OPEN-SOURCE TEST-BENCH DESIGN FOR APPLICATIONS IN AUTONOMOUS ULTRASOUND IMAGING

by

ALEX D. ROMAN

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2019

Open-Source Test-Bench Design for Applications in Autonomous Ultrasound Imaging

Case Western Reserve University
Case School of Graduate Studies

We hereby approve the thesis[1] of

ALEX D. ROMAN

for the degree of

Master of Science

Dr. Soumyajit Mandal

Committee Professor, Adviser — March 26th, 2019
Department of Electrical Engineering and Computer Science

Dr. Francis Merat

Committee Professor — March 26th, 2019
Department of Electrical Engineering and Computer Science

Dr. Farhad Kaffashi

Committee Professor — March 26th, 2019
Department of Electrical Engineering and Computer Science

[1] We certify that written approval has been obtained for any proprietary material contained therein.

Dedicated to my family for
their unending support.

Table of Contents

List of Tables

List of Figures

Acknowledgements

0.1 Acknowledgements

I want to thank my research advisor, Dr. Soumyajit Mandal, for his support and patience with my research and studies during my time at CWRU. He has guided over the last few years have provided me with many insights towards my work, and I continue to learn a great deal from him.

Also, I would also like to thank my parents and my sisters for their unconditional love and support in me during my time at the university. The continuous encouragement from all of you, helped me stay the course and motivated me to finish what I started.

I want to thank all of my friends that I have made over the years in ICSP Lab. I specifically want to thank David Ariando, Vida Pashaei, and Parisa Dehghanzadeh for their assistance and feedback on this project. Without all of your contributions and input with this project, I couldn't have pushed this system to its current status.

Also, I would like to thank the other members of the ICSP lab, who were not directly involved with this, but helped me with their opinions and insights. The guidance and perspectives that we had over the years provided a better outside perspective regarding this project and always reminds me that one needs to be flexible with research. Without all of you, I wouldn't have been able to complete this thesis, and I am grateful for the times we shared.

I would also like to thank Professor Francis Merat and Professor Farhad Kaffashi for being members of my committee and reviewing this thesis. Also, I appreciate all the academic support and personal advice that you both have provided me during my collegiate years at CWRU. Many of our conversations have had a significant impact on my professional and personal development, and I am glad for your support.

Abstract

Open-Source Test-Bench Design for Applications in Autonomous Ultrasound Imaging

Abstract

by

ALEX D. ROMAN

0.2 Abstract

Over the last few years, there has been an increased interest in the use of ultrasound in a variety of research applications. Due to this increased interest, there is a need for the development of customized transmission, reception, and processing algorithms, along with low-cost programmable ultrasound imaging devices that would be more desirable for the variety of research applications. This thesis describes a programmable 64-channel system-on-chip (SoC)-based test bench that will be utilized for future research and development on autonomous wearable and implantable medical ultrasound imaging applications. The current setup consists of a custom transmitter, a custom high-voltage (HV) multiplexer, an off-the-shelf receiver, and embedded software on a System on Chip. This prototype results in a modular design that can be easily updated with improved hardware and software for different applications. This thesis describes the current system hardware and software design along with the initial imaging results on tissue phantoms.

1 Introduction

1.1 Motivation

Ultrasound is a common medical imaging modality known for its versatility, low cost, and minimal risk to the patient. During an ultrasound scan, high-speed data acquisition and processing is required to provide real-time information to the physician or operator. This task is usually accomplished via a PC-based system requiring special add-on processing cards. Recent performance advancements in field-programmable gate arrays (FPGAs), digital signal processors (DSPs), and systems-on-chip (SoCs), along with the development of integrated front-end circuits for medical ultrasound applications, have accelerated the creation and availability of programmable research platforms to verify and refine techniques by manipulation of transmission, reception, and beam-forming methods.

The main focus of this thesis is explaining the overall system development of a programmable low-cost 64-channel SoC-based test-bench for research on ultrasound imaging. SoCs are attractive for this application because they combine an FPGA fabric (for real-time processing) with a hard processor (for running the user interface, networking,

etc.) within the same package. On the other hand, the modular design of the system enables easy control of the hardware and software configuration for various applications, including implantable and wearable imaging devices.

1.2 Previous Work in Ultrasound Feasibility in Autonomous Applications

Previous colleagues from Case Western Reserve University (Abhishek Basak, Vaishnavi Ranganathan, and Dr. Swarup Bhunia) worked on several feasibility studies to show the viability and necessary system configurations needed for independent and continuous monitoring applications.

In "A Wearable Ultrasonic Assembly for Point-of-Care Autonomous Diagnostics of Malignant Growth"[11], and "Implantable Ultrasonic Imaging Assembly for Automated Monitoring of Internal Organs"[12], our peers focused on the feasibility of ultrasound to be used as a means of the early detection of anomalous growths via automated high-resolution monitoring either through a wearable or implantable package. Their proposal focused on the development of a point-of-care ultrasonic imaging assembly to perform unsupervised, periodic monitoring of soft-tissue organs (e.g., the prostate, uterus, kidney, ovary, and bladder) via an external wearable unit or an implanted unit. The potential benefits that they mentioned from these systems included low-cost, miniaturization, and amenability for continuous online monitoring for each of their proposals.

This thesis extends upon their earlier results by describing and designing the hardware and software components of a complete open-source prototype, along with the recorded imaging results on tissue phantoms.

1.3 Basics of Ultrasound Imaging

Most ultrasound imaging devices use a pulse-echo approach with a brightness-mode (B-mode) display. The B-mode imaging method uses the transmission of an ultrasound echo pulse that travels directly into a body/phantom via a transducer array. The generated waves propagate into the body/phantom until they are reflected back to the transducer. These echo reflections will occur based on the changes in the acoustic impedance from one tissue to another within its path (e.g., fat to bone/lung tissue to air). The echoes that return to the transducer will have a reduced intensity compared to the initial pulse, due to the conversion of the mechanical waves to heat; this is known as absorption. Once the echo signals return to the transducer, they will be recorded as digital signals and be processed to construct an image. This generated image provides two-dimensional information (lateral and axial distances) of any objects detected in the body of the phantom, where the axial distance is the depth of the object within the phantom, and the lateral length is the location perpendicular to the axial line.[13,14]

1.3.1 Generation of Ultrasound Pulses

For the generation of ultrasound waves, most transducers use piezoelectric crystals to convert electrical energy into mechanical vibrations, which is known as the piezoelectric effect. Conversely, for detecting these waves, the same crystals are used to re-convert mechanical vibrations to electrical energy, which is known as the reverse-piezoelectric effect[2,14,15]. The mechanical vibrations of these ultrasound waves generated by the piezoelectric elements create areas of compression and rarefaction that will propagate through the body/phantom during a scan. The characteristics of these waves include

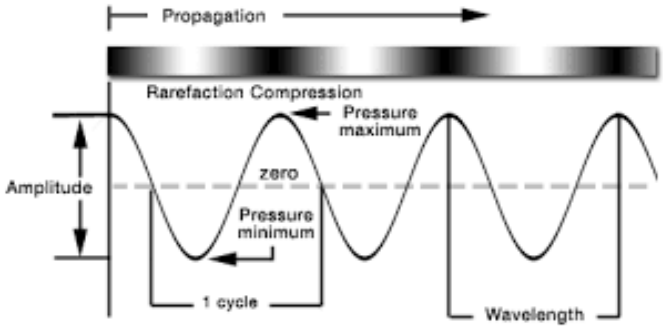

Figure 1.1. Sound wave characteristics, including compression and rarefaction zones.

the parameters of amplitude (in dB), wavelength (in mm), and frequency (in MHz). Fig. 1.1 illustrates these characteristics and zones of compression and rarefaction.

1.3.2 Wavelength and Frequency

Medical ultrasound waves have frequencies within the range of 1-20 MHz. While this is a significant range, the frequency of the generated ultrasound pulse, is adjusted to a center frequency and bandwidth based on the construction of the transducer to provide the best signal gain for the application. These parameters usually depend on the physical parameters of the transducer and the fabrication of the individual elements. (e.g., size of the elements, the distance between elements, the pitch of elements). Also, while the above frequencies are classified as ultrasound, there are subcategories of ultrasound waves dedicated to specific applications, namely high-frequency ultrasound waves (10-15 MHz) and low-frequency ultrasound waves (2-5 MHz). High-frequency waves are used to generate images of high spatial resolution, due to their shorter wavelength. While this higher resolution allows for better discrimination between separate structures within a body/phantom, but the drawback is that these waves will suffer more

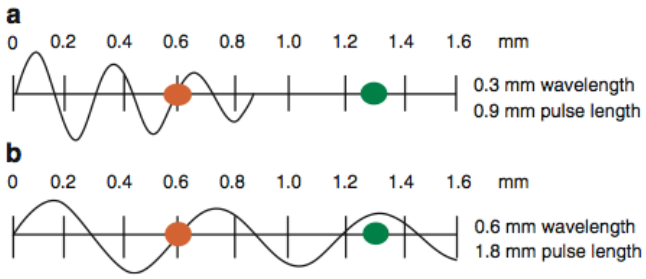

Figure 1.2. Attenuation of ultrasound waves and their relationship to wave frequency. Figure comes from ref.[1]

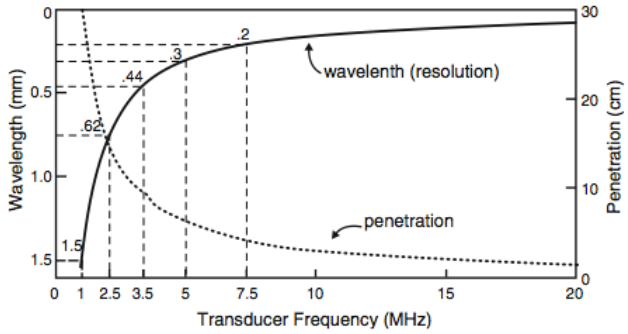

Figure 1.3. A comparison of the resolution and penetration of different ultrasound transducer frequencies. This figure was published in ref.[2]

absorption losses when traveling through the body/phantom. On the other hand, low-frequency waves don't suffer as much absorption over the same distance, but at the cost of lower spatial resolution within the resulting data.[16] As a result of these physical properties, transducers with a center frequency and bandwidth based on the high-frequency ultrasound waves are more suited for scanning structures closer to the surface of the body/phantom, while transducers developed within the low-frequency range are more suited for the scanning of deeper structures.

Table 1.1. Acoustic impedances of different body tissues and organs. Values sourced from ref.[1]

Body Tissue	Acoustic Impedance (10^6 Rayls)
Air	0.0004
Lung	0.18
Fat	1.34
Liver	1.65
Blood	1.65
Kidney	1.63
Muscle	1.71
Bone	7.8

1.3.3 Ultrasound-Tissue Interactions

Echo Generation - Acoustic Impedance. Once the generated waves go from the transducer into the body/phantom, the most important waves are the ones echoing back to the transducer. These echoed waves that come back to the transducer will be smaller in intensity compared to the initial ultrasound because of attenuation within the body/phantom. This attenuation of the wave happens from a combination of the varying acoustic impedances between multiple tissues along with the travel distance. Acoustic impedance Z_0 is an intrinsic physical property of a material, defined as $Z_0 = \rho c$ where ρ is the density of the material and c is the ultrasound velocity within the material. As a result, higher density organs such as bone will have a high acoustic impedance, while lower density organs like the lung will have a low acoustic impedance[17]. Table 1.1 shows the acoustic impedances of several body tissues.

The intensity of a reflected echo is proportional to the fractional difference in acoustic impedance between two adjacent tissues. Tissues/material that have the same acoustic impedance should not generate an echo. Therefore, two body tissues with a slight variance between their acoustic impedances will create low-intensity echoes, while two tissues with significant variation in acoustic impedance, like soft tissue to bone or soft tissue to air, will cause a strong echo.

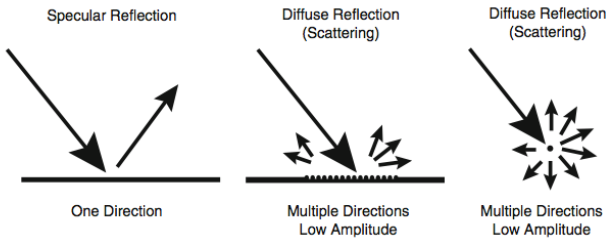

Figure 1.4. Different types of ultrasound wave-tissue interactions. Reproduced with permission from ref. [1]

Sound Waves - Reflection and Refraction. On top of absorption, there are a few other wave effects that are a significant impact in ultrasound imaging, which are specular reflection, diffuse reflection, and refraction shown in Fig. 1.4. First, specular reflection occurs when the echo intensity generated off the border of two different tissues is proportional to the acoustic impedance gradient between them, shown in Table 1.1. In this event, the incoming ultrasound wave needs to approach the boundary of the two tissues at 90 degrees. When this occurs, the generated echo will have full intensity based on the difference in acoustic impedances. An approach angle that isn't at this 90-degree angle will result in a reduced echo intensity.

On the other hand, if the ultrasound pulse encounters an object that has dimensions that are smaller than its wavelength, or interacts with an irregular tissue interface, there will be a reduction in the echo intensity due to the diffuse reflection/scattering of the echo. While this scattering may reduce the strength of the overall echo, these diffused echoes will often reach the transducer, which ends up providing some texture and feature information of the boundaries between tissues/organs.

Refraction refers to the change in the direction of the ultrasound pulse after interacting at the boundary of two tissues, due to variance in the speed of sound in each tissue. Since the ultrasound pulse has a constant frequency, the wavelength of the ultrasound

beam will change to accommodate this difference in the speed of sound transmission between the two tissues, which results in the redirection of the traveling ultrasound pulse as it moves through this boundary. This redirection due to refraction can cause the incorrect localization of a structure in an ultrasound image, an example of this is at fat/soft tissue interfaces where the speed of sound goes from 1450 m/s to 1540 m/s[14].

Attenuation. As mentioned previously, when ultrasound pulses travel through tissue, there will be a reduction in their intensity due to absorption. The overall attenuation of an ultrasound pulse is the result of wave-tissue interactions and friction-like losses (absorption). These absorption losses result from the induced oscillatory tissue motion produced by the pulse, which causes conversion of energy from the original mechanical form into heat. This energy loss from absorption is the most important contributor to ultrasound attenuation. Additional things that result in greater attenuation effects are a longer travel distance for the echo and the use of higher frequency waves. Attenuation also varies among body tissues, with the highest degree in bone, less in muscle and solid organs, and lowest in blood for any given frequency. Most ultrasound devices increase the gain (brightness/intensity) of signals from deeper areas of the scan to address this attenuation from tissue-interactions effect from scanning depth.

1.4 Literature Review

Multiple research ultrasound systems have been designed to meet the need for platforms to develop new transmission, reception, and beam-forming algorithms. Here we are interested in reviewing the parameters of these systems, and what they provide to their users.

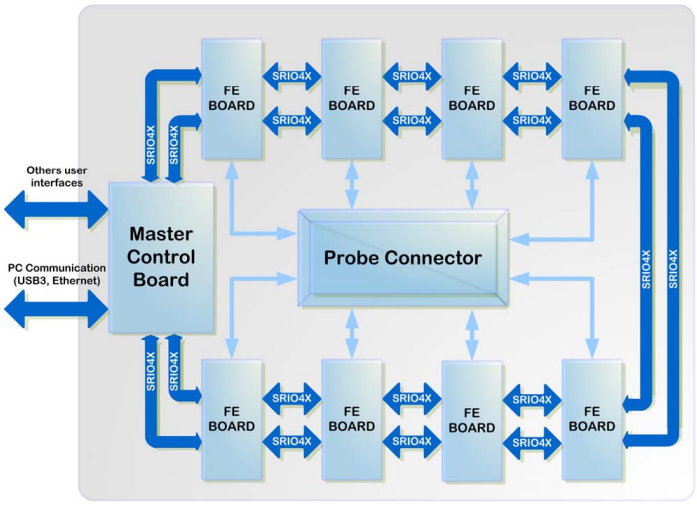

Figure 1.5. General architecture of ULA-OP 256, from ref.[3].

1.4.1 ULtrasound Advanced Open Platform - ULA-OP

The first system is the ULtrasound Advanced Open Platform - ULA-OP 256 system, developed at the University of Florence, Italy[3]. The system featured a modular architecture which integrated the necessary electronics for the storage of large amounts of ultrasound data along with the processing power to perform computationally intensive image processing algorithms. Their total system provides a total of 256 active channels, through the connection of eight custom front end active channel boards (32 channels each) interconnected through a SerialRapidIO link. In its full configuration, the system acquisition clock is synchronized to all eight boards and runs at a frequency of 78.125 MHz. Fig. 1.5 shows the general architecture of their system.

ULA-OP 256 Hardware. As mentioned in the previous section, the individual front-end module board of the ULA-OP 256 integrates the components for transmission, reception, and real-time processing for 32 active channels. The board runs off an Altera ARRIA

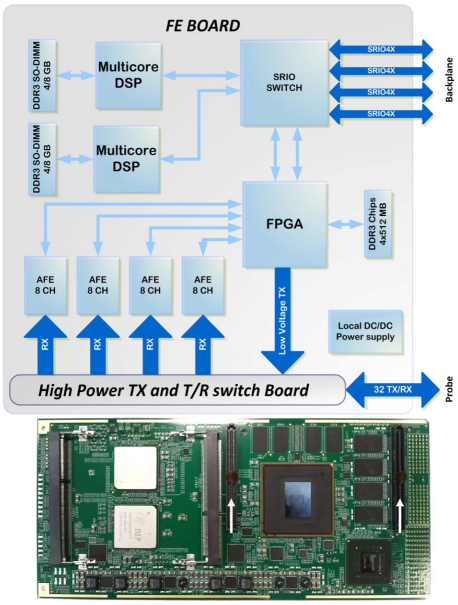

Figure 1.6. Front-end board of ULA-OP 256: block diagram and physical board, from ref[3].

V GX family FPGA and is programmed to generate 32-bit streams in the low-voltage differential signaling (LVDS) format, for conversion into an analog waveform for transmission via a low-pass filter and a power amplifier for each of the 32 connected piezoelectric elements. Their front-end (FE) utilizes integrated transmit/receive (T/R) switches to electronically steer the 32 ultrasound echo signals back to the system, to amplify and digitize them at 78.125 MSPS with 12-bit resolution via four eight-channel ultrasound FE integrated circuits (AFE5807, Texas Instruments). Along with this, the FE board contains two DSPs from the 320C6678 family to handle real-time processing operations like compounding, cross-correlation, and fast Fourier transforms. In order to provide users space to collect large amounts of ultrasound data for offline evaluation and analysis, the system provides 10 GB of memory storage per FE board through the installation of two standard 4-GB DDR3 SODIMMs combined with the 2 GB capacity of the beamforming FPGA.

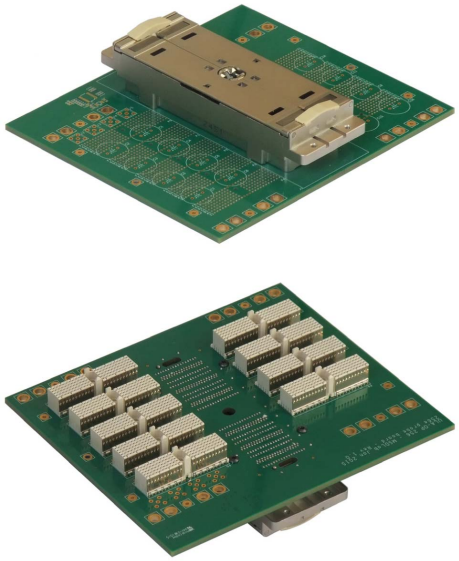

Figure 1.7. Probe adapter board, from ref.[3].

The ULA-OP system utilizes a custom internal probe adapter board to change between multiple probe configurations. This system contains a specific probe connector that routes toward an array of high-speed shielded connectors fitted on their system backplane. Their design allows for easy removal and replacement by pulling it out of the arrayed connectors.

ULA-OP Software. The ULA-OP 256 was designed to be managed by a modular and re-configurable C++ software that is running on a separate host PC. Their software initializes the hardware upon start-up, provides a user-friendly interface, and displays the results of ultrasound data/image processing on the screen of the host PC.

The user can adjust text configuration files for the main start-up settings for their system. When configured, the acquisition begins and provides the real-time processing results on the user's screen. Depending on the selected configuration, the software can give multiple display frames, where each one designated to a different processing module. Also, it offers an user interface with a toolbar and user controls to allow the user to

freeze the system, save the acquired signals, or start screen capturing, while also allowing the modification of additional parameters for the processing modules of the system through different control panels.

The updates to the system configuration and parameters transfer via a USB 3.0 link to the DSP modules on the ULA-OP, where each DSP interprets the command and modifies itself accordingly. The system software manages the real-time acquisition, processing, and streaming of the ultrasound data from the system to the host PC as concurrent processes. The system continuously saves the ultrasound data onto the provided DDR memory, until the ULA-OP terminates the acquisition. Once ended, the current protected data stored into binary files. The current configuration of the system allows continuous data streaming, but is limited by the bandwidth of the USB 3.0 connection between the system and the host PC.

1.4.2 Embedded High-Performance Ultrasonic Signal Processing System

The second system we consider is the Embedded High-Performance Ultrasonic Signal Processing System (EUSP), developed at Sichuan University, Chengdu, Sichuan, China[4]. This project focused on the development on an embedded-ultrasound signal processing system to provide real-time processing of B-mode, color flow, and spectrum Doppler signals through the use of four high-performance DSPs so that multiple methods can achieve an acceptable frame rate.

EUSP - Hardware. The overall system architecture of the EUSP involved the following subsystems: 1) a digital front-end connected to an ultrasound transducer to provide beamforming and real-time control; 2) a mid-processor subsystem to provide signal processing of the front end signals, which results in a decimated envelope and Doppler I/Q data sent to the EUSP; 3) the main EUSP which is meant to be a programmable back

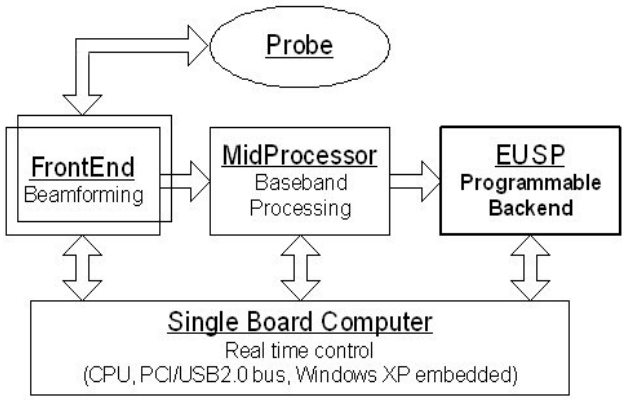

Figure 1.8. Block diagram of the Embedded Ultrasonic Signal Processing (EUSP) system, from ref. [4].

end, which extracts the necessary information for the mixed modes data and converts it into video/audio format based on the application; and 4) a single board computer (SBC) to receive the video/audio formatted data from the EUSP to display it to the user, while also providing support as the user interface for data measurement, reporting, transfer, and control to the other subsystems.

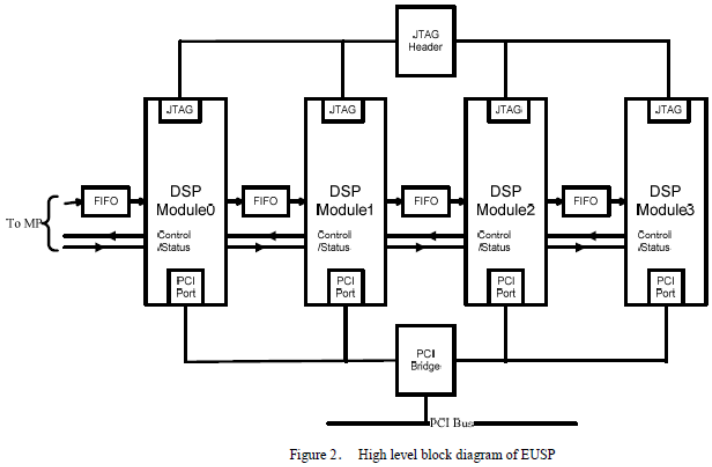

Figure 2. High level block diagram of EUSP

Figure 1.9. High-level block diagram of EUSP, from ref. [4].

Fig. 1.9 shows the high-level block diagram of the EUSP subsystem, which is the main focus of this system. The configuration uses four DSP modules to support the high computational requirements needed for multimodal ultrasound signal and image processing. The DSP modules of the EUSP utilize a Texas Instruments (TI) fixed-point digital signal processor (TMS320C6416) and 64MB SDRAM with 150 MHz access speed. The TMS320C6416 is a high-performance fixed-point DSP, which provides a performance of up to 8000 million instructions per second (MIPS) at a clock rate of 1 GHz. Also, the TMS320C6416 core processor provides 64 general-purpose registers of 32-bit word length and eight highly independent functional units - two multipliers for a 32-bit result, and six arithmetic logic units. The EUSP also contains a 16-bit wide FIFO with speed up to 100 MHz, which provides 200 MB/s data throughput between every two DSPs. The DSP connects to two memory interfaces which are the FIFO and the SDRAM, to allow access to them at the same time, which provides for the possibility to process data in SDRAM while reading/writing data from/to FIFO on background simultaneously.

EUSP-Software. Based on the hardware architecture, the four DSPs in the EUSP are chained serially by FIFOs. This configuration forces the ultrasound signal data to process through the four DSPs one at a time, creating a four-stage pipeline. As a result, the processing functions are divided into sub-functions and distributed across the four DSPs. Also, the system utilizes a multi-task real-time operating system (RTOS) that provides short interrupt routines to handle the urgent needs of the hardware and set signals, eliminate the use of a processing loop, and provide the ability to select what runs next based on a flexible priority scheme.

The EUSP utilizes DSP/BIOS as the real-time operating kernel; this is a scalable real-time kernel freely-provided by TI for applications that require real-time scheduling and

synchronization, host-to-target communication, or real-time instrumentation. This TI DSP/BIOS provides preemptive multi-threading, hardware abstraction, and real-time analysis. Based on their selection of the C6416 DSP, their DSPs have fast access on-chip memory with a size up to 1 MB; therefore the DSP/BIOS can process on-chip memory data relatively fast.

On the other hand, the data stored in the external memory can't be accessed quickly due to speed limitations in the external bus hardware. When the DSP processes data in the external memory, it must wait for the data transfer between external to internal memory before handled, which is significantly larger than the DSP computation operation time. Due to the interest in ultrasound signal processing algorithms which involve a large amount of data, the DSPs need to read in external memory data after processing data in its on-chip memory, due to the size limitation of the internal memory on each DSP. To improve the speed of processing data from external memory, the EUSP utilized the built-in DMAs on each DSP, where these hardware data transfer controllers are being used to automatically transfer data in parallel with CPU computation operations, via a double buffering strategy.

1.4.3 Inexpensive 1024-Channel 3D Sonography System on FPGA

The third system is the Inexpensive 1024-Channel 3D Sonography System on FPGA, developed at the Ecole Polytechnique Federale de Lausanne, Switzerland. This work focused on the development of a telesonography-capable medical imaging system suitable for a portable, battery-operated 3D US device that can enable telesonography that has the capability to support up to 1024 channels, which is on par with state of the art devices, in order to allow more access to this type of ultrasound imaging capability. Their

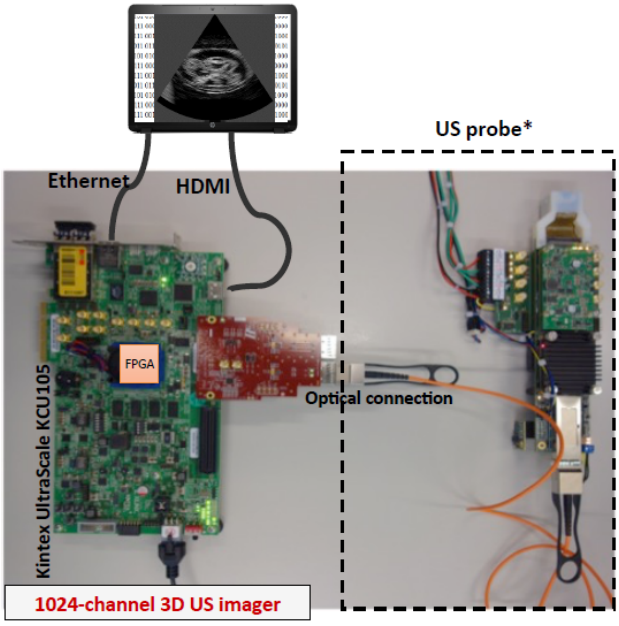

Figure 1.10. Setup of the telesonographic-capable US imager, from ref.[5].

design involved the utilization of a single development Kintex FPGA board with an estimated power consumption of 5 W, to perform beamforming for a real-time 3D ultrasound. Their design allows for two types of data inputs, namely i) real-time via an optical connection, and ii) offline over Ethernet. The reconstructed images generated from this system are then displayed on an HDMI screen[18]. Fig. 1.10 shows the setup of their system.

Inexpensive 1024-Channel 3D Sonography System on FPGA - Hardware. The overall FPGA architecture of their design is based off a Xilinx Microblaze with dedicated on-chip memories and peripherals, with an intra-FPGA communication that occurs over a 32-bit AXI interconnect which connects directly to their proposed beamformer block. Their beamformer code involves a mix of Verilog and VHDL. The Microblaze FPGA is in

charge of supervising the platform operations, while an ethernet controller provides all communications with the external world. For the inputs, data that represents the digitized transducer signals transfer over an Ethernet port, and the Microblaze loads the information into the beamformer's input block RAMs. Once loaded, the Microblaze FPGA reads back demodulated data from the beamformer and pushes the results through the Ethernet port onto an off-system laptop. This laptop is running a custom Visual C application which controls the board, provides it with inputs, and visualizes its outputs.

Inexpensive 1024-Channel 3D Sonography System on FPGA - Software. In this work, the system software mainly focused on the computational optimization of the Xilinx MicroBlaze FPGA, which is achieved through the following methods: nappe-by-nappe processing, delay "steering", static apodization, and demodulation. The nappe-by-nappe process was utilized to deal with the limitations of on-chip memory. Delay "Steering" was utilized to address the evaluation of complex functions for real-time application in an FPGA. Static apodization deals with the limitation of memory requirements and bandwidth for delay and apodization functions. Demodulation is the simplest way to visualize the RF data, i.e., by passing the data through a low pass FIR filter.

This group had to utilize simulated data, i.e., ultrasound data generated in MATLAB via the Field II package. Further, they developed MATLAB scripts to produce a set of files, e.g., constants and initialization data, that can be used to synthesize the FPGA design. These files are loaded into the beamformer either at synthesis time, via include files, or at FPGA boot time, via BRAM initialization in the bitstream.

1.5 Structure of the Thesis

The remaining structure of the thesis is as follows. Chapter 2 describes the system architecture and primarily focuses on the hardware designs of the transmitter, receiver, control subsystems, and interfacing hardware. Chapter 3 describes the embedded software, including real-time image processing algorithms. Chapter 4 describes the experimental results achieved with these systems. Chapter 5 concludes the paper, while Chapter 6 discusses potential future work for this system.

2 System Architecture

This chapter breaks down the hardware of the system in detail. The physical setup of the ultrasound test bench setup shown in Fig. 2.1. The image displays the main subsystems of the prototype and shows the unconnected and fully connected configurations of the system.

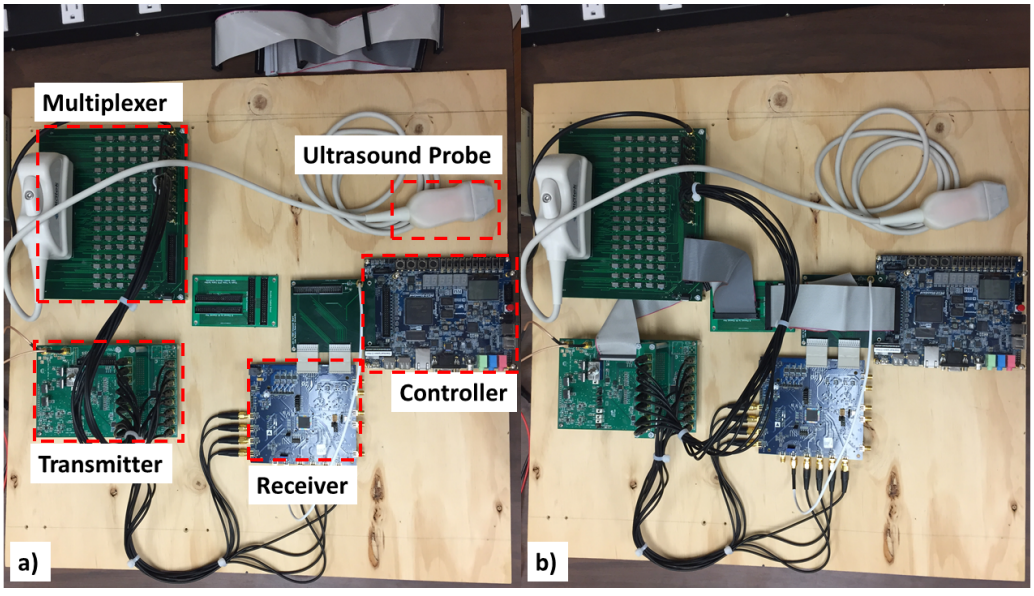

Figure 2.1. a) Prototype setup of the proposed test-bench with subsystem labels, b) Fully connected test-bench.

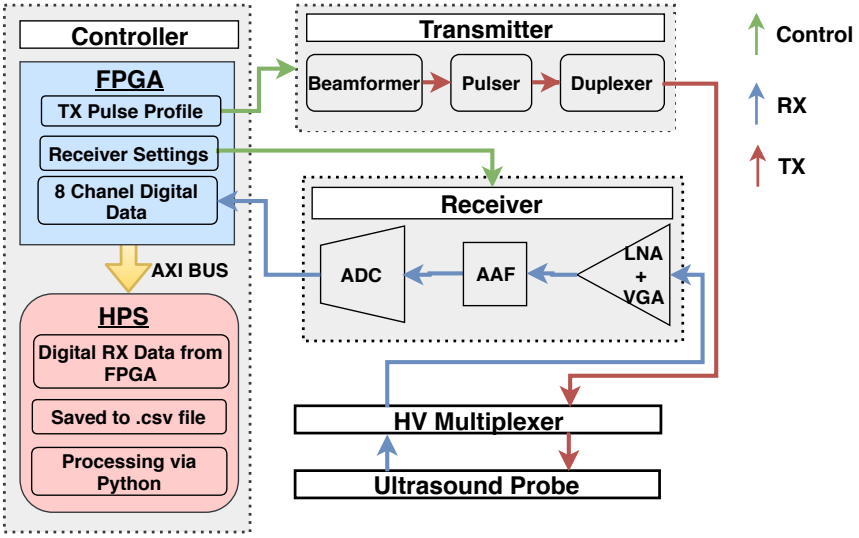

Figure 2.2. Functional block diagram of the proposed test-bench.

2.1 Overall System Architecture and Block Diagram

The system design consists of the following sub-systems: i) an ultrasound probe; ii) an analog front-end (high-voltage [HV] pulser, transmit/receive switch (duplexer), and dynamic beam steering and focusing logic (beamformer)); iii) data acquisition (high-speed ADC and memory); iv) an SoC-based reconfigurable digital back-end for control signal generation and image rendering, and v) interfacing hardware between these systems. Fig. 2.2 shows a block diagram of the design, described in more detail in the following sections.

2.2 Transmitter Design

The main body of the transmitter uses an 8-channel chipset from Texas Instruments (TI), in particular, a beamformer (LM96570), a high-voltage (HV) bipolar pulser (LM96551, up to 50 V, 2 A), and a duplexer or transmit/receive switch (LM96530). The beamformer can be programmed via a standard SPI interface to customize the transmission profile.

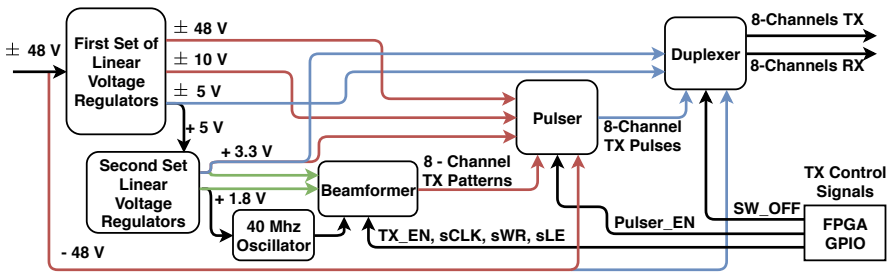

Figure 2.3. Block diagram of the custom 8-channel HV transmitter.

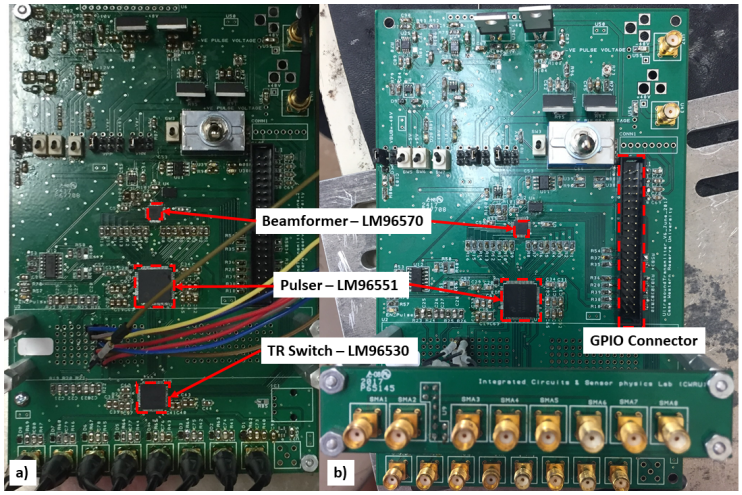

Figure 2.4. Image of custom transmitter subsystem: a) Transmitter without SMA connector board; b) SMA connector board included.

In particular, we can modify the pulse length (up to 64 cycles), center frequency (up to 80 MHz), and time delays between the eight channels for beam-steering (0.78 ns resolution, up to 102.4 μs).

Fig. 2.3 shows a simplified block diagram of the transmitter. The board requires 11 different regulated DC power supply voltages that need to follow a specific start-up and shut-down sequence. Since the goal of this test-bench is the development of wearable/implantable devices, all 11 voltages are generated on-board by a set of low-noise linear regulators powered from a single 48 V bipolar DC input, thus minimizing the overall system footprint. The transmitter consumes ~750 mW during the active mode

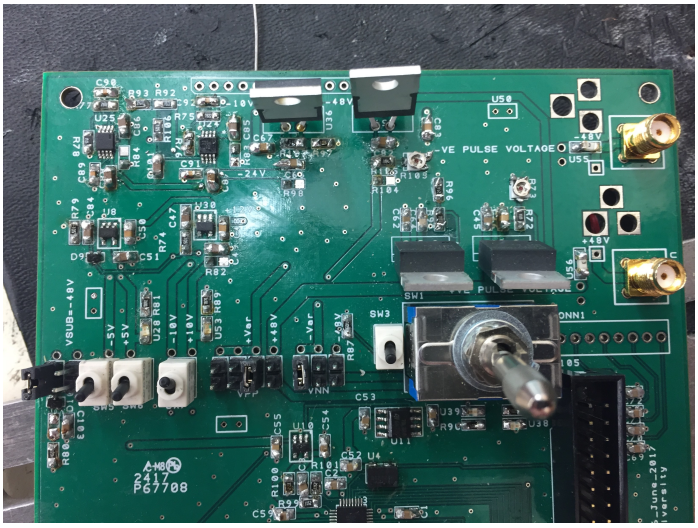

Figure 2.5. Image showing the power circuitry for custom transmitter subsystem.

and ~620 mW in standby mode. Fig. 2.4 shows the custom transmitter subsystem with and without an add-on SMA connector board, including the location of the 8-channel TI chipset. Fig. 2.5 provides a closer view of the power generation circuitry needed to power the 8-channel chipset, shown on the silkscreen of the PCB.

2.3 Receiver Design

The receiver builds on the AD9276 (Analog Devices) chip, which is an 8-channel single-chip ultrasound receiver that contains a low-noise preamplifier (gain: 15.6 to 21.3 dB), variable-gain amplifier (VGA; gain range: -21 to 30 dB), programmable anti-aliasing filter, I/Q demodulator, and 12-bit ADC (10-80 MSPS). The combination of low input-referred noise and over 56 dB of gain control results in a dynamic range of ~114 dB for a typical bandwidth of 1 MHz; this is sufficient for imaging a variety of target geometries. The quiescent power consumption is ~1.6 W with all channels enabled and decreases to ~175 mW in standby mode.

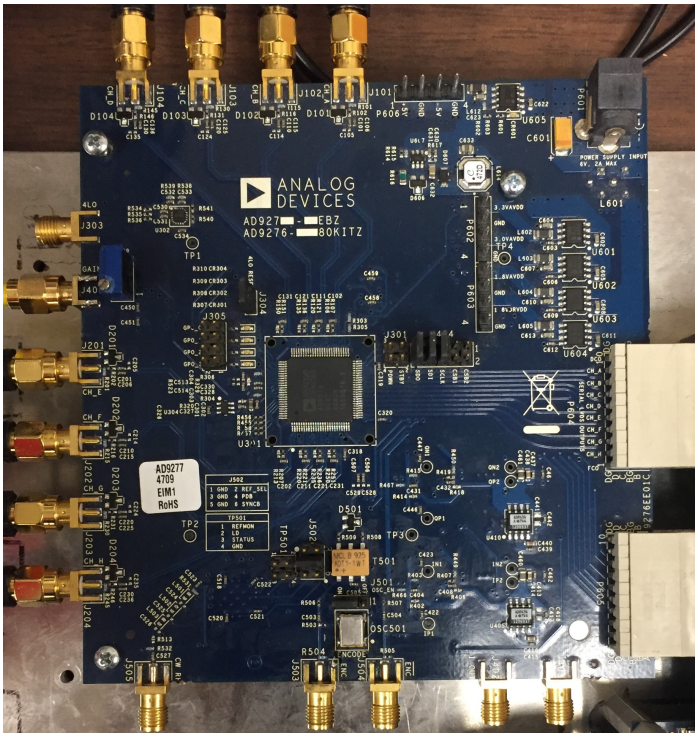

Figure 2.6. Image of the AD9276 evaluation board used in the receiver subsystem.

2.4 High-Voltage (HV) Multiplexer

Both the transmitter and receiver provide parallel channels. Thus, a multiplexer is needed to interface with probes that have > 8 transducers. The chosen design must i) switch bi-directional signals; ii) maintain high linearity and low noise for handling both HV outputs in transmit mode and small echoes in receive mode; iii) have a small footprint and high switching speed; and iv) provide reasonable operating lifetimes.

Given the switching challenges above, Silicon MOSFETs, GaN FETs, and solid-state relays were possible solutions. FET switches are fast and reliable but require bulky isolated gate drivers. Solid-state relays, on the other hand, have built-in separate gate drivers but are slow. In this thesis, the surface-mount reed relays (Coto 9900 series) were

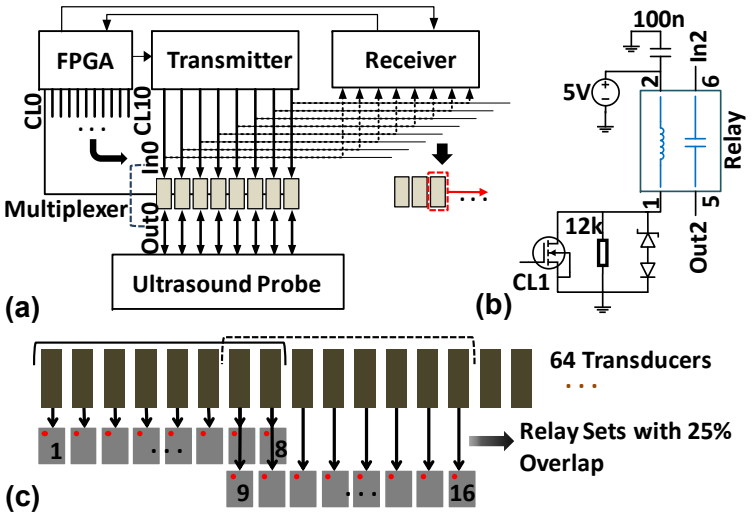

Figure 2.7. Channel selection hardware. (a) Block diagram of the multiplexer; (b) schematic of a single relay and its driver circuit; amd (c) switch selection logic, which is designed to select adjacent sets of 8 transducers with 25% overlap.

a better selection due to their relatively high speed and small size compared to regular relays. Unlike FET switches, these devices have excellent switching properties (very low $R_{on} \times C_{off}$ figure of merit) and don't need bulky isolated gate drivers. Moreover, they have rated for 10^9 switching cycles, which is lower than semiconductor switches but sufficient for this application.

The custom HV multiplexer board includes reed relays, driver circuits, power regulators, control signals and an ultrasound probe connector. Fig. 2.7(a) shows a block diagram of the multiplexer channel selection logic. Fig. 2.7(b) shows the schematic of a single relay and its driver circuit. The latter is based on a power MOSFET and a set of protection diodes. A total of 11 control signals (CL0-CL10) trigger the driver MOSFETs in each set of 8 relays, with 25% overlap between the events. This allows us to generate images using all the transducers of a 64-element linear array (see Fig. 2.7(c)).

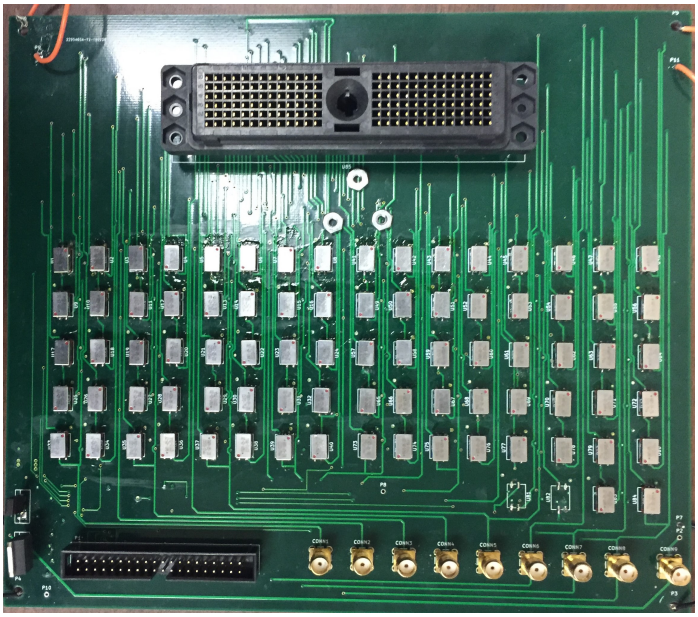

Figure 2.8. Image of custom multiplexer subsystem.

2.5 System Controller

The system controller provides control signals for all the other subsystems. We use a low-cost SoC development board (Altera DE10) for this purpose, shown in Fig. 2.9. The Terasic DE10-SoC development board has shares many similarities to the earlier Terasic DE1-SoC development board, as mentioned below.

The Terasic DE10-SoC is a System-on-Chip (SoC) which combines a dual-core Cortex-A9 hard processor system (HPS) with a Cyclone-V FPGA. The communication between the hard processor system and the FPGA fabric is due to the use of a high-bandwidth interconnect backbone, implemented using an AXI bus or Avalon-to-Memory-Map interface. This backbone is configured using Altera Quartus, an FPGA compiler, synthesizer, and programmer specifically used for Altera's FPGA development kits. The interconnect backbone allows for communication between three different directions, namely

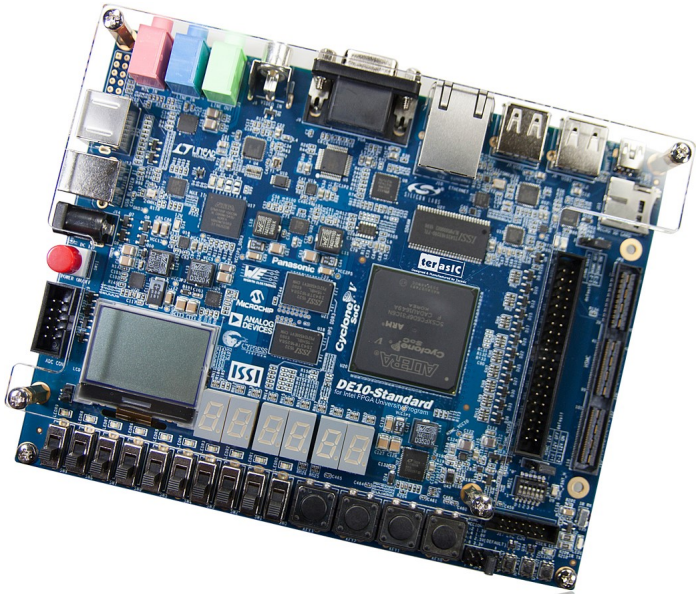

Figure 2.9. DE-10 standard development kit,

from the HPS-to-FPGA, the FPGA-to-HPS, and lightweight HPS-to-FPGA. The DE10-SoC board has 1 GB of DDR3 RAM and therefore can run an embedded Linux operating system (O/S) with standard input/output (I/O) such as a mouse, keyboard, and VGA display.

While the Terasic DE10-SoC shares the majority of its architecture with the DE1-SoC, there are several differences. These include the number of FPGA logic elements (110K for the DE10-SoC), on-board EPCS128 flash memory, and the availability of both a GPIO 40-pin connector and a HSMC (High-Speed Mezzanine Connector) on the DE10. By contrast, the DE1-SoC has 85K FPGA logic elements, no flash memory, and two GPIO 40-pin connectors.

The Cylcone-V FPGA within the DE10-SoC is programmed to i) configure transmitter parameters (delay profile, pulse pattern, and width, etc.), ii) trigger the transmitter to generate HV pulses, iii) configure the gain settings of the receiver, iv) trigger the receiver to acquire data, and v) select the specific set of ultrasound probe channels for imaging.

The SoC is also used to store the received echo data from the receiver subsystem for post-processing. For this purpose, the FPGA i) collects blocks of digital data from the receiver, and ii) transmits them to the SoC's hard processor (a dual-core ARM Cortex-A9 running Linux) via the on-chip AXI bus. The transferred data is saved as a text file and used for on-board post-processing using Python, thus allowing the development of autonomous imaging applications on the embedded Linux system.

2.6 Interfacing Hardware

The previous subsystems mentioned above were developed as separated system modules, to allow easier testing/repair of the subsystems and provide greater flexibility for future application-specific improvements. As a result, this modular design requires interfacing hardware to provide secure and robust connections between the main Altera DE10-SoC control board and the rest of the subsystems.

2.6.1 SMA Cables

The main transmission and reception paths of the ultrasound test-bench are connected by male-to-male SMA coaxial cables. These have a characteristic impedance of 50 Ω and can be used from DC to 18 GHz, which is more than adequate for ultrasound signals. Also, since our receiver board uses SMA connectors for the analog signal inputs, it is easier to utilize SMA connectors throughout the system design to match cable impedances between the transmission and reception paths. We used 2 feet long cables to provide enough slack for the positioning of subsystems on the mounting plate and bundled them together with zip-ties; these are the black cables shown in Fig. 2.1.

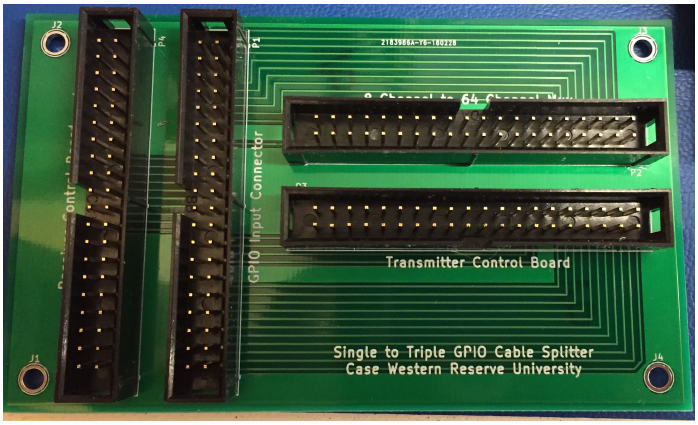

Figure 2.10. GPIO breakout board.

2.6.2 GPIO Breakout Board

The majority of the control signals from the Altera DE10-SoC, which include the signals for SPI programming (Chip Select (CS), SCLK, SDI, and SDO), channel switching, and transmission enable, are classified as low-frequency control signals since they usually fall below a frequency of 1 MHz. As a result, these signals transfer over a normal GPIO ribbon cable. A GPIO breakout board (see Fig. 2.10) was developed to split the pins accordingly and connect the proper control pins to their respective systems (transmitter, receiver, and HV multiplexer). This board was designed to connect all the boards via keyed ribbon cables to prevent incorrect physical connections between the subsystems.

2.6.3 Receiver to DE-10: ADC to HSMC Interposer Board

A major obstacle to overcome was connecting the AD9276 receiver board to the Altera DE10 board, to bring the converted digital echo signals into the control subsystem. The main issue was that even though the AD9276 has an ADC sampling rate between 10-80 MSPS, the ADC automatically multiplies the sample rate clock for the appropriate LVDS (low voltage differential signaling) serial data rate. Since the ADC is set up for DDR (Dual

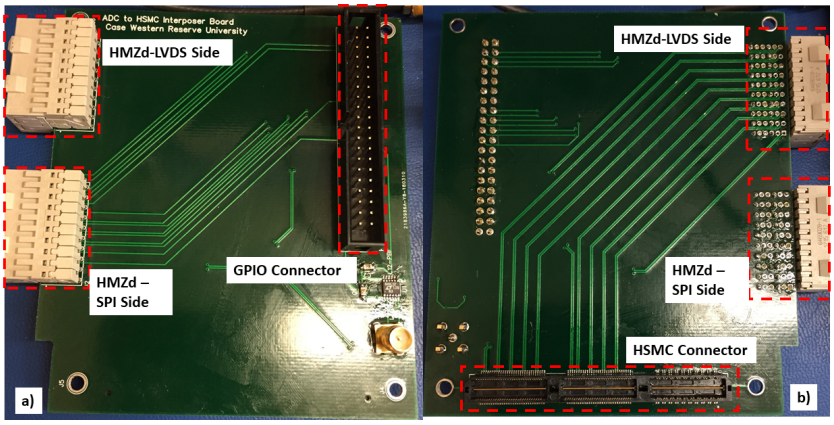

Figure 2.11. ADC-to-HSMC interposer board: a) top side; b) bottom side

Data Rate) mode, this serial data rate was usually six times the ADC sampling rate, which meant that the actual speed of the data that would be going into the DE10 would range from 60-480 MSPS. As a result, the interface between the AD9276 and the DE10 had to account for these high-frequency serial data lines.

Since the DE10 Standard has an HSMC connector on the board, the FPGA has designated pins to directly take in these LVDS signals. Thus, the main issue was designing the board to correctly transport the LVDS data to this connector. Based on the LVDS standard, the routing on the PCB needs a differential impedance of 100 Ω, which was achieved via edge-coupled differential micro-strip transmission lines. The resulting trace widths and spacing necessary to ensure this differential impedance were 28 mils and 5 mils respectively.

Fig. 2.11 shows the top and bottom sides of this custom PCB, where the LVDS signals are on the bottom side of the board, and the general SPI programming lines are on the top of the board.

2.6.4 Ultrasound Transducer Probe

As shown in Fig. 2.1, the ultrasound transducer probe used for the overall system setup was a 64-element linear array from Blatek Inc. The performance parameters of the transducer probe include a center frequency of 8 MHz, a frequency bandwidth from 5-12 MHz, a pitch of 0.3 mm between the linear elements, and an elevation of 4.0 mm.

3 System Software

This chapter breaks down the software approach and the necessary tools for the ultrasound test bench setup. A block diagram showing the overall configuration of the system is shown in Fig. 2.2. The system code is broken down into three subsystems, which involve Verilog for the FPGA firmware, C code for the embedded control program, and the image processing code which is either based in MATLAB for off-system processing or Python for on-system processing. The following sections provide details of each software subsystem. Fig 3.1 summarizes the structure of the necessary software to operate this system.

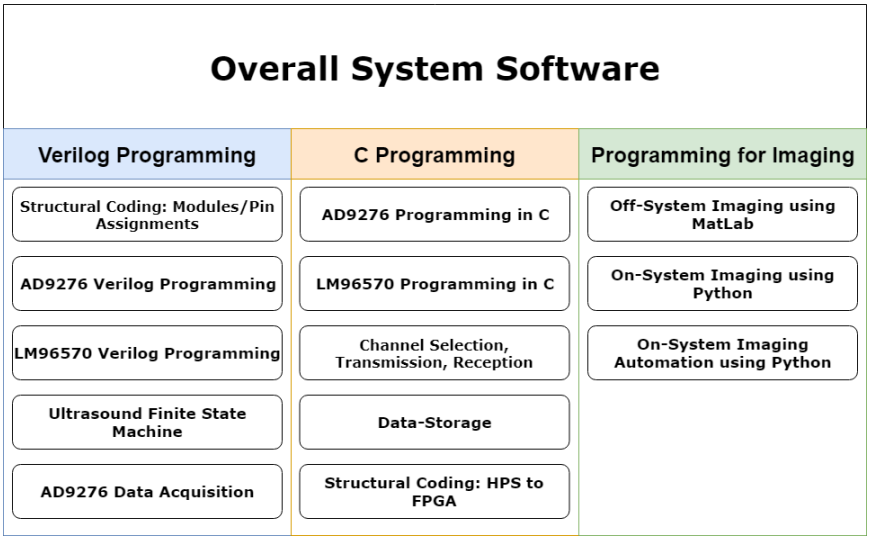

Figure 3.1. Visualization of the software architecture of the prototype.

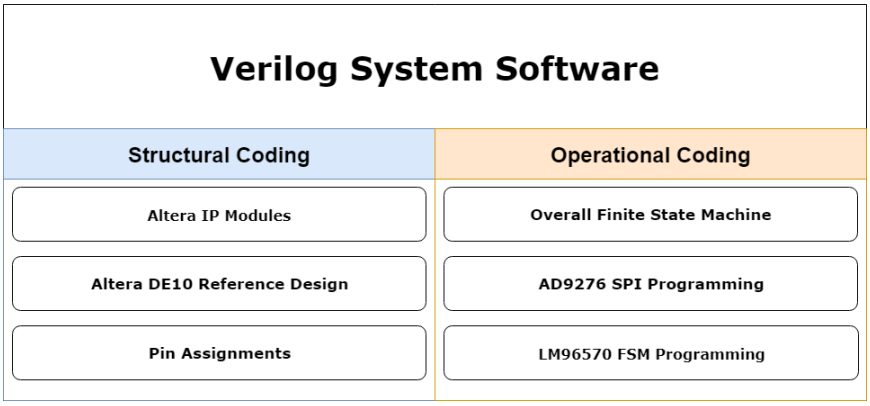

Figure 3.2. Visualization of Verilog software used for the prototype.

3.1 Firmware Code - Verilog

The firmware of the system consists of Verilog code loaded into the FPGA during startup of the embedded Linux Environment on the DE10-SoC. This code is used to set up i) the assignment of pins for the control and data acquisition, and ii) the timing of signal events during operation of the ultrasound system through a finite state machine (FSM). The main parts of the code involve the structural code (DE10-Standard and Qsys), the overall FSM, the AD9276 acquisition module, the AD9276 ADC capture block, and the LM96570 serial programming interface (SPI). Fig 3.2 summarizes the structure of the necessary software to operate this system.

3.1.1 DE10-Standard GHRD Modification via Qsys

The structure of this code relies on the DE10-Standard Golden Hardware Reference Design (GHRD), which is a reference design provided as part of the Altera DE10-Standard demo disc. The purpose of this design template is to establish the connections between the hard processor system (HPS) of the development board to the peripherals that are connected to the FPGA fabric. Any information on the peripherals of the system can

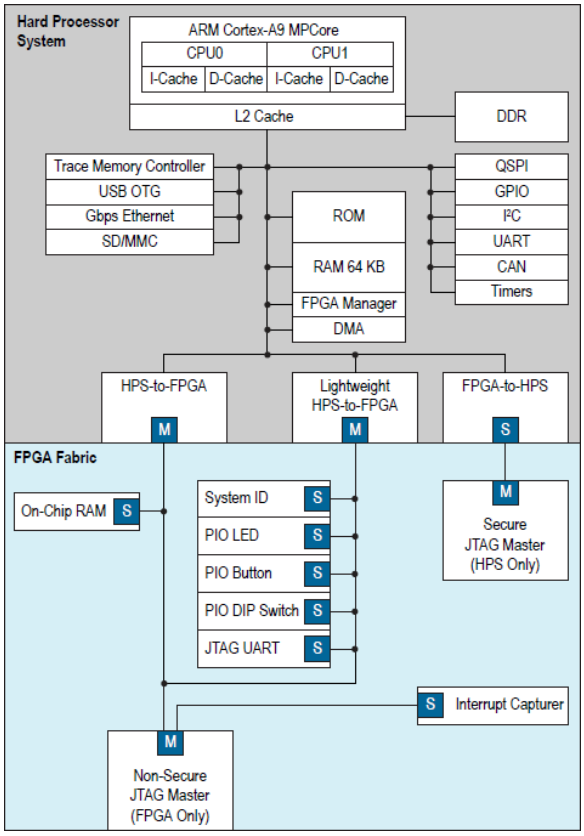

Figure 3.3. Block diagram of GHRD, from ref.[6].

be controlled or received by the hard processor system of the DE10-Standard via a reference address. All active peripheral connections within the system port to an address map observed from the HPS. Fig. 3.3 provides a visual representation of the architecture between the FPGA fabric and the HPS.

To modify the types of peripherals and additional intellectual property (IP) functions/subsystems to the central DE10-Standard system, the GHRD will need to be edited by "Qsys System and Platform Integration Designer," which is a built-in tool within the Altera Quartus Prime software. This tool is meant to improve design time by automatically generating the interconnect logic and connecting multiple built-in IP functions and subsystems into a visual interface for the design.

Most of the subsystems from the GHRD were adjusted and re-utilized in the ultrasound test-bench in order to ensure proper timing during system operation. A few examples of these subsystems include the system source clock, clock bridges, and memory pipeline bridges. On the other hand, a few additional modules added to the GHRD system design for our ultrasound test-bench include the Avalon-ST single clock FIFO, Avalon FIFO memory, SPI core, and parallel I/O modules.

The Avalon-ST single clock FIFO module is a FIFO buffer which operates from one common clock for both the input (sink) and output (source) ports. The single FIFO block can have customized parameters like symbols per beat (i.e., clock period), bits per symbol, depth, channel width, and error width based on the system application. For this design, the FIFO buffers were set to have one symbol per beat at 16 bits per symbol with a total FIFO depth of 16 bits, a channel width of zero, and an error width of zero. The way this module works is by accepting the incoming data on the in the interface (Avalon-ST data sink) and forwards it to the out interface (Avalon-ST data source). The core asserts a valid signal on the Avalon-ST source interface to indicate that there is data available at the interface. Fig 3.4 provides the fundamental representation of this FIFO core.

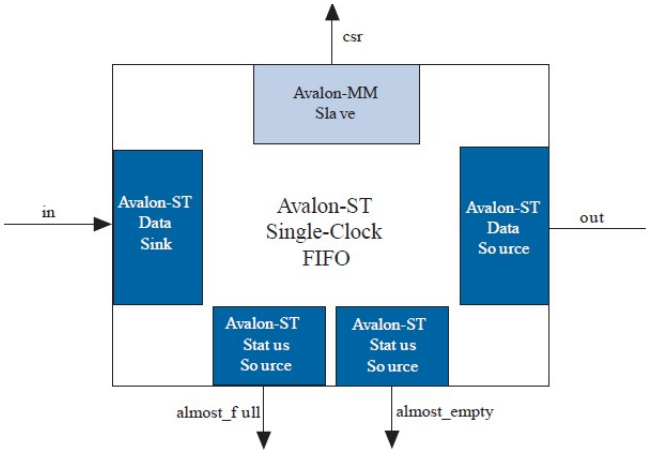

Figure 3.4. Diagram of ST single clock FIFO, from ref.[7].

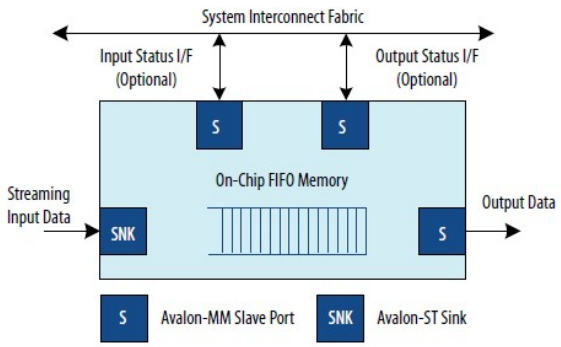

Figure 3.5. Diagram of the FIFO memory module (MM) core, from ref.[7].

The Avalon FIFO memory module utilized for this system has the configuration of the Avalon-ST sink to Avalon-MM read slave. This sets the input as an Avalon-ST sink and the output as an Avalon-MM read slave with a width of 32 bits, which is the minimum acceptable bit width for this memory module. Due to this configuration, the Avalon-ST input (sink) data width must also be set 32 bits. Similar to the Avalon-ST single clock FIFO, there are additional parameters configured, including bits per symbol, symbols per beat, and the width of the channel and error signals. For this application, the parameter choices were 16 bits per symbol, two symbols per beat, a channel width of zero, and an error width of zero. The symbols per beat have to be set to 2 to achieve the minimum 32-bit width input on this module. Fig 3.5 provides the fundamental representation of this FIFO memory core.

Utilizing the Avalon FIFO memory modules mentioned above, an Avalon-MM master can then be used to read data from the FIFO memory. During this process, the signals are mapped into bits in the Avalon's address space, where the output interface uses a wait request message to determine whether there is data to be read from the FIFO. For example, the wait request is asserted for reading operations when there is no data to

be read from the FIFO, but it will be de-asserted when the FIFO has data to send to the address space.

The Avalon-ST single clock FIFO and Avalon FIFO memory are connected via the Qsys toolkit, which allows the LVDS data from the AD9276 receiver subsystem to be entered into the DE10 FPGA as 16-bit data streams and then saved directly to the embedded memory of the HPS of the DE10-Standard system.

The AD9276 receiver subsystem will need some coding to allow the user to configure it. Fortunately, the AD9276 is programmed to the proper configuration over SPI using simple 24-bit (3-byte) codes. SPI is an industry-standard serial protocol that is commonly used in embedded systems to connect microprocessors to a variety of devices. Fortunately, the Avalon SPI core can be utilized to set up this 24-bit SPI for the system. Fig 3.6 provides the fundamental representation of this SPI core. The Avalon SPI core implements the SPI protocol while providing an Avalon memory-mapped (Avalon-MM) interface on the back end. This SPI core implementation is either in a master or slave protocol. The width of the receive and transmit registers can be configurable between 1 and 32 bits. Also, the core provides an interrupt output that can flag an interrupt whenever a transfer completes.

The rest of the system utilizes parallel input/output cores. These cores provide a memory-mapped interface between an Avalon-MM slave port and general purpose I/O (GPIO) ports. The GPIO ports are usually either connected to on-chip user logic or the I/O pins of devices externally connected to the FPGA. This setup allows for easy I/O access to user logic or external devices in situations where a bit-banging approach is sufficient. For the ultrasound system, the uses for the PIO cores include the following: acquiring data bits from the receiver channels, control pins for channel selection, serial

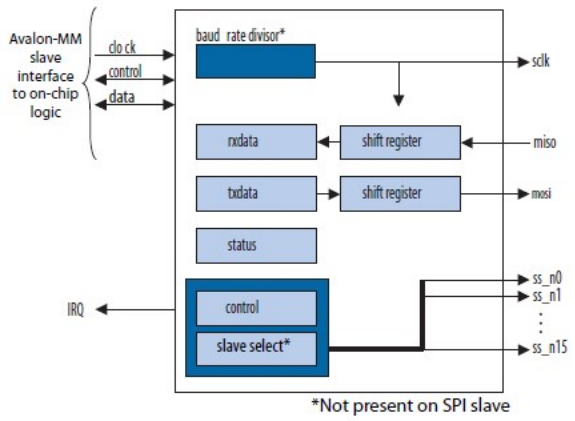

Figure 3.6. Diagram of the SPI core, from ref.[7].

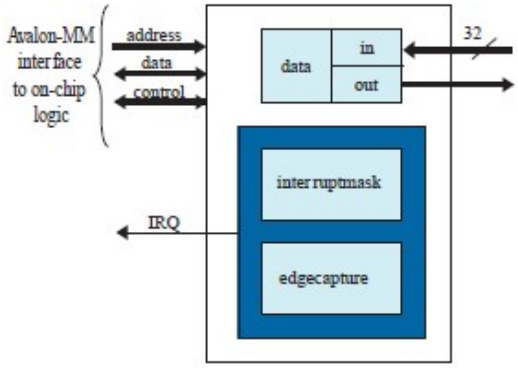

Figure 3.7. Diagram of the PIO core, from ref.[7].

programming of LM96570, pulse firing, and the transfer of data from the FPGA registers to the embedded memory of the HPS. Fig 3.7 provides the fundamental representation of this PIO Core.

3.1.2 Finite State Machine (FSM)

The main Verilog operation module is the ultrasound FSM. This FSM was generated in order to establish the proper timing to allow for timing windows and necessary control signals (e.g., start data acquisition after transmission of the ultrasound pulses, prevent

another transmission from occurring during data acquisition, and resetting the machine during the next acquisition). Fig 3.8 shows the flowchart for this FSM.

Input/Output Signals and Parameters. The input/output signals used in the FSM are the following: START - input signal that notifies the FSM to begin, DONE - output signal that notifies the rest of the system of completed data acquisition, TX EN - output signal that is sent to the LM96570 chip to fire the transmission pulse, ADC START LENGTH - input that is held high for a certain period in order to ensure the enabling of the ADC, ADC INIT DELAY - input that informs of the FSM for the period of time to wait before enabling the ADC, ADC SAMPLES PER ECHO - input that determines the length of the the ADC data acquisition, FIFO EN - output signal that goes to the on-board FIFOs to inform them that the current data is valid and should be stored in the FIFO, CLK - input

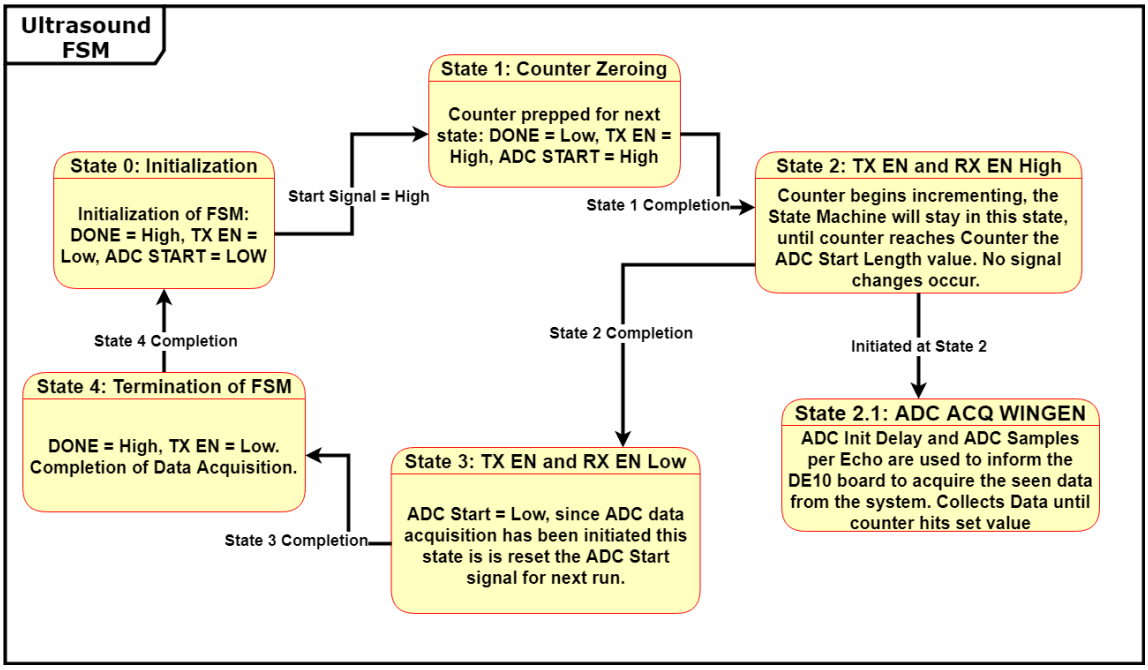

Figure 3.8. Block diagram of the FSM used in the ultrasound prototype.

clock from the system in order to allow proper timing of the system, and RESET - input that is used to hard-reset the system.

Breakdown of the FSM States. The following section breaks down the states and sub-states that occur within the FSM that controls operation of the entire system.

State 0: Initialization: This state is used to initialize an FSM system for the next set of data acquisition. At this state, the DONE signal is high, while the TX EN and ADC START signals are held to low, and are continuously held in this state until the START signal moves the FSM into State 1.

State 1: Counter Zeroing: This state is used to set the DONE signal to low, the ADC START signal and the TX EN signal shifts to high, and a counter is initialized to count to the ADC START LENGTH value, before moving to the next state. At this point, the system has sent out the transmission pulse signal to the system and is now starting the ADC data acquisition.

State 2: TX EN and RX EN High: During the state, the counter is iterating up to the ADC START LENGTH value, before moving to the next state. No signal changes occur during this state.

State 2.1: ADC ACQ WINGEN: State 2 of the Ultrasound FSM, also includes the running of another FSM named the "ADC ACQ WINGEN." The point of this FSM is to take the ADC INIT DELAY and the ADC SAMPLES PER ECHO, to generate the proper FIFO EN signal to inform the DE10 Standard on-board FIFOs to start acquiring the valid data. This stage utilizes the ADC INIT DELAY and the ADC SAMPLES PER ECHO values within counters to move from one state to another in this subsystem.

State 3: TX EN and RX EN Low: During this state, the ADC Start signal is driven low. At this point, the ADC data acquisition has already started, and this signal resets to zero for the next iteration.

State 4: Termination of FSM: This state is used to set the DONE signal to high, and the TX EN signal to low. Data acquisition is complete on the system, and the data can now transfer from the embedded memory FIFOs to the Linux system. After this point, the FSM will go back to State 0 to wait for the next data acquisition period.

3.1.3 ADC acquisition block

This module, named "ADC AD9276" by the Verilog firmware, establishes the necessary pins and Altera IP modules in order to convert the LVDS signals from the AD9276 chipset into 16 bit data that can fit into the DE10 Standard HPS system via the FIFO-to-FIFO memory block generated by the Qsys System Builder for our system.

This code module takes in a total of 20 differential pins (P/N) from the HSMC connector on the DE10-Standard, which include two differential clock signals (Data Clock Output - DCO and Frame Clock Output - FCO) and eight differential channel data signals (Channel A to Channel H). This code takes the differential pin signals, pushes them through differential input buffers to make single-ended signal streams. The signal streams from the channel data are then driven through dual-data rate input buffers (based on the timing of the FCO and DCO clocks) that allow us to extract DDR data from these signals.
Altera Differential Input Buffer. The ALTINBUFDIFF primitive allows us to name and connect positive and negative pins to a differential I/O standard. This primitive allows us to do the following: make a location assignment, make an I/O standard assignment, enable bus-hold circuitry, and enable a weak pull-up resistor[19]. Since the AD9276

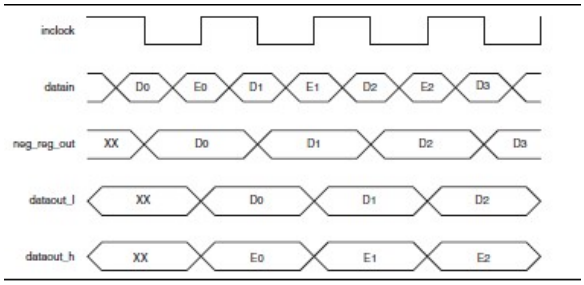

Figure 3.9. Basic timing diagram of Altrera DDR I/O, from ref.[8].

chipset is outputting LVDS signals into the DE10 Board, the ALTINBUFDIFF will convert them into a single digital signal for the system.

Altera DDR Input/ALTDDIOIN Implementation. Handling the AD9276 DDR data, the ALTDDION core has to be implemented to receive data on both edges of a reference clock. The parameters adjusted for this core are the following: width (bits), implementation of asynchronous clear and asynchronous set ports, implementation of synchronous clear and synchronous set ports, the utilization of an inclocken port, and falling/rising edge preference.

ALTDDIO IN Timing. Fig 3.9 provides a visualization of this functionality of this ALTDDIOIN mega-function. The signals utilized/generated by this core are the port names used in the ALTDDIO-IN IP core, which are datain, dataout-l, dataout-h. The datain signal is the input data from the pin into the DDR circuitry. The outputs from the circuitry are dataout-l, and dataout-h. Both dataout-h and dataout-l feed the logic array and show the conversion of the data from a DDR implementation to positive-edge triggered data.

Integrating AD9276 with System DDR IO. With the single clock signals we have above (DCO and FCO), and the resulting received DDR data saved into separate registers (q1

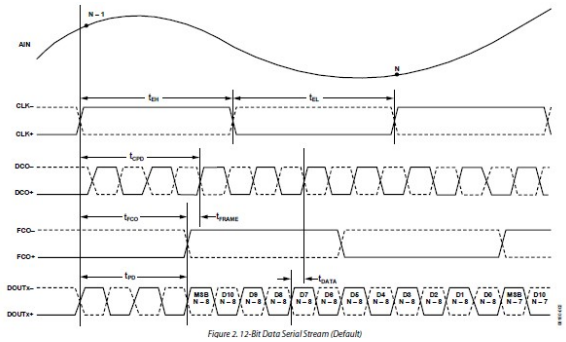

Figure 3.10. Streaming data outputs of the AD9276, from ref.[9].

channel,q2 channel) the next major step is to organize these bits into the correct combination to get the right data.

To start the system needs to look at the FCO and DCO clocks generated from the AD9276 differential signals and their relation to each other, based on the AD9276 datasheet the DCO clock will always be at six times the frequency of the FCO clock. The reason for this is due to it being a 12-bit ADC set for DDR mode; as a result, all 12 bits of data exported from one frame of the FCO clock, in six clock cycles of the DCO. Based on the AD9276 datasheet, these clock signals depend on the rising edges of the clocks. Fig 3.10 shows this expected output from the AD9276 when it is in operation.

The first issue was that the Altera ALTDDIOIN mega-function starts detection with the falling-edge first and then rising-edge, while the AD9276 chipset starts with rising-edge and then falling-edge with its DDR. The output data from the rising edge (dataout-h) had to be delayed by one DCO clock cycle to capture the signals, which were late by one period. With this issue resolved, the q1 (dataout-l) and q2 dly (dataout-h) bits for each channel load into the two separate 6-bit registers for each channel, where one register has all the q1 values, and the other contains all the q2 delay values. At every positive edge of the DCO clock, the data in these registers updates.

The next step, which is exporting the register data, is related to a specific frame of the FCO clock. To ensure that the data isn't shifting from the DCO clock and the FCO clock, a signal called fcostb was generated to detect a change from FCO from low to high or high to low at every positive DCO edge. When this change occurs, the data stored in the two 6 bit registers is reorganized into one 16 bit register, which exports into the FIFOs of the DE10.

3.1.4 Programming of LM96570

For the LM96570 TI beamformer chip, pulse patterns and delay settings are transferred through a serial interface (LSB first), which is intended to simplify the timing requirements on the driving circuitry. Programming occurs based on the internal memory registers of the device. Unfortunately, the serial data bit length of these registers varies significantly based on the address of this chip (5-bit address, 1 R/W bit and 4 to 64-bit data: total 10-bit to 70-bit sequence). To handle this programming sequence, another FSM needed to be developed to deal with the variance between the data length and send/receive the proper number of serial bits when reading/writing to a specific address.

Input/Output Signals and Parameters. The input/output signals used in the LM96570 FSM are the following: START - input signal that notifies the state machine to begin, DONE - output signal that notifies the rest of the system that FSM finishes, RD DATA - data wires that are used to hold the data that will be programmed to the LM96570, NUM OF BIT - length of data that is necessary for register, DATA IN - data wires that are used to hold data that comes from the LM96570, SPI CS N - Chip Select Low Enable, needed to but the chip into programming mode, SPI SCK - programming clock for the chip, SPI SDI - serial data (one bit) coming from the LM96570, SPI SDO - serial data (one bit) being

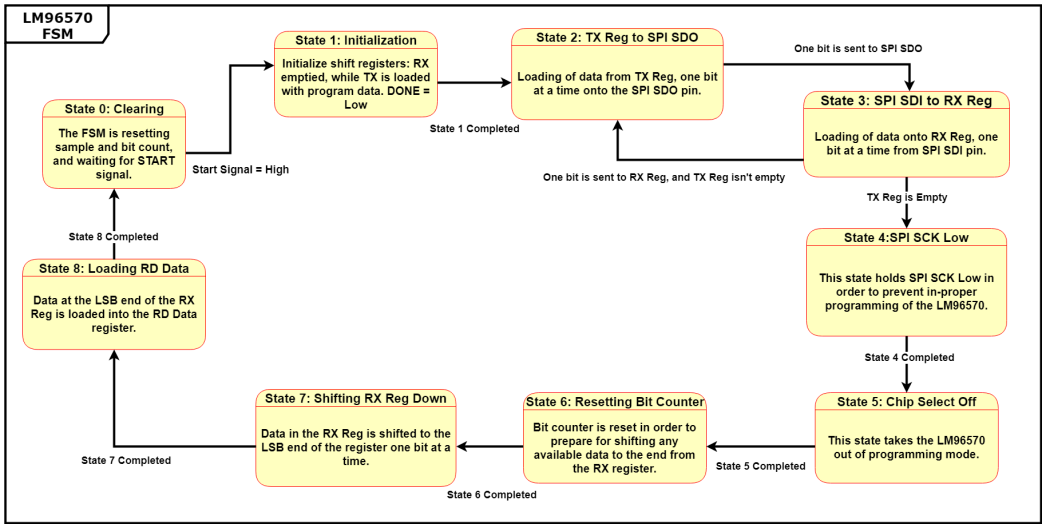

Figure 3.11. FSM needed to configure the LM96570.

written to the chip, CLK - input clock from the system in order to allow proper timing of the system, and RESET - input that is used to hard reset the system.

FSM State Breakdown - LM96570. The following section breaks down the states and sub-states that occur within the finite state machine for register operation of the entire system. Fig. 3.11 provides a visualization of the states that this FSM goes through during operation.

State 0: Clearing During this state, the FSM resets the sample count and bit count, and then waits for the inputted START signal before it moves to State 1.

State 1: Initialization This state initializes the shift registers based on the programming information. Specifically, it empties the RX register while programming the TX register. It then drives the DONE signal low, asserts the Chip Select signal, and resets the bit count before moving to State 2.

State 2: TX Reg to SPI SDO This state takes the first bit from the TX shift register and loads it into the SPI SDO signal location, which removes it from the TX register and shifts the remaining bits down. At this point, the SPI SCK is held low, so the bit hasn't been sent

the LM96570 chip. On the other hand, the bit counter is incremented by 1 to count this bit shifting event. After this sequence completes, the FSM will move onto state 3.

State 3: SPI SDI to RX Reg During this state, any information that is on the SPI SDI location will be loaded into the RX shift register at the MSB and shift any previous data down one bit. At this moment, the SPI SCK signal will go high, which means that the bit loaded onto the SPI SDO location, will be pushed to the LM96570. After these events, the bit counter is checked to determine whether the last bit has unloaded from the TX register. If more bits are available, the FSM will go back to State 2 and repeat this process until this there are no more bits available. Once met, the FSM will move onto state 4.

State 4: Setting SPI SCK Low During this state, the FSM fixes the SPI SCK to low, thereby preventing any information on the SPI SDO pin from being sent to the LM96570 and thus creating errors in the programming configuration. Upon completion, the FSM moves to state 5.

State 5: Chip Select Off This state, de-asserts the Chip Select signal, which officially notifies the LM96570 to go out of programming mode. All of the programmed configurations are now stored on the LM96570's registers. This event will then move the FSM to state 6.

State 6: Resetting Bit Counter During this state the bit counter is being reset, instead of the counter set to the bit count width and number of bits. This bit counter will be set based on the data width as well, because its purpose is to keep track of the necessary bit shifting operations needed to push the MSB-loaded data in the RX register to the LSB end of the RX register, which occurs in state 7.

State 7: Shifting RX Reg down This state shifts the location of the data in the RX shift register, all the way to the least significant bit of the register, while keeping track of its

position based on the bit counter from State 6. Upon completion, the FSM will move to state 8.

State 8: Loading RD Data During this state, the FSM will then take the RX shift register and copy the data to the RD DATA register, which is connected to the HPS of the DE10. This will allow us to access the data from Linux. On completed transfer, the FSM will return to State 0 and wait for the next initialization signal.

3.2 C-Based Control Software

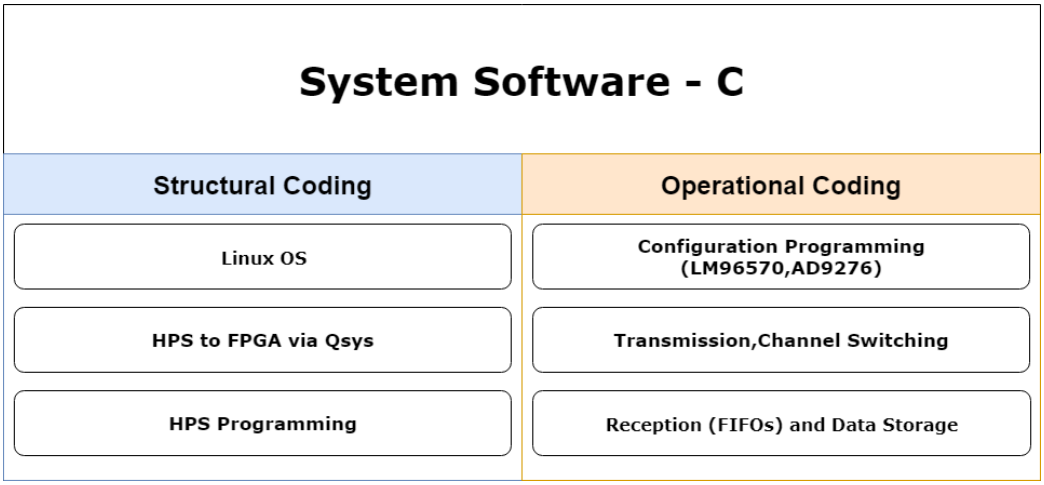

Figure 3.12. Visualization of C-based software.

The operation of the ultrasound test bench is controlled by a C program named the DE10-Standard US C program. This program operates on the ARM processor under a Linux environment on the Altera DE10-SoC board. It serves as a master controller for all the hardware connected to the FPGA fabric side, which includes the transmitter board, receiver board, multiplexer board, and data acquisition FIFOs within this test bench. Fig 3.12 provides a visualization of the specific parts of the C code developed for this system.

The DE10-Standard US C program provides initialization parameters to the transmitter and receiver boards via serial programming. After initialization of the operation parameters, the program activates a set of 8 channels on the custom multiplexer board. Once the channel set is selected, the program will pass start signals to the transmitter board. These signals will i) enable the high-power pulser, and ii) start the sequence for firing transmission pulses out to the eight selected channels of the ultrasound probe.

The wave-tissue interactions that occur within the sample will generate echoes. The initial ultrasound pulse and resulting echoes will be sent back via the receiver path of the system. Along this path, the receiver board (which is constantly streaming data) digitizes the analog input signals are sends them continuously into the Altera DE10-SoC for processing and storage. During this time, the DE10 Standard US program sends a start data acquisition signal to the embedded memory FIFOs on the system; this signal informs the FIFOs to start storing the data seen at the FIFO until the FIFOs are full. Once the FIFOs are full, the program will load the data from the FIFO into a data bank variable, thus allowing it to i) save data from the chosen sub-window until the end of the program, and ii) clear the FIFOs for the next sub-window. The transmission, reception, and data acquisition steps are repeated until we reach the final sub-window. After data has been acquired from this sub-window, the databank variable, which now contains the digital ultrasound data, will be converted into a .txt file for data analysis and image processing at a later time either on- or off-system, or sent directly to an external on-system program for real-time image processing/generation.

3.2.1 HPS and FPGA Connections - Setup

Since most of the connections between the HPS and FGPA have been set up through Verilog coding, the next step in developing an embedded system is to implement C code

to utilize the DE10 HPS. For that to occur, first, a Linux operation system needs to be installed on the DE10.

The benefits of using an embedded application over a Linux-based OS include i) the availability of processors upon booting, and ii) initialization of most of the HPS peripherals for utilization by the programmer/user, who has access to many device drivers. Also, another significant benefit is that the Linux kernel has no restrictions on running compiled C programs. As long as the required run-time environment is installed, the Linux kernel will be able to run code in another programming language. Of course, this depends on the availability of suitable run-time environments for the processor.

On the other hand, the main drawback in running an embedded application on top of a Linux OS is that there will not be direct access to the peripherals through the physical memory-mapped addresses since the OS puts a virtual memory system in place. As a result, the physical addresses of the peripherals of interest will have to be mapped to the virtual address space of the running programs to access the peripheral registers. The user needs to map these physical addresses into the virtual address space of the running program in order to access the peripheral registers.

Steps to Setup Embedded Linux OS. The following subsection describes the items/-software/tools needed to set up the Linux environment for the DE10 board successfully. Due to the DE10-SoC having the same Cyclone V SoC as the DE1-SoC Board, many of the steps from the SoC-FPGA Design Guide [DE1-SoC] from Ecole Polytechnique Federale de Lausanne[20] were used to set up the Linux OS on the DE10.

The DE10-SoC needs to boot off of a microSD card, which needs to be partitioned before one can write to it. Proper setup requires the SD card to be plugged into another

computer in order to wipe the partitions on the card and re-partition the card for the system.

The card needs to have the following file systems: i) a FAT32 partition for various boot-time files (FPGA raw binary file, Linux kernel zImage file, U-Boot configuration script), and ii) an EXT3 partition for the Linux root filesystem. For this project, these files were created by following the instructions in the SoC-FPGA Design Guide[20].

Once these partitions are available, they can be loaded with the required files above to set up the system. After the boot-time files and Linux root filesystem are on the SD card, the HPS needs to be set up accordingly to programmatically access the peripherals that are part of the FPGA fabric via a generated header file. Executing the sopccreate-headerfiles command from the SoC EDS command shell, included from Altera, as part of their embedded development suite will make this header file. Successful completion will create an "hps_soc_system.h" header file that contains the macro information for the devices that are implemented in the system and connected to its master ports.

HPS Programming Theory. Once the header files of the system are generated to allow connections to the device peripherals, we can then use them to program the HPS via C code. The software that is used to program the DE10-SoC for the ultrasound testbench was Eclipse for DS-5 v5.28.1 community edition.

The HPS has the same functionality as most "microcontrollers," which means that access to the peripherals occurs through reading and writing at the device's address. Therefore, a peripheral connected via a system bus has a unique address that can be found by adding the device's offset to the bus's own address.

With the established addresses of the peripherals, the next step is to read from and write to these peripheral devices. Fortunately, there are a few utility functions that can

handle instructions of multiple sizes (byte - 8 bits, hword - 16 bits, word - 32 bits, dword - 64 bits) for reading/writing of devices on the system. These functions come from the "socal.h" header file, which is provided along with Eclipse. For C programming of the ultrasound testbench, we specifically use the functions alt_read_word(src_addr) and alt_write_word(dest_addr,word_data) to read/write the peripherals, respectively.

3.2.2 Beamformer SPI Programming

Since the system uses Verilog code to read and write varying data lengths from the LM96570 Beamformer via the FPGA, the next step is to connect the HPS system to the FPGA so that the LM96570 can be written to/read from the C program. Since the Verilog code is set up with 70-bit long registers, we needed three PIOs in our system in order to handle the resulting data. Specifically, any information written from the HPS to the FPGA had to be split up between these three PIOs (PIO0 = 32 bits, PIO1 = 32 bits, PIO2 = 6 bits). The same PIOs were also used by the HPS to read information from the FPGA registers.

The interaction between the C program on the system and the LM96570 chip occurs through the C method “write_beamformer_spi,” which is a void method that takes the parameters: unsigned char spi_reg_length, unsigned char read, unsigned char spi_addr, unsigned long spi_data_out, unsigned int *spi_in0, unsigned int *spi_in1, unsigned int *spi_in2.

The spi_reg_length is a value that informs the program the length of the data for the specific register address for the LM96570; these preset values occur in the lm96570_vars.h header file as “REG_XX_LENGTH” in order allow for easier reference for the size of specific registers of the LM96570 chip. The read variable represents the read/write bits that inform the system to read/write from the LM96570 beamformer. The spi_addr is the register address of the LM96570 beamformer that this method is reading from/writing.

The spi_data_out is the data written to the LM96570 beamformer for programming. The spi_in0, spi_in1, spi_in2 variables will be the data that comes into the system from the LM96570, whenever it is set up to read data.

Whenever this method gets calls, it will take the variables of spi_reg_length, read, spi_addr, and spi_data_out and load them into the three device PIOs accordingly via the alt_write_word() method. The 32 bit PIO0 holds the 4 bit spi_addr, 1 bit read, and first 26 bits spi_data_out, PIO1 holds the next 32 bits of spi_data_out, and PIO2 holds the remaining bits of spi_data_out. Also, the total length of the read/write instruction is stored and send to the FPGA as well. Once those parameters held by the system devices, the methods will send out a control signal which will initialize the LM96570 Verilog FSM to program the beamformer accordingly, and turn off the control signal when programming is complete.

With this method, the C program is used to fully program the registers of the LM96570 to generate different transmission profiles for the ultrasound system. Also, this method allows users to verify the correct programming of the registers on the LM96570 as well.

3.2.3 AD9276 SPI Programming

Since the AD9276 chipset utilizes the SPI protocol, we can connect to the system and program it with the SPI system block in Qsys. The method that controls the writing to the AD9276 is the write_adc_spi(unsigned int comm). This method works by looking at two specific bits that come from the SPI device on the HPS. These two bits are the status_TRDY_bit and the status_TMT_bit, which are used to determine the start of the writing stage and the reading stage respectively. The comm variable provides information that will be written to the SPI device, while the data variable stores the information read from the device. Since the Altera Qsys software generated this device, we are able

the obtain the definitions of the offsets for SPI_RXDATA, SPI_TXDATA, and SPI_STATUS, along with the definitions of the TRDY and TMT status bits from the avalon_spi.h header file.

Fortunately, the serial information (address, R/W, data) written to the AD9276 doesn't change size from one register to another, which makes programming the AD9276 easier compared to the LM96570. The AD9276 is set up to read in MSB mode, and according to the datasheet the total command that needs to be programmed is the following: (1 Read/Write bit, 2 data length bits, five address bits, 8 data bits). To achieve this correct order bit shifting and masking were used accordingly. Also, the definitions of the R/W bit and data length bits were defined in the "ad9276_vars.h" header file in order to improve the readability of the code.

3.2.4 Channel Selection

The programming and operation of the multiplexers occur through one PIO connection with a length of 10 bits; therefore each bit location will represent a set of 8 relays activated on the multiplexer board. The control of the channels are purely done using the alt_write_word() methods, but to select only eight channels at a time, the program bit-shifts a high bit to the left based on the switch events. Once we reach the final event via a while loop, all the switches will be turned off to protect the system.

3.2.5 Transmission and Reception (FIFOs)

Earlier sections talked about the C programming required to set up the system for operation. In this section, we will dig into the code that runs the entire system, which includes transmission, reception, and data storage during operation.

Transmission Code. Once the LM96570 beamformer and AD9276 receiver chips finish configuration, the system should be ready to fire ultrasound pulses, since the majority of the firing controls utilize one pin, we can fit most of the signal onto one PIO which we call the general control PIO. To begin, going back to the Verilog FSM that is used to run the entire FPGA side of the system, the C code first needs to reset this FSM before firing; this is to ensure that the system is in an empty state and that any possible glitching is mitigated. This reset signal is sent out through the general control PIO.

After resetting completes, the multiplexer switches to the next set of 8 channels. Afterward, the LM96530 T/R switch needs to have all eight channels turned on for the echoes to move to the AD9276 receiver, which occurs by setting the TR_EN and SW_OFF bits appropriately. The next step is the activation of the LM96551 pulser chip, which, as mentioned in the previous chapter, amplifies the pulse patterns generaated by the LM96570 beamformer. Once the TR switch and pulser are active; then the C code sends out the TX_Fire signal to the system, which will fire the pulse and start the Verilog FSM to go through the process of acquiring data. Once the FSM finishes, the LM96551 pulse will be turned off to converse the system's energy between transmissions.

As mentioned above, all these control signals come from a single PIO on the HPS. This PIO has a specific bit that corresponds to an FPGA pin that controls a part of the system. Due to this, bit masks were developed to make it easier to keep track of which PIO bit corresponded to the specific FPGA pin. The definitions of these masks are in the "general.h" header files for easier reference.

Reception - FIFOs. After the echoes in the ultrasound system move through the receiver, they are converted from analog to LVDS and transferred into the DE10-SoC. At this point, the FSM of the system uses the LVDS signals to create 16-bit words to fill the

FIFOs on the HPS. Once they are full from one set of transmission, they trigger a C program that unloads them into an in-program variable on the onboard memory.

The method that was generated to read these FIFOs was: read_adc_val(void *channel_csr_addr, void *channel_data_addr, unsigned int * adc_data). The inputs needed for the system were the channel_csr_addr, channel_data_addr, adc_data, which represent the control signal address, data address, and data extracted from the FIFO. The main goals of this method are to i) read the memory level of a specific FIFO, and ii) keep extracting data off of the FIFO until its memory level reaches zero.

While the method above works for one FIFO, due to the design of the system, we have 8 FIFOs set up that grab the ultrasound data in parallel, and multiple transmission events happening in one code. As a result, we need to store all this data into a holding variable until the system is done running through all the scanning. The method developed for this is store_data(unsigned int * adc_data, unsigned int data_bank [num_of_switches] [num_of_channels] [num_of_samples], unsigned int sw_num, unsigned int ch_num, unsigned int num_of_samples). The main function of this method was to organize the data from the FIFO into a three-dimensional array, where the data points organized by sample number, channel number, and switching number. Also, since the data from the FIFOs was 32 bits long, and we know the AD9276 has 16 bits of data per sample, this method also breaks the 32 bit FIFO data in half to provide the system with the 16 bit ADC data. This method occurs at every transmission to fill up the three-dimensional variable.

3.2.6 On-board Data Storage

Once the system has completed an entire ultrasound scan, the system should have a full 3-dimensional data bank. The next step would be to transfer this data or save it accordingly. The methods that were generated for this data transfer are write_data_bank (unsigned int data_bank[num_of_switches] [num_of_channels] [num_of_samples]), and print_data_bank (unsigned int data_bank[num_of_switches] [num_of_channels] [num_of_samples]).

Write_data_bank works to convert the 3d array variable and print it to a text file, to allow for later analysis and image processing. The configuration of the file places all written data samples on one line, with each different channel, will have its line, and each switching event will be separated from the previous event by an empty line between the two. On the other hand, the print_data_bank method works by printing each sample one at a time, running through from the first FIFO channel to the last final FIFO channel per switching event, and will repeat it for the next switching event. The reason for this for faster printing of the values to the system console, which is also a way to communicate between different programs that is useful for real-time applications.

3.3 Image and Video Processing

Once the system saves the data, the next step is to process it into meaningful information for the user. In our case, the data is presented either in the form as a single image or a video stream. For single image generation, the user saves the data on the system and can either choose to view it on the system or pull it off the system for further storage or analysis. On the other hand, moving to video imaging would involve the saving of images, each of which becomes one frame of a video, and continuously taking multiple frames of images to display the information to the user immediately.

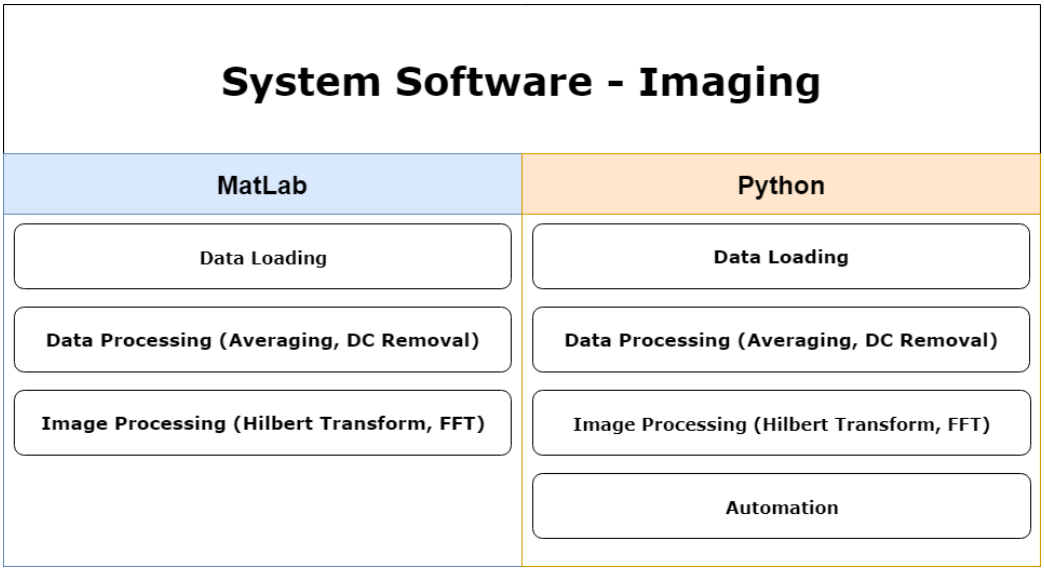

Figure 3.13. Diagram of imaging code for the system.

On the embedded side of the system, image and video processing relies on Python. Python provides a considerable advantage for imaging due to its built-in libraries, which allow for easier data manipulation and processing. Off-system image processing can utilize other programs like MATLAB or C. Since this system stores the information from one scan into a text file, such off-system processing helps in comparing our performance to other systems.

The on-board image processing and video processing of the ultrasound test bench operates using a Python program named the DE10 US imaging program. This program takes raw data acquired by the DE10 Standard C program and performs the necessary operations needed to extract valuable information from the raw data. The image formation screen and the methods used are explained in the following. Fig 3.13 provides a visualization of the overall image processing flow, which involves data extraction, data processing, and image generation.

3.3.1 Beamforming Methods

With the raw ultrasound data obtained from the system, the first image processing step is beamforming of the signals. We considered several approaches for this task, including both delay-and-sum (DAS) and phase-shift (PS) beamforming.

The Bartlett or DAS beamformer[21] is often used to construct ultrasound images. A DAS beamformer takes the digital transducer data sampled at the rate of f_s samples/s, and buffers/stores it before the beamforming process. After the storage of the last data sample, the beam output is computed. A disadvantage of this approach is the need to sample at rates much higher than the Nyquist frequency to approximate the time delays required for proper beam steering. As a result, large amounts of memory are needed to generate high-resolution images, which could make DAS inappropriate for resource-constrained and low power ultrasound imaging hardware.

On the other hand, the phase-shift (PS) beamformer[22] is a beamforming algorithm commonly used for narrowband signals. Here the beam steering delays are replaced by phase shifts which are easier to implement and are independent of the sampling frequency. This approach allows for a lower ADC sampling rate and power consumption. Also, in the far-field range, a PS beamformer is efficiently implemented through the application of a discrete Fourier transform (DFT) to the digital along the spatial dimension. The upside of this approach is its realization through computationally-efficient fast DFT algorithms (generally known as FFTs). The downside of this is that the FFT-based approach is only applicale to narrowband signals in the acoustic far-field, and its use on other signals (e.g., broadband and/or near-field ones) will result in imaging artifacts.

In the current state the proposed system uses sub-arrays of 8 transducer elements during each scan; the resulting acoustic apertures are small enough for all targets of interest to satisfy the far-field approximation. As a result, the system employs an approximate DFT (aDFT) beamformer[23]. The aDFT approximates the complex coefficients of a DFT matrix with simple integer coefficients for hardware-efficient beamforming and subsequent image construction. Specifically, the aDFT (i) preserves symmetry, (ii) replaces multiplications with bit-shifts, and (iii) can be factorized into FFT-like fast algorithms. The lack of multipliers in aDFT reduces hardware usage and power. The aDFT also results in faster image construction than FFT when implemented in hardware, which increases the ultrasound frame rate.

3.3.2 Image Construction Scheme

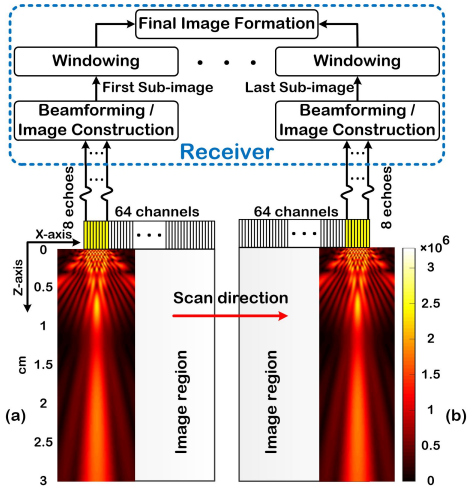

Figure 3.14. The image formation scheme. A plane wave exciting a) the first, and b) the last section of the imaging region through an 8-transducer sub-window are shown. Each excitation pattern results in a 8-pixel sub-image. FOCUS[10] was used to compute and display the pressure field for each pattern.

Based on the hardware design of the system, the image formation scheme in Fig. 3.14. First, the 64 ultrasound transducer channels divided into sub-arrays of 8 channels per scan. To improve the quality of ultrasound image construction, we apply a 25% overlap between the sub-arrays; this was done to reduce possible edge effects while merging multiple sub-images. Accordingly, in each incremental scan, six new channels, as well as the two last channels of the previous set of elements, are used as the active channels (i.e., to send and receive ultrasound signals).

The next stage is to convert the received real signals from the 8 active channels in each scan to a complex analytic representation via the Hilbert transform. Then, the aDFT beamformer is applied, which results in receive-mode beamforming and dynamic focusing at different depths and generates a sub-image. The windowed sub-images are then merged (stacked) in order to generate a complete image. Finally, dynamic range compression (e.g., using a logarithmic mapping and gamma correction) is applied to the merged image before display.

The pressure fields for the first and last scans in the imaging region are shown in Fig. 3.14. Based on the pressure field profile shown in Fig. 3.14, the target under test (described in the next section) falls within the far-field region, which verifies that an aDFT can efficiently implement a PS beamformer on the digitized signals along the spatial dimension.

3.3.3 Single Image Generation - Off System Validation

The text files generated by the C program were uploaded into MATLAB running on an off-system computer to parse the data and perform image processing. The process of collecting this data and forming an offline image is as follows:

Data loading occurs from the text file into a numeric matrix, which is 88 rows (number of channels collected) by 8192 columns (number of data samples per scan). Once this matrix is loaded, the averages for each specific echo is subtracted from its echo channel to remove any potential DC offsets coming from the system. With the averages removed from the echoes, we then apply the Hilbert transform to the resulting matrix with the built-in MATLAB function Hilbert(xr), thus separating the data into real and complex parts (i.e., generating an analytic signal).

The analytic signal is represented by the equation $x = xr + jxi$, i.e., is composed of a real part xr (the original data) and an imaginary part xi (generated by the Hilbert transform). This Hilbert-transformed data series has the same amplitude and frequency as the original data, but now contains both magnitude and phase information.

The main use of the Hilbert transform is to calculate the instantaneous amplitude and frequency of the data. The instantaneous amplitude is the amplitude of the complex Hilbert transform, while the instantaneous frequency is the time rate of change of the instantaneous phase.

Since we are dealing with a 25 percent overlap, the last two channels in the previous scan are the first two channels in the subsequent scan. To deal with this, there needs to be averaging of these two specific channels across the two scans. Such averaging is done via a trapezoidal window of eight elements. This window is used to combine image segments obtained from eight-element sub-arrays.

Before windowing of the image segments, the discrete Fourier transform of the eight channels for a specific image segment is obtained. This uses the MATLAB function FFT(X,n,dim), where X represents the data matrix, $n = 8$ represents the length of the sub-array, and $dim = 1$ to ensure that the algorithm operates along the columns of X.

After the MATLAB FFT function, the trapezoidal window is applied to the acquired FFT data, via array multiplication across eight rows of FFT data. The next step is to remove some of the rows of X. The reason for this removal is due to the setup of the HV multiplexer. Specifically, during the final scan, only the last two channels are connected to the transducer. Thus, there is no relevant ultrasound data from the other channels, which are thus removed. The sub-arrays are now combined to to generate a 64-pixel-wide image. the absolute intensity values of this image are then extracted using the MATLAB abs() function.

Since the ultrasound system is firing pulses centered around a specific frequency, any spatial information that occurs between the start and end of the pulse pattern is undetectable by the system. Due to the wavelength of this pulse signal and spatial resolution, there is no noticeable difference between adjacent data samples in the final dataset. Thus, the image matrix can be compressed with no loss of spatial resolution in the image.

We chose a compression factor of 16 based on the number of data samples used during the length of the transmission signal. The image columns were then averaged across 16-element windows, and the results stored in a smaller data matrix (64×512 instead of 64×8192). The compressed data was converted to grayscale intensities via MATLAB's mat2gray() function and then readjusted using the imadjust() function. The resulting image was displayed using MATLAB's imshow() or imagesc() functions.

3.3.4 Code needed for a Single Image - On-System Validation

The acquired data was first validated and verified off-system using MATLAB, as described above. The next step was to replicate the image processing code on the DE10. Due to memory constraints on the embedded system and the unavailability of MATLAB on

ARM processors, the code was rewritten in Python. The necessary libraries and sub-files included: time, time delta from the DateTime library, numpy library, os library, matplotlib.pyplot, and Hilbert from scipy.signal library.

Time, datetime, and timedelta are used for time tracking within the function, which allows for the evaluation of the time between subprocesses in the Python program. Numpy is used to define an array and allows basic math on arrays. Os enable the program to find and change directories as the need to find specific files. The scipy.signal library containing a large number of signal processing functions, like the Hilbert function.

The Python program first detects which directory it is in and changes its directory to where the databank.txt file. Once in that directory, the program loads the resulting time data from the text file into an array. From there, the Python program follows the same steps as the off-system MATLAB program except for the use of Python libraries. It removes the DC offsets from each echoes using the .sum() function, applies the Hilbert transformation to the data via the Hilbert() function, performs the FFT across 8 rows of echos using the fft.fft2() and fft.fftshift() functions, removes unnecessary rows, multiplies sub-array images with trapezoidal windows, compresses the image along the time dimensions, and displays images using the .canvas.draw() function. Also, a significant difference between the MATLAB and Python versions is that the array types need to be predefined in Python, while MATLAB does type conversion automatically.

3.3.5 Codes Used for Automated Image Construction/Video

After setting up the single image code for the embedded system, the next step is to allow the system to continuously run and generate images after data acquisition, which allows for the system to move into video processing. The system needs to set up the necessary parameters/arrays needed for the image processing and also enable external programs

to control the firing of the system and extract the data accordingly. Compared to the single image version of the code on the embedded system there are a few improvements in the Python Code to allow it to run external programs and improve the imaging times to allow continuous imaging.

At this stage, this version of the Python code contains many of the same libraries and parameters loaded in the single-image version of the code. It can start the Ultrasound Transmit C program, which configures the system and acquires the image data. However, loading and processing the resulting data file takes up significant memory and time. Thus, this version of the Python code can initialize and run the DE10 Standard US C program, thus allowing it to directly read data into Python for quicker image processing.

The pOpen function from the subprocess library is used for this purpose. This function looks for a predefined program, which is entered into the method and sets up PIPEs for it. The purpose of these PIPEs is to allow information printed in the system console to be ported directly into another program. In this way, the raw data from the ultrasound system is directly sent into the Python code. Of course, the fact that the information is first outputted to the console and then read into Python means there a few adjustments are needed in order to ensure that only the raw data is read. After being read, the data is loaded into an array via the .split() and .array() functions. From there, the program is similar to the single image processing code, but with a few minor improvements for efficiency.

4 Experimental Setup and Results

Given the software and hardware designs mentioned in previous chapters, we are ready to test system performance by generating single ultrasound images, as well as continuous image sequences. This chapter breaks down the work involved in getting the system to its current working version.

4.1 System Summary and Experimental Setup

When fully assembled, the bench-top prototype is capable of using eight physically active transmit/receive channels across a linear array transducer with 64 piezoelectric elements. Fig. 2.1 summarizes the prototype and labels its subsystems, and Table 4.1 summarizes its performance parameters.

For system testing, a phantom was created to simulate a human artery and surrounding body tissue. The main phantom used for scanning (see Fig. 4.1) consists of a thin-walled silicone tube (diameter = 6 mm) filled with water and embedded within gelatin that mimics the acoustic properties of human body tissue where the speed of sound is around 1540 m/s. The silicon tube was implanted about 1.5 cm from the scanning side of the gelatin phantom.

Table 4.1. Parameters of the Experimental Prototype

Transducer type	Linear Array
Number of elements	64
Transducer center frequency	8.0 MHz
Transducer bandwidth (BW)	5-10 MHz
Element pitch	0.3 mm
Depth of focus	2 cm
Drive voltage level	up to 48 V
Receiver dynamic range	107 dB (for BW = 5 MHz)
On-board memory	16 GB

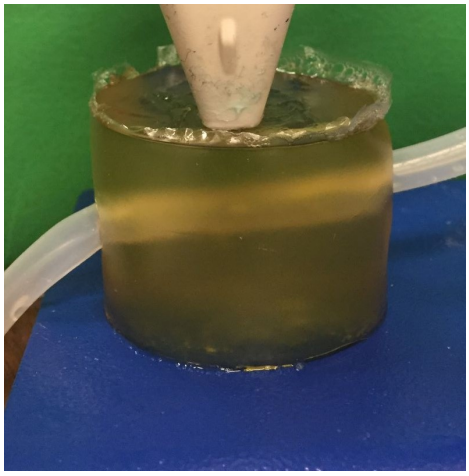

Figure 4.1. Example of Gelatin Phantom used for Imaging

4.2 Transmission Profile

The system was configured to fire pulses that fit within the transducer bandwidth. Specifically, it was set to a firing frequency of 8 MHz in order to match the center frequency of the Blatek transducer. We fist verified that proper transmission waveforms were being fired by the pulser (see Fig. 4.2 for an example). Next, we confirmed proper functionality of the transmit/receive (T/R) switches. These switches clamp the receiver voltage to a range of ± 0.7 V, thus ensuring that the sensitive low-noise analog front-end is protected from the high-voltage transmit pulses.

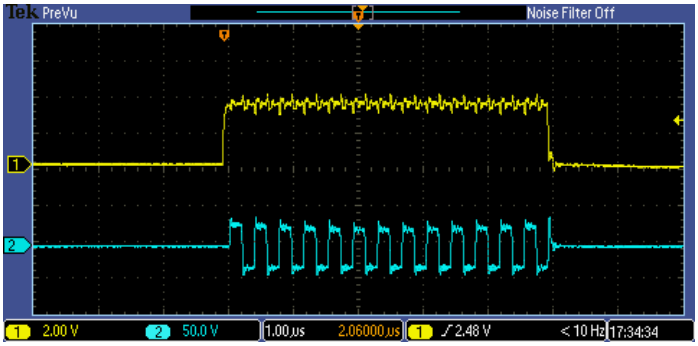

Figure 4.2. Example of an 8 MHz transmit pulse generated by the system.

4.3 Validation of Digital Signals to DE10

After confirming transmitter operation, the next step is to ensure the accuracy of digitized LVDS signals generated by the receiver. Since these signals are assigned to specific pins on the HSMC connector on the DE10 board, the digital signals observed at these pins must be interpreted to check the accuracy of the signals. Fortunately, the Altera Quartus Prime Software toolset comes with a built-in Logic Analyzer named Signal Tap. With Signal Tap, a user can run the Verilog-based firmware in real-time on the DE10 FPGA fabric and select specific pins, nets, or registers in order to observe their states while the system is running.

Given that Signal Tap allows signals at points of interest on the FPGA fabric to be observed, we can test the ADC-FGPA interface by generating known input signals. Fortunately, the AD9276 receiver chip has an assortment of test output configurations set via the SPI lines. For example, these output test patterns can range from several shorts (Mid-Scale, +Full-Scale, −Full-Scale), predetermined patterns (e.g., checkerboard) and user-inputted patterns. By utilizing the FPGA's response to such known inputs, the parameters of the AD9276 (e.g., its gain and impedance) can be configured to achieve the best accuracy.

Table 4.2. Mutual information (MI) values between images constructed by the Vantage 64 research ultrasound system and different custom beamformers.

Beamformer	MI value
PS with fixed focal point (Fig. 4.3(c))	0.9846
PS with floating focal point (Fig. 4.3(d))	1.0387
DFT (Fig. 4.3(e))	1.0035
aDFT (Fig. 4.3(f))	0.9999

By utilizing SignalTap and the test outputs of the AD9276, the parameters of the AD9276 were set to the following: PGA gain: 21 dB; LNA gain: 15.6 dB; output driver impedance: 100 Ω, Output phase: 420 degrees. After these parameters were loaded into the system, Signal Tap was utilized to verify the signals acquired with unknown inputs (e.g., echoes generated by transmit pulses).

4.4 Off-system Image Generation

After the ADC-FGPA interface has been optimized, the next step is to take the generated text files from the system and perform image processing on an off-system computer to confirm which image processing method would be the most advisable to use on the embedded system.

In this scanning set, the transmitter frequency, pulse length, and voltage of the system were set to 8 MHz, eight cycles, and 35 V, respectively, which results in a spatial pulse length (SPL) and axial resolution of 1.54 mm and 0.77 mm within our phantom. The lateral resolution (which is set by element pitch of the transducer) is ~0.3 mm at depths of several mm. Since the prototype has eight physical channels and eight transducers were simultaneously turned on to transmit and receive during each scan. The sub-windows are defined with 25% overlap to sequentially excite the whole image region with plane waves within 11 scans (see Fig. 3.14).

Given one set of raw data acquired by the prototype, several images of the phantom were generated off-system with various beamforming methods, as shown in Figs. 4.3. Specifically, Figs. 4.3 (c)-(f) show images constructed using the PS beamformer with a fixed focal point, PS beamformer with moving focal point, the DFT beamformer, and the aDFT beamformer, respectively. These images were compared to a Verasonics Vantage 64 research ultrasound system equipped with a 128-channel linear probe (bandwidth: 4-11 MHz), and set to the same center frequency of 8 MHz. The results verify that the proposed prototype works and generates reasonable images. For example, the inner and outer diameters of the artery phantom can be measured with good accuracy. The mutual information (MI) values between the images shown in Fig. 4.3(c)-(f) and the reference image shown in Fig. 4.3(b) are listed in Table 4.2. These results confirm that each of our beamforming methods results in images with high MI. Thus, they are all suitable for use in our custom ultrasound imaging system.

4.5 Embedded Image Processing: Single and Multi

With off-system image processing demonstrated, the next step was to move the image processing to the embedded HPS side of the system. While all the proposed beamformers would work, the aDFT beamformer was selected for implementation on the embedded processor because of its low computational complexity.

To run the system and observe the generated images on the on-board Linux environment, a terminal emulator and remote X-server (PuTTY and Xming, respectively) had to be initialized to allow remote graphical access into the system. By executing the on-system C and Python programs, the system was able to generate single images in < 1 sec, and when running in continuous acquisition mode was able to generate a new

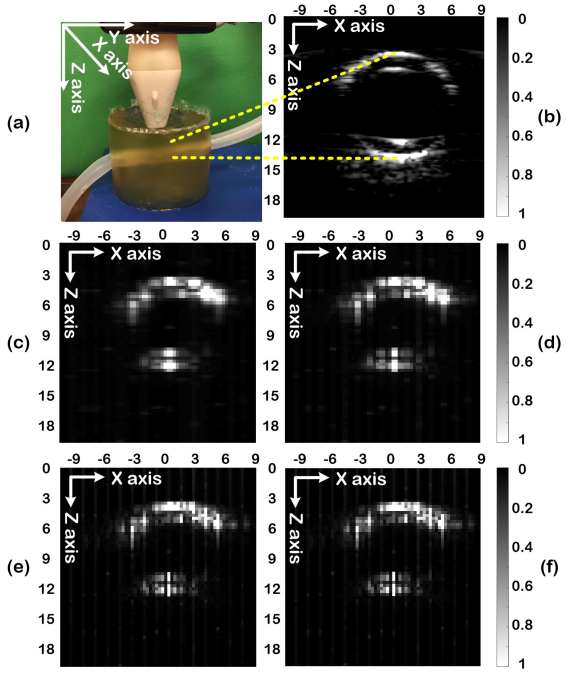

Figure 4.3. (a) An artery phantom consisting of a thin-walled silicone tube (diameter = 6 mm) embedded within gelatin; (b) Image of the cross section of the artery phantom measured using the Verasonics system; (c)-(f): images of the same structure constructed using the proposed bench-top prototype by applying (c) phase shift beamforming with fixed focal point of 2 cm, (d) phase shift beamforming with floating focal point, (e) DFT beamforming, and (f) aDFT beamforming (all values on the images are in mm).

image every 5 sec. A typical image of the artery phantom as generated by the system is shown in Fig. 4.4.

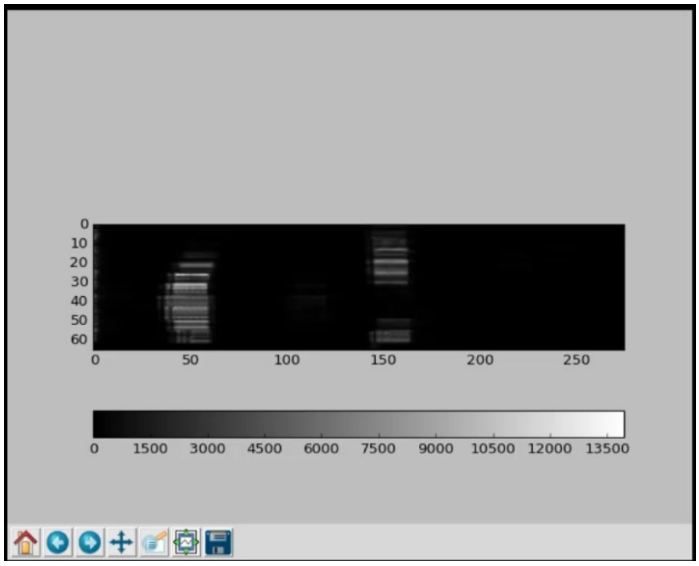

Figure 4.4. Typical image of the artery phantom generated on the embedded ARM processor.

5 Conclusion

This thesis has demonstrated a fully-working prototype of a programmable low-cost 64-channel ultrasound imaging system. The system allows easy prototyping and fast development of wearable and autonomous devices. Our goal is to keep the system free and open-source[1].

PCB design/assembly, Verilog-based firmware, and software development in C and Python, were necessary in order to complete this project. The current configuration of the system was validated against a PC-based research ultrasound system. Initial results are promising: they confirm that the current test-bench can generate images that are similar to the PC-based system without relying on external processing. Such embedded imaging capabilities are expected to be particularly useful for autonomous devices in portable and wearable form factors. These results have been published in several peer-reviewed publications[24–26].

[1] All design files are downloadable from https://goo.gl/zpSWbw.

6 Suggested Future Research

Future research projects will drive further improvements to the system that enable various applications in autonomous, wearable, and implantable ultrasound imaging. A few suggestions for improvements are as follows:

6.1 Hardware

- Addition of direct digital synthesis (DDS) for electronic control of the sampling frequency of the receiver subsystem.
- Merging all custom hardware subsystems onto a single board in order to reduce dependency on cables and external connecting hardware.
- Improvement of power stability and generation to reduce the hardware footprint of the system.
- Increasing number of transmission/reception active channels in one scan.

6.2 Software

- Development of a user interface to provide easier control of the system.

- Improvement of data acquisition and image formation software to allow for faster scan times, thus enabling real-time scanning and also allowing video processing methods.
- Development of additional image processing techniques to extract clinically-relevant information for specific biomedical ultrasound applications.

Appendix A
Schematics and Layouts of PCBs of System Prototype

This section covers the schematics and layouts of the custom PCBs generated for the system Prototype.

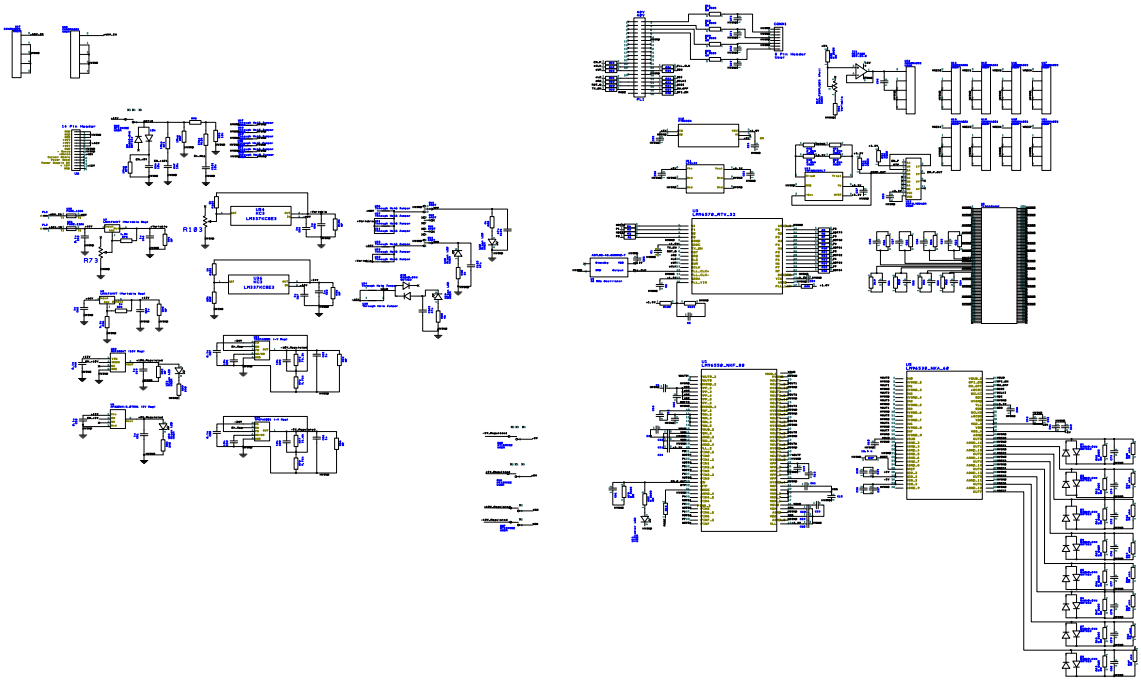

Figure A.1. Schematic of Custom Transmitter Board.

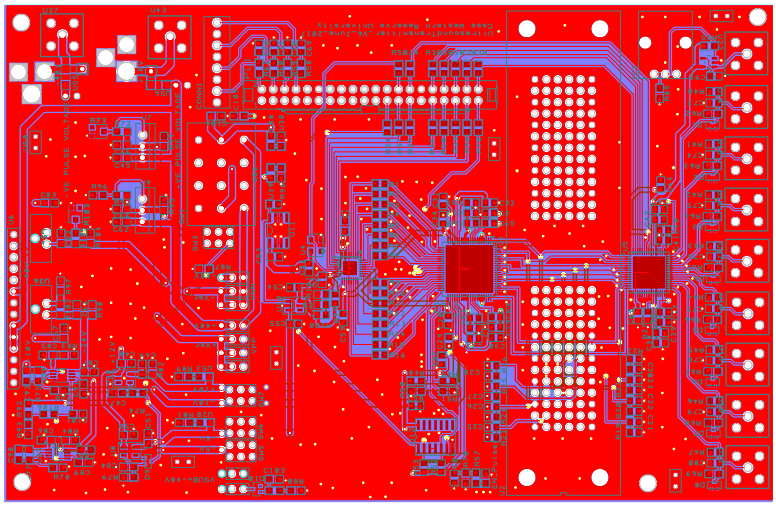

Figure A.2. PCB Layout of Custom Transmitter Board. All Layers.

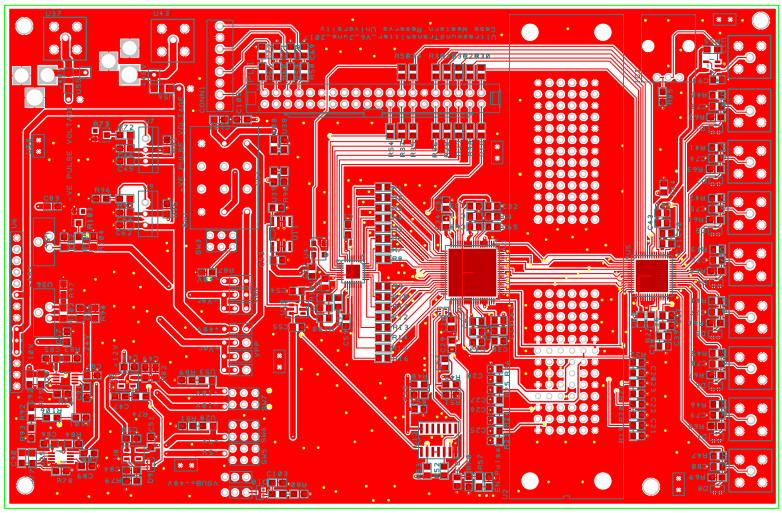

Figure A.3. PCB Layout of Custom Transmitter Board. Top Layer.

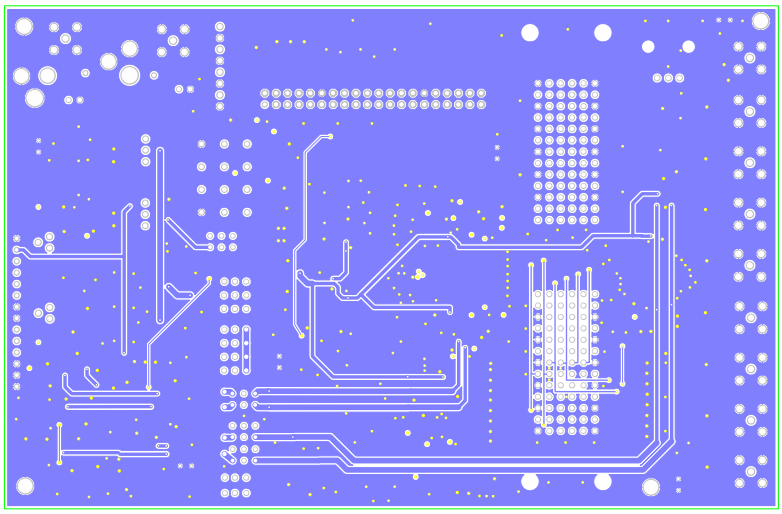

Figure A.4. PCB Layout of Custom Transmitter Board. Inner Layer 2.

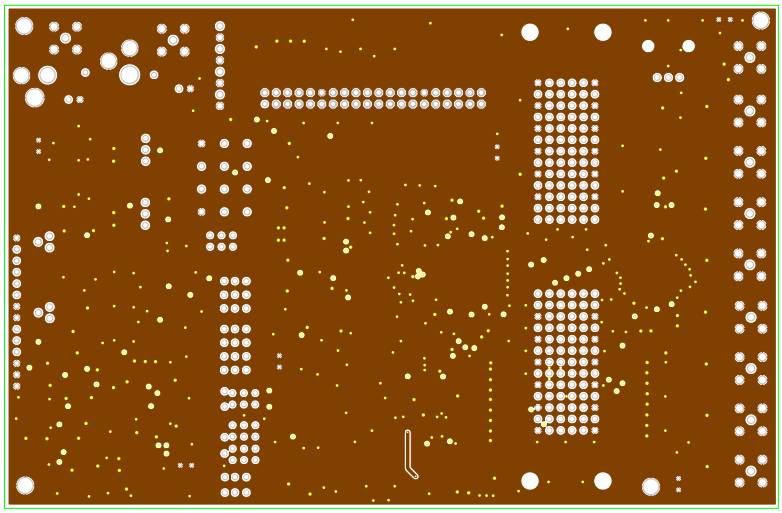

Figure A.5. PCB Layout of Custom Transmitter Board. Inner Layer 3.

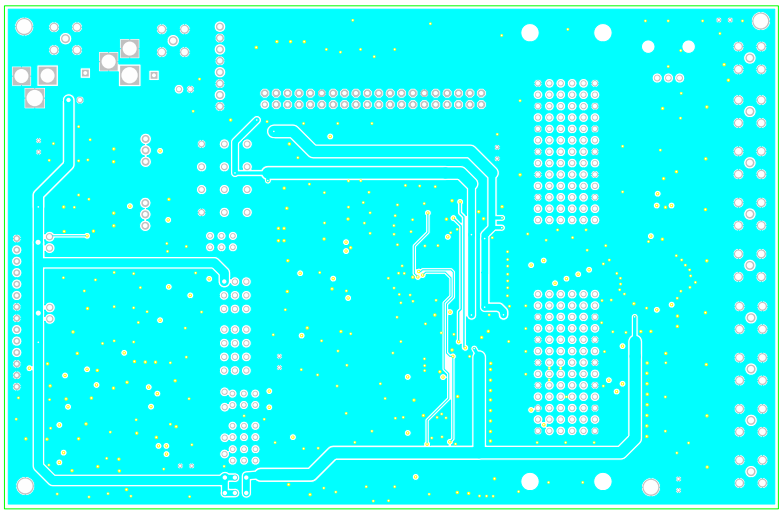

Figure A.6. PCB Layout of Custom Transmitter Board. Bottom Layer.

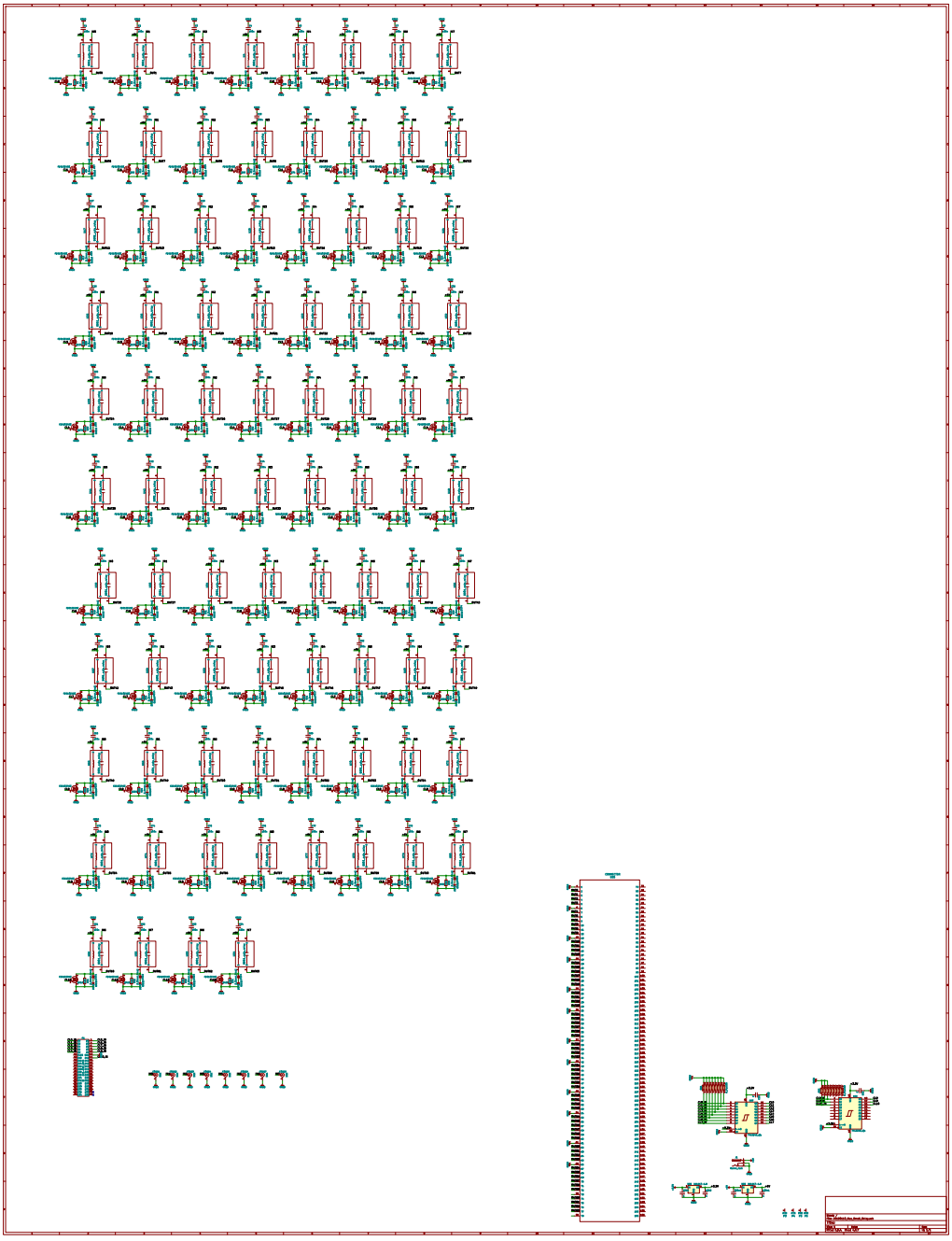

Figure A.7. Schematic of Custom Multiplexer Board.

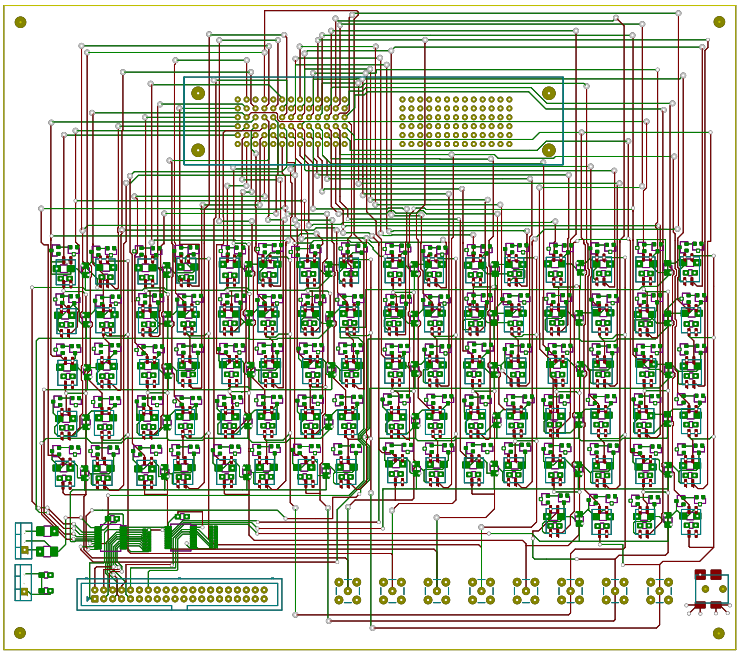

Figure A.8. Layout of Custom Multiplexer Board.

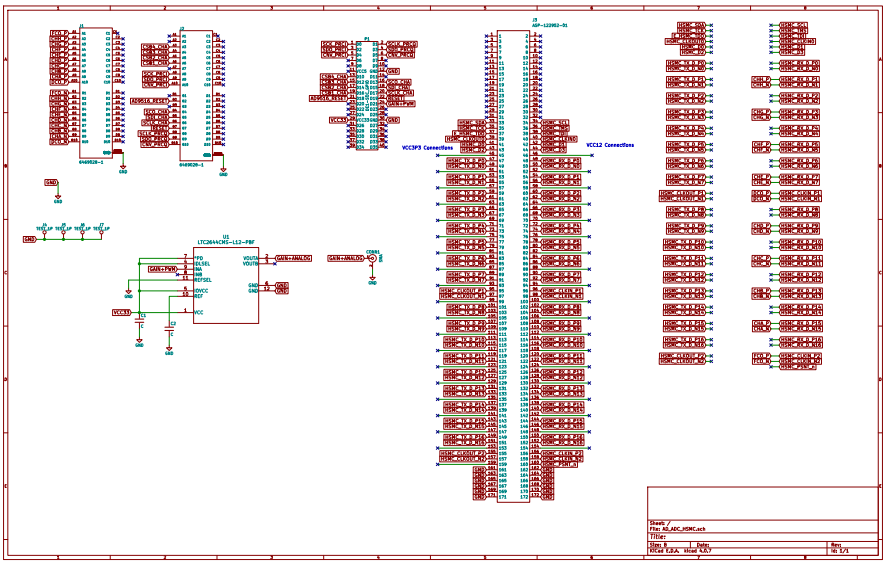

Figure A.9. Schematic of ADC HSMC Interposer Board

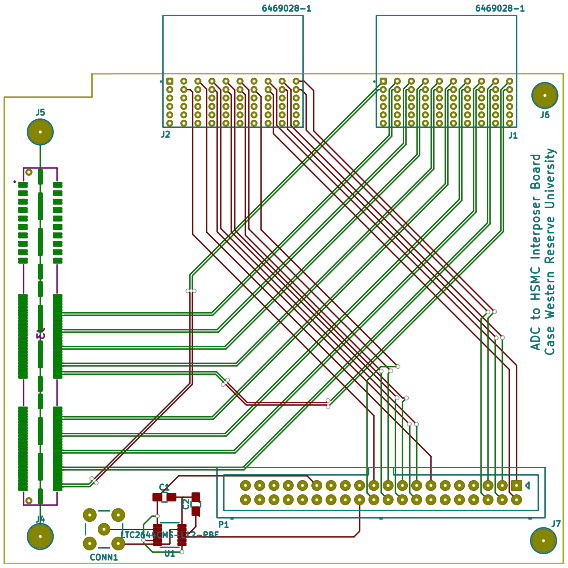

Figure A.10. Layout of ADC HSMC Interposer Board

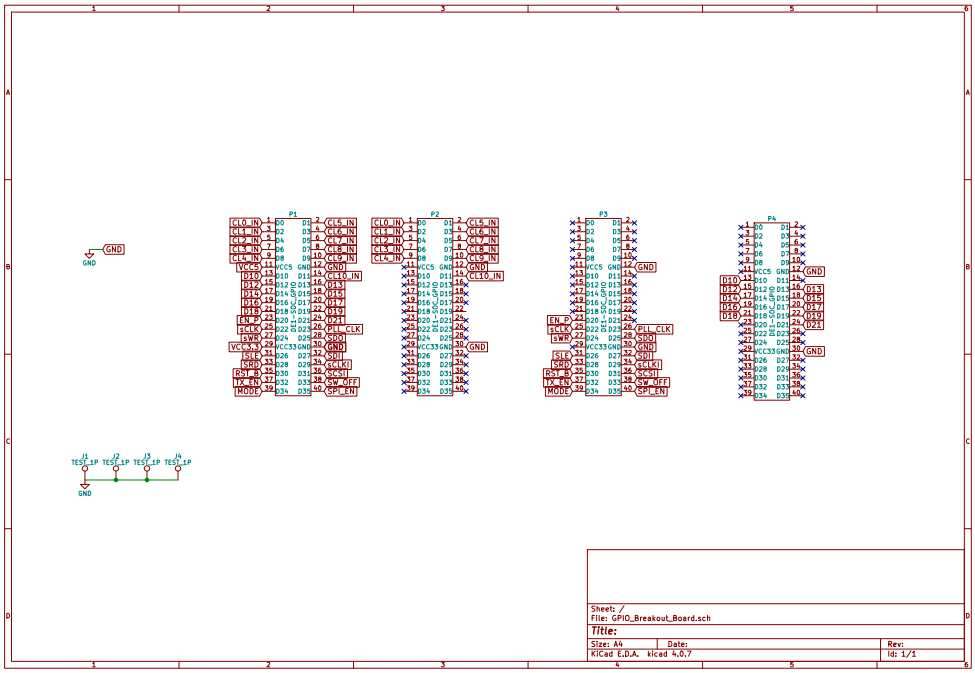

Figure A.11. Schematic of GPIO Breakout Board.

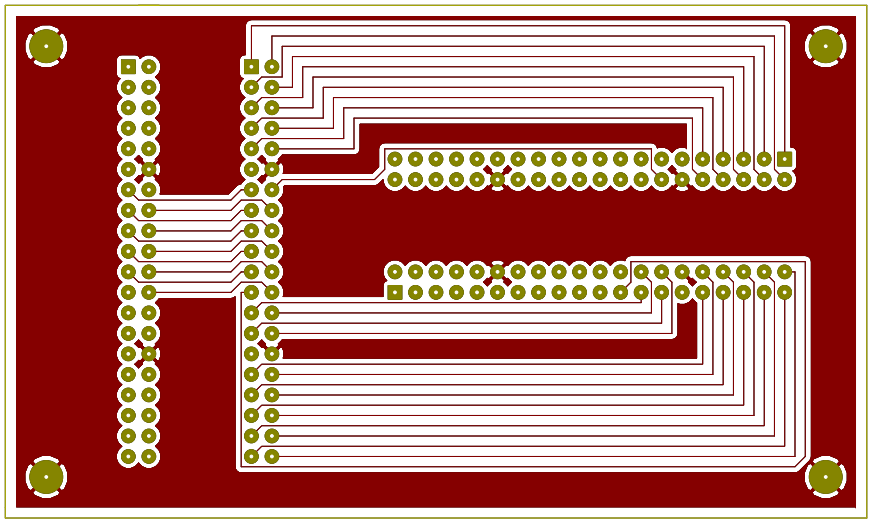

Figure A.12. Layout of GPIO Breakout Board.

Appendix B
Verilog Code of System Prototype

This section covers the Verilog Code that was utilized in the system prototype.

1 LM96570_SPI.v

```
‘timescale 1ns / 1ps
module LM96570_SPI
#(
parameter DATA_WIDTH = 70,
parameter BIT_COUNT_WIDTH = 8 // to accomodate shifting 70 bits of data
)
(
// Control Signals
input START,// start signal
output reg DONE,// acquisition done

// Data Signals
output reg [(DATA_WIDTH-1):0]  RD_DATA,// data wires

// SPI output signals
input [BIT_COUNT_WIDTH-1:0]  NUM_OF_BIT,// number of bits being shifted
input [DATA_WIDTH-1:0] DATA_IN,

// SPI Bus Signals
output reg  SPI_CS_N,
output reg  SPI_SCK,
input       SPI_SDI,
output reg SPI_SDO,

// System Signals
input CLK,
input RESET
);

// Include StdFunctions for bit_index()
// ‘include "StdFunctions.vh"

// localparam BIT_COUNT_WIDTH = bit_index(DATA_WIDTH);
```

```
//
// The SPI Bus Rate is half of the system clock input (CLK)
//

//
// SPI Transceiver State Machine
//
reg [8:0] State;
localparam [8:0]
S0 = 9'b000000001,
S1 = 9'b000000010,
S2 = 9'b000000100,
S3 = 9'b000001000,
S4 = 9'b000010000,
S5 = 9'b000100000,
S6 = 9'b001000000,
S7 = 9'b010000000,
S8 = 9'b100000000;


reg [(DATA_WIDTH-1):0] shift_reg_rx, shift_reg_tx;

reg [BIT_COUNT_WIDTH-1:0] bit_count; // count shifted bit



initial
begin
bit_count <= {BIT_COUNT_WIDTH{1'b0}};
RD_DATA <= {DATA_WIDTH{1'b0}};
end

always @(posedge CLK, posedge RESET)
begin

if (RESET)
begin

shift_reg_rx <= {DATA_WIDTH{1'b0}};

SPI_CS_N <= 1'b1;
SPI_SCK <= 1'b0;
```

```
DONE <= 1'b0;

State <= S0;

end
else
begin

case (State)

S0 :
begin

// Reset the sample count

// reset the bit count
bit_count <= {1'b1, { (BIT_COUNT_WIDTH-1) {1'b0} }} - NUM_OF_BIT;

// Wait for the Start signal
if (START)
State <= S1;

end

S1 :
begin

// Initialize the shift register
shift_reg_rx <= { (DATA_WIDTH) {1'b0} };
shift_reg_tx <= DATA_IN;

// Clear the Done signal
DONE <= 1'b0;

// Assert Chip Select
SPI_CS_N <= 1'b0;

// Reset the bit count
bit_count <= {1'b1, { (BIT_COUNT_WIDTH-1) {1'b0} }} - NUM_OF_BIT;

State <= S2;
```

```
end

S2 :
begin

SPI_SDO <= shift_reg_tx[0];
shift_reg_tx <= {1'b0, {shift_reg_tx[DATA_WIDTH-1:1]} };

// SCK Falling-Edge
SPI_SCK <= 1'b0;

// increment bit counter
bit_count <= bit_count + 1'b1;

State <= S3;

end

S3 :
begin

// shift data in
shift_reg_rx <= { SPI_SDI, shift_reg_rx [DATA_WIDTH-1:1] };

// SCK Rising-Edge
SPI_SCK <= 1'b1;

if (bit_count[BIT_COUNT_WIDTH-1])
begin

State <= S4;

end

else
State <= S2;

end

S4 :
begin

SPI_SCK <= 1'b0;
```

```
State <= S5;

end

S5 :
begin

// Deassert Chip Select
SPI_CS_N <= 1'b1;

State <= S6;

end

S6 :
begin

// Reset the bit count
bit_count <= {1'b1, { (BIT_COUNT_WIDTH-1) {1'b0} }} -
(DATA_WIDTH-NUM_OF_BIT-1);

State <= S7;

end

S7 :
begin

bit_count <= bit_count + 1'b1;
shift_reg_rx <= { 1'b0, shift_reg_rx [DATA_WIDTH-1:1] };

if (bit_count[BIT_COUNT_WIDTH-1])
begin

State <= S8;

end

else
begin
State <= S7;
end
```

```
end

S8 :
begin

RD_DATA <= shift_reg_rx;

// Assert Done signal
DONE <= 1'b1;

State <= S0;

end

endcase

end

end

endmodule
```

2 ADC_AD9276.v

```
‘timescale  1 ns / 1 ps
module ADC_AD9276 (
rst, wren,
dco_p, dco_n, fco_p, fco_n,
dina_p, dina_n, dinb_p, dinb_n,
dinc_p, dinc_n, dind_p, dind_n,
dine_p, dine_n, dinf_p, dinf_n,
ding_p, ding_n, dinh_p, dinh_n,
doutA, doutB, doutC, doutD, doutE, doutF, doutG, doutH,
clkout, load, dco_locked, fco_locked
);

input rst, wren;
input dco_p, dco_n;
input fco_p, fco_n;
input dina_p, dina_n;
input dinb_p, dinb_n;
```

```
input dinc_p, dinc_n;
input dind_p, dind_n;
input dine_p, dine_n;
input dinf_p, dinf_n;
input ding_p, ding_n;
input dinh_p, dinh_n;

output [13:0] doutA; //Setting Up for 16-bit words
output [13:0] doutB;
output [13:0] doutC;
output [13:0] doutD;
output [13:0] doutE;
output [13:0] doutF;
output [13:0] doutG;
output [13:0] doutH;
output clkout, load;
output dco_locked;
output fco_locked;

reg load, gate;
reg [6:0] dat_q1_a, dat_q2_a; //Setting Up for 12-bit words
reg [6:0] dat_q1_b, dat_q2_b;
reg [6:0] dat_q1_c, dat_q2_c;
reg [6:0] dat_q1_d, dat_q2_d;
reg [6:0] dat_q1_e, dat_q2_e;
reg [6:0] dat_q1_f, dat_q2_f;
reg [6:0] dat_q1_g, dat_q2_g;
reg [6:0] dat_q1_h, dat_q2_h;

reg [13:0] doutA;
reg [13:0] doutB;
reg [13:0] doutC;
reg [13:0] doutD;
reg [13:0] doutE;
reg [13:0] doutF;
reg [13:0] doutG;
reg [13:0] doutH;

reg [13:0] dout_a, dout_b;
reg [13:0] dout_c, dout_d;
reg [13:0] dout_e, dout_f;
reg [13:0] dout_g, dout_h;
```

```
reg q2_a_dly;
reg q2_b_dly;
reg q2_c_dly;
reg q2_d_dly;
reg q2_e_dly;
reg q2_f_dly;
reg q2_g_dly;
reg q2_h_dly;

reg [15:0] count;
reg fco1,fco2;
reg fcostb;

wire fco;

wire dco;
wire dcoshifted;

//Altera Implementations of above buffers, Check this again...
alt_inbuf_diff IB1 (.i(dco_p), .ibar(dco_n), .o(dco_in));
alt_inbuf_diff IB2 (.i(fco_p), .ibar(fco_n), .o(fco_in));

alt_inbuf_diff IB3 (.i(dina_p), .ibar(dina_n), .o(din_a));
alt_inbuf_diff IB4 (.i(dinb_p), .ibar(dinb_n), .o(din_b));
alt_inbuf_diff IB5 (.i(dinc_p), .ibar(dinc_n), .o(din_c));
alt_inbuf_diff IB6 (.i(dind_p), .ibar(dind_n), .o(din_d));
alt_inbuf_diff IB7 (.i(dine_p), .ibar(dine_n), .o(din_e));
alt_inbuf_diff IB8 (.i(dinf_p), .ibar(dinf_n), .o(din_f));
alt_inbuf_diff IB9 (.i(ding_p), .ibar(ding_n), .o(din_g));
alt_inbuf_diff IB10 (.i(dinh_p), .ibar(dinh_n), .o(din_h));

//Altera Implementation of DCO Clk (Previous MegaFunctions: Alt_PLL)
pll_adc_datain pll_adc_datain1 (
.refclk (dco_in),
.locked (dco_locked),
.rst (rst),
.outclk_0 (dco)
);

/*
dcm2 DCM2 (
.refclk (fco_in),
.locked (fco_locked),
```

```
.rst (rst),
.outclk_0 (fco)
);
*/
assign fco = fco_in;

//Altera Implementation of DDR Inputs...
IDDR1 I1 (
.datain (din_a),
.inclock (dco),
.dataout_l (q1_a),
.dataout_h (q2_a),
.inclocken (1'b1),
.aset (1'b0)
);

IDDR1 I2 (
.datain (din_b),
.inclock (dco),
.dataout_l (q1_b),
.dataout_h (q2_b),
.inclocken (1'b1),
.aset (1'b0)
);

IDDR1 I3 (
.datain (din_c),
.inclock (dco),
.dataout_l (q1_c),
.dataout_h (q2_c),
.inclocken (1'b1),
.aset (1'b0)
);

IDDR1 I4 (
.datain (din_d),
.inclock (dco),
.dataout_l (q1_d),
.dataout_h (q2_d),
.inclocken (1'b1),
.aset (1'b0));

IDDR1 I5 (
```

```
.datain (din_e),
.inclock (dco),
.dataout_l (q1_e),
.dataout_h (q2_e),
.inclocken (1'b1),
.aset (1'b0)
);

IDDR1 I6 (
.datain (din_f),
.inclock (dco),
.dataout_l (q1_f),
.dataout_h (q2_f),
.inclocken (1'b1),
.aset (1'b0)
);

IDDR1 I7 (
.datain (din_g),
.inclock (dco),
.dataout_l (q1_g),
.dataout_h (q2_g),
.inclocken (1'b1),
.aset (1'b0)
);

IDDR1 I8 (
.datain (din_h),
.inclock (dco),
.dataout_l (q1_h),
.dataout_h (q2_h),
.inclocken (1'b1),
.aset (1'b0)
);

// the altera IDDR megafunction starts with falling-edge first and
   then rising-edge
// in the other hand, the AD9276 starts with rising-edge and then
// falling-edge therefore we need to make dataout_h delayed by 1 cycle
// and capture the signal late by 1 cycle
always @(posedge dco)
begin
q2_a_dly <= q2_a;
```

```
q2_b_dly <= q2_b;
q2_c_dly <= q2_c;
q2_d_dly <= q2_d;
q2_e_dly <= q2_e;
q2_f_dly <= q2_f;
q2_g_dly <= q2_g;
q2_h_dly <= q2_h;
end


always @(posedge dco)
begin
dat_q1_a[6:0] <= {dat_q1_a[5:0], q1_a};
dat_q2_a[6:0] <= {dat_q2_a[5:0], q2_a_dly};
dat_q1_b[6:0] <= {dat_q1_b[5:0], q1_b};
dat_q2_b[6:0] <= {dat_q2_b[5:0], q2_b_dly};
dat_q1_c[6:0] <= {dat_q1_c[5:0], q1_c};
dat_q2_c[6:0] <= {dat_q2_c[5:0], q2_c_dly};
dat_q1_d[6:0] <= {dat_q1_d[5:0], q1_d};
dat_q2_d[6:0] <= {dat_q2_d[5:0], q2_d_dly};
dat_q1_e[6:0] <= {dat_q1_e[5:0], q1_e};
dat_q2_e[6:0] <= {dat_q2_e[5:0], q2_e_dly};
dat_q1_f[6:0] <= {dat_q1_f[5:0], q1_f};
dat_q2_f[6:0] <= {dat_q2_f[5:0], q2_f_dly};
dat_q1_g[6:0] <= {dat_q1_g[5:0], q1_g};
dat_q2_g[6:0] <= {dat_q2_g[5:0], q2_g_dly};
dat_q1_h[6:0] <= {dat_q1_h[5:0], q1_h};
dat_q2_h[6:0] <= {dat_q2_h[5:0], q2_h_dly};
end
//start new fco stuff

//generate new FCO
always @ (posedge dco)
begin
fco1 <= fco;
fco2 <= fco1;
fcostb <= fco1 & ~fco2;
end

always @(posedge dco)
if (fcostb)
begin
dout_a <=  {
```

```
dat_q2_a[6],dat_q1_a[6],
dat_q2_a[5],dat_q1_a[5],
dat_q2_a[4],dat_q1_a[4],
dat_q2_a[3],dat_q1_a[3],
dat_q2_a[2],dat_q1_a[2],
dat_q2_a[1],dat_q1_a[1],
dat_q2_a[0],dat_q1_a[0]
};
end

always @(posedge dco)
if (fcostb)
begin
dout_b <= {
dat_q2_b[6], dat_q1_b[6],
dat_q2_b[5], dat_q1_b[5],
dat_q2_b[4], dat_q1_b[4],
dat_q2_b[3], dat_q1_b[3],
dat_q2_b[2], dat_q1_b[2],
dat_q2_b[1], dat_q1_b[1],
dat_q2_b[0], dat_q1_b[0]
};
end

always @(posedge dco)
if (fcostb)
begin
dout_c <= {
dat_q2_c[6], dat_q1_c[6],
dat_q2_c[5], dat_q1_c[5],
dat_q2_c[4], dat_q1_c[4],
dat_q2_c[3], dat_q1_c[3],
dat_q2_c[2], dat_q1_c[2],
dat_q2_c[1], dat_q1_c[1],
dat_q2_c[0], dat_q1_c[0]
};
end

always @(posedge dco)
if (fcostb)
begin
dout_d <= {
dat_q2_d[6], dat_q1_d[6],
```

```
dat_q2_d[5], dat_q1_d[5],
dat_q2_d[4], dat_q1_d[4],
dat_q2_d[3], dat_q1_d[3],
dat_q2_d[2], dat_q1_d[2],
dat_q2_d[1], dat_q1_d[1],
dat_q2_d[0], dat_q1_d[0]
};
end

always @(posedge dco)
if (fcostb)
begin
dout_e <= {
dat_q2_e[6], dat_q1_e[6],
dat_q2_e[5], dat_q1_e[5],
dat_q2_e[4], dat_q1_e[4],
dat_q2_e[3], dat_q1_e[3],
dat_q2_e[2], dat_q1_e[2],
dat_q2_e[1], dat_q1_e[1],
dat_q2_e[0], dat_q1_e[0]
};
end

always @(posedge dco)
if (fcostb)
begin
dout_f <= {
dat_q2_f[6], dat_q1_f[6],
dat_q2_f[5], dat_q1_f[5],
dat_q2_f[4], dat_q1_f[4],
dat_q2_f[3], dat_q1_f[3],
dat_q2_f[2], dat_q1_f[2],
dat_q2_f[1], dat_q1_f[1],
dat_q2_f[0], dat_q1_f[0]
};
end

always @(posedge dco)
if (fcostb)
begin
dout_g <= {
dat_q2_g[6], dat_q1_g[6],
dat_q2_g[5], dat_q1_g[5],
```

```
dat_q2_g[4], dat_q1_g[4],
dat_q2_g[3], dat_q1_g[3],
dat_q2_g[2], dat_q1_g[2],
dat_q2_g[1], dat_q1_g[1],
dat_q2_g[0], dat_q1_g[0]
};
end

always @(posedge dco)
if (fcostb)
begin
dout_h <= {
dat_q2_h[6], dat_q1_h[6],
dat_q2_h[5], dat_q1_h[5],
dat_q2_h[4], dat_q1_h[4],
dat_q2_h[3], dat_q1_h[3],
dat_q2_h[2], dat_q1_h[2],
dat_q2_h[1], dat_q1_h[1],
dat_q2_h[0], dat_q1_h[0]
};
end


always @(posedge dco)
if (fcostb)
begin
doutA <= dout_a;
doutB <= dout_b;
doutC <= dout_c;
doutD <= dout_d;
doutE <= dout_e;
doutF <= dout_f;
doutG <= dout_g;
doutH <= dout_h;
end



assign clkout = fco;

always @(posedge dco)
begin
if (rst)
```

```
begin
count <= 1'b0;
gate <= 1'b0;
end
else if (fcostb)
begin
count <= count + 16'b1;
if (&count)
gate <= 1;
end
end
always @(posedge dco)

if (gate & fcostb)
begin
load <= wren;
end

endmodule
```

3 Ultrasound_FSM.v

```
    module Ultrasound_FSM
#(
parameter ADC_SAMPLES_PER_ECHO_WIDTH = 32,
parameter ADC_INIT_DELAY_WIDTH = 32
)
(
// Control Signals
input START,   //Starting of TX Firing and Data Acquisiton
output reg DONE,//Notification of Completed Data Acquistion
output reg TX_EN,    //TX_EN Signal

// timing parameters
input [31:0] ADC_START_length,

// ACQ WINDOW
input [ADC_INIT_DELAY_WIDTH-1:0]  ADC_INIT_DELAY,
input [ADC_SAMPLES_PER_ECHO_WIDTH-1:0] ADC_SAMPLES_PER_ECHO,
output  FIFO_EN,

// System Signals
```

```
input CLK,
input RESET
);
//
// State Machine Variable Definition
//
reg [8:0] State;
localparam [8:0]
S0 = 8'b000000001,   //State Machine Initialization
S1 = 8'b000000010,   //State Machine Counter Zeroing
S2 = 8'b000000100,//State Machine TX_EN & RX_EN High
S3 = 8'b000001000,   //State Machine TX_EN & RX_EN Low
S4 = 8'b000010000,
S5 = 8'b000100000,
S6 = 8'b001000000,
S7 = 8'b010000000,
S8 = 8'b100000000;

reg [31:0] counter;
reg ADC_START;
wire ACQ_DONE;


always @(posedge CLK, posedge RESET)
begin
if (RESET)
begin
// Reset Register Assignments
DONE <= 1'b0;
ADC_START <= 1'b0;
// Reset State Variable
State <= S0;
end
else
begin
// State Machine Operations (Actions / Transitions)
case (State)

S0 : // State 0
begin
// Clear the Done status
DONE <= 1'b1;
TX_EN <= 1'b0;
```

```
ADC_START <= 1'b0;

// Wait for Start
if (START)
State <= S1;
end

S1 : // State 1
begin
DONE <= 1'b0;
ADC_START <= 1'b1;
TX_EN <= 1'b1;
counter <= {1'b1,31'h0} - ADC_START_length + 2'd2;

State <= S2;
end

S2 : // reset the counter to count ADC_START length
begin
counter <= counter + 1'b1;

if (counter[31])
State <= S3;

end

S3 : // wait until ADC stops
begin
ADC_START <= 1'b0;

if (ACQ_DONE)
State <= S4;
end

S4 : //
begin
TX_EN <= 1'b0;
DONE <= 1'b1;
State <= S0;
end
```

```
endcase
end
end

ADC_ACQ_WINGEN
# (
.SAMPLES_PER_ECHO_WIDTH (ADC_SAMPLES_PER_ECHO_WIDTH),
.ADC_INIT_DELAY_WIDTH (ADC_INIT_DELAY_WIDTH)
)
ADC_ACQ_WINGEN1
(

// parameters
.ADC_INIT_DELAY  (ADC_INIT_DELAY-1'b1), // minus 1 factor because of
                                        // the delay from the FSM to
                                        // this module
.SAMPLES_PER_ECHO  (ADC_SAMPLES_PER_ECHO),

// control signal
.ACQ_WND  (ADC_START),
.ACQ_EN  (FIFO_EN),
.DONE (ACQ_DONE),

// system signal
.CLK  (CLK),
.RESET  (RESET)
);


endmodule
```

Appendix C
C code for System Prototype

This section provides the C code mentioned that was used for the current operation of the system.

1 hps_linux.c

```
#include <assert.h>
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include <time.h>

#include "alt_generalpurpose_io.h"
#include <hwlib.h>
#include <socal/alt_gpio.h>
#include <socal/hps.h>
#include <socal/socal.h>
#include "functions/avalon_spi.h"

#include "hps_linux.h"
#include "./soc_variables/soc_system.h"
#include "./soc_variables/general.h"
#include "./soc_variables/lm96570_vars.h"
#include "./soc_variables/ad9276_vars.h"
#include "functions/AlteraIP/altera_avalon_fifo_regs.h"

// parameters
unsigned int num_of_samples = 8000;
const unsigned int num_of_switches = 11;
const unsigned int num_of_channels = 8;

extern int fd_dev_mem;
```

```
unsigned int cnt_out_val;

void create_measurement_folder(char * foldertype) {
time_t t = time(NULL);
struct tm tm = *localtime(&t);
char command[60];
sprintf(foldername,"%s_%04d_%02d_%02d_%02d_%02d_%02d"
,foldertype,tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday, tm.tm_hour,
tm.tm_min, tm.tm_sec);
sprintf(command,"mkdir %s",foldername);
system(command);

// copy the executable file to the folder
sprintf(command,"cp ./thesis_nmr_de1soc_hdl2.0 %s/execfile",foldername);
system(command);
}

void init() {
//printf("ULTRASOUND SYSTEM STARTS!\n");

// open device memory
    fd_dev_mem = open("/dev/mem", O_RDWR | O_SYNC);
    if(fd_dev_mem  == -1) {
        printf("ERROR: could not open \"/dev/mem\".\n");
        printf("    errno = %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    // mmap hps peripherals
    hps_gpio = mmap(NULL, hps_gpio_span, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_dev_mem, hps_gpio_ofst);
if (hps_gpio == MAP_FAILED) {
printf("Error: hps_gpio mmap() failed.\n");
printf("    errno = %s\n", strerror(errno));
close(fd_dev_mem);
exit(EXIT_FAILURE);
}

// mmap fpga peripherals
h2f_lw_axi_master = mmap(NULL, h2f_lw_axi_master_span, PROT_READ |
PROT_WRITE, MAP_SHARED, fd_dev_mem, h2f_lw_axi_master_ofst);
if (h2f_lw_axi_master == MAP_FAILED) {
printf("Error: h2f_lw_axi_master mmap() failed.\n");
```

```
printf("    errno = %s\n", strerror(errno));
close(fd_dev_mem);
exit(EXIT_FAILURE);
}

//h2p_fifo_sink_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_OUT_CSR_BASE;
//h2p_fifo_sink_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_OUT_BASE;
h2p_fifo_sink_ch_a_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_A_OUT_CSR_BASE;
h2p_fifo_sink_ch_a_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_A_OUT_BASE;
h2p_fifo_sink_ch_b_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_B_OUT_CSR_BASE;
h2p_fifo_sink_ch_b_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_B_OUT_BASE;
h2p_fifo_sink_ch_c_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_C_OUT_CSR_BASE;
h2p_fifo_sink_ch_c_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_C_OUT_BASE;
h2p_fifo_sink_ch_d_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_D_OUT_CSR_BASE;
h2p_fifo_sink_ch_d_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_D_OUT_BASE;
h2p_fifo_sink_ch_e_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_E_OUT_CSR_BASE;
h2p_fifo_sink_ch_e_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_E_OUT_BASE;
h2p_fifo_sink_ch_f_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_F_OUT_CSR_BASE;
h2p_fifo_sink_ch_f_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_F_OUT_BASE;
h2p_fifo_sink_ch_g_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_G_OUT_CSR_BASE;
h2p_fifo_sink_ch_g_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_G_OUT_BASE;
h2p_fifo_sink_ch_h_csr_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_H_OUT_CSR_BASE;
h2p_fifo_sink_ch_h_data_addr = h2f_lw_axi_master +
                                    FIFO_SINK_CH_H_OUT_BASE;
h2p_led_addr = h2f_lw_axi_master + LED_PIO_BASE;
h2p_sw_addr = h2f_lw_axi_master + DIPSW_PIO_BASE;
```

```
h2p_button_addr = h2f_lw_axi_master + BUTTON_PIO_BASE;
h2p_adcspi_addr = h2f_lw_axi_master + AD9276_SPI_BASE;
h2p_adc_samples_per_echo_addr = h2f_lw_axi_master +
                                    ADC_SAMPLES_PER_ECHO_BASE;
h2p_init_delay_addr = h2f_lw_axi_master +
                                    ADC_INIT_DELAY_BASE;
h2p_spi_num_of_bits_addr = h2f_lw_axi_master +
                                    LM96570_SPI_NUM_OF_BITS_BASE;
h2p_general_cnt_int_addr = h2f_lw_axi_master +
                                    GENERAL_CNT_IN_BASE;
h2p_general_cnt_out_addr = h2f_lw_axi_master +
                                    GENERAL_CNT_OUT_BASE;
h2p_lm96570_spi_out2_addr = h2f_lw_axi_master +
                                    LM96570_SPI_OUT_2_BASE;
h2p_lm96570_spi_out1_addr = h2f_lw_axi_master +
                                    LM96570_SPI_OUT_1_BASE;
h2p_lm96570_spi_out0_addr = h2f_lw_axi_master +
                                    LM96570_SPI_OUT_0_BASE;
h2p_lm96570_spi_in2_addr = h2f_lw_axi_master +
                                    LM96570_SPI_IN_2_BASE;
h2p_lm96570_spi_in1_addr = h2f_lw_axi_master +
                                    LM96570_SPI_IN_1_BASE;
h2p_lm96570_spi_in0_addr = h2f_lw_axi_master +
                                    LM96570_SPI_IN_0_BASE;
h2p_adc_start_pulselength_addr = h2f_lw_axi_master +
                                    ADC_START_PULSELENGTH_BASE;
h2p_mux_control_addr = h2f_lw_axi_master + MUX_CONTROL_BASE;


// write default value for cnt_out
cnt_out_val = CNT_OUT_DEFAULT;
alt_write_word( (h2p_general_cnt_out_addr) ,  cnt_out_val);

}

void leave() {
    close(fd_dev_mem);

    // munmap hps peripherals
    if (munmap(hps_gpio, hps_gpio_span) != 0) {
            printf("Error: hps_gpio munmap() failed\n");
            printf("    errno = %s\n", strerror(errno));
            close(fd_dev_mem);
```

```
            exit(EXIT_FAILURE);
        }

        hps_gpio = NULL;


if (munmap(h2f_lw_axi_master, h2f_lw_axi_master_span) != 0) {
printf("Error: h2f_lw_axi_master munmap() failed\n");
printf("    errno = %s\n", strerror(errno));
close(fd_dev_mem);
exit(EXIT_FAILURE);
}

h2f_lw_axi_master = NULL;
h2p_led_addr = NULL;
h2p_sw_addr = NULL;
h2p_fifo_sink_ch_a_csr_addr = NULL;
h2p_fifo_sink_ch_a_data_addr = NULL;
h2p_fifo_sink_ch_b_csr_addr = NULL;
h2p_fifo_sink_ch_b_data_addr = NULL;
h2p_fifo_sink_ch_c_csr_addr = NULL;
h2p_fifo_sink_ch_d_data_addr = NULL;
h2p_fifo_sink_ch_d_csr_addr = NULL;
h2p_fifo_sink_ch_d_data_addr = NULL;
h2p_fifo_sink_ch_e_csr_addr = NULL;
h2p_fifo_sink_ch_e_data_addr = NULL;
h2p_fifo_sink_ch_f_csr_addr = NULL;
h2p_fifo_sink_ch_f_data_addr = NULL;
h2p_fifo_sink_ch_g_csr_addr = NULL;
h2p_fifo_sink_ch_g_data_addr = NULL;
h2p_fifo_sink_ch_h_csr_addr = NULL;
h2p_fifo_sink_ch_h_data_addr = NULL;
h2p_led_addr = NULL;
h2p_sw_addr = NULL;
h2p_button_addr = NULL;
h2p_adcspi_addr = NULL;
h2p_adc_samples_per_echo_addr = NULL;
h2p_init_delay_addr = NULL;
h2p_spi_num_of_bits_addr = NULL;
h2p_general_cnt_int_addr = NULL;
h2p_general_cnt_out_addr = NULL;
h2p_lm96570_spi_out2_addr = NULL;
h2p_lm96570_spi_out1_addr = NULL;
```

```
h2p_lm96570_spi_out0_addr = NULL;
h2p_lm96570_spi_in2_addr = NULL;
h2p_lm96570_spi_in1_addr = NULL;
h2p_lm96570_spi_in0_addr = NULL;
h2p_mux_control_addr = NULL;

//printf("\nULTRASOUND SYSTEM STOPS!\n");
}

unsigned int write_adc_spi (unsigned int comm) {
unsigned int data;

while (! (alt_read_word(h2p_adcspi_addr + SPI_STATUS_offst)
& (1<<status_TRDY_bit)));
alt_write_word( (h2p_adcspi_addr + SPI_TXDATA_offst) ,  comm);
while (! (alt_read_word(h2p_adcspi_addr + SPI_STATUS_offst)
& (1<<status_TMT_bit)));
data = alt_read_word(h2p_adcspi_addr + SPI_RXDATA_offst);
// wait for the spi command to finish
return (data);
}

void write_beamformer_spi (unsigned char spi_reg_length,
unsigned char read, unsigned char spi_addr, unsigned long spi_data_out,
unsigned int *spi_in0, unsigned int *spi_in1, unsigned int *spi_in2) {
unsigned int spi_out0, spi_out1, spi_out2;
spi_out0 = 0;
spi_out1 = 0;
spi_out2 = 0;

spi_out0 = (spi_addr & 0x1F) | ((read & 0x01)<<5) |
( (spi_data_out & 0x3FFFFFF)
<<6);
spi_out1 = (spi_data_out>>26) & 0xFFFFFFFF;
//spi_out2 = (unsigned int)(spi_data_out>>58) &
(unsigned int)0x3F; WARNING!

alt_write_word( h2p_lm96570_spi_in0_addr ,  spi_out0);
alt_write_word( h2p_lm96570_spi_in1_addr ,  spi_out1);
alt_write_word( h2p_lm96570_spi_in2_addr ,  spi_out2);
alt_write_word( h2p_spi_num_of_bits_addr ,  spi_reg_length + 6);
// 6 is the command length
```

```
cnt_out_val |= lm96570_start_MSK;
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// start the beamformer SPI
usleep(100);
cnt_out_val &= (~lm96570_start_MSK);
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// stop the beamformer SPI

*spi_in0 = alt_read_word(h2p_lm96570_spi_out0_addr);
*spi_in1 = alt_read_word(h2p_lm96570_spi_out1_addr);
*spi_in2 = alt_read_word(h2p_lm96570_spi_out2_addr);

//printf("beamformer_spi_in = 0x%04x_%04x_%04x\n",
*spi_in2, *spi_in1, *spi_in0);

}

void read_adc_id () {
unsigned int command, data;
unsigned int comm, addr;

comm = (1<<7) | (0<<6) | (0<<5); // read chip settings
addr = 0x00;
command = (comm<<16) | (addr<<8) | 0x00;
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

comm = (1<<7) | (0<<6) | (0<<5); // read chip ID
addr = 0x01;
command = (comm<<16) | (addr<<8) | 0x00;
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

comm = (1<<7) | (0<<6) | (0<<5); // read chip ID
addr = 0x02;
command = (comm<<16) | (addr<<8) | 0x00;
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

}

void init_adc() {
unsigned int command, data;
```

```
unsigned int comm, addr;

comm = (AD9276_SPI_WR<<7) | AD9276_1BYTE_DATA;

//addr = 0x04;
//command = (comm<<16) | (addr<<8) | 0x0F; // select data channel E-H
//data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

//addr = 0x05;
//command = (comm<<16) | (addr<<8) | 0x0F; // select data channel A-D
//data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0x11;
command = (comm<<16) | (addr<<8) | (0x0E); // set PGA Gain to 21 dB,
LNA Gain to 15.6 dB
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0xFF;
command = (comm<<16) | (addr<<8) | (0x01); // update the device
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0x15;
command = (comm<<16) | (addr<<8) | (0x02<<4); // set output driver
                                              // to 100 ohms
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0xFF;
command = (comm<<16) | (addr<<8) | (0x01); // update the device
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0x16;
command = (comm<<16) | (addr<<8) | (0x08); // set output phase
                                           // to 480 degrees
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0xFF;
```

```
command = (comm<<16) | (addr<<8) | (0x01); // update the device
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

// read register
comm = (AD9276_SPI_WR<<7) | AD9276_1BYTE_DATA;

addr = 0x05;
command = (comm<<16) | (addr<<8) | 0x01; // select data channel A
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

comm = (AD9276_SPI_RD<<7) | AD9276_1BYTE_DATA;

addr = 0x15;
command = (comm<<16) | (addr<<8) | 0x00; // read channel A
                                         // termination status
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0x05;
command = (comm<<16) | (addr<<8) | 0x01; // select data channel A
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

comm = (AD9276_SPI_RD<<7) | AD9276_1BYTE_DATA;

addr = 0x11;
command = (comm<<16) | (addr<<8) | 0x00; // read channel A Gain status
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

addr = 0x05;
command = (comm<<16) | (addr<<8) | 0x01; // select data channel A
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);

comm = (AD9276_SPI_RD<<7) | AD9276_1BYTE_DATA;

addr = 0x16;
command = (comm<<16) | (addr<<8) | 0x00; // read channel A Phase status
data = write_adc_spi (command);
//printf("command = 0x%06x -> datain = 0x%06x\n", command, data);
```

```
}

void init_beamformer() {
unsigned int spi_in0, spi_in1, spi_in2;
spi_in0 = 0x00;
spi_in1 = 0x00;
spi_in2 = 0x00;

write_beamformer_spi(REG_1A_LENGTH,  LM86570_SPI_WR, 0x1A, 0x4040,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x4600, Last Zero needed
                               //for padding.

write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x00, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x0000, D2:0x0000, D3:0x0000, D4:0x02C7, D5:0x057A
write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x01, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x0103, D2:0x00A4, D3:0x003F, D4:0x02A0, D5:0x04EE
write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x02, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x01F4, D2:0x0133, D3:0x006A, D4:0x0265, D5:0x044E
write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x03, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x02D1, D2:0x01AE, D3:0x007F, D4:0x0214, D5:0x0399
write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x04, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x0399, D2:0x0214, D3:0x007F, D4:0x01AE, D5:0x02D1
write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x05, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x044E, D2:0x0265, D3:0x006A, D4:0x0133, D5:0x01F4
write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x06, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x04EE, D2:0x02A0, D3:0x003F, D4:0x00A4, D5:0x0103
write_beamformer_spi(REG_00_07_LENGTH,  LM86570_SPI_WR, 0x07, 0x0000,
&spi_in0, &spi_in1, &spi_in2);
//D0:0x0000, D1:0x057A, D2:0x02C7, D3:0x0000, D4:0x0000, D5:0x0000


//write_beamformer_spi(4,  LM86570_SPI_WR, 0x08, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x09, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005
```

```
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x0A, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x0B, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x0C, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x0D, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x0E, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x0F, 0x0005,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0005

//write_beamformer_spi(4,  LM86570_SPI_WR, 0x10, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x11, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x12, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x13, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x14, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x15, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x16, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A
//write_beamformer_spi(4,  LM86570_SPI_WR, 0x17, 0x000A,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x000A


write_beamformer_spi(4,  LM86570_SPI_WR, 0x18, 0x0005,
&spi_in0, &spi_in1, &spi_in2); // write all p registers,
                               // Default: 0x0005, 4
write_beamformer_spi(4,  LM86570_SPI_WR, 0x19, 0x000A,
&spi_in0, &spi_in1, &spi_in2); // write all n registers,
                               // Default: 0x000A, 4

write_beamformer_spi(4,  REG_1BA_LENGTH, 0x1B, 0x0000,
&spi_in0, &spi_in1, &spi_in2); //Default: 0x0000

}
```

```
void read_adc_val (void *channel_csr_addr, void *channel_data_addr,
unsigned int * adc_data) {//, char *filename) {
unsigned int fifo_mem_level;
//fptr = fopen(filename, "w");
//if (fptr == NULL) {
// printf("File does not exists \n");
// return;
//}

// PRINT # of DATAS in FIFO
fifo_mem_level =
alt_read_word(channel_csr_addr+ALTERA_AVALON_FIFO_LEVEL_REG);
// the fill level of FIFO memory
//printf("num of data in fifo: %d\n",fifo_mem_level);
//

// READING DATA FROM FIFO
fifo_mem_level =
alt_read_word(channel_csr_addr+ALTERA_AVALON_FIFO_LEVEL_REG);
// the fill level of FIFO memory
for (i=0; fifo_mem_level>0; i++) {
adc_data[i] = alt_read_word(channel_data_addr);

//fprintf(fptr, "%d\n", rddata[i] & 0xFFF);
//fprintf(fptr, "%d\n", (rddata[i]>>16) & 0xFFF);

fifo_mem_level--;
if (fifo_mem_level == 0) {
fifo_mem_level =
alt_read_word(channel_csr_addr+ALTERA_AVALON_FIFO_LEVEL_REG);
}
}
usleep(10);

fifo_mem_level =
alt_read_word(channel_csr_addr+ALTERA_AVALON_FIFO_LEVEL_REG);
// the fill level of FIFO memory
//printf("num of data in fifo: %d\n",fifo_mem_level);

//fclose(fptr);

}
```

```
void store_data (unsigned int * adc_data, unsigned int
data_bank[num_of_switches][num_of_channels][num_of_samples],
unsigned int sw_num,unsigned int ch_num, unsigned int num_of_samples) {
unsigned int ii;
for (ii=0; ii<num_of_samples/2; ii++) {
data_bank[sw_num][ch_num][ii*2] = adc_data[ii] & 0xFFF;
data_bank[sw_num][ch_num][ii*2+1] = (adc_data[ii]>>16) & 0xFFF;
}
}

void write_data_bank (unsigned int data_bank
[num_of_switches][num_of_channels][num_of_samples]) {
unsigned int ii, jj, kk;

fptr = fopen("databank.txt", "w");
if (fptr == NULL) {
printf("File does not exists \n");
return;
}

for (ii=0; ii<num_of_switches; ii++) {
for (jj=0; jj<num_of_channels; jj++) {
for (kk=0; kk<num_of_samples; kk++) {
fprintf(fptr, "%d ", data_bank[ii][jj][kk] & 0xFFF);
}
fprintf(fptr, "\n");
}
//fprintf(fptr, "\n");
}
}

void print_data_bank (unsigned int data_bank
[num_of_switches][num_of_channels][num_of_samples]) {
unsigned int ii, jj, kk;

/*fptr = fopen("databank.txt", "w");
if (fptr == NULL) {
printf("File does not exists \n");
return;
}*/

for (ii=0; ii<num_of_switches; ii++) {
for (jj=0; jj<num_of_channels; jj++) {
```

```
for (kk=0; kk<num_of_samples; kk++) {
printf("%d ", data_bank[ii][jj][kk] & 0xFFF);
}
//fprintf(fptr, "\n");
}
//fprintf(fptr, "\n");
}
}

int main (){

// Initialize system
    init();
    read_adc_id();

    init_beamformer();
    init_adc();



    alt_write_word( h2p_adc_start_pulselength_addr , 10 );
    alt_write_word( h2p_adc_samples_per_echo_addr ,  num_of_samples);

    unsigned int data_bank[num_of_switches][num_of_channels][num_of_samples];
    unsigned int adc_data [num_of_samples]; // data for 1 acquisition
    unsigned int sw_num = 0;
for (sw_num=0; sw_num < num_of_switches; sw_num++) {

//Reset FSM in order to address glitching
cnt_out_val |= FSM_RST_MSK;
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// restart the Ultrasound FSM
usleep(1000);
cnt_out_val &= (~FSM_RST_MSK);
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// set the Ultrasound FSM
usleep(1000);


//write mux
alt_write_word(h2p_mux_control_addr , ((0x00) & 0x7FF));
usleep(500);
alt_write_word(h2p_mux_control_addr , ((1<<sw_num) & 0x7FF));
```

```
init_beamformer();
usleep(500);


// tx_path on
//cnt_out_val |= tx_path_en_MSK;
//alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// start the beamformer SPI
//usleep(100000);

// sw_off on
//cnt_out_val |= sw_off_MSK;
//alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// start the beamformer SPI
//usleep(100000);

// pulser on
cnt_out_val |= pulser_en_MSK;
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// start the beamformer SPI
usleep(1000);

// tx_enable fire
cnt_out_val |= lm96570_tx_en_MSK;
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// start the beamformer SPI
usleep(1000);
cnt_out_val &= (~lm96570_tx_en_MSK);
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// stop the beamformer SPI

// pulser off
//usleep(200000);
cnt_out_val &= (~pulser_en_MSK);
alt_write_word( h2p_general_cnt_out_addr ,  cnt_out_val);
// stop the beamformer SPI

read_adc_val(h2p_fifo_sink_ch_a_csr_addr, h2p_fifo_sink_ch_a_data_addr,
adc_data);
store_data (adc_data, data_bank, sw_num, 0, num_of_samples);

read_adc_val(h2p_fifo_sink_ch_b_csr_addr, h2p_fifo_sink_ch_b_data_addr,
adc_data);
```

```
store_data (adc_data, data_bank, sw_num, 1, num_of_samples);

read_adc_val(h2p_fifo_sink_ch_c_csr_addr, h2p_fifo_sink_ch_c_data_addr,
adc_data);
store_data (adc_data, data_bank, sw_num, 2, num_of_samples);

read_adc_val(h2p_fifo_sink_ch_d_csr_addr, h2p_fifo_sink_ch_d_data_addr,
adc_data);
store_data (adc_data, data_bank, sw_num, 3, num_of_samples);

read_adc_val(h2p_fifo_sink_ch_e_csr_addr, h2p_fifo_sink_ch_e_data_addr,
adc_data);
store_data (adc_data, data_bank, sw_num, 4, num_of_samples);

read_adc_val(h2p_fifo_sink_ch_f_csr_addr, h2p_fifo_sink_ch_f_data_addr,
adc_data);
store_data (adc_data, data_bank, sw_num, 5, num_of_samples);

read_adc_val(h2p_fifo_sink_ch_g_csr_addr, h2p_fifo_sink_ch_g_data_addr,
adc_data);
store_data (adc_data, data_bank, sw_num, 6, num_of_samples);

read_adc_val(h2p_fifo_sink_ch_h_csr_addr, h2p_fifo_sink_ch_h_data_addr,
adc_data);
store_data (adc_data, data_bank, sw_num, 7, num_of_samples);

//printf("Completed Event: %d\n",sw_num);

}

write_data_bank(data_bank);
print_data_bank(data_bank);
    // exit program

    leave();
    //printf("%s \n",data_bank);

    return 0;
}
```

2 ad9276_vars.h

```
#define AD9276_SPI_RD 1
#define AD9276_SPI_WR 0

#define AD9276_1BYTE_DATA (0x00<<5)
#define AD9276_2BYTE_DATA (0x01<<5)
#define AD9276_3BYTE_DATA (0x02<<5)
```

3 general.h

```
// offsets for output control signal
#define FSM_RST_OFST                (7)
#define sw_off_OFST (6)
#define tx_path_en_OFST (5)
#define pulser_en_OFST (4)
#define lm96570_pin_reset_OFST (3)
#define lm96570_tx_en_OFST (2)
#define lm96570_spi_reset_OFST (1)
#define lm96570_start_OFST (0)

// mask for output control signal
#define FSM_RST_MSK (1<<FSM_RST_OFST )
#define sw_off_MSK (1<<sw_off_OFST )
#define tx_path_en_MSK (1<<tx_path_en_OFST )
#define pulser_en_MSK (1<<pulser_en_OFST )
#define lm96570_pin_reset_MSK (1<<lm96570_pin_reset_OFST )
#define lm96570_tx_en_MSK (1<<lm96570_tx_en_OFST )
#define lm96570_spi_reset_MSK (1<<lm96570_spi_reset_OFST )
#define lm96570_start_MSK (1<<lm96570_start_OFST )

// default for output control signal
#define CNT_OUT_DEFAULT 0b000000
```

4 lm96570_vars.h

```
#define LM86570_SPI_RD 1
#define LM86570_SPI_WR 0

#define REG_1A_LENGTH  14
#define REG_00_07_LENGTH  22
```

```
#define REG_1BA_LENGTH  8
```

5 soc_system.h

```
#ifndef _ALTERA_HPS_SOC_SYSTEM_H_
#define _ALTERA_HPS_SOC_SYSTEM_H_

/*
 * This file was automatically generated by the swinfo2header utility.
 *
 * Created from SOPC Builder system 'soc_system' in
 * file '../SoC/soc_system.sopcinfo'.
 */

/*
 * This file contains macros for module 'hps_0' and devices
 * connected to the following masters:
 *   h2f_axi_master
 *   h2f_lw_axi_master
 *
 * Do not include this header file and another header file created for a
 * different module or master group at the same time.
 * Doing so may result in duplicate macro names.
 * Instead, use the system header file which has macros with unique names.
 */

/*
 * Macros for device 'ILC', class 'interrupt_latency_counter'
 * The macros are prefixed with 'ILC_'.
 * The prefix is the slave descriptor.
 */
#define ILC_COMPONENT_TYPE interrupt_latency_counter
#define ILC_COMPONENT_NAME ILC
#define ILC_BASE 0x0
#define ILC_SPAN 256
#define ILC_END 0xff

/*
 * Macros for device 'fifo_sink_CH_A_in_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_A_IN_CSR_'.
 * The prefix is the slave descriptor.
 */
```

```
#define FIFO_SINK_CH_A_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_A_IN_CSR_COMPONENT_NAME fifo_sink_CH_A
#define FIFO_SINK_CH_A_IN_CSR_BASE 0x100
#define FIFO_SINK_CH_A_IN_CSR_SPAN 32
#define FIFO_SINK_CH_A_IN_CSR_END 0x11f
#define FIFO_SINK_CH_A_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_A_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_A_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_A_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_A_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_A_IN_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_A_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_A_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_A_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_A_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_A_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_A_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_A_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_A_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_A_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_A_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_A_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_A_IN_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_A_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_A_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_A_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_A_OUT_CSR_COMPONENT_NAME fifo_sink_CH_A
#define FIFO_SINK_CH_A_OUT_CSR_BASE 0x120
#define FIFO_SINK_CH_A_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_A_OUT_CSR_END 0x13f
#define FIFO_SINK_CH_A_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_A_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_A_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_A_OUT_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_A_OUT_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_A_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_A_OUT_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_A_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_A_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
```

```
#define FIFO_SINK_CH_A_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_A_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_A_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_A_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_A_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_A_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_A_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_A_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_A_OUT_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'ad9276_spi', class 'altera_avalon_spi'
 * The macros are prefixed with 'AD9276_SPI_'.
 * The prefix is the slave descriptor.
 */
#define AD9276_SPI_COMPONENT_TYPE altera_avalon_spi
#define AD9276_SPI_COMPONENT_NAME ad9276_spi
#define AD9276_SPI_BASE 0x140
#define AD9276_SPI_SPAN 32
#define AD9276_SPI_END 0x15f
#define AD9276_SPI_IRQ 0
#define AD9276_SPI_CLOCKMULT 1
#define AD9276_SPI_CLOCKPHASE 0
#define AD9276_SPI_CLOCKPOLARITY 0
#define AD9276_SPI_CLOCKUNITS "Hz"
#define AD9276_SPI_DATABITS 24
#define AD9276_SPI_DATAWIDTH 32
#define AD9276_SPI_DELAYMULT "1.0E-9"
#define AD9276_SPI_DELAYUNITS "ns"
#define AD9276_SPI_EXTRADELAY 0
#define AD9276_SPI_INSERT_SYNC 0
#define AD9276_SPI_ISMASTER 1
#define AD9276_SPI_LSBFIRST 0
#define AD9276_SPI_NUMSLAVES 1
#define AD9276_SPI_PREFIX "spi_"
#define AD9276_SPI_SYNC_REG_DEPTH 2
#define AD9276_SPI_TARGETCLOCK 4200000
#define AD9276_SPI_TARGETSSDELAY "2000.0"

/*
 * Macros for device 'fifo_sink_CH_H_in_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_H_IN_CSR_'.
 * The prefix is the slave descriptor.
```

```
 */
#define FIFO_SINK_CH_H_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_H_IN_CSR_COMPONENT_NAME fifo_sink_CH_H
#define FIFO_SINK_CH_H_IN_CSR_BASE 0x160
#define FIFO_SINK_CH_H_IN_CSR_SPAN 32
#define FIFO_SINK_CH_H_IN_CSR_END 0x17f
#define FIFO_SINK_CH_H_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_H_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_H_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_H_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_H_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_H_IN_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_H_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_H_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_H_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_H_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_H_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_H_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_H_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_H_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_H_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_H_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_H_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_H_IN_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'mux_control', class 'altera_avalon_pio'
 * The macros are prefixed with 'MUX_CONTROL_'.
 * The prefix is the slave descriptor.
 */
#define MUX_CONTROL_COMPONENT_TYPE altera_avalon_pio
#define MUX_CONTROL_COMPONENT_NAME mux_control
#define MUX_CONTROL_BASE 0x180
#define MUX_CONTROL_SPAN 16
#define MUX_CONTROL_END 0x18f
#define MUX_CONTROL_BIT_CLEARING_EDGE_REGISTER 0
#define MUX_CONTROL_BIT_MODIFYING_OUTPUT_REGISTER 0
#define MUX_CONTROL_CAPTURE 0
#define MUX_CONTROL_DATA_WIDTH 11
#define MUX_CONTROL_DO_TEST_BENCH_WIRING 0
#define MUX_CONTROL_DRIVEN_SIM_VALUE 0
#define MUX_CONTROL_EDGE_TYPE NONE
#define MUX_CONTROL_FREQ 50000000
```

```
#define MUX_CONTROL_HAS_IN 0
#define MUX_CONTROL_HAS_OUT 1
#define MUX_CONTROL_HAS_TRI 0
#define MUX_CONTROL_IRQ_TYPE NONE
#define MUX_CONTROL_RESET_VALUE 0

/*
 * Macros for device 'fifo_sink_CH_H_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_H_OUT_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_H_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_H_OUT_COMPONENT_NAME fifo_sink_CH_H
#define FIFO_SINK_CH_H_OUT_BASE 0x190
#define FIFO_SINK_CH_H_OUT_SPAN 8
#define FIFO_SINK_CH_H_OUT_END 0x197
#define FIFO_SINK_CH_H_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_H_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_H_OUT_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_H_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_H_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_H_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_H_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_H_OUT_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_H_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_H_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_H_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_H_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_H_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_H_OUT_USE_IRQ 0
#define FIFO_SINK_CH_H_OUT_USE_PACKET 0
#define FIFO_SINK_CH_H_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_H_OUT_USE_REGISTER 0
#define FIFO_SINK_CH_H_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_G_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_G_OUT_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_G_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_G_OUT_COMPONENT_NAME fifo_sink_CH_G
#define FIFO_SINK_CH_G_OUT_BASE 0x198
```

```
#define FIFO_SINK_CH_G_OUT_SPAN 8
#define FIFO_SINK_CH_G_OUT_END 0x19f
#define FIFO_SINK_CH_G_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_G_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_G_OUT_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_G_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_G_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_G_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_G_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_G_OUT_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_G_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_G_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_G_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_G_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_G_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_G_OUT_USE_IRQ 0
#define FIFO_SINK_CH_G_OUT_USE_PACKET 0
#define FIFO_SINK_CH_G_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_G_OUT_USE_REGISTER 0
#define FIFO_SINK_CH_G_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'ADC_START_pulselength', class 'altera_avalon_pio'
 * The macros are prefixed with 'ADC_START_PULSELENGTH_'.
 * The prefix is the slave descriptor.
 */
#define ADC_START_PULSELENGTH_COMPONENT_TYPE altera_avalon_pio
#define ADC_START_PULSELENGTH_COMPONENT_NAME ADC_START_pulselength
#define ADC_START_PULSELENGTH_BASE 0x1a0
#define ADC_START_PULSELENGTH_SPAN 16
#define ADC_START_PULSELENGTH_END 0x1af
#define ADC_START_PULSELENGTH_BIT_CLEARING_EDGE_REGISTER 0
#define ADC_START_PULSELENGTH_BIT_MODIFYING_OUTPUT_REGISTER 0
#define ADC_START_PULSELENGTH_CAPTURE 0
#define ADC_START_PULSELENGTH_DATA_WIDTH 32
#define ADC_START_PULSELENGTH_DO_TEST_BENCH_WIRING 0
#define ADC_START_PULSELENGTH_DRIVEN_SIM_VALUE 0
#define ADC_START_PULSELENGTH_EDGE_TYPE NONE
#define ADC_START_PULSELENGTH_FREQ 50000000
#define ADC_START_PULSELENGTH_HAS_IN 0
#define ADC_START_PULSELENGTH_HAS_OUT 1
#define ADC_START_PULSELENGTH_HAS_TRI 0
#define ADC_START_PULSELENGTH_IRQ_TYPE NONE
```

```
#define ADC_START_PULSELENGTH_RESET_VALUE 0

/*
 * Macros for device 'ADC_SAMPLES_PER_ECHO', class 'altera_avalon_pio'
 * The macros are prefixed with 'ADC_SAMPLES_PER_ECHO_'.
 * The prefix is the slave descriptor.
 */
#define ADC_SAMPLES_PER_ECHO_COMPONENT_TYPE altera_avalon_pio
#define ADC_SAMPLES_PER_ECHO_COMPONENT_NAME ADC_SAMPLES_PER_ECHO
#define ADC_SAMPLES_PER_ECHO_BASE 0x1b0
#define ADC_SAMPLES_PER_ECHO_SPAN 16
#define ADC_SAMPLES_PER_ECHO_END 0x1bf
#define ADC_SAMPLES_PER_ECHO_BIT_CLEARING_EDGE_REGISTER 0
#define ADC_SAMPLES_PER_ECHO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define ADC_SAMPLES_PER_ECHO_CAPTURE 0
#define ADC_SAMPLES_PER_ECHO_DATA_WIDTH 32
#define ADC_SAMPLES_PER_ECHO_DO_TEST_BENCH_WIRING 0
#define ADC_SAMPLES_PER_ECHO_DRIVEN_SIM_VALUE 0
#define ADC_SAMPLES_PER_ECHO_EDGE_TYPE NONE
#define ADC_SAMPLES_PER_ECHO_FREQ 50000000
#define ADC_SAMPLES_PER_ECHO_HAS_IN 0
#define ADC_SAMPLES_PER_ECHO_HAS_OUT 1
#define ADC_SAMPLES_PER_ECHO_HAS_TRI 0
#define ADC_SAMPLES_PER_ECHO_IRQ_TYPE NONE
#define ADC_SAMPLES_PER_ECHO_RESET_VALUE 0

/*
 * Macros for device 'ADC_INIT_DELAY', class 'altera_avalon_pio'
 * The macros are prefixed with 'ADC_INIT_DELAY_'.
 * The prefix is the slave descriptor.
 */
#define ADC_INIT_DELAY_COMPONENT_TYPE altera_avalon_pio
#define ADC_INIT_DELAY_COMPONENT_NAME ADC_INIT_DELAY
#define ADC_INIT_DELAY_BASE 0x1c0
#define ADC_INIT_DELAY_SPAN 16
#define ADC_INIT_DELAY_END 0x1cf
#define ADC_INIT_DELAY_BIT_CLEARING_EDGE_REGISTER 0
#define ADC_INIT_DELAY_BIT_MODIFYING_OUTPUT_REGISTER 0
#define ADC_INIT_DELAY_CAPTURE 0
#define ADC_INIT_DELAY_DATA_WIDTH 32
#define ADC_INIT_DELAY_DO_TEST_BENCH_WIRING 0
#define ADC_INIT_DELAY_DRIVEN_SIM_VALUE 0
#define ADC_INIT_DELAY_EDGE_TYPE NONE
```

```
#define ADC_INIT_DELAY_FREQ 50000000
#define ADC_INIT_DELAY_HAS_IN 0
#define ADC_INIT_DELAY_HAS_OUT 1
#define ADC_INIT_DELAY_HAS_TRI 0
#define ADC_INIT_DELAY_IRQ_TYPE NONE
#define ADC_INIT_DELAY_RESET_VALUE 0

/*
 * Macros for device 'lm96570_spi_num_of_bits', class 'altera_avalon_pio'
 * The macros are prefixed with 'LM96570_SPI_NUM_OF_BITS_'.
 * The prefix is the slave descriptor.
 */
#define LM96570_SPI_NUM_OF_BITS_COMPONENT_TYPE altera_avalon_pio
#define LM96570_SPI_NUM_OF_BITS_COMPONENT_NAME lm96570_spi_num_of_bits
#define LM96570_SPI_NUM_OF_BITS_BASE 0x1d0
#define LM96570_SPI_NUM_OF_BITS_SPAN 16
#define LM96570_SPI_NUM_OF_BITS_END 0x1df
#define LM96570_SPI_NUM_OF_BITS_BIT_CLEARING_EDGE_REGISTER 0
#define LM96570_SPI_NUM_OF_BITS_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LM96570_SPI_NUM_OF_BITS_CAPTURE 0
#define LM96570_SPI_NUM_OF_BITS_DATA_WIDTH 32
#define LM96570_SPI_NUM_OF_BITS_DO_TEST_BENCH_WIRING 0
#define LM96570_SPI_NUM_OF_BITS_DRIVEN_SIM_VALUE 0
#define LM96570_SPI_NUM_OF_BITS_EDGE_TYPE NONE
#define LM96570_SPI_NUM_OF_BITS_FREQ 50000000
#define LM96570_SPI_NUM_OF_BITS_HAS_IN 0
#define LM96570_SPI_NUM_OF_BITS_HAS_OUT 1
#define LM96570_SPI_NUM_OF_BITS_HAS_TRI 0
#define LM96570_SPI_NUM_OF_BITS_IRQ_TYPE NONE
#define LM96570_SPI_NUM_OF_BITS_RESET_VALUE 0

/*
 * Macros for device 'general_cnt_in', class 'altera_avalon_pio'
 * The macros are prefixed with 'GENERAL_CNT_IN_'.
 * The prefix is the slave descriptor.
 */
#define GENERAL_CNT_IN_COMPONENT_TYPE altera_avalon_pio
#define GENERAL_CNT_IN_COMPONENT_NAME general_cnt_in
#define GENERAL_CNT_IN_BASE 0x1e0
#define GENERAL_CNT_IN_SPAN 16
#define GENERAL_CNT_IN_END 0x1ef
#define GENERAL_CNT_IN_BIT_CLEARING_EDGE_REGISTER 0
#define GENERAL_CNT_IN_BIT_MODIFYING_OUTPUT_REGISTER 0
```

```
#define GENERAL_CNT_IN_CAPTURE 0
#define GENERAL_CNT_IN_DATA_WIDTH 32
#define GENERAL_CNT_IN_DO_TEST_BENCH_WIRING 0
#define GENERAL_CNT_IN_DRIVEN_SIM_VALUE 0
#define GENERAL_CNT_IN_EDGE_TYPE NONE
#define GENERAL_CNT_IN_FREQ 50000000
#define GENERAL_CNT_IN_HAS_IN 1
#define GENERAL_CNT_IN_HAS_OUT 0
#define GENERAL_CNT_IN_HAS_TRI 0
#define GENERAL_CNT_IN_IRQ_TYPE NONE
#define GENERAL_CNT_IN_RESET_VALUE 0

/*
 * Macros for device 'general_cnt_out', class 'altera_avalon_pio'
 * The macros are prefixed with 'GENERAL_CNT_OUT_'.
 * The prefix is the slave descriptor.
 */
#define GENERAL_CNT_OUT_COMPONENT_TYPE altera_avalon_pio
#define GENERAL_CNT_OUT_COMPONENT_NAME general_cnt_out
#define GENERAL_CNT_OUT_BASE 0x1f0
#define GENERAL_CNT_OUT_SPAN 16
#define GENERAL_CNT_OUT_END 0x1ff
#define GENERAL_CNT_OUT_BIT_CLEARING_EDGE_REGISTER 0
#define GENERAL_CNT_OUT_BIT_MODIFYING_OUTPUT_REGISTER 0
#define GENERAL_CNT_OUT_CAPTURE 0
#define GENERAL_CNT_OUT_DATA_WIDTH 32
#define GENERAL_CNT_OUT_DO_TEST_BENCH_WIRING 0
#define GENERAL_CNT_OUT_DRIVEN_SIM_VALUE 0
#define GENERAL_CNT_OUT_EDGE_TYPE NONE
#define GENERAL_CNT_OUT_FREQ 50000000
#define GENERAL_CNT_OUT_HAS_IN 0
#define GENERAL_CNT_OUT_HAS_OUT 1
#define GENERAL_CNT_OUT_HAS_TRI 0
#define GENERAL_CNT_OUT_IRQ_TYPE NONE
#define GENERAL_CNT_OUT_RESET_VALUE 0

/*
 * Macros for device 'lm96570_spi_out_2', class 'altera_avalon_pio'
 * The macros are prefixed with 'LM96570_SPI_OUT_2_'.
 * The prefix is the slave descriptor.
 */
#define LM96570_SPI_OUT_2_COMPONENT_TYPE altera_avalon_pio
#define LM96570_SPI_OUT_2_COMPONENT_NAME lm96570_spi_out_2
```

```
#define LM96570_SPI_OUT_2_BASE 0x200
#define LM96570_SPI_OUT_2_SPAN 16
#define LM96570_SPI_OUT_2_END 0x20f
#define LM96570_SPI_OUT_2_BIT_CLEARING_EDGE_REGISTER 0
#define LM96570_SPI_OUT_2_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LM96570_SPI_OUT_2_CAPTURE 0
#define LM96570_SPI_OUT_2_DATA_WIDTH 6
#define LM96570_SPI_OUT_2_DO_TEST_BENCH_WIRING 0
#define LM96570_SPI_OUT_2_DRIVEN_SIM_VALUE 0
#define LM96570_SPI_OUT_2_EDGE_TYPE NONE
#define LM96570_SPI_OUT_2_FREQ 50000000
#define LM96570_SPI_OUT_2_HAS_IN 1
#define LM96570_SPI_OUT_2_HAS_OUT 0
#define LM96570_SPI_OUT_2_HAS_TRI 0
#define LM96570_SPI_OUT_2_IRQ_TYPE NONE
#define LM96570_SPI_OUT_2_RESET_VALUE 0

/*
 * Macros for device 'lm96570_spi_out_1', class 'altera_avalon_pio'
 * The macros are prefixed with 'LM96570_SPI_OUT_1_'.
 * The prefix is the slave descriptor.
 */
#define LM96570_SPI_OUT_1_COMPONENT_TYPE altera_avalon_pio
#define LM96570_SPI_OUT_1_COMPONENT_NAME lm96570_spi_out_1
#define LM96570_SPI_OUT_1_BASE 0x210
#define LM96570_SPI_OUT_1_SPAN 16
#define LM96570_SPI_OUT_1_END 0x21f
#define LM96570_SPI_OUT_1_BIT_CLEARING_EDGE_REGISTER 0
#define LM96570_SPI_OUT_1_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LM96570_SPI_OUT_1_CAPTURE 0
#define LM96570_SPI_OUT_1_DATA_WIDTH 32
#define LM96570_SPI_OUT_1_DO_TEST_BENCH_WIRING 0
#define LM96570_SPI_OUT_1_DRIVEN_SIM_VALUE 0
#define LM96570_SPI_OUT_1_EDGE_TYPE NONE
#define LM96570_SPI_OUT_1_FREQ 50000000
#define LM96570_SPI_OUT_1_HAS_IN 1
#define LM96570_SPI_OUT_1_HAS_OUT 0
#define LM96570_SPI_OUT_1_HAS_TRI 0
#define LM96570_SPI_OUT_1_IRQ_TYPE NONE
#define LM96570_SPI_OUT_1_RESET_VALUE 0

/*
 * Macros for device 'lm96570_spi_out_0', class 'altera_avalon_pio'
```

```
 * The macros are prefixed with 'LM96570_SPI_OUT_0_'.
 * The prefix is the slave descriptor.
 */
#define LM96570_SPI_OUT_0_COMPONENT_TYPE altera_avalon_pio
#define LM96570_SPI_OUT_0_COMPONENT_NAME lm96570_spi_out_0
#define LM96570_SPI_OUT_0_BASE 0x220
#define LM96570_SPI_OUT_0_SPAN 16
#define LM96570_SPI_OUT_0_END 0x22f
#define LM96570_SPI_OUT_0_BIT_CLEARING_EDGE_REGISTER 0
#define LM96570_SPI_OUT_0_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LM96570_SPI_OUT_0_CAPTURE 0
#define LM96570_SPI_OUT_0_DATA_WIDTH 32
#define LM96570_SPI_OUT_0_DO_TEST_BENCH_WIRING 0
#define LM96570_SPI_OUT_0_DRIVEN_SIM_VALUE 0
#define LM96570_SPI_OUT_0_EDGE_TYPE NONE
#define LM96570_SPI_OUT_0_FREQ 50000000
#define LM96570_SPI_OUT_0_HAS_IN 1
#define LM96570_SPI_OUT_0_HAS_OUT 0
#define LM96570_SPI_OUT_0_HAS_TRI 0
#define LM96570_SPI_OUT_0_IRQ_TYPE NONE
#define LM96570_SPI_OUT_0_RESET_VALUE 0

/*
 * Macros for device 'lm96570_spi_in_2', class 'altera_avalon_pio'
 * The macros are prefixed with 'LM96570_SPI_IN_2_'.
 * The prefix is the slave descriptor.
 */
#define LM96570_SPI_IN_2_COMPONENT_TYPE altera_avalon_pio
#define LM96570_SPI_IN_2_COMPONENT_NAME lm96570_spi_in_2
#define LM96570_SPI_IN_2_BASE 0x230
#define LM96570_SPI_IN_2_SPAN 16
#define LM96570_SPI_IN_2_END 0x23f
#define LM96570_SPI_IN_2_BIT_CLEARING_EDGE_REGISTER 0
#define LM96570_SPI_IN_2_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LM96570_SPI_IN_2_CAPTURE 0
#define LM96570_SPI_IN_2_DATA_WIDTH 6
#define LM96570_SPI_IN_2_DO_TEST_BENCH_WIRING 0
#define LM96570_SPI_IN_2_DRIVEN_SIM_VALUE 0
#define LM96570_SPI_IN_2_EDGE_TYPE NONE
#define LM96570_SPI_IN_2_FREQ 50000000
#define LM96570_SPI_IN_2_HAS_IN 0
#define LM96570_SPI_IN_2_HAS_OUT 1
#define LM96570_SPI_IN_2_HAS_TRI 0
```

```
#define LM96570_SPI_IN_2_IRQ_TYPE NONE
#define LM96570_SPI_IN_2_RESET_VALUE 0

/*
 * Macros for device 'lm96570_spi_in_1', class 'altera_avalon_pio'
 * The macros are prefixed with 'LM96570_SPI_IN_1_'.
 * The prefix is the slave descriptor.
 */
#define LM96570_SPI_IN_1_COMPONENT_TYPE altera_avalon_pio
#define LM96570_SPI_IN_1_COMPONENT_NAME lm96570_spi_in_1
#define LM96570_SPI_IN_1_BASE 0x240
#define LM96570_SPI_IN_1_SPAN 16
#define LM96570_SPI_IN_1_END 0x24f
#define LM96570_SPI_IN_1_BIT_CLEARING_EDGE_REGISTER 0
#define LM96570_SPI_IN_1_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LM96570_SPI_IN_1_CAPTURE 0
#define LM96570_SPI_IN_1_DATA_WIDTH 32
#define LM96570_SPI_IN_1_DO_TEST_BENCH_WIRING 0
#define LM96570_SPI_IN_1_DRIVEN_SIM_VALUE 0
#define LM96570_SPI_IN_1_EDGE_TYPE NONE
#define LM96570_SPI_IN_1_FREQ 50000000
#define LM96570_SPI_IN_1_HAS_IN 0
#define LM96570_SPI_IN_1_HAS_OUT 1
#define LM96570_SPI_IN_1_HAS_TRI 0
#define LM96570_SPI_IN_1_IRQ_TYPE NONE
#define LM96570_SPI_IN_1_RESET_VALUE 0

/*
 * Macros for device 'lm96570_spi_in_0', class 'altera_avalon_pio'
 * The macros are prefixed with 'LM96570_SPI_IN_0_'.
 * The prefix is the slave descriptor.
 */
#define LM96570_SPI_IN_0_COMPONENT_TYPE altera_avalon_pio
#define LM96570_SPI_IN_0_COMPONENT_NAME lm96570_spi_in_0
#define LM96570_SPI_IN_0_BASE 0x250
#define LM96570_SPI_IN_0_SPAN 16
#define LM96570_SPI_IN_0_END 0x25f
#define LM96570_SPI_IN_0_BIT_CLEARING_EDGE_REGISTER 0
#define LM96570_SPI_IN_0_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LM96570_SPI_IN_0_CAPTURE 0
#define LM96570_SPI_IN_0_DATA_WIDTH 32
#define LM96570_SPI_IN_0_DO_TEST_BENCH_WIRING 0
#define LM96570_SPI_IN_0_DRIVEN_SIM_VALUE 0
```

```
#define LM96570_SPI_IN_0_EDGE_TYPE NONE
#define LM96570_SPI_IN_0_FREQ 50000000
#define LM96570_SPI_IN_0_HAS_IN 0
#define LM96570_SPI_IN_0_HAS_OUT 1
#define LM96570_SPI_IN_0_HAS_TRI 0
#define LM96570_SPI_IN_0_IRQ_TYPE NONE
#define LM96570_SPI_IN_0_RESET_VALUE 0

/*
 * Macros for device 'led_pio', class 'altera_avalon_pio'
 * The macros are prefixed with 'LED_PIO_'.
 * The prefix is the slave descriptor.
 */
#define LED_PIO_COMPONENT_TYPE altera_avalon_pio
#define LED_PIO_COMPONENT_NAME led_pio
#define LED_PIO_BASE 0x260
#define LED_PIO_SPAN 16
#define LED_PIO_END 0x26f
#define LED_PIO_BIT_CLEARING_EDGE_REGISTER 0
#define LED_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LED_PIO_CAPTURE 0
#define LED_PIO_DATA_WIDTH 10
#define LED_PIO_DO_TEST_BENCH_WIRING 0
#define LED_PIO_DRIVEN_SIM_VALUE 0
#define LED_PIO_EDGE_TYPE NONE
#define LED_PIO_FREQ 50000000
#define LED_PIO_HAS_IN 0
#define LED_PIO_HAS_OUT 1
#define LED_PIO_HAS_TRI 0
#define LED_PIO_IRQ_TYPE NONE
#define LED_PIO_RESET_VALUE 0

/*
 * Macros for device 'dipsw_pio', class 'altera_avalon_pio'
 * The macros are prefixed with 'DIPSW_PIO_'.
 * The prefix is the slave descriptor.
 */
#define DIPSW_PIO_COMPONENT_TYPE altera_avalon_pio
#define DIPSW_PIO_COMPONENT_NAME dipsw_pio
#define DIPSW_PIO_BASE 0x270
#define DIPSW_PIO_SPAN 16
#define DIPSW_PIO_END 0x27f
#define DIPSW_PIO_IRQ 0
```

```
#define DIPSW_PIO_BIT_CLEARING_EDGE_REGISTER 1
#define DIPSW_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define DIPSW_PIO_CAPTURE 1
#define DIPSW_PIO_DATA_WIDTH 10
#define DIPSW_PIO_DO_TEST_BENCH_WIRING 0
#define DIPSW_PIO_DRIVEN_SIM_VALUE 0
#define DIPSW_PIO_EDGE_TYPE ANY
#define DIPSW_PIO_FREQ 50000000
#define DIPSW_PIO_HAS_IN 1
#define DIPSW_PIO_HAS_OUT 0
#define DIPSW_PIO_HAS_TRI 0
#define DIPSW_PIO_IRQ_TYPE EDGE
#define DIPSW_PIO_RESET_VALUE 0

/*
 * Macros for device 'button_pio', class 'altera_avalon_pio'
 * The macros are prefixed with 'BUTTON_PIO_'.
 * The prefix is the slave descriptor.
 */
#define BUTTON_PIO_COMPONENT_TYPE altera_avalon_pio
#define BUTTON_PIO_COMPONENT_NAME button_pio
#define BUTTON_PIO_BASE 0x280
#define BUTTON_PIO_SPAN 16
#define BUTTON_PIO_END 0x28f
#define BUTTON_PIO_IRQ 1
#define BUTTON_PIO_BIT_CLEARING_EDGE_REGISTER 1
#define BUTTON_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define BUTTON_PIO_CAPTURE 1
#define BUTTON_PIO_DATA_WIDTH 4
#define BUTTON_PIO_DO_TEST_BENCH_WIRING 0
#define BUTTON_PIO_DRIVEN_SIM_VALUE 0
#define BUTTON_PIO_EDGE_TYPE FALLING
#define BUTTON_PIO_FREQ 50000000
#define BUTTON_PIO_HAS_IN 1
#define BUTTON_PIO_HAS_OUT 0
#define BUTTON_PIO_HAS_TRI 0
#define BUTTON_PIO_IRQ_TYPE EDGE
#define BUTTON_PIO_RESET_VALUE 0

/*
 * Macros for device 'fifo_sink_CH_A_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_A_OUT_'.
 * The prefix is the slave descriptor.
```

```
 */
#define FIFO_SINK_CH_A_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_A_OUT_COMPONENT_NAME fifo_sink_CH_A
#define FIFO_SINK_CH_A_OUT_BASE 0x290
#define FIFO_SINK_CH_A_OUT_SPAN 8
#define FIFO_SINK_CH_A_OUT_END 0x297
#define FIFO_SINK_CH_A_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_A_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_A_OUT_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_A_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_A_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_A_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_A_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_A_OUT_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_A_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_A_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_A_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_A_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_A_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_A_OUT_USE_IRQ 0
#define FIFO_SINK_CH_A_OUT_USE_PACKET 0
#define FIFO_SINK_CH_A_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_A_OUT_USE_REGISTER 0
#define FIFO_SINK_CH_A_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_F_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_F_OUT_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_F_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_F_OUT_COMPONENT_NAME fifo_sink_CH_F
#define FIFO_SINK_CH_F_OUT_BASE 0x298
#define FIFO_SINK_CH_F_OUT_SPAN 8
#define FIFO_SINK_CH_F_OUT_END 0x29f
#define FIFO_SINK_CH_F_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_F_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_F_OUT_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_F_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_F_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_F_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_F_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_F_OUT_SYMBOLS_PER_BEAT 2
```

```
#define FIFO_SINK_CH_F_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_F_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_F_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_F_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_F_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_F_OUT_USE_IRQ 0
#define FIFO_SINK_CH_F_OUT_USE_PACKET 0
#define FIFO_SINK_CH_F_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_F_OUT_USE_REGISTER 0
#define FIFO_SINK_CH_F_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_H_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_H_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_H_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_H_OUT_CSR_COMPONENT_NAME fifo_sink_CH_H
#define FIFO_SINK_CH_H_OUT_CSR_BASE 0x2a0
#define FIFO_SINK_CH_H_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_H_OUT_CSR_END 0x2bf
#define FIFO_SINK_CH_H_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_H_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_H_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_H_OUT_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_H_OUT_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_H_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_H_OUT_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_H_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_H_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_H_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_H_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_H_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_H_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_H_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_H_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_H_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_H_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_H_OUT_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_G_in_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_G_IN_CSR_'.
```

```
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_G_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_G_IN_CSR_COMPONENT_NAME fifo_sink_CH_G
#define FIFO_SINK_CH_G_IN_CSR_BASE 0x2c0
#define FIFO_SINK_CH_G_IN_CSR_SPAN 32
#define FIFO_SINK_CH_G_IN_CSR_END 0x2df
#define FIFO_SINK_CH_G_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_G_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_G_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_G_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_G_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_G_IN_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_G_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_G_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_G_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_G_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_G_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_G_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_G_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_G_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_G_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_G_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_G_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_G_IN_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_G_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_G_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_G_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_G_OUT_CSR_COMPONENT_NAME fifo_sink_CH_G
#define FIFO_SINK_CH_G_OUT_CSR_BASE 0x2e0
#define FIFO_SINK_CH_G_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_G_OUT_CSR_END 0x2ff
#define FIFO_SINK_CH_G_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_G_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_G_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_G_OUT_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_G_OUT_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_G_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_G_OUT_CSR_SINGLE_CLOCK_MODE 0
```

```
#define FIFO_SINK_CH_G_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_G_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_G_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_G_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_G_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_G_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_G_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_G_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_G_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_G_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_G_OUT_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_F_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_F_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_F_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_F_OUT_CSR_COMPONENT_NAME fifo_sink_CH_F
#define FIFO_SINK_CH_F_OUT_CSR_BASE 0x300
#define FIFO_SINK_CH_F_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_F_OUT_CSR_END 0x31f
#define FIFO_SINK_CH_F_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_F_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_F_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_F_OUT_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_F_OUT_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_F_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_F_OUT_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_F_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_F_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_F_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_F_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_F_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_F_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_F_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_F_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_F_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_F_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_F_OUT_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_F_in_csr', class 'altera_avalon_fifo'
```

```
 * The macros are prefixed with 'FIFO_SINK_CH_F_IN_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_F_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_F_IN_CSR_COMPONENT_NAME fifo_sink_CH_F
#define FIFO_SINK_CH_F_IN_CSR_BASE 0x320
#define FIFO_SINK_CH_F_IN_CSR_SPAN 32
#define FIFO_SINK_CH_F_IN_CSR_END 0x33f
#define FIFO_SINK_CH_F_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_F_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_F_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_F_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_F_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_F_IN_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_F_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_F_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_F_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_F_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_F_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_F_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_F_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_F_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_F_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_F_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_F_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_F_IN_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_E_in_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_E_IN_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_E_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_E_IN_CSR_COMPONENT_NAME fifo_sink_CH_E
#define FIFO_SINK_CH_E_IN_CSR_BASE 0x340
#define FIFO_SINK_CH_E_IN_CSR_SPAN 32
#define FIFO_SINK_CH_E_IN_CSR_END 0x35f
#define FIFO_SINK_CH_E_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_E_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_E_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_E_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_E_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_E_IN_CSR_FIFO_DEPTH 4096
```

```
#define FIFO_SINK_CH_E_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_E_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_E_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_E_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_E_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_E_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_E_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_E_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_E_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_E_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_E_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_E_IN_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_E_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_E_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_E_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_E_OUT_CSR_COMPONENT_NAME fifo_sink_CH_E
#define FIFO_SINK_CH_E_OUT_CSR_BASE 0x360
#define FIFO_SINK_CH_E_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_E_OUT_CSR_END 0x37f
#define FIFO_SINK_CH_E_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_E_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_E_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_E_OUT_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_E_OUT_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_E_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_E_OUT_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_E_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_E_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_E_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_E_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_E_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_E_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_E_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_E_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_E_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_E_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_E_OUT_CSR_USE_WRITE_CONTROL 1

/*
```

```
 * Macros for device 'fifo_sink_CH_D_in_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_D_IN_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_D_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_D_IN_CSR_COMPONENT_NAME fifo_sink_CH_D
#define FIFO_SINK_CH_D_IN_CSR_BASE 0x380
#define FIFO_SINK_CH_D_IN_CSR_SPAN 32
#define FIFO_SINK_CH_D_IN_CSR_END 0x39f
#define FIFO_SINK_CH_D_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_D_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_D_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_D_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_D_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_D_IN_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_D_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_D_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_D_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_D_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_D_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_D_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_D_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_D_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_D_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_D_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_D_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_D_IN_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_D_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_D_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_D_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_D_OUT_CSR_COMPONENT_NAME fifo_sink_CH_D
#define FIFO_SINK_CH_D_OUT_CSR_BASE 0x3a0
#define FIFO_SINK_CH_D_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_D_OUT_CSR_END 0x3bf
#define FIFO_SINK_CH_D_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_D_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_D_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_D_OUT_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_D_OUT_CSR_ERROR_WIDTH 0
```

```
#define FIFO_SINK_CH_D_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_D_OUT_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_D_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_D_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_D_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_D_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_D_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_D_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_D_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_D_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_D_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_D_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_D_OUT_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_C_in_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_C_IN_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_C_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_C_IN_CSR_COMPONENT_NAME fifo_sink_CH_C
#define FIFO_SINK_CH_C_IN_CSR_BASE 0x3c0
#define FIFO_SINK_CH_C_IN_CSR_SPAN 32
#define FIFO_SINK_CH_C_IN_CSR_END 0x3df
#define FIFO_SINK_CH_C_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_C_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_C_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_C_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_C_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_C_IN_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_C_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_C_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_C_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_C_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_C_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_C_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_C_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_C_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_C_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_C_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_C_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_C_IN_CSR_USE_WRITE_CONTROL 1
```

```
/*
 * Macros for device 'fifo_sink_CH_C_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_C_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_C_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_C_OUT_CSR_COMPONENT_NAME fifo_sink_CH_C
#define FIFO_SINK_CH_C_OUT_CSR_BASE 0x3e0
#define FIFO_SINK_CH_C_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_C_OUT_CSR_END 0x3ff
#define FIFO_SINK_CH_C_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_C_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_C_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_C_OUT_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_C_OUT_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_C_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_C_OUT_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_C_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_C_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_C_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_C_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_C_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_C_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_C_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_C_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_C_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_C_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_C_OUT_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_B_out_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_B_OUT_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_B_OUT_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_B_OUT_CSR_COMPONENT_NAME fifo_sink_CH_B
#define FIFO_SINK_CH_B_OUT_CSR_BASE 0x400
#define FIFO_SINK_CH_B_OUT_CSR_SPAN 32
#define FIFO_SINK_CH_B_OUT_CSR_END 0x41f
#define FIFO_SINK_CH_B_OUT_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_B_OUT_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_B_OUT_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_B_OUT_CSR_CHANNEL_WIDTH 0
```

```
#define FIFO_SINK_CH_B_OUT_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_B_OUT_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_B_OUT_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_B_OUT_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_B_OUT_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_B_OUT_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_B_OUT_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_B_OUT_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_B_OUT_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_B_OUT_CSR_USE_IRQ 0
#define FIFO_SINK_CH_B_OUT_CSR_USE_PACKET 0
#define FIFO_SINK_CH_B_OUT_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_B_OUT_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_B_OUT_CSR_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_B_in_csr', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_B_IN_CSR_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_B_IN_CSR_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_B_IN_CSR_COMPONENT_NAME fifo_sink_CH_B
#define FIFO_SINK_CH_B_IN_CSR_BASE 0x420
#define FIFO_SINK_CH_B_IN_CSR_SPAN 32
#define FIFO_SINK_CH_B_IN_CSR_END 0x43f
#define FIFO_SINK_CH_B_IN_CSR_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_B_IN_CSR_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_B_IN_CSR_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_B_IN_CSR_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_B_IN_CSR_ERROR_WIDTH 0
#define FIFO_SINK_CH_B_IN_CSR_FIFO_DEPTH 4096
#define FIFO_SINK_CH_B_IN_CSR_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_B_IN_CSR_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_B_IN_CSR_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_B_IN_CSR_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_B_IN_CSR_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_B_IN_CSR_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_B_IN_CSR_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_B_IN_CSR_USE_IRQ 0
#define FIFO_SINK_CH_B_IN_CSR_USE_PACKET 0
#define FIFO_SINK_CH_B_IN_CSR_USE_READ_CONTROL 1
#define FIFO_SINK_CH_B_IN_CSR_USE_REGISTER 0
#define FIFO_SINK_CH_B_IN_CSR_USE_WRITE_CONTROL 1
```

```
/*
 * Macros for device 'fifo_sink_CH_E_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_E_OUT_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_E_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_E_OUT_COMPONENT_NAME fifo_sink_CH_E
#define FIFO_SINK_CH_E_OUT_BASE 0x440
#define FIFO_SINK_CH_E_OUT_SPAN 8
#define FIFO_SINK_CH_E_OUT_END 0x447
#define FIFO_SINK_CH_E_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_E_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_E_OUT_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_E_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_E_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_E_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_E_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_E_OUT_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_E_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_E_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_E_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_E_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_E_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_E_OUT_USE_IRQ 0
#define FIFO_SINK_CH_E_OUT_USE_PACKET 0
#define FIFO_SINK_CH_E_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_E_OUT_USE_REGISTER 0
#define FIFO_SINK_CH_E_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_D_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_D_OUT_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_D_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_D_OUT_COMPONENT_NAME fifo_sink_CH_D
#define FIFO_SINK_CH_D_OUT_BASE 0x448
#define FIFO_SINK_CH_D_OUT_SPAN 8
#define FIFO_SINK_CH_D_OUT_END 0x44f
#define FIFO_SINK_CH_D_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_D_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_D_OUT_BITS_PER_SYMBOL 16
```

```
#define FIFO_SINK_CH_D_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_D_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_D_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_D_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_D_OUT_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_D_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_D_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_D_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_D_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_D_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_D_OUT_USE_IRQ 0
#define FIFO_SINK_CH_D_OUT_USE_PACKET 0
#define FIFO_SINK_CH_D_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_D_OUT_USE_REGISTER 0
#define FIFO_SINK_CH_D_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_C_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_C_OUT_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_C_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_C_OUT_COMPONENT_NAME fifo_sink_CH_C
#define FIFO_SINK_CH_C_OUT_BASE 0x450
#define FIFO_SINK_CH_C_OUT_SPAN 8
#define FIFO_SINK_CH_C_OUT_END 0x457
#define FIFO_SINK_CH_C_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_C_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_C_OUT_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_C_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_C_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_C_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_C_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_C_OUT_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_C_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_C_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_C_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_C_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_C_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_C_OUT_USE_IRQ 0
#define FIFO_SINK_CH_C_OUT_USE_PACKET 0
#define FIFO_SINK_CH_C_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_C_OUT_USE_REGISTER 0
```

```
#define FIFO_SINK_CH_C_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'fifo_sink_CH_B_out', class 'altera_avalon_fifo'
 * The macros are prefixed with 'FIFO_SINK_CH_B_OUT_'.
 * The prefix is the slave descriptor.
 */
#define FIFO_SINK_CH_B_OUT_COMPONENT_TYPE altera_avalon_fifo
#define FIFO_SINK_CH_B_OUT_COMPONENT_NAME fifo_sink_CH_B
#define FIFO_SINK_CH_B_OUT_BASE 0x458
#define FIFO_SINK_CH_B_OUT_SPAN 8
#define FIFO_SINK_CH_B_OUT_END 0x45f
#define FIFO_SINK_CH_B_OUT_AVALONMM_AVALONMM_DATA_WIDTH 32
#define FIFO_SINK_CH_B_OUT_AVALONMM_AVALONST_DATA_WIDTH 32
#define FIFO_SINK_CH_B_OUT_BITS_PER_SYMBOL 16
#define FIFO_SINK_CH_B_OUT_CHANNEL_WIDTH 0
#define FIFO_SINK_CH_B_OUT_ERROR_WIDTH 0
#define FIFO_SINK_CH_B_OUT_FIFO_DEPTH 4096
#define FIFO_SINK_CH_B_OUT_SINGLE_CLOCK_MODE 0
#define FIFO_SINK_CH_B_OUT_SYMBOLS_PER_BEAT 2
#define FIFO_SINK_CH_B_OUT_USE_AVALONMM_READ_SLAVE 1
#define FIFO_SINK_CH_B_OUT_USE_AVALONMM_WRITE_SLAVE 0
#define FIFO_SINK_CH_B_OUT_USE_AVALONST_SINK 1
#define FIFO_SINK_CH_B_OUT_USE_AVALONST_SOURCE 0
#define FIFO_SINK_CH_B_OUT_USE_BACKPRESSURE 1
#define FIFO_SINK_CH_B_OUT_USE_IRQ 0
#define FIFO_SINK_CH_B_OUT_USE_PACKET 0
#define FIFO_SINK_CH_B_OUT_USE_READ_CONTROL 1
#define FIFO_SINK_CH_B_OUT_USE_REGISTER 0
#define FIFO_SINK_CH_B_OUT_USE_WRITE_CONTROL 1

/*
 * Macros for device 'sysid_qsys', class 'altera_avalon_sysid_qsys'
 * The macros are prefixed with 'SYSID_QSYS_'.
 * The prefix is the slave descriptor.
 */
#define SYSID_QSYS_COMPONENT_TYPE altera_avalon_sysid_qsys
#define SYSID_QSYS_COMPONENT_NAME sysid_qsys
#define SYSID_QSYS_BASE 0x460
#define SYSID_QSYS_SPAN 8
#define SYSID_QSYS_END 0x467
#define SYSID_QSYS_ID 2899645186
#define SYSID_QSYS_TIMESTAMP 1539131202
```

```
/*
 * Macros for device 'jtag_uart', class 'altera_avalon_jtag_uart'
 * The macros are prefixed with 'JTAG_UART_'.
 * The prefix is the slave descriptor.
 */
#define JTAG_UART_COMPONENT_TYPE altera_avalon_jtag_uart
#define JTAG_UART_COMPONENT_NAME jtag_uart
#define JTAG_UART_BASE 0x20000
#define JTAG_UART_SPAN 8
#define JTAG_UART_END 0x20007
#define JTAG_UART_IRQ 2
#define JTAG_UART_READ_DEPTH 64
#define JTAG_UART_READ_THRESHOLD 8
#define JTAG_UART_WRITE_DEPTH 64
#define JTAG_UART_WRITE_THRESHOLD 8


#endif /* _ALTERA_HPS_SOC_SYSTEM_H_ */
```

Appendix D
MatLab Code for System Prototype

This section provides the MatLab code that was mentioned for the current prototype of the system.

1 Single Image Matlab Code

```
%Image construction with data taken from ultrasound system
with 25 percent overlap

function Percent25overlapV2 (echos)
Time_sample = size(echos,2);%8192; %number of time steps
N =size(echos,1);% Number of separate received echos by the system

I = echos;

%% remove the offset from the data:
I_deoff = zeros(N,Time_sample);

for i=1:N

   I_deoff(i,:) = I(i,:) -(1/Time_sample)*sum(I(i,:));
   %subtract average of each echo from the echo signal(itself)

end

%% Apply hilbert transform to make IQ data out of signal

I_analytical = zeros(N,Time_sample);

for i = 1:N

    I_analytical(i,:) = hilbert(I_deoff(i,:));

end

%% using fft to make one segment of image out of each 8 signals
L = 8;
w = [0.3,0.7,1,1,1,1,0.7,0.3];
```

```
K = zeros(8,Time_sample);
I_final = zeros(N,Time_sample);


for i = 1:8:N
    K = I_analytical(i:i+7,:); % store each 8 echos in a new matrix
    I_final(i:i+7,:) = fft(K,8,1);% take dft of the 8 channel
end

%% do windowing for each sub-image


I_win = zeros(N,Time_sample);


for j = 1:8:N

    I_win(j:j+7,:) = bsxfun(@times,I_final(j:j+7,:),w');

end

%% First we need to throw away first six echos in last scan and keep
last two.
A = zeros(82,Time_sample);
A(1:80,:) = I_win(1:80,:);
A(81,:) = (1/w(1,2))*I_win(87,:);
A(82,:) = (1/w(1,1))*I_win(88,:);

Imag_final(1,:)  =  (1/w(1,1)) * A(1,:);
Imag_final(2,:)  =  (1/w(1,2)) * A(2,:);
Imag_final(61,:) =  (1/w(1,2)) * A(79,:);
Imag_final(62,:) =  (1/w(1,1)) * A(80,:);
Imag_final(63:64,:) = A(81:82,:);


for i = 4:8:80

    Imag_final((3*i/4):(3*i/4)+3,:) = A(i-1:i+2,:);
end

for i = 8:8:72
    Imag_final((3*i/4)+1:(3*i/4)+2,:) = A(i-1:i,:) + A(i+1:i+2,:);
```

```
end
P_magnitude = abs(Imag_final);


% scale the image
P_scaled = zeros(size(P_magnitude,1),size(P_magnitude,2)/16);

 for i = 1:size(P_magnitude,1)
    for k = 1:size(P_magnitude,2)/16
        P_scaled(i,k) = sum(P_magnitude(i,16*(k-1)+1:16*k)/16);
    end
end


P_red = P_scaled(:,37:100);
F_mat = mat2gray(P_red);
J = imadjust(F_mat);

%figure;imshow(F_mat)
figure; imshow(J);
figure;imagesc(F_mat); axis image;colorbar;
figure;imagesc(J); axis image;colorbar; colormap(gray);
end
```

Appendix E
Python Code of System Prototype

This section covers the Python codes that were used in the current system prototype.

1 Python Code for Single Image

```
'''25% overlap python code optimized for single frame imaging'''
import time
from datetime import timedelta

start_time = time.monotonic()

import numpy as np  # To define array and some basic math on arrays
import os  # To find and change directories
import matplotlib.pyplot as plt  # To plot the data
from scipy.signal import hilbert  # To do hilbert transform

cwd = os.getcwd()  # find current directory
print(os.getcwd())  # print current directory

time_sample = 8000  # number of time samples
                      coming from sampling frequency
channel = 88  # number of total initial channels including overlapping

while True:

    I = np.zeros(shape=(channel, time_sample))
    # define an array to keep all of imported data
    I = np.loadtxt('databank.txt', delimiter=' ', usecols=range(8000))
    # in matlab (dlmwrite('timesample.txt', B))

    time_point = np.zeros(shape=(1, time_sample))

    I_new = np.zeros(shape=(88, 8000))  # define a new matrix to store
                                          unoffseted data

    for i in range(0, 88):  # remove the offset from each echo signal
                              and store them in a new array of I_new
        I_new[i, :] = I[i, :] - (1 / 8000) * I[i, :].sum()
        # subtract mean of each echo signal from all time points
```

```
    analytic_signal = np.zeros(shape=(88, 8000)).astype(complex)
    # define a matrix to store IQ data

    for i in range(0, 88):
        analytic_signal[i, :] = hilbert(I_new[i, :])
        # use hilbert transform
          to find the IQ data
   N = 8

    P = np.zeros(shape=(8, 8000)).astype(complex)
    M = np.zeros(shape=(8, 8)).astype(complex)
    G = np.zeros(shape=(8, 8000)).astype(complex)
    I_final = np.zeros(shape=(88, 8000)).astype(complex)

    for i in range(0, 81, 8):  # find 8 point DFT of each sub window
        G = analytic_signal[range(i, i + 8), :]
        for j in range(0, time_sample, 8):
            M = G[:, range(j, j + 8)]
            P[:, range(j, j + 8)] = np.fft.fft2(M[:, :])
            I_final[range(i, i + 8), :] = P  # P_shift

    window = np.array(
        [[0.3], [0.7], [1], [1], [1], [1], [0.7], [0.3]])
        # define a triangular window

    S = np.zeros(shape=(8, 8000))
    S_tri = np.zeros(shape=(8, 8000)).astype(complex)
    I_tri = np.zeros(shape=(88, 8000)).astype(complex)
    I_windowed = np.zeros(shape=(65, 8000)).astype(complex)

    for j in range(0, 81, 8):
        S = I_final[range(j, j + 8), :]
        S_tri = S * window
        I_tri[range(j, j + 8), :] = S_tri

    I_windowed[range(0, 2), :] = I_tri[range(0, 2), :]
    I_windowed[range(63, 65), :] = I_tri[range(83, 85), :]

    for i in range(4, 81, 8):
        I_windowed[range(int(3 * i / 4), int(3 * i / 4) + 4), :]
        = I_tri[range(i - 1, i + 3), :]
```

```
    for i in range(8, 81, 8):
        I_windowed[range(int(3 * i / 4) + 1, int(3 * i / 4) + 3), :]
        = I_tri[range(i - 1, i + 1), :] + I_tri[range(i + 1, i + 3), :]

    P_magnitude = np.abs(I_windowed)
    P_dimention =  P_magnitude.shape

    P_scaled = np.zeros(shape=(P_dimention[0], P_dimention[1]//10))

    #for i in range(0, 64):
    for k in range(0, P_dimention[1]//10):
       P_scaled[:, k] = np.mean(P_magnitude[:,
       (range(10 * k, 10 * (k + 1)))], axis = 1)

    im = plt.imshow(P_scaled[:, 100:240], cmap='gray')
    end_time = time.monotonic()
    print(timedelta(seconds=end_time - start_time))
    plt.colorbar(im, orientation='horizontal')

    fig = plt.gcf()
    fig.show()

    fig.canvas.draw()
    plt.pause(0.01)
    fig.clf()
```

2 Python Code for Automated Imaging

```
'''25% overlap python code optimized for automated imaging'''

import time
from datetime import timedelta

import numpy as np                          # To define array and some
                                              basic math on arrays
import os                                   # To find and change directories
import matplotlib.pyplot as plt                 # To plot the data
from scipy.signal import hilbert                # To do hilbert transform
from subprocess import Popen, PIPE

cwd = os.getcwd()                          # find current directory
```

```
print(os.getcwd())                    # print current directory

time_sample = 8000                  # number of time samples coming
                                      from sampling frequency
channel = 88                        # number of total initial channels
                                      including overlapping

while True:
    print("start\n")
    start_time = time.monotonic()

    I = np.zeros(shape=(channel, time_sample))  # define an array to
                                                  keep all of imported
                                                  data
    process = Popen(['./de10-standard_test'],
     stdout=PIPE, stderr = PIPE, shell=True)

    end_time = time.monotonic()
    print(timedelta(seconds=end_time - start_time))

    stdout, stderr = process.communicate()

    end_time = time.monotonic()
    print(timedelta(seconds=end_time - start_time))

    stdchar = stdout.split()

    int_lst = [int(x) for x in stdchar]
    int_lst = np.array(int_lst)
    j = 0
    int_shape = np.zeros(shape=(88, 8000))

    for i in range(0, 704000, 8000):
        int_shape[j, :] = int_lst[range(i, i + 8000)]
        j = j + 1

    end_time = time.monotonic()
    print(timedelta(seconds=end_time - start_time))

    I[:, :] = int_shape[:,:]

    time_point = np.zeros(shape=(1, time_sample))
```

```
I_new = np.zeros(shape=(88, 8000))  # define a new matrix to store
                                      unoffsetted data

for i in range(0, 88):  # remove the offset from each echo signal
                          and store them in a new array of I_new
    I_new[i, :] = I[i, :] - (1 / 8000) * I[i, :].sum()
    # subtract mean of each echo signal from all time points

analytic_signal = np.zeros(shape=(88, 8000)).astype(complex)
# define a matrix to store IQ data

for i in range(0, 88):
    analytic_signal[i, :] = hilbert(I_new[i, :])
    # use hilbert transform
    #  to find the IQ data
N = 8

P = np.zeros(shape=(8, 8000)).astype(complex)
P_shift = np.zeros(shape=(8, 8000)).astype(complex)
M = np.zeros(shape=(8, 8)).astype(complex)
G = np.zeros(shape=(8, 8000)).astype(complex)
I_final = np.zeros(shape=(88, 8000)).astype(complex)

for i in range(0, 81, 8):  # find 8 point DFT of each sub window
    G = analytic_signal[range(i, i + 8), :]
    for j in range(0, time_sample, 8):
        M = G[:, range(j, j + 8)]
        P[:, range(j, j + 8)] = np.fft.fft2(M[:, :])
        # P_shift[:, j] = np.fft.fftshift(P[:, j])
    I_final[range(i, i + 8), :] = P  # P_shift

window = np.array(
    [[0.3], [0.7], [1], [1], [1], [1], [0.7], [0.3]])
    # define a triangular window

S = np.zeros(shape=(8, 8000))
S_tri = np.zeros(shape=(8, 8000)).astype(complex)
I_tri = np.zeros(shape=(88, 8000)).astype(complex)
I_windowed = np.zeros(shape=(65, 8000)).astype(complex)

for j in range(0, 81, 8):
    S = I_final[range(j, j + 8), :]
    S_tri = S * window
```

```
        I_tri[range(j, j + 8), :] = S_tri

    I_windowed[range(0, 2), :] = I_tri[range(0, 2), :]
    I_windowed[range(63, 65), :] = I_tri[range(83, 85), :]

    for i in range(4, 81, 8):
        I_windowed[range(int(3 * i / 4), int(3 * i / 4) + 4), :]
        = I_tri[range(i - 1, i + 3), :]

    for i in range(8, 81, 8):
        I_windowed[range(int(3 * i / 4) + 1, int(3 * i / 4) + 3), :]
        = I_tri[range(i - 1, i + 1), :]
          + I_tri[range(i + 1, i + 3), :]

    P_magnitude = np.abs(I_windowed)
    P_dimention = P_magnitude.shape

    P_scaled = np.zeros(shape=(P_dimention[0], P_dimention[1]//10))

    #for i in range(0, 64):
    for k in range(0, P_dimention[1]//10):
        P_scaled[:, k] = np.mean(P_magnitude[:,
        (range(10 * k, 10 * (k + 1)))], axis = 1)

    im = plt.imshow(P_scaled[:, 20:240], cmap='gray')
    end_time = time.monotonic()
    print(timedelta(seconds=end_time - start_time))
    plt.colorbar(im, orientation='horizontal')

    fig = plt.gcf()
    fig.show()


    fig.canvas.draw()
    plt.pause(0.01)
    fig.clf()
```

Appendix F
Preparation of this document

This document was prepared using pdfLaTeX and other open source tools. The (free) programs implemented are as follows:

- LaTeX implementation:

  MiKTeX

  http://www.miktex.org/

  TeXLive

  https://www.tug.org/texlive/

  MacTeX

  https://tug.org/mactex/

- TeX-oriented editing environments:

  Vim Text Editor

  https:http://www.vim.org/

- Bibliographical:

  BibTeX

  http://www.bibtex.org/

  Zotero

  https://www.zotero.org/

Complete References

[1] VWS Chan, S Abbas, R Brull, B Morrigl, and A Perlas. Ultrasound imaging for regional anesthesia. A Practical Guide. 2nd ed. Toronto, ON: Toronto Publishing Company, 2008.

[2] Catherine M Otto. Textbook of clinical echocardiography, 5e (endocardiography). 2004.

[3] Enrico Boni, Luca Bassi, Alessandro Dallai, Francesco Guidi, Valentino Meacci, Alessandro Ramalli, Stefano Ricci, and Piero Tortoli. ULA-OP 256: A 256-channel open scanner for development and real-time implementation of new ultrasound methods. IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control, 63(10):1488–1495, 2016.

[4] H. Li and D. C. Liu. An embedded high performance ultrasonic signal processing subsystem. In Intl. Conference on Embedded Software and Systems, pages 125–130, May 2009.

[5] Aya Ibrahim, Damien Doy, Claudio Loureiro, Elieva Pignat, Federico Angiolini, Marcel Arditi, J-P Thiran, and Giovanni De Micheli. Inexpensive 1024-channel 3D telesonography system on FPGA. In Biomedical Circuits and Systems Conference (BioCAS), 2017 IEEE, pages 1–4, 2017.

[6] Radu Bacaru. Rocketboards.org: Gsrd-ghrd overview.

[7] Intel Corporation Altera. Embedded Peripherals IP User Guide. Intel Corporation - Altera.

[8] Intel Corporation Altera. ALTDDIO Megafunction User Guide. Intel Corporation - Altera, 9 2010.

[9] Analog Devices. Octal LNA/VGA/AAF/12-Bit ADC and CW I/Q Demodulator. Analog Devices, 2009.

[10] Robert J. McGough. Fast Object-oriented C++ Ultrasound Simulator (FOCUS).

[11] A. Basak, V. Ranganathan, and S. Bhunia. A wearable ultrasonic assembly for point-of-care autonomous diagnostics of malignant growth. In IEEE Point-of-Care Healthcare Technologies, pages 128–131, Jan 2013.

[12] A. Basak, V. Ranganathan, and S. Bhunia. Implantable ultrasonic imaging assembly for automated monitoring of internal organs. IEEE Trans. Biomedical Circuits Syst., 8(6):881–890, Dec 2014.

[13] Nicholas J Hangiandreou. Aapm/rsna physics tutorial for residents: topics in us: B-mode us: basic concepts and new technology. Radiographics, 23(4):1019–1033, 2003.

[14] Vincent Chan and Anahi Perlas. Basics of ultrasound imaging. In Atlas of ultrasound-guided procedures in interventional pain management, pages 13–19. Springer, 2011.

[15] Arthur E. Weyman. Principles and practice of echocardiography, 2nd edition. 1994.

[16] John P Lawrence. Physics and instrumentation of ultrasound. Critical care medicine, 35(8):S314–S322, 2007.

[17] George Kossoff. Basic physics and imaging characteristics of ultrasound. World journal of surgery, 24(2):134–142, 2000.

[18] Aya Ibrahim, W Simon, Ahmet Caner Yüzügüler, Marcel Arditi, Jean-Philippe Thiran, and Giovanni De Micheli. 1024-channel single 5W FPGA towards high-quality portable 3D ultrasound platform. In PhD Forum at DATE 2017, number EPFL-POSTER-233779, 2017.

[19] Intel Corporation Altera. Designing with Low-Level Primitives. Intel Corporation - Altera, 4 2007.

[20] Rene Beuchat Sahand Kashani-Akhavan. SoC-FPGA Design Guide DE1-SoC Edition. LAP IC EPFL.

[21] Jørgen Grythe and AS Norsonic. Beamforming algorithms-beamformers. Squarehead Technology AS, Oslo, Norway, 2017.

[22] Antoine B Abche, Aldo Maalouf, and Elie Karam. A fast approach for ultrasound image reconstruction. In Signal Processing Algorithms, Architectures, Arrangements, and Applications (SPA), pages 207–212. IEEE, 2008.

[23] Sunera Kulasekera, Arjuna Madanayake, Dora Suarez, Renato J Cintra, and Fábio M Bayer. Multi-beam receiver apertures using multiplierless 8-point approximate DFT. In Radar Conference (RadarCon), 2015 IEEE, pages 1244–1249. IEEE, 2015.

[24] Alex Roman, Parisa Dehghanzadeh, Vida Pashaei, Abhishek Basak, Swarup Bhunia, and Soumyajit Mandal. An open-source test-bench for autonomous ultrasound imaging. In 2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS), pages 524–527. IEEE, 2018.

[25] Vida Pashaei, Alex Roman, and Soumyajit Mandal. Live demonstration: An open-source test-bench for autonomous ultrasound imaging. In 2018 IEEE Biomedical Circuits and Systems Conference (BioCAS), pages 1–1. IEEE, 2018.

[26] Vida Pashaei, Alex Roman, and Soumyajit Mandal. Conformal ultrasound transducer array for image-guided neural therapy. In 2018 IEEE Biomedical Circuits and Systems Conference (BioCAS), pages 1–4. IEEE, 2018.