Acoustic-Warfare: FPGA sampling

Ivar Nilsson och Jacob Drotz

12 augusti 2022

Contents

1 Introduction

This project is based on an 8x8 microphone array, where beamforming is the main purpose. The FPGA(Field programmble Gate Array) is going to sample the microphones and send the data via ethernet (1000Mb/s) to a PC who is responsible for the beamforming algorithm. The development board used is the Zybo Z7-20. Both PL(programmble logic) and PS(processing system) will be used, PL will host sampling of the microphones and PS is used for the ethernet UDP-socket.

The microphones used in the array are the ICS-52000, which includes an analog-to-digital converter, antialiasing filters. Each microphone outputs an 24-bit twos compliment TDM-interface(Time-division-multiplexing-interface). The TDM-interfaces allows 16 microphones to be connected in a daisy-chained structure.

2 Method

The FPGA part of the project is built on RTL(register-transfer level) code in VHDL and the algorithmic code is written in C. The FPGA board is from Xilinx therefore Vivado and its built in software development kit (SDK) will be used.

2.1 Microphone array

The microphones in the array are daisy-chained in four groups with 16 microphones per chain. The sample frequency of the array is the WS signal. Every chain starts to send data on an pulse from WS signal. After a WS-pulse is received in a chain each microphone will send a 32-bit TDM-slot, one at a time. This is repeated for all the 16 microphones in a chain until a new WS pulse is received and the process is restarted. Keep in mind only 24-bits out of the 32bits is actual data since the resolution of ICS-5200 is 24-bit.

Each microphone creates an TDM-slot per sample and each bit in the TDM-slot gets output at an rate of SCK which is (n x 32 x fs), where n is the number of microphones in the chain and fs is the WS-pulse frequency, see Figure 1.

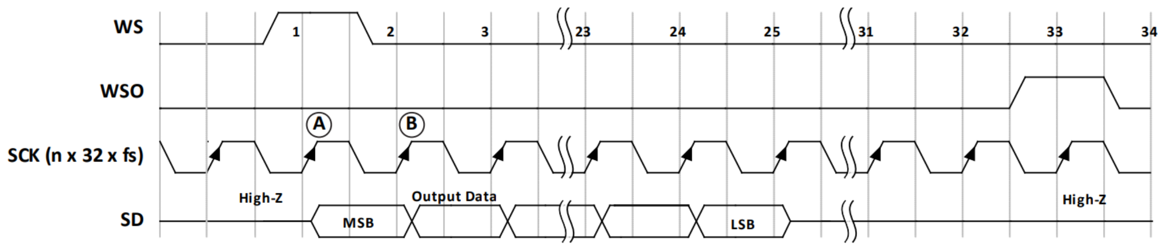

Figure 1: TDM Slot Timing Diagram for one microphone

2.2 PL

The general flow of the programmable logic can be seen in Figure 2. Every component will be further explained in this chapter.

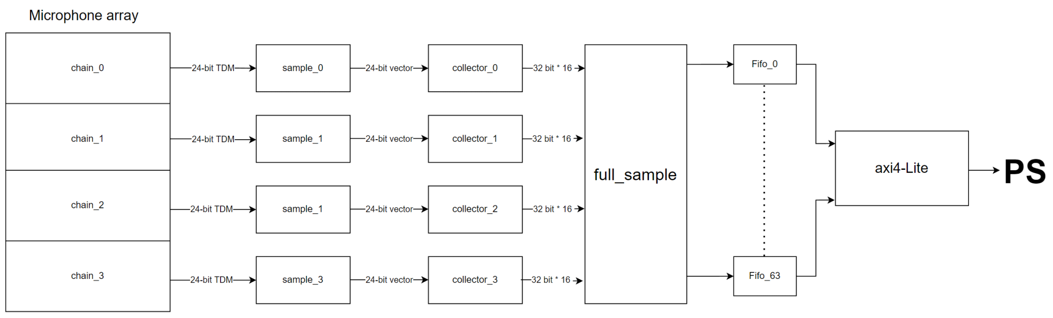

Figure 2: Structure of PL

2.2.1 Component: Sample

```
entity sample is
   port (
      clk                  : in std_logic;
      reset                : in std_logic;
      bit_stream           : in std_logic;
      ws                   : in std_logic;
      mic_sample_data_out  : inout std_logic_vector(23 downto 0);
      mic_sample_valid_out : out std_logic := ’0’
   );
end entity;
```

The sample component is the first component who is in contact with the bit-stream from the microphone array. It will sample each bit in the TMD-slots at an rate of CLK (5x SCK) and put it in an vector, when all 24 bits of an microphone is sampled a valid signal is set to enable the next component **collector** to read the output vector.

The sample component is a state-machine with three states, IDLE, RUN and PAUSE.

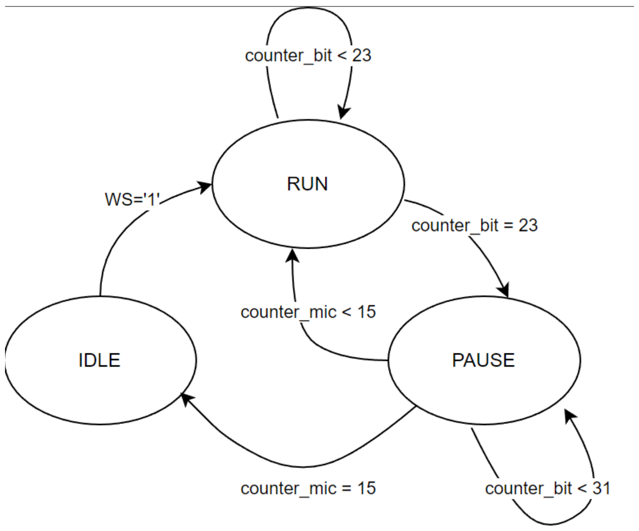

Figure 3: State-diagram for Sample block

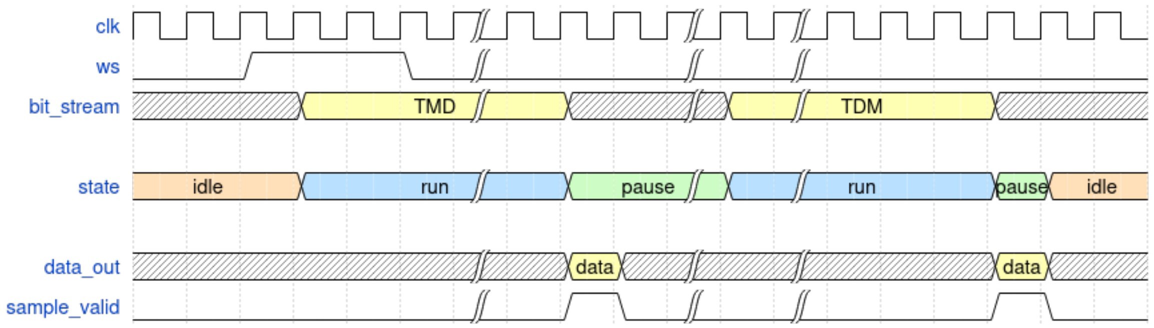

Figure 4: Wave timing for sample

IDLE is the starting state and there the machine will wait for a WS-pulse, as can be seen in Figure 3 and Figure 4. This advances the machine to enter the RUN state. While in the RUN state the bit_stream gets sampled at the rate of

5x SCK which translates to five samples per incoming bit. The majority of 1:s or 0:s determines if the bit from the bit_stream is 1 or 0. The determined bit will be shifted in to a register and this process is repeated until all 24 bits from a microphone has been accumulated. This activates the valid signal which lets the next component to read the vector data_out from the shift register.

After the valid signal have been activated the machine change state to the PAUSE state where it will wait the last 8-bits (the empty bits from a TDM-slot) have passed. After these 8-bits the sample from one microphone is complete and the machine goes back to the RUN state and starts sample the next mic. This is repeated until all 16 microphones in a chain have been sampled. After which the IDLE state is resumed until a new WS pulse is detected and the process restarts.

2.2.2 Component: Collector

```
entity collector is
   port (
      clk                    : in std_logic;
      reset                  : in std_logic;
      mic_sample_data_in     : in std_logic_vector(23 downto 0);
      mic_sample_valid_in    : in std_logic;
      chain_matrix_data_out  : inout matrix_16_32_type;
      chain_matrix_valid_out : out std_logic := '0'
   );
end collector;
```

Collector is an fairly small component who stores and collects all 16 samples from a chain and put it into a single matrix. When the Collector receives the valid signal from the Sample component it reads the sample with the 24 bits from one microphone and put it into shift-register. This process is repeated until data for all 16 microphones have accumulated.

As figure 1 displayed, PL and PS communication is implemented with an AXI4-Lite protocol. This version of the protocol is based on 32-bit data width. It is therefore necessary to pad the data with 8-bits. Since the data is described as twos-complement the padding has to be zeros or ones depending one the samples most significant bit. The padded data is then put in a shift-register, and when the register is full a valid signal is sent out.

2.2.3 Component: Full sample

```
entity full_sample is
   port (
      clk                     : in std_logic;
      reset                  : in std_logic;
      chain_x4_matrix_data_in : in matrix_4_16_32_type;
      chain_matrix_valid_in   : in std_logic_vector(3 downto 0);
      array_matrix_data_out   : out matrix_64_32_type;
      array_matrix_valid_out  : out std_logic;
      sample_counter_array    : out std_logic_vector(15 downto 0)
   );
```

This component merges together samples from all the four chains of the microphone array into single RTL-matrix which now contains a full sample from all microphones, each sample now has and index from 0 to 63. When this component have received samples from all four collector's it sets a valid signal to the next component fifo(First in first out).

2.2.4 Axi4-Lite

This version Axi-communication uses a classic master/slave relation between PS and PL, and is a generated IP block from Vivado. Each microphone has an associated fifo who receives data from Full sample. The fifo's used in this projects are also generated by Vivado from the IP block *fifo generator*. The Axi-buss reads data from all the fifo's and shift the data into a corresponding slave-register for that fifo, which means every microphone have its own fifo and every fifo have its own slave-register.

When the master reads data from a slave-register, a read enable signal gets set and that same slave-register receives new data from its fifo. To make sure the slave does not read the same data from a fifo multiple times the read enable signal is only high for one Axi clock cycle.

2.3 Clocks

The System clock (CLK) is the main clock for all components while the SCK is used output data from the microphone array. CLK, SCK and AXI-CLK are generated by vivado's Clock-wizard. The WS pulse that is sent to the array is taken from SCK through a simple clock divider component. Lastly the ethernet PHY-CLK is predefined by hardware on the Zybo board.

The tables below shows all the clocks at an WS frequency of 48828,125 Hz.

Clocks	**48,8 kHz Microphone output: frequencies**
CLK	125 MHz
SCK	25 MHz
AXI-CLK	125 MHz
WS	48828,125 Hz
Ethernet PHY-CLK	125 MHz

2.4 PS Zynq

The processing system runs on an Arm cortex-A9 processor and the C program have two main purposes, first read from the slave-registers and but it in a buffer, and then transfer that buffer to another PC via UDP. There are some necessary header-files that needs to be included, but by using an Xilinx default example in SDK all of the necessary header-files will be put in the project repository.

2.4.1 Read data from PL

When PS reads a sample from a slave-register, it signals the slave to read and take in new data from its fifo. In addition to the 64 microphone slaves there are two slave-registers is reserved for flags, these flags indicates if one of the fifos is empty. In case of a empty fifo, PS waits to read the data on the slaves until the flag is set to 0 again.

The order of how the microphones enters the slaver-registers is show in Figure 5.

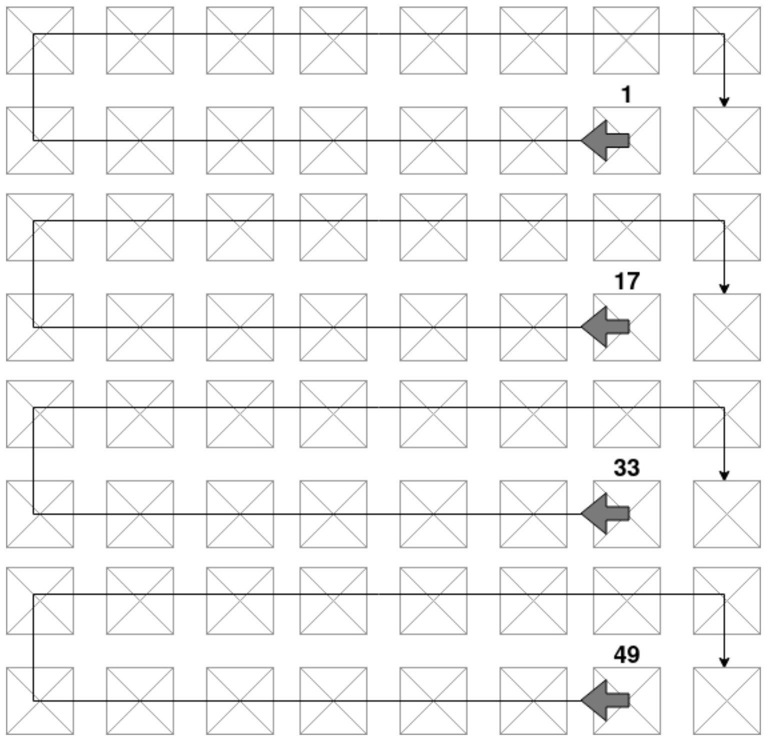

Figure 5: Order of how the microphones enter the slave-registers. (P-MOD connector is on the right and inputs of the microphones is facing the reader).

The function *Xil_ in32* reads data from a slave-register. Each registers is defined by an unique address, starting at base address 0x40000000 and is then incremented by an offset of 0x04. The program reads from all the slave-registers and store it in a buffer, see example code below.

```
for (int i = 0; i < nr_arrays * 64; i++) {
    data[i] = Xil_In32(0x40000000 + 4 * i);
}
```

However, the beamforming algortihm is counting on receiving each microphone sample in a specific order, therefore the program needs to store the samples in the buffer shown in Figure 6.

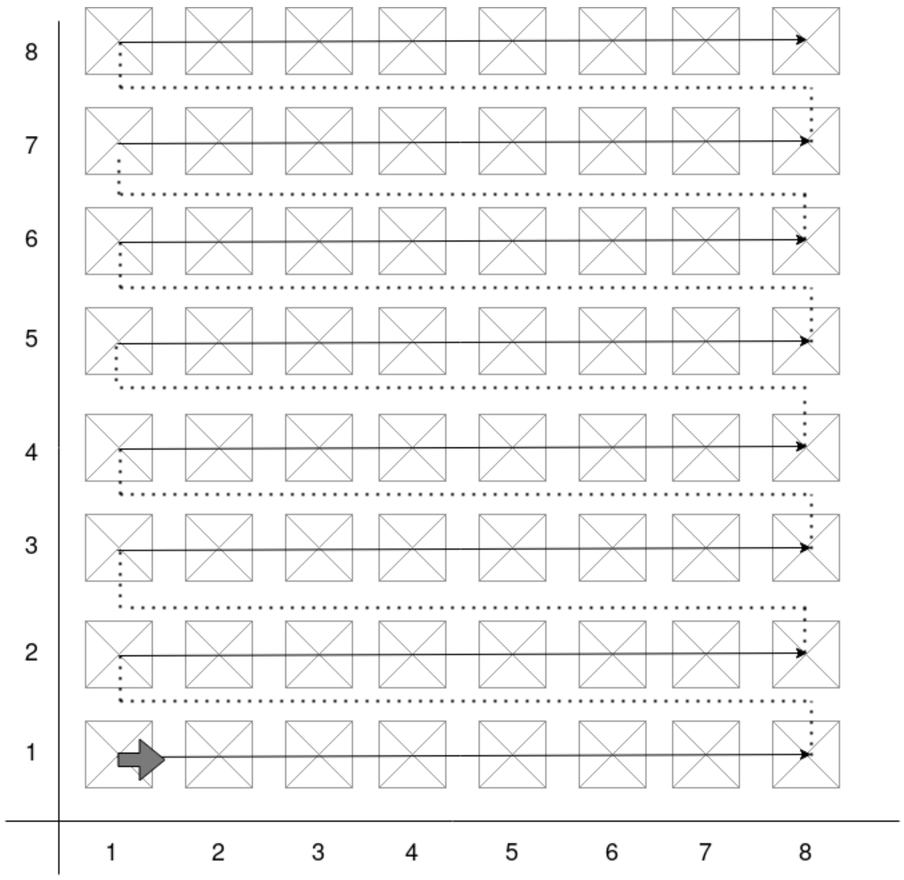

Figure 6: Order of how the microphone samples is stored in the buffer. (P-MOD connector is on the right and inputs of the microphones is facing the reader).

2.4.2 UDP

The size of the buffer who stores data from the slave-registers is 1024 bytes, which means it can contain 256 microphone samples at a time. However the full size of the buffer is not used for the sake of simplicity at the receiving end. Each package only contains one sample from each microphone which is now stored in the buffer and in the order shown in Figure 6. This is equal to a full sample of the microphone array inside each UDP package.

A payload header is included in each UDP packet as can be seen in Figure 7 . This protocol is mostly for debug purposes at the receiver. **array_nr** is number of microphones array used, described in 4 bytes. **protocol_ver** defines the version of the protocol in 4 bytes, which is more important if changes is being implemented in the future. **samp_freq** defines the sample frequency of the microphone-array and **fs_nr** is a counter of the current array-sample included in the payload, both also described in 4 bytes. The total payload for each UDP package is 272 bytes (protocol + buffer)

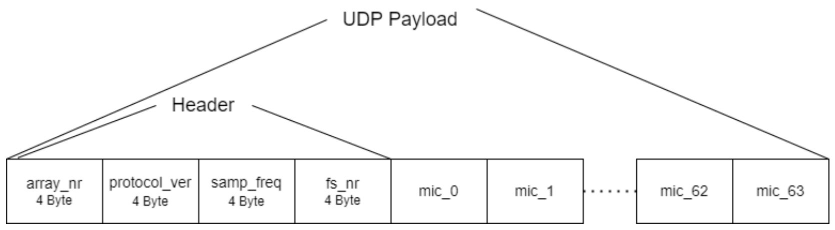

Figure 7: UDP payload protocol

The UDP receiver have its own buffer to collect the UDP package. This buffer is also the size of 1024 bytes and each sample can be reach by the same indexation as it was put into the buffer at the transmitter.

3 Results

Clocks	**f (kHz)**	**Relation**
WS	48.828125	WS * 1
SCK	25000	WS * 512
CLK	125000	WS * 2560

During this test the microphone array was sampling a generated sinus signal at 440 Hz. Figure 8 shows 110 samples from all microphones in the array. The phase differences of the microphones is dependent on each microphones distance from the audio source. Samples from some of the microphones differs from the rest, this is is especially visible on microAnalysisphone 8, 38, 40 and 56. This can be seen more clearly in Figure 9.

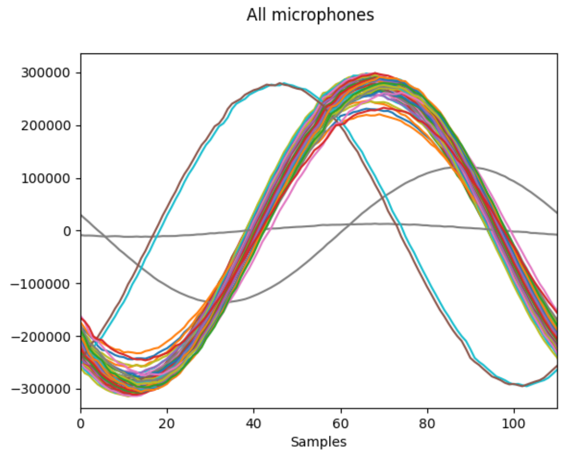

Figure 8: Sampling an 440 Hz sinus signal

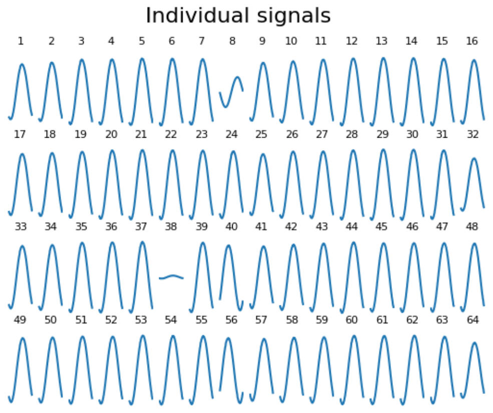

Figure 9: Microphone 8, 38, 40 and 56 differs from the rest

4 Analysis and Discussion

The FPGA implementation worked as desired and a sampling-frequency of 48 kHz was achieved.

Four of the microphones output bad data. Mic 38 seems to be broken and the sound output is very noisy. Mic 8, 40 and 56 on the other hand are all last in tier respective mic-chains which indicates that this might be due to problems with the clocks. This could be due to the array distorting the sampling pulse or that the Zybo is not being able to send out a clear enough pulse at these frequencies.

The current bottleneck is the AXI-lite communication. The PS side is able to get samples from 78 mics before data is lost due to this bandwidth limitations. This can be solved in many ways. One solution is to lower the sampling frequency, which reduces the amount of data that needs to be transferred from PL to PS. This is however not an optimal solution since the beamforming algorithms become less accurate at lower sampling frequencies. A better solution would be to use AXI-full, this makes it possible to send data in bursts which increases the bandwidth. Worth noting is that UDP packages could be transferred directly via ethernet PHY from the FPGA, but this would require Xilinx Ip blocks and it would change the structure and the complexity of the project.

To utilize the entire capacity of the 1000Mb/s ethernet, the packet sizes need to be increased by sending multiple samples in every packet. However this will not be needed unless the sample frequency is greatly increased by using other microphones or if many arrays are sampled at the same time. The Zybo Z7 will run out of logic space before the ethernet bandwidth is the bottleneck.