Expanded FPGA Synthesizer Final Report

Mel Murphy

David Hanson

Jonathan Law

Portland State University
ECE 544 Final Project
Spring 2021

Project Overview

For our final project, we designed and built an FPGA-based digital synthesizer. We built the system on a Nexys A7-100T board which contains an Artyx-7 FPGA. Using this FPGA, we implemented an embedded system using a Microblaze soft core processor and several peripherals. The system runs on FreeRTOS, and utilizes frequent interrupts and task switching.

The synthesizer is able to generate audio frequency PCM waveforms by calculating the 24-bit samples in software, and sending them at a 48 kHz sample rate to an I2S module connected to a PMOD port. The I2S module contains a DAC, which translates the samples to an analog audio signal and makes that audio signal available through a standard 3.5mm phone connector. The user can connect headphones or a speaker to this port.

The synthesizer can receive MIDI input signals and translate them into the corresponding notes. Alternatively, the synthesizer is capable of generating output signals using the MIDI protocol which can signal an external instrument to generate its own sounds. The MIDI input utilizes a combination of modified and novel dedicated MIDI processing hardware, and the MIDI output utilizes a UART. Both signals are accessible through a PMOD port, and are connected through a level shifter (from the PMOD 3.3 V to the MIDI 5 V level) to a MIDI DIN connector breakout board.

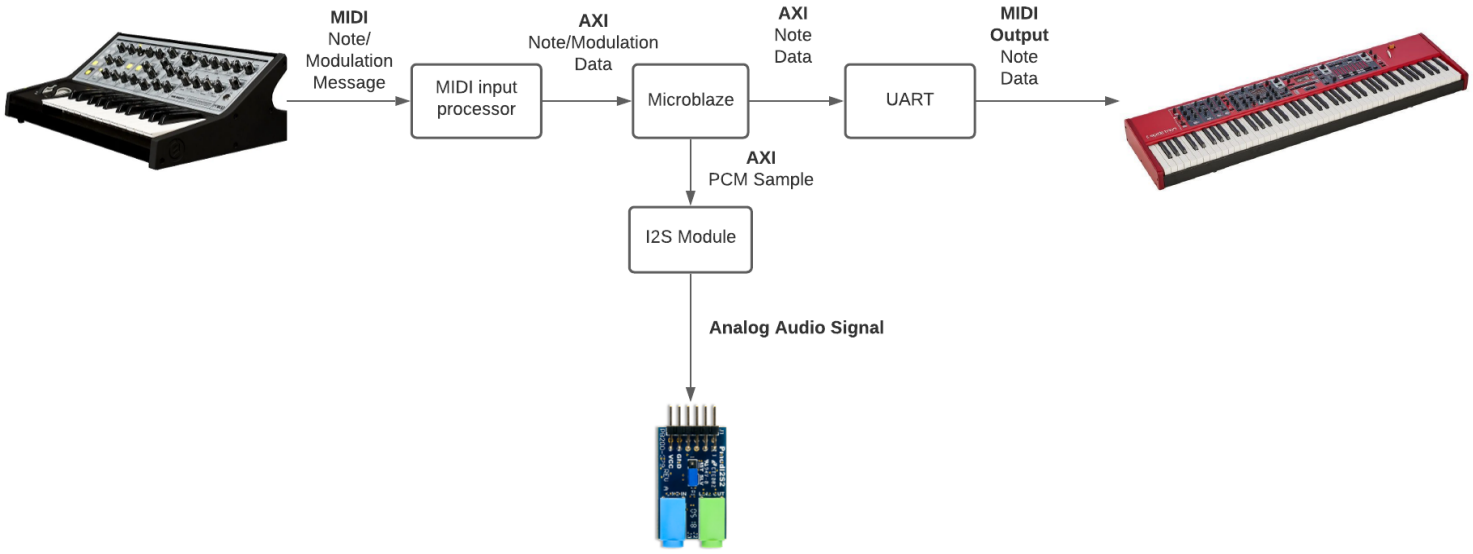


Figure 1: Audio/MIDI signal path

To expand this capability, we added sequencer functionality that enables the synthesizer to store a series of notes and play them back. This is done either through the onboard

waveform synthesis, or by sending corresponding MIDI messages to an external instrument. Sequencer parameters are visible on an OLED display connected to a PMOD port, and can be customized using a combination of the pushbuttons and a rotary encoder that is also connected to a PMOD port. Up to 16 sequences and their corresponding customizable parameters--tempo, note subdivision, swing, and pattern--can be stored to and loaded from onboard memory using the menu system shown on the display. This system is also used to set the system parameters, volume and mode, and to load and store sequences.

Block Diagram

See embsys.pdf for a detailed block diagram of the project.

Theory of Operation: Hardware

The system's hardware consists of several main parts: the processor, interrupt controller, clock generator, input controls, memory, timers, OLED display, MIDI output, MIDI input processor, and I2S output.

Processor

The system uses a Microblaze soft core processor. For this project, we expanded its capabilities beyond the defaults so that it would more efficiently calculate the audio sample values at the desired rate. Key enhancements include the barrel shifter, integer divider, extended floating point unit, and the branch target cache.

Interrupt Controller

Our Microblaze receives an interrupt from the controller when it is signaled by 6 different sources: AXI Timers 0 and 1, the MIDI Input Processor, the GPIO Module, the I2S Synchronizer, or the PMOD Rotary Encoder. Note that AXI Timer 0 is only used by freeRTOS, so our system accounts for 5 of the 6 interrupts.

Clock Generator

The clock generator outputs three different clocks: a 100 MHz system clock for the processor and AXI modules, a 50 MHz clock for the OLEDrgb SPI bus, and an 18.432 MHz clock for the audio clock domain. We also instantiated a second system reset module to

ensure that the audio clock domain modules receive a reset synchronous with that of the 100 MHz domain.

Input Controls

The user is able to change the system state using the pushbuttons, switches, and the PMOD rotary encoder connected to the PMOD JC pins. The pushbuttons and switches are connected to a GPIO module, which sends a level-triggered interrupt to the processor when the button or switch states change. We modified the PmodENC544 module provided for the course so that it would send an edge-triggered interrupt when the button, switch, or rotary encoder states change. To do so, we included an extra port, *int_sig*, which we connected to the interrupt controller. This signal is asserted when the button or switch changes state, or when there's a rotary event. It's then deasserted at the next clock edge.

```
17          // Users to add ports here
18          input wire encA,     // A and B quadrature inputs from PmodENC
19          input wire encB,
20          input wire encBTN,   // pushbutton input from PmodENC
21          input wire encSWT,   // slide switch input from PmodENC
22          output reg int_sig,
23          // User ports ends
```

Figure 2: PMOD encoder expanded ports

```
438     // implement the rotary count register
439     always @(posedge S_AXI_ACLK or posedge clr_rotary_cnt) begin
440         if (clr_rotary_cnt)
441             rotary_count <= 32'd0;
442         else if (rotary_event) begin
443             int_sig <= 1'b1;
444             if (rotary_left)
445                 rotary_count <= rotary_count - 1;
446             else
447                 rotary_count <= rotary_count + 1;
448         end
449         else begin
450             rotary_count <= rotary_count;
451             btn_sw_prev <= {db_encSWT, db_encBTN};
452             if (btn_sw_prev != {db_encSWT, db_encBTN}) begin    // check if button or switch value has changed
453                 int_sig <= 1'b1;                                // send interrupt signal
454             end
455             else begin
456                 int_sig <= 1'b0;                                // deassert interrupt signa at next clock edge
457             end
458         end
```

Figure 3: Interrupt signal handling

Memory

To account for memory limitations we encountered running a fairly complex program on FreeRTOS with just a 128 KB maximum Microblaze memory size, we added memory to

the system using the block ram IP. We added a dual-port BRAM module which we used to contain the audio sample buffer, and we added a single-port BROM module which contained a pre-loaded sequence of MIDI data to be accessed in MEM mode. Use of the Nexys A7's external DDR2 SDRAM memory was investigated for this purpose as well. The BRAM option ended up being the better option for our purposes. With the four BRAM blocks added to the IP (two for the memory and two for the controllers) BRAM usage is still quite low at about 30%. If larger blocks of memory were required, the DDR2 is still available. Using the BRAM option also has the advantage of being relatively easy to interface with. The controller blocks connect directly to the AXI interface and as such, are communicated with in much the same way as the other AXI peripherals.

```
void bufferSample(uint32_t sample, uint32_t index)
{
   XBram_WriteReg(BUFFER_BRAM_BASEADDR, index, sample);
   return;
}
```

Figure 4: Writing to BRAM in signal_generator.c

The figure above provides an example of writing to the audio sample buffer BRAM. The other memory block is read only and is intended to provide an option to store larger sequences of notes. Loaded into the memory with the release of this project is the intro to Bachs' Prelude in C Major.

Timers

We included two timers in this system. The first, AXI Timer 0, is used exclusively by FreeRTOS for system tick management. The second, we use to manage the sequencer and LED blink timing. Both generate level-triggered interrupts when they count down to zero.

OLED Display

To produce a navigable menu system, we connected the PMOD OLEDrgb display to the PMOD JB pins. It receives messages over a SPI connection, clocked at 50 MHz.

MIDI Output

MIDI messages are sent over a UART. Since the MIDI baud rate of 31,250 wasn't among the available baud rate options for the UARTlite, we use a UART16550 module with only the TX output used. This *midi_out* signal is connected to pin JD4, then through a level shifter to the MIDI TX pin on the MIDI breakout board, to which VCC and GND are also connected. The signal then traverses a 5-pin MIDI DIN cable (only 3 pins actually have a function) to reach the input of the desired instrument.

MIDI Input Processor

MIDI input was made possible by a combination of pre-existing open-source hardware and a module we designed. To read the MIDI bytes, we adapted a module designed by Andy West of Element14. The module receives a MIDI RX signal from the PMOD JD1 pin and imitates the functionality of a UART receiver, but without the added customizability and functionality unnecessary for a MIDI connection. We modified this module so that it sends an edge-triggered interrupt at a modulation message or a note change. We also modified it so that instead of reading the signal for NOTE ON, NOTE OFF, and modulation messages and using those to determine the number of ticks between successive predetermined samples, the module sends the NOTE ON and NOTE OFF signals through to a second module, the polyphonic processor. This module keeps track of which notes are currently active and which have been turned off.

The polyphonic processor takes note messages as its inputs, and outputs the value of a register which stores the MIDI note value of each active note. It also outputs an edge-triggered interrupt when the state of its note register changes. It's capable of keeping track of 8 concurrent voices. Essentially, we needed a module that will:

- Store note values that correspond to new NOTE ON messages to a note data register
- Ignore NOTE ON messages for notes already stored
- Clear note values from the register that match a NOTE OFF message
- Ignore NOTE OFF messages that don't correspond to a note value in the register

Since MIDI note data is 7 bits, we decided to store each MIDI note as a byte with one bit of padding. For 8 voices, we created a 64 bit register. To implement this module, we used a one-hot encoded state machine with 12 states.

```
26      // encoding of machine states
27      typedef enum logic [11:0] {
28          WAIT        = 12'b000000000001,
29          FETCH       = 12'b000000000010,
30          C0          = 12'b000000000100,
31          C1          = 12'b000000001000,
32          C2          = 12'b000000010000,
33          C3          = 12'b000000100000,
34          C4          = 12'b000001000000,
35          C5          = 12'b000010000000,
36          C6          = 12'b000100000000,
37          C7          = 12'b001000000000,
38          ADD         = 12'b010000000000,
39          SUB         = 12'b100000000000
40      } states_t;
```

Figure 5: PolyphonicProcessor.sv machine states

The module is initialized to the WAIT state, with a *current_note* value of 0. If the input note value differs from the *current_note* value, it moves to the FETCH state, at which point it stores the value of the note data on its input pins to an internal register. The register then moves through states C0 to C7, corresponding to each byte in the note data register. It checks these bytes in sequence to see if the note it stored is the same as any of these notes. If the incoming message is a NOTE ON message and it matches a note in the register, we ignore the note and return to the WAIT state. If the incoming message is a NOTE ON message and the machine traverses C0-C7 without finding a match, it moves to the ADD state where it stores the note data to the first open spot in the note data register. If the incoming message is a NOTE OFF message and it matches a value in the NOTE DATA register, we remove that value from the register. If the message is a NOTE OFF message and it doesn't match any of the 8 note values in the register, we ignore the message and move back to the WAIT state. Note that as the FSM traverses the states C0 to C7 and checks the note data register for matches, it keeps track of the current index and the index of the last note data slot which contains a value of zero. This way, when it goes into the ADD or SUB states, it's able to place a new note in an empty slot or remove an existing note from the correct index. The note data register is exposed to the top MIDI input module, as is the modulation data register, so that they can be read over AXI.

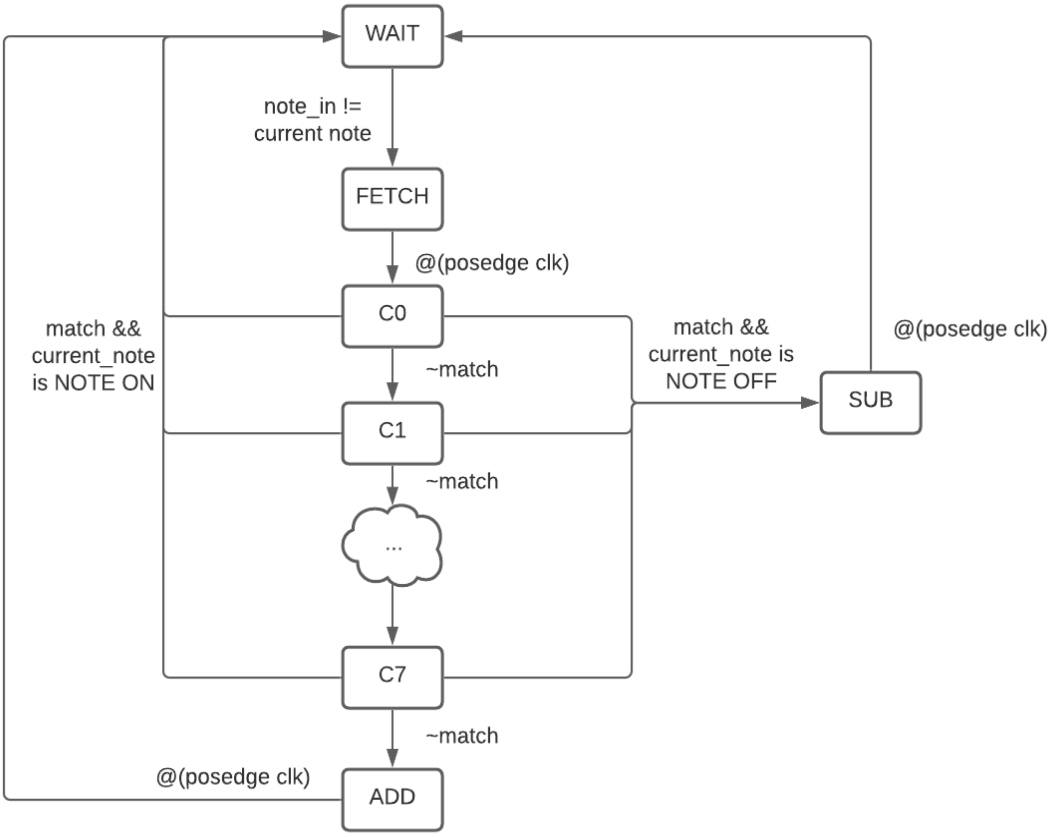


Figure 6: PolyphonicProcessor.sv state machine

I2S Output

We also built a custom module to handle the I2S output. The module consists of a synchronizer and a standard I2S clock designed to receive 24-bit samples of data at a 48 kHz sample rate and read out the individual bits over a serial data out line. To do so, it uses a word clock--also called an LR clock, as it generates stereo audio by toggling on output to the left and right channels--with a frequency equal to the sample rate. It shifts out individual bits at 48 times that frequency--24 bits per word clock toggle. For our module, this required a serial clock of 2.304 MHz. Since the PMOD I2S2 module requires an integer multiple of our serial clock (bit clock) rate, we set our our audio clock domain to 18.432 MHz, or 384 * 48 kHz. The main clock *mclk*, the word clock *lrclk*, the serial clock *sclk*, and the serial data out line are connected to the PMOD port to which the PMOD I2S2 module is connected.

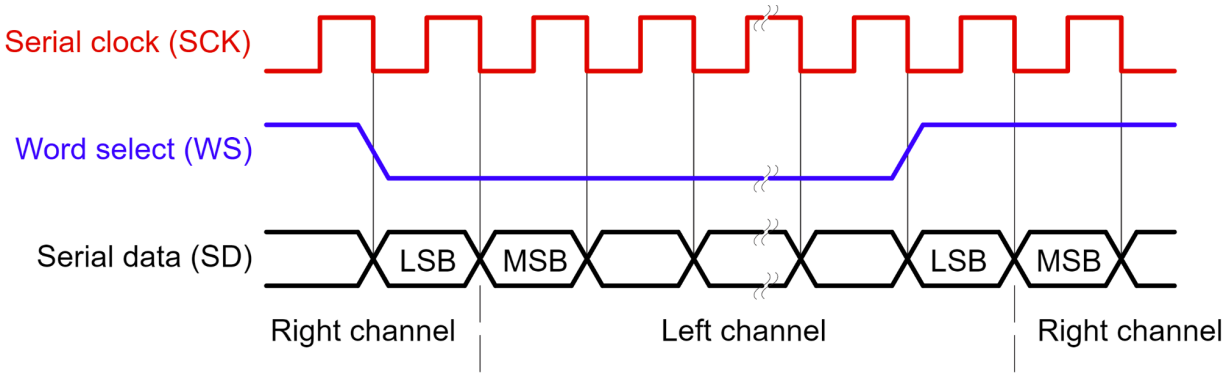


Figure 7: I2S protocol

Since the I2S module operates in a separate clock domain from the rest of our system, we designed a synchronizer to send data from the 100 MHz system clock domain to the 18.432 MHz audio clock domain. We modeled our synchronizer after a design provided in our ECE 540 lecture notes. This synchronizer places data on the bus and then sends a handshake signal through a two register synchronizer so that signals appear in the other clock domain after the data is ready on the bus.

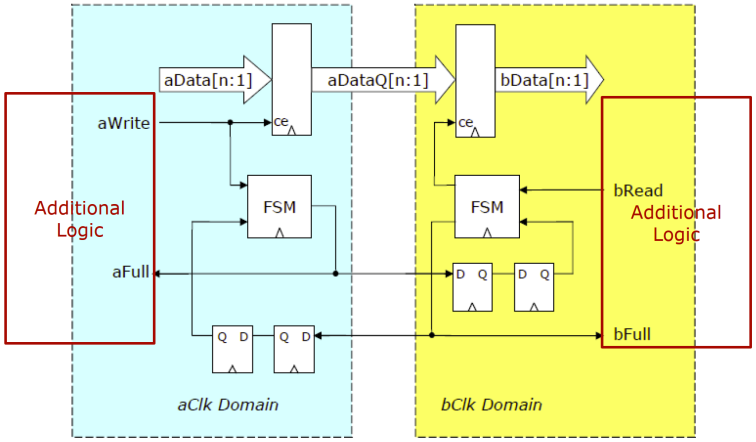


Figure 8: Bus synchronizer used in synchronizer.sv

We modified this design slightly so that the system clock domain isn't only notified when the data has been received by the audio clock domain, but also notifies when the audio clock domain is ready for a signal. This is essential for getting this synchronizer to send information at 48 kHz, and at the rate the I2S module needs it.

In our implementation, the I2S module reads the data on its input and then sends a read signal to the synchronizer so that the next sample will arrive by the time the I2S module needs it.

```
61          if (lr_counter == 191) begin            // every 24 bits sent (every 24 8-tick intervals), toggle the word clock
62              lrclk <= ~lrclk;                    // toggle word clock
63              sample <= pcm;                      // read pending sample into sample register
64              getNewSample <= 1'b1;               // set bit to signal that the I2S module needs a new sample
65              lr_counter <= 0;                    // reset word clock
66          end
67          else begin
68              lrclk <= lrclk;
69              lr_counter <= lr_counter + 1;
70              if (getNewSample && PCMready) begin // if I2S module needs a new sample and a sample is ready,
71                  readPCM <= 1'b1;                //  send signal to synchronizer to read in new PCM value
72                  getNewSample <= 1'b0;
73              end
74              else begin
75                  readPCM <= 1'b0;
76                  getNewSample <= getNewSample;
77              end
78          end
79
80          s_counter <= s_counter + 1;     // increment serial clock
```

Figure 9: i2s.sv word clock and ready signal logic

The synchronizer then registers a signal to the system clock domain that it needs a new sample. If both the ready and the not full signals in the system clock domain are set, it sends an edge-triggered interrupt to signal that it's ready for a new sample.

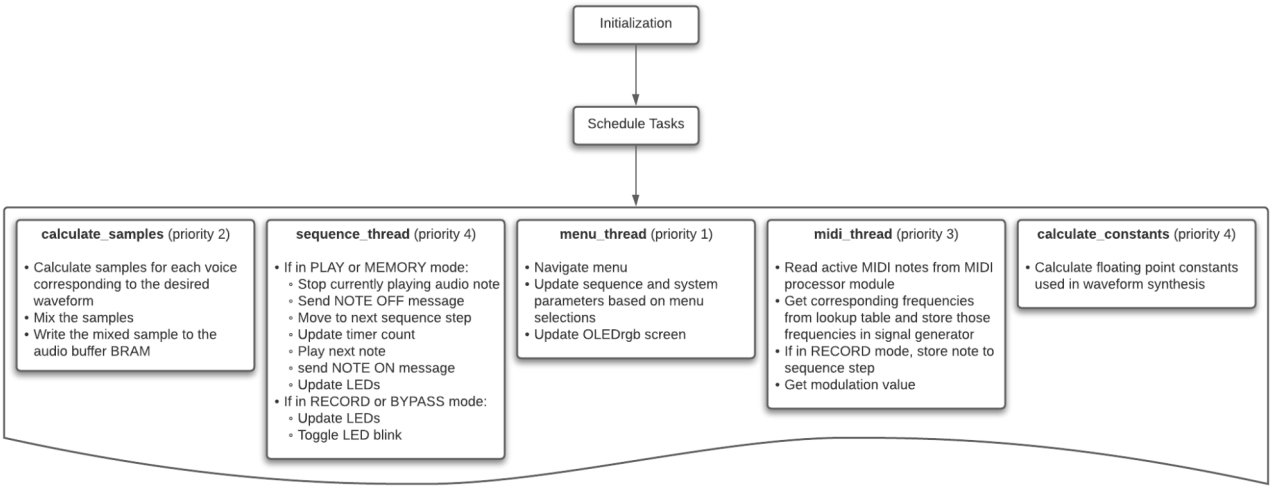
```
50      always_ff @(posedge clk, negedge rstn) begin
51
52          if (~rstn) begin
53              ready = 1'b0;
54          end
55
56          else begin
57              if ((!synchFull) && (synchReady)) begin     // if the synchronizer is not currently full and is ready for a new sample,
58                  ready = 1'b1;                           //  send an interrupt signal to the system to tell it to send a new sample
59              end
60              else begin
61                  ready = 1'b0;
62              end
63          end
64
65      end //ff
```

Figure 10: i2s2_top.sv

The system can then respond by sending a sample over the AXI bus. The sample is placed on the data bus along with a signal that tells the synchronizer it has been written to, which is clocked through the synchronization registers into the audio clock domain. The I2S module can then read the sample.

Theory of Operation: Software

The synthesizer's software runs on FreeRTOS configured for a 16 kB heap and a tick rate of 1000 ticks per second. The program first executes an initialization process and then schedules five tasks: *calculate_samples*, *sequence_thread*, *menu_thread*, *midi_thread*, and *calculate_constants*. These tasks are scheduled by the operating system, and frequently interrupted by six different sources--AXI Timers 0 and 1, the MIDI Input Processor, the I2S Module, the GPIO Inputs, and the Rotary Encoder. Note that AXI Timer 0 is exclusively used by FreeTOS to manage system ticks, so it's not linked in the program to



an interrupt handler.

Figure 12: FreeRTOS program sequence in main.c

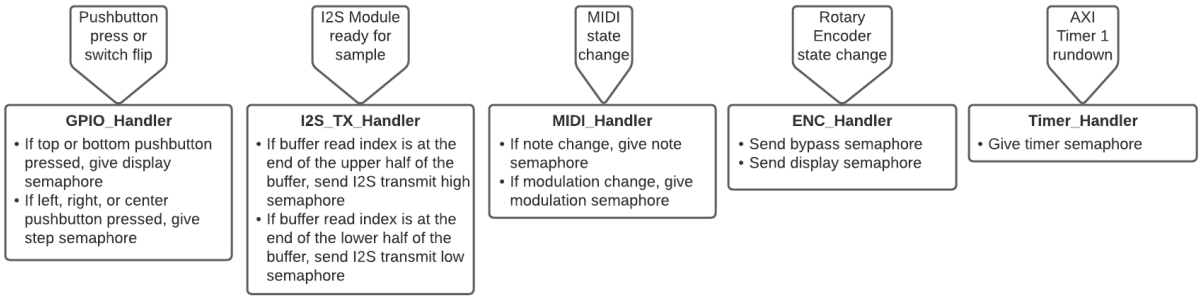


Figure 13: Interrupt service routines in interrupts.c

The interrupt handlers signal the tasks, and the tasks signal each other, using 9 different binary semaphores. 8 of these are sent from interrupt handlers to tasks: *I2S_TX_high_sem*, *I2S_TX_low_sem*, *timer_sem*, *step_sem*, *bypass_sem*, *display_sem*, *note_sem*, and *mod_sem*. The remaining semaphore, *calc_sem*, is used for inter-task communication.

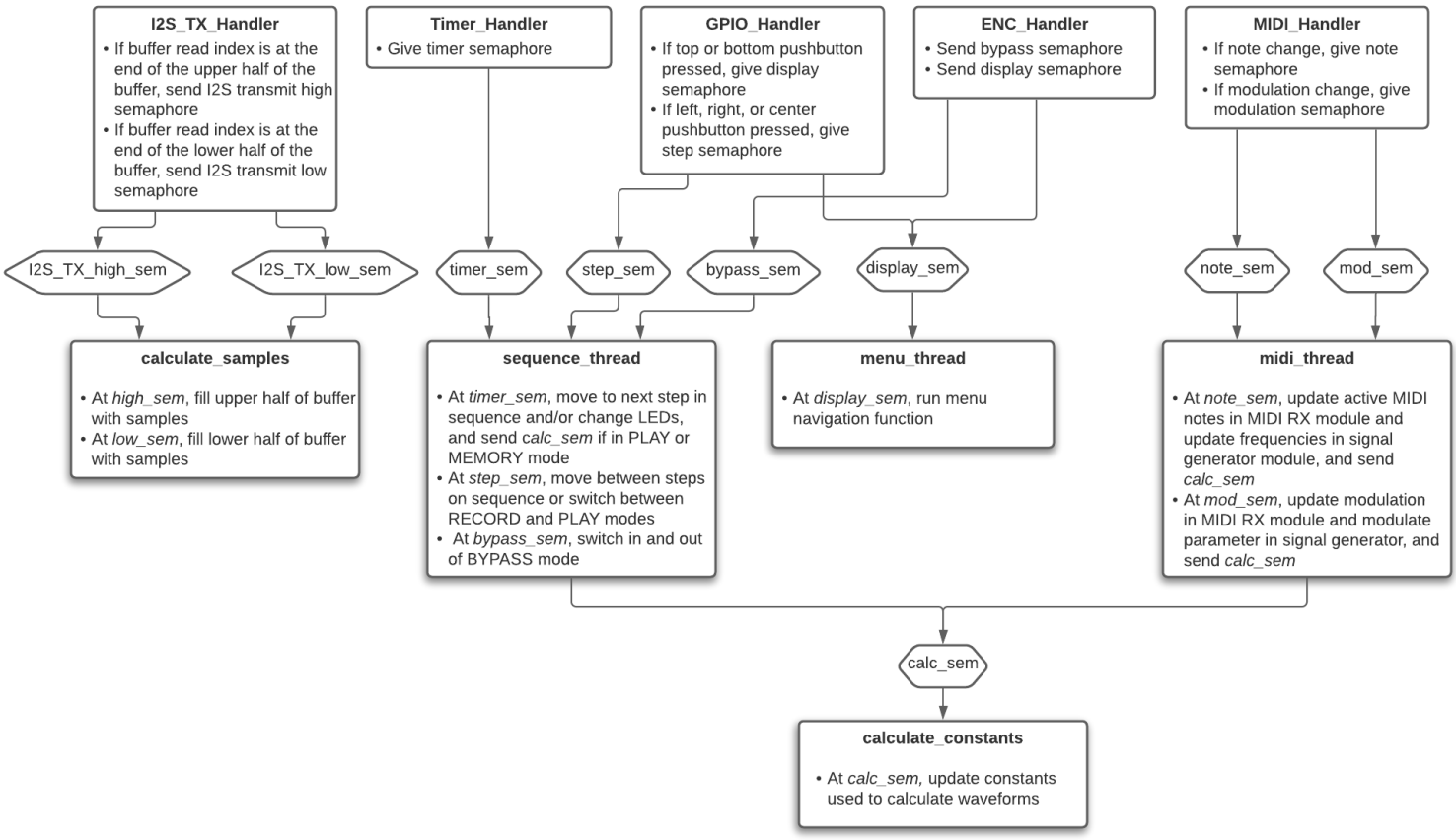


Figure 14: Semaphore communication

Control.c/control.h	Button and switch control functions
display.c/display.h	OLED display functions included in Project 1
initialization.c/initialization.h	Hardware driver initialization functions
interrupts.c/interrupts.h	Interrupt initialization and service routines
luts.c/luts.h	Lookup tables for note number to frequency and note number to
main.c/main.h	Main program function including tasks and semaphores
menu.c/menu.h	OLEDrgb display menu
midi.c/midi.h	MIDI receiver and transmitter
sequencer.c/sequencer.h	Sequencer
signal_generator.c/signal_generator.h	Waveform and sample generation
system.c/system.h	API to make system parameters available to menu and other system-level functionality

Figure 15: Software files

Results

The goal of this project as outlined in our project proposal was to create a synthesizer on the Artix-7 FPGA. We also specified “to demonstrate success, we’ll show that the sequencer can read input signals from an instrument through MIDI, play a corresponding sequence over the I2S audio, and also play a corresponding sequence to an external instrument using MIDI.” These goals have been met, along with a few extra stretch goals as well. Below is an outline of the system's functionality as seen by the user.

When the board is powered on and programmed, the OLED display provides settings for the user to change. The second value on the first screen is mode. There are 4 modes available to choose from.

Menu navigation

The sequence can be paused and resumed using the center push button. By pressing the up and down buttons, the user can select the parameter that can be changed by turning the rotary encoder.

Figure 16: OLED display

System

- Volume - Adjusts the volume from 0-100% linearly.
- Mode:
    - **PLAY** - Play mode plays a sequence of notes out through the 3.5mm headphone jack, as well as two voices played by the user over MIDI input
    - **REC** - Record will save the next 16 inputs from the MIDI input.
    - **BYP** - Bypass will skip any processing and pass-through any MIDI input to the I2S output.
    - **MEM** - Memory will play the sequence that is stored in the BRAM at HW build time, as well as two voices played by the user over MIDI input. The sequence is saved in the .coe file.
- Slot - Selects the slot to be used by the load and store commands. (1-16)
- Load - Selects the currently selected slot sequence to be active.
- Store - Saves the current sequence into the currently selected slot.

Sequence

- Tempo - Changes the playback timing. (60-240)
- Subdiv - Changes the note subdivision, or number of beats per measure (1/1 - 1/64)
- Swing - changes the relative successive note lengths (e.g. a swing of 50% means the first note will be 50% longer, the second 50% shorter. A swing of -50% results in the inverse) (+/- 100)
- Patt - Changes how the sequence is played back; forward, backward, random, single-note sustain

Challenges

A project this size would be very lucky to not encounter any major challenges in its development. We experienced our fair share and almost all were further hindered by the onlineness of this year and the inability to collaborate in person. It is also the opinion of the entire group that Vivado's file management system does not provide a reasonable solution for working as a team online. We attempted sharing project files with each other in a significant number of ways but never came up with a system that would reliably work on all of our setups. This meant that every time a major update occurred, significant time would be spent attempting to integrate the changes on the other two setups.

Towards the end of development we experienced a series of issues with integrating IP together. Once more Vivado IP blocks were put into the hardware, inconsistencies and build issues started appearing. This was most felt after adding the BRAM block that is initialized with a .coe file. We were working towards adding VGA and floating point hardware but both were practically abandoned after the memory issues to avoid messing with an already fragile system in the last two nights before the demo.

Which leads us into the time constraints on this project. We as a group were quite challenged by the amount of time we had to work on such a large endeavor. This project was an especially fun one because it felt like continuous progress was being made every day and with that came new features and things to learn. A major factor for us was the overlap between project 2 and the final project. Because of the way the due dates were set up, we only had a very short amount of time where the group was entirely focused on this project. If more time had been available, we are confident that many of our stretch goals would have been met. The largest additions if we'd had more time would've been VGA output and floating point hardware.

Conclusion

For an embedded system class's final project, this one was perfect. It had a well balanced mix of new hardware and software challenges to explore. We were successful in meeting all of our initial goals and even met a few stretch goals done as well. The final product includes custom IP, Vivado IP, PMOD attachments, external hardware, multiple tasks in FreeRTOS, interrupts, and much more. Getting all of these independent parts working together in one embedded system has been a challenging but rewarding experience. While we hit our fair share of road bumps along the way, the result at the end of our term in ECE 544 is a functional FPGA synthesizer that is fun and easy to work with.

Appendix

Equipment list

Nexys A7: FPGA Trainer Board Recommended for ECE Curriculum

https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/

Pmod ENC: Rotary Encoder

https://store.digilentinc.com/pmod-enc-rotary-encoder/

Pmod OLEDrgb: 96 x 64 RGB OLED Display with 16-bit Color Resolution

https://store.digilentinc.com/pmod-oledrgb-96-x-64-rgb-oled-display-with-16-bit-color-resolution/

Pmod I2S2: Stereo Audio Input and Output

https://store.digilentinc.com/pmod-i2s2-stereo-audio-input-and-output/

Level shifter:

https://www.adafruit.com/product/757?gclid=Cj0KCQjw8IaGBhCHARIsAGIRRYodpru8_STPfILlo6Vgp9Y00RwC8pQK7WqNdr0jmwh6EFTQ9cl0CWQaAjp7EALw_wcB

Moog Sub Phatty Synthesizer:

https://www.moogmusic.com/products/sub-phatty

Nord Stage 2 Keyboard:

https://www.nordkeyboards.com/products/nord-stage-2