Optional Semester Project in Computer Science

Extension board for CycloneV
Multi microphone acquisition - Signal Analysis

Extending the Pyramic Array

Author:
Corentin Ferry (EPFL)
Supervision:
René Beuchat (LAP, EPFL)
Robin Scheibler (LCAV, EPFL)

Autumn Semester, 2016

Processor Architecture Laboratory (LAP)
Audiovisual Communications Laboratory (LCAV)
Swiss Federal Institute of Technology, Lausanne (EPFL)

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank warmly my instructors René and Robin for unblocking me numerous times, and helping me creating a full VHDL design for the first time!

Also, I would like to thank Juan Azcarreta Ortiz who accepted to spend a long time talking with me and explaining many details about his freshly completed design.

1 Introduction

1.1 Motivation

Digital signal processing algorithms are mainly run on high-end platforms that involve powerful processors to compute Fourier transforms, feature extraction or analysis. Recent applications have seen the introduction of dedicated digital signal processing platforms as embedded systems with numerous sensors and powerful processors to perform complex computations on the signal.

Juan Azcareta Ortiz's work led to the construction of the Pyramic Array [1] as his Master's thesis. The base Pyramic array is intended to catch the same audio signal in many directions. Such a device has an interest for instance for physicists, to compute the actual direction of an audio signal as well as the geometry of the geometry of a room.

The existing Pyramic array is the base of our work.

This hardware design is a 48-microphone tetrahedric array, which comprises 8 microphones per edge (see fig. 3). Custom pieces of hardware have been designed in order to build such a system. The original design implemented on a Field-programmable Gate Array (FPGA) is made of:

- A control and capture part, that controls analog-to-digital converters which actually perform the sampling from the $n_{mic} = 48$ microphones, and is able to store the samples in a dedicated memory through a Direct Memory Access controller,
- A non-functional accelerator, that does a beamforming operation, which is divided into two parts:
  - Digital filtering: for one sequence of samples (from *one microphone*), given a finite sequence of coefficients $(\alpha_i)_{0 \leqslant i \leqslant N}$ and a sequence $(x_i)_{0 \leqslant i \leqslant N}$ of samples, the digital filter computes $F = \sum_{i=0}^{N} \alpha_i x_i$,
  - Beamforming: the filtering results for all the microphones are summed, thus we compute $\sum_{k=0}^{n_{mics}} F_k$,

  Since the digital filtering part is not done, this chain is not yet functional.

1.2 Aim of the project

The aim of this project is to create components to have a dataflow from the microphones to the sound output of the DE1-SoC board, that enables the user to run custom algorithms on the input signal from the 48 microphones.

Figure 1 shows an execution flow that is made possible through the project.

We want an application developer to be able to exploit the features of the Pyramic array as well. In order to achieve that, a library interfacing the hardware is needed.

1.3 How this report is organized

The conception of this report follows the two main axes of the project : we modified the existing FPGA design and added new components to the design.

The first section describes the hardware components that we have used in this project; it also provides a brief description of the work done by Juan Azarreta Ortiz that served as a base for this project.

The second section presents the modifications that we have done in the hardware design; it provides comprehensive information on the components that have been added to the design.

The third section presents the software we have written to interface the hardware design with, which consists in a library and a test program for this library.

2 The system : the Pyramic Array

The system developed in Juan Azcareta Ortiz's master thesis is fully described in his report [1]. However, given that we redefined some parts of the system, some of the hardware components should also be described in this report.

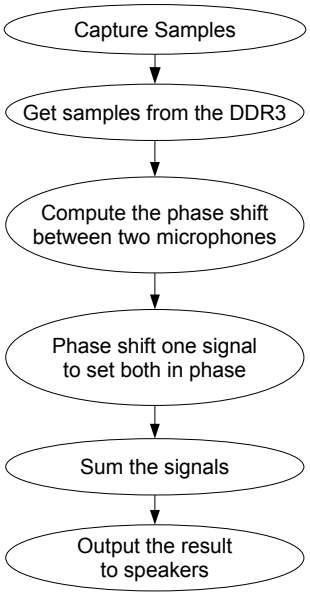

Figure 1: Example of an execution flow performed with the Pyramic array

2.1 Overview of the hardware components

During this project, we used the design that had been done by Juan Azcarreta Ortiz. This design includes two hardware pieces:

- A Terasic DE1-SoC board,
- Custom boards of 8 microphones + 1 ADC

The components we have used are subject to a description in this section. Figure 2 shows the Pyramic array as it is assembled.

2.1.1 The microphone boards

These microphone boards are made of 8 microphones per board plus an ADC.

The design of these microphones is part of Juan Azcarreta Otiz's thesis [1]. A built microphone board is shown on figure 3 .

2.1.2 The DE1-SoC board

This board, made by Terasic, embodies numerous hardware controllers plus a system-on-a-chip (SoC) made of a hardware processor (ARM) and an FPGA (Altera Cyclone V).

We can also find on this board :

- A Cyclone V SoC (HPS + FPGA)
- Two GPIO connectors with xx GPIOs each,
- A Wolfson WM8731 CODEC.

A full description of every component can be found in Terasic's documentation. Figure 4 shows an external view of this board.

2.1.3 The Analog Devices AD7606 Analog to Digital Converters

On each board, there is one AD7606 analog-to-digital converter. This chip is intended to be used on audio applications; its role is to sample the voltage at the microphones on the board, and output a digital value corresponding to that voltage for each of them.

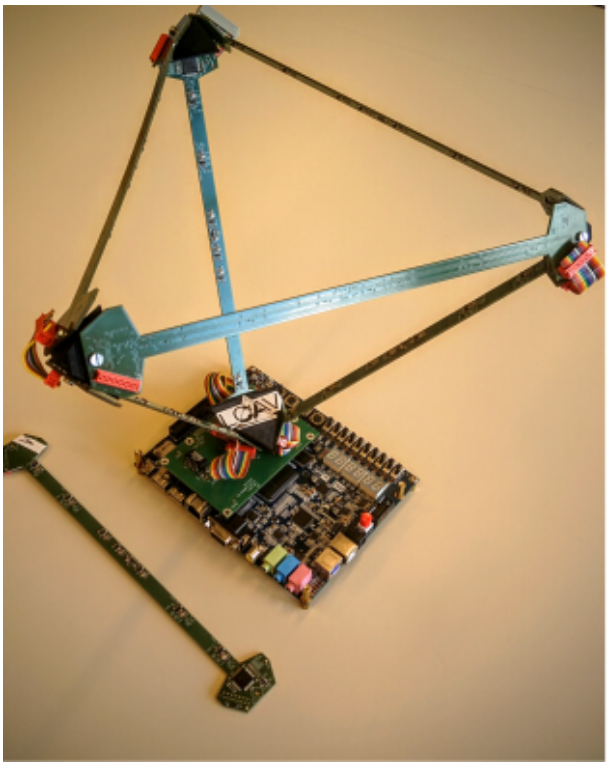

Figure 2: The Pyramic array [1]

Figure 3: Microphone board with an ADC and 8 microphones [1]

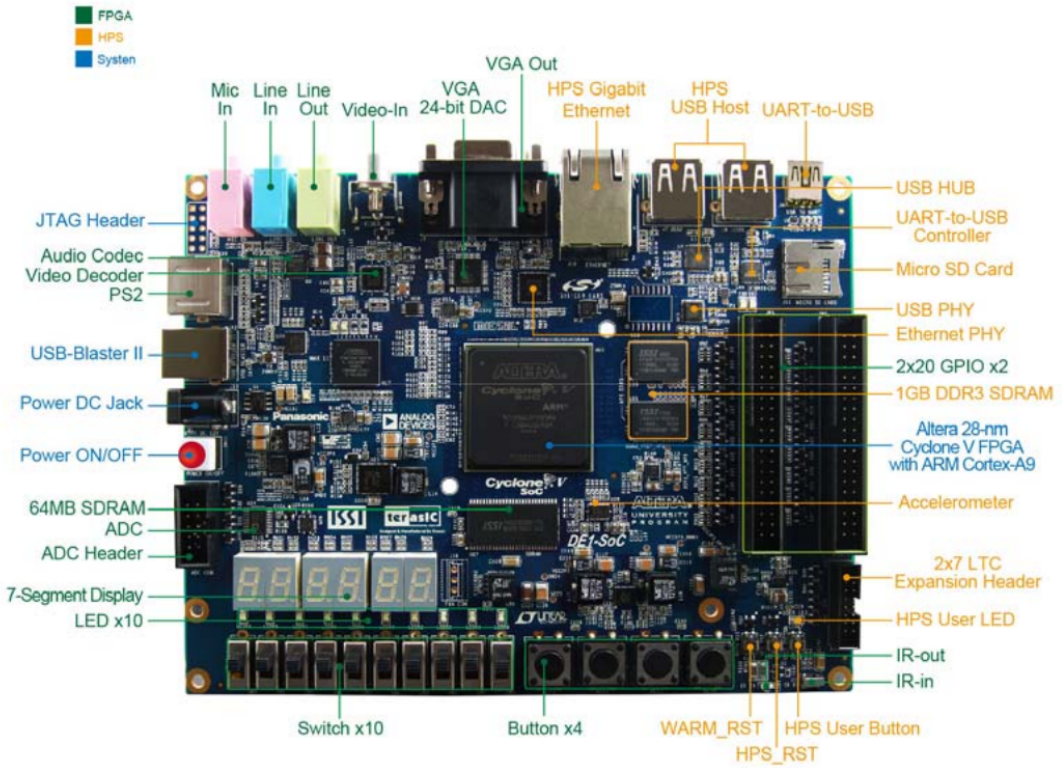

Figure 4: The DE1-SoC board [3]

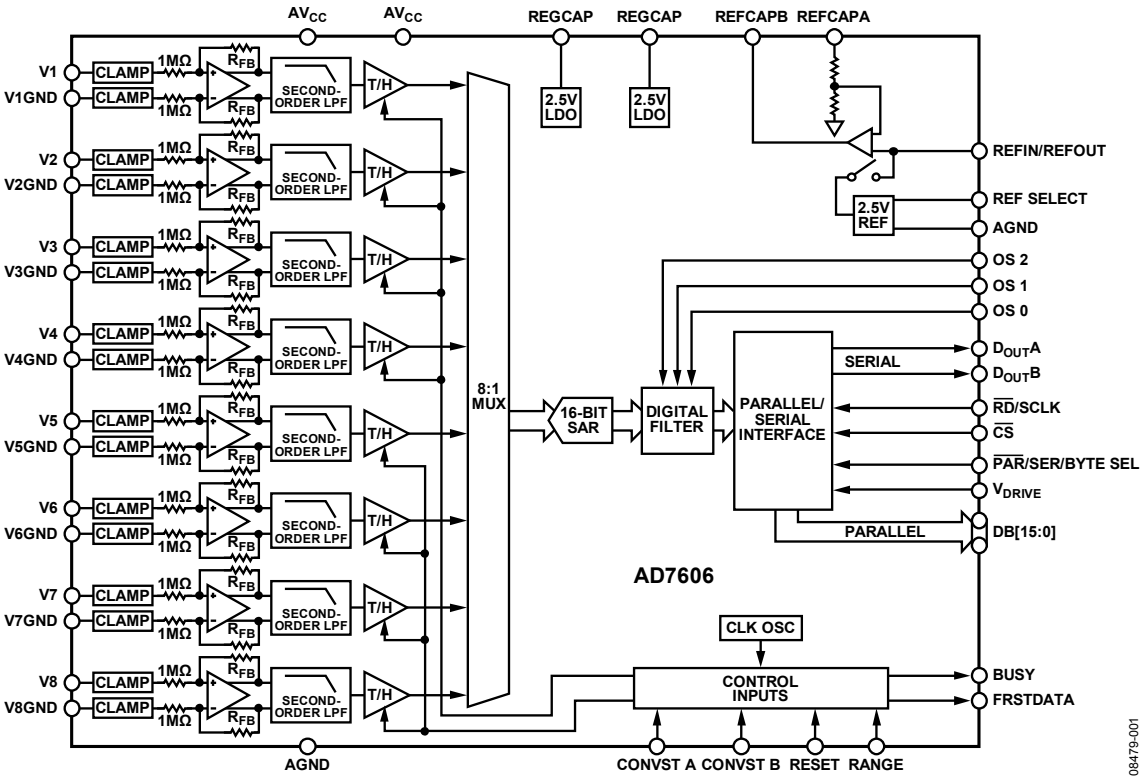

Figure 5: Block schematic for the AD7606 ADC [4]

Signal Name	FPGA Pin	Description	I/O Standard
AUD_ADCLRCK	PIN_K8	Audio CODEC ADC LR Clock	3.3V
AUD_ADCDAT	PIN_K7	Audio CODEC ADC Data	3.3V
AUD_DACLRCK	PIN_H8	Audio CODEC DAC LR Clock	3.3V
AUD_DACDAT	PIN_J7	Audio CODEC DAC Data	3.3V
AUD_XCK	PIN_G7	Audio CODEC Chip Clock	3.3V
AUD_BCLK	PIN_H7	Audio CODEC Bit-stream Clock	3.3V
I2C_SCLK	PIN_J12 or PIN_E23	I2C Clock	3.3V
I2C_SDAT	PIN_K12 or PIN_C24	I2C Data	3.3V

Table 1: Signals available from the exterior of the Wolfson codec [3]

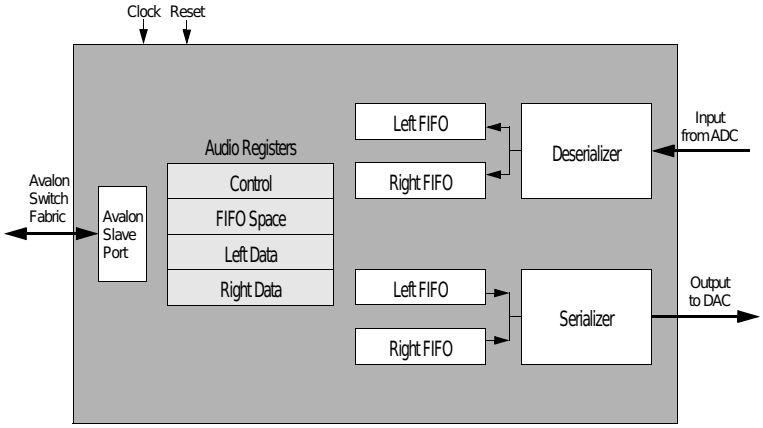

Figure 6: Block schematic for the Altera IP audio controller [5]

The block schematic for this ADC is shown on figure 5.

The choice and setup of this codec is part of Juan Azcarreta's thesis. We used the same setup parameters as Juan Azcarreta, namely:

- Sampling frequency: $f_s = 48000$ Hz
- Oversampling: OS $= 4 =$ "100"
- Sample resolution: 16 bits per sample
- Reading through the Serial interface, using the signal $\texttt{DOUT}_A$.

2.2 Hardware components implemented on the Field Programmable Gate Array

A large part of the components inside the design are actually implemented on the FPGA. This section describes these components.

2.2.1 Altera University Program IP Audio Controller, Audio and Video Configuration

The DE1-SoC board has an audio codec *Wolfson WM8731*. This codec contains a Digital-to-Analog converter (DAC), an ADC, and is controlled using the I2C protocol.

Table 1 shows the signals available from the exterior of the codec.

Altera provides, in its University program, an IP component that generates the input signal for the codec DAC. Its block schematic is shown on figure 6.

Altera also provides an IP component that automatically configures and initializes the Wolfson codec to match the parameters of the audio signal we are giving. This spares us from writing a dedicated controller to output sound.

Signal name	Width	Direction	Role
`Ready`	1	Sink → Source	Indicates whether the Avalon Streaming sink can accept input data.
`Data`	16	Source → Sink	Input data for the sink.
`Valid`	1	Source → Sink	Indicates whether the data found on the Data bus is valid.

Table 2: Signals used by the Avalon Streaming source and sink interfaces

Signal	Direction	Description
`SCLK`	Master → Slave	The clock that drives the SPI bus
`Slave Select`	Master → Slave	Equivalent to a Chip Select, this signal allows to select one slave among several. The master must have as many lines for this signal as there are slaves.
`MISO`	Slave → Master	Master In, Slave Out data line
`MOSI`	Master → Slave	Master Out, Slave In data line

Table 3: Serial Peripheral Interface signals

Used together, these two components allow us to output sound through the DE1-SoC codec by just providing data to the Avalon streaming sinks of the Audio controller. We are only using the `Ready`, `Data` and `Valid` signals from the Avalon sink reference, which are enough to output data to the audio controller. Table 2 shows the role of these signals.

Since we are outputting 16-bit words, which is the width of the data bus, we don't need the Start of Packet and End of Packet signals that the Avalon Streaming Interface provides [2].

We instantiated an audio controller as well as an audio configurator with the following parameters:

- Bit depth: 16 bits
- Interface: Avalon Streaming
- Sampling rate: 48000 Hz

Note that instatiating these components also requires to instantiate a dedicated Phase-locked loop, with no settings, that accompanies the audio core; this phase-locked loop generates the clock signal used by the audio core at 12.288 MHz.

2.2.2 Altera IP Phase-locked loops

A Phase-locked loop (PLL) is a hardware device that generates periodic signals given another periodic signal as an input. Through the use of a feedback loop, a PLL can lock itself to the phase of the input signal and ensure a relation between the instantaneous phase of the input and that of the output.

A PLL has oscillators, that are used in order to multiply the frequency of the signal. Counters are used to divide it; this allows to perform a multiplication of the input frequency by a given rational number.

PLLs used in FPGAs are specifically used for the multiplication and division of the frequency of a clock signal, while ensuring that the signals are in phase.

Altera IP PLLs have a single input signal (refclk) and can output up to 9 clocks on a Cyclone V device [7]. It can optionally output a signal that indicates whether the lock has been acquired, and a counter which is used by the clock divider for fractional PLLs.

2.2.3 Serial Peripheral Interface controller

The core design of Juan Azcaretta Ortiz's Master's thesis is a Serial Peripheral Interface (SPI) controller. The SPI bus communication between a master and a slave involves four kind of signals, as table 3 describes.

The communication between the master and the slave is full-duplex : this means that both the master and the slave send bits at the same time, on the rising edge of SCLK.

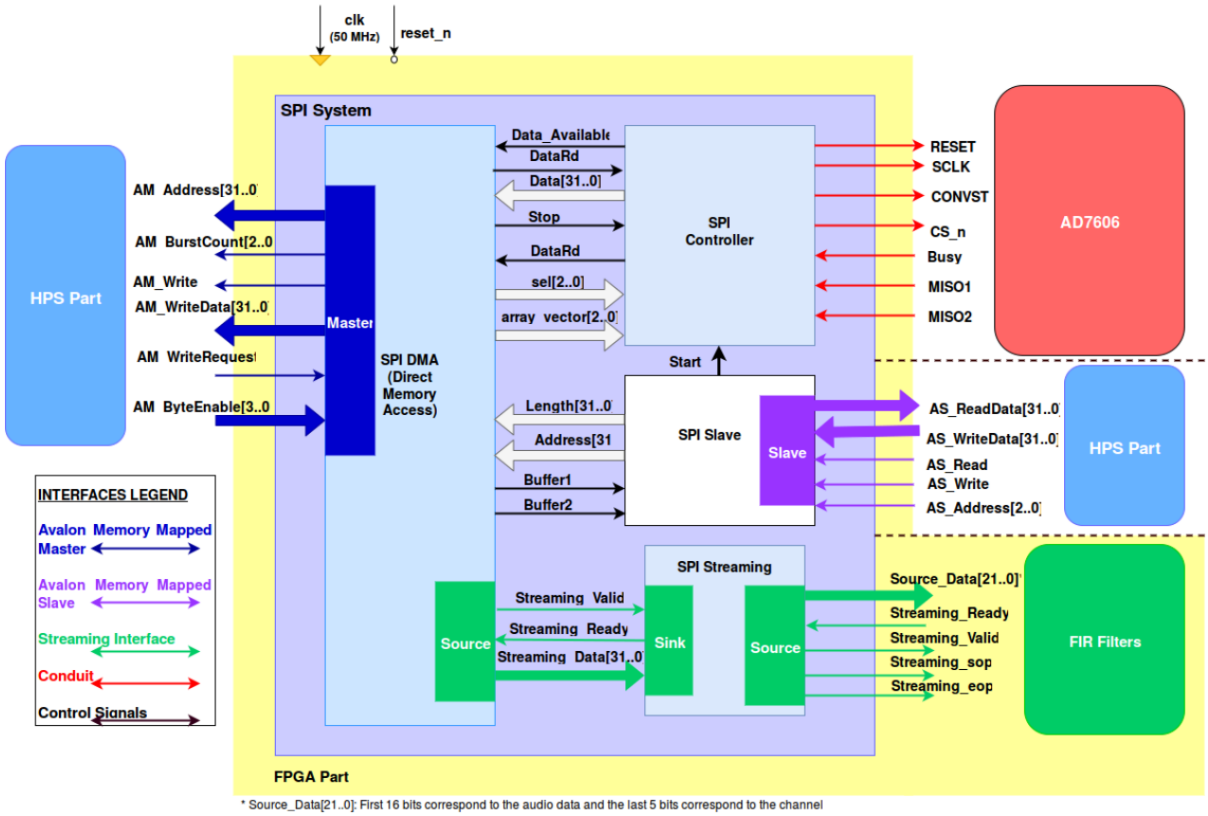

Figure 7: The SPI system [1]

The binary values are read and written using shift registers of a fixed length, which is the same for both the master and the slave. A single master can address multiple slaves with shared MOSI and MISO buses, through the use of the Slave Select signal.

The ADCs output audio data using this bus. For this reason, an SPI controller had to be designed in order to take the samples from the ADC, and feed them to a DMA controller. These two components were created as part of Juan Azcarreta Ortiz's work. Figure 7 shows the design he created.

2.2.4 The Hardware Processor System (HPS)

The Hardware Processor System is embedded inside the System-on-a-Chip (SoC) that the DE1-SoC provides. It runs a Linux system.

The HPS is made of many components amongst which [3] :

- An ARM dual-core processor (that runs the Linux system),
- A 1GB DDR3 memory chip,
- An HPS-to-FPGA bridge, implemented as an Advanced eXtensible Interface (AXI) master.

The HPS is linked to the FPGA through an HPS-to-FPGA bridge: it includes an AMBA design implemented as an AXI master [3]; in order to connect this master to Avalon slaves, an Avalon interconnect is introduced in Qsys.

In order for masters implemented on the FPGA to be able to pass data to the HPS through its memory, we also used an FPGA-to-HPS bridge that connects to the DDR3 memory through an SDRAM interface. Thus, the Direct Memory Access masters we will implement will have direct access to the HPS memory.

Signal	Direction	Role
`CONVST`	Master → Slave	The "conversion start" signal. Both $\texttt{CONVST}_A$ and $\texttt{CONVST}_B$ for all the ADCs are set to this signal.
`CSn`	Master → Slave	The "chip select" signal; asserting it starts the readback of the audio data from the ADC.
`SCLK`	Master → Slave	The clock for the SPI bus. **The data is valid on falling edges of this signal.**
$\texttt{MISO}_i$, $i \in \{1, \ldots, 6\}$	Slave → Master	The data from the ADCs.
`BUSY`	Slave → Master	Indicates whether a conversion is in process. Since we are using 6 ADCs simultaneously, this signal is the computed OR of all the BUSY signals from all ADCs.
`RESET`	Master → Slave	Resets the ADC.

Table 4: Signals used for capturing audio data

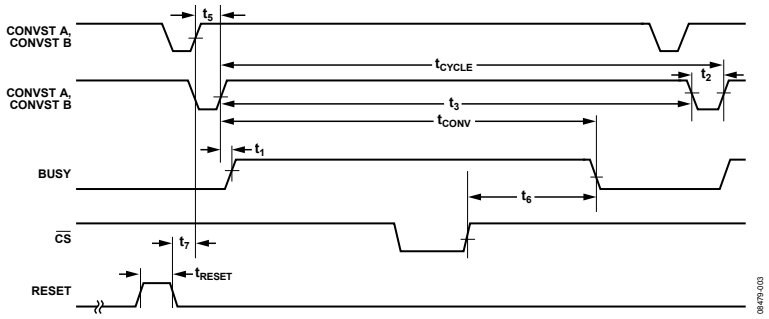

Figure 8: ADC timings [4]

3 Improvements done on the hardware system

We did two main adjustments in the VHDL code of the project : we changed the main state machine behavior in Juan Azcarreta's SPI controller implementation, and we wrote a design for a device that enables the user to control the origin of the data.

3.1 Adjustment of the sampling frequency to 48 kHz

The Pyramic array design contains a state machine controlling the SPI Controller. This state machine drives the SPI bus and the signals that order the start of the conversion and the readback of the captured data.

Table 4 shows the signals that have to be driven in order to start a capture and get the data.

The resistors `OS0`, `OS1`, `OS2` on the ADCs have been set such that the oversampling value is 4. The ADC documentation gives a maximum sampling frequency of 50000 Hz in this case, thus a sampling rate of 48000 Hz meets this constraint.

This state machine has to respect two timing constraints at the same time :

- It must honor the ADC timings as shown on figure 8:
  - Minimum time between `RESET` down and `CONVST` up: $t_7 = 25$ ns
  - Minimum time for `RESET` up: $t_{RESET} = 50$ ns
  - Minimum `CONVST` up time: $t_2 = 25$ ns
  - Minimum time between `BUSY` falling edge and `CSn` rising edge: $t_4 = 0$ ns
- It must honor a sampling rate of 48000 Hz, thus trigger a conversion every 20.833µs .

The aim of this part of the project is to get the system capturing samples at *exactly* 48000 Hz.

3.1.1 Status before the project

Prior to the project, the frequency of the main system clock was 50 MHz; this clock was the one used in the SPI controller.

The 48 kHz frequency was obtained using the following computation:

It was noticed that the conversion time was exactly 17.42 µs for each sample. Knowing that the SPI bus frequency was 12.5 MHz, we knew that the 20.833 µs were reached in 42.625 cycles starting from the `BUSY` down time, when bit 6 of the third microphone of each board was being captured.

However, given that the number of cycles needed to reach an exact 48 kHz wasn't an integer, the actual sampling frequency was not rigourously 48000 Hz.

Moreover, the specification [4] only gives an interval for the conversion time, that may vary with respect to other parameters such as temperature. If this time varies, then the number of clock cycles between two `CONVST` pulses has to change as well.

3.1.2 Changes done

Through the use of a Phase-locked loop, we could adjust the working frequency of the system to 48 MHz exactly : the PLL has the coefficients

$$m = 24 \text{ and } d = 25$$

since $48 = \frac{24}{25} \times 50$.

Having a working frequency of 48 MHz allows us to trigger a conversion every 1000 clock cycles. The exactness of the PLLs are crucial in order to reach exactly 48000 Hz as a sampling rate.

The documentation of the ADCs specify that a conversion can be started before all the data from the previous transmission has been sent. It specifies that a maximum of $t_6 = 25$ ns separates the end of the readback of the previous data (which is materialized by asserting `CSn`) and the availability of the next data. It should be noted that the readback of serial data is performed simultaneously for all the ADCs.

We also don't want to rely on the actual sampling time anymore, given that the datasheet only provides an interval for this value. Thus, we introduced a new state machine that only drives the `CONVST` signal using a counter that goes from 0 to 999.

We can easily check that our design is correct with respect to the ADC timing requirements.

The longest sampling time with an oversampling of 4 is 18.8 µs, the minimum being 16.05 µs [4], while the cycle time t_{cycle} is 20.83 µs. The samples are 16-bit long, and there are 8 microphones per board to be read, with a bit rate of 12.5 Mbits / s. The readback time is thus $\frac{16 \times 8}{12.5 \times 10^6} = 10.24$ µs.

Should one conversion take $t_A = 18.8$ µs and the following one take $t_B = 16.05$ µs, we would have

$$(t_{cycle} - t_A) + t_B - t_6 = 18.055 \text{ µs}$$

to complete the readback of the samples, which is achievable given the readback time of 10.24 µs.

The new state machines that control the ADCs are shown on figure 9. It should be noted that the captured data is sent to a FIFO once all the bits for all the microphones have been captured. The six waiting states allow to satisfy

Given that the minimum `CONVST` up time is 25 ns, assuming that one clock cycle is 20,83 ns, 2 clock cycles of `CONVST` up are sufficient to meet the timing constraint from the ADC.

3.1.3 Note on the capture start time

One should note that the the CONVST pulses are synchronous for all the ADCs, to the following extent: the CONVST signal is output for all the ADCs at the same time by the FPGA; however, how the connections on the FPGA were done implies that we have three chains of two microphone arrays seen at section 2.1.1.

Given the celerity of the electrical signals, we estimate the propagation time of the signal to be less than 1 ns, given that the signal has approximately 20 cm to run.

With a working frequency of 48000 Hz, we have one pulse every 20.8 µs, which means that the imprecision of the conversion start time relative to the sampling frequency is of

$$\frac{\Delta_t}{t} \approx 4,8 \times 10^{-5} = 0.0048\%$$

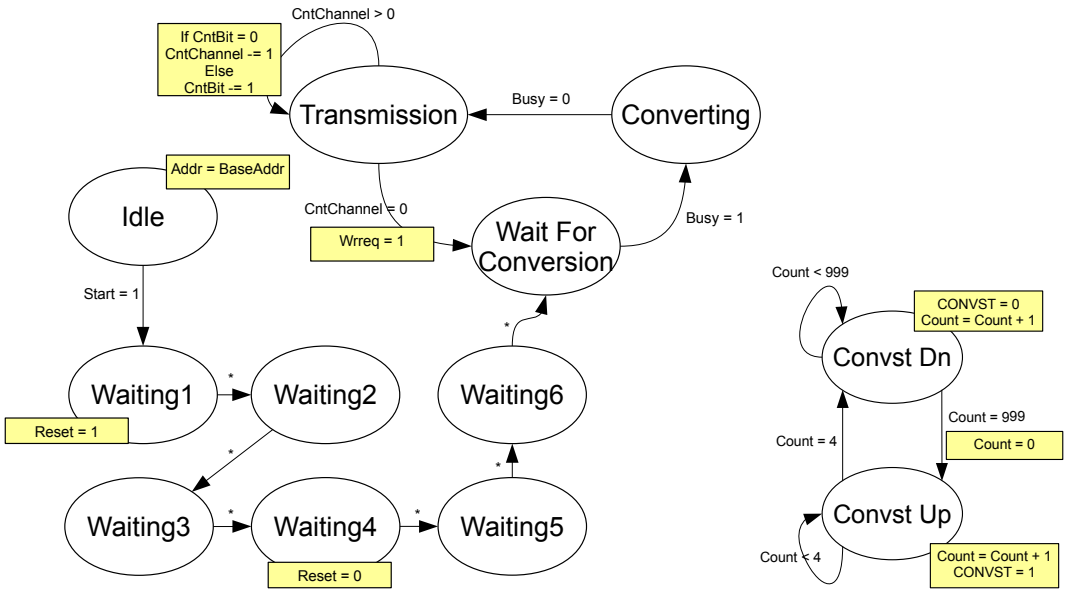

Figure 9: New state machines for the SPI controller

3.2 Adding of an output capability

We designed and added into the project a controller which enables the user to control where the data fed to the audio controller comes from.

The beamformer part on the FPGA is not currently implemented. In order to allow a faster connection between the beamformer output and the audio output, we decided that it would be possible to choose between two inputs:

- The beamformer, which is a chain of digital filters and summation operators implemented as a hardware accelerator,
- The DDR3 memory, which enables software to output sound.

Therefore, we needed to create a configurable component, which would allow us to set the input to either the beamformer or the memory, and configure the settings relative to the memory (base address of the samples, length of the buffer).

The block schematic for this component is shown on figure 10 .

The design is made of :

- two Avalon Streaming Sink interfaces,
- two Avalon Streaming Source interfaces,
- a Direct Memory Access (DMA) master unit as an Avalon Memory-Mapped Master.

3.2.1 Avalon Memory-Mapped slave for configuration

The output controller is configurable, through the use of an Avalon Memory-mapped slave. This slave is linked to the master on the HPS-to-FPGA bridge.

It contains several registers, table 5 shows the register map for the device.

The Avalon Memory-mapped slave that this interface provides is directly connected to the Avalon Memory-mapped master of the HPS-to-FPGA interface.

3.2.2 Direct Memory Access controller

The Direct Memory Access controller is connected to the FPGA-to-HPS bridge; as such, it can access the shared memory between the HPS and the FPGA. Its role is to fetch data from the memory at a rate of 48 kHz and feed them to the Avalon streaming sources.

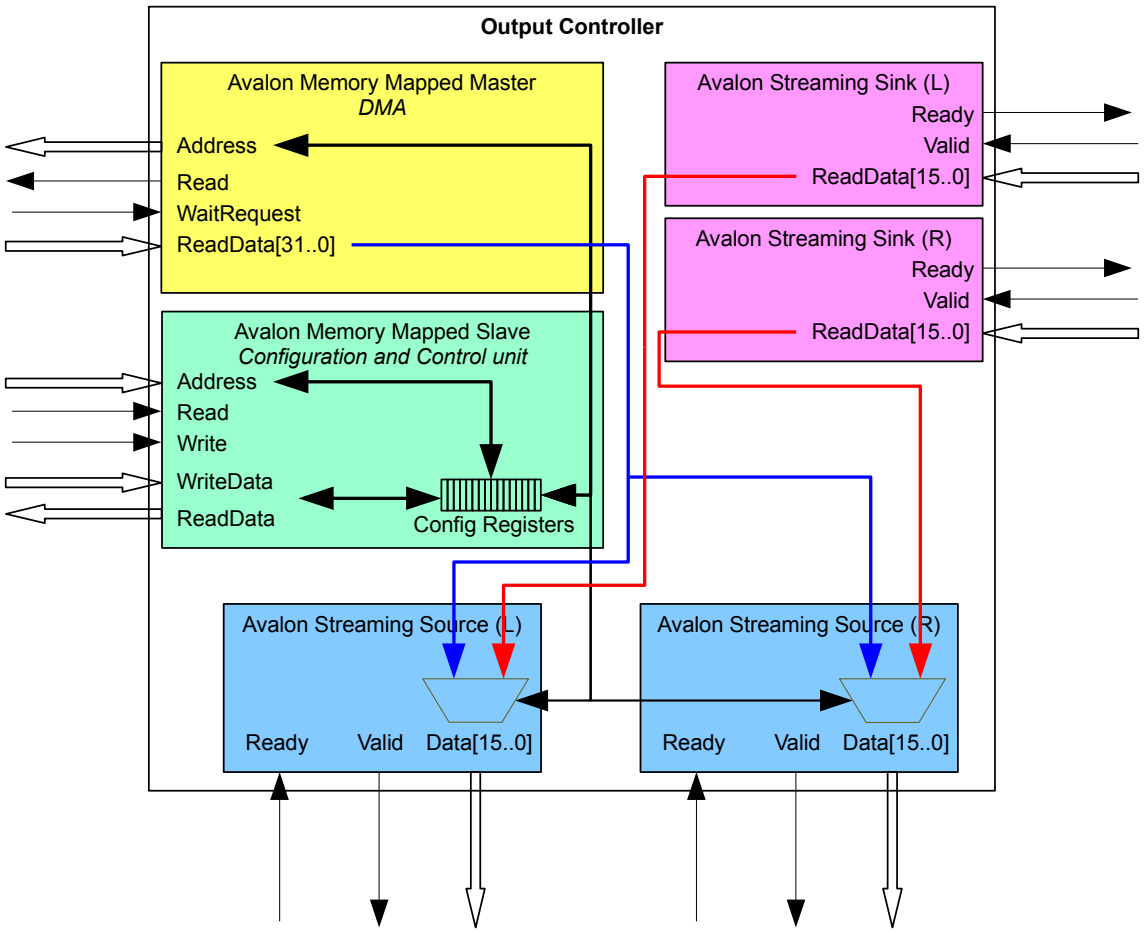

Figure 10: Block schematic for the output controller

Register	Address	Width	Role
`baseAddress`	0	32	Starting offset in the memory, where the sample array begins.
`soundLength`	1	32	Length of the sample array.
`useMemory`	2	1	Flag that configures whether the memory or the streaming sinks are used as an input.

Table 5: Register map for the output controller

Signal	Width	Direction	Description
`Address`	32	Master → Slave	The address of the data to be read in memory.
`Read`	1	Master → Slave	Asserted by the master when beginning a read operation. Released only when WaitRequest=0.
`WaitRequest`	1	Slave → Master	Asserted when the slave needs to hold the master signals for being busy.
`ReadData`	32	Slave → Master	The response data for a read operation.
`ReadDataValid`	1	Slave → Master	Asserted when the response data is valid.

Table 6: Signals used by the DMA master

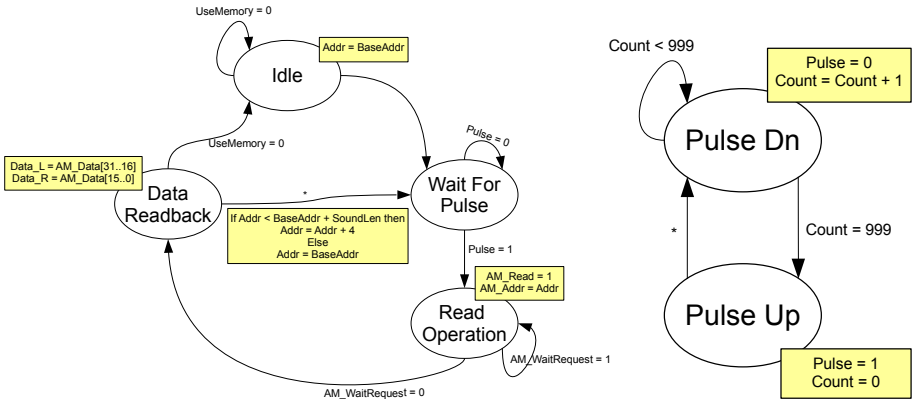

Figure 11: State machines devised for the DMA of the output controller

Table 6 shows the signals that the DMA master is using.

Two state machines drive the DMA controller : one that generates a 48 kHz pulse to trigger a DMA read, and another one that performs the actual DMA readout. Figure 11 shows these state machines, and figure 12 shows a timing diagram for a read operation.

It should be noted that, since the width of the data bus (32 bits) is equal to the width of one sample, we don't have to perform a burst read.

Once we have got the left and right values of the sample from the DMA, we set the data on the output lines `Data_L` and `Data_R` to those values.

3.2.3 Output via an Avalon Streaming Source

An Avalon streaming source feeds data into the Altera Audio IP core. It uses the Avalon Streaming interfaces of this IP core, thus we have created two Avalon Streaming sources (one for the left channel,

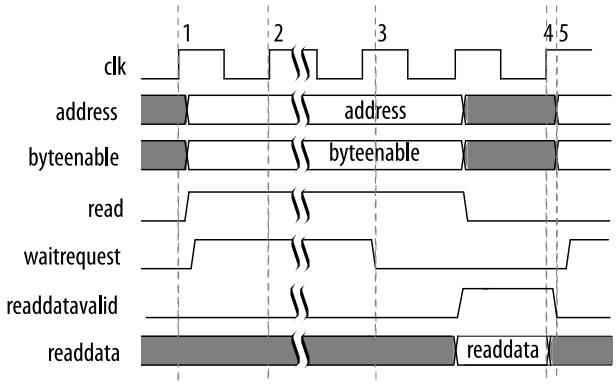

Figure 12: Timing diagram for a read operation

one for the right). They are driven:

- either by the Avalon Streaming sinks that are connected to the beamformer; in that case, the `data` and `valid` signals at the source are just set to the values they take at the sink, and `ready` is set in the opposite direction,
- or by the DMA; in that case, the `data` line is fed by the data read in the DMA, and the `valid` signal is asserted has the same value as the `readdatavalid` signal of the DMA.

The audio core then handles the signals; refer to section 2.2.1 for the configuration details.

3.3 FPGA design after the project

Figure 13 shows an overview of the system (as connected in Qsys) after the completion of the project. In green are the part subject to major changes: the output buffer and the SPI system.

4 Improvements done on the software part

One of the aims of the project was to enable signal processing on the samples by using custom algorithms. Those need to be implemented in software that runs on the HPS. Ordinarily, interfacing a program on the HPS with a hardware design on the FPGA requires the use of a Quartus tool that creates headers containing the adresses of the design parts connected to the HPS-to-FPGA bridge.

We expect that multiple algorithms will be developed against a single Pyramic array design; thus, we needed to provide the application developer a library that gives easy control over the Pyramic array functions.

We also don't expect the application developer to configure network interfaces by hand. Thus, we modified the Linux system configuration and wrote scripts that automatically configure the Pyramic network interfaces, making the Pyramic a plug-and-play device from the network point of view.

4.1 Access library and API : `libpyramicio`

We created a library that interfaces the Pyramic array and provides to the applications with a set of functions that can be used to manage the capture and playback of sound.

The aim of this library is to enable the development of applications directly against the Pyramic array design without having to extract any header files using Altera Quartus' tools. Thus, an application developer only needs to be shipped a compiled version of the library, and the API header file `pyramicio.h` to interact with the Pyramic array.

The library is a dynamic linked library, known under Linux as a shared library. This enables an update of the design and thus of the library without having to recompile the application.

Three structures are used by this API, two of them are actually useful for the application developer:

- The structure `struct inputBuffer` gives the developer access to the samples captured by the Pyramic array. It notably contains a pointer to an array, `uint16_t* samples`, that holds the samples, which length is a member of the structure.
  The length indicated in the structure is half of the total length of the buffer.
- The structure `struct outputBuffer` enables the developer to store samples in a buffer that is read in a circular way by the Output Controller. Like its input counterpart, it contains an array holding the samples, but it's up to the application to divide the buffer if it needs to do so.

Both in input and output, the samples have the same format as those output by the ADCs, which is to say signed 16-bit with a sampling rate of 48000 Hz. The Doxygen-generated API documentation is attached as appendix A.

Table 7 lists the available functions and briefly describes them.

One should note that the implementation of this API uses the `mmap()` function, that maps the adresses of the hardware components on the FPGA to userspace addresses that the applications can use. As this requires a file descriptor on `/dev/mem`, the functions can only be run as root.

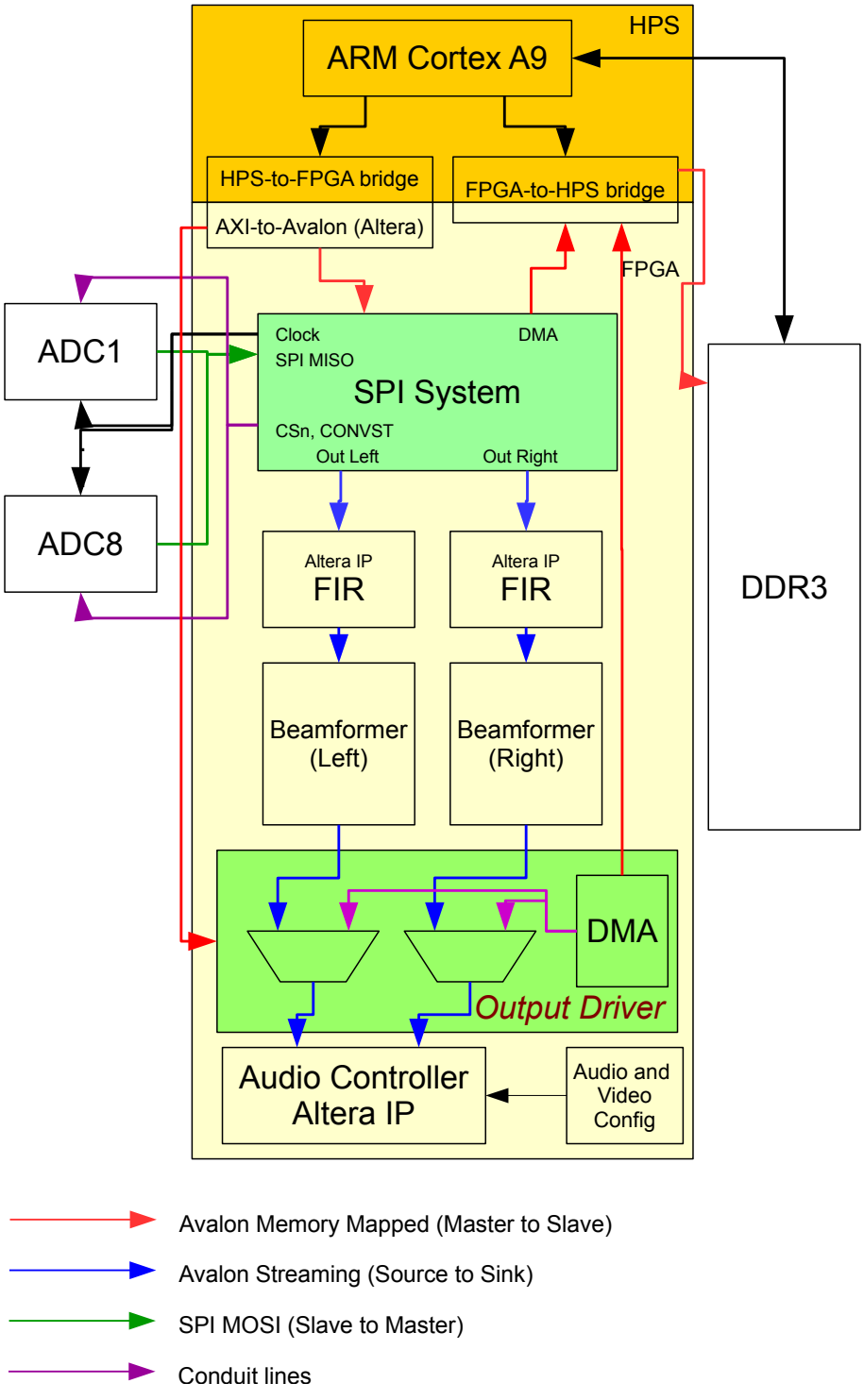

Figure 13: Design after the project

Function name	Role
pyramicInitPyramic	Initializes the Pyramic array and returns a reference to the associated Pyramic object.
pyramicDeinitPyramic	Closes the file descriptors assoiated with the Pyramic and frees the reserved memory resources.
pyramicGetInputBuffer	Returns an input buffer structure where the samples are the result of the ongoing capture, or the last capture if none is in progress.
pyramicGetCurrentBufferHalf	Returns the current half of the buffer on which the DMA controller is currently writing to. The other half can then be used safely to read.
pyramicAllocateOutputBuffer	Returns an output buffer structure, with a sample array of the requested size.
pyramicDeallocateOutputBuffer	Frees the resources used by the output buffer.
pyramicStartCapture	Starts a continuous capture on the Pyramic array, with a specified buffer length.
pyramicFixedLengthCapture	Starts a captue on the Pyramic array, that will last for the specified duration.
pyramicStopCapture	Stops the ongoing capture on the Pyramic after the current sample has been captured.
pyramicSelectOutputSource	Configures the output driver with either a beamformer input, or a memory input.
pyramicSetOutputBuffer	Configures the output driver and sets the base address and the length to those of the specified output buffer structure.

Table 7: List of functions provided by the API

4.2 Configuration of the Linux network stack

We configured the Linux network stack that runs on the HPS so that access to the Pyramic array is made easy. The configuration consists in the following :

- The interface eth0 has been defined in /etc/network/interfaces to acquire its configuration via a DHCP server. This means that this interface gets its IP address, network mask and DNS configuration through the use of the Dynamic Host Configuration Protocol (DHCP) on a server which is usually found in most Plug-and-Play routers.
- The Pyramic array has been assigned the "LCAVPyramic" hostname, and mDNS has been enabled through the use of Avahi and NSS. Thus, an user who has either Microsoft Windows, or any Unix-like system with Avahi installed, can reach the Pyramic array through the "LCAVPyramic.local" hostname, without having to find the IP address that has been assigned.

Should the user not have the Avahi and NSS installed on their computer, a post-configuration script is run when the network link of the Pyramic is up, and displays the IP address of the eth0 interface on the JTAG / UART interface. This interface requires a direct USB connection between the computer and the Pyramic.

5 Results

5.1 VHDL Compilation results

We compiled the design proposed in this report using Altera Quartus. For the Cyclone V FPGA 5CSEMA5, the compilation results are as table 8 shows.

The low occupancy on the Look-up table (LUT) allows for further combinational design implementation.

Property	Value
Logic utilization	2'622 / 32'070 (8%) of the gates
Pin utilization	179 / 457 (39 %)
Memory utilization	1'718'288 / 4'065'280 (42%)
PLL utilization	3 / 6 (50%)
DLL utilization	1 / 4 (25%)

Table 8: Quartus compilation results

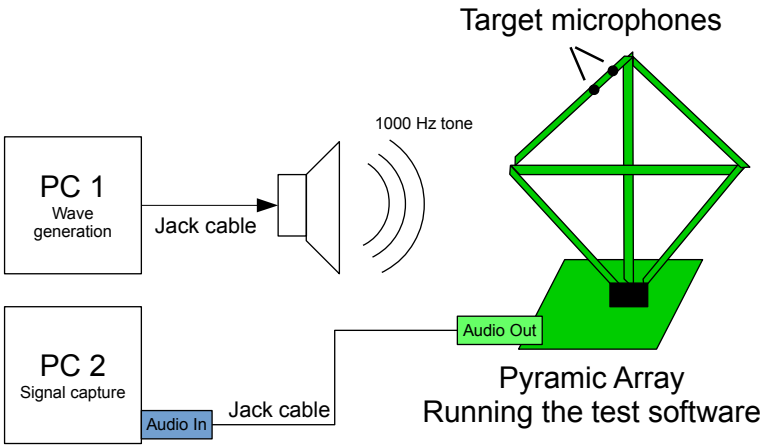

Figure 14: Setup used for testing the Pyramic array

5.2 Tests carried

We tested the design and software against a 1000 Hz tone, output by a computer placed near the Pyramic array. Figure 14 shows the setup.

We wrote an example software using `libpyramicio`, which captures samples for 5 seconds and sends the capture from Microphone 0 to the left channel, the capture from Microphone 1 on the right channel in an output buffer. This output buffer is assigned as a source to the output driver, and another computer captures the resulting signal.

A sample from the source code of the software exploiting libpyramicio is :

```
pyramicSelectOutputSource(p, SRC_BEAMFORMER); // This is a way
    to get silence, since the beamformer isn't connected.
    Another way could be setting the output to a buffer full of
    zeroes
pyramicFixedLengthCapture(p, 5); // Start a capture that will
    last for 5 seconds
usleep(200); // Start waiting for the capture to end
while(pyramicGetCurrentBufferHalf(p) != 0) { // Poll to check
    for the capture being done

}
printf("5s␣capture␣finished\n");
struct inputBuffer* inBuf = pyramicGetInputBuffer(p, 0); // We
    choose the half 0 for the 1st half of the buffer
struct outputBuffer* outBuf = pyramicAllocateOutputBuffer(p, 2*
    inBuf->samplesPerMic); // We allocate as many samples as
    there are in the input buffer
```

After running the software, we captured the signal output from the audio output of the DE1-SoC board, and plotted a frequency spectrum using Audacity. Figure 15 shows the graph obtained. We can observe the fundamental frequency at 1000 Hz.

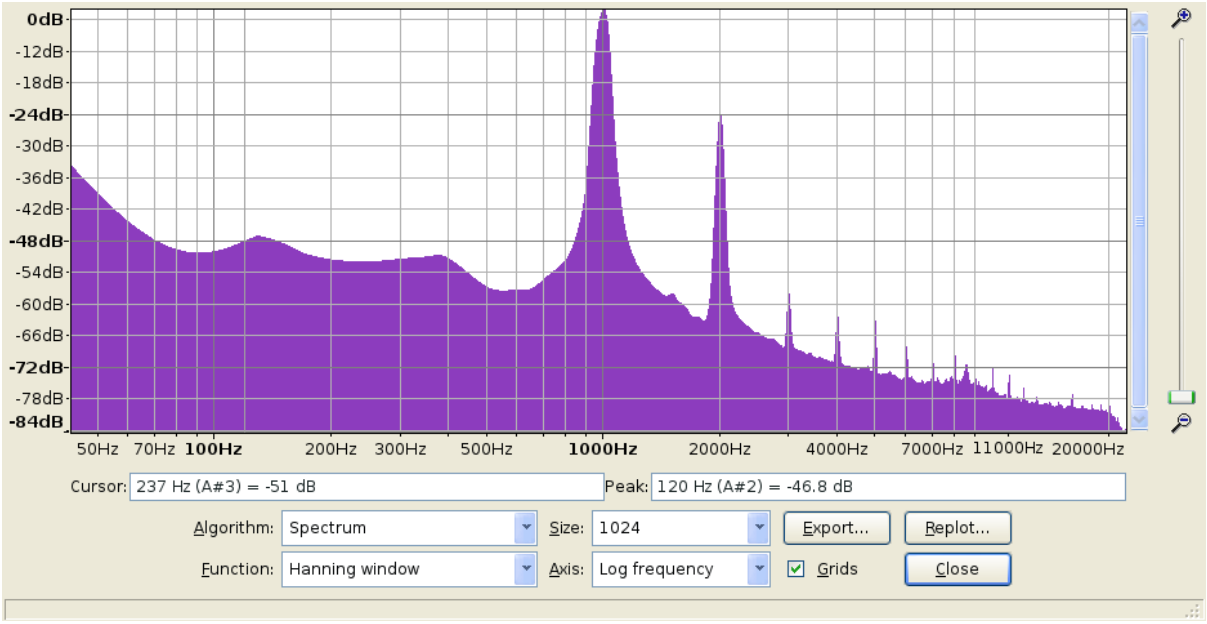

Figure 15: Frequency spectrum for the captured signal

In order to confirm that the capture at 48.000 Hz works for every ADC, we repeated the same operation for one microphone pair for each ADC. Since the CONVST signal is common to all ADCs and all ADCs are identically configured, we observed no different behavior for all the microphone boards.

The width of the peak can be explained by many factors : for the experiments, we didn't use a pure sinusoidal signal generator, but speakers from a computer; distortion can also happen in the ADCs both in the Pyramic and the PC 2 that captures and analyses the data.

6 Conclusion and futher work

The goal of this project was to improve the Pyramic array, first of all to adjust the sampling frequency, which is a crucial point for applications in physics. The stability of the capture frequency ensures that the captured sound signal is consistent and the phaseshift observed between two signals is exact.

It is desirable that the Pyramic array could be used by physicists or in industrial applications; such users typically want to be shipped a device they can directly develop their application on. Using `libpyramicio`, this is on the way to be possible: one would be provided with the compiled design, and would only need a C compiler to build their application against the library.

A part of the desired functionalities remain unimplemented: the beamformer accelerator is still not functional. However, one can perform beamforming on the HPS.

A further work in this project would have been to implement the filters; yet, the computing resources and the memory inside the FPGA are an issue to be tackled. For instance, given that we capture at 48000 Hz, if we want the filters to be able to process half a second of signal, we would have to create a component that fits these constraints:

- Store and hold 24000 samples for 48 microphones, which is to say 1'152'000 samples,
- Store and hold as many coefficients as samples,
- Be able to adapt to the user's algorithm i.e. reconfigurable coefficients, that have to be fetched, and the computations run before the next sample arrives.

Given these constraints, one would have to devise a design that fits in the Cyclone V FPGA.

References

[1] Juan Azcarreta Ortiz, *An FPGA based platform for many-channel audio acquisition*. Master's Thesis, EPFL (Lausanne, Switzerland), 2016.

[2] Altera Corporation, *Avalon Interface Specifications*. 201x. Pages 26, 32. https://www.altera.com/literature/manual/mnl_avalon_spec.pdf

[3] Terasic Technologies, Inc. *DE1-SoC User Manual v 1.2.2*. 2014. PP 7, 30, 31.

[4] Analog Devices, AD7606 datasheet. 2007. http://www.analog.com/media/en/technical-documentation/data-sheets/AD7606_7606-6_7606-4.pdf

[5] Altera Corporation, University Program. *Audio Core for Altera DE-series boards*. 2007. ftp://ftp.altera.com/up/pub/Altera_Material/11.1/University_Program_IP_Cores/Audio_Video/Audio.pdf

[6] Altera Corporation, University Program. Audio/Video Configuration Core for DE-Series boards. ftp://ftp.altera.com/up/pub/Altera_Material/11.0/University_Program_IP_Cores/Audio_Video/Audio_and_Video_Config.pdf

[7] Altera Corporation. *ALTPLL (Phase-Locked Loop) IP Core User Guide*. 2014. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_altpll.pdf

A Documentation for libpyramicio

Contents

Chapter 1

Documentation for libpyramicio

1.1 What is libpyramicio ?

This library is an abstraction layer for the Pyramic array -made at LCAV (EPFL)- input/output functions. It enables the use of the Pyramic array by designing software against an existing hardware design without having to use Altera Quartus Prime tools, or recompiling the application at each change of the design in VHDL.

1.2 How to use it ?

In order to use the library, one just has to include the <pyramicio.h> file, then initialize a Pyramic object through the pyramicInitializePyramic() function.

All the usable functions are documented in the pyramicio.h file reference in this documentation.

Note that programs that use the Pyramic have to be run as root, because the library is using direct references to memory areas that are reserved for the system.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 inputBuffer Struct Reference

This structure represents an input buffer, which direction is the microphone array towards the memory.

`#include <pyramicio.h>`

Public Attributes

- int microphoneCount

  How many microphones the Pyramic array has.
- uint32_t totalSampleCount

  How many samples the buffer contains.
- uint32_t samplesPerMic

  How many samples are found for each microphone in the buffer.
- int16_t * samples

  The actual samples, organized the RIFF way.

4.1.1 Detailed Description

This structure represents an input buffer, which direction is the microphone array towards the memory.

4.1.2 Member Data Documentation

4.1.2.1 microphoneCount

`int inputBuffer::microphoneCount`

How many microphones the Pyramic array has.

4.1.2.2 samples

```
int16_t* inputBuffer::samples
```

The actual samples, organized the RIFF way.

4.1.2.3 samplesPerMic

```
uint32_t inputBuffer::samplesPerMic
```

How many samples are found for each microphone in the buffer.

4.1.2.4 totalSampleCount

```
uint32_t inputBuffer::totalSampleCount
```

How many samples the buffer contains.

The documentation for this struct was generated from the following file:

- pyramicio.h

4.2 outputBuffer Struct Reference

This structure represents an output buffer, which direction is the memory towards the FPGA CODEC.

```
#include <pyramicio.h>
```

Public Attributes

- uint32_t **baseAddress**
- uint32_t **length**
- int16_t * **samples**

4.2.1 Detailed Description

This structure represents an output buffer, which direction is the memory towards the FPGA CODEC.

This CODEC is configured to work at 48000 Hz, hence the sampling rate of the injected audio *has* to be 48000 Hz.

The documentation for this struct was generated from the following file:

- pyramicio.h

4.3 pyramic Struct Reference

Structure contanining the addresses used by the library internals .

`#include <pyramicio.h>`

Public Attributes

- void * **h2f_lw_axi_master**
- size_t **h2f_lw_axi_master_span**
- size_t **h2f_lw_axi_master_ofst**
- void * **fpga_SPI_System**
- void * **fpga_Output_Controller**
- int **fd_dev_mem**
- void * **reserved_memory**
- void * **output_memory**
- int **captureDuration**

4.3.1 Detailed Description

Structure contanining the addresses used by the library internals .

The documentation for this struct was generated from the following file:

- pyramicio.h

Chapter 5

File Documentation

5.1 pyramicio.h File Reference

A library that allows an easy access to the Pyramic array.

```
#include <inttypes.h>
#include <unistd.h>
```

Classes

- struct inputBuffer

  This structure represents an input buffer, which direction is the microphone array towards the memory.
- struct outputBuffer

  This structure represents an output buffer, which direction is the memory towards the FPGA CODEC.
- struct pyramic

  Structure contanining the addresses used by the library internals .

Macros

- #define **SRC_BEAMFORMER** 0
- #define **SRC_MEMORY** 1

Functions

- struct pyramic ∗ pyramicInitializePyramic ()

  Initializes the Pyramic array and returns a reference to the associated Pyramic object.
- void pyramicDeinitPyramic (struct pyramic ∗p)

  Closes the file descriptors assoiated with the Pyramic and frees the reserved memory resources.
- struct inputBuffer ∗ pyramicGetInputBuffer (struct pyramic ∗p, int bufferHalf)

  Gets the current input buffer.
- int pyramicGetCurrentBufferHalf (struct pyramic ∗p)

  Gets the number of the half on which the Pyramic is currently recording samples.
- struct outputBuffer ∗ pyramicAllocateOutputBuffer (struct pyramic ∗p, uint32_t lengthInSamples)

Allocates memory as a buffer to output samples.

- void pyramicDeallocateOutputBuffer (struct pyramic *p, struct outputBuffer *outputBuffer)

  Sets the Pyramic output buffer to be the specified address space.
- int pyramicStartCapture (struct pyramic *p, int bufferLengthInSeconds)

  Starts a continuous capture on the Pyramic array.
- int pyramicFixedLengthCapture (struct pyramic *p, int durationInSeconds)

  Starts a fixed length capture on the Pyramic array.
- int pyramicStopCapture (struct pyramic *p)

  Stops the ongoing capture on the Pyramic array at the end of the current sample.
- int pyramicSelectOutputSource (struct pyramic *p, int source)

  Selects if the output samples come from the Beamformer or a software buffer.
- int pyramicSetOutputBuffer (struct pyramic *p, struct outputBuffer *outputBuffer)

  Sets the Pyramic's output buffer as the designated one.

5.1.1 Detailed Description

A library that allows an easy access to the Pyramic array.

This library is compiled using headers derived from the VHDL code available at: http://github.com/lcav/pyramic.git. In order to compile this library, one has to run the "headers_rbf.sh" file that can be found in the toplevel MIC_ARRAY directory to provide the Quartus generated header files.

The pyramicio.h file gives access to the API provided by libpyramicio. It enables the use of a Pyramic array with an abstraction layer that removes the hassle of the FPGA addresses. This enables programming applications that use the Pyramic array against an existing design without using the Quartus Prime tools.

Author

Corentin Ferry

Date

December 2016

See also

https://github.com/cferr/pyramic.git

5.1.2 Function Documentation

5.1.2.1 pyramicAllocateOutputBuffer()

```
struct outputBuffer* pyramicAllocateOutputBuffer (
            struct pyramic * p,
            uint32_t lengthInSamples )
```

Allocates memory as a buffer to output samples.

Parameters

p	The Pyramic object structure on which the function is executed.
lengthInSamples	The nummber of samples that the output buffer will hold. Note that the output frequency is 48000 Hz.

5.1.2.2 pyramicDeallocateOutputBuffer()

```
void pyramicDeallocateOutputBuffer (
            struct pyramic * p,
            struct outputBuffer * outputBuffer )
```

Sets the Pyramic output buffer to be the specified address space.

Parameters

p	The Pyramic object structure on which the function is executed.
outputBuffer	The output buffer that has to be freed.

5.1.2.3 pyramicDeinitPyramic()

```
void pyramicDeinitPyramic (
            struct pyramic * p )
```

Closes the file descriptors assoiated with the Pyramic and frees the reserved memory resources.

Parameters

p	The Pyramic object structure on which the function is executed.

5.1.2.4 pyramicFixedLengthCapture()

```
int pyramicFixedLengthCapture (
            struct pyramic * p,
            int durationInSeconds )
```

Starts a fixed length capture on the Pyramic array.

Parameters

p	The Pyramic object structure on which the function is executed.
durationInSeconds	The duration of the capture. After the capture, you will be able to get the samples through the pyramicGetInputBuffer() function.

5.1.2.5 pyramicGetCurrentBufferHalf()

```
int pyramicGetCurrentBufferHalf (
            struct pyramic * p )
```

Gets the number of the half on which the Pyramic is currently recording samples.

The other half can be safely used for processing the signal.

Parameters

p	The Pyramic object structure on which the function is executed.

5.1.2.6 pyramicGetInputBuffer()

```
struct inputBuffer* pyramicGetInputBuffer (
            struct pyramic * p,
            int bufferHalf )
```

Gets the current input buffer.

Parameters

p	The Pyramic object structure on which the function is executed.
bufferHalf	If this parameter is 0, the samples start at the beginning of the buffer (thus giving you access to the first half and the second half as well). If it is 1, the samples start at the beginning of the second half of the buffer. This parameter is useful for continuous captures where it is safe to use a single half of the buffer at a time.

5.1.2.7 pyramicInitializePyramic()

```
struct pyramic* pyramicInitializePyramic ( )
```

Initializes the Pyramic array and returns a reference to the associated Pyramic object.

This initialization is exclusive: only one thread can have control over the Pyramic array at the same time.

5.1.2.8 pyramicSelectOutputSource()

```
int pyramicSelectOutputSource (
            struct pyramic * p,
            int source )
```

Selects if the output samples come from the Beamformer or a software buffer.

Parameters

p	The Pyramic object structure on which the function is executed.
source	Either SRC_BEAMFORMER (not implemented yet, gives silence) or SRC_MEMORY (the pyramic then takes its input from an output buffer in the DDR3). It is recommended to set the output buffer
	address through pyramicSetOutputBuffer() before calling this function with SRC_MEMORY.

5.1.2.9 pyramicSetOutputBuffer()

```
int pyramicSetOutputBuffer (
            struct pyramic * p,
            struct outputBuffer * outputBuffer )
```

Sets the Pyramic's output buffer as the designated one.

Parameters

p	The Pyramic object structure on which the function is executed.
outputBuffer	An output buffer that has been allocated with pyramicAllocateOutputBuffer().

5.1.2.10 pyramicStartCapture()

```
int pyramicStartCapture (
            struct pyramic * p,
            int bufferLengthInSeconds )
```

Starts a continuous capture on the Pyramic array.

Parameters

p	The Pyramic object structure on which the function is executed.
bufferLengthInSeconds	The duration of the sample buffer. Note that the sample buffer is divided into two halves, and you can safely read and write into each half while it is not being processed, using pyramicGetCurrentBufferHalf(). The buffer has to be long enough so that half of it can be entirely processed while the other half is under capture. You can get the capture buffers through the pyramicGetInputBuffer() function.

5.1.2.11 pyramicStopCapture()

```
int pyramicStopCapture (
            struct pyramic * p )
```

Stops the ongoing capture on the Pyramic array at the end of the current sample.

Parameters

p	The Pyramic object structure on which the function is executed.

Index

B Source code for the example program using libpyramicio

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pyramicio.h>

int main(void)
{
        struct pyramic* p = pyramicInitializePyramic();

        if(p) {
                printf("Success␣in␣Initializing␣Pyramic␣!\n");

                // Silence
                pyramicSelectOutputSource(p, SRC_BEAMFORMER);

                pyramicFixedLengthCapture(p, 5);

                usleep(200);
                while(pyramicGetCurrentBufferHalf(p) != 0) {
                // We wait for capture to finish
                }
                printf("5s␣capture␣finished\n");
                struct inputBuffer* inBuf = pyramicGetInputBuffer(p, 0); // 0
                    for 1st half, but we can indeed get all samples
                struct outputBuffer* outBuf = pyramicAllocateOutputBuffer(p, 2*
                    inBuf->samplesPerMic);

                int i;
                for(i = 0; i < inBuf->samplesPerMic; i++) {
                        outBuf->samples[2*i] = inBuf->samples[48*i]; // L :
                            microphone 1
                        outBuf->samples[2*i+1] = inBuf->samples[48*i + 1]; // R
                             : microphone 2
                }

                printf("Output␣buffer␣filled␣!\n");
                pyramicSetOutputBuffer(p, outBuf);
                pyramicSelectOutputSource(p, SRC_MEMORY);
                printf("Source␣selected!\n");

                pyramicDeinitPyramic(p);
        }
        else
                printf("Failed␣to␣init␣Pyramic␣!\n");

        return 0;
}
```