Analyzing a Low-bit rate Audio Codec - Codec2 - on an FPGA

Peter Jamieson and Santhiya Sampath Kumar
Electrical and Computer Engineering
Miami University, Oxford, OH, USA
jamiespa@miamioh.edu

José Augusto M. Nacif, Ricardo Ferreira
Universidade Federal de Viçosa
Viçosa, Minas Gerais, Brazil

***Abstract*—Audio compression codecs are an important application in the Internet of Things (IoT) space where small sensing devices may gather voice signals, but then need to transmit the information to aggregating servers at a low cost. In this work, we implement and evaluate a hardware implementation of the Codec2 (a lossy speech compression codec) in Verilog Hardware Description Language (HDL) and map it to an Intel CycloneIV FPGA. We describe the details of our implementation approach, including how we represent data inside the hardware implementation and the associated cost of this implementation on an FPGA. We analyze our implementation compared to a microprocessor implementation of the original software to observe what performance we get on an FPGA versus a microprocessor. Our hardware implementation of Codec2 is qualitatively the same in terms of hearing the spoken transmission and has an error rate of 6.55 bits per frame (48 bits), and is 10 times slower and consumes slightly more power on a small FPGA compared to a microprocessor implementation. We provide this design as open-source hardware so that we have an HDL version of this application that can be mapped to both FPGAs and ASICs providing a research reference point, a baseline to optimize, and a described methodology to convert C to hardware for researchers.**

***Keywords*-Voice Compression, Lossy, FPGA, Hardware Description Language**

I. Introduction

Human speech is a common signal that is transmitted in a variety of applications in radio communications. Depending on the compression of the speech signal, there are trade-offs with the quality of speech versus the associated communication signal size. For example, signal size can be quantified in terms such as bits per sample and sampling rate in a digital communication sense. Larger signals require higher capacity communication links, more time for transmission, more power consumption to transmit the signal, and more storage capacity at both the transmission and receiving nodes. Compressing a signal reduces many of these factors but comes with a computational cost to perform the compression processing. For several applications, such as Internet of Things (IoT), one potential goal is to transmit information contained in speech with minimal power and computation resources at the expense of a low-quality speech signal [1]. Many audio compression codecs (COder and DECoder) have been created for this purpose, and we focus on Codec2 as our compression algorithm [2].

Codec2 is an open-source and patent-free audio codec that compresses audio to the lowest bit rate, at the publication date of this work. Codec2 is implemented in C, and the algorithm is typically executed on a microprocessor. One of the key benefits of this algorithm is that it has been developed without any existing commercial intellectual property restrictions, and researchers and developers can use this codec for any application without restriction.

This work implements a Register Transfer Level (RTL) version of Codec2 targeting a Field Programmable Gate Array (FPGA) in Verilog HDL. The key questions of this are to see what trade-offs in terms of computation speed and quality a hardware implementation of the Codec2 encoder on FPGA results in compared to the software implementation executing on a microprocessor. We hypothesize that a hardware implementation will be faster than a software implementation on a microprocessor due to custom parallel implementation capabilities possible on an FPGA. Additionally, the creation of this hardware core will allow future chip designers to test and implement Codec2 on other FPGAs and even ASIC versions of this core (if the market demand ever is high enough to justify the high cost of manufacturing ASICs). To verify our hypothesis, we implement and test a Codec2 encoder in Verilog and mapped it to Terasic's DE2-115 [3] prototyping board with an Intel Cyclone IV FPGA [4]. Our design is more efficient than an implementation running on a RaspberryPi, and its ARM processor, but the FPGA clock rate results in the design taking more time.

One additional contribution of this work for the IoT community is our approach maps the Codec2 design to Verilog HDL to target any FPGA or ASIC implementation by converting the C into simple synthesizable HDL. In this process, we provide a method to convert C into HDL. Alternatively, we might choose to use a High-level synthesis (HLS) tool that automatically converts C (with some modifications) into HDL, such as Legup [5] or tools provided by Intel and Xilinx. An HLS approach is viable, but those tools tend to be tightly coupled with the FPGAs that those companies provide, and understanding the hardware created by the HLS tool is more indirect. Instead, by creating a low-level Verilog implementation, our core is more portable to various FPGAs and ASICs and is open and accessible along similar lines to the main thrust of the Codec2 principles.

The remainder of this paper is organized as follows: Section II provides background information on audio compression techniques, Codec2, and previous work on the hardware implementation of Audio Codecs. Section III describes the details of our hardware implementation of the Codec2 encoder in Verilog, targeting an FPGA. Section IV discusses our

experiments, results, and analysis of the performance of our implementation. Section V discusses possible improvements and future work, and section VI concludes this work.

II. Background

In this section, we provide details on audio compression, Codec2, and describe existing hardware implementations of audio Codecs on hardware - mainly FPGAs.

A. Audio Compression

Audio compression techniques attempt to shrink the signal size via lossy or lossless compression techniques. Lossless compression techniques find redundant data in the encoding of the signal at the transmission side and are completely recoverable at the receiver side. A well-known algorithm in this domain that can be applied to a digital signal is the Lempel-Ziv algorithm [6]. Lossy compression, on the other hand, attempts to remove data that captures the quality of the signal, but it is not completely necessary to receive the information content. This makes the signal less like the original signal, but the signal contains the relevant information. For example, an original speech signal includes characteristics that would allow a listener to identify who the speaker is based on the acoustics, but lossy compression can remove these qualities to only transmit the information (the words spoken). A well-known algorithm for lossy audio compression is MP3 encoding [7], but for this case the information that is removed is less perceivable by humans, and from a human perspective audio signal seems to be of high quality.

The research areas of audio compression and data compression are vast areas of research. We suggest that unfamiliar readers start with Kavitha's recent survey on lossy and lossless compression techniques [8] and Uthayakumar *et. al.* survey on data compression techniques [9].

B. Codec2

The focus in this work is on lossy compression for speech. In particular, we focus on the Codec2 [2] implementation.

Codec2 is a patent-free and open-source speech codec software developed by Dr. David Rowe, who has implemented a number of low bit-rate speech compression and High-Frequency (HF) modems, with a focus on combining these technologies for digital voice over radio applications [2]. Codec2 is designed for lossy speech compression and operates at different rates, including 3200, 2400, 1600, 1400, 1300, 1200, 700, and 450 bit/s. There exist several competing codecs in this domain, such as MELP (a licensed codec) [10] and Speex (another open-source codec) [11] that operates in the range of 5000 bits/s.

Figure 1 shows the Digital Voice Radio System [2], which is an HF modem, and Codec2 was originally designed to be incorporated into this system. This system uses a microphone to capture the input speech, which is then fed into Analog to Digital converter (ADC), and the ADC converts the inputs and converts the analog speech into 8000 samples per second. The Codec2 encoder compresses the input samples from 8000 samples/s to a compressed form (for example, 2400 bits/s depending on the mode of the Codec2 software chosen). A Forward Error Correction (FEC) encoder takes 2400 bits/s from the Codec2 encoder, and after modulation, the data is transmitted over the radio channel. The decoding stages reverse the processing to generate an analog audio signal that can be listened to with a speaker.

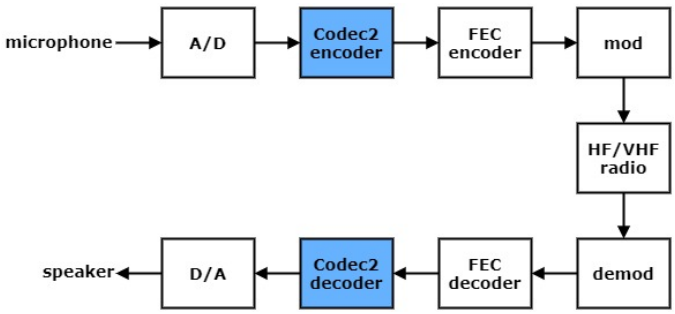

Fig. 1. Digital Voice Radio System

1) Codec2 Encoder: Our work focuses on developing a hardware implementation of the Codec2 encoder. We choose one compression mode to implement in hardware, which is the 2400 bits/s (called $MODE2400$ from now on). The other modes could be implemented from our example hardware codec, but the process of conversion is not trivial.

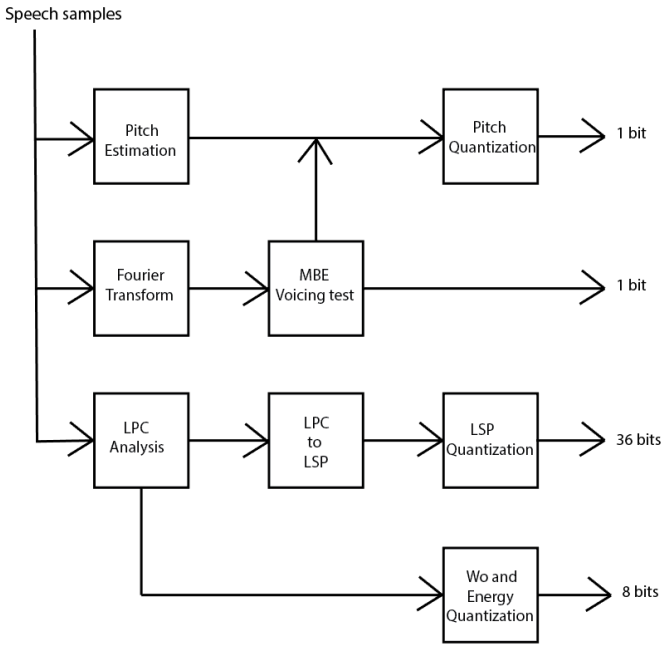

Fig. 2. Codec2 Encoder Block Diagram [2]

C. Hardware Implementations of Lossy Audio Compression Codec

Lossy compression codecs can be implemented on various computational system substrates. In this section, we provide

existing examples of state-of-the-art hardware implementations of lossy audio compression Codecs. Examples include an MP3 codec on a microprocessor [12], an MP3 codec on a DSP processor [13], and an image compression algorithm on a GPU [14]. Our focus is on FPGA implementations of compression audio codecs, and we briefly survey existing implementations in the research literature.

The MELP encoder which operates at 2400 bits/s, is implemented using Vivado HLS on a Zync-7 FPGA [15]. The algorithm is coded in C and is converted to Verilog with the HLS tool. This work then compares the area utilization and latency of the C-synthesis with the post-synthesis of the design to the Verilog RTL model. This work shows that the Verilog model utilizes fewer resources when compared to a C-Synthesis model.

The Advanced Multi-Band Excitation (AMBE) work is a Codec module that uses Intel's Quartus to synthesize a VHDL design targeting a Cyclone II FPGA [16]. This work demonstrates that an FPGA can replace both DSPs and microcontrollers in a traditional voice communication systems. Here, the FPGA acts as a Codec and AMBE chip controller. It also acts as interface support for the Subscriber Line Interface Card (SLIC) and handles the Dual Tone Multi-Frequency (DTMF) decoding. They demonstrated that an FPGA can be used to implement a complete speech system on a single chip.

The work on 'An Efficient Hardware Architecture of Codec2' implemented Codec2 decoder to process 1200 bits/s using Verilog and synthesized on Xilinx's Artix-7 XC7A100T FPGA [17]. Their approach was to implement a 'Very Long Instruction Word(VLIW)' architecture for parallelism and pipelining of each of the DSP algorithms. For the Fast Fourier Transform (FFT) and IFFT blocks, a butterfly operation is designed with pipelining scheme to reduce the computation time of 512-point FFT for every frame. They showed that this decoder implementation reduces processing time up to 20 times when compared to a Cortex-M4 CPU.

Our low-level RTL version in Verilog is a direct translation of the C code to HDL. We provide this implementation as an open-source codec that researchers can use as a base starting point for their work. This means there is plenty of potential to implement optimizations such as pipelining, parallel cores for separate frames, etc.

III. Codec2 Hardware Implementation

In this section, we describe Codec2 Hardware Implementation.

Rowe's open-source Codec2 is written in the C language[18] and can be executed on a Linux system with a microprocessor such as the RaspberryPi. To synthesize this design directly from the C to an FPGA, all the variables, constants, and functions in the code-base must be implemented in fixed-point representation as opposed to floating-point to achieve results with a reasonable number of hardware resources. For example, multiplication and division in floating-point representation involve operations that consume considerable amounts of FPGA logic resources. We used a fixed-point multiplier from "Opencores" Fixed point Math Library [19], and the floating-point multiplier is generated with Quartus' "IP Catalog" tool.

All of the other basic C operators such as division, square root, logarithm, etc. are also fixed-point representations, and all the existing variables and functions from C are converted to Verilog using this fixed-point representation. We chose to use a 32-bit fixed-point representation for our base representation in the system (meaning in some cases we convert to other formats), which includes the input samples, constants, and other variables.

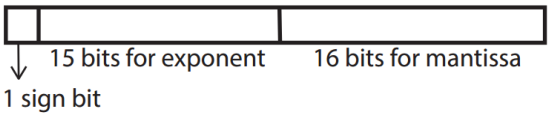

Fig. 3. 32-bit Fixed-point representation

Our fixed-point representation uses a customized 32-bit data representation that contains 16 bits for the exponent part (most significant bit for sign representation) and 16 bits for the mantissa, as shown in Figure 3. For certain operations, an 80-bit fixed-point representation is used for squaring values and for all variables where the decimal values exceed 15 bits in the exponent portion of the number. The 80-bit data representation contains 64 bits for the exponent and 16 bits for the mantissa. Whenever there is a demand for accuracy in the fractional part for some variables in the Verilog modules, the length of the exponent and mantissa are modified while retaining the length of 32-bit and 80-bit data representation throughout the design.

A. C code to FSM

The main step in converting the C codebase into Verilog is to implement the sequential computation in Verilog using Finite State Machines (FSM). The FSM model of the Codec2 encoder for one $FRAME$ is shown in Figure 4, and this figure shows the sequential implementation of the C code.

B. External and Internal IP Cores for Base Operations

For the basic set of mathematical operations in the Verilog design, we use existing IP and create our IP for base operations as described below.

For our fixed-point operations of addition and multiplication, we use cores from "Opencores" Fixed point Math Library [19]. The division and square root operations are created by implementing the Newton-Raphson algorithm [20] as an FSM. The logarithmic and trigonometric operations such as cos, $acos$ are written using a CORDIC algorithm [21] implemented as Verilog FSMs.

The full list of IP cores used by the encoder modules is given in Table I. In this table, column 1 shows the source of the core, column 2 shows the name of the Verilog module in our design, and column 3 shows the operation performed by the core.

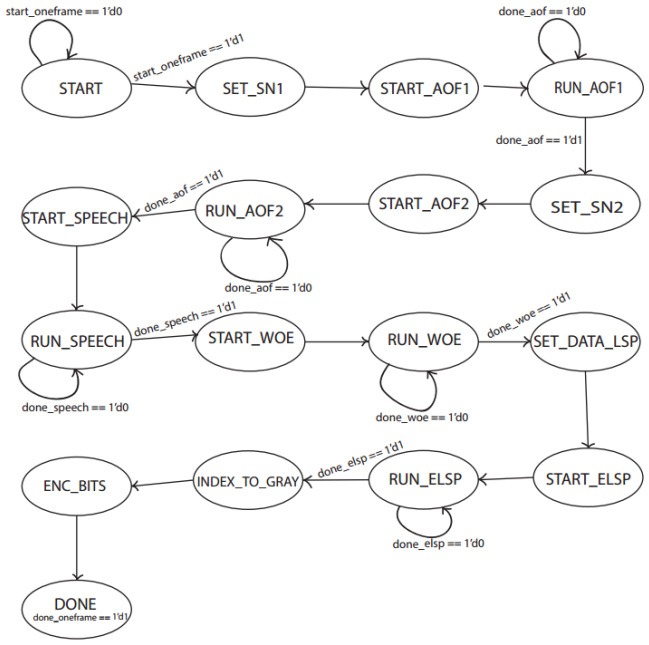

Fig. 4. FSM model of the Codec2 encoder for one FRAME (20 ms)

TABLE I
LIST OF IP CORES FOR BASE OPERATIONS IN OUR 32-BIT REPRESENTATION USED IN THE IMPLEMENTATION.

Source	IP Cores	Operation
Opencores	qadd	Addition
	qmult	Multiplication
Custom-built	fp_div_clk	Division
	fpgreaterthan	inequality check
	fp_log10	log10
	acosf	arc cosine
	cossin_cordic	sine and cosine
	fpmod	modulo
	fpsqrt	square root

C. Memory model implementation of the C code in Verilog

Arrays in C are implemented as on-chip Random Access Memories (RAMs) or Read Only Memories (ROMs) based on array usage. The Verilog modules for the memories are created with the "IP Catalog" available in Quartus. These on-chip memories can be customized in bit-width and number of memory locations due to the FPGA technology, and in theory can be implemented more efficiently than on a microprocessor, but there is very little gain from these optimizations.

D. Codec2 Modules

We convert a "C" block to a Verilog implementations. Once we've implemented a sequential version of each module, we then consider if it can be parallelized and how to achieve that parallelization with traditional optimizations such as loop unrolling, parallel operation execution, etc.

Table II shows each of the Verilog modules and the sub-modules of our design.Column 1 lists each of the encoder blocks, column 2 has the modules and sub-modules implemented in Verilog, column 3, 4, and 5 shows the respective utilization for the module in terms of Logic Elements, Memory bits, and 9-bit multiplier blocks.

IV. RESULTS

For our experiments, we evaluate our FPGA implementation of Codec2 encoder (which we call $FPGA\_C2$ from now on in this work) with that of the software implementation of Codec2 running on a RaspberryPi.

Our FPGA is a small FPGA available from Intel that is part of the DE2-115 prototyping board from Terasic [3]. The FPGA is a Cyclone IV EP4CE115F29C7 FPGA, and we use Intel's Quartus tool (Quartus Prime 16.1.0) to synthesize and program the FPGA. Our testing framework uses a combination of simulation and a tool called signal-tap, which is an on-board logic analysis tool [22]. The DE2-115 has a 50MHz clock, and our results are reported at this clock rate. Note, that the maximum frequency achieved with the design is no significantly higher, and therefore, our choice was to produce results at this fixed clock frequency without using the Phase-locked Loops available on the FPGA. When synthesizing our design for analysis, we use the following optimization parameters in the Quartus tool: "Optimization Technique" for area or speed is chosen as "Balanced"; "Optimize Timing" is chosen as "Normal".

We compare our FPGA implementation to a RaspberryPi 2 that includes the quad-core ARM Cortex-A7 CPU and runs the Raspbian Linux distribution. This processor has a 900 MHz clock frequency.

With our experiments, we address the following questions:

1) What is the hardware utilization of our proposed implementation $FPGA\_C2$?
2) How does $FPGA\_C2$ perform against software implementation of Codec2 in RaspberryPi in terms of quality?
3) How fast are the major modules of $FPGA\_C2$ compared against the RaspberryPi implementation in terms of time?
4) How do the major modules of $FPGA\_C2$ compare against RaspberryPi implementation in terms of clock cycles?
5) What is the energy consumption of the $FPGA\_C2$ compared to the RaspberryPi?

A. FPGA Resource Utilization

The resource utilization of the FPGA for our Codec2 encoder processing 150 $FRAME$s (which is 3 seconds of input speech) is shown in Table III. In this table, column 1 lists the FPGA resource type, column 2 shows the total available resource count on the Cyclone IV EP4CE115F29C7, column 3 shows the number of those resources used in our synthesized design, and column 4 gives a utilization percentage of that resource (divide column 3 by column 2) for the Cyclone FPGA. In the previous/section, we provided resource

TABLE II
RESOURCE UTILIZATION OF THE ENCODER BLOCKS

Encoder Blocks	Modules & Sub-modules	Logic Elements	Memory bits	Multipliers
FT	fft	1,053	40,960	16
Pitch estimation	nlp	8,995	277,504	174
	fft_nlp	2,379	90,112	40
	post_process_sub_multiples	1,591	0	28
Pitch Refinement	two_stage_pitch_refinement	5,182	0	40
	hs_pitch_refinement	1,642	0	8
MBE Voicing test	estimate_amplitudes	2,277	0	16
	estimate_voicing_mbe	3,933	16,384	40
LPC Analysis	speech_to_uq_lsps	18,287	15,712	138
	levinson_durbin	7,594	0	60
LPC to LSP	lpc_to_lsp	6,643	0	48
LSP Quantization	encode_lsp_scalar	3,091	0	80
	quantise	2,187	0	0
W0 and Energy Quantization	encode_WoE	6,123	0	76
	compute_weights	963	0	16
	find_nearest_weighted	2,053	0	32

TABLE III
CYCLONE IV EP4CE115F29C7 RESOURCE UTILIZATION OF CODEC2 ENCODER TO PROCESS 150 $FRAMES$

	Available	Used	Utilization
Logic Elements	114,480	60,890	53%
I/O Pins	529	424	80%
Memory Bits	3,981,312	1,230,528	31%
9-bit DSPs	532	532	100%

utilization details for each of the design components. Note that these utilization numbers can be improved by analyzing the design and determining which functional units can be shared. Also, in the last row of the table we show that 100% of the multipliers are used on the FPGA, and to achieve this we ensure that there are no multipliers that are converted to soft logic multipliers by implementing some multiplier sharing to achieve this for our particular Cyclone IV FPGA. We also note that the high I/O pin usage is relative to the module that implements the Codec2 and does not include a buffer memory to store the speech, which requires large ports to pass the speech samples in.

B. Qualitative and Quantitative Comparison of Encoding Results

Because the FPGA implementation uses different functional units and different number representations, we expect that there are differences in the quality of the compressed signals that might degrade the functionality of the encoder. In this section, we do a qualitative comparison and an error bit-rate comparison of our implementation to the software version.

The module codec2_encoder_2400 processes 900 bytes of the input data, which contains 150 frames and gives an encoded output of 7200 bits (2400 bits/s). The output bits from the Verilog module are copied to a ".bit" file. This encoded file is then decoded using the software version of Codec2 decoder running on a Linux system to generate a ".raw" file. Finally, the decoded ".raw" file from codec2_encoder_2400 is compared with the decoded file of the same input data from codec2 software running on a Linux system.

We took two samples of speech to verify our Codec2 written in Verilog. The speech samples are "hts1a.raw" and "hts2a.raw", which are male and female voice samples that contain 3-seconds of speech (150 frames). When listening to the decoded samples of the encoded bits from the Verilog module, they are intelligible when compared to the original samples. The source code and the speech files are uploaded on our GitHub repository for others to listen to [23].

Figures 5 show the Codec2 output of the speech samples "hts1a.raw" when processed on the software version. Figures 6 show the Codec2 output from the software Codec2 processing the encoded bits from Verilog implementation of the Codec2 encoder. We have stacked the software and hardware figures on top of each other so the reader can do an easy visual comparison of the sound waveforms.

Our design loses some accuracy in the 16-bit fractional part of the fixed-point representations while performing repetitive multiplications and additions in the modules for Fourier Transform and auto-correlation. So, for each frame of encoded data, which is 48 bits, we can see that some bits are off from the expected encoded bits in each frame. The average number of bits that vary per frame is 6.55 bits/frame. The bit error rate for 3 seconds of input data (900 bytes) is 13.6%. Note that this error rate is a direct comparison between the software approach and the hardware approach where the hardware incurs more errors due to the number representation choice. Note, however, that as demonstrated in the cognitive results, the speech is still recognizable.

C. Performance Results

For this analysis, we compare the software and hardware implementations in terms of speed. As mentioned, the clock

frequency of the FPGA is 50 MHz, while the clock frequency of the ARM processor on the RaspberryPi is 900 MHz. So, in terms of clock speed, the processor is 18 times faster than the FPGA we are using. However, a hardware implementation is customized for the application, and it may outperform the microprocessor.

In the Verilog implementation, the time taken to process 3s of input speech by the Codec2 encoder is calculated by using clock tick counters to record the number of clock cycles for the execution of the codec2_encoder_2400 module. Then, given the FPGA's 50 MHz clock, the counter variable multiplied by 20 ns gives the execution time for processing 3s input data. When we run the same encoder block to process the same input data of 3s in RaspberryPi 2, we use the "system time" to compute internal timing results [24]. These two measurements allow for this comparison.

Since the modules for the other encoder blocks of LPC Analysis, LPC to LSP, LSP Quantization, Wo & energy Quantization execute faster in FPGA C2 when compared to the corresponding functions in 'C' processed on the ARM processor, we could see that the module codec2_encoder_one_frame which processes one FRAME of input, executes faster on the FPGA compared to the Arm processor.

The time of execution to process 150 $FRAMES$ on FPGA is, currently, 3.4466 seconds with the 50 MHZ clock while the time taken in RaspberryPi is 0.3319 seconds with a 900 MHz clock. When comparing the ratio of clocks to execute, $FPGA\_C2$ implementation is 1.73 times faster than the Codec2 encoder on RaspberryPi. If we have to speed up the $FPGA\_C2$ design further, we would need to pipeline our architecture so that we can more quickly process the computation. This is doable since there is little dependence between frames that are processed.

D. Power Analysis

Our FPGA implementation is more efficient in terms of clocks to process, but due to clock speed of the Cyclone IV the execution time is a factor of ten slower than the RaspberryPi. In this section, we look at the power consumption of the two applications.

To estimate the power and energy consumption of Codec2 on a RaspberryPi, we use the estimates from Kaup *et. al.* [25], which is for the RaspberryPi original model B. Their CPU estimate at approximately 90% of utilization is close to 1.75 Watts. Therefore, the energy usage is 0.5808 joules for the compression of the 3s frames to process. Note that this estimation is just for the CPU and none of the RaspberryPi system, which we will follow for our FPGA power analysis.

For our FPGA analysis, we use the power analyzer tool available in Quartus with default estimation settings. The tool reports a 203.52 milliWatt power consumption, and the total energy to analyze the 3s of speech is 0.7013 joules. This is greater than the energy consumption of the microprocessor implementation.

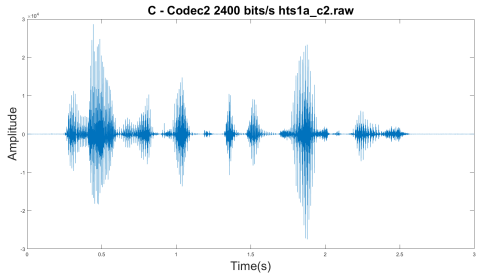

Fig. 5. Codec2 output of the hts1a.raw processed in C

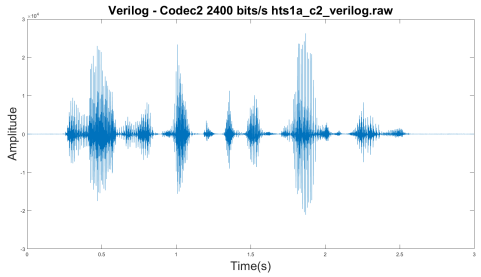

Fig. 6. Codec2 output of the hts1a.raw processed in Verilog

V. Discussion

A. Analyzing our Results and Improvements

The hardware design, we have created, is 10 times slower and consumes 1.2 times more power than the microprocessor software-based system. We, however, are very satisfied with these results as our hardware version is a direct conversion of the C code and is a base hardware version that can then be expanded on by other researchers. This means that it would not be too difficult to improve our hardware version to beat the head-on comparison to a microprocessor with optimizations and improved hardware. Additionally, we note that our processing time is right on the verge of being capable of real-time processing (3.4466 seconds for 3 seconds of speech audio).

In particular, there are two key techniques to improve our system. First, by pipelining our design the time to process the frames, arguably, decreases by a factor of the number of pipeline stages. This is, relatively, easy with FPGAs since flip-flops are in abundance due to the FPGA architecture, and with just two stages we would be able to encode the design in real-time. Additionally, pipelining will, likely, improve energy consumption [26].

Another approach to improving the hardware implementation is using a better FPGA as the one used in this analysis is an older FPGA as that is what we have available. We could, easily, improve both device and increase clock speed to improve the processing time. Energy consumption will remain constant other than the change in the power properties of the devices.

B. Implementing other Codec2 modes

Codec2 operates on different compression rates of 3200, 2400, 1600, 1400, 1300, 1200, 700, and 450 bit/s. We chose to implement Codec2 encoder operating at 2400 bits/s because of the following reasons.

1) The other available open-source speech Codecs are MELP, AMBE, and LPC-10. They operate in the range of 2000 to 2400 bits/s. So, the $MODE2400$ of the $FPGA\_C2$ would be much suitable if we want to compare the hardware implementations of other Codecs.
2) The software implementation of $MODE2400$ has most of the functions which can be reused for the other modes lower than $MODE2400$. So, building the Verilog modules for
$MODE2400$ allows having most modules that can be utilized while implementing the Codec2 encoder in other modes. The list of modules to be altered is discussed below.

As our work is based on the $MODE2400$, some modifications should be taken to convert to other modes in the Codec2. The Verilog modules for the encoder blocks such as FT, Pitch estimation, Pitch Quantization, MBE Voicing test, LPC analysis, and LPC to LSP can be reused when converting to other bit rates. Also, the IP cores from the "OpenCores" and the custom-built cores can be reused without any modifications. All the custom-built IP Cores have parameters that can be altered to any other bit-width of the fixed-point representation.

The significant changes will focus on a subset of modules, including LSP Quantization, Wo, and Energy Quantization blocks, which must be created according to the C implementations for each mode. In each of these cases, it took approximately 15 days to implement these cores, and we would expect similar design time to make new ones with a slight reduction given that our implementations could be used as templates.

VI. Conclusions

In this work, we have provided a low-level implementation of the Codec2 in Verilog HDL implementation. We implemented this design on an Intel FPGA and provided implementation results for the resource consumption, speed, and power consumption comparing some of these results to an implementation on a RaspberryPi. Our main interest was to provide this design as an open-source hardware version for other researchers and determine what performance a simple hardware version would have to the software implementation. Our design is available at: https://github.com/santhiyaskumar/FPGA_Codec2Encoder.

References

[1] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards blockchain-based auditable storage and sharing of iot data," in *Proceedings of the 2017 on Cloud Computing Security Workshop*. ACM, 2017, pp. 45–50.

[2] D. Rowe, "Codec 2," http://www.rowetel.com/?page_id=452, Accessed: 06-30-2019.

[3] "Altera de2-115 development board," https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/terasic-inc-/board/altera-de2-115-development-and-education-board.html, Accessed: 07-07-2019.

[4] "Cyclone IV FPGA," https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-iv.html, Accessed: 07-07-2019.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 33–36.

[6] "Lempel–ziv algorithm," https://en.wikipedia.org/wiki/Lempel-Ziv-Welch, Accessed: 07-07-2019.

[7] "MP3," https://en.wikipedia.org/wiki/MP3, Accessed: 07-07-2019.

[8] P. Kavitha, "A survey on lossless and lossy data compression methods," *International Journal of Computer Science & Engineering Technology*, vol. 7, no. 03, pp. 110–114, 2016.

[9] J. Uthayakumar, T. Vengattaraman, and P. Dhavachelvan, "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications," *Journal of King Saud University-Computer and Information Sciences*, 2018.

[10] "Melp codec," https://www.vocal.com/speech-coders/melp/, Accessed: 07-07-2019.

[11] "Speex : A free codec for free speech," https://www.speex.org/, Accessed: 07-07-2019.

[12] S. Gadd and T. Lenart, "A hardware accelerated mp3 decoder with bluetooth streaming capabilities," *Master of science Thesis*, 2001.

[13] S. Hong, B. Park, Y. Song, H. See, J. Kim, H. Lee, D. Kim, and M. Song, "A full accuracy mpeg1 audio layer 3 (mp3) decoder with internal data converters," in *Proceedings of the IEEE 2000 Custom Integrated Circuits Conference (Cat. No. 00CH37044)*. IEEE, 2000, pp. 563–566.

[14] L. Santos, E. Magli, R. Vitulli, J. F. López, and R. Sarmiento, "Highly-parallel gpu architecture for lossy hyperspectral image compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 6, no. 2, pp. 670–681, 2013.

[15] M. Koushik, S. Shivanagi, J. Qumar, J. Yadav, and D. Saravanan, "Implementation of melp encoder on zynq fpga using hls," in *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*. IEEE, 2017, pp. 87–91.

[16] K. Mahawar, V. Kumar, and H. Gupta, "Design and implementation of ambe based voice codec module over custom fpga platform," in *2012 International Conference on Computing, Communication and Applications*. IEEE, 2012, pp. 1–5.

[17] S. Wisayataksin, "An Efficient Hardware Architecture of Codec2 Low Bit-rate Speech Decoder," in *2019 5th International Conference on Engineering, Applied Sciences and Technology (ICEAST)*. IEEE, 2019, pp. 1–4.

[18] "Codec2 software," https://svn.code.sf.net/p/freetel/code/codec2/branches/, Accessed: 01-12-2019.

[19] "Fixed-point Math Library," https://opencores.org/projects/verilog_fixed_point_math_library, Accessed: 07-07-2019.

[20] "Newton-Raphson Algorithm," https://en.wikipedia.org/wiki/Division_algorithm, Accessed: 07-07-2019.

[21] "CORDIC Algorithm," https://people.sc.fsu.edu/~jburkardt/cpp_src/cordic/cordic.html, Accessed: 07-07-2019.

[22] "Signal Tap Logic Analyzer," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/signal.pdf, Accessed: 01-17-2019.

[23] S. S. Kumar, "Verilog Codec2," https://github.com/santhiyaskumar/FPGA_Codec2Encoder, 2020.

[24] "sys/time.h - Time types," http://manpages.ubuntu.com/manpages/trusty/man7/sys_time.h.7posix.html, Accessed: 01-17-2019.

[25] F. Kaup, P. Gottschling, and D. Hausheer, "Powerpi: Measuring and modeling the power consumption of the raspberry pi," in *39th Annual IEEE Conference on Local Computer Networks*. IEEE, 2014, pp. 236–243.

[26] S. J. Wilton, S.-S. Ang, and W. Luk, "The impact of pipelining on energy per operation in field-programmable gate arrays," in *International Conference on Field Programmable Logic and Applications*. Springer, 2004, pp. 719–728.