Washington University in St. Louis

James McKelvey School of Engineering

William Silberstein
McKelvey School of Engineering
Washington University in St. Louis
wsilberstein@wustl.edu

May 6, 2022

Richarduino v4: An Exploration of the Conversion of Audio to and From Digital and Analog Signals Through Hardware Design

Author: William Silberstein

Washington University in St. Louis
CSE 462: Computer Systems Design

Project Completion: April 21, 2022
Paper Completion: May 6, 2022

Abstract - This paper describes the system of the Richarduino v4: a system for converting audio to and from digital and analog signals. The system was developed as an educational project, but it can be used as a microphone as well as a digital synthesizer. The paper itself will go over the components of what makes up the Richarduino, as well as dive into how all the components work together. A deep explanation will be given explaining the development process including the microcontroller system architecture, how components were built using VHDL, as well as the various testing strategies used. In addition, a brief overview of the challenges and bugs during development will also be explained.

I. Introduction

The Richarduino v4 is a design which builds off of the previous versions of the Richarduino, as well as designs researched in CSE 362. The purpose of this Richarduino version (version 4) is to convert audio samples from an analog source such as a microphone and RF connector jack to a digital value. In addition, the Richarduino is able to convert the audio digital signal back into an analog signal which can be output to an external source. Furthermore, digital signals from the Richarduino can also be output through a UART serial module, allowing for serial communication with the Richarduino through an external device. The design consists of a microcontroller with various memory and output modules embedded in the FPGA. The microcontroller is connected

to an XADC input for converting analog audio from a microphone and digitizing it. The design features a digital to analog converter connected to an RF jack to output the converted digital to analog audio. There is an additional RF connected that can connect to the analog pin of the XADC in order to convert analog audio from an external source and digitize it. It also comes with additional storage for storing programs or additional data. A host downloader and boot monitor program was created in order to debug and load programs onto the device itself. Finally, a demo program was created and loaded onto the device, fully displaying the results of the audio conversion process.For the rest of the document the Richarduino will refer to the Richarduino v4.

II. Richarduino Microcontroller System Architecture

The system design of the Richarduino consists of many modules embedded inside the Xilinx Artix-7 FPGA. These modules include an RSRC microcontroller, EPROM, SRAM, output pins module, UART module, DCM, I2C module, and an XADC module [1]. All of these modules are connected to a 32 bit address and data bus as well as an external reset signal. A block diagram of all the components can be found in figure 1. These components, other than the RSRC micro controller, DCM, and reset synchronizer are mapped to a certain address space. The system contains a 32 bit address space that ranges from 0x00000000 - 0xffffffff as shown in figure 2 [2]. When the address on the address bus contains an address corresponding to any of the modules, an appropriate chip enable signal is given to the component. Because the device is a word based

system (4 bytes), all addresses are 4 byte aligned. This means that the bottom two bits in any address will not matter. An in depth explanation of these modules are as follows.

RSRC Microcontroller

The heart of the Richarduino design is an RSRC microcontroller embedded into the Xilinx FPGA. The RSRS is a 32 bit multi use microcontroller that can execute specific assembly instructions [3]. This RSRC processor can compute any common mathematical bitwise operation, as well as load and store data to and from memory. It comes with a fully functional branching system for programmatic conditional statements as well as looping functionality. The RSRC processor is also responsible for asserting the respective READ/WRITE commands for specific actions of the other components in the Richarduino. Components of the RSRC microcontroller are given below and a visual representation of the RSRC is shown in figure 3.

Program Counter — A register to keep track of the current program counter

A Register — A register to hold an input going into the ALU

C Register — A Register to hold the output from the ALU

Shift Counter — A Register to hold the current shift counter

Register File — 32 independent multi use registers

MA Register — A Register to hold an output address

MD Register — A Register to hold data to output to the bus or input data from the bus

Instruction Register — A Register to keep track of the current instruction

Con Bit — Determines the specific condition for branching

Control — A control system to determine all signals in the RSRC

All of these components work together to make up the RSRC. The RSRC processor uses a simple fetch execute structure for executing specific binary instructions. When an instruction is loaded and read, the appropriate control signals will be asserted according to the control component's current state and instruction. With the appropriate control signals asserted, the rest of the components can act accordingly. An example load instruction execution is shown in figure 4. Since the processor waits for a DONE signal after a corresponding READ/WRITE signal, a READ or WRITE operation to an address not allocated in the device will cause the processor to stall. This means that the RSRC is designed to continue executing instructions until a STOP instruction is found, or the program counter exceeds the size of the allocated memory. One thing to note is that the RSRC was researched extensively in CSE 362 and was designed in a previous Richarduino version.

EPROM

The Richarduino consists of 4KB of EPROM storage mapped to the address space 0x00000000 - 0x00001000. The purpose of the EPROM component is for storing the boot monitor program of the device. The boot monitor program in EPROM will be discussed in section 4. The binary code for the boot monitor is hardcoded into the EPROM as a binary instruction. When an address from EPROM is asserted, the device will put the instruction at the address location onto the bus. This is used for the RSRC to pull the instructions to execute. When the Richarduino first powers on, code starts

executing from address 0x00000000 which is mapped to the first address space of EPROM.

SRAM

The Richarduino, contains an additional 4KB of SRAM for storage mapped to the address space 0x00001000 - 0x00002000. When an address for SRAM is asserted, because memory is 4 byte aligned, bits 11- 2 correspond to the address in SRAM where the data is stored [4]. The SRAM will store data to an address or read data from that address when the RSRC asserts an appropriate READ/WRITE signal. The main purpose of the SRAM is to hold additional data such as a downloaded program or parameters when the registers are all full. This allows for a program to be downloaded and executed from the Boot Monitor as discussed in section 4.

DCM

A DCM is also included in the Richarduino for the purpose of generating a correct clock signal. The original source clock given by the Xilinx FPGA is a 12 MHz source clock. The DCM component takes that 12 MHz source clock and generates a 48 MHz clock for the rest of the components. This speed of 48MHz is very important for the Richarduino to keep timing. It is noted that the DCM is not mapped to the address space and is always turned on, unless the external reset signal is applied.

PINS Module

The Pins Module is a component for outputting digital signals using the 44 digital I/O Pins on the Cmod. The Pins Module is mapped to the 16 byte address space 0xfffffff0 - 0xffffffff. Although the Pins Module was never used for this project, a single digital IO pin is connected to an offboard LED allowing for control over the LED light.

UART Module

The UART component is responsible for transmitting and receiving serial messages. It can communicate with an external device using a serial communication of 115200 Baud rate (bits per second). The UART is mapped to a 16 byte address space from 0xffffffe0 - 0xffffffef allowing for four registers to reside in this space [5]. The TX_BUSY register is mapped to the address 0xffe0. The value of this register determines whether the UART transmitter is busy. A value of 1 in the bottom bit signifies that the transmitter is busy, while a value of 0 signifies that the transmitter is ready. The Output register is mapped to the address 0xffe4. The bottom byte in this register holds the data that will be output from the UART through the transmitter. The RX_DATA register, which is mapped to the address 0xffe8, supplies the RX_DATA_FLAG in the least significant bit when read. This flag determines whether the UART is ready to read in data. The Input register holds the serial data from an external device when communicating with the UART. This register is mapped to the address 0xffec. For a READ and WRITE transfer, serial communication can only transfer one byte at a time. Since the Richarduino is a word based process (4 bytes), a READ or WRITE will require the use of four byte transfers. During a serial READ operation, the input data will only be valid on the RX_DATA_VALID flag when the

UART is finished with the transfer [6]. Example simulation is shown in fig. 5. For both the Transmitter and Receiver, a counter is started when the transaction first begins. This counter is used to synchronize the input/output bits with the external serial device as per the UART specification [7]. Every multiple of 417 on the counter, a single bit is either taken in or output to the serial bus depending on the type of operation. This magic number of 417 is due to the 48MHz source clock and a 115200 baud rate. By dividing the source clock speed with the clock rate time, 416.666 is the resulting time. It is worth noting that the action of placing a single bit or receiving a single bit occurs at the midpoint of the associated bit's timeslot. This allows for a bit of leeway when the bit is either changed or read. The UART was mostly used to communicate with an external boot monitor, but it is capable of communicating with any serial compatible external device with a suitable Baud Rate. The UART Module was originally designed for a previous version of the Richarduino.

I2C Module

The I2C Module's purpose is to act as an intermediary between the AD5311 Digital to Analog Converter and the RSRC. The I2C module is mapped to a 16 byte address space from 0xfffffff0 - 0xffffffdf. The module contains three active registers. The BUSY Flag register. Which is mapped to the address 0xffd0, contains the I2C_BUSY_FLAG in the least significant bit. When this flag is currently active, the I2C is currently in a WRITE operation and must wait until the flag is set to zero before it can output another value. The Output register is mapped to the address 0xffd4. This register holds two bytes of data to send to the I2C. It is worth mentioning that only the bottom 10 bits of the

output register is the data that will be output to the DAC, since the AD5311 DAC can only transfer 10 bits at a time [8]. The remaining 6 bits are reserved for possible address options, and powerdown options. The third active register is the ACK register. This register is meant to record and save ACKS from the DAC. Each ACK is recorded as a single bit and is placed in the respective location based on order. The ACK register is mapped to the address 0xffd8. The final register in the I2C module is a reserved register for future use. The I2C Module works in a very similar manner to the UART Module. A counter is started when a READ operation begins. Using the same number as the UART, 417, the SCL and SDA bus can be asserted according to the I2C specification [9]. Although the I2C specification allows for two way communication, the design implemented in the Richarduino only allows for a write to the I2C DAC. As shown in figure 6, when an appropriate address is asserted onto the bus and when the appropriate WRITE signal is output from the RSRC, the I2C Module will output the bottom ten bits of the data bus to the DAC. The I2C WRITE operation writes to the DAC using three separate writes. The first write outputs the address, and the second and third writes write the ten data bits to the DAC. The address of the DAC is asserted as 0001101. One thing to note is that the last address bit is set to a one, since the address pin in the DAC is tied to VDD on the board. When the transfer is complete, users can read from the ACK register to debug the design and make sure that the device is receiving the appropriate ACK response defined in the I2C specification. A valid ACK response will give a value of 0x7, where each of the bottom three bits is the corresponding ACK response.

XADC Module

The XADC Module is another intermediary component that acts as a translator between the onboard XADC and the RSRC. The XADC Module is mapped to a 512 Byte address space starting from 0xfffffc00. The 128 configuration and status registers, as shown in fig. 6, of the XADC can be read or written to using this address space. The purpose of the XADC Module is to collect the converted digital audio signal from the XADC and output it to the bus when an appropriate address is put on the bus and a READ signal is asserted. Although any status or control registers from the XADC can be read and output onto the bus, it is noted that only bits 15 - 4 of the response will be a valid audio signal as shown in figure 8 [10]. The bottom 4 bits are meant to be used for calibration purposes, but were not used in this design. The XADC Module allows for single channel communication with the onboard XADC, but can be reconfigured to read from the microphone or RF jack. The address of the microphone register is 0xfffffc50 and the RF jack register is 0xffffc70 assuming the appropriate channel is selected for the XADC and the appropriate pins are set up. The XADC Module also allows for WRITE commands to the configuration registers for setup or debug purposes. The configurations of the XADC that was used for this device is as follows. The device was in single channel mode. The XADC used the source clock of 48MHz with an ADC Conversion Rate of 44.1 samples per second (sampling rate of CD audio). Because of the actual constraints of the XADC, a conversion rate of 43.96 samples per second was actually used. No alarms or calibration was used with the XADC. The XADC allows for the use of two channels: VAUXP12/VAUXN12 and VAUXP4/VAUXN4, where channel 12 corresponds to the RF jack input and channel 4 corresponds to the microphone input. The XADC is also in

continuous mode, meaning that analog inputs to the channel selected are automatically converted to a digital signal when the previous conversion is finished. This allows for the quickest method of converting the analog audio signal to a digital signal.

III. Richarduino Microcontroller FPGA VHDL Overview

The RSRC as well as the rest of the components that make up the Richarduino are all embedded inside of a Xilinx Artix-7 FPGA. The FPGA itself is embedded onto the Digilent Cmod A7 as shown in figure 9. The board also contains a 48 pin DIP connector with 44 Digital I/O pins and w Analog input pins (0-3.3V) [11]. Figure 10 displays the pin layouts of the Richarduino [12]. Every component embedded inside of the FPGA was coded in the hardware description language VHDL [13]. All of the components are contained within a unified component called Testbench. This structure creates a hierarchical design with Testbench at the top level, with components underneath and components underneath those components as well. Testbench itself contains pins for external communication outside of the FPGA, as well as all of the chip enable, chip write enable, chip read enable, and the done signals for each of the components. Testbench deals with the address decoding described earlier for the chip enable of the lower components. The rest of the components described in the earlier section are port mapped into Testbench and all signals are assigned accordingly. One thing to note is that some of the components inside Testbench were created automatically using the Xilinx IP Wizard. These components included the DCM [14], FIFO memory for the UART, and the onboard XADC [15]. In addition to automatically generating components

using the Xilinx IP Wizard, the EPROM VHDL file itself was generated using a file generator. With only a binary program file, the eprom.exe tool is able to create a functioning EPROM file with the binary program hardcoded into the memory [16]. This tool made it very easy to load different programs into EPROM. In addition to the VHDL, a constraint file was also needed to connect the VHDL components to the physical pins within the FPGA [17]. The constraints file also defined attributes of pins such as the pin standard, drive strength, slew rate, and introduced internal pulldown resistors on all of the IO pins and pullup resistors on the SDA and SCL pins.

IV. Richarduino Boot Monitor

Included with the Richarduino project is a boot monitor program. The purpose of the boot monitor program is to allow for serial communication with the Richarduino for testing and demo purposes. The monitor allowed for four different actions:

Poke — Sends 4 bytes of data to a specified 4 byte address

Peek — Retrieves 4 bytes of data from a specified 4 byte address

Version — Returns the version number of the Richarduino

Program — Downloads a binary compatible program to the Richarduino

The boot monitor program is stored in EPROM. This means that when the Richarduino first turns on, the boot monitor program will run continuously until another program is downloaded into SRAM, or the world explodes. The boot monitor program begins by

continuously checking if the RX DATA FLAG is not busy. Once the RX DATA FLAG is unset, the program checks if the UART input is any of the four commands Peek(R), Poke (W), Version(V), and Program (P). These commands will be represented by their ascii hex values in the UART input register. Once the letter for a command is found, then the corresponding command is executed. The three commands version, poke, and peek were used extensively for debug and testing purposes, and were not used in the actual final product. Version can be an easy way to debug the design to make sure that it is working properly. Poke was used to test writing a value to an address such as the I2C bus. Peek can be used to check the value at a certain place in memory. One example may be to peek the value of the XADC register to check the converted digital value. The program instruction will store the binary machine instructions into SRAM starting from the first address. One important aspect of the program command is that once the program is finished downloading into SRAM, the Richarduino will then start executing that program instead of the boot monitor program. This allows for the user to create and download their own binary compatible program for their own purpose. One note is that the program cannot exceed the size of SRAM. If the program exceeds 4KB of memory, then the Richarduino will stall and no commands will be executed until the device is reset.

V. Host Downloader

The Host Downloader is a GUI that can interact with the Richarduino using the four commands described in the previous section. The main component of the Host

Downloader is a serial connection script programmed in Python. The serial connection script utilizes the serial.py library [18] to automatically establish serial parameters such as parity, baud rate, and port. The script defines a Richarduino object which comes with four methods for sending and receiving serial data with the Richarduino. The methods can be called by using the syntax Richarduino.peek(…) or Richarduino.poke(...). The second python file creates a local server using Flask [19]. When the endpoints “/peek”, “/poke”...etc are received, the server will create a Richarduino object with the current com port, baud rate, and timeout time, and run the corresponding Richarduino method. The Host Downloader also consists of a client side UI created in HTML, CSS, and Javascript. Users can input the parameters for functions like Peek and Poke and just click the corresponding button to complete their action. It is worth noting that if the Richarduino is not running or the Boot Monitor program is powered off, the Host Downloader will return a null value for Peek and NaN for the Version command. To run the Host Downloader, the user’s machine must have Python3 and Google Chrome installed, and be running a version of Windows. A start file, “run.bat”, is provided to allow for the user to easily run the virtual environment, Flask server, and client side front end without the need for the user to have knowledge in these concepts. The start file will also automatically install all dependencies for the user. The end result should be a UI running in Google Chrome as shown in figure 11.

VI. Richarduino Demo Shield Architecture

In addition to the Cmod A7, the design also features a surrounding shield. The shield is connected to the Cmod through a 48 DIP connector on the shield and features a microphone amplifying circuit, an I2C interface circuit, and a voltage regulator as seen in Figure 12 [20]. The microphone [21] is connected to a MAX4466 amplifier [22] in order to amplify the signal from the microphone. The output of the amplifier goes into analog pin 0 of the Richarduino. This allows for a stronger input to the XADC. In addition, an RF jack is connected to analog pin 1 of the Richarduino allowing for an additional source of analog audio to go into the Richarduino. The shield contains a MIC5255 linear regulator in order to output a constant 3.3V to the rest of the components on the shield [23]. The shield also features an LED connected to pin IO1 on the Cmod. The final component on the shield is an I2C interface bus connected to an AD5311 Digital to Analog converter. It's worth noting that the SDA and SCL lines are pulled up. The output for the DAC is connected to an RF jack allowing for the analog audio signal to be output to an external device. Another note is that the address line into the DAC is connected to power. This is addressed in the address that is sent to the DAC on an I2C WRITE command. It's also worth noting that the Cmod, microphone, and RF jack are connected to the opposite side of the board than the rest of the components. A complete schematic of the circuit described is shown in figure 13. In addition to the schematic for the circuit, a pcb layout was also created for fabrication of the board. This layout can be seen in figure 14.

VII. Richarduino Shield Demo Program

Since the purpose of the Richarduino is to digitize an analog audio signal as well as convert a digital audio signal into an analog signal, a way to test out this functionality is needed. For testing out the I2C design, one can write a high value such as 0x3ff to the DAC and measure the output voltage on the RF jack to make sure it is 3.3V using a voltage measuring device such as a Multimeter. In addition, one can debug the XADC design by generating a tone into the microphone and continuously running the POKE command through the Host Downloader and see how much the value changes by. Since this method of checking the digital value of the audio signal is very imprecise, a new method was created to test out this design. The Shield Demo Program is a binary program that can be downloaded through the Host Downloader into the Richarduino. While this program is running, the Richarduino will continuously pull the converted analog signal from the XADC, and write that value to the I2C. In this way, audio will be converted from an analog wave, into a digital signal, and then back into an analog signal through the DAC. This makes it so that while using a wave generator, the corresponding analog wave seen on the I2C output will be the same wave. Again it's worth noting that while this program is running, the four Boot Monitor commands cannot be accessed through the Host Downloader. To download a new program, the user must restart the Richarduino device and use the Host Downloader to download the new program.

VIII. Results/ Discussion

The resulting board was designed and built using the methods and structure described in the previous sections. Images of the board can be found in figures 15 and 16. The board worked as expected. Audio could be recorded through the microphone, converted to digital signals, and then converted back to an analog wave and output through the I2C. This same process could also be repeated with the RF jack. Using the input RF jack, an audio mimicking wave could be input to the Richarduino, and the same wave would be output to the I2C. This can easily be seen while using the shield demo program, which continuously outputs the converted digital signal from the RF jack, and writes the digital value to the I2C. The result of the demo program can be seen in figure 17. In addition to the board working as expected, the host downloader also worked as expected. All of the four functions version, peek, poke, and program were able to function as anticipated. The host downloader is able to debug the board by using peek and poke to read and write to various places in memory. In addition, the host downloader was able to correctly download any binary program under 4kb to the Richarduino and run the program accordingly. One problem that occurred during the design of the pcb layout is that the CMOD was mirrored, causing the pins to misalign. To fix this problem, the CMOD was placed on the opposite end side of the board.In addition to the results of completing this Richarduino version, one addition that could be done to improve the Richarduino design in the next version is to operate the XADC in sequencing mode. With this mode selected, both channel 4 and 12 can be selected when accessing the converted values. This means that audio samples can be collected

from both the microphone and the RF jack without needing to reset and change the Richarduino design.

IX. Conclusions

The Richarduino is a device which can digitize audio and convert an audio sample back into an analog output. The Richarduino was mostly researched as an educational project, but it can be used in many scenarios as a microphone or an audio synthesizer. Since the design was made as an education project, simple methods were used to design the Richarduino such as using a simple counter for the UART and I2C as well as only using single channel mode for the XADC. Future versions of the Richarduino can build on these implementations as well as use this version as a model to further explore audio through hardware design.

X. References

[1] W. D. Richard, “Richarduino V4.0 Block Diagram.” 18-Jan-2022.

[2] W. D. Richard, “Richarduino V4 Memory Map.” 18-Jan-2022.

[3] W. D. Richard, “The Really Simple RISC Computer.” 17-Nov-2020.

[4] W. D. Richard, “RSRC_4KB_Sync_SRAM.” 26-Nov-2018.

[5] W. D. Richard, “Richarduino V4 Registers Word Based.” 26-Mar-2022.

[6] W. D. Richard, “UART Simulation.” .

[7] W. D. Richard, “Richarduino V3. UART Specification.” 08-Jan-2021.

[8] “AD5301/AD5311/AD5321 Data Sheet (Rev. C).” Analog Devices, INC, Jun-2016.

[9] “THE I2C-BUS SPECIFICATION VERSION 2.1.” Philips Semiconductors, Jan-2000.

[10] “7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter (v1.10.1).” Xilinx, 23-Jul-2018.

[11] “Cmod A7 Reference Manual .” Diligent, Inc, 04-Oct-2019.

[12] W. D. Richard, “Richarduino V4.0 Pins.” .

[13] W. D. Richard, “EVERYTHING YOU ALWAYS WANTED TO KNOW ABOUT SYNTHESIZABLE VHDL BUT WERE AFRAID TO ASK.” 06-Apr-2021.

[14] W. D. Richard, “CSE 362M LAB 0 TUTORIAL.” 03-Sep-2021.

[15] “XADC Wizard v3.3 LogiCORE IP Product Guide.” Xilinx, Inc, 05-Oct-2016.

[16] W. D. Richard, “eprom.exe.” .

[17] W. D. Richard, “cmod_a7_sch.” .

[18] “Welcome to pySerial's documentation,” *Welcome to pySerial's documentation - pySerial 3.4 documentation*. [Online]. Available: https://pyserial.readthedocs.io/en/latest/. [Accessed: 03-May-2022].

[19] “Jinja,” *Jinja*, 09-Nov-2021. [Online]. Available: https://jinja.palletsprojects.com/en/3.1.x/. [Accessed: 03-May-2022].

[20] W. D. Richard, “Demo Shield Block Diagram.” .

[21] “MIC5255.” Micrel, Inc, Nov-2006.

[22] “Low-Cost, Micropower, SC70/SOT23-8, Microphone Preamplifiers with Complete Shutdown.” Maxim Integrated Products, Inc, Jun-2012.

[23] “AD5301/AD5311/AD5321 Data Sheet (Rev. C).” Analog Devices, Jun-2016.

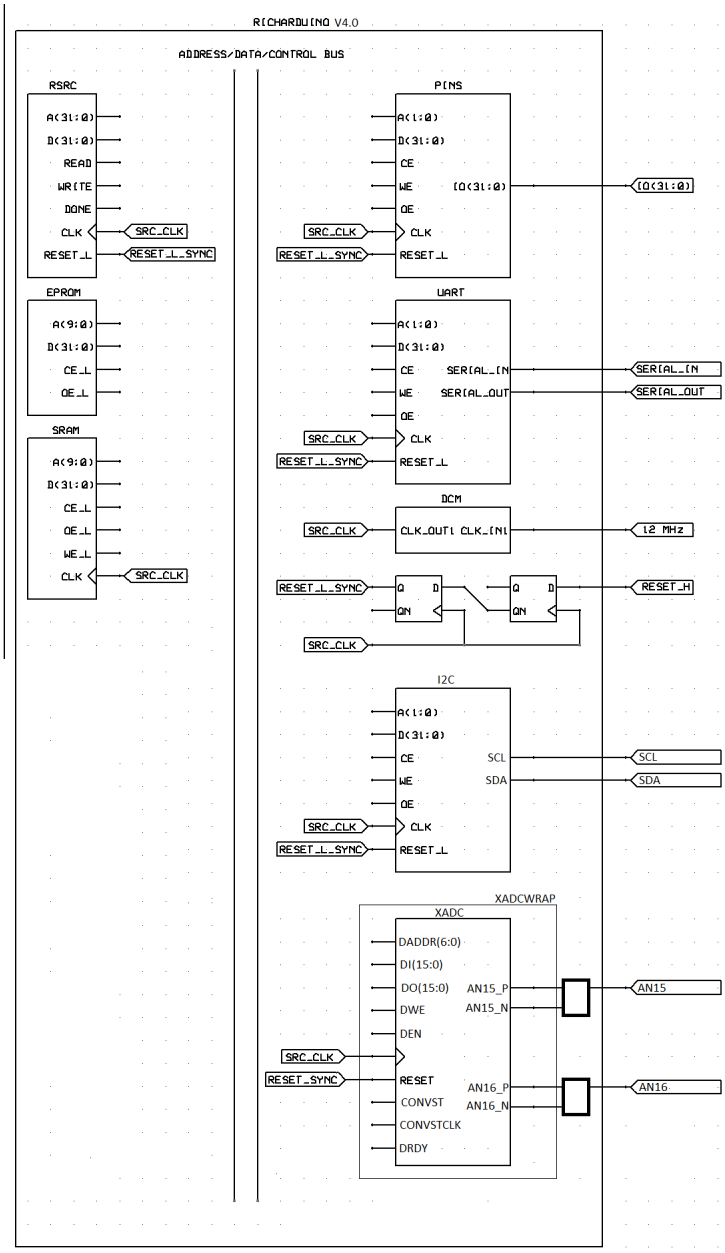

Fig. 1. Block Diagram of Testbench. The diagram includes all of the components included in the Richarduino, as well as an address and data bus.

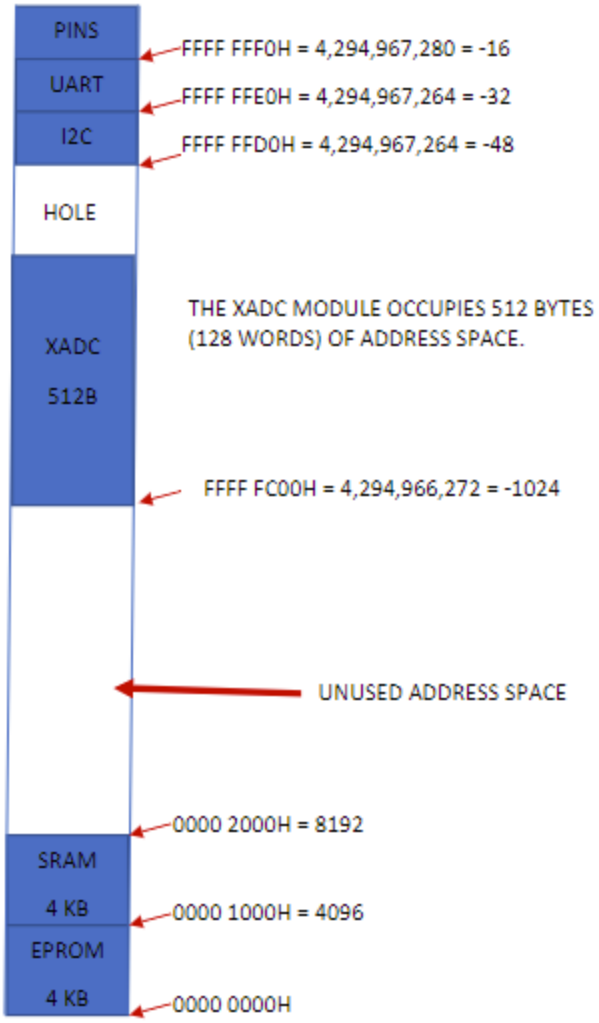

Fig. 2. Memory map of the Richarduino. All of the components from the Richarduino are mapped to a certain address space.

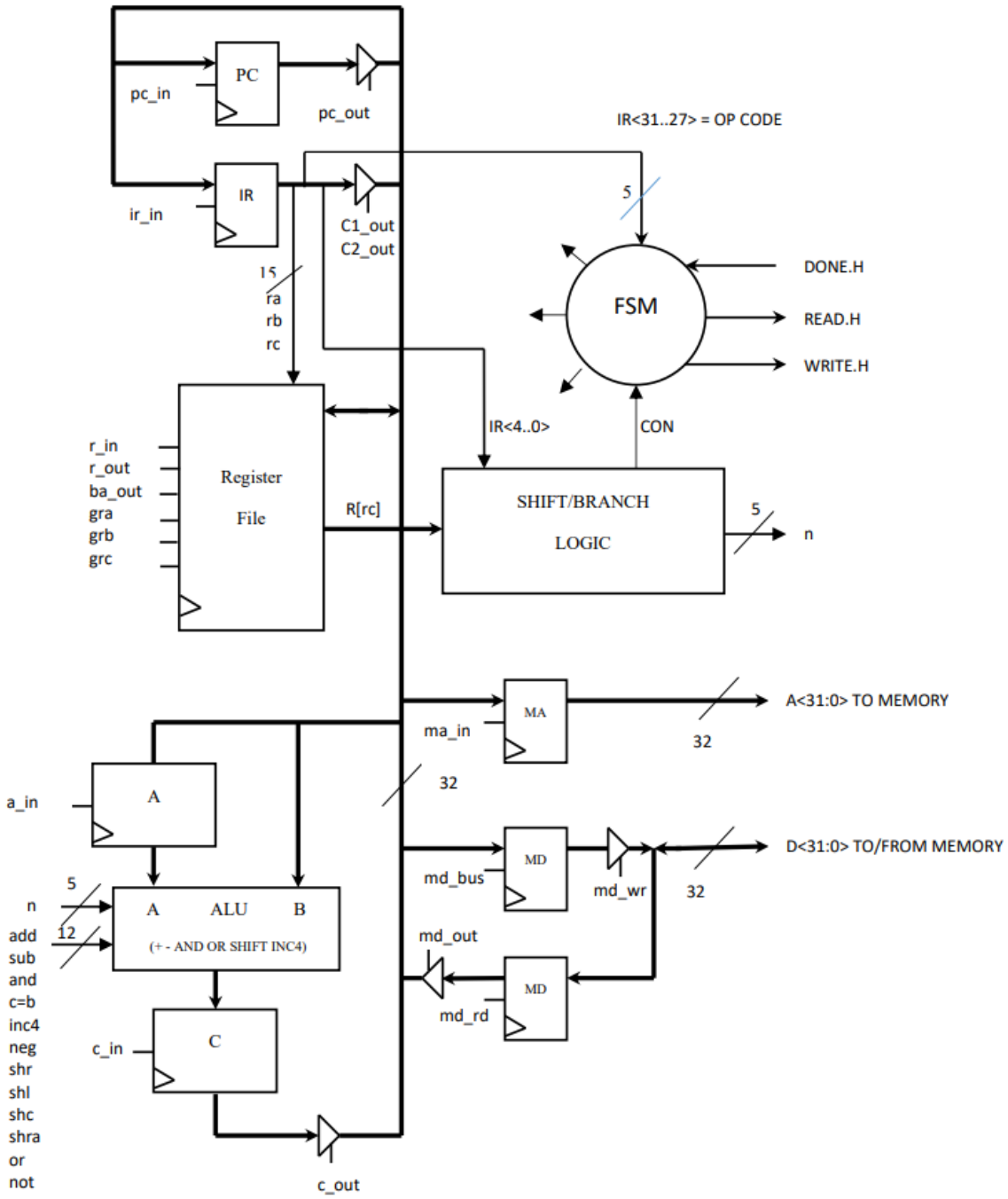

Fig. 3. Shows the structure and components of the RSRC processor. All of the control signals labeled are determined by the FSM. The RSRC asserts both READ and WRITE signals, as well as can read or write data to the address or data bus.

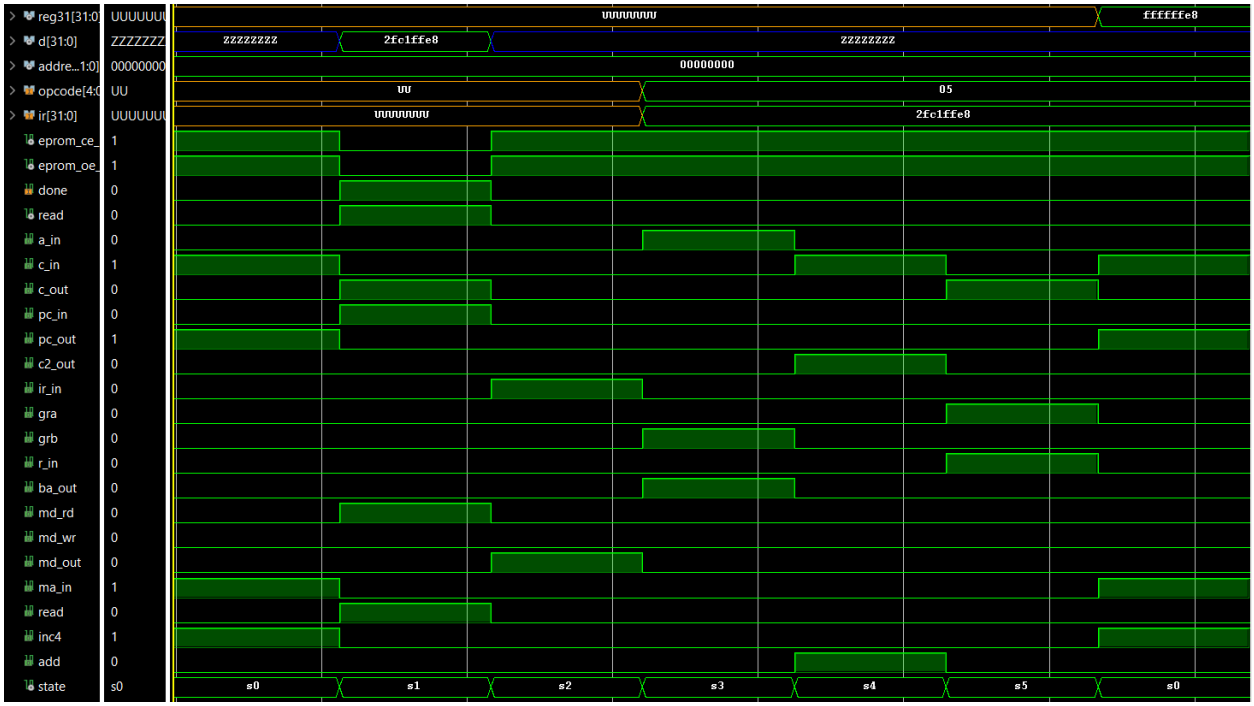

Fig. 4. A simulation of a load instruction (la r31,-24). In states 0 - 2, the instruction is loaded into the Instruction register. Once loaded, the control unit asserts each control signal based on the opcode of the instruction and the current state. After state 5 it can be seen that register 31 is updated with the correct value.

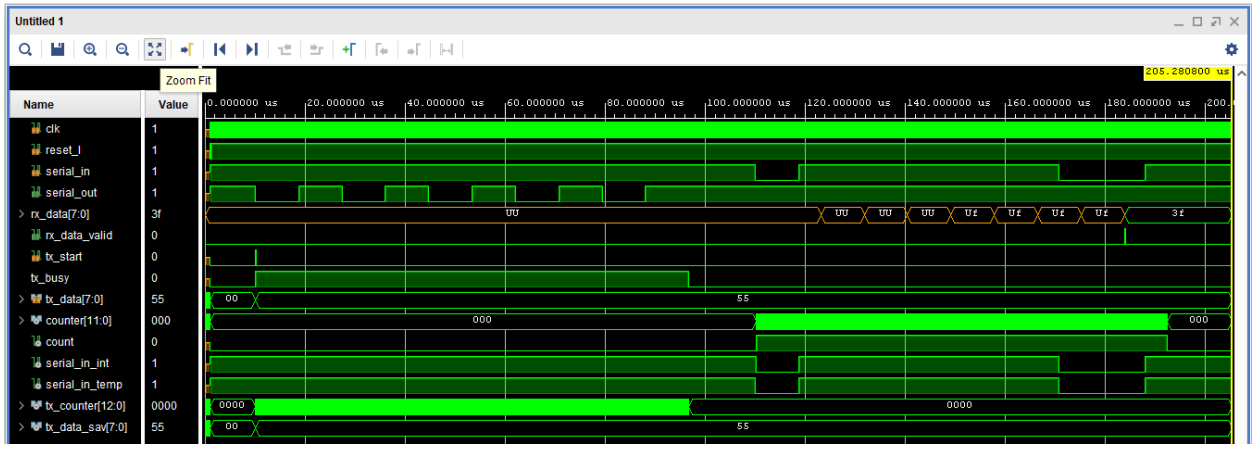

Fig. 5. A simulation of the UART receiver receiving a serial input. The input register is filled one bit at a time starting from the least significant bit. The data in the input register of the UART is not valid until the last bit is read. The counter stops when the transfer is complete.

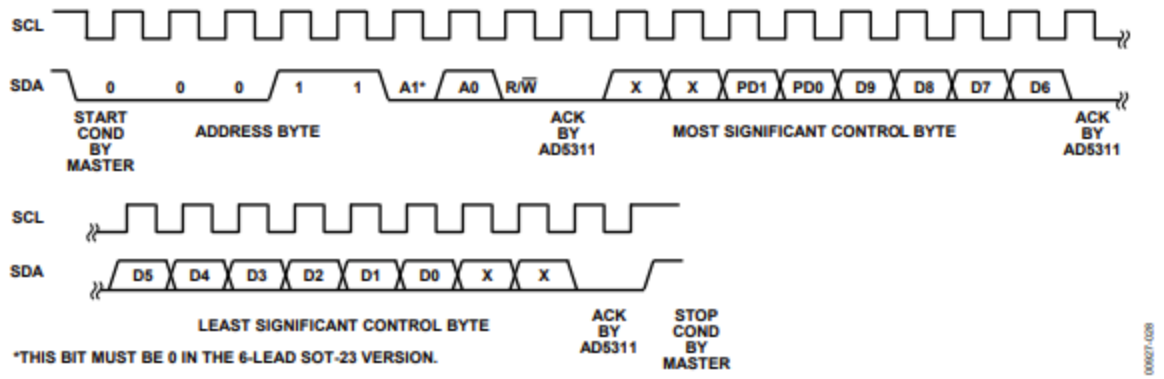

Fig. 6. A reference for a write to the I2C interface. The transfer is initiated by dropping SDA LOW while SCL is HIGH. A seven bit address and a READ/WRITE bit is asserted onto SDA. After each byte of data transferred, SDA is not asserted by the master for one SCL clock tick, allowing for the AD5331 DAC to send an ACK. The rest of the data is sent in a similar fashion. The transfer ends with the raising of SDA while SCL is HIGH.

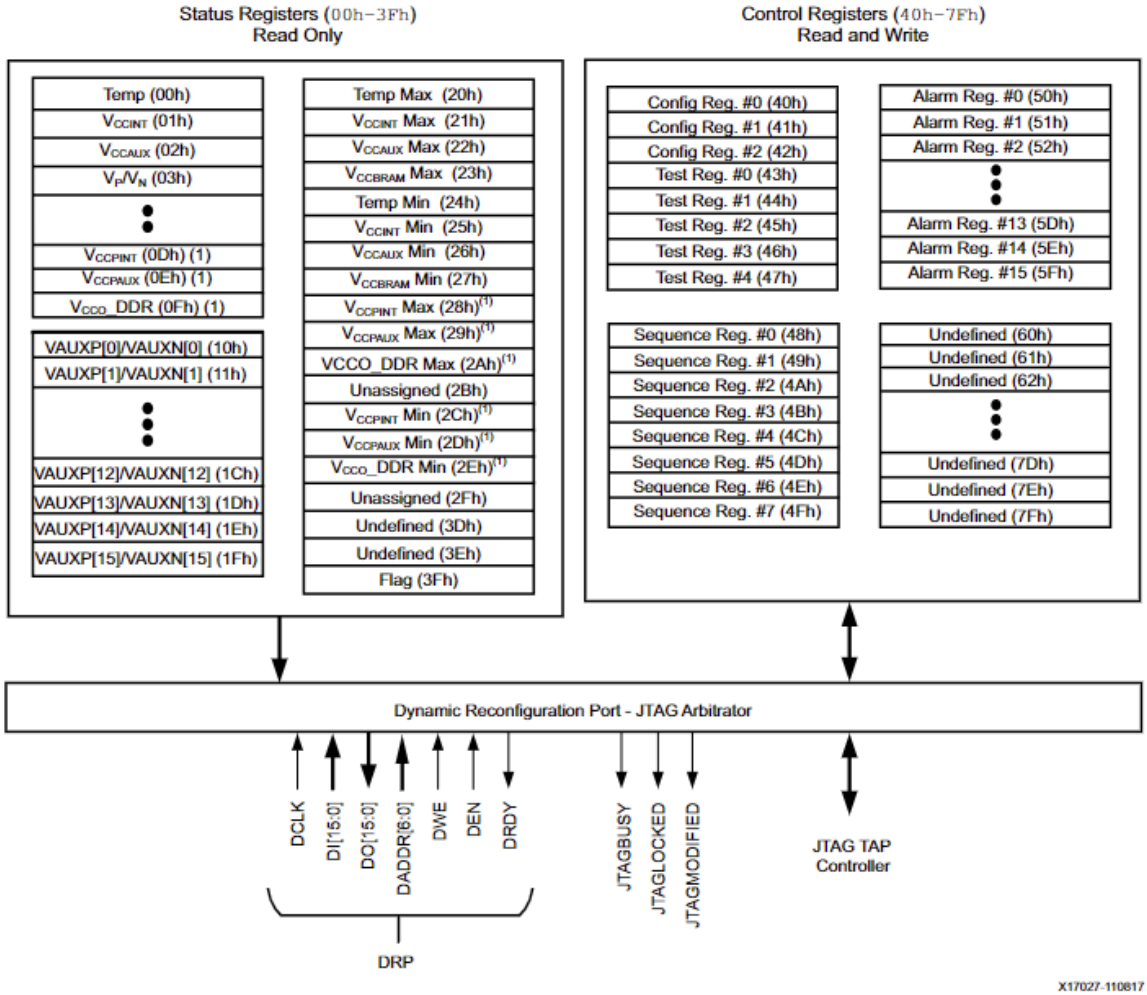

Fig. 7. All registers accessible through the DRP. Converted digital values are held in the VAUXP/N[##] registers. The richarduino makes use of VAUXP/N4 and VAUXP/N[12] registers. In addition, config registers are used to configure the behavior of the XADC.

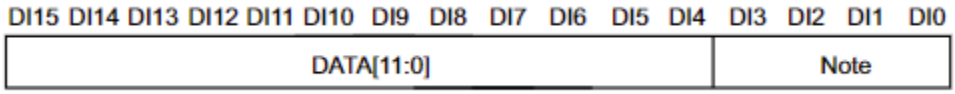

Fig. 8. Structure of the output from an XADC status register. For audio conversions, the digital value will be stored in the 12 most significant bits. The lower 4 bits are for calibration purposes.

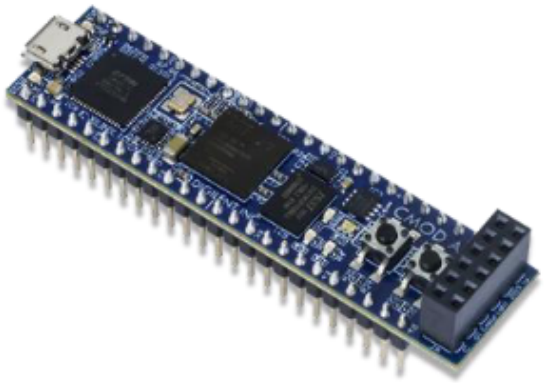

Fig. 9. The CMOD A7 board.

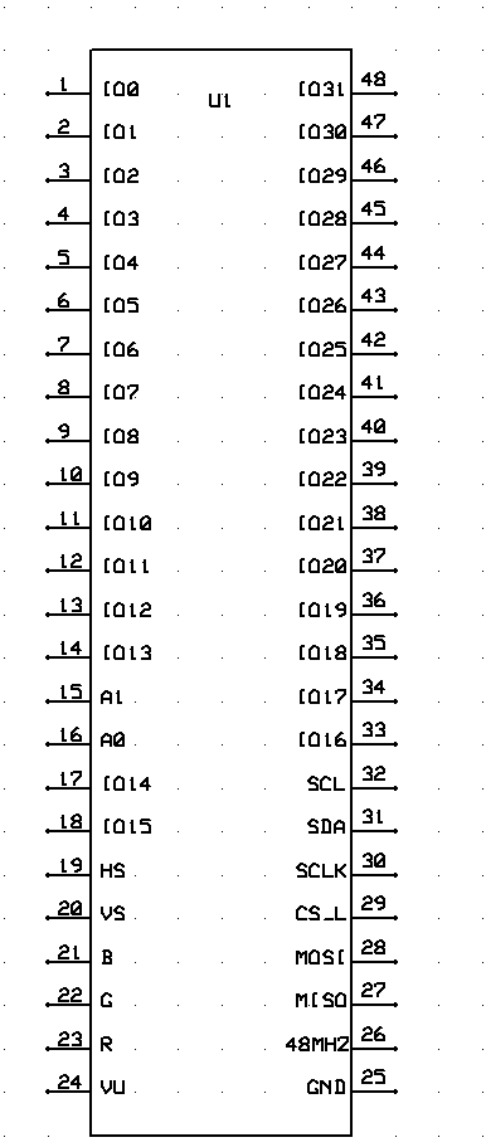

Fig. 10. 48 Pin layout for the Richarduino. IO0 is connected to the external LED. A1 and A0 are connected to analog inputs from the microphone and RF jack. SDA and SCL are connected to the I2C DAC.VU is connected to the voltage regulator and GND is connected to ground.

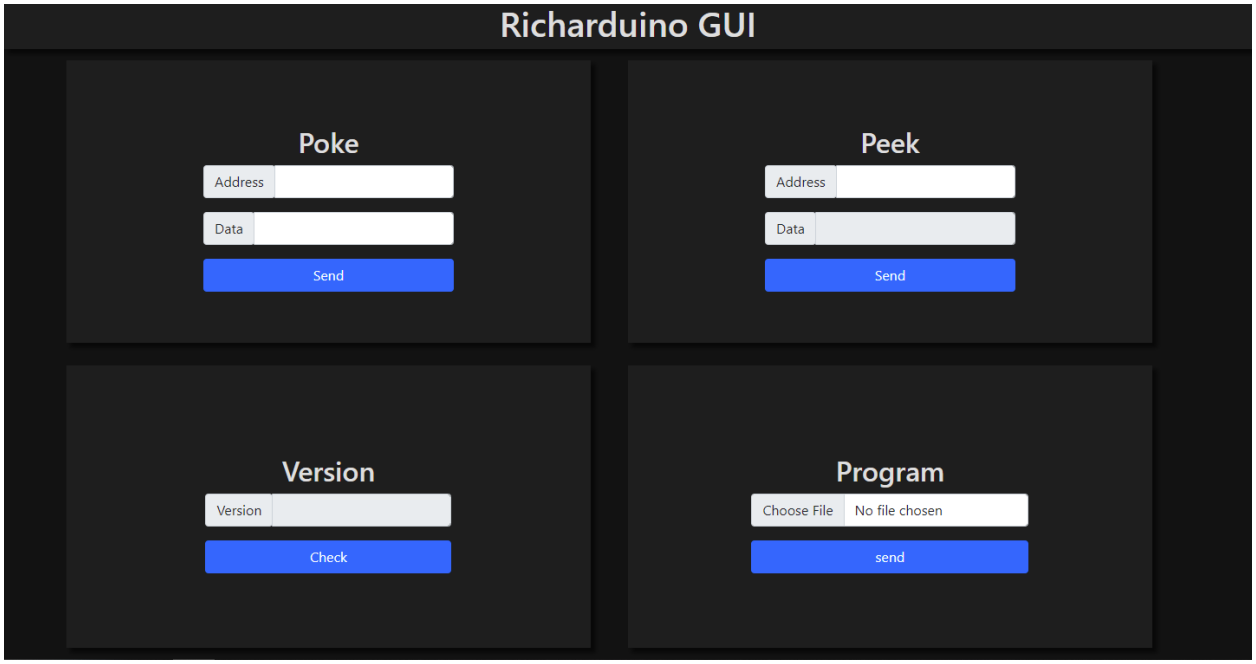

Fig. 11. UI layout for communicating with the Richarduino. The Host Downloader allows for all of the four communication functions PEEK, POKE, VERSION, and PROGRAM.

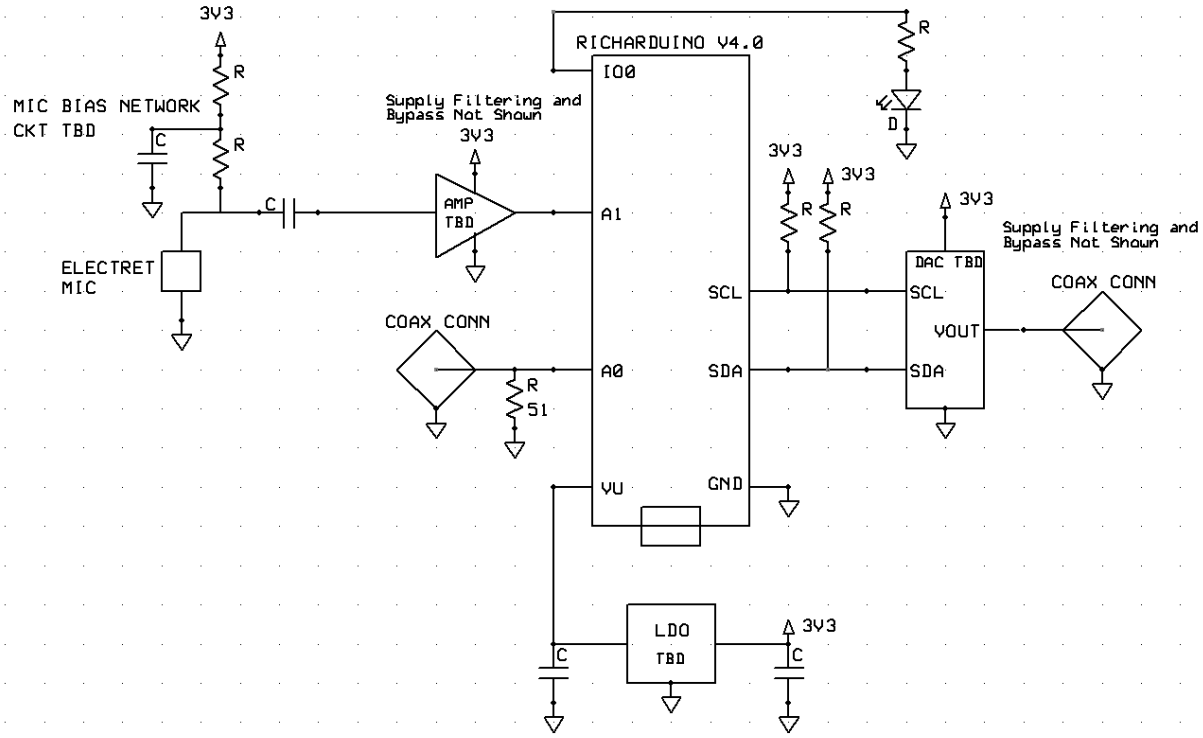

Fig. 12. Block diagram of the Demo Shield. Contains a microphone input circuit and RF jack input circuit for analog inputs. The AD5311 DAC is connected to the SCL and SDA output pins on the Richarduino as well as an RF jack for outputting an analog signal. A voltage regulator is connected to a voltage output from the Richarduino

to supply the rest of the Demo Shield with power. A single LED is also connected to pin IO1.

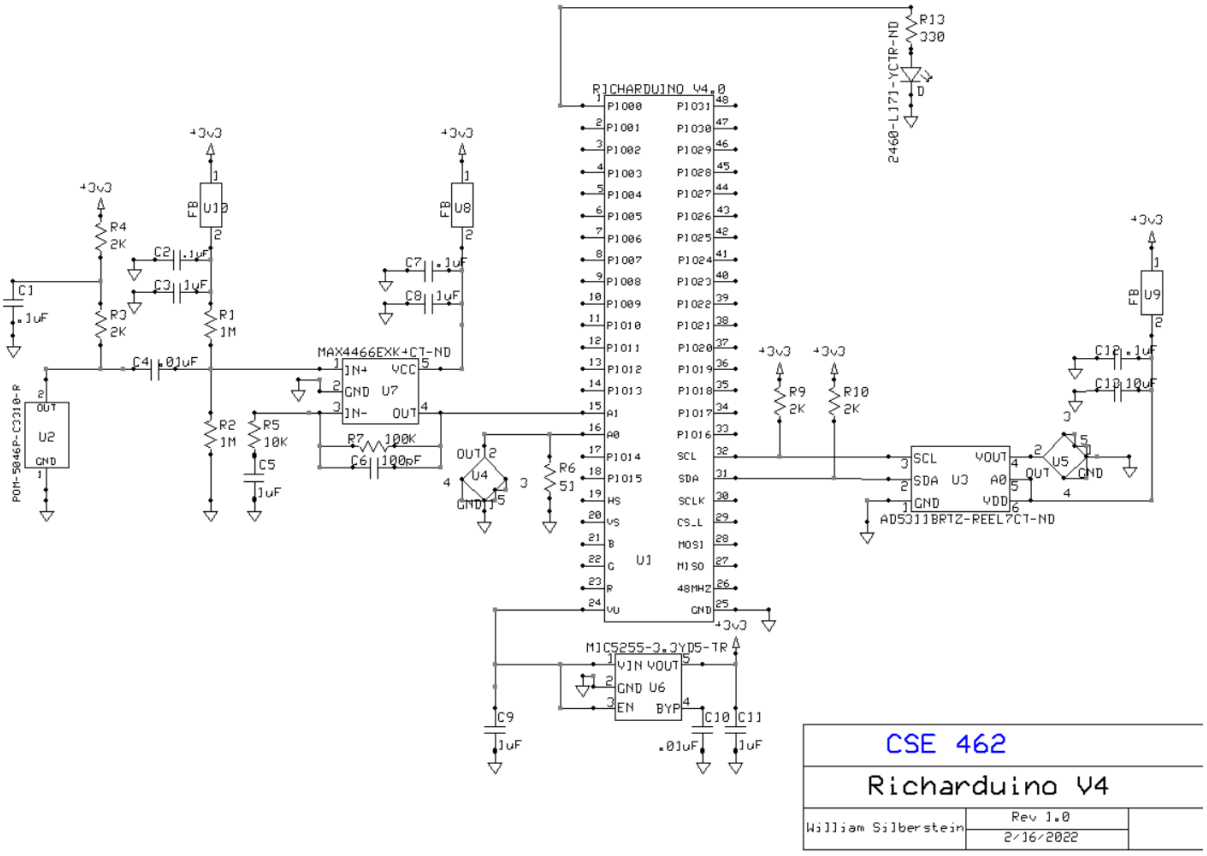

Fig. 13. Full schematic of the Demo Shield circuit.

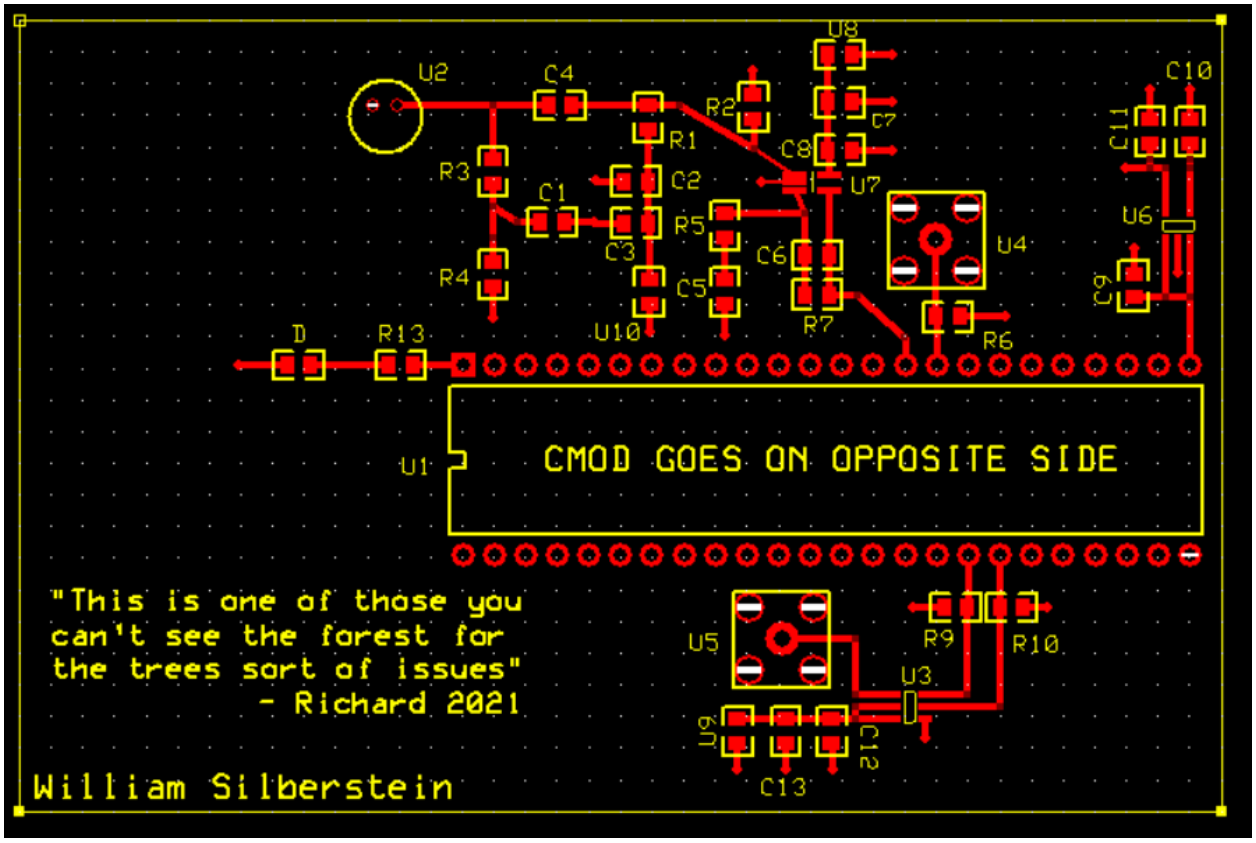

Fig. 14. PCB layout of the Richarduino. Shows all of the locations of the components and connections.

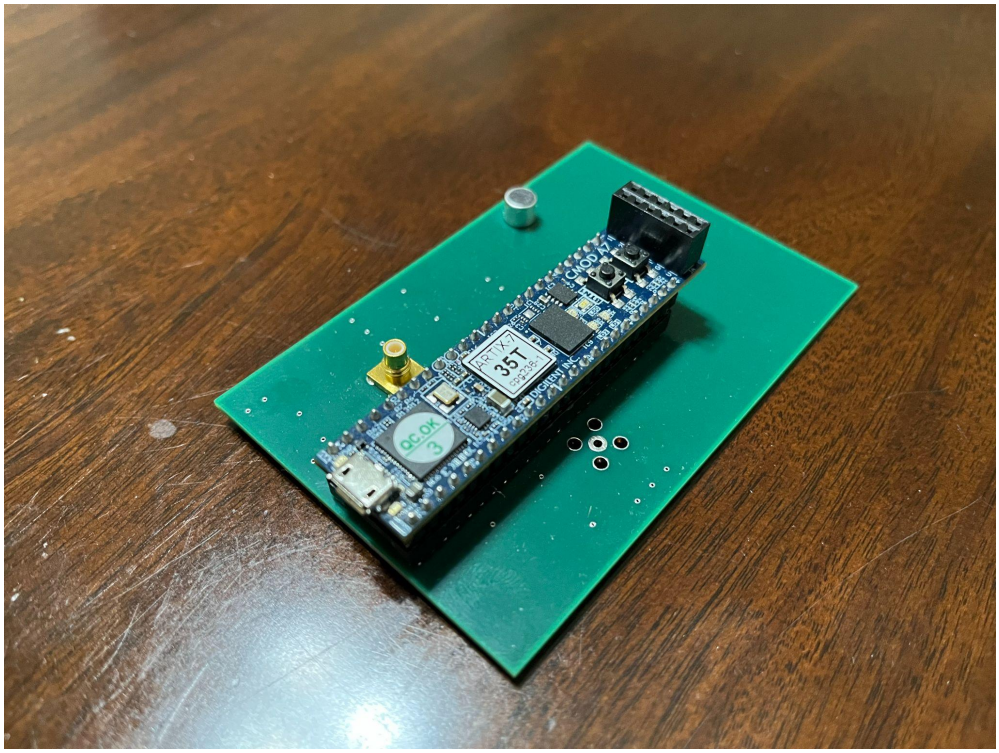

Fig. 15. Front side of the Richarduino. Displays the CMOD, microphone, analog input RF jack, and I2C output.

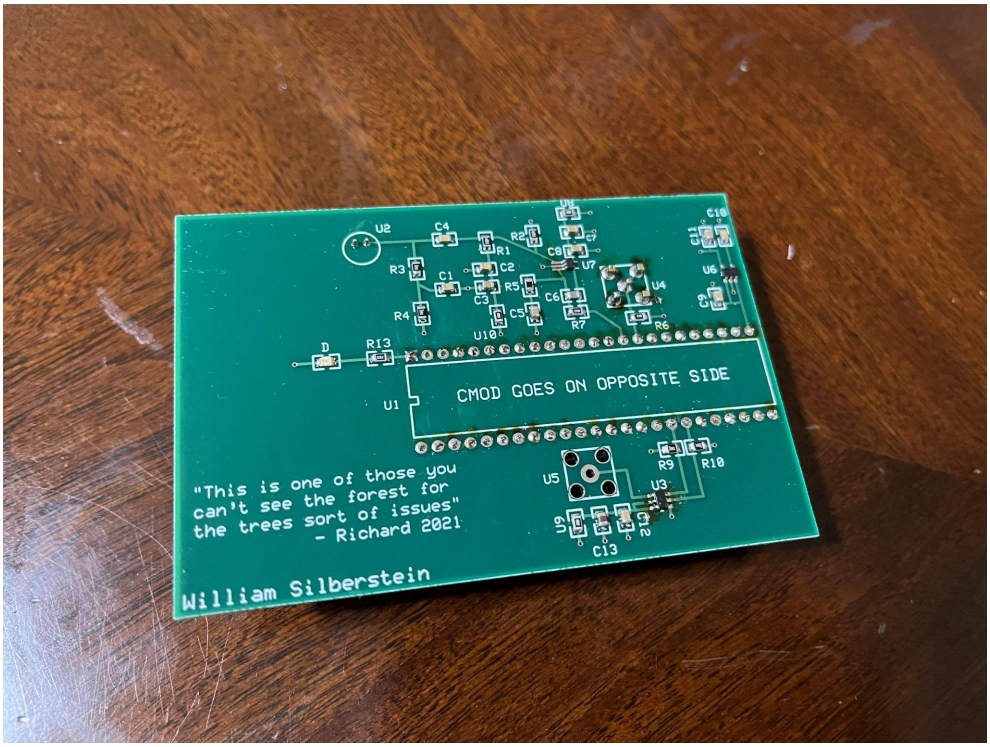

Fig. 16. Back side of the Richarduino. Shows the circuit components.

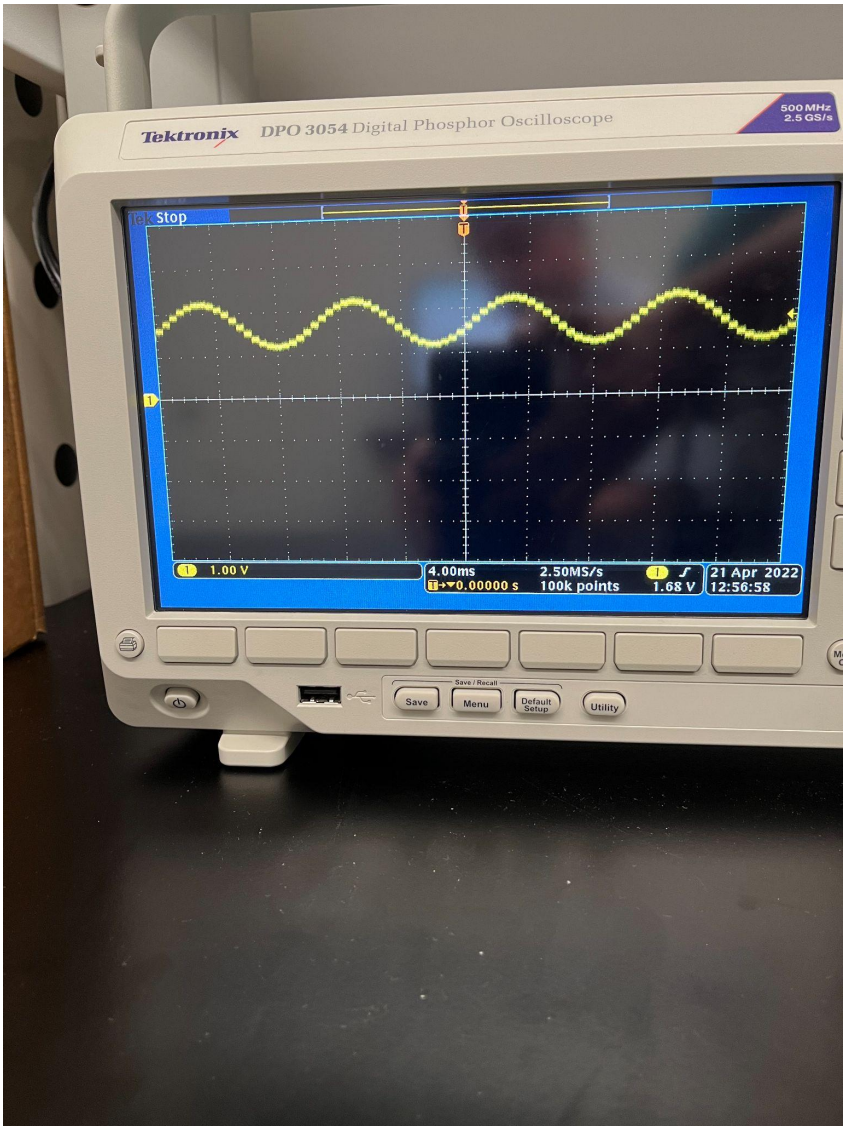

Fig. 17. Demo program test. A wave generator was connected to the RF input jack and outputted a sine wave. After digitizing the analog wave, the digital value was written to the I2C output where the digital signal was converted back to an analog signal. The resulting output is shown as this sine wave.