Nexys 4 DDR Microphone and Audio Implementation

Jordan Reeser

December 9, 2018

1 Executive Summary

For this project, the designer chose to learn about the on-board omnidirectional MEMS microphone and mono audio output port on the Nexys 4 DDR FPGA by programming it to take data from the microphone and play it through the audio port. To accomplish this task, modules were created for each device along with a top module to connect the two devices. Then, the designer created wrapper modules for the microphone and audio output port and attempted to implement the devices into the Vanilla SoC provided by Pong Chu to enable sending the sound data from the microphone to the audio output. The modules to program the board to play microphone data through the audio output were successful, however the full implentation of this functionality into the Vanilla SoC was not successful. The GitHub repository containing all of the project files can be accessed using the following link: `https://github.com/Joreeser/Microphone_Nexys4ddr`

2 Problem

To learn about interfacing with the omnidirectional MEMS microphone and mono audio output on the Nexys 4 DDR, the goal of this project was to successfully program the FPGA to read sound from the microphone and play it in real-time through the audio output and then implement this same functionality into the Vanilla SoC. The devices have specifications that are available in the Nexys 4 DDR Reference Manual that need to be met in order to ensure proper functionality.

2.1 Microphone

The microphone embedded in the Nexys 4 DDR is an omnidirectional MEMS microphone. It has three wires: clock, L/R Sel, and data. The clock for the microphone is generated by the board and needs to be at a frequency between 1 MHz and 3.3 MHz. The L/R Sel input determines whether data is read on the

positive edge or negative edge of the microphone clock; a value of 0 reads data on the positive edge while 1 reads data on the negative edge. The data output is sent as a single bit in Pulse Density Modulation format.

2.2 Audio Output

The mono audio output on the Nexys 4 DDR has two inputs: power and data. The power signal is set to 1 to turn the device on while a value of 0 turns it off. The data signal inputs the sound data that will be sent through connected headphones.

3 FPGA Programming Modules

In order to implement the desired design on the FPGA, three modules were needed:

1. Microphone_practice
2. mic_test
3. audio_out

These modules were implemented successfully to play sound read by the microphone through the audio output in real time.

3.1 Microphone_practice

This module is the top module for the basic programming portion of the project. It instantiates both the microphone and audio output modules and connects the data from the microphone to the audio output as well as connecting all of the appropriate board signals. This module was used only for testing purposes and was not further implemented into the Vanilla SoC.

3.2 mic_test

This module controls the microphone functionality. It sets up a 5-bit counter that creates the clock for the microphone utilizing the most significant bit; the microphone clock frequency must lie between 1 MHz and 3.3 MHz for operation, and this counter lies within that range. The module also sets the L/R Sel value to 0 and reads the microphone data on the positive edge of the microphone clk, which is appropriate for the low L/R Sel value.

3.3 audio_out

The audio_out module sets up the control of the audio output. It simply sets the port power (either 1 or 0 for on and off respectively) and inputs the data. In this case, the data entering the audio module is the microphone data, and the power level is hard-wired high to turn on power.

4 SoC Implementation

The second part of the project was to implement the microphone/audio setup into the Vanilla SoC provided by Pong Chu in his book *FPGA Prototyping by SystemVerilog Examples*. To accomplish this task, wrappers were made for the microphone and audio output modules, and these wrappers were placed into empty slots in the FPRO bridge. To control the functionality of the SoC, the main_vanilla_test.cpp file created by Pong Chu was edited to include a new function that turns the audio output on and sends the microphone output data to the audio output. Header files were also created for the microphone and audio output to set up their respective classes and functions.

4.1 Microphone

The microphone class was set up to connect the microphone to its FPRO bus slot, slot 14. A function is included called load_mic_data that simply loads the data read by the microphone. This function is used in main_vanilla_test.cpp to recieve the microphone data to send to the audio output. The implementation of the microphone into the Vanilla SoC was not successful–no sound data is played through the audio output.

4.2 Audio Output

The Audio Output class connects the audio port to its FPRO bus slot, slot 15. It includes two functions: set_power and set_data. These functions set the audio output power level (1 for on or 0 for off) and provide the audio output with data. In main_vanilla_test.cpp, the power of the audio output is set to 1 and the data from the microphone is entered into the audio data. The implementation of the audio port into the Vanilla SoC is assumed to be successful since users can hear the audio port turn on when headphones are connected despite no sound data being played after.

5 Conclusions

There were several successes and failures experienced in this final project. The first task of interfacing with the microphone and audio output on the FPGA through independant modules was successful and real-time sound detected by the microphone was able to be played through headphones connected to the audio output. Through this success, the designer was able to learn about the functionality of the microphone and audio output. However, implementation of the design into the Vanilla SoC was not successful; the audio connection is assumed to have been successful due to its ability to turn on, but no sound data is sent from the microphone. This could be due to a variety of factors such as incorrect creation of the wrapper modules or C-files, or it is possible that the timing of the microphone clock was slowed in the SoC. The microphone has

specific timing for the frequency and data collection that needs to be met in order to send correct data. A previous lab displayed a difference in a clock generated by a module in the SoC verses standing alone; the refresh rate generated for the SSEG project was much slower in the SoC implementation than when the top module alone was programmed to the FPGA. To fix this issue, the designer would have to create a clock that is guaranteed to operate within the specified 1 MHz-3.3 MHz frequency range for the microphone within the SoC.

6 Code Appendix

Listing 1: Verilog code for implementing the microphone.

```
`timescale 1ns / 1ps

module mic_test(
    input logic clk,
    input logic reset,
    input logic M_DATA,
    output logic M_CLK,
    output logic M_LRSEL,
    output logic data_out
    );

    logic [4:0] mic_clk_counter;
    logic pwm_val_reg;

    always_ff @(posedge clk)
    begin
        mic_clk_counter <= mic_clk_counter + 1;

        if (mic_clk_counter == 5'b01111)
            pwm_val_reg <= M_DATA;
    end

    assign M_CLK = mic_clk_counter[4];
    assign M_LRSEL = 0;
    assign data_out = pwm_val_reg;

endmodule
```

Listing 2: Verilog code for implementing the audio output

```
`timescale 1ns / 1ps

module audio_out(
```

```
    //input logic clk,
    input logic power,
    input logic data_in,
    output logic audio_pdm,
    output logic audio_on
    );

    assign audio_on = power;
    assign audio_pdm = data_in;

endmodule
```

Listing 3: C++ code for implementing the SoC functionality with the microphone and audio output.

```
/*****************************************************************//**
 * @file main_vanilla_test.cpp
 *
 * @brief Basic test of 4 basic i/o cores
 *
 * @author p chu
 * @version v1.0: initial release
 *****************************************************************/

//#define _DEBUG
#include "chu_init.h"
#include "gpio_cores.h"


/**
 * blink once per second for 5 times.
 * provide a sanity check for timer (based on SYS_CLK_FREQ)
 * @param led_p pointer to led instance
 */
void timer_check(GpoCore *led_p) {
   int i;

   for (i = 0; i < 5; i++) {
      led_p->write(0xffff);
      sleep_ms(500);
      led_p->write(0x0000);
      sleep_ms(500);
      debug("timer check - (loop #)/now: ", i, now_ms());
   }
}
```

```
/**
 * check individual led
 * @param led_p pointer to led instance
 * @param n number of led
 */
void led_check(GpoCore *led_p, int n) {
   int i;

   for (i = 0; i < n; i++) {
      led_p->write(1, i);
      sleep_ms(200);
      led_p->write(0, i);
      sleep_ms(200);
   }
}

/**
 * leds flash according to switch positions.
 * @param led_p pointer to led instance
 * @param sw_p pointer to switch instance
 */
void sw_check(GpoCore *led_p, GpiCore *sw_p) {
   int i, s;

   s = sw_p->read();
   for (i = 0; i < 30; i++) {
      led_p->write(s);
      sleep_ms(50);
      led_p->write(0);
      sleep_ms(50);
   }
}

/**
 * uart transmits test line.
 * @note uart instance is declared as global variable in chu_io_basic.h
 */
void uart_check() {
   static int loop = 0;

   uart.disp("uart test #");
   uart.disp(loop);
   uart.disp("\n\r");
   loop++;
}
```

```
// Audio in from mic, out from audio output
void audio_play(MicCore *mic_p, AudioCore *audio_p) {
        int data;

        data = (int)mic_p->load_mic_data();
        audio_p->set_power((int)1);
        audio_p->set_data(data);
}


// instantiate switch, led
GpoCore led(get_slot_addr(BRIDGE_BASE, S2_LED));
GpiCore sw(get_slot_addr(BRIDGE_BASE, S3_SW));
//GpiCore mic(get_slot_addr(BRIDGE_BASE, S14_MIC));
//GpoCore audio(get_slot_addr(BRIDGE_BASE, S15_AUDIO));

MicCore mic(get_slot_addr(BRIDGE_BASE, S14_MIC));
AudioCore audio(get_slot_addr(BRIDGE_BASE, S15_AUDIO));


int main() {

   while (1) {
          audio_play(&mic, &audio);
      timer_check(&led);
      led_check(&led, 16);
      sw_check(&led, &sw);
      uart_check();
      debug("main - switch value / up time :", sw.read(), now_ms());
   } //while
} //main
```

Listing 4: C++ code for implementing the microphone core with function to load microphone data.

```
/*****************************************************************//**
 * @file mic_core.cpp
 *
 * @brief implementation of MicCore class
 ****************************************************************/

#include "mic_core.h"

MicCore::MicCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
}
```

```
MicCore::~MicCore() {
}

int MicCore::load_mic_data() {
        return (io_read(base_addr, DATA_REG));
}
```

Listing 5: C++ header file for the mic_core.

```
/*****************************************************************//**
 * @file mic_core.h
 *
 * @brief access microphone
 *****************************************************************/

#ifndef _MIC_CORE_H_INCLUDED
#define _MIC_CORE_H_INCLUDED

#include "chu_io_rw.h"
#include "chu_io_map.h"
/**
 * Microphone core driver
 * - Take in data from microphone
 */
class MicCore {
public:
// Register map
   enum {
      DATA_REG = 0, // Data register
   };

   // Constructor
   MicCore(uint32_t core_base_addr);
   ~MicCore();

   // Load mic data
   int load_mic_data();

private:
   uint32_t base_addr;
};

#endif  // _MIC_CORE_H_INCLUDED
```

Listing 6: C++ code for implementing the audio core with functions to set power and set data.

```
/*****************************************************************//**
 * @file audio_core.cpp
 *
 * @brief implementation of AudioCore class
 *****************************************************************/

#include "audio_core.h"

AudioCore::AudioCore(uint32_t core_base_addr) {
    base_addr = core_base_addr;
    set_power(1); // Default value
}

AudioCore::~AudioCore() {
}

void AudioCore::set_power(int power) {
    io_write(base_addr, POWER_REG, power);
}

void AudioCore::set_data(int data) {
        io_write(base_addr, DATA_REG, data);
}
```

Listing 7: C++ header file for the audio_core.

```
/*****************************************************************//**
 * @file audio_core.h
 *
 * @brief Access audio output and
 *        set power and data values
 *****************************************************************/

#ifndef _AUDIO_CORE_H_INCLUDED
#define _AUDIO_CORE_H_INCLUDED

#include "chu_io_rw.h"
#include "chu_io_map.h"
/**
 * Audio core driver
 * - Set power level
 * - Set data to be played
 */
class AudioCore {
```

```
public:
// Register map
    enum {
        DATA_REG = 0, // Data register
            POWER_REG = 1 // Power register
    };

    // Constructor
    AudioCore(uint32_t core_base_addr);
    ~AudioCore();

    void set_power(int power);

    void set_data(int data);

private:
    uint32_t base_addr;
};

#endif  // _AUDIO_CORE_H_INCLUDED
```