Audio Engineering Society

Convention Paper 10428

Presented at the 149th Convention
2020 October 27 – 30, Online

Design of Audio Processing Systems with Autogenerated User Interfaces for System-on-Chip Field Programmable Gate Arrays

Trevor Vannoy[1,2], Dylan Wickham[2], Dustin J. Sobrero[2], Connor Dack[2], Ross K. Snider[1,2], and Tyler B. Davis[2]

[1]*Electrical & Computer Engineering, Montana State University, Bozeman, MT USA 59717*
[2]*AudioLogic Inc, Bozeman, MT USA 59718*

Correspondence should be addressed to Trevor Vannoy (trevorvannoy@montana.edu)

ABSTRACT

System-on-Chip (SoC) Field Programmable Gate Arrays (FPGAs) are well-suited for real time audio processing because of their high performance and low latency. However, interacting with FPGAs at runtime is complex and difficult to implement, which limits their adoption in real-world applications. We present an open source software stack that makes creating interactive audio processing systems on SoC FPGAs easier. The software stack contains a web app with an autogenerated graphical user interface, a proxy server, a deployment manager, and device drivers. An example design comprising custom audio hardware, a delay and sum beamformer, an amplifier, filters, and noise suppression is presented to demonstrate our software. This example design provides a reference that other developers can use to create high performance interactive designs that leverage the processing power of FPGAs.

1 Introduction

High fidelity, real time audio processing requires low latency and high throughput. Field Programmable Gate Arrays (FPGAs) are high performance devices that can achieve very low latency in digital signal processing (DSP) applications. As a result, FPGAs can excel at audio processing, but they come with the big drawback of slow algorithm development.

The source of slow algorithm development on FPGAs is the use of hardware description languages (HDLs). Compared to traditional programming languages, HDLs are difficult develop audio processing systems with because they are low-level languages designed for describing digital hardware, not algorithms. Additionally, HDLs lack an ecosystem of libraries with optimized audio processing functions that are often available to traditional programming languages. All of this results in longer development times. To make developing audio algorithms on FPGAs faster, we have previously presented a model-based development framework using Mathworks' Simulink [1].

In addition to having long development times, FPGAs are difficult to interface with. System-on-Chip FPGAs (SoC FPGAs) integrate a hardened processing system, such as an ARM processor, which enables the use of

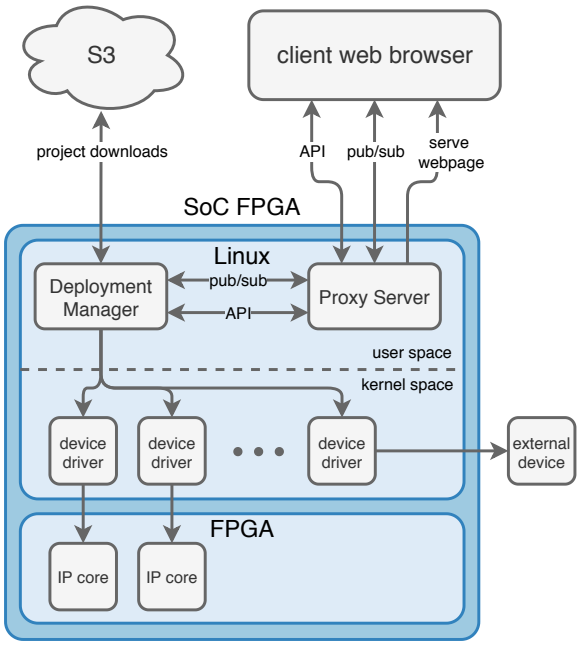


Fig. 1: Software stack overview. The client web app communicates with a proxy server. The proxy server serves the web page to the client, brokers communication between the client and the deployment manager, and acts as the publisher/subscriber broker. The deployment manager handles interactions with device registers, and it deploys projects stored on S3.

traditional software with an FPGA for control purposes. However, current methods for controlling algorithms on an SoC FPGA involve complex command line tools that can be difficult to use. To make SoC FPGAs easier to control, we present the following software stack: a web app that automatically generates a GUI for the algorithms, a proxy server that receives commands from the app, a deployment manager that installs and controls algorithms on the FPGA, and device drivers that provide an interface to algorithms running in the FPGA.

2 Software Stack

The goal of our software stack is to provide a simple framework and interface that developers can use to easily deploy and control algorithms on an SoC FPGA. An overview of our system is shown in Fig. 1. To provide a way for designs to be controlled with an autogenerated GUI, a configuration file, called model.json, is used to describe the design. This model.json file is a central piece of the architecture and allows the software stack to work for any design.

2.1 model.json

The model.json is the configuration file for the entire framework. The key pieces, with regards to the autogenerated GUI, are the devices and registers. Each device has a name and a list of registers. Each register has a name, default value, dataType object and register number. The dataType object is defined with a word length, fraction length, whether it is signed, and an optional string describing the data type. An example is shown in Listing 1 in Appendix A.

2.2 Web App

2.2.1 Controlling Deployed Designs

The cornerstone of the autogenerated user interface architecture is the web app[1]. The web app can deploy projects, control deployed projects via an autogenerated and editable user interface, and synchronize user interfaces across multiple users. It is a single page application written with React-Bootstrap which enables the UI to scale properly on any screen size. An example UI is shown in Fig. 2.

Normally, deploying projects on an SoC FPGA involves many steps, including transferring files to the Linux OS, programming the FPGA, and loading device drivers. In contrast, deploying projects from the web app is a simple one-click solution in which the web app lists all available projects, as shown in Fig. 3. Clicking the download button triggers the Deployment Manager to handle project deployment, described in Section 2.4.

2.2.2 User Interface

When loading a new project, the web app uses information from the model.json file to create a UI. A widget is created for each register, and registers are grouped into cards based upon their device. When choosing what widget type to use for a register, the app compares the register's properties—data type, min, max, and step size—against a list of widget types and chooses the best match. As of this writing, the available widget types are selectable buttons and sliders.

[1] https://github.com/fpga-open-speech-tools/autogen_webapp

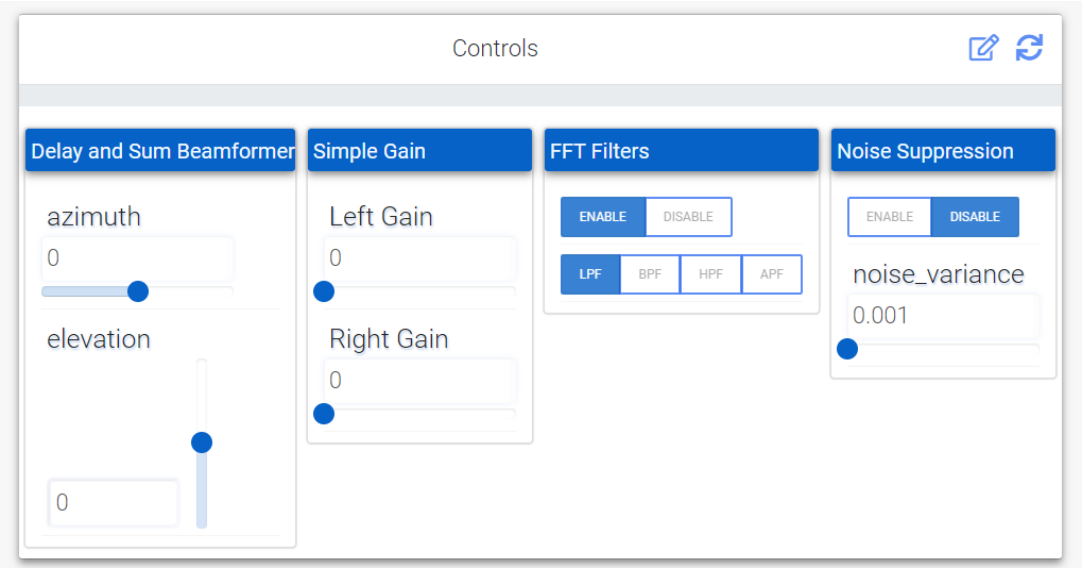


Fig. 2: Example Controls for the Beamer presented in Section 3

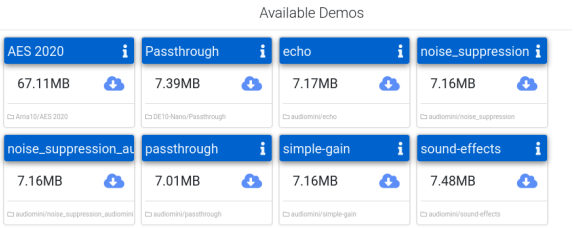


Fig. 3: The Available Demos Section of the Web App

After a UI widget is generated, users can edit it to better fit their needs. Each widget is directly modifiable via the editor dialog shown in Fig. 4. The editor allows users to choose alternative widget types that match the register's properties, (e.g. name, units, min, max, and step size) as well as change the widget's display variant (e.g. changing a horizontal slider to a vertical slider).

The autogenerated UI widgets are used to control the deployed design. Each widget has an event listener that sends the widget's value to the publisher-subscriber hub on the proxy server. For sliders, users can edit values by dragging the slider or entering values directly into a textbox.

Since the web app uses a publisher-subscriber framework, the user interface is synchronized between users. For example, when one user changes a slider value, all other users will see the same slider value change in their web app. This makes it easier for multiple users to interact with the same design since all users will always have an up-to-date representation of the device's state.

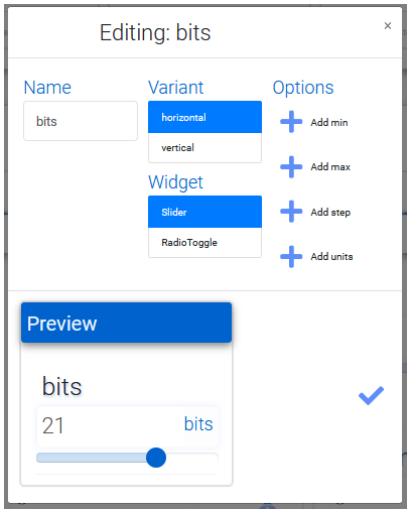


Fig. 4: The Widget Editor Dialog

2.3 Proxy Server

The proxy server[2] serves the web app to clients, brokers communication between the web app and the deploy-

[2] https://github.com/fpga-open-speech-tools/autogen_webapp

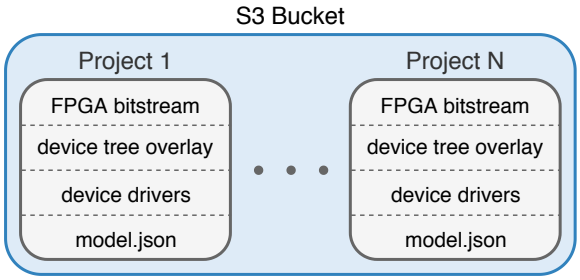


Fig. 5: How designs are stored in S3. In an S3 bucket, an arbitrary number of projects can be stored as folders. Within each project folder, an FPGA bitstream, device tree overlay, model.json, and device drivers are stored.

ment manager, and handles publisher-subscriber connections. For one-off communication between the web app and the deployment manager, such as requesting project deployment, the proxy server forwards the message to the recipient. A publisher-subscriber framework implemented in SignalR is used to handle algorithm parameter updates from the web app. Multiple web app clients can connect to the SignalR hub to control the device.

2.4 Deployment Manager

The Deployment Manager[3] is responsible for controlling algorithm parameters and deploying projects to the SoC FPGA.

2.4.1 Controlling Projects

When a client changes a value in the web app, the Deployment Manager receives an update from the proxy server's SignalR hub. An update contains the device name, device attribute that corresponds to a tunable parameter, and the value. After the update is decoded, the FPGA register is updated by a device driver. The Deployment Manager uses the model.json to determine what algorithm parameters it can control.

2.4.2 Deploying Projects

The Deployment Manager can take designs stored on Amazon Web Service's Simple Storage Service (S3), as shown in Fig. 5, and load them onto the SoC FPGA at runtime. Deploying projects at runtime requires the use of device tree overlays. A device tree is a data structure containing nodes that describe the hardware components (e.g. CPUs, peripherals, etc.) in an embedded system [2]. Each node has properties that specify device attributes and device driver compatibility. Device tree overlays are a special type of device tree node that can modify the kernel's live device tree [3].

The sequence of steps for deploying a project is as follows:

1. The Deployment Manager downloads project files from S3
2. The overlaymgr installs the overlay and programs the FPGA
3. The drivermgr loads the relevant device drivers

S3 When the Deployment Manager receives a request from the web app to download a project from S3, it calls a Python script[4] that downloads the files shown in Fig. 5 and places them in their desired locations. The bitstream and device tree overlay are placed in `/lib/firmware/`, and the device drivers are placed in `/lib/modules/<project_name>/`.

Overlay Manager Once a project is downloaded, the overlaymgr bash script[4] handles installing the project via device tree overlays, like the one shown in Listing 2 in Appendix A. Using the configfs overlay interface [4], the overlaymgr creates a directory for the overlay:

```
/sys/kernel/config/device-tree/
↪ overlays/<project_name>
```

Linux then creates several files in previously mentioned directory. The overlay is applied by writing the device tree overlay's filename to

```
/sys/kernel/config/device-tree/
↪ overlays/<project_name>/path
```

Removing an overlay is done by removing the the `<project_name>` directory. For more information on programming FPGAs with device tree overlays, see [5, 6].

[3] https://github.com/fpga-open-speech-tools/deployment_manager

[4] https://github.com/fpga-open-speech-tools/utils

Driver Manager Once an overlay is installed, device drivers need to be loaded into the kernel. The drivermgr script[4] manages installing and removing device drivers for the projects. It takes the project name as an input to determine which drivers to load/remove; all drivers for a given project live in `/lib/modules/<project_name>`. When loading a project, drivermgr loads all drivers in `/lib/modules/<project_name>` using `insmod`; the same is done using `rmmod` when removing drivers.

2.5 Device Drivers

In order to facilitate run time configuration of algorithm parameters, device drivers are created to communicate these parameter updates between the HPS and FPGA. Device drivers are generated based upon our work in "An Open Audio Processing Platform Using SoC FPGAs and Model-Based Development" [1].

3 An Example Design: The Beamer

A single example design, the Beamer, was developed to showcase the software stack controlling an audio processing algorithm. The Beamer, shown in Fig. 6, shows the processing blocks required to collect data from the AudioLogic Microphone Array, then filter the signal and/or reduce the noise from an audio source. The example design utilizes the flexibility of the autogenerated GUI to provide the user with a simple interface to control each processing element, as seen in Fig. 2.

The runtime programming discussed in Section 2.4.2 was not used in this example design because the Arria 10 SoC FPGA does not support full FPGA reconfiguration from Linux, and we have not yet tested partial reconfiguration.

The hardware implementation is discussed in Section 3.1 while the GUI and processing blocks of the Beamer are describe in Section 3.2. The Beamer was implemented in Simulink using the workflow described in our previous work [1].

3.1 Hardware System

The Beamer was implemented using the AudioLogic Microphone Array and the AudioLogic Arria 10-based Audio Research Daughter Card[5], shown in Fig. 7. The Microphone Array is a square 16 element, 4x4 micro-electro-mechanical systems (MEMS) microphone array spaced 25mm apart. The Audio Research Daughter Card connects to the Reflex Achilles Arria 10 SoM via the Low Pin Count FMC Connector. The Audio Research provides standard audio interfaces such as line in and line out along with the Microphone Array RJ45 connector. The Microphone Array connects to the Audio Research using a standard Ethernet cable that provides power and differential data lines. The Arria 10 communicates bidirectionally with the Microphone Array using serialized packets to send and receive data and control information.

3.2 Audio Processing Control

Each processing element of The Beamer was controlled and verified using the autogenerated GUI.

3.2.1 Beamforming Control

The delay-and-sum beamformer exposes an azimuth and elevation parameter which are implemented in the GUI as horizontal and vertical slider, shown in Fig. 8.

When the slider is moved, the web app sends a command containing the register name and value to the proxy server on the embedded Linux OS of the Arria 10. The proxy server then relays the command to the Deployment Manager while simultaneously updating the other subscribed web apps. Finally, the Deployment Manager communicates with the beamformer's device driver which then updates the register of the FPGA. The updated register in the FPGA is then used by the beamformer to change the sample delays from each microphone and thereby steer the beam.

In this example, the beam was steered so that a null would be placed at the sound source, as shown in Fig. 8, located directly in front of the array. This should have the effect of significantly attenuating the 10.0 kHz tone. Two amplitude measurements were taken to test this hypothesis; the first when the beam was steered to 0.00° and the second when the beam was steered to 43.5°, the null of the Microphone Array pattern. When the amplitudes were compared, the tone was attenuated by approximately 23.2 dB. This shows that the beamformer and, more importantly the interface used to control it, works as intended.

[5] `https://github.com/fpga-open-speech-tools/hardware/`

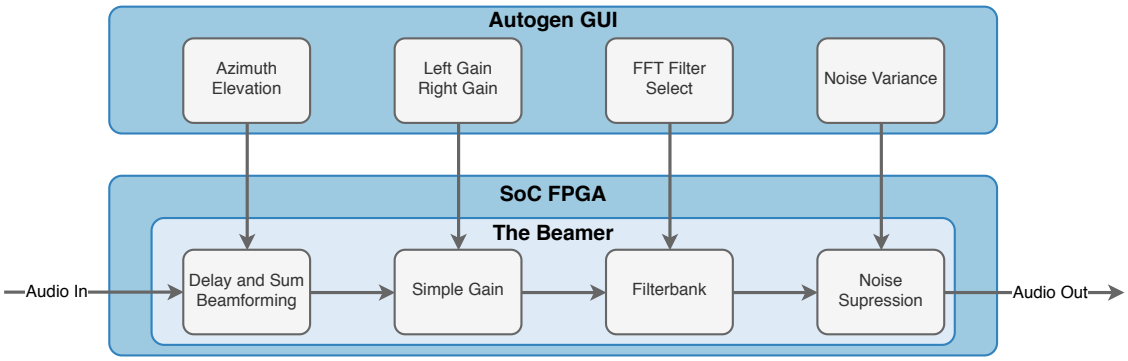


Fig. 6: The Beamer Processing Block Diagram

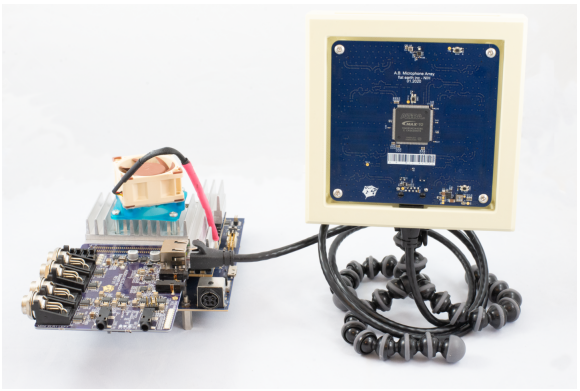


Fig. 7: The Audio Research Arria 10 Daughter Card connected to the AudioLogic Microphone Array

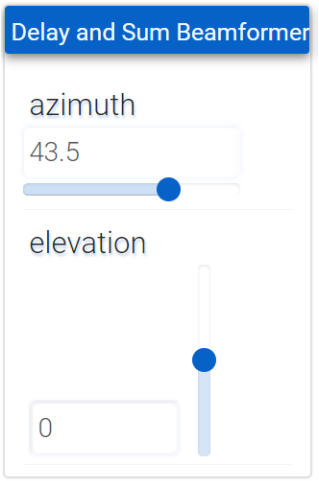


Fig. 8: Beamforming controls with showing the azimuth angle so a null is steered toward the audio source.

3.3 Filter Selection Control

The GUI is also capable of generating a list of selectable buttons. Such an interface is useful when the processing component contains a small number of values or a qualitative list. In this example, the Beamer includes four filters: low-pass, band-pass, high-pass, and all-pass. The GUI interface for this processing system can be seen in Fig 9.

The enumerated list uses buttons instead of sliders, however the communication channels remain the same. The main difference occurs in the GUI when the button descriptions are translated from a user-friendly representation to the representation the device driver uses. For example, the filter options are represented as a number between 0 and 3 in the FPGA but as the filter name in the GUI.

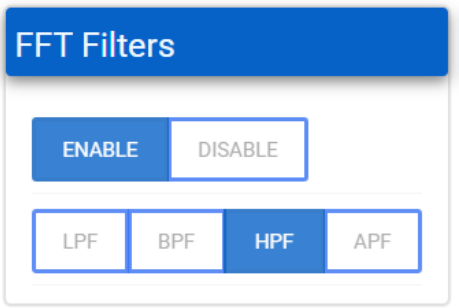


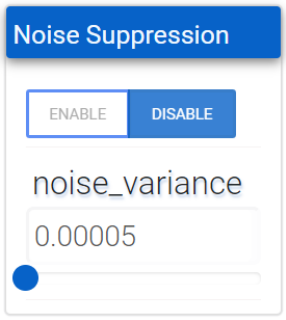
Fig. 9: Filter Controls with the High-Pass Filter Selected

3.3.1 Noise Suppression Control

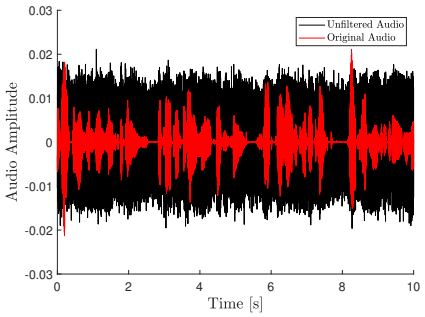
The sliders and the toggle buttons can also be combined in GUI components. For example, the noise suppression component can be enabled or disabled and the "amount" of noise suppression can be varied. The Beamer's noise suppression component is designed to remove normally distributed white noise from an audio signal in order to improve the clarity of speech and is based on the algorithm presented in "Speech enhancement with an adaptive Wiener filter" [7]. The controls of this processing component are shown in Fig. 10a and Fig. 11a.

To show the control in use, zero-mean white Gaussian noise with a standard deviation of 0.0167 was added to a prerecorded segment of audio that included a woman speaking various simple phrases. The noisy audio was then played through a speaker and recorded using the AudioLogic Microphone Array. Fig. 10b shows the unfiltered audio with the original audio overlaid. Despite it's appearance, the original audio can still be heard through the noise.

The audio was then filtered by enabling the noise suppression in the GUI and using a noise variance of 0.00005. This value was found experimentally to be sufficient to reduce the amount of noise in the signal without attenuating the speech as well. As shown in Fig 11b, the noise is significantly reduced.



(a) Noise Suppression Control when the Suppression is Disabled



(b) Plot of the unfiltered noisy audio versus the original audio.

Fig. 10: The noise suppression control and the associate time-domain representation of a noisy speech signal when the speech is unfiltered.

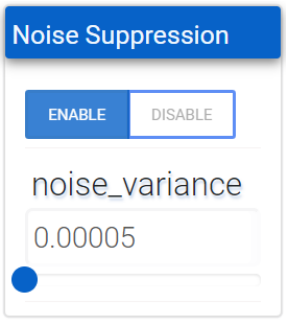
4 Conclusion

We presented an open source software framework that makes deploying and controlling audio algorithms on an SoC FPGA an easy and automated process. This framework consists of a reconfigurable web application, a proxy server to broker communication, autogenerated device drivers, and a deployment manager. This software stack successfully streamlines the process of creating an audio processing system and provides a convenient method of controlling algorithms running on the SoC FPGA in real time.

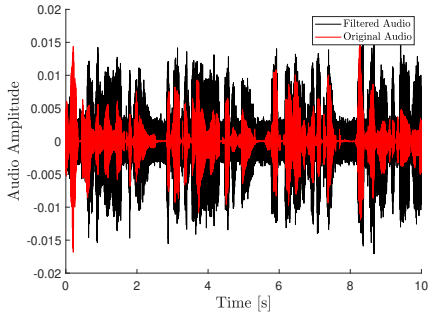
The framework also adapts to the design deployed on the device, allowing the it to be used for any number of designs. The automated deployment process makes loading and changing between SoC FPGA projects a one-click process that leverages Amazon S3—something that we have not seen elsewhere in the FPGA or audio communities. The example design shows that the software stack facilitates control of audio algorithms on target hardware and is able to easily adapt to different designs. This extension we have presented to our Open Audio Processing Platform [1] makes SoC FPGAs a viable platform for audio processing development that can now compete with the ease of use of more traditional audio processing platforms.

Though this design simplifies the complicated process of developing audio processing systems on an SoC FPGA, several avenues of advancement are possible:

- adding more UI widget types
- adding support for programming partial FPGA regions so designs can be combined at run time with the web app
- extending the software stack to support other embedded Linux devices, such as the Analog Devices ADSP-SC589 that contains an ARM processor and two SHARC DSPs



(a) Noise Suppression Control when the Suppression is Enabled



(b) Plot of the filtered noisy audio versus the original audio.

Fig. 11: The noise suppression control and associated time-domain representation of a noisy speech signal when the speech is filtered.

Acknowledgments

This work was supported by the National Institutes of Health - Grant No: 4R44DC015443.

References

[1] Vannoy, T., Davis, T., Dack, C., Sobrero, D., and Snider, R., "An Open Audio Processing Platform Using SoC FPGAs and Model-Based Development," in Audio Engineering Society Convention 147, Audio Engineering Society, 2019.

[2] Simmonds, C., Mastering embedded Linux programming, Packt Publishing Ltd, 2015.

[3] Linux Kernel Maintainers, "Device Tree Overlay Notes," https://www.kernel.org/doc/html/latest/devicetree/overlay-notes.html, 2020.

[4] Antoniou, P., "Howto use the configfs overlay interface," https://github.com/altera-opensource/linux-socfpga/blob/socfpga-5.4.34-lts/Documentation/devicetree/configfs-overlays.txt, 2013.

[5] Linux Kernel Maintainers, "FPGA Region Device Tree Binding," https://github.com/altera-opensource/linux-socfpga/blob/socfpga-5.4.34-lts/Documentation/devicetree/bindings/fpga/fpga-region.txt, 2018.

[6] Tull, A., "Linux FPGA Subsystem Documentation," https://www.kernel.org/doc/html/latest/driver-api/fpga/index.html, 2020.

[7] El-Fattah, M. A. A., Dessouky, M. I., Abbas, A. M., Diab, S. M., El-Rabaie, E.-S. M., Al-Nuaimy, W., Alshebeili, S. A., and El-samie, F. E. A., "Speech enhancement with an adaptive Wiener filter," International Journal of Speech Technology, 17(1), pp. 53–64, 2013, doi:10.1007/s10772-013-9205-5.

A Code Examples

```
    "name":
     ↪  "delay_and_sum_beamformer",
    "registers": [
        {
            "dataType": {
                "fractionLength":
                 ↪  8,
                "signed": true,
                "type":
                 ↪  "sfix16_en8",
                "wordLength": 16
            },
            "defaultValue": 0,
            "name": "azimuth",
            "registerNumber": 0
        },
        {
            "dataType": {
                "fractionLength":
                 ↪  8,
                "signed": true,
                "type":
                 ↪  "sfix16_en8",
                "wordLength": 16
            },
            "defaultValue": 0,
            "name": "elevation",
            "registerNumber": 1
        }
    ]
},
```

Listing 1: Section of `model.json` that describes the delay and sum beamformer component.

```
/dts-v1/;
/plugin/;

&base_fpga_region {
  firmware-name = "aes.rbf";

  #address-cells = <2>;
  #size-cells = <1>;
  ranges = < 0x00000000 0x00000000
   ↪  0xc0000000 0x20000000>,
  <0x00000001 0x00000030 0xff200030
   ↪  0x00000008>,
  <0x00000001 0x00000020 0xff200020
   ↪  0x00000008>;

  fft_filters@100000030 {
    compatible =
     ↪  "fe,fft_filters-1.0",
     ↪  "dev,al-fft_filters";
    reg = <0x00000001 0x00000030
     ↪  0x00000008>;
  };
```

Listing 2: An excerpt of the device tree overlay used in Section 3. The FPGA gets programmed with the `"aes.rbf"` bitstream. The `ranges` property defines the address mapping for where the FPGA devices are located. The `reg` property of each device specifies the base address and range of the device's registers.