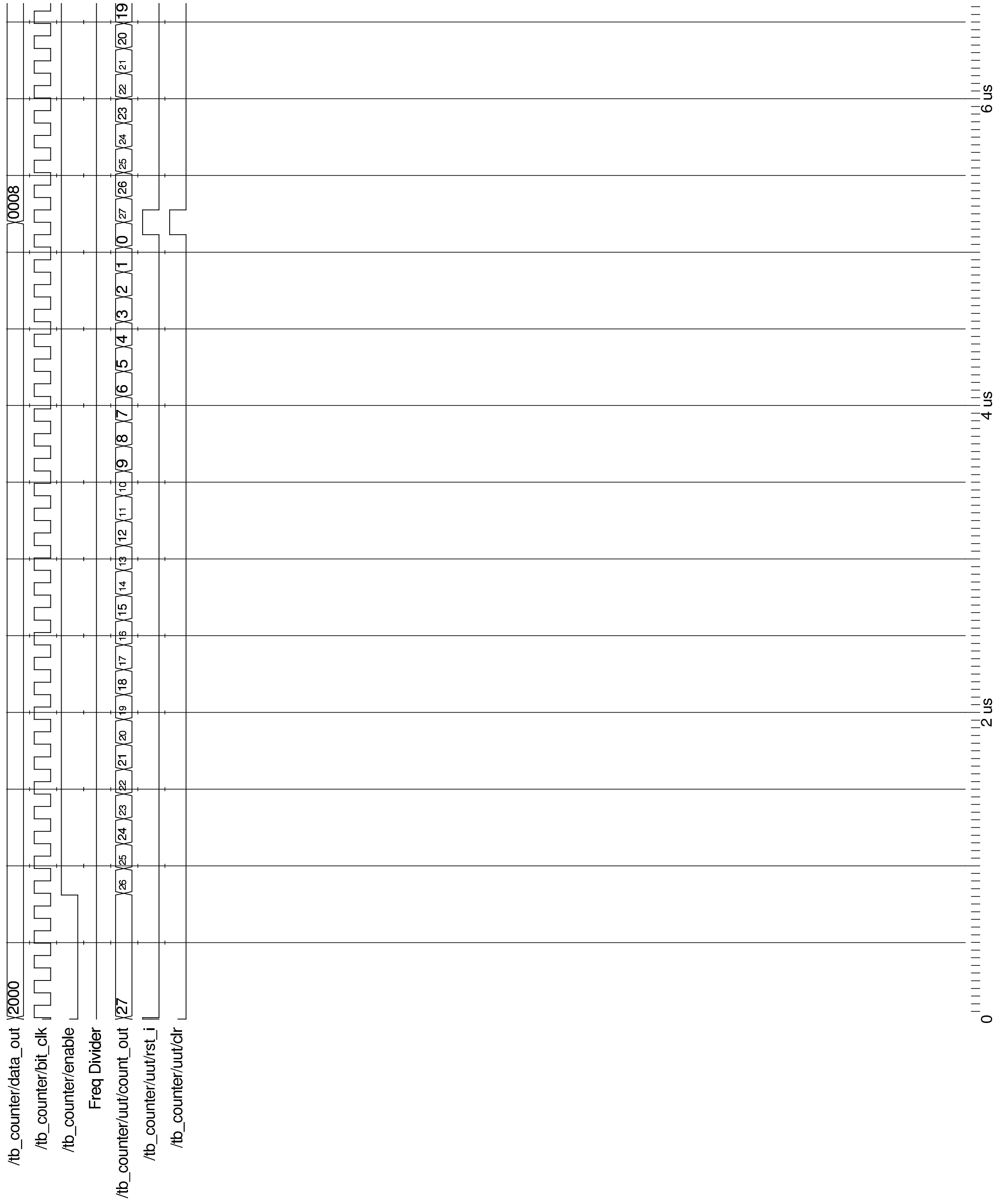
```
{
XGpio_DiscreteWrite(&Gpio, 6);
}

/* Below is Code that would be used to find the average of a given colour in an image
  It only scans one particular line */

#define rowno 239
#define pixels 320

/* this finds the average of the colour red*/
int findred(*picbaseaddy)  /* The base address of the image needs to be passed */
{
 int pixel;    /* the value of the colour portion of the given pixel */
 int avg = 0 ;  /* the running average */
  for(i=0;i<pixels;i++)  /* sum over one row */
    {
    pixel = (*(picbaseaddy+rowno*pixels*1+i*1) & 0xFF000000) >> 24 ;  // retrieve from memory a
    avg += ( pixel & 0x000000FF );  // when shifting 1's get padded in, these are here removed
    }
  avg /= pixels;
 return avg;  /returns the average
}


/* same as above but for blue */
int findblue(*myaddy)
{
 int pixel;
 int avg = 0 ;
  for(i=0 ; i<pixels ; i++)
    {
    pixel = (*(picbaseaddy+rowno*pixels*1+i*1) & 0xFF000000) >> 24 ;
    avg += ( pixel & 0x000000FF );
    }
  avg /= pixels;
 return avg;

}

/* same as above but for green */
int findgreen(*myaddy)
{
 int pixel;
 int avg = 0 ;
  for(i=0 ; i<pixels ; i++)
    {
    pixel = (*(picbaseaddy+rowno*pixels*1+i*1) & 0xFF000000) >> 24 ;
```

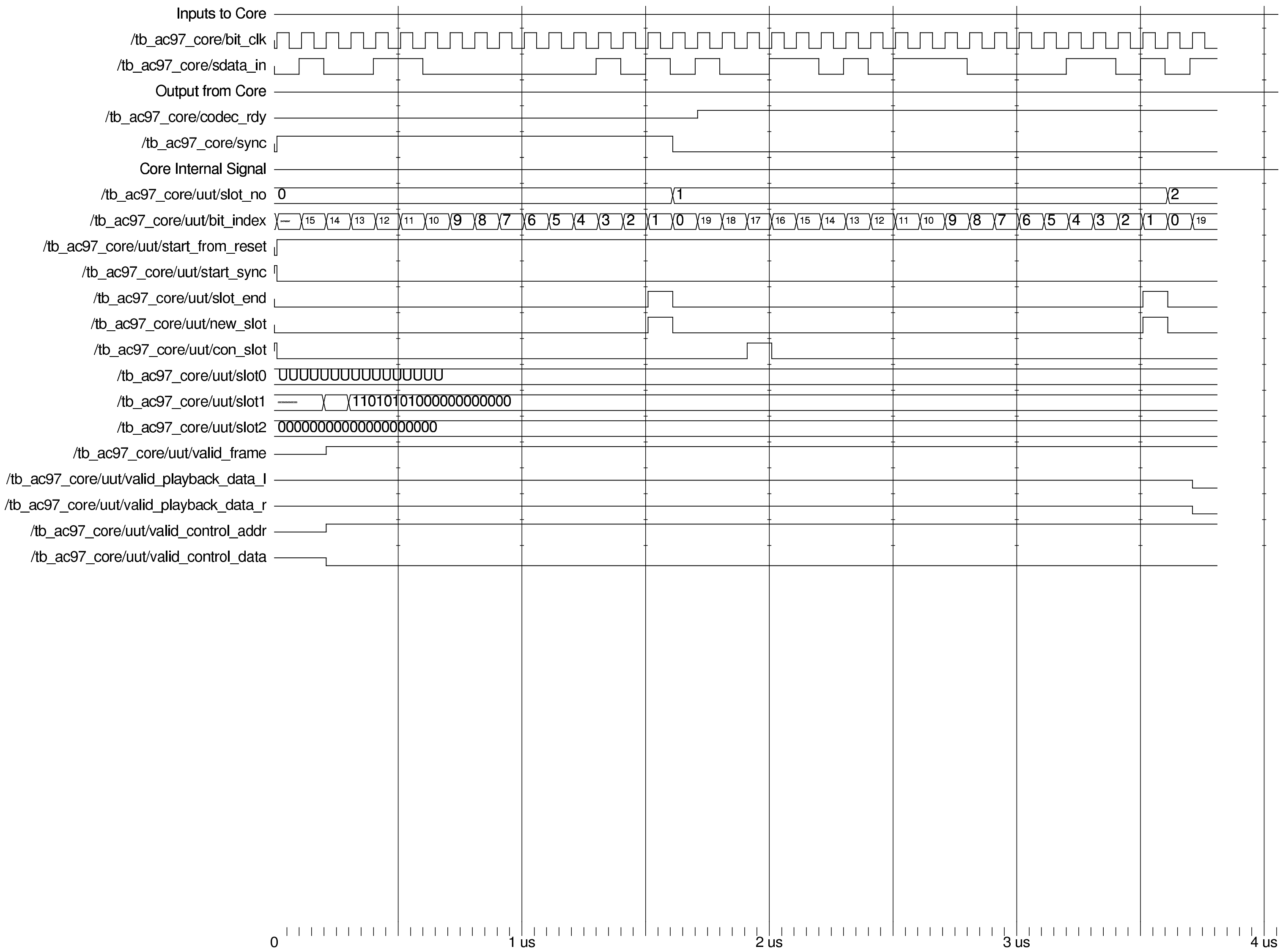
```
    avg += ( pixel & 0x000000FF );
    }
 avg /= pixels;
 return avg;
}
```

D Simulations

E Specifications Sheets

Relevant spec sheets to this project could not be included. Please see the LM4549A audio controller, and the ADV7194 for the video encoder and ADV7185 for the video decoder. They are all available easily on the web in copyrighted PDF format.


/tb_counter/data_out
2000
0008
/tb_counter/bit_clk
/tb_counter/enable
Freq Divider
/tb_counter/uut/count_out
27
/tb_counter/uut/rst_i
/tb_counter/uut/clr
0
2 us
4 us
6 us
Entity:tb_counter Architecture:testbench_arch Date: Mon Apr 12 15:14:42 Eastern Standard Time 2004 Row: 1 Page: 1


Inputs to Core
/tb_ac97_core/bit_clk
/tb_ac97_core/sdata_in
Output from Core
/tb_ac97_core/codec_rdy
/tb_ac97_core/sync
Core Internal Signal
/tb_ac97_core/uut/slot_no
/tb_ac97_core/uut/bit_index
/tb_ac97_core/uut/start_from_reset
/tb_ac97_core/uut/start_sync
/tb_ac97_core/uut/slot_end
/tb_ac97_core/uut/new_slot
/tb_ac97_core/uut/con_slot
/tb_ac97_core/uut/slot0
/tb_ac97_core/uut/slot1
/tb_ac97_core/uut/slot2
/tb_ac97_core/uut/valid_frame
/tb_ac97_core/uut/valid_playback_data_l
/tb_ac97_core/uut/valid_playback_data_r
/tb_ac97_core/uut/valid_control_addr
/tb_ac97_core/uut/valid_control_data
UUUUUUUUUUUUUUUU
11010101000000000000
00000000000000000000
0
1 us
2 us
3 us
4 us
Entity:tb_ac97_core Architecture:testbench_arch Date: Mon Apr 12 14:32:03 Eastern Standard Time 2004 Row: 1 Page: 1