ECE 385 Final Project Report

Digital Audio Workstation with Audio Processor

By: Soumil Gupta

TA: Wei Ren

10th May 2023

I2C and I2S

Our first step in the journey to make a DAW was to implement I2S and I2C. I2C, or "Inter-Integrated Circuit" is a protocol meant to establish communication between physical hardware ICs. For our purposes, we need an I2C to integrate with the SGTL5000 chip on the I/O shield, which is responsible for setting the parameters upon which the SGTL5000 will operate.

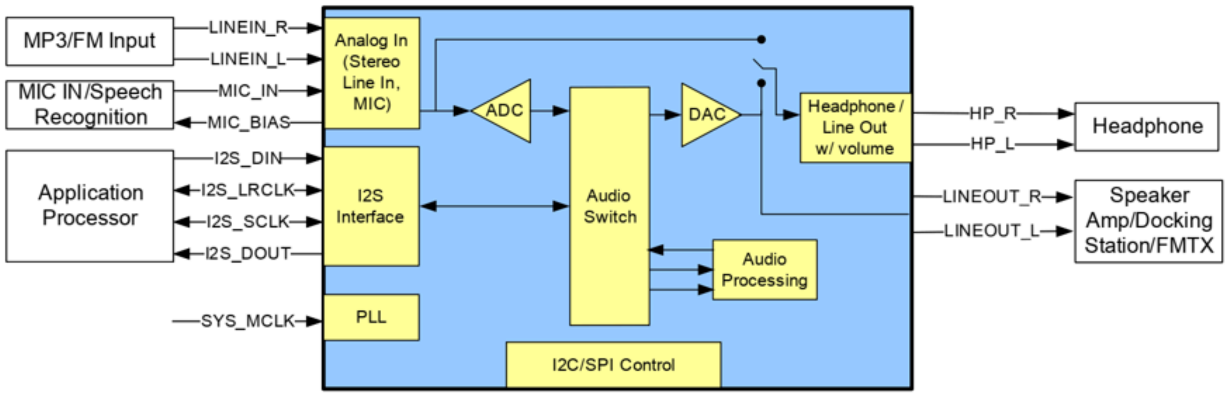
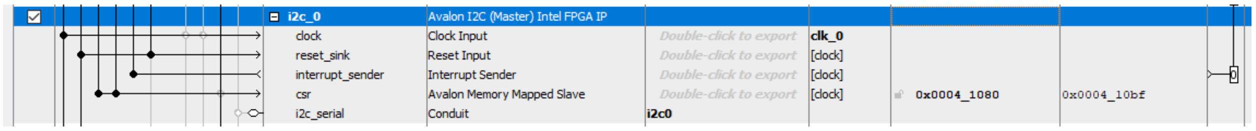


Fig. 1. SGTL5000 Codec

To instantiate I2C, we can simply connect the I2C peripheral device in the platform designer as demonstrated below:

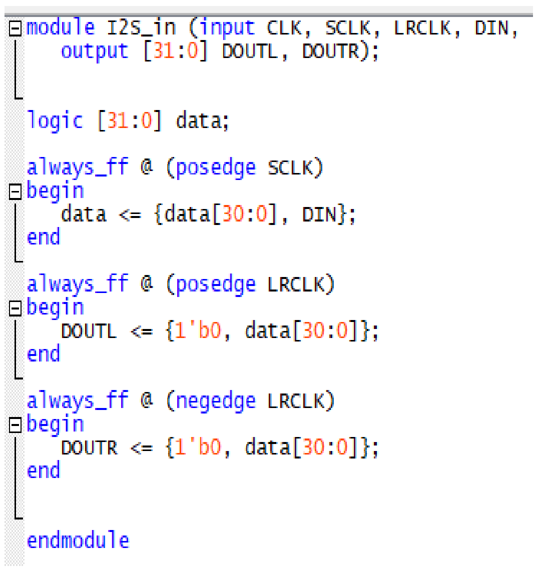


i2c_0	Avalon I2C (Master) Intel FPGA IP				
clock	Clock Input	Double-click to export	clk_0		
reset_sink	Reset Input	Double-click to export	[clock]		
interrupt_sender	Interrupt Sender	Double-click to export	[clock]		
csr	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0004_1080	0x0004_10bf
i2c_serial	Conduit	i2c0			

Fig. 2. Creating I2C component on platform designer and connecting to Avalon bus and interrupt

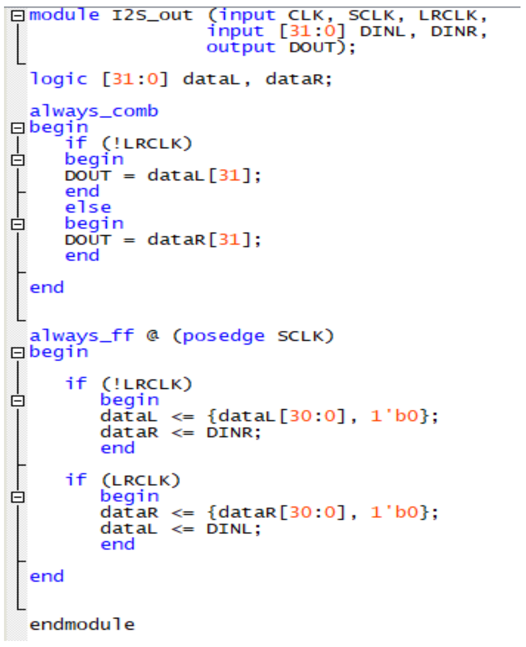
Next, we have I2S, which stands for "Inter-IC Sound," which is a serial bus interface utilized for connecting digital audio devices. For I2S, it's not as easy as generating HDL from Platform Designer for a given IP, but we have to program it in SystemVerilog. The purpose of the man-made HDL code is to produce a sound made to operate with right and left channels (because humans have two ears). To do this, I have an "I2S_input" module that takes in one bit of data at a time from a source (either line-in or SDRAM depending on stage of development), and I place the input bit either in a 32-bit shift register for the left channel, or a 32-bit shift register for the right channel. The decision for which register to add the bit to is based on LRCLK, a 44.1 kHz clock to indicate whether a sample should be sent to the left channel or right channel. By constantly switching on a set frequency for which channel should sampled audio go to, we make a more stereo sound. More info on the clocks will be explained in the next section.

Remember, the HDL we are writing and generating aren't instantiating the codec parts from Fig. 1 on the FPGA, but rather establishing the FPGAs ability to control the already built hardware on the SGTL5000 IC.



```
module I2S_in (input CLK, SCLK, LRCLK, DIN,
    output [31:0] DOUTL, DOUTR);


logic [31:0] data;

always_ff @ (posedge SCLK)
begin
    data <= {data[30:0], DIN};
end

always_ff @ (posedge LRCLK)
begin
    DOUTL <= {1'b0, data[30:0]};
end

always_ff @ (negedge LRCLK)
begin
    DOUTR <= {1'b0, data[30:0]};
end


endmodule
```

Fig. 3. HDL Code for I2S_in. Audio data is packaged in 32-bit words. That data is shifted one-bit in from I2S_DOUT of the I2S interface into the Application Processor (DAW on FPGA). The reason we set the MSB of our left and right channel I2S modules as 0 is because that is a junk bit. For 64 SCLK cycles we sample data to fill a 32-bit register for the left sample data and the right sample data.



```
module I2S_out (input CLK, SCLK, LRCLK,
                input [31:0] DINL, DINR,
                output DOUT);

logic [31:0] dataL, dataR;

always_comb
begin
    if (!LRCLK)
    begin
    DOUT = dataL[31];
    end
    else
    begin
    DOUT = dataR[31];
    end

end

always_ff @ (posedge SCLK)
begin

    if (!LRCLK)
        begin
        dataL <= {dataL[30:0], 1'b0};
        dataR <= DINR;
        end

    if (LRCLK)
        begin
        dataR <= {dataR[30:0], 1'b0};
        dataL <= DINL;
        end

end

endmodule
```

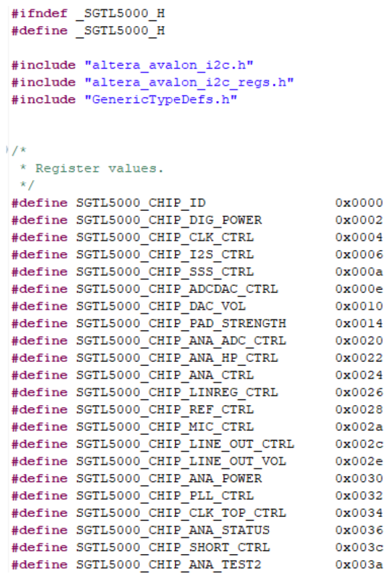
Fig. 4. HDL Code for I2S_out. Data processed and manipulated by the FPGA is put as I2S_DIN into the I2S interface on the SGTL5000 Codec. The Audio Switch routes the bits sent from I2S_out through the I2S_DIN data line to a Digital-to-Analog converter and out the headphone jack. If LRCLK is high, that means data from dataL will be fed from the MSB every time the LRCLK is high as the output bit from I2S_out, and vice-versa. On the faster sample clock, if LRCLK is high, then we are setting dataL to be the next 32-bit word for the left channel, and in that cycle we feed another junk bit of 1'b0 to the dataR that isn't being used at that time, so the registers for the left and right channels are synchronous.

There is another important detail to take note of. I2S_in takes in input from two different possible sources during the execution of the DAW program. At first, it is getting audio data from the line-in AUX jack on the I/O shield (ARDUINO_IO[1] pin which connects to I2S_out on SGTL5000), and splitting that input data into left and right channel data. Then, when we're reading audio data from our music library on SDRAM, then the I2S_in will take music data input from there and split that input data into left and right channels to be mixed together and/or apply Gain and EQ to. I2S_out on the other hand can either send data serial data 1-bit at a time from taking in two 32-bit shift registers to the ARDUINO_IO[2] pin which connects to the I2S_DIN input on the SGTL5000 which would be directed to the headphone jack output, or used as the way to then take multi-channel audio and send those bits to our macrofunction for SDRAM Read/Writes to write edited audio to SDRAM.

*Insert Diagram here of MUX connected to two shift registers connected to second MUX.

Eclipse for SGTL5000 Usage

As mentioned previously, the SGTL5000 has internal parameters to be set that will be set by embedded C programming language code. The SGTL5000 has a lot of registers designed for operating the different actions and parts of it, such as the Audio Switch, DAC and ADC converters, and Interface modules. It also has an Audio Processing component that I was not able to fully explore due to time constraints and because of its sheer complexity of making it synchronously operate with HDL code.



```
#ifndef _SGTL5000_H
#define _SGTL5000_H

#include "altera_avalon_i2c.h"
#include "altera_avalon_i2c_regs.h"
#include "GenericTypeDefs.h"


/*
 * Register values.
 */
#define SGTL5000_CHIP_ID                0x0000
#define SGTL5000_CHIP_DIG_POWER         0x0002
#define SGTL5000_CHIP_CLK_CTRL          0x0004
#define SGTL5000_CHIP_I2S_CTRL          0x0006
#define SGTL5000_CHIP_SSS_CTRL          0x000a
#define SGTL5000_CHIP_ADCDAC_CTRL       0x000e
#define SGTL5000_CHIP_DAC_VOL           0x0010
#define SGTL5000_CHIP_PAD_STRENGTH      0x0014
#define SGTL5000_CHIP_ANA_ADC_CTRL      0x0020
#define SGTL5000_CHIP_ANA_HP_CTRL       0x0022
#define SGTL5000_CHIP_ANA_CTRL          0x0024
#define SGTL5000_CHIP_LINREG_CTRL       0x0026
#define SGTL5000_CHIP_REF_CTRL          0x0028
#define SGTL5000_CHIP_MIC_CTRL          0x002a
#define SGTL5000_CHIP_LINE_OUT_CTRL     0x002c
#define SGTL5000_CHIP_LINE_OUT_VOL      0x002e
#define SGTL5000_CHIP_ANA_POWER         0x0030
#define SGTL5000_CHIP_PLL_CTRL          0x0032
#define SGTL5000_CHIP_CLK_TOP_CTRL      0x0034
#define SGTL5000_CHIP_ANA_STATUS        0x0036
#define SGTL5000_CHIP_SHORT_CTRL        0x003c
#define SGTL5000_CHIP_ANA_TEST2         0x003a
```

Fig 5. Code snippet of provided C code of SGTL5000 registers being configured.

Because I wanted to be able to save SDRAM addresses for reading and writing from SDRAM, and because of the time lost in running Eclipse on top of programing the FPGA for every test, I configured Eclipse to send the C code to the FPGA as part of the SOF file, which I had also configured to use OCM instead of SDRAM. More on this later.

Clocking Considerations and FIFO Buffers

Before we go any further, let's go through the different clocks that we have to use in this project and how each one is managed.

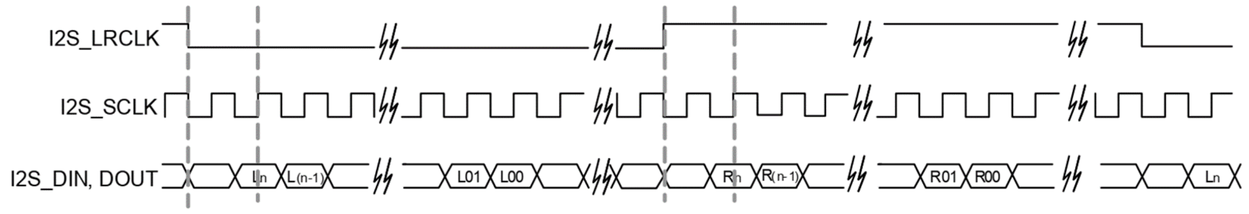
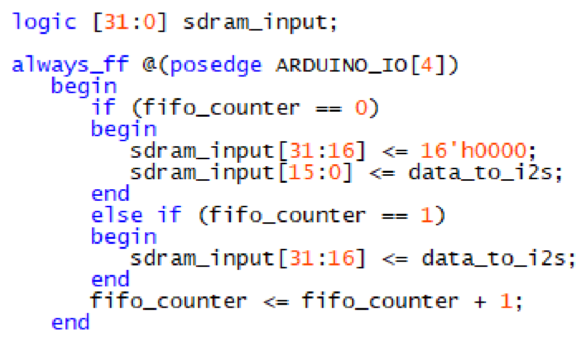


Fig 6. Clock diagram of Audio-Related Clocks

1. LRCLK: LRCLK is the clock to define for which audio channel (left or right ear) are we sampling for. If LRCLK is high, then we are sampling for the left channel, and if it is low, we are sampling for the right channel. LRCLK operates at a frequency of 64 times slower than the sample clock/frequency, known as SCLK. Funnily enough, the LRCLK operates at 44.1 kHz, which is the sampling frequency for the digital audio we are working with, which makes sense because that is the frequency at which we want to receive and send our audio data at.
2. SCLK: SCLK operates at the sampling frequency of 64 times LRCLK, and therefore in a full clock cycle of LRCLK as shown by Fig. 6, 32 bits for the left channel and 32 bits for the right channel are put through our I2S shift registers.
3. MCLK: While MCLK is not as important for our work, MCLK is utilized by the SGTL5000 for its internal operations.
4. MAX10_CLK1_50: 50 MHz clock that FPGA and NIOS operate on.
5. Outputclk: 50 MHz clock divided by 16 for FIFO buffer purposes mentioned below.

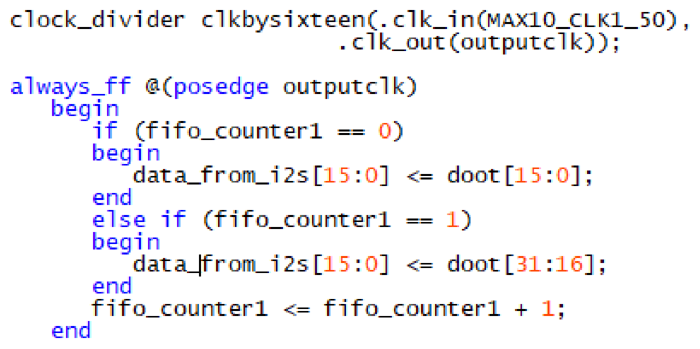
*All of these clocks are generated by the SGTL5000 IC internally, and not generated by any other item on the FPGA board or I/O shield. Our FPGA receives these clocks to operate its internal channel registers and other HDL I wrote, through the "ARDUINO_IO" pins on the FPGA board. ARDUINO_IO[5] is SCLK, ARDUINO_IO[4] is LRCLK, and ARDUINO_IO[3] is MCLK (*Also as insinuated previously, ARDUINO_IO[1] is I2S_DOUT and ARDUINO_IO[2] is I2S_DIN).

This brings us to our FIFO buffers. We had to have 2 FIFO buffers in our code, both pertaining to control the transfer of audio data from SDRAM to I2S and I2S to SDRAM.



```
logic [31:0] sdram_input;

always_ff @(posedge ARDUINO_IO[4])
   begin
      if (fifo_counter == 0)
      begin
         sdram_input[31:16] <= 16'h0000;
         sdram_input[15:0] <= data_to_i2s;
      end
      else if (fifo_counter == 1)
      begin
         sdram_input[31:16] <= data_to_i2s;
      end
      fifo_counter <= fifo_counter + 1;
   end
```

Fig 7. FIFO buffer to send data to I2S. SDRAM only has 16 bit words per address, and so two addresses have to be read, and concatenated together before being sent as a 32-bit word to my I2S_in module. This buffer operates on LRCLK because that is the clock by which the receiver of the data, I2S, operates.



```
clock_divider clkbysixteen(.clk_in(MAX10_CLK1_50),
                           .clk_out(outputclk));

always_ff @(posedge outputclk)
   begin
      if (fifo_counter1 == 0)
      begin
         data_from_i2s[15:0] <= doot[15:0];
      end
      else if (fifo_counter1 == 1)
      begin
         data_from_i2s[15:0] <= doot[31:16];
      end
      fifo_counter1 <= fifo_counter1 + 1;
   end
```

Fig 8. FIFO buffer to store 32-bit words from I2S into 16-bit words to be stored in SDRAM. I had to make a clock-divider module to do this in the right time frame. This buffer operates on our 50 MHz / 16 clock because we can write a lot faster than we can read, both because of the SDRAM bridge design (More on that later) and our audio implementation.

This brings us to the generic clock divider, which divides our 50 MHz clock by 16, to tone down the operations to stay in line with the audio clocks. How it works is it counts up to 16 on an always_ff based on the input master clock, and when we count up to 16, the output block wire is set to high. We possibly could have set this clock using an .sdc file, but this approach felt quicker at the time of implementation.

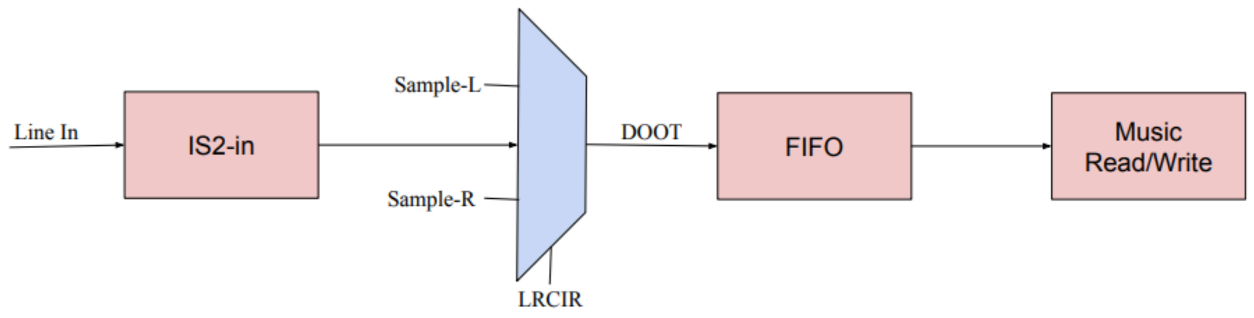


Fig. 9. The following is a snippet of the full audio data pipeline. This multi-use part of the pipeline can take data in from SDRAM or line-in, and put it through I2S_in and the FIFO to be written to SDRAM. Again, the line-in would be from the aux input from the I/O shield if we were writing all tracks (state 1 on FSM in Fig. 23), and otherwise could be audio data that has been mixed by the audio module.

Audio Effects

Gain:

I experimented with a lot of different ways to manipulate the left and right channel bits in their respective shift registers to increase the volume. I studied the theory of how sampling works and how manipulating the bits affects the gain of the sound.

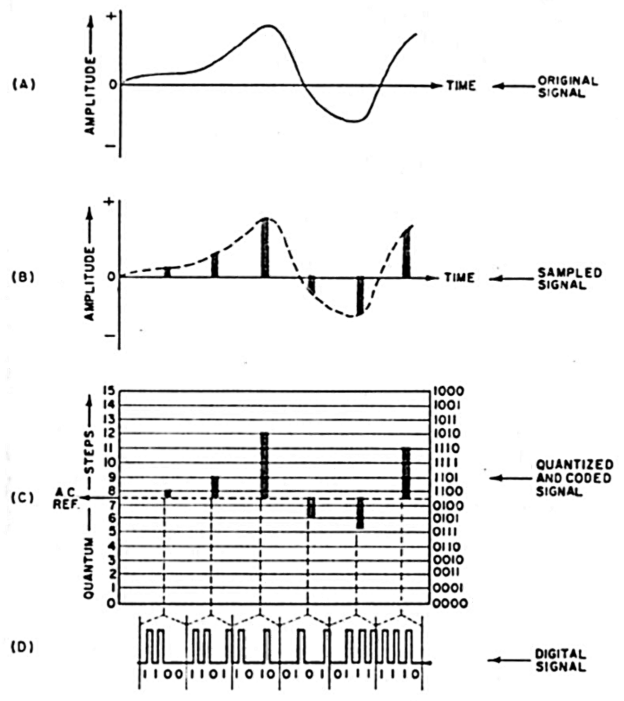


Fig. 2. Basic operating steps in pulse-code modulated transmission. (A) The original speech signal is (B) sampled to produce pulses corresponding to instantaneous voltage amplitude of audio waveform which is then (C) quantized into incremental levels and (D) encoded into equivalent digital signal.

Fig 10. Diagram provided from lecture. Digital Signals are discrete signals that together can recreate an original signal. The original signal is sampled at a sampling

frequency and we can recreate the signal from our individual, discrete samples with DSP.

I ended up going with the most simple implementation, which was bit shifting. By right or left shifting the bits on each discrete signal of our digital "impulse train," we increase or decrease the intensity by a power of 2. I routed the volume control to the controlled by switches, and by changing three switches, you can change the volume through the gain function that we feed our right and left channel both in between my I2S_in and I2S_out modules.

GAIN[2]	GAIN[1]	GAIN[0]	Effect:
0	0	0	Nothing
0	0	1	Lower volume by factor of 2
0	1	0	Lower volume by factor of 4
0	1	1	Lower volume by factor of 6
1	0	0	Nothing
1	0	1	Increase volume by factor of 2
1	1	0	Increase volume by factor of 4
1	1	1	Increase volume by factor of 6

Fig. 11. Truth table of Gain input to audio effect. Changing the gain switches causes bit shifts of magnitude GAIN[1:0] which have the following effects. Granted, raw bit shifting causes the discrete train to change shape tremendously, so to counteract that we are actually working with 24-bit samples. I apologize if this may come off as misleading earlier, but our 32 bit words actually have 24-bit samples in them, and the rest is "padded" with zeros to protect the data from becoming too noisy with gain and mixing, and to encode the data in an appropriate manner for the SGTL5000. We still want 32 bit samples just because of the clocks available to us and because of the fact that SDRAM stores 16-bit words so to avoid extreme headaches, we want data sampled into packets in a size of a factor of 16. GAIN[2] decides whether to right-shift (lower volume) or left-shift (raise-volume). There is a separate enable control signal to enable the Gain during certain states in my FSM.

Equalizer:
The equalizer is also a very valuable addition to our variety of effects we can apply to our audio from our audio library (data from line-in downloaded to SDRAM). Taking inspiration from ECE 210, my equalizer actually involves a function with bounds on only lower frequencies to our input data, which makes the sound seem like it's playing underwater which is pretty cool.

The convolution happens in an always_comb block adding certain amounts of the bits of the previous sample of the left channel and right channel which is saved in a separate variable, with the next left and right channel samples. My EQ[1:0] control signals dictate the cutoff for which frequencies to allow.

EQ[1]	EQ[0]	Effect:
0	0	Nothing
0	1	Apply filter
1	0	Apply more limited filter
1	1	Apply most limited filter

Fig. 12. Truth table of EQ input to audio effect. There is a separate enable control signal to enable the EQ during certain states in my FSM.

Audio Mixing:
Audio mixing is where we can simply add two of our 32 bit samples from two different tracks in our audio library together. By simple bitwise operations of two 32-bit sample words, we can produce a mixed sound. There are some more complications that can be discussed later. The biggest complication is getting two tracks from SDRAM. See, the tracks are stored in different locations on SDRAM and the avalon bridge we are utilizing can only perform one read or one write at a time. It is not multi-ported. Therefore, on a clock I had to make it so it would switch from reading one place to reading another place and then combining the two read data together. That meant the macrofunction musicReadWrite would constantly be alternating in functionality and its control signals for the duration for which the audio mixing module is required to operate, also set by control signals.

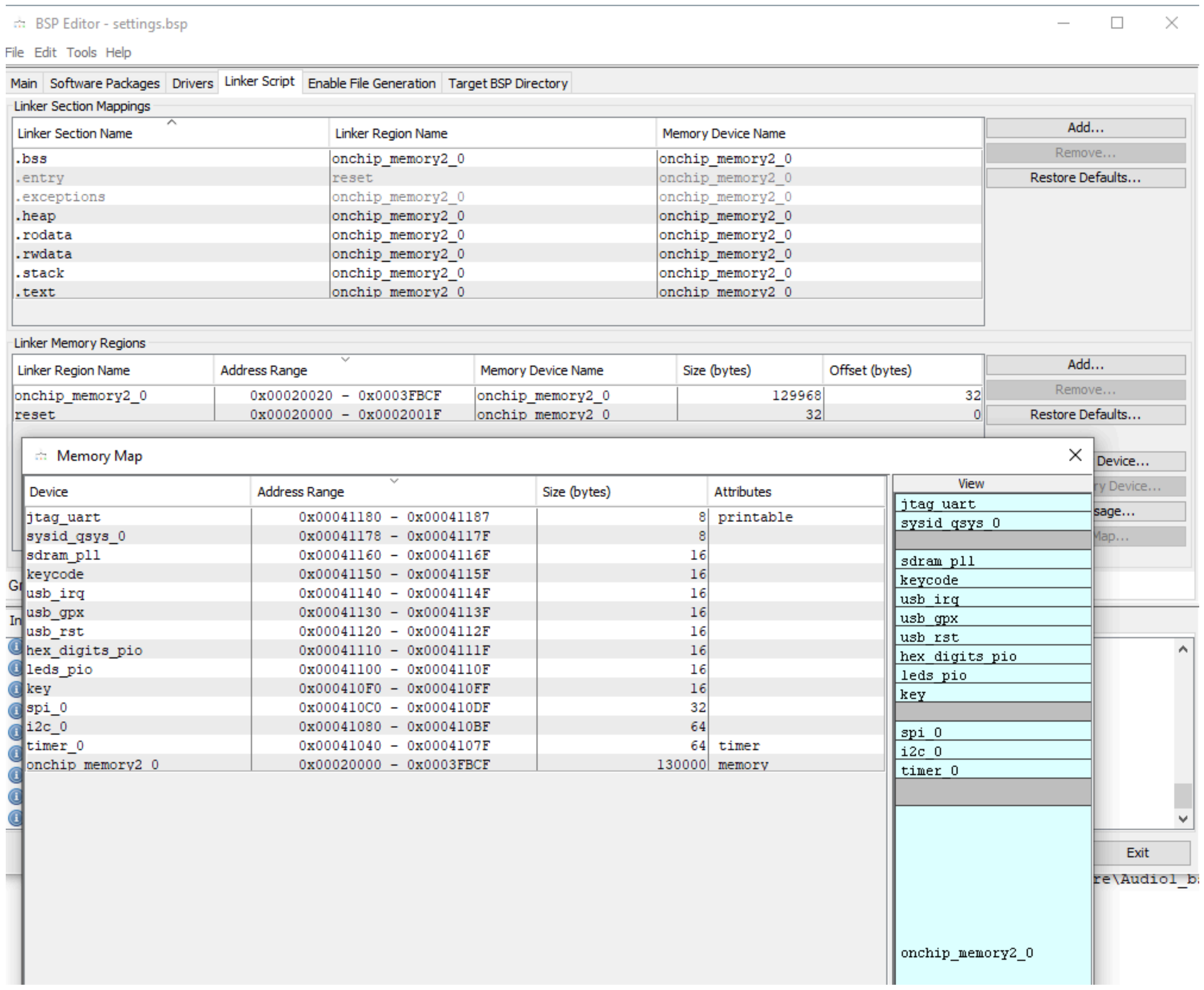
Switching NIOS II to OCM

Upon establishing basic audio functionalities (play and lightly manipulate some sound via the FPGA), I went ahead and started working on the audio library portion of the DAW, which would be necessary to work with Audio mixing at all. To take on the daunting task of storing data, I first looked into storing data on OCM, but found out it is too small to hold multiple 4-8 second long audio recordings. So, my alternative was either SDcard or SDRAM. I attempted to work with SDcard which would have my audio data formatted on it and then be downloaded to SDRAM as a local cache to then be utilized by the DAW on the FPGA, but then I moved to working solely with SDRAM. (More details on SDcard in later section).

To work with SDRAM, I wanted to have no issue with the NIOS II utilizing the same memory space I wanted to read and write audio data to. To ensure this was not a problem, I changed the Platform Designer and Eclipse BSP Linker Script to utilize only OCM.

The first step to do this is to change the NIOS II CPU settings on the platform designer to have its important signals be stored on OCM. Then, on OCM I increase the storage space to 130,000 bytes, which is somewhat overkill, but playing around with the file size led me to find out the NIOS II requires at least 90,000 bytes to operate. OCM is unfortunately limited in size on the FPGA, as it depends on the number of M9K memory modules you have on your specific FPGA. We are equipped with only 182 M9K blocks, each with only ~9,000 bits of space.

The third step is to update the Board Support Package (BSP) linker script and exception and interrupt stacks to all direct to On-Chip Memory.



Linker Section Mappings

Linker Section Name	Linker Region Name	Memory Device Name
.bss	onchip_memory2_0	onchip_memory2_0
.entry	reset	onchip_memory2_0
.exceptions	onchip_memory2_0	onchip_memory2_0
.heap	onchip_memory2_0	onchip_memory2_0
.rodata	onchip_memory2_0	onchip_memory2_0
.rwdata	onchip_memory2_0	onchip_memory2_0
.stack	onchip_memory2_0	onchip_memory2_0
.text	onchip_memory2_0	onchip_memory2_0

Linker Memory Regions

Linker Region Name	Address Range	Memory Device Name	Size (bytes)	Offset (bytes)
onchip_memory2_0	0x00020020 - 0x0003FBCF	onchip_memory2_0	129968	32
reset	0x00020000 - 0x0002001F	onchip_memory2_0	32	0

Memory Map

Device	Address Range	Size (bytes)	Attributes
jtag_uart	0x00041180 - 0x00041187	8	printable
sysid_qsys_0	0x00041178 - 0x0004117F	8	
sdram_pll	0x00041160 - 0x0004116F	16	
keycode	0x00041150 - 0x0004115F	16	
usb_irq	0x00041140 - 0x0004114F	16	
usb_gpx	0x00041130 - 0x0004113F	16	
usb_rst	0x00041120 - 0x0004112F	16	
hex_digits_pio	0x00041110 - 0x0004111F	16	
leds_pio	0x00041100 - 0x0004110F	16	
key	0x000410F0 - 0x000410FF	16	
spi_0	0x000410C0 - 0x000410DF	32	
i2c_0	0x00041080 - 0x000410BF	64	
timer_0	0x00041040 - 0x0004107F	64	timer
onchip_memory2_0	0x00020000 - 0x0003FBCF	130000	memory

Fig. 13. Memory configuration of all Platform Designer components with memory.

Combine .elf Executable with SOF File

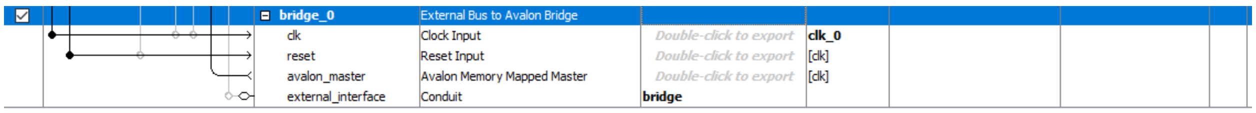
Another action I took to optimize memory and also my development process was to follow the “Combining a Nios II ELF executable into a Hardware Project SOF file” tutorial by the Intel FPGA group. This process allowed me to cut off Eclipse from the development stack entirely, as I could put the .elf image required for NIOS II directly onto the FPGA everytime I program the FPGA with updated HDL code. This also eased the number of states and steps I had to code up, because I didn’t need to wait for states that I didn’t have time to implement anyway to take into account the extra step for Intel FPGA users of configuring Eclipse every time we want to execute embedded C code scripts.

SDRAM Usage and SignalTap Debugging

Now comes time for the infamous portion of this project, SDRAM. To remove NIOS II's access to SDRAM memory so I could have all 64 MB of space for a DAW's built-in Audio Library, I had to configure such in Platform Designer. Please refer to the platform designer for the modified .qsys file which completely disconnected SDRAM from NIOS II, and just kept it in tune with a PLL module and Bridge module, each receiving the 50 MHz master clock.

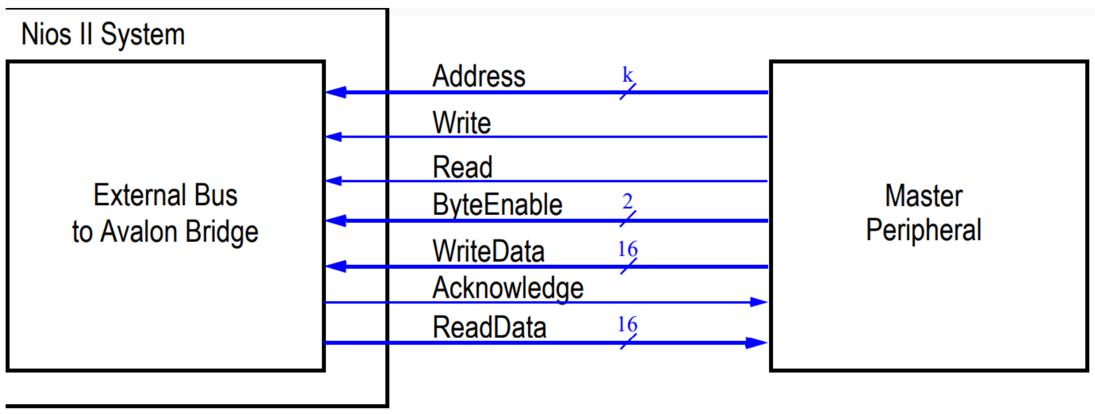
Even though I had decided early on after consulting multiple CAs and Prof. Cheng, I had to still take into consideration a few of the obstacles or nuances of using SDRAM without the Avalon bus (disconnected from Avalon bus so we have full control of SDRAM and NIOS II does not have any control over it). The main concern was latency. SDRAM reads could take anywhere from 2, 8, or maybe even 20 clock cycles. It is unpredictable due to the design of the memory itself. As explored in previous labs, a capacitor discharges and has to refresh, and if you put a read/write during the wrong time like when SDRAM is refreshing, then there could be long wait times. However, that concern was nulled when further research indicated that since SDRAM operates with the 50 MHz master clock, which is order of magnitudes faster than our audio sampling rate, latency of a couple clock cycles on a read or write on a 50 MHz clock would not effect the audio getting the data it needs.

Next, I had to identify a method to actually read/write from SDRAM. So, I found an IP called the "Altera University Program External Bus to Avalon Bridge." I instantiated the IP using Platform Designer, and now just had to understand how the bus control signals and timing diagram operated, then write a master peripheral in HDL that could utilize this bus, but for audio data, which was especially challenging.

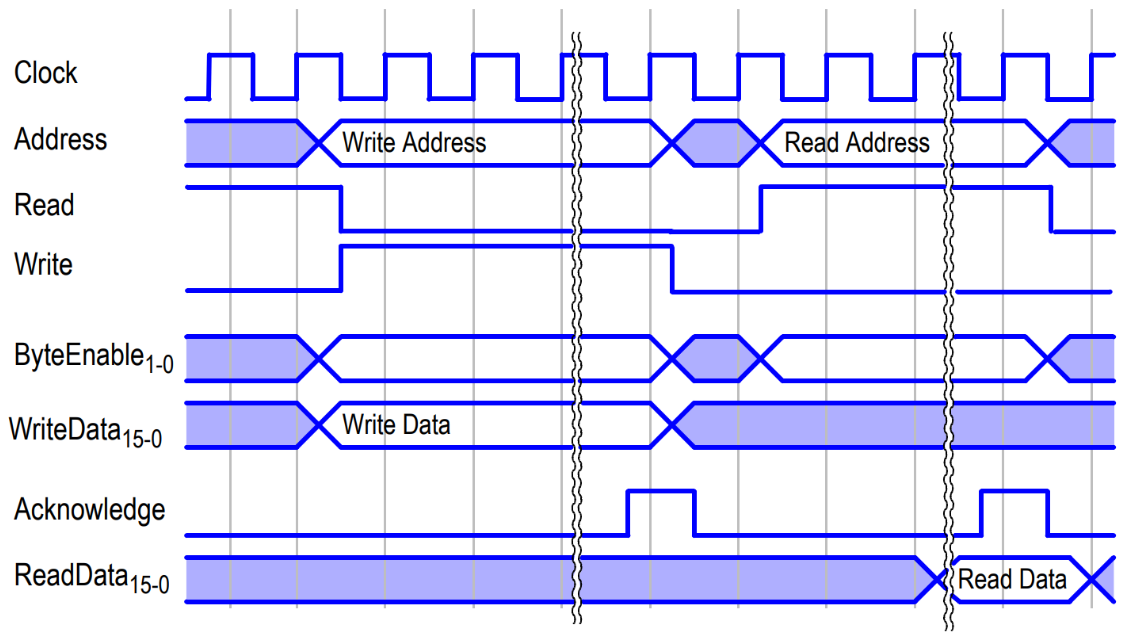


⊟ **bridge_0**	External Bus to Avalon Bridge		
clk	Clock Input	*Double-click to export*	**clk_0**
reset	Reset Input	*Double-click to export*	[clk]
avalon_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]
external_interface	Conduit	**bridge**	

Fig. 14. External Bus to Avalon Bridge Instantiation on Platform Designer. This IP is also not actually connected to the NIOS II in any way, it is just using the Avalon Bus to receive master clock and reset signals, and receive and send data to the SDRAM module in platform designer.



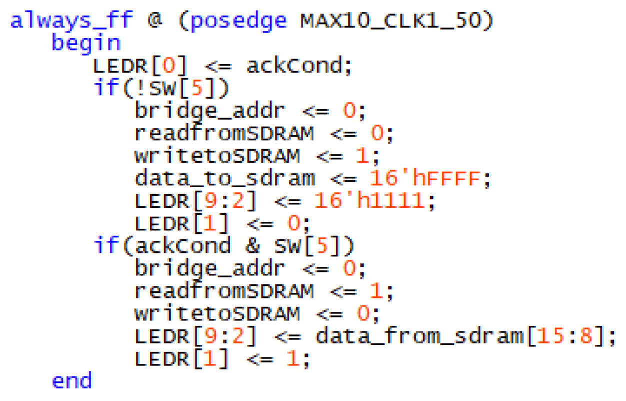
(a) External bus signals



(b) External bus timing diagram

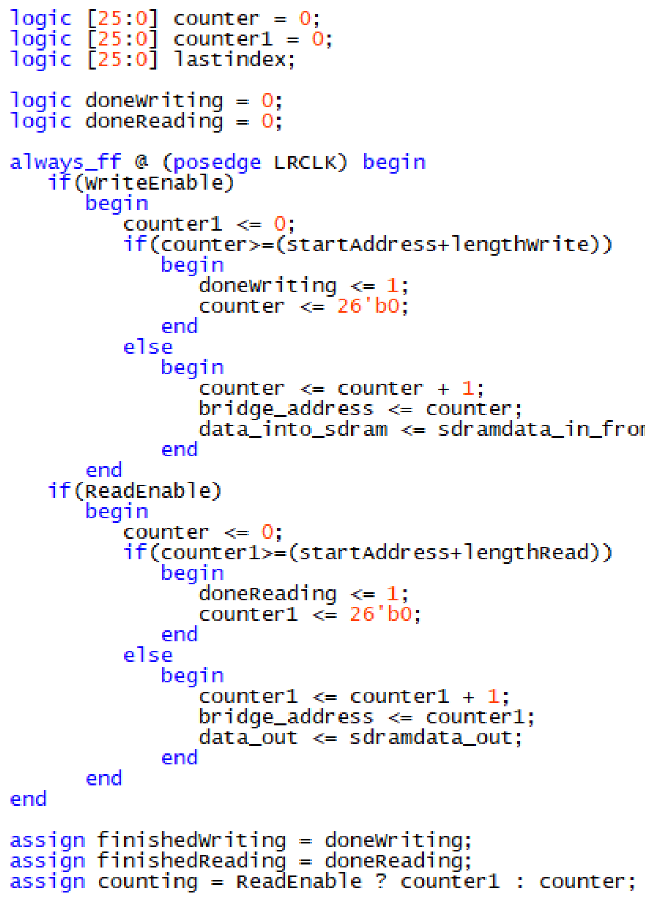
Fig. 15 & 16. Part A is a diagram of the relevant portion of the bridge functionality and composition for our intents and purposes, and part 2 is a timing diagram for the Bridge data. Notice how the Read Data line has a significant latency in how soon data is available from a change in the address and the Read Enable signal going high. That is another benefit of the FIFO buffers, and again, that latency is not of much concern anyway.

Once the bridge was understood, and the Platform Generator generated the HDL for the IP as one of our soc files, I could move on and implement some test code to understand the bridge better and most importantly, double check whether my Platform Designer instantiation of SDRAM, PLL, and the Bridge was correct.



```
always_ff @ (posedge MAX10_CLK1_50)
   begin
      LEDR[0] <= ackCond;
      if(!SW[5])
         bridge_addr <= 0;
         readfromSDRAM <= 0;
         writetoSDRAM <= 1;
         data_to_sdram <= 16'hFFFF;
         LEDR[9:2] <= 16'h1111;
         LEDR[1] <= 0;
      if(ackCond & SW[5])
         bridge_addr <= 0;
         readfromSDRAM <= 1;
         writetoSDRAM <= 0;
         LEDR[9:2] <= data_from_sdram[15:8];
         LEDR[1] <= 1;
   end
```

Fig. 15. Test code to see if I can write something to the very first address, and read it back. This code took some fine-tuning, but luckily worked eventually, indicating that the SDRAM-related components were instantiated correctly via Platform Designer.



```
logic [25:0] counter = 0;
logic [25:0] counter1 = 0;
logic [25:0] lastindex;

logic doneWriting = 0;
logic doneReading = 0;

always_ff @ (posedge LRCLK) begin
   if(WriteEnable)
      begin
         counter1 <= 0;
         if(counter>=(startAddress+lengthWrite))
            begin
               doneWriting <= 1;
               counter <= 26'b0;
            end
         else
            begin
               counter <= counter + 1;
               bridge_address <= counter;
               data_into_sdram <= sdramdata_in_from_file;
            end
      end
   if(ReadEnable)
      begin
         counter <= 0;
         if(counter1>=(startAddress+lengthRead))
            begin
               doneReading <= 1;
               counter1 <= 26'b0;
            end
         else
            begin
               counter1 <= counter1 + 1;
               bridge_address <= counter1;
               data_out <= sdramdata_out;
            end
      end
end

assign finishedWriting = doneWriting;
assign finishedReading = doneReading;
assign counting = ReadEnable ? counter1 : counter;
```

Fig. 17. Excerpt from Macrofunction to read or write data from SDRAM. This Macrofunction was created to be able to write variable sizes of data and read variable sizes of data. It operates on our clock from our clock divider, so it operates at a master clock divided by 16.

Debugging this code and resolving all issues, including realizing and implementing my FIFO buffers took multiple days. Most of my debugging was done on SignalTap, such as the following figure:

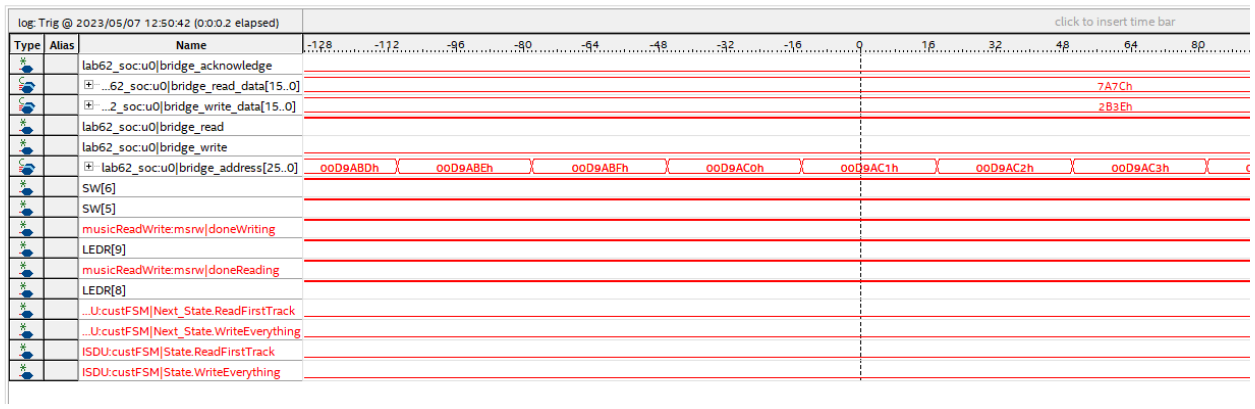
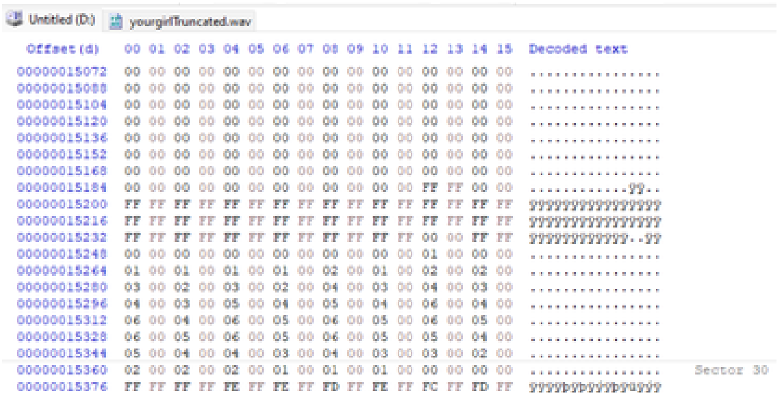

log: Trig @ 2023/05/07 12:50:42 (0:0:0.2 elapsed)
click to insert time bar
Type
Alias
Name
-128 -112 -96 -80 -64 -48 -32 -16 0 16 32 48 64 80
lab62_soc:u0|bridge_acknowledge
...62_soc:u0|bridge_read_data[15..0]
7A7Ch
...2_soc:u0|bridge_write_data[15..0]
2B3Eh
lab62_soc:u0|bridge_read
lab62_soc:u0|bridge_write
lab62_soc:u0|bridge_address[25..0]
00D9ABDh
00D9ABEh
00D9ABFh
00D9AC0h
00D9AC1h
00D9AC2h
00D9AC3h
SW[6]
SW[5]
musicReadWrite:msrw|doneWriting
LEDR[9]
musicReadWrite:msrw|doneReading
LEDR[8]
...U:custFSM|Next_State.ReadFirstTrack
...U:custFSM|Next_State.WriteEverything
ISDU:custFSM|State.ReadFirstTrack
ISDU:custFSM|State.WriteEverything

Fig. 18. Diagram of attempt to poll reading audio data from SDRAM. Due to a bug in my top level, at the time of this image the data read from SDRAM wire wasn't sending new data as the address of the bridge was incrementing. This was due to data being written wrong in the first place but was resolved. More signaltap screenshots are lost and unfortunately unobtainable given FPGAs were turned in so more can't be taken. Having the counter in Fig. 16 also be the address was a good design choice that led to much convenience in implementing the HDL code.

Audio Data Formatting

Initially I believed that .wav files because of the extra data on each file that dictate how to read them, would cause issues because of decoding the .wav file, I soon learned that through audacity I can export as 16-bit pcm (Pulse-Coded Modulation) which would work with the rest of my HDL code for audio.

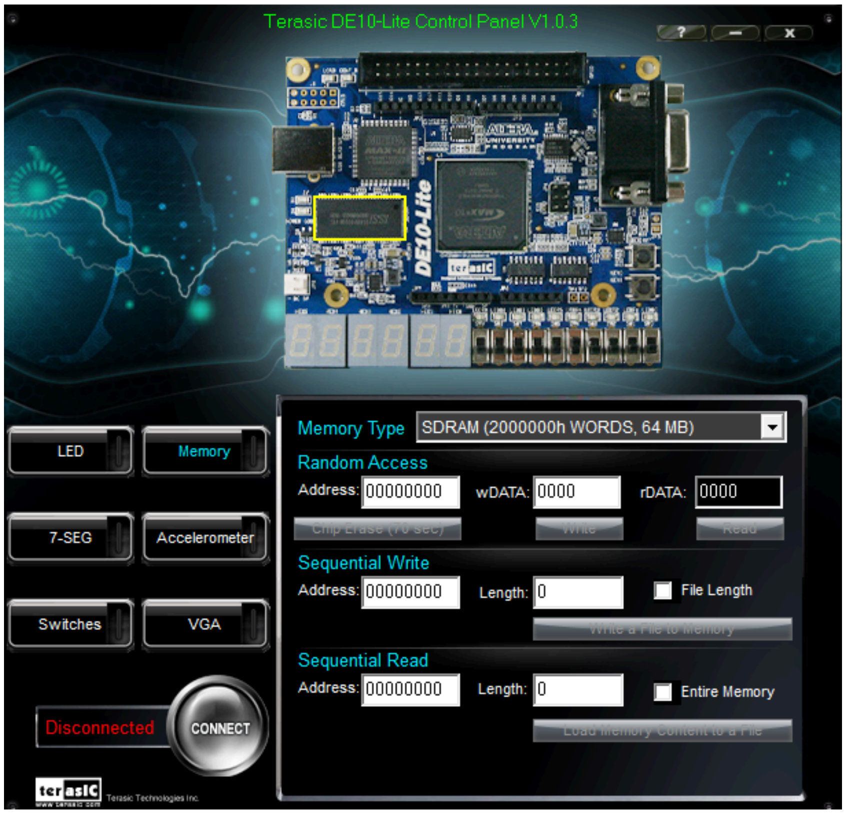


```
Offset(d)    00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15   Decoded text
00000015072  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000015088  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000015104  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000015120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000015136  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000015152  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000015168  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000015184  00 00 00 00 00 00 00 00 00 00 00 00 FF FF 00 00   ............ÿÿ..
00000015200  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000015216  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000015232  FF FF FF FF FF FF FF FF FF FF FF FF 00 00 FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿ..ÿÿ
00000015248  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00   ................
00000015264  01 00 01 00 01 00 01 00 02 00 01 00 02 00 02 00   ................
00000015280  03 00 02 00 03 00 02 00 04 00 03 00 04 00 03 00   ................
00000015296  04 00 03 00 05 00 04 00 05 00 04 00 06 00 04 00   ................
00000015312  06 00 04 00 06 00 05 00 06 00 05 00 06 00 05 00   ................
00000015328  06 00 05 00 06 00 05 00 06 00 05 00 05 00 04 00   ................
00000015344  05 00 04 00 04 00 03 00 04 00 03 00 03 00 02 00   ................
00000015360  02 00 02 00 02 00 01 00 01 00 01 00 00 00 00 00   ................   Sector 30
00000015376  FF FF FF FF FE FF FE FF FD FF FE FF FC FF FD FF   ÿÿÿÿþÿþÿýÿþÿüÿýÿ
```

Fig. 19. I can further manipulate my .wav files in a program called HxD to check the actual bits at specific addresses that I will be feeding into SDRAM.

Other Implementation Methodologies

There were other things I did in an attempt to complete parts of the project or debug parts of the project. One such thing was SDcard reading. With the given HDL for SDcard, I attempted to interface with the SDcard and feed bits to SDRAM. But, there were issues with the endianness and addressing even after reformatting the entirety of the SDcard on HxD, so I moved to the Avalon Bridge IP. I also wanted to use the Terasic GUI to see if I could interface with SDRAM directly with that, and unfortunately the Terasic GUI downloads a new .sof file to the FPGA every time you connect the GUI application to the FPGA. On top of that, the data written to SDRAM through the software doesn’t stay written when you download new .sof files ot the FPGA (we found that out through a lot of testing), and the GUI would crash when trying to read/write larger than 100 bytes of data with the FPGA SDRAM.

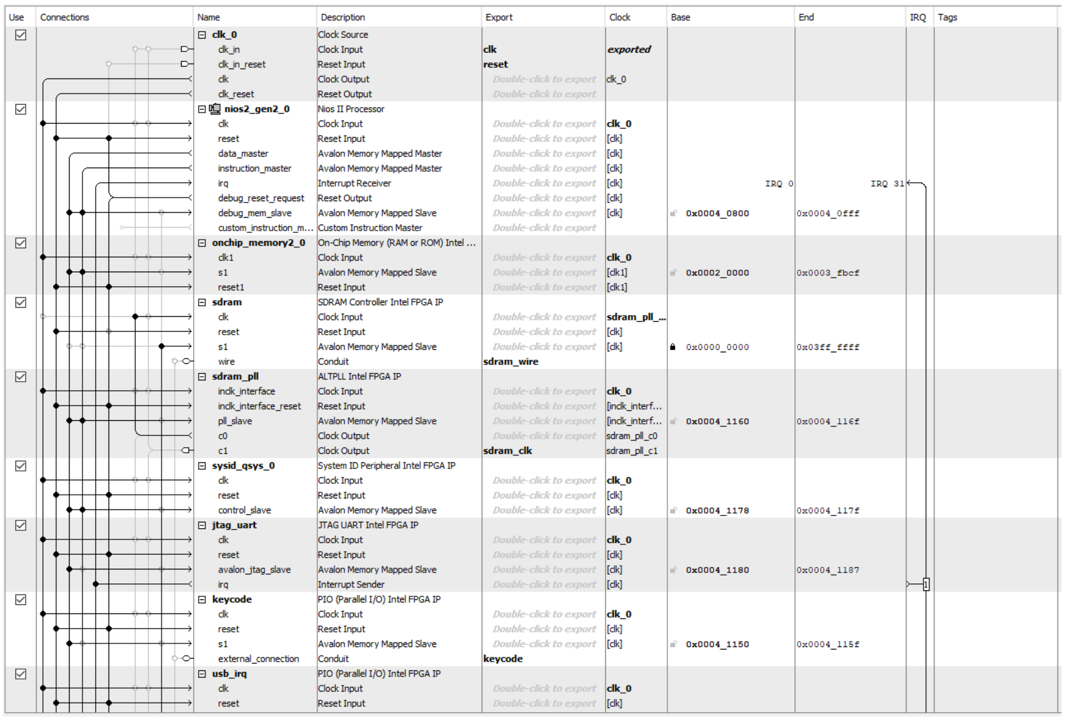


Memory Type: SDRAM (2000000h WORDS, 64 MB)
Random Access
Address: 00000000 wDATA: 0000 rDATA: 0000
Sequential Write
Address: 00000000 Length: 0 File Length
Sequential Read
Address: 00000000 Length: 0 Entire Memory

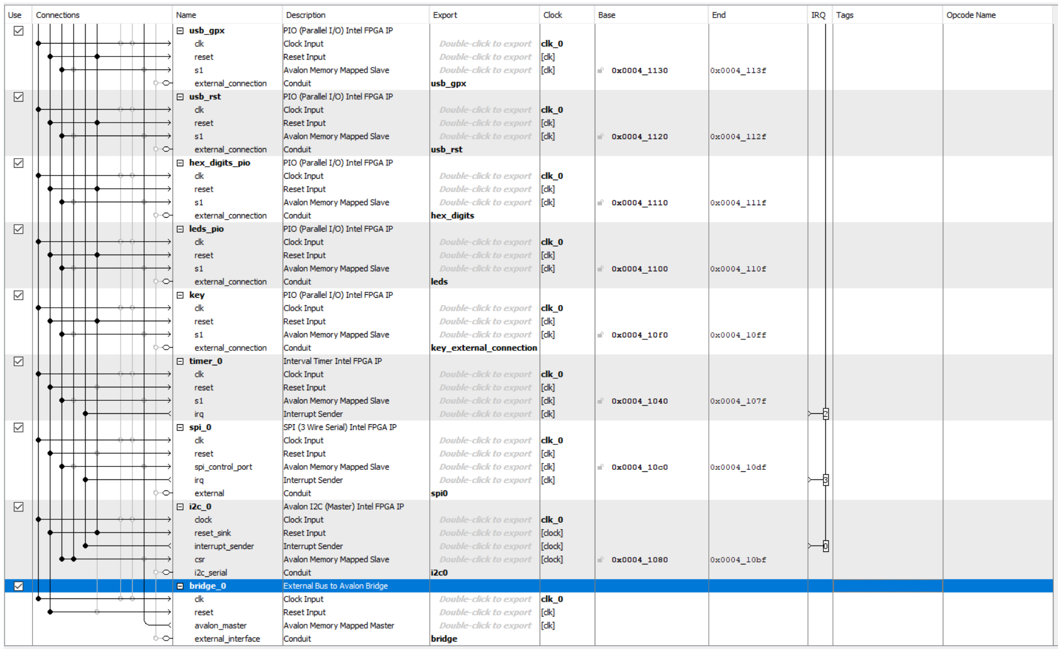
Fig. 20. Picture of Terasic GUI.

I also attempted to write a custom python script that could utilize the JTAG protocol to interface with the FPGA, and also to convert .wav files into .hex files, but it became too slow and impractical in comparison to debugging the SDRAM bridge IP.

Platform Designer



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
✓		**clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	***exported***				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to export*	clk_0				
		clk_reset	Reset Output	*Double-click to export*					
✓		**nios2_gen2_0**	Nios II Processor						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31		
		debug_reset_request	Reset Output	*Double-click to export*	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_0800	0x0004_0fff		
		custom_instruction_m...	Custom Instruction Master	*Double-click to export*					
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) Intel ...						
		clk1	Clock Input	*Double-click to export*	**clk_0**				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0002_0000	0x0003_fbcf		
		reset1	Reset Input	*Double-click to export*	[clk1]				
✓		**sdram**	SDRAM Controller Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**sdram_pll_...**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0000	0x03ff_ffff		
		wire	Conduit	**sdram_wire**					
✓		**sdram_pll**	ALTPLL Intel FPGA IP						
		inclk_interface	Clock Input	*Double-click to export*	**clk_0**				
		inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_interf...				
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interf...	0x0004_1160	0x0004_116f		
		c0	Clock Output	*Double-click to export*	sdram_pll_c0				
		c1	Clock Output	**sdram_clk**	sdram_pll_c1				
✓		**sysid_qsys_0**	System ID Peripheral Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1178	0x0004_117f		
✓		**jtag_uart**	JTAG UART Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1180	0x0004_1187		
		irq	Interrupt Sender	*Double-click to export*	[clk]			1	
✓		**keycode**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1150	0x0004_115f		
		external_connection	Conduit	**keycode**					
✓		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
✓		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1130	0x0004_113f			
		external_connection	Conduit	**usb_gpx**						
✓		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1120	0x0004_112f			
		external_connection	Conduit	**usb_rst**						
✓		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1110	0x0004_111f			
		external_connection	Conduit	**hex_digits**						
✓		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1100	0x0004_110f			
		external_connection	Conduit	**leds**						
✓		**key**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_10f0	0x0004_10ff			
		external_connection	Conduit	**key_external_connection**						
✓		**timer_0**	Interval Timer Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1040	0x0004_107f			
		irq	Interrupt Sender	*Double-click to export*	[clk]			2		
✓		**spi_0**	SPI (3 Wire Serial) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		spi_control_port	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_10c0	0x0004_10df			
		irq	Interrupt Sender	*Double-click to export*	[clk]			3		
		external	Conduit	**spi0**						
✓		**i2c_0**	Avalon I2C (Master) Intel FPGA IP							
		clock	Clock Input	*Double-click to export*	**clk_0**					
		reset_sink	Reset Input	*Double-click to export*	[clock]					
		interrupt_sender	Interrupt Sender	*Double-click to export*	[clock]			0		
		csr	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0004_1080	0x0004_10bf			
		i2c_serial	Conduit	**i2c0**						
✓		**bridge_0**	External Bus to Avalon Bridge							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		avalon_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]					
		external_interface	Conduit	**bridge**						

Fig. 21. Platform Designer Component Instantiations. USB PIO components were not used in my lab, just stayed over from when the .qsys file was being repurposed from my lab 6 to my final project. Notice SDRAM and SDRAM Bridge are disconnected from the rest of the components.

Component Descriptions:

1. Clk_0: *Provides the main clock utilized as the reference for all components.*
2. Nios2_gen2_0: Custom CPU part of the Intel Intellectual Property portfolio accessible for this course. The CPU is the source for the data bus that goes to other components of the SoC.
3. Sdram: SDRAM off-chip memory that stores our audio data.
4. Bridge_0: The Avalon bus to SDRAM bridge IP instantiated to be able to perform read/write operations with SDRAM from our HDL code.
5. Sdram_pll: There is a clock phase offset to account for when incorporating the SDRAM. Our ALTPLL IP component produces a phase-shifted clock that fixes this issue.
6. Sysid_qsys_0: Ensures that the system we want to operate on the hardware and software is matched, important in the case of multiple processors
7. Jtag_uart: JTAG UART IP is a component used here for the purpose of implementing basic console I/O between the FPGA and our Eclipse IDE. It is connected to the interrupts system.
8. Keycode: This PIO component has USB keycode input written to it by the processor which can then be read by the VGA hardware
9. Usb_irq, Usb_gpx, Usb_rst: Another PIO component that instantiates an interrupt service relayed to the embedded system program intended for instructing the MAX3421E USB controller. The interrupt is to take into account keyboard presses so the processor can be undergoing other operations but freeze to take care of registering a keyboard press, instead of having to do polling based I/O.
10. Hex_digits_pio: Another PIO module to utilize the 7-segment hex displays on the FPGA to display the keycode pertaining to the key pressed that was registered by the FPGA
11. Leds_pio: PIO Component used as a status register to display internal signals in the SoC
12. Key: Another PIO component used to take in input from the user through a button that indicates reset. Button also has inverted input.
13. Timer_0: Instantiates the Interval Timer IP to operate the USB driver
14. Spi_0: SPI stands for Serial Peripheral Interface and the 3 Wire Serial component is utilized for instantiating a master to slave system for the MAX3421E\

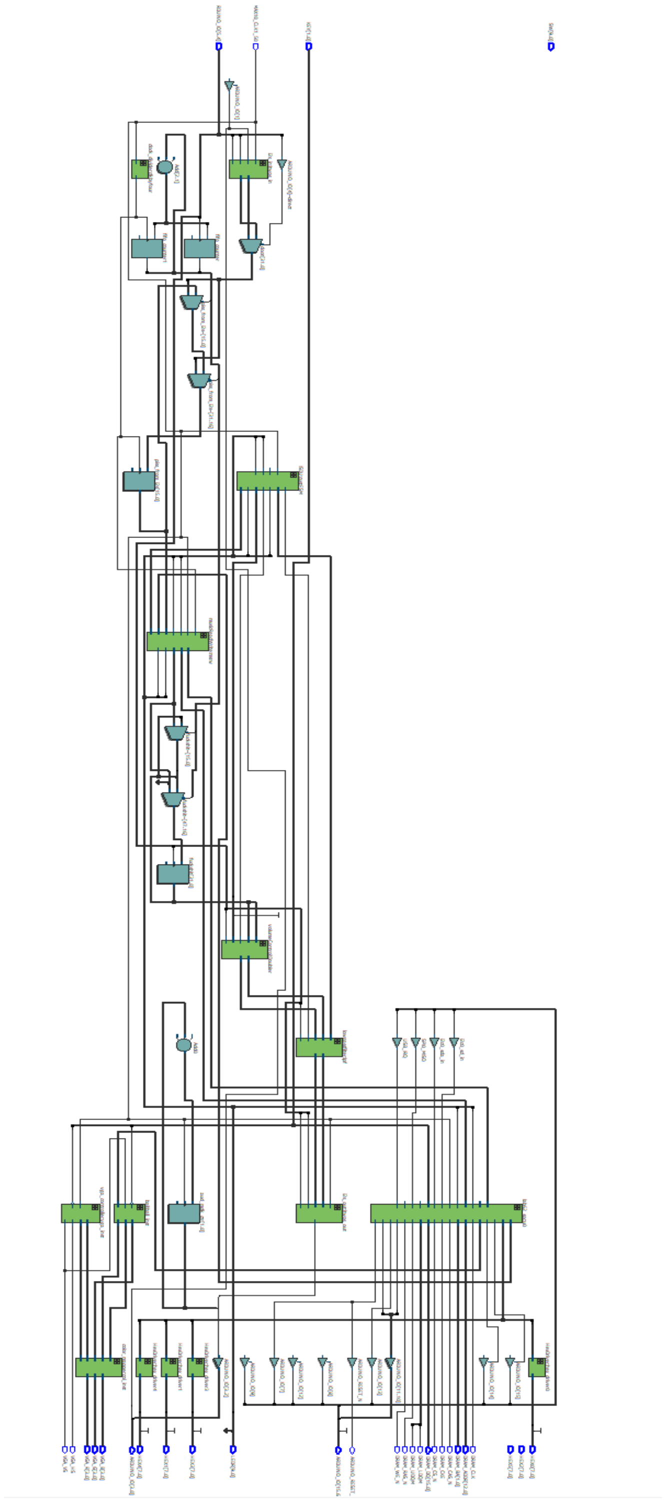


Fig. 23. RTL Diagram of full DAW.

FSM Diagram

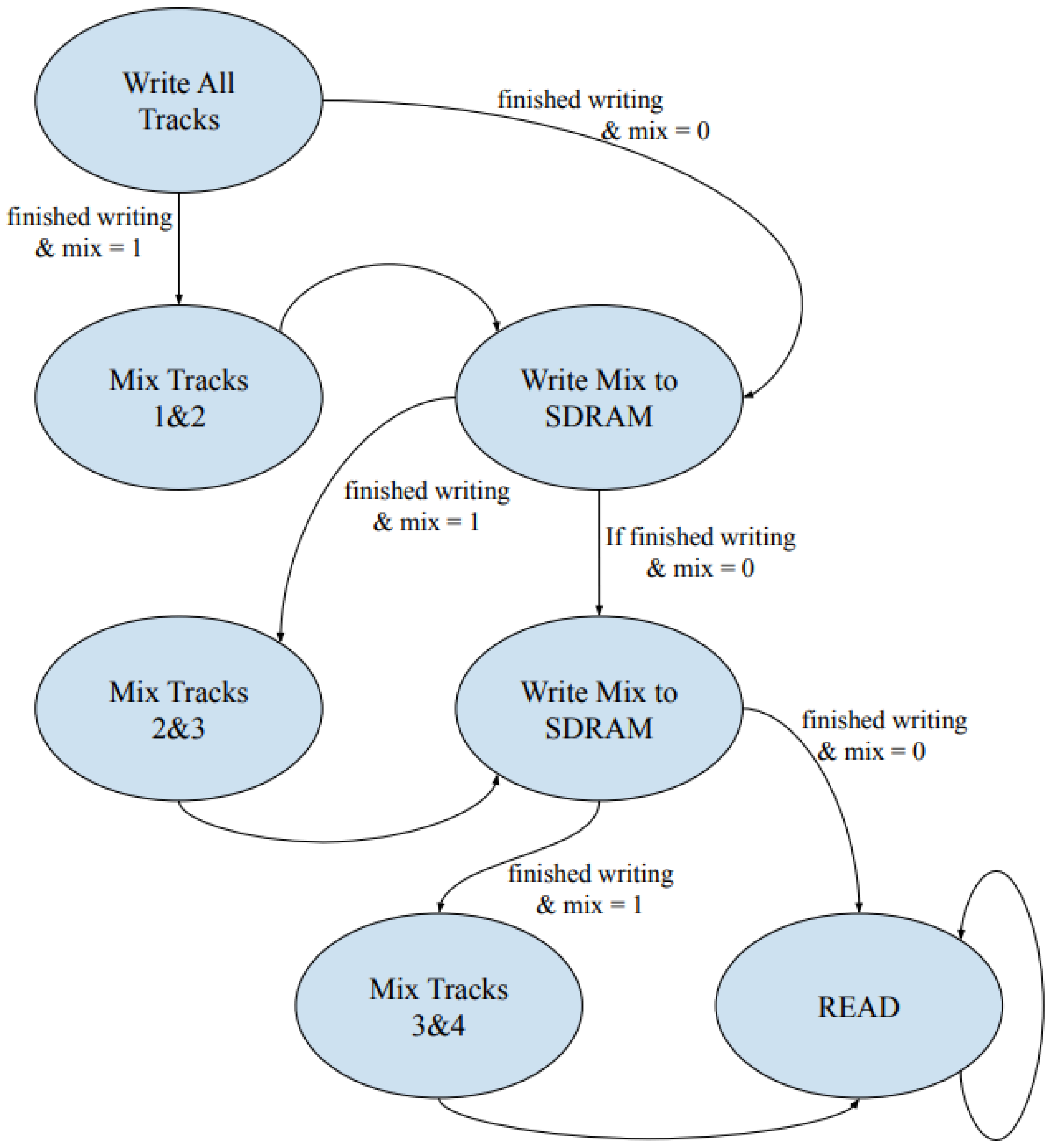

Write All Tracks
finished writing & mix = 0
finished writing & mix = 1
Mix Tracks 1&2
Write Mix to SDRAM
finished writing & mix = 1
If finished writing & mix = 0
Mix Tracks 2&3
Write Mix to SDRAM
finished writing & mix = 0
finished writing & mix = 1
Mix Tracks 3&4
READ

Fig. 24. Rough FSM from early in project close to ISDU module implementation. Control bits for macrofunction that performs read/writes with SDRAM are set, as well as I2S_in data routing. Keep in mind that Gain and EQ applications for individual tracks are hard set due to lack of keyboard input, and that Gain and EQ are also set for the mixed tracks through the switches, and don't need to be stated in an FSM because enable signals for those modules are set by switches. Also FSM would have FPGA loop through reading all tracks infinitely once mix is applied.