SWE-350 Design Report Template

Topic:	*Topic 4: C Programming Language: Memory Management, Arrays, Structures and Pointers*
Date:	*10/15/25*
Revision:	*1.0*
Milestone Summary:	(see table below)
GIT URL:	*The GIT URL that I can use to clone your code.*

User Story / Task	Hours Worked	Hours Remaining
Complete Design Report	*12*	*0*
Implement HPS-to-FPGA Verilog	*2*	*10*
Implement I2S Communication Verilog	*3*	*12*
Complete HPS logic, implement stub methods pseudocode	*5*	*20*

Design Documentation

General Technical Approach:

This project is an audio-based memory game running on Intel's SoC FPGA platform. Running on the DE10 Standard embedded board, it maps the four buttons to four unique audio samples and generates a sequence of these audio samples to play from the WM8731 audio codec through the line out port on the board. The goal of the player is to press the buttons which are mapped to the four audio samples to generate the same sequence of audio samples outputted by the board, with each round the sequence will increment by 1, increasing difficulty of the game. By utilizing both the Hard Processing System (HPS) and FPGA fabric, the application delivers a responsive interaction that combines audio, button input, and hex display feedback.

The HPS is responsible for executing the game logic features of this application. For example, the audio sample sequence generation/randomization as well as keeping track of sequence length. The HPS will check to see if the user's button input is correct and matches the sequence, if not it is also responsible for resetting the game upon failure. The HPS communicates the audio samples to play over the lightweight AXI bridge into the FPGA's memory mapped I/O.

Audio output is configured using an I^{2}C line connected to the FPGA and HPS, although the HPS only is responsible for the actual configuration. The output is set to operate in 16-bit I^{2}S mode at 48 kHz so that it is prepared for audio input from the FPGA over I^{2}S. The audio codec functions in slave mode, allowing the clock cycles to be set by the FPGA through its VLSI logic. Game progress and feedback such as current sequence length is displayed on the hex display.

Development of HPS logic such as '.c' files was performed using Visual Studio Code, while compilation, linking, and running occurs through a Makefile-based build system. The structure of the project is modular and separated into sections in the filesystem as shown in *Figure 10*.

System Design

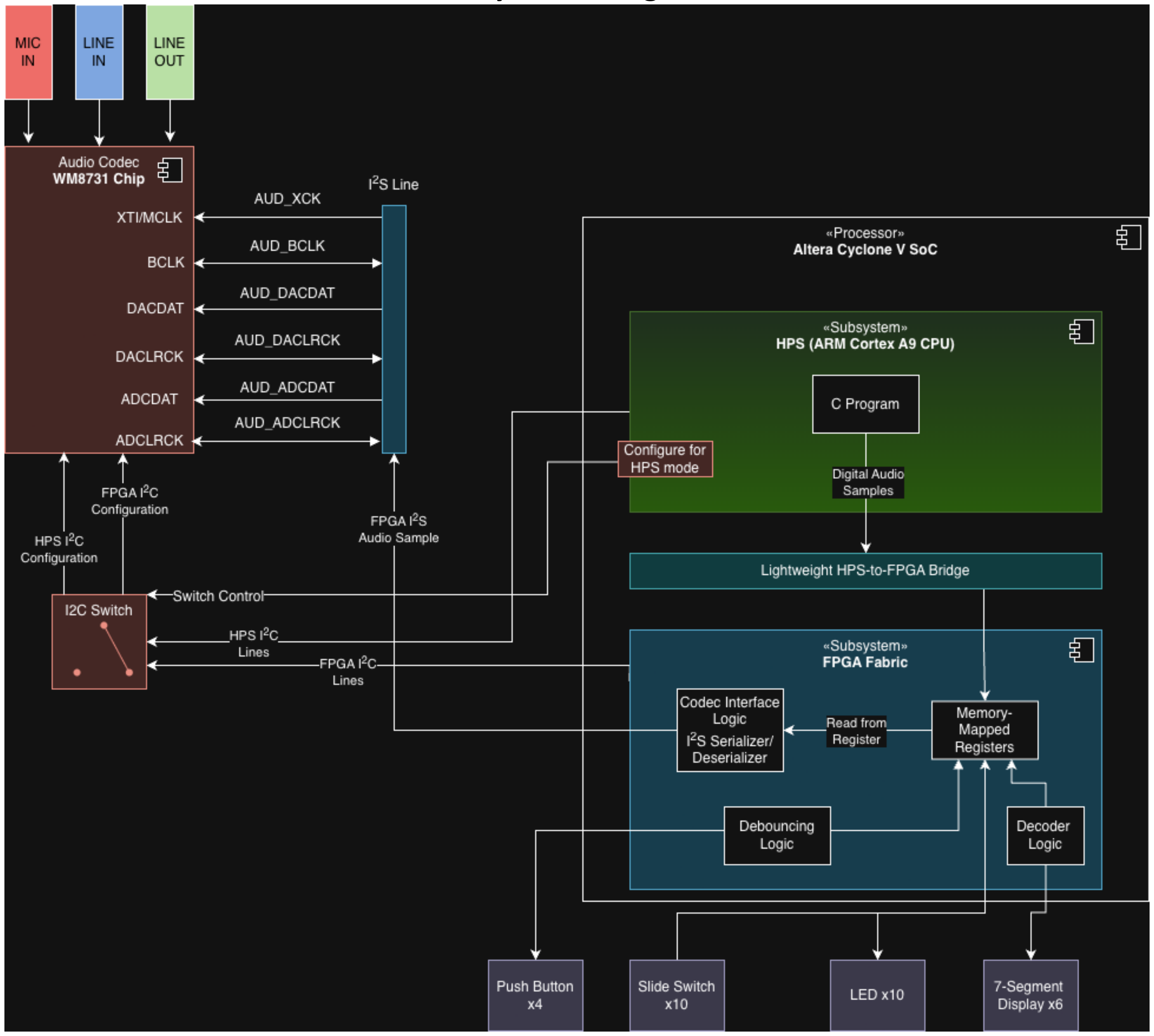


Figure 1: System Design – UML Component Diagram

Application Design

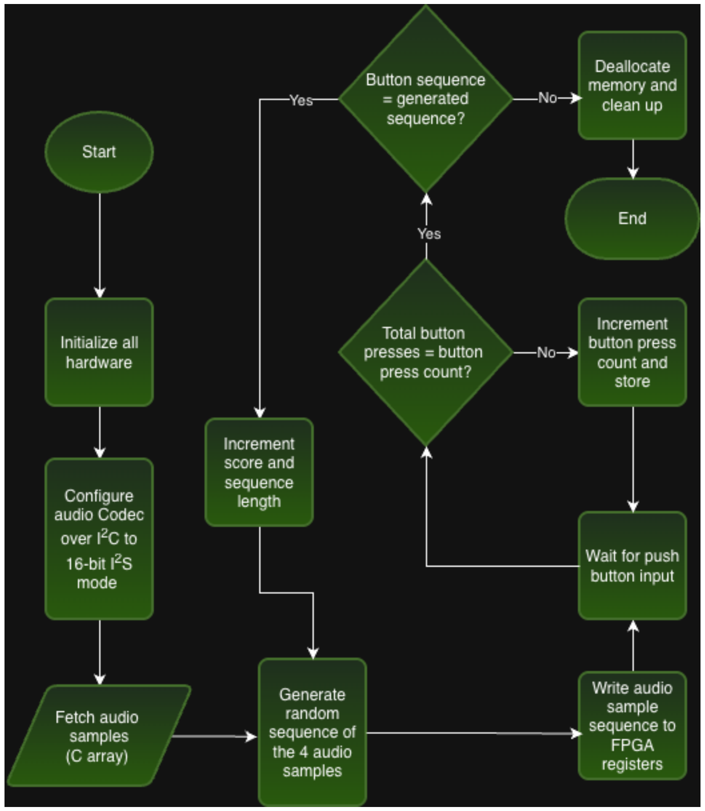

Start
Initialize all hardware
Configure audio Codec over I²C to 16-bit I²S mode
Fetch audio samples (C array)
Generate random sequence of the 4 audio samples
Increment score and sequence length
Write audio sample sequence to FPGA registers
Wait for push button input
Increment button press count and store
Total button presses = button press count?
No
Yes
Button sequence = generated sequence?
Yes
No
Deallocate memory and clean up
End

Figure 2: Application Design – HPS Logic Diagram

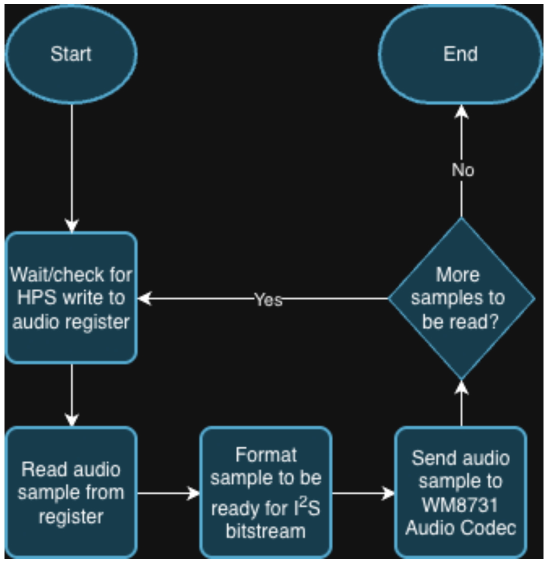

Start
End
No
Wait/check for HPS write to audio register
Yes
More samples to be read?
Read audio sample from register
Format sample to be ready for I2S bitstream
Send audio sample to WM8731 Audio Codec

Figure 3: Application Design – FPGA Logic Diagram

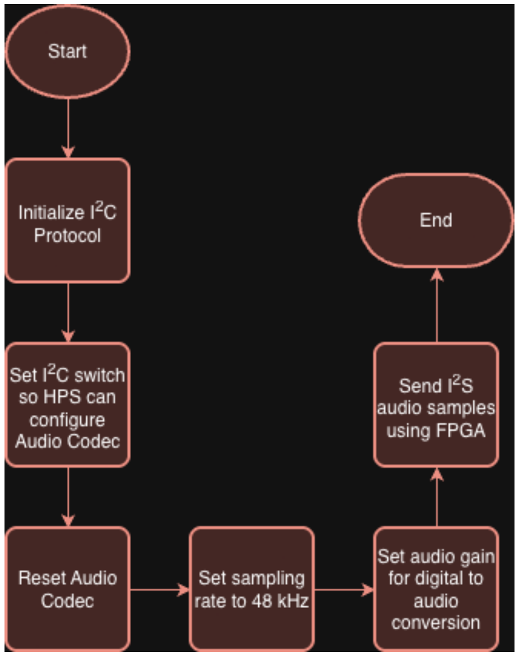

Start
Initialize I2C Protocol
End
Set I2C switch so HPS can configure Audio Codec
Send I2S audio samples using FPGA
Reset Audio Codec
Set sampling rate to 48 kHz
Set audio gain for digital to audio conversion

Figure 4: Application Design – Audio Codec I^{2}C Configuration Logic

```
START
// Initialize all hardware
Init_hardware()
Configure_codec(bit_depth, sampling_rate, gain)

// Game tracking variables
Sequence = []
Pattern_length = 1
Game_over = FALSE

     WHILE !(Game_over)

          New_sample = generate_sample()
          Sequence.append(new_sample)

               FOR sample IN Sequence
                    Sample_to_FPGA(sample)
                    Wait_for_playback()
               END FOR
               // User input
               FOR num IN pattern_length
                    Input = read_button()
                    If(Input != sequence[num])
                         Game_over = TRUE
                         BREAK
                    END IF
               END FOR

               IF NOT Game_over
                    Pattern_length+=1
               END IF
     END WHILE

     Game_complete_display()
END
```

Figure 5: Application Design – Main Application Logic Pseudocode

```
Module hps_to_fpga (  // This is for using the HPS to write to the FPGA registers for left and right audio
     Input clk, // FPGA system clock
     Input rst, // For reset
     Input [15:0] hps_sample_left,    // Audio sample for the left channel, 16-bit
     Input [15:0] hps_sample_right,   // Audio sample for the right channel, 16-bit
     Input hps_data_ready, // Flag from HPS for when new data is ready
     Output reg [15:0] sample_left,   // Current sample register to be sent for left side
     Output reg [15:0] sample_right,  // Current sample register to be sent for right side
     Output reg sample_ready     // Signal for I2S module that new data is ready
);

     Always @(posedge clk or posedge rst) begin  // This executes when on the rising edge of the clock
                                                 // Or when the reset goes high, acting as a flip-flop

          If(rst) begin    // If reset is high, set everything to zero and clear sample_ready
               Sample_left <= 16'b0
               Sample_right <= 16'b0
               Sample_ready <= 1'b0
          End
          Else if (hps_data_ready) begin
               Sample_left <= hps_sample_left
               Sample_right <= hps_sample_right
               Sample_ready <= 1'b1
          End
          Else begin
               Sample_ready <= 1'b0
          End
     End
Endmodule
```

Figure 6: VLSI Design – HPS to FPGA Register Write Verilog Logic Outline

```
Module 12s_transmission (
      Input clk, // FPGA clock
      Input rst, // For reset
      Input [15:0] sample_left,  // Left side sample
      Input [15:0] sample_right, // Right side sample
      Input sample_ready    // Flag for when the sample is ready
      Output reg i2s_sck,   // Serial clock for audio codec
      Output reg i2s_ws,    // Word select (left/right)
      Output reg i2s_sd     // Serial audio data

      Reg [4:0] bit_count   // 0-15 for 16-bit samples
      Reg [15:0] shift_register
      Reg current_channel   // 0 for left, 1 for right

      Reg [7:0] sck_div     // Clock divider counter (increments each time FPGA clock ticks)
                            // We use this to
      Always @(posedge clk or posedge rst) begin
            If(rst) sck_div <= 0
            Else sck_div <= sck_div + 1
      End
      Always @(posedge clk) i2s_sck <= sck_div[7] // Output clock frequency is <= frequency at sck_div[7]

      // We use the clock divider because the FPGA base clock is 50 MHz. Therefore, to set sampling rate
      // to the desired 48 kHz, we divide it down by a power of 2 each bit of sck_div, sck_div[4] is
      // approximately 1.56 MHz, where 1.536 MHz is the desired output frequency.
      // We get 1.536 MHz from the desired 48kHz sampling rate in the configuration, multiplied
      // by the 16 bit sample size and the 2 channels for left and right sides.
// This edge trigger logic is for shifting into the shift register
     Always @(negedge i2s_sck or posedge rst) begin
          If (rst) begin  // Reset condition
                bit_count <= 0;
                shift_register <= 16'b0;
                current_channel <= 0;
                i2s_ws <= 0;
                i2s_sd <= 0;
          End

          Else begin
          If (bit_cnt == 0) begin
               // Load new sample at start of channel
               Shift_reg <= current_channel ? sample_right : sample_left;
               I2s_ws <= current_channel;
               Current_channel <= ~current_channel;
          End
          Else begin
               // Shift out next bit
               Shift_register <= {shift_register[14:0], 1'b0};
          End
          i2s_sd <= shift_register[15];
          Bit_count <= (bit_count + 1) % 16;
          End
     End
Endmodule
```

Figure 7: VLSI Design – I2S Line Data Transmission Verilog Logic Outline

Risks and Issues:

Risk/Issue	Description	Impact	Likelihood	Priority	Mitigation
Timing Mismatch	Because we are setting the clock rate for the codec using the clock division method (shown above in *Figure 7)*, the FPGA clock division might not match the sample rate needed for the codec.	High	Medium	High	Adjust the FPGA clock division in the FPGA Verilog logic.
HPS and FPGA out of sync	The FPGA must be configured in such a way that the HPS only sends data when it is ready to receive data in its registers. There is a risk that these are not in sync with each other and data is sent when FPGA is not ready.	High	Low	Medium	Use a handshaking method between HPS and FPGA, something like the flag hps_data_ready in the Verilog logic above (*Figure 6).*
System Complexity	The audio I2S pipeline from the C program on the HPS to the audio codec is highly complex, so there is a risk of faulty system design resulting in functional issues or system failure.	High	High	High	Mirror existing official documentation from Terasic, conduct extensive research on communication protocols like I2S and I2C, as well as the WM8731 audio codec.
Codec Misconfiguration	When configuring the audio codec over I2C	High	Low	Medium	Validate the correct register writes

	using the HPS, an incorrect configuration (wrong sampling rate, gain, etc.) (see *Figure 4*), can cause prevented audio output.				happen over I2C, test with a known audio codec configuration.

Other Documentation

One important aspect to understand about the I^{2}S communication between the FPGA and the audio codec chip in this project is that, unlike a system where the codec chip works as a master, determining the necessary clock cycles for itself, it is better in this application for the audio codec chip to be in slave configuration, letting the FPGA generate the necessary clocks. The two clocks here are the serial clock (sck) and word select (ws) pictured below in *Figure 8* and as used in the Verilog logic in *Figure 7. Figure 8* shows a block diagram for what the shift register that receives the audio data might look like with a master codec design. Note that the below diagram is a master configuration, not slave configuration.

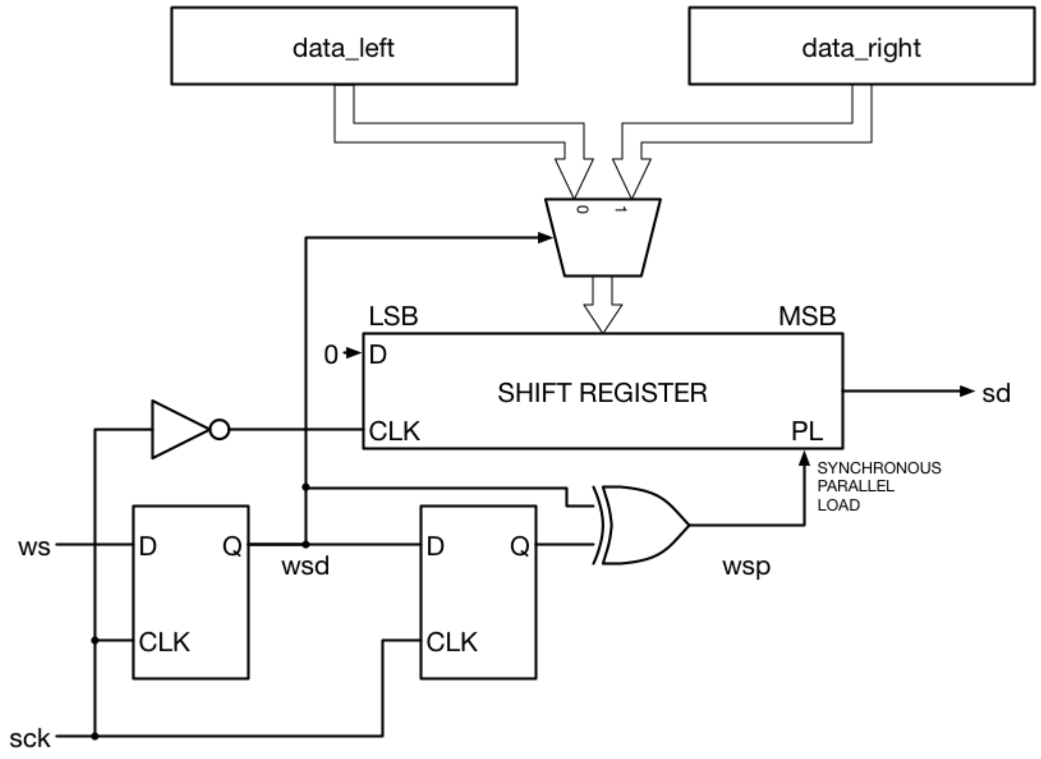


Figure 8: Traditional I2S transmitter block diagram. Reprinted from Beyond Circuits, "Tutorial 19: I2S Transmitter," https://www.beyond-circuits.com/wordpress/tutorial/tutorial19/.

Using a master configuration as shown above in this embedded application could result in inaccurate audio samples because there is a risk of generating inaccurate frequencies since it is handled by the codec. The bigger risk here, however, is synchronizing the FPGA clock with the WM8731 Codec clock. If the FPGA handles the clock generation, such as in a slave design, the FPGA has full control over timing, shown by the FPGA clock S_AXIS_ACLK shown in *Figure 9*. This makes it far easier to synchronize HPS audio sample writes to FPGA registers that will output audio, however the FPGA must generate accurate clocks. This is an important distinction because it affects the Verilog significantly; the FPGA must generate clocks and thus, requires modules that can help the system function in slave mode.

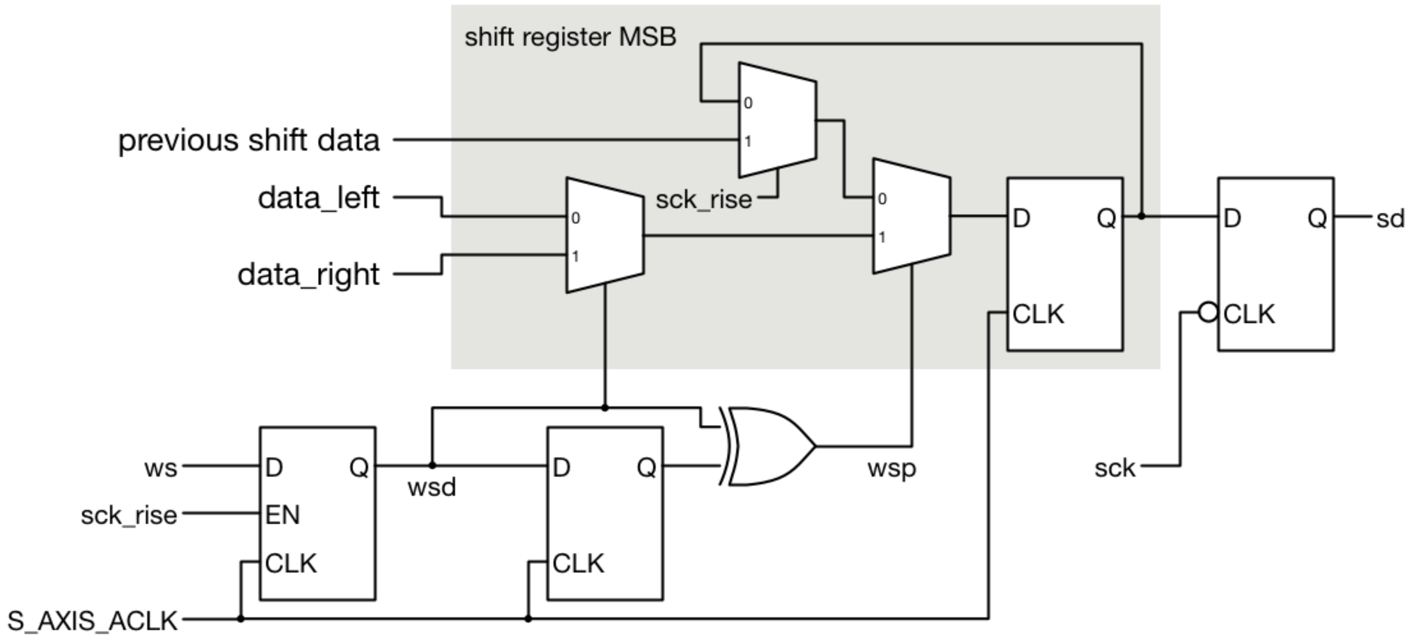
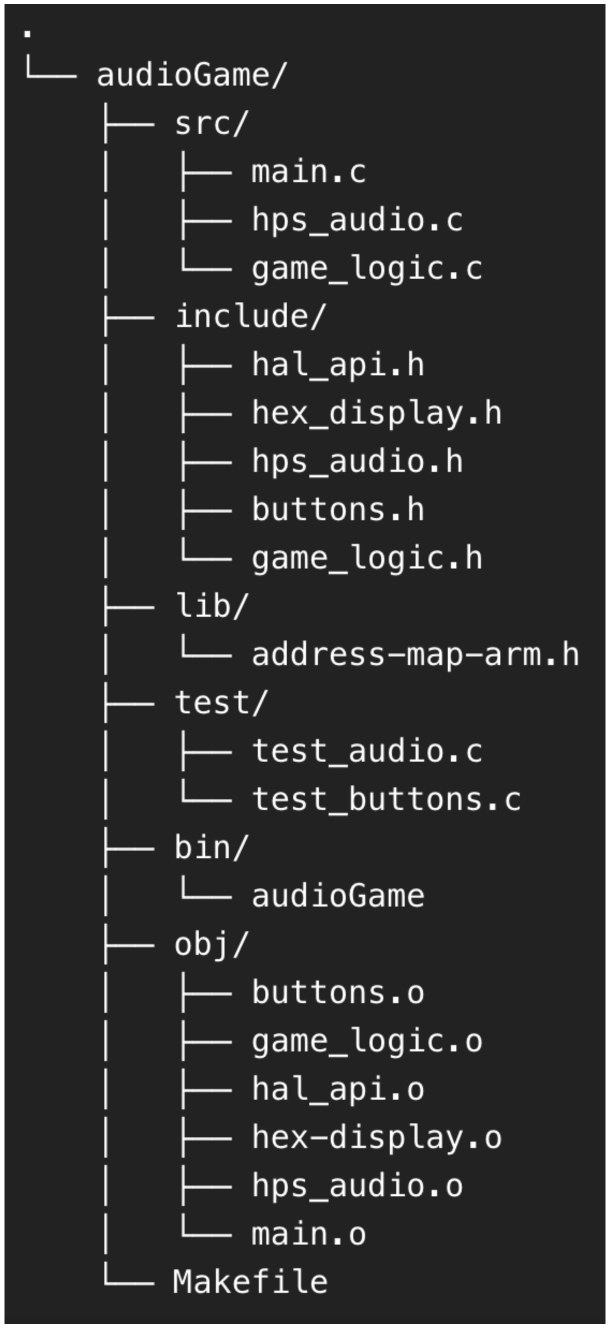


Figure 9: FPGA Clock I2S transmitter block diagram. Reprinted from Beyond Circuits, "Tutorial 19: I2S Transmitter," https://www.beyond-circuits.com/wordpress/tutorial/tutorial19/.



```
.
└── audioGame/
    ├── src/
    │   ├── main.c
    │   ├── hps_audio.c
    │   └── game_logic.c
    ├── include/
    │   ├── hal_api.h
    │   ├── hex_display.h
    │   ├── hps_audio.h
    │   ├── buttons.h
    │   └── game_logic.h
    ├── lib/
    │   └── address-map-arm.h
    ├── test/
    │   ├── test_audio.c
    │   └── test_buttons.c
    ├── bin/
    │   └── audioGame
    ├── obj/
    │   ├── buttons.o
    │   ├── game_logic.o
    │   ├── hal_api.o
    │   ├── hex-display.o
    │   ├── hps_audio.o
    │   └── main.o
    └── Makefile
```

Figure 10: File Tree Structure for Audio Game project

Reference

Beyond Circuits. (n.d.). Tutorial 19: I2S Transmitter. Retrieved October 18, 2025, from https://www.beyond-circuits.com/wordpress/tutorial/tutorial19/