Final Report: The Fully-Synchronized Synthesizer (FSS) Interface Prototype

Computer Design Laboratory ECE 3710
Fall 2021
The University of Utah

Jacob Peterson
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Brady Hartog
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Isabella Gilman
Computer Engineering 2023
University of Utah
Salt Lake City, UT

Nate Hansen
Computer Engineering 2023
University of Utah
Salt Lake City, UT

***Abstract*—We present the design and implementation of a small-scale prototype for a fully-synchronized synthesizer ("FSS") interface. The prototype is designed to address a major drawback of contemporary musical synthesizers. As a proof of concept for an innovative user interface, the prototype casts a vision for a more versatile and powerful synthesizer for the musical artists of tomorrow. Pursuant to the course objectives of ECE 3710, the principal component of the prototype is an Intel Cyclone V FPGA. This report chronicles our process of programming and interfacing with the FPGA as well as crafting a hardware system to deliver a prototype with a highly intentional and interactive user experience.**

I. Introduction

In 1978 the music technology firm Sequential Circuits introduced the Prophet-5 musical synthesizer. Prophet-5 was among the world's first *fully-programmable* synthesizers, meaning that every controllable parameter could be stored in user-defined programs. Full programmability was a major milestone in the history of the synthesizer, and Prophet-5 would set the precedent of synthesizer design for decades to come.

In fact, over 40 years later, Prophet-5 remains the archetype of modern hardware synthesizers. Whereas the fully-programmable paradigm of Prophet-5 is highly useful, it suffers a major drawback. Fig. 1 illustrates the drawback. In the fully-programmable paradigm, each parameter has both a *physical value* and an *active value*. The physical value is the value indicated by the position of the dial. The active value is set either by: 1) changing the position of the dial (in which case it is the same as the physical value); or by 2) recalling a user-defined program. Naturally, recalled values do not necessarily match—and in practice seldom do match—the physical values of the dials. We say that parameters controlled in this way are *unsynchronized*.

Fig. 1. The drawback with traditional, analog level meters. The physical and active values are out of sync!

The drawback described above assumes the use of traditional, analog level meters as shown in Fig. 1. Fig. 2 depicts a proposed alternative to the traditional level meter. This level meter consists of an LED ring display with digital control. By leveraging digital control, the level meter can display the active value of the parameter at all times, whether it is set by changing the dial or by recalling a program. We say that a parameter controlled in this way is *synchronized*.

We envision a musical synthesizer for which every parameter on the control panel is synchronized; such would be a *fully-synchronized* synthesizer ("FSS").

In this report we present the design and implementation of a small-scale prototype for an FSS interface (Fig. 3) with respect to the learning objectives of ECE 3710. We believe our prototype is a successful proof of concept for the FSS interface as well as a clear, articulate, and creative application of our computer engineering skills to date. We hope you enjoy our presentation of the FSS interface prototype.

Fig. 2. The proposed digital level meter to implement synchronized parameters.

Fig. 3. The FSS prototype.

II. Overview

The following is an overview of the user-facing functionality of the FSS prototype.

The FSS prototype has three synchronized parameters and three user-defined programs. The objective of the device is to demonstrate that all three parameters can be stored in each of the programs and recalled on demand.

Turning any of the parameter dials will change the value in its ring display by a corresponding amount. As soon as any parameter is changed beyond the programmed state, the save button LED turns on. The user may save their changes by pressing the save button, whereupon the save button LED turns off. The user may recall programs by pressing the corresponding program buttons.

The play/pause button is intended for auxiliary functionality, such as playing a sound if the FSS prototype is connected to an audio engine. In our implementation, the play/pause button is used to trigger animations of the display elements for aesthetic purposes.

III. Hardware

The hardware assembly of our device consists of 3 main components: A DE1-SoC Cyclone V FPGA board, a small breakout box for power supply and I^{2}C bus interfacing, and our FSS prototype device. These components and their connections can be seen at a high level in Fig. 4.

A. FPGA Board

The FPGA device drives all of the logical control for the FSS device. This is accomplished by instantiating a single module of our CR16 processor, and adding to that a set of peripheral modules. The modules that are peripheral to CR16 are responsible for allowing the assembly code access to important device data such as the state of the buttons and rotary encoders, the state of the LED indicators, and a 48-bit microsecond counter for I^{2}C timing. All of these modules were coded in SystemVerilog HDL and are available within the GitHub repository "FSSPrototype." The FPGA acts as an I^{2}C master, where GPIO pins 1 and 2 are the SCL and SDA lines respectively, as shown in Fig. 5 The FPGA is also attached to a common ground shared by the FSS prototype device and the breakout box. In order for the device to function, the FPGA is programmed with the FSS prototype Verilog modules (`fss_top.sv` being the top-level module), the CompactRISC16 (CR16) Verilog modules, and a block RAM (BRAM) initialization file containing the machine code of our firmware.

The FPGA device allowed our team to employ a variety of useful testing methods, such as displaying data from BRAM on the bank of seven-segment displays with the slide switches acting as the BRAM address, assigning an FPGA button to a global reset, and testing I^{2}C communication. Our device relies on the internal 50MHz clock on the FPGA board, but we could also slow down runtime execution during debugging by tying the clock signal to a push button. Successful design and testing of this project relied on substantial interfacing with the FPGA.

B. Breakout Box

The breakout box is responsible for powering the FSS prototype interface as well as forwarding the I^{2}C signal coming from the FPGA to the port expander chips on the main PCB. The housing of the breakout box contains a USB type-A receptacle, a USB type-B receptacle, 2 male pin headers, a buck converter, and a rocker switch. The USB type-B receptacle is responsible for receiving power from the USB cable voltage bus and delivering it to the breakout PCB and then the main PCB. This power cable supplies 5V to the board from a DC power source, which is generally a wall adapter plugged into a standard outlet. From here the voltage is stepped down to 3.3V by a DC-DC buck converter contained within the breakout box to ensure the main PCB is receiving an operating voltage of 3.3V. Then, the USB type-A cable supplies this power to the main PCB board. To make control of power simple, the breakout box also includes a rocker switch that toggles the power on and off.

The control logic sent by the FPGA is also routed through the breakout box. The FPGA drives the I^{2}C bus lines, both SDA (serial data) and SCL (serial clock), using the GPIO PMOD pin headers on the FPGA board with an open-drain configuration. Breadboard wires then connect the two bus lines from the FPGA board to the breakout box, and the breakout PCB routes the two bus lines to the USB type-A cable which is then connected to the FSS prototype. The USB type-A cable's D+ and D- differential data lines are adapted to be the SCL and SDA signals of the I^{2}C bus respectively.

The breakout board schematic and PCB layout are shown in Figures 6 and 7 respectively.

C. FSS PCB

The FSS prototype's PCB is responsible for all the electronic circuitry on the device. In this document, we refer to this PCB as the "main" PCB and it resides within the prototype's housing. We wanted to preserve the beauty factor of our FSS prototype, so instead of using many breadboard wires to interface the main PCB to the FPGA board, we opted to use a single USB cable that is inserted into the back of the prototype's housing. A USB Mini type-B receptacle connects this USB cable from the breakout box to the main board, and is used to provide power throughout the board and, as mentioned previously, adapts the D+ and D- differential data lines as the SCL and SDA signals of the I^{2}C bus. All communication from the FPGA to the main board must take place serially since it's impractical to drive the board's components individually using the PMOD pin headers on the FPGA board. So, to drive the board's components serially, the `GMSB0522102Y7EU` 8-bit I/O port expander IC is used. Two of these port expander chips in conjunction provide complete I/O to the entirety of the main PCB and the serial communication protocol to interface with them is I^{2}C in standard mode (100 kHz clock speed). This port expander chip is quasi-bidirectional, meaning that there is no configuration register to specify whether a port is an input or an output. Instead, to use a port as an input, the port is driven high and can be pulled low by an external component. 5 tactile push buttons with a pull-up configuration are connected to one of the I/O port expander chips. Additionally, there are 3 quadrature-encoded, mechanical rotary encoders each with two channels — A and B. The individual channels use a pull-up configuration and are connected to one of the I/O port expander chips. These two channels in conjunction make up the quadrature-encoded output of these rotary encoders.

The FSS prototype contains 62 LEDs with an amber hue. There are 19 LEDs in each of the 3 ring displays and 1 LED for each of the 5 button indicator lights. To drive all of these LEDs using minimal I/O, 4 of the `STP16CPC05MTR` 16-bit, latching, shift-register LED driver ICs are used. Each driver contains the following pins: `SDI` (serial data in) which the input to the shift register, `SDO` (serial data out) which contains the bit shifted out of the shift register, `CLK` (clock) which synchronizes `SDI` and `SDO`, `LE` (latch enable) which latches the driver output when asserted, `OE` (output enable) which is the active-low enable pin, and `R-EXT` which controls the chip's internal current limiter to provide a constant-current source to the LEDs. These 4 LED driver ICs on the main board are daisy-chained together and the first driver in the chain has its `SDI` pin connected to one of the ports on the I/O port expander chip. Additionally, one of the I/O port expander chips drives all 4 `CLK` and `LE` signals of the LED drivers so that they are all synchronized.

The SDA and SCL lines of the I^{2}C bus are short-circuit protected using 300Ω series resistors. This resistance is low enough that it does not affect the I^{2}C bus capacitance, which is limited to 400 picofarads in standard mode. Additionally, the 6 ft. braided USB cable used to interface with the main board is shielded, has a low resistance, and a low parasitic capacitance, so the I^{2}C bus timing requirements are met without an issue. As part of the I^{2}C specification, the SDA and SCL lines are directly connected to 3.3V using two 2kΩ pull-up resistors.

As another precaution, series resistors are used to protect against accidental short-circuits on the pull-up/pull-down inputs of the I/O port expander chip. In the event that a quasi-

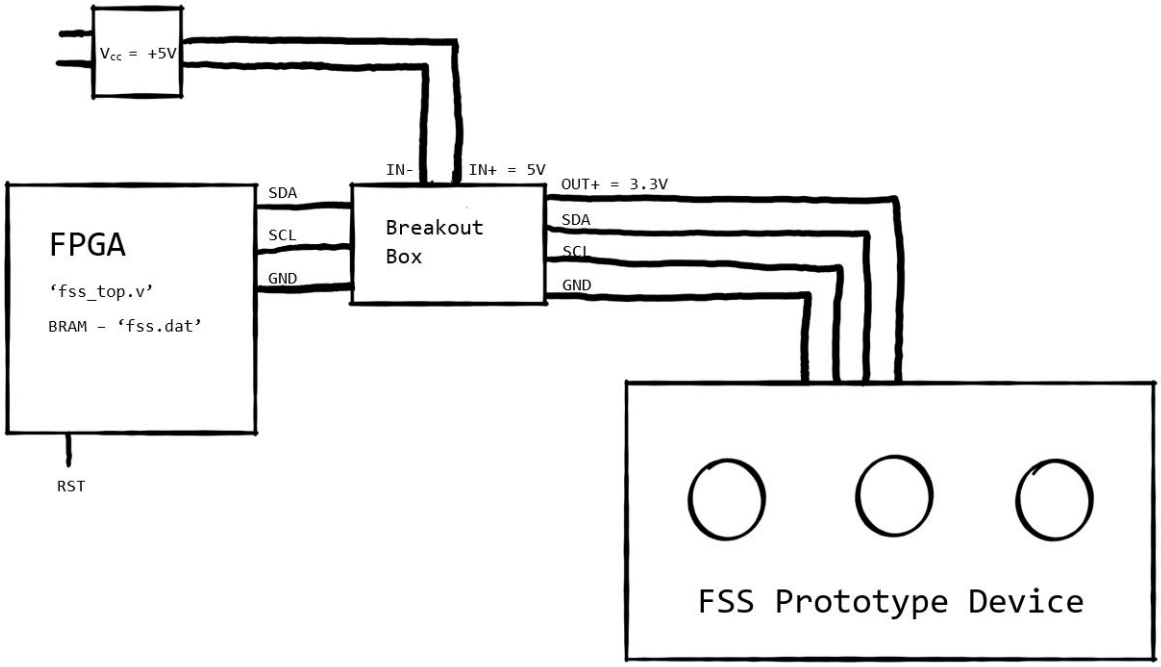

Fig. 4. High level block diagram of the full FSS hardware assembly.

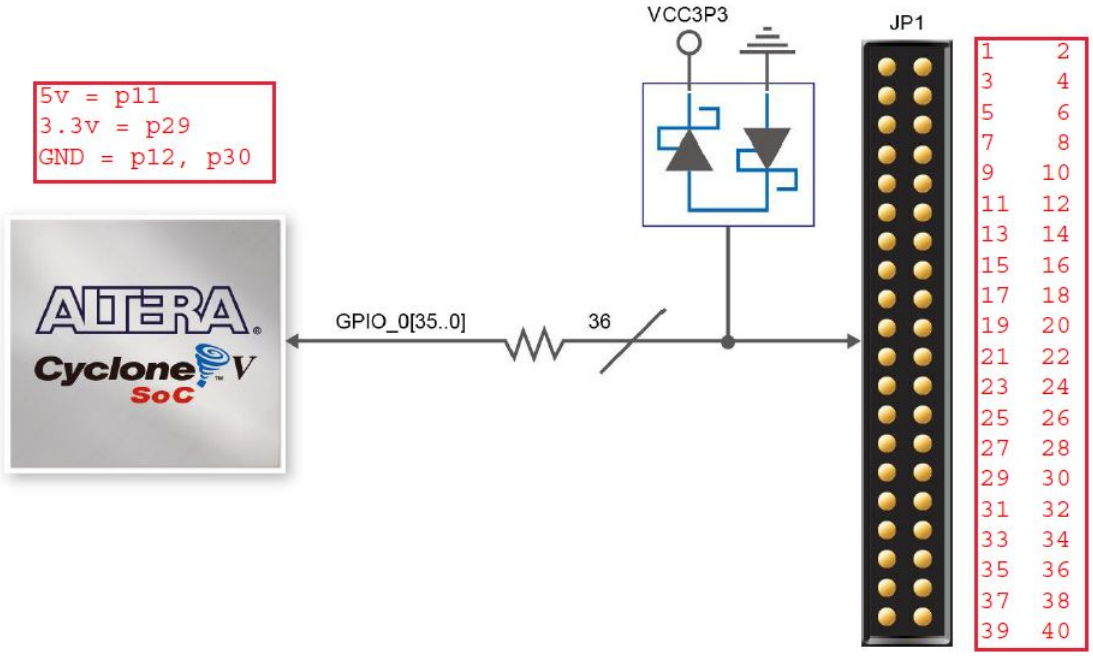

Fig. 5. GPIO pin diagram as obtained from the Cyclone V user manual with mappings added [4].

bidirectional port that is treated as an input is accidentally driven low via I^2C, a short-circuit can occur if a pull-down input configuration is pulled high — directly connecting a logical high to a logical low with no series resistance.

The ECAD tool used to create the schematics and PCB layouts is called easyEDA, an online cloud-based PCB design tool. The Gerber files were exported from easyEDA and uploaded to JLCPCB, our PCB fabricator of choice. This process was done for both the main board and the breakout board schematic and PCB. We also ordered an SMT stencil for our main PCB which allowed us to apply solder paste to the SMD pads efficiently. We then placed the various SMD components onto the solder paste areas accordingly and reflowed the components using a reflow oven.

The main board schematic and PCB layout are shown in Figures 8 and 9 respectively. Additionally, Figure 10 shows a picture of the final PCB assembly.

D. FSS Housing

For the external housing, our team decided to go with a minimal design. The final assembly was constructed from stacked layers of acrylic panels, providing a solid structure with pleasing weight. The top layer was a sheet of translucent black acrylic, which allowed for our LED to shine through in a low amber color. Veneer surrounds the perimeter of the housing, giving a more natural feeling to the box. Finally, the knobs and control buttons were constructed out of acrylic to give them that same weight and sleek feeling the rest of the

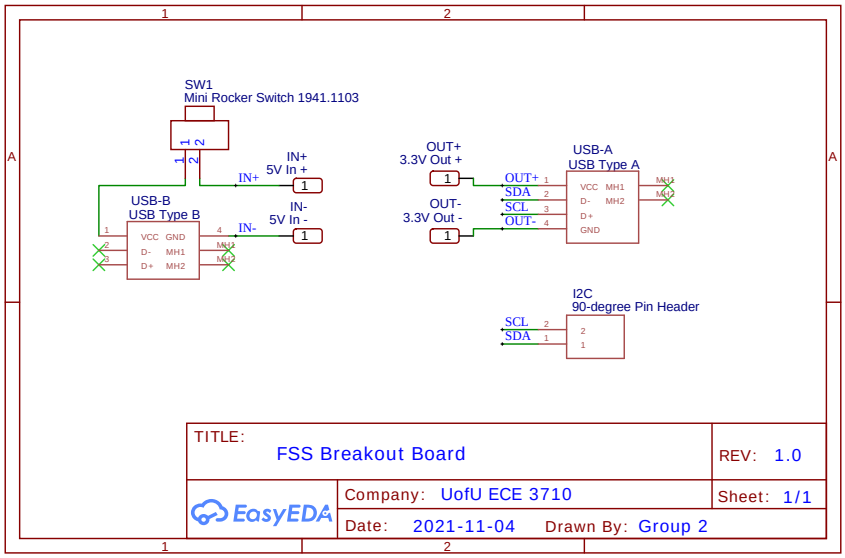

Fig. 6. Schematic of Breakout PCB.

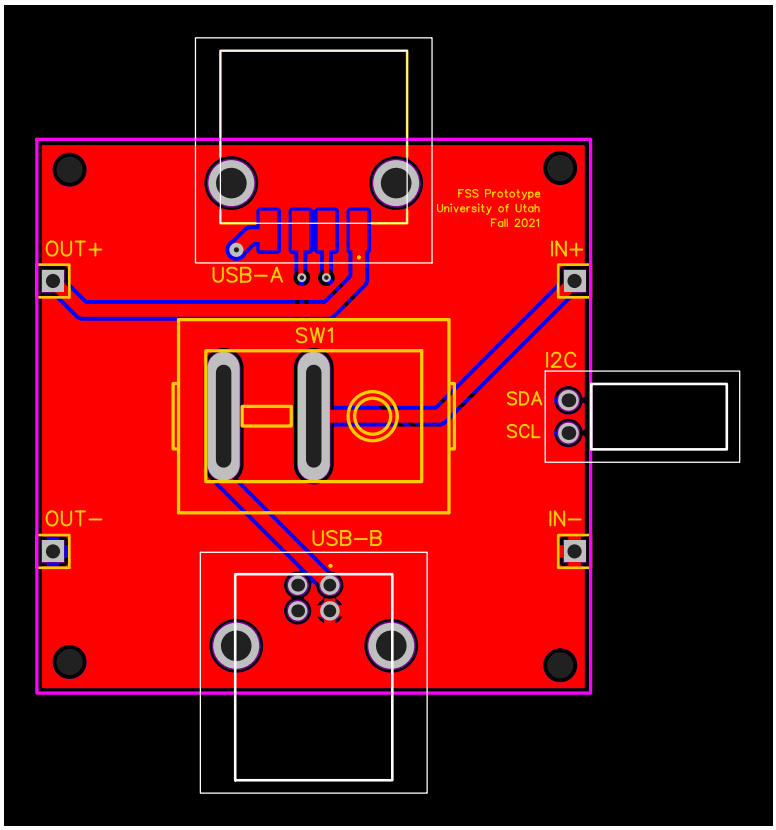

Fig. 7. Layout of Breakout PCB.

synthesizer gives. This design is minimal, but gives the user a sense of quality. By following a design process that takes the user's needs into account, we were able to implement a device that is equal in both form and function. The final result has the quality and professional feeling that is geared toward musicians and producers.

The internals of the FSS housing can be seen in Figure 8. This picture is of an older version where we had used plywood for the internal layers, before we replaced them with acrylic.

IV. CR16 Processor

The details of our CR16 processor are contained in the previous lab reports, but for the purposes of the FSS prototype's interface firmware, it would be enlightening to review some of the aspects of our CR16 ISA that were implemented specifically to abstract away some of the complexity in writing assembly code. Although the processor we designed is mostly equipped for RISC instructions, these added instructions may have more of a CISC "feel" to them. The FSM controller of the CR16 CPU manages the fetch-decode-execute procedure for these instructions in the same way as the other instructions. Most of these instructions could be replaced by a series of RISC instructions, but we saved clock cycles by implementing their behavior into the CPU FSM. Each of the following instructions is accompanied by an explanation for their implementation:

1) `LOADX/STOREX`: These instructions are external `LOAD` and `STORE` instructions which will be capable of loading and storing data directly into an external memory bank.

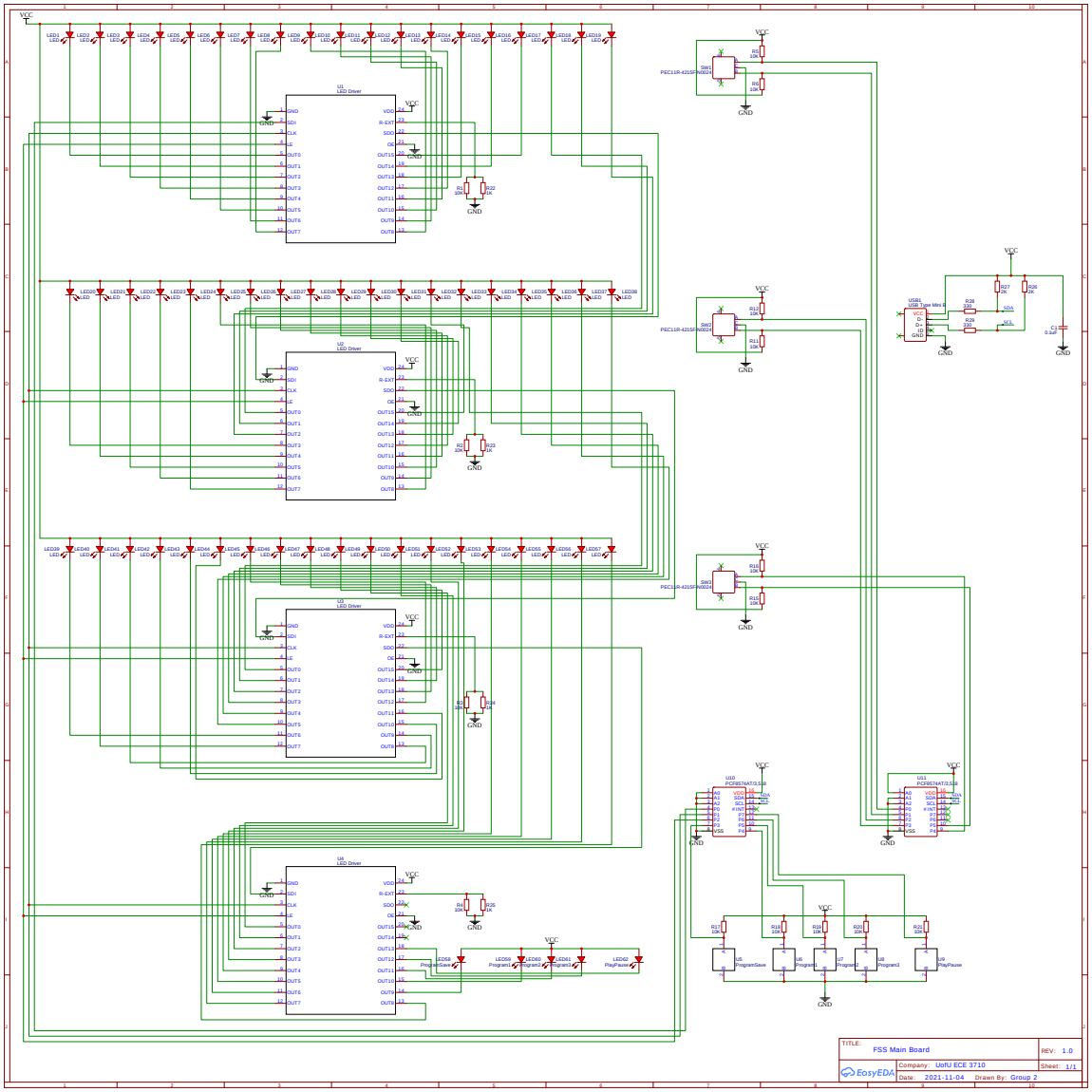

Fig. 8. Schematic of main PCB.

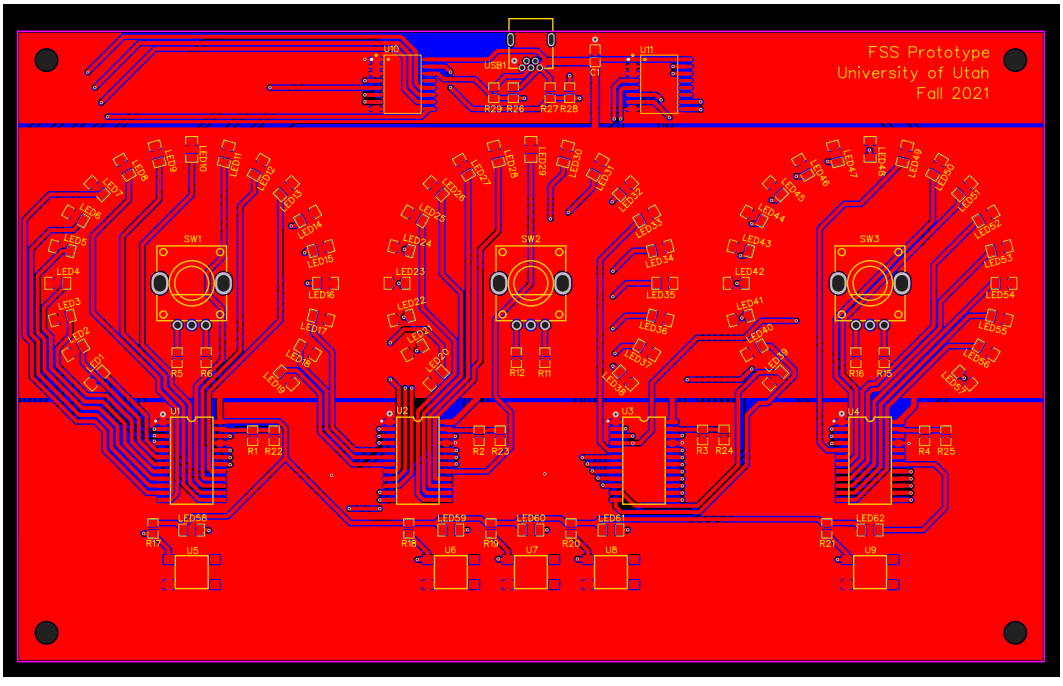

Fig. 9. Layout of main PCB.

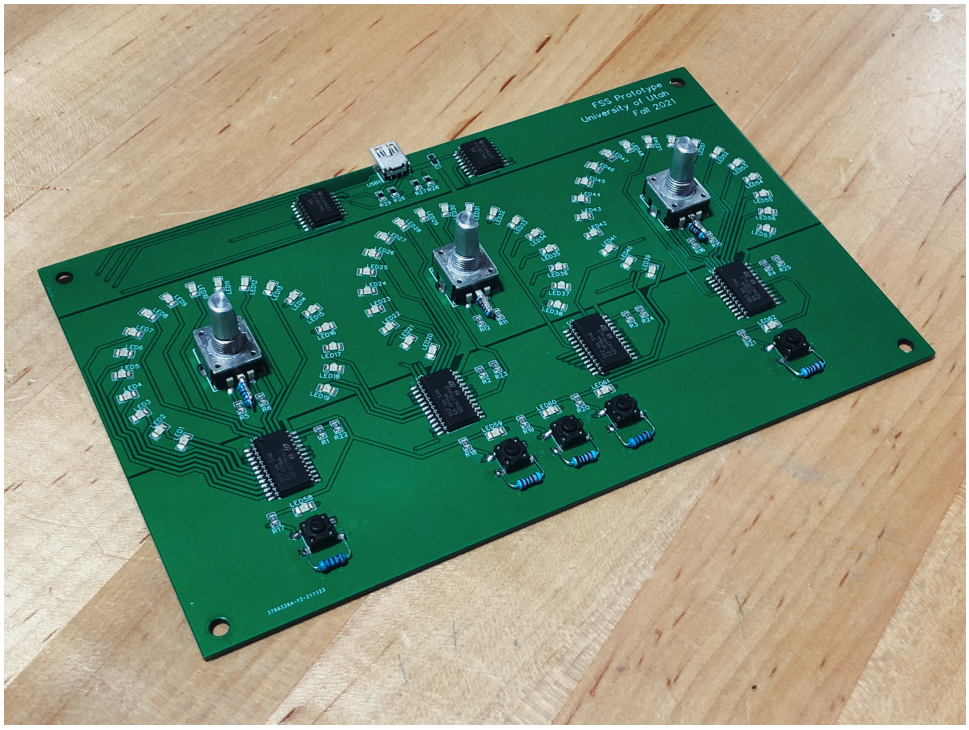

Fig. 10. Picture of main PCB assembly.

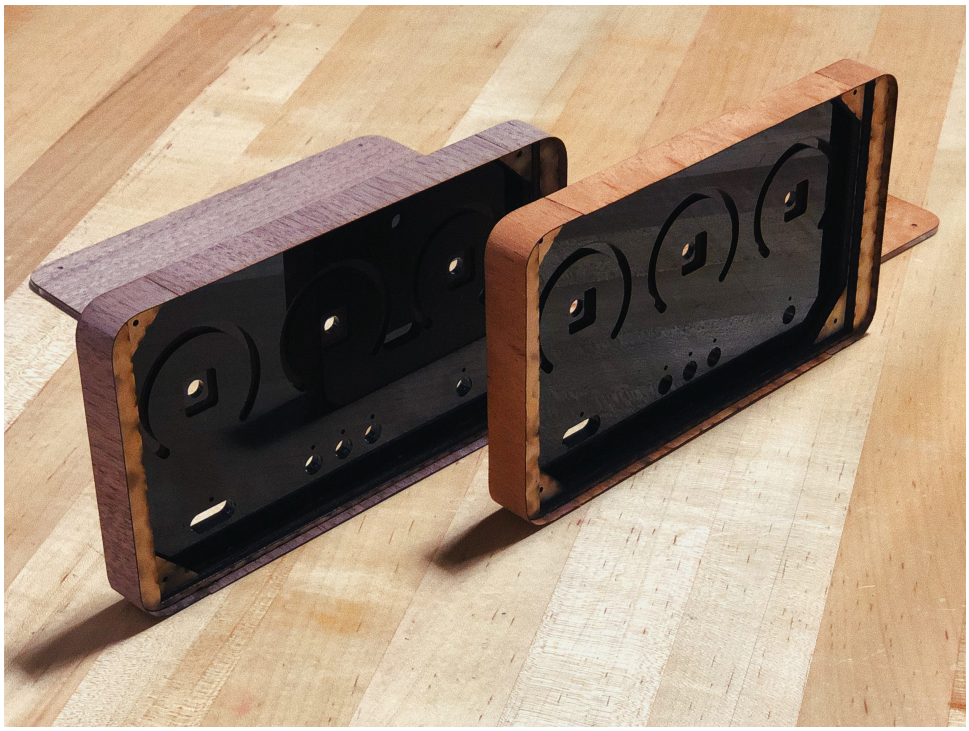

Fig. 11. Picture of FSS housing backside.

This was implemented for two reasons. We want `LOAD` and `STORE` to have access to the whole address space of BRAM, and we want to be able to communicate in a modular way with GPIO registers that are sending and receiving information from the FSS GPIO port expanders over I^2C. These instructions also allow us to read a value from the microsecond counter, which is a separate Verilog module we created to assist with timing. Details on these modules are explained in the "Peripheral Interfacing" section.

2) `CALL/CALLD/RET` *instead of* `JAL/JUC`: Our firmware is quite complex, and, for ease of development, we planned to push and pop instruction addresses on and off the stack to manage nested subroutine calls. It seems to make more logical sense to encapsulate the management of the return address within the instruction. The `CALL` and `CALLD` (`CALL` with displacement) instructions manage the return address implicitly, and `RET` will be an unconditional jump to the previously calculated return address, which is popped off the stack. This way, we don't need a dedicated register for the return address of a subroutine call.
3) `PUSH/POP`: These instructions encapsulate the behavior of modifying the stack pointer and loading/storing data

on the stack. In order for them to work properly, the first few instructions of any program that is run on the CR16 processor must assign an address to the stack pointer register `rsp`. By convention, the stack starts at the highest memory address and grows downward. The stack "lives" in the same file as the code, so its capacity is between the EOF and the final machine-encoded instruction. Our processor has no kernel, and therefore has no notion of segmentation faults, so stack overflow has unintended and sometimes sneaky consequences.

The full ISA is located in the appendix of this document. This ISA may seem unconventional, and differs from most standardized ISAs that fall cleanly into CISC or RISC categories. However, by the process of formulating our ISA, we have learned much about the considerations computer architects must make when designing an ISA. Our final CR16 ISA was assembled with a lot of careful thought and effort. The assembly software was a burdensome aspect of our project, so we are glad we had these features to develop it intuitively and cleanly.

V. Peripheral Interfacing

There are two peripherals that our FSS prototype's firmware communicates with using the `LOADX/STOREX` instructions: the I^{2}C bus and a microsecond counter. The I^{2}C bus simply consists of two open-drain I/O pins — one for SCL and one for SDA. These ports use the `inout` port direction in SystemVerilog to designate them as being used as both an input and an output. To create the open-drain configuration for these pins, the following Verilog code is used:

```
assign O_SCL = I_SCL_T ? 1'bZ : 1'b0;
assign O_SDA = I_SDA_T ? 1'bZ : 1'b0;
```

This essentially creates two ground-driven, tri-state buffers with inverted triggers.

The microsecond counter is simply a clock divider in conjunction with 48-bit counter. Knowing that our CR16 processor clock uses our FPGA board's 50 MHz clock, we can divide that by 50 to create a clock signal with a period of 1 microsecond. Then with every pulse of the divided clock signal, a counter increments by 1.

These two peripherals are SystemVerilog modules that are instantiated in a another SystemVerilog module entitled `ext_mem`. This module maps addresses from the CR16 external memory port to the various external peripheral instantiations, namely: `clock_divided_counter` and `i2c_bus`. When the `LOADX/STOREX` instructions are decoded in the CR16 processor, the external memory port addresses into the peripheral instantiations accordingly so that 16-bit values can be read or written to directly from our assembly code. This is very similar to how memory-mapped I/O is done in commercial-grade microcontrollers and SoCs.

VI. Assembler

Our CR16 processor project includes an accompanying custom-built assembler to compile assembly code written in our specialized ISA. The assembler itself is written in Java and uses the Gradle build tool for dependency management and distribution building. A library entitled "JCommander" is used to handle command-line argument parsing seamlessly. Listing 12 shows the command-line usage of our assembler.

When run, the input file is read into memory and the code lines are sanitized, meaning that all code comments and unnecessary whitespace is removed. Comments in our assembly syntax are delimited using the hashtag symbol (e.g. "#"). Then macros are indexed and processed. A macro is used to create a mapping between a string (the key) and another string (the value). The following shows an example of how a macro is used in our firmware assembly code to define the upper and lower 8-bits of the stack pointer:

```
# Initialize the stack pointer at BRAM
# address 0x0FFF (which is 2^12 - 1)
`define STACK_PTR_LOWER 0xFF
`define STACK_PTR_UPPER 0x0F
```

Macros are used in several places in our firmware assembly code as they are primarily used to give names to numerical constants.

Once macros have been processed, labels are then indexed and processed. Labels are used to reference arbitrary static memory addresses, which allows a programmer to declare named functions and define static memory blocks. Listing 13 following shows an example of how labels are used in our firmware assembly code. This example listing contains an `.array_copy` function implementation that's prepended with our "AssemblyDoc" code documentation. Note that the assembler will compile the `JLO .array_copy:loop` line to a branch instruction since the `.array_copy:loop` label is within the branch instruction's displacement range (which is $\pm$127). Similarly, a `CALL` instruction will compile to a `CALLD` instruction in the event that the given label is within the `CALLD` instruction's displacement range (which is $\pm$2048). Also note that if a programmer uses a `CALL` or jump instruction and the desired label to call/jump to lies outside the displacement range of the equivalent displacement instruction, then the assembler will force the programmer to use an "address loading register" with the following syntax: `JLO .far_away_label$r0`. In this example, the absolute memory address of `.far_away_label` will be loaded in register `r0`. This forces the programmer to make a conscience decision about which register a label address should be loaded into via the `MOVIL` (move-immediate for lower 8-bits) and `MOVIU` (move-immediate for upper 8-bits) instructions.

Finally, the assembler will map all the instructions to their machine code equivalent as the CR16 ISA specifies. This machine code file is written according to the command-line arguments as shown in Listing 12.

VII. Firmware

The following sections discuss the functionality of our firmware assembly code. We divided complex functions into smaller subroutines, allowing for the complexity of our firmware to be abstracted away piece-by-piece. This makes the assembly code much easier to read and is good programming practice.

A. I^{2}C Logic

To interface with the I/O port expander chips via I^{2}C, a number of subroutines to handle writing and reading to and from the I^{2}C bus were programmed. Some simple "getter" and "setter" functions were created for the binary values of SDA and SCL lines. Then `START` and `STOP` conditions were programmed into separate functions to take control and release the bus respectively. A function for getting the acknowledge (ACK) bit from a slave and a function for sending the not-acknowledge (NACK) bit to a slave were also programmed. Lastly, functions for reading and writing arbitrary bits on the bus lines were programmed.

These simpler subroutines are called sequentially in the `.i2c_read_byte` and `.i2c_write_byte` functions.

```
Usage: assembler [options] <assembly code file path>
Options:
  -d, --debug
    Turns on debug mode.
    Default: false
  -p, --max-padding-line
    The line number to which padding lines should be added to an output
    binary.
    Default: 0
  -v, --max-padding-line-value
    The decimal value of the padding lines.
    Default: 0
  -b, --number-base
    The number base of the output binary.
    Default: HEX
    Possible Values: [BINARY, DECIMAL, HEX]
  -o, --output
    The output binary file path. Defaults to <input assembly file>.dat.
  -s, --output-processed
    True to write the processed assembly to <output binary file
    path>.processed.asm.
    Default: false
```

Fig. 12. Command-line usage of CR16 Assembler.

```
##
# Copies the array at 'r11' to 'r12' with length 'r13'.
#
# @param r11 - a pointer to the array to copy from
# @param r12 - a pointer to the array to copy to
# @param r13 - the number of words to copy
#
# @return void
##
.array_copy
    MOVIL   r0  0x00
    MOVIU   r0  0x00

    .array_copy:loop

    LOAD    r5  r11
    STORE   r12  r5

    ADDI    r11 1
    ADDI    r12 1

    ADDI    r0  1
    CMP     r0  r13
    JLO     .array_copy:loop

    RET
```

Fig. 13. The ".array_copy" function from our firmware assembly code.

These function signatures are shown in Listings 14 and 15 respectively. The argument list, as shown in the function's AssemblyDoc, gives a sense of how easily it is to read and write a byte to an addressable slave on the I^2C bus from anywhere in our firmware assembly code. Note that in our setup, the FPGA board serves as the I^2C bus master and the I/O port expander ICs serve as the slaves. More information on the I^2C protocol can be found here.

B. Microsecond Counter

The microsecond counter module was added as a distinct Verilog module. The counter is 48 bits wide, and acts like a peripheral to the FSS device. We determined this to be a necessary bit width because 2^{32} microseconds is only about 71 minutes, and the software could misbehave if the counter ever resets. Since 2^{48} microseconds is about 8.9 years, the counter should never reset while the device is operating. Because the counter acts like a peripheral, its 48-bit count is stored in a set of 3 discrete 16-bit registers which can be loaded into the processor's regfile using the `LOADX` instruction. When the microsecond counter data is fetched, it can be used to store timestamps and calculate elapsed time. This is only feasible if the software can accommodate 48-bit unsigned subtraction. Thankfully unsigned subtraction and 2's complement subtraction are congruent. Therefore, we can use the `SUB` instruction to subtract the least significant bits first, and then propagate any borrows to the next most significant bits. A borrow will never occur out of the MSB since the time elapsed will usually never reset to a lower number.

The most significant advantage of this functionality is that we can implement a helper function in the assembly that "sleeps" the processor by a certain number of microseconds. This behavior is imperative to ensure correct protocol behavior with I^2C communication and LED animations.

C. Rotary Encoder Polling

To retrieve the state of the three rotary encoders, the port expander chip connected to the rotary encoder channel pins is polled and a decoding function is called to determine its rotational direction — either clockwise or counter-clockwise. Our chosen rotary encoders output a quadrature-encoded signal. Quadrature encoding is performed using two distinct bit values, where each value represents the state of a channel in the rotary encoder device. Fig. 16 shows the schematic of our rotary encoders with a suggested filter circuit. From a high level, the rotational direction can be determined by the appearance of the timing diagram for the two distinct signals. Fig. 17 illustrates how a timing diagram may look when turning a rotary encoder in the clockwise direction, given that terminal A is the upper waveform and terminal B is the lower waveform.

The assembly code uses a static memory address in BRAM to store the previously obtained state of each rotary encoder, and the "decode" functions works to detect whether or not the rotary encoder has traversed a whole period of the quadrature output waveform. If it has, then the control knob has been moved far enough to turn on or off an LED in the ring display.

D. LED Animations

One of our goals was to interface with the audio codec on our FPGA boards to generate sound, just as a commercial music synthesizer would, although, this goal was secondary. Due to time constraints, we instead implemented some very cool animations for the LEDs in the ring displays on the FSS prototype. That includes a start up animation and 5 "idle" animation sequences. The pause/play button on the FSS was originally intended to pause and play the sound generated by the audio codec, but is now used to toggle between the various "idle" animation sequences. Some of these sequences programatically shift a certain number of 1's and 0's into the LED driver shift register with various delays. Other animation sequences use frame data located in static memory. These frames are encoded with the values (0 - 19) of the 3 ring displays and an animation execution function, entitled "`.execute_animation_sequence`" steps through the frames with a 2 millisecond delay, decoding the frame and updating the LED driver shift registers accordingly.

E. Business Logic and Final Device Integration

With all of the sophisticated communication and data processing abstracted away, the business logic of the main routine actually becomes quite simple. On initialization, the FSS ring displays light up with a brief startup animation. When the animation finishes, the FSS program loads program 1. The default program values are hard-coded into the firmware in a static location in memory which is referenced with a label. Because the FPGA must be reprogrammed every time it is turned off, we do not yet have a way to allow saved programs to persist after shutdown, thus the saved program values only persist throughout program runtime. The firmware then enters into an infinite loop which integrates all the logic explained above to poll the state of the inputs from buttons and rotary encoders, process that data, and push the correct chain of values into the LED driver shift registers. All of this occurs in a short time period, so no concurrency or event listening is necessary. The user receives practically instantaneous feedback to any stimulus applied to the dials or buttons.

VIII. Lessons Learned

As a team, we feel that we have acquired many skills from the work we had to do for this project. Some of the most memorable lessons are learned when we make mistakes, and we would like to address a number of those lessons. These concepts apply to the software, electronics, and physical assembly aspects of our project, as well as our project management skills.

1) Schematic Imperfections: Unfortunately, we added a series resistor in the wrong place to our push button and the rotary encoder channel pull-down configurations on our main PCB. Due to the fact that we were so concerned about accidentally shorting our I/O port expander chips, we placed resistors for short-circuit protection instead of resistors for normal operation. This issue was rectified easily by soldering through-hole resistors in series to the push button and rotary encoder channel pull-down configurations.
2) Common Ground: We neglected one important detail about I^2C when we designed our breakout PCBs — all masters and slaves on the bus need a common ground. The `GND` wire as shown in Figure 4 was added after the PCBs were fabricated since the I^2C requirement of a common ground was neglected during the schematic development process. This issue was easily rectified by soldering a wire to the `OUT-` pin of the buck converter which could then be connected to the FPGA board's ground pin on the PMOD pin header.
3) Rotary Encoder Filter Circuit: The datasheet for our rotary encoders contains a schematic for a suggested filter circuit, as seen in Fig. 16. We didn't implement this filter circuit under the impression that, due to our sampling period, the rotary encoder channels would be "pseudo-debounced." During the hardware debugging process

```
##
# Requests to read a byte from a slave on the I2C bus.
#
# @param r11 - the 7-bit address of the I2C slave
#
# @return r10 - the byte read from the I2C bus or '0x0100' if a byte could not be read
##
.i2c_read_byte
```

Fig. 14. The ".i2c_read_byte" function signature from our firmware assembly code.

```
##
# Requests to write a byte to a slave on the I2C bus.
#
# @param r11 - the 7-bit address of the I2C slave
# @param r12 - the byte to write to the I2C slave
#
# @return r10 - 1 if successful, 0 if byte could not be written
##
.i2c_write_byte
```

Fig. 15. The ".i2c_write_byte" function signature from our firmware assembly code.

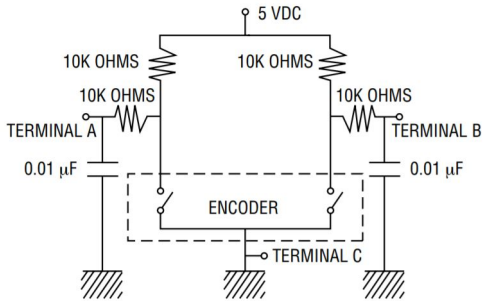

Fig. 16. Schematic for PEC11R Series rotary encoders as seen in [3]

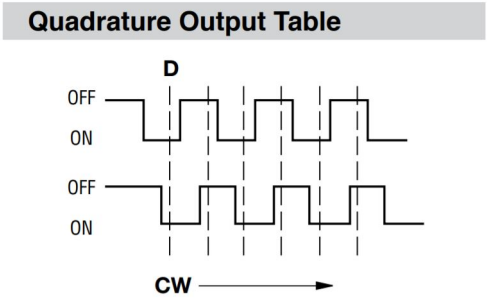

Fig. 17. Waveform for a quadrature-encoded signal as seen in [3]

however, we observed some noise around the channel transition thresholds, and we believe that a combination of the filter circuit and software debouncing would alleviate this issue.

4) Water Jet Issues: Our first design was to utilize an anodized aluminum plate for the top panel of the device. We contracted a company here in the Salt Lake Valley to use a water jet to route out the windows for the LED lights and push buttons. When we received the plates, there was evidence of a low-quality water jet job, with imprecise cuts and extremely rough edges. This was a disappointing waste of time, money, and materials. To combat this, we could have invested time in visiting the facility to view samples of the water jet work on anodized aluminum before committing to have them do our work.
5) Wood is an Unreliable Material: Wood dimensions are not always as advertised depending on the climate and humidity of the supplier. In our particular application, the wood layers were too thin, and bowing made the assembly process unnecessarily difficult. Real wood veneer was also quite fragile, though the aesthetic was pleasing.
6) Acrylic is an Higher-Cost Material: To compensate for the challenges we faced with wood, we built the walls of the housing using layers of laser-cut acrylic sheets. This resulted in significant material waste, and was much more expensive than wood. In retrospect, something that could be 3D modelled and cast out of plastic would be much cheaper and more realistic for a preliminary prototype design, though the surface acrylic panelling was professional and sleek.
7) Time Management: Many aspects of this project presented significant pressure for our group because of the ambitious nature of our project. We underestimated the amount of time we would spend testing, debugging, and correcting our CR16 processor and other software elements. As a rule of thumb, we should have doubled the time that we initially thought it would take to perform any task.
8) Foam Buffers for Structural Support: The bottom panels make direct contact with foam buffers in the corners near the assembly screws, and this detracts from the rigidity of the rest of the design. The bottom plate can be pressed up into the rest of the device. To prevent this, the bottom plate should make direct contact with a rigid corner material and fit tightly. This issue would be rectified in a future prototype.

IX. Videos

We wanted to create a professional video in the form of a product advertisement showing the beauty we captured in the FSS prototype's design. That video is available here. Additionally, a demo video showing the basic usage and functional overview of the FSS prototype is available here.

X. About the Team

There were many tasks involved in creating a functioning synthesizer, and due to this there was much overlap in the division of labor. To list a few:

- Concept Art
- Assembly Programming
- SystemVerilog Programming
- Test Programs
- Demo Programs
- PCB Design
- PCB Assembly and Reflow
- Assembler
- External Housing (Construction)
- External Housing (Design)
- Breakout Box Construction

Jacob and Nate covered much of the software, namely writing the FSS prototype Verilog modules, the `fss.asm` assembly source code, creating demo programs, and implementing the core functionality of the synthesizer. They also worked together to physically assemble, reflow, and test the PCBs, and design the schematic for the breakout box PCB.

Jacob was also responsible for the peripheral interfacing — writing the I^2C procedure in assembly — and writing the assembler. He worked with Brady to design the Main FSS Board schematic.

Brady conceptualized the idea of a fully synchronized synthesizer and created all of the concept art. He also had a hand in creating test programs, designing the PCB, and designing and constructing the external housing. He and Jacob selected all of the device materials and acquired datasheets in their efforts to design the PCB and physical device housing.

Isabella assisted in creating test programs and constructing the external housing. She also created the synthesizer's breakout box and was responsible for modeling components in SolidWorks before 3D printing.

XI. Conclusion and Future Work

Overall the final synthesizer came together well and the program reflects most our initial requirements. We were able to design and create a machine that properly updates parameters based on user inputs and can save those into memory to be loaded again when needed later. The final physical product is sleek, fully functional, and is simple to use.

In this version of our synthesizer, we did not have time to work with the audio codec on the FPGA. In a future implementation we hope to get this working so that our synthesizer may play music and have the three dials correspond to audio settings that adjust the sound of the music playing. Beyond this, we would also like to add more inputs, than just the three simple dials. Other commercially available synthesizers have numerous knobs and switches that give users extensive control over the sound the synthesizer makes, and three knobs simply isn't enough to do this.

With these changes, our team could create a high quality synthesizer that has just as much form as function.

References

[1] Github, Github Repository for "CompactRISC16", Fall 2021, Online.

[2] Github, Github Repository for "FSSPrototype", Fall 2021, Online.

[3] PEC11R Series 12mm Incremental Encoder, Bourns, PEC11R-4015F-N0024, Online.

[4] DE1-SoC User Manual, TerAsic Technologies, April 20216, Online.

Appendix

Custom CompactRISC16 (CR16) Instruction Set Architecture (ISA)

Computer Design Laboratory ECE 3710 Group 2
Fall 2021
The University of Utah

Table 1: Assembly Instructions and Machine Encodings

Mnemonic	Operands	Function	Opcode [15:12]	Rdest [11:8]	ImmHi/ Opcode Ext [7:4]	ImmLo/ Rsrc [3:0]	Notes	Clock Cycles
ADD	Rdest, Rsrc	`Rdest = Rdest + Rsrc`	0000	Rdest	0000	Rsrc		3
ADDI	Rdest, Imm	`Rdest = Rdest + Imm`	0001	Rdest	ImmHi	ImmLo	Sign extended Imm	3
ADDC	Rdest, Rsrc	`Rdest = Rdest + Rsrc + 1`	0000	Rdest	0001	Rsrc		3
ADDCI	Rdest, Imm	`Rdest = Rdest + Imm + 1`	0010	Rdest	ImmHi	ImmLo	Sign extended Imm	3
MUL	Rdest, Rsrc	`Rdest = Rdest * Rsrc`	0000	Rdest	0010	Rsrc		3
MULI	Rdest, Imm	`Rdest = Rdest * Imm`	0011	Rdest	ImmHi	ImmLo	Sign extended Imm	3
SUB	Rdest, Rsrc	`Rdest = Rdest - Rsrc`	0000	Rdest	0011	Rsrc		3
SUBI	Rdest, Imm	`Rdest = Rdest - Imm`	0100	Rdest	ImmHi	ImmLo	Sign extended Imm	3
CMP	Rdest, Rsrc	`Rdest - Rsrc`	0000	Rdest	0100	Rsrc		3
CMPI	Rdest, Imm	`Rdest - Imm`	0101	Rdest	ImmHi	ImmLo	Sign extended Imm	3
NOT	Rdest, Rsrc	`Rdest = !Rsrc`	0000	Rdest	0101	Rsrc		3
NOTI	Rdest, Imm	`Rdest = !Imm`	0110	Rdest	ImmHi	ImmLo	Zero extended Imm	3
AND	Rdest, Rsrc	`Rdest = Rdest & Rsrc`	0000	Rdest	0110	Rsrc		3
ANDI	Rdest, Imm	`Rdest = Rdest & Imm`	0111	Rdest	ImmHi	ImmLo	Zero extended Imm	3
OR	Rdest, Rsrc	`Rdest = Rdest | Rsrc`	0000	Rdest	0111	Rsrc		3
ORI	Rdest, Imm	`Rdest = Rdest | Imm`	1000	Rdest	ImmHi	ImmLo	Zero extended Imm	3
XOR	Rdest, Rsrc	`Rdest = Rdest ^ Rsrc`	0000	Rdest	1000	Rsrc		3
XORI	Rdest, Imm	`Rdest = Rdest ^ Imm`	1001	Rdest	ImmHi	ImmLo	Zero extended Imm	3
LSH	Rdest, Ramount	`Rdest = Rdest << Ramount`	0000	Rdest	1001	Ramount	$0 \leq$ Ramount ≤ 15 since registers are only 16-bits	3
LSHI	Rdest, ImmLo	`Rdest = Rdest << Imm`	0000	Rdest	1010	ImmLo	$0 \leq$ ImmLo ≤ 15	3
RSH	Rdest, Ramount	`Rdest = Rdest >> Ramount`	0000	Rdest	1011	Ramount	$0 \leq$ Ramount ≤ 15	3
RSHI	Rdest, ImmLo	`Rdest = Rdest >> Imm`	0000	Rdest	1100	ImmLo	$0 \leq$ ImmLo ≤ 15	3
ALSH	Rdest, Ramount	`Rdest = Rdest <<< Ramount`	0000	Rdest	1101	Ramount	$0 \leq$ Ramount ≤ 15	3
ALSHI	Rdest, ImmLo	`Rdest = Rdest <<< Imm`	0000	Rdest	1110	ImmLo	$0 \leq$ ImmLo ≤ 15	3
ARSH	Rdest, Ramount	`Rdest = Rdest >>> Ramount`	0000	Rdest	1111	Ramount	$0 \leq$ Ramount ≤ 15	3
ARSHI	Rdest, Imm	`Rdest = Rdest >>> Imm`	1111	Rdest	0000	ImmLo	$0 \leq$ ImmLo ≤ 15	3
MOV	Rdest, Rsrc	`Rdest = Rsrc`	1111	Rdest	0001	Rsrc	Copies Rsrc into Rdest	3
MOVIL	Rdest, Lower Imm	`Rdest[7:0] = Imm`	1010	Rdest	ImmHi	ImmLo	Zero extended Imm, moves immediate value into lower bits of Rdest	3
MOVIU	Rdest, Upper Imm	`Rdest[15:8] = Imm`	1011	Rdest	ImmHi	ImmLo	Zero padded Imm, moves immediate value into upper bits of Rdest	3
J[condition]	Rtarget	`if [condition]:` `PC = Rtarget`	1111	condition	0010	Rtarget	[condition] bit patterns are in Table 2.	T: 5 F: 3
B[condition]	Displacement Imm	`if [condition]:` `PC += Imm + 1`	1100	condition	ImmHi	ImmLo	[condition] bit patterns are in Table 2. Immediate is sign extended 2's complement for program counter/address displacement.	T: 4 F: 3
CALL	Rtarget	`Pushes PC + 1 onto stack,` `PC = Rtarget`	1111	xxxx	0011	Rtarget	Used for nested subroutines	5

CALLD	Displacement Imm	`Pushes PC + 1 onto stack, PC += Imm + 1`	1101	ImmHi	ImmMid	ImmLo	Used for nested subroutines. Immediate is sign extended 2's complement for program counter/address displacement.	4
RET		`Pops top of stack into PC`	1111	xxxx	0100	xxxx	Used to return from nested subroutine	6
LPC	Rdest	`Rdest = PC`	1111	Rdest	0101	xxxx	Sets Rdest to PC	3
LSF	Rdest	`Rdest = status flags`	1111	Rdest	0110	xxxx	Sets Rdest to the current status flags (zero extended)	3
SSF	Rsrc	`Status flags = Rsrc[4:0]`	1111	xxxx	0111	Rsrc	Sets the current status flags to `Rsrc[4:0]`	3
PUSH	Rsrc	`Main memory value at rsp = Rsrc, rsp--`	1111	xxxx	1000	Rsrc	Pushes Rsrc onto top of stack	4
POP	Rdest	`rsp++, Rdest = Main memory value at rsp`	1111	Rdest	1001	xxxx	Pops top of stack into Rdest	4
LOAD	Rdest, Raddr	`Rdest = Main memory value at Raddr`	1111	Rdest	1010	Raddr	Used to load data at Raddr into Rdest from main memory	4
STORE	Raddr, Rsrc	`Main memory value at Raddr = Rsrc`	1111	Raddr	1011	Rsrc	Used to store data at Raddr from Rsrc to main memory	4
LOADX	Rdest, Raddr	`Rdest = External memory at Raddr`	1111	Rdest	1100	Raddr	Used to load data at Raddr into Rdest from external/peripheral memory/registers	4
STOREX	Raddr, Rsrc	`External memory value at Raddr = Rsrc`	1111	Raddr	1101	Rsrc	Used to store data at Raddr from Rsrc to external/peripheral memory/registers	4
NOP		`No Operation`					Pseudo instruction for: `OR R0, R0`	3

Note that during the execution cycle of an instruction, PC (the "Program Counter") always points to the next instruction (e.g. `PC + 1`). As a result of this, `B[condition]` and `CALLD` will displace the current PC by `Imm + 1`. This can be thought of as executing the "next instruction" after PC displacement.

Table 2: Bit Patterns of Conditions for B[condition] and J[condition]

Mnemonic	Bit Pattern	Description	Function	Status Flags
EQ	0000	Equal	`Rsrc == Rdest`	`Z=1`
NE	0001	Not Equal	`Rsrc != Rdest`	`Z=0`
CS	0010	Carry Set	`C == 1`	`C=1`
CC	0011	Carry Clear	`C == 0`	`C=0`
FS	0100	Flag Set	`F == 1`	`F=1`
FC	0101	Flag Clear	`F == 0`	`F=0`
LT	0110	Less Than	`signed: Rdest < Rsrc`	`N=0 and Z=0`
LE	0111	Less than or Equal	`signed: Rdest <= Rsrc`	`N=0`
LO	1000	Lower than	`unsigned: Rdest < Rsrc`	`L=0 and Z=0`
LS	1001	Lower than or Same as	`unsigned: Rdest <= Rsrc`	`L=0`
GT	1010	Greater Than	`signed: Rdest > Rsrc`	`N=1`
GE	1011	Greater than or Equal	`signed: Rdest >= Rsrc`	`N=1 or Z=1`
HI	1100	Higher than	`unsigned: Rdest > Rsrc`	`L=1`
HS	1101	Higher than or Same as	`unsigned: Rdest >= Rsrc`	`L=1 or Z=1`
UC	1110	Unconditional		`N/A`
	1111	Never Jump		`N/A`

Table 3: Register Naming and Conventions

Register Index	Register Name	Meaning
`4'd15`	`rsp`	Stack pointer with an address starting at `0xFFFF` (2^{16}) and grows downward towards dynamically allocated memory
`4'd14`	`r14`	4th subroutine argument
`4'd13`	`r13`	3rd subroutine argument
`4'd12`	`r12`	2nd subroutine argument
`4'd11`	`r11`	1st subroutine argument
`4'd10`	`r10`	Return value of subroutine
`4'd9`	`r9`	Caller-owned
`4'd8`	`r8`	Caller-owned
`4'd7`	`r7`	Caller-owned
`4'd6`	`r6`	Caller-owned
`4'd5`	`r5`	Callee-owned
`4'd4`	`r4`	Callee-owned
`4'd3`	`r3`	Callee-owned
`4'd2`	`r2`	Callee-owned
`4'd1`	`r1`	Callee-owned
`4'd0`	`r0`	Callee-owned