ELC 4396 02
Final Report

Jordan Cook

December 5, 2021

1 Introduction and Problem Statement

This project was a final lab to demonstrate a feature learned in this class. It utilized several concepts, including both GPI and GPO, the XADC, the SSEG-display, and PWM. With all these components put together, a system was built that would read the voltage or the temperature on the board's chip while giving visual feedback and taking user input. Hardware was written using System Verilog in Vivado and software was written using C++ in Vitis.

Link to the Project: https://github.com/jordanstarr/System-on-Chip
Functionality was demonstrated in person and presented to the class.

2 Method and Approach

2.1 Hardware

Like most previous assignments, there were 16 files that needed to be imported into Vivado in order to make things work.Those included files were:

- chu_gpi.sv
- chu_gpo.sv
- chu_timer.sv
- fifo.sv
- fifo_ctrl.sv
- reg_file.sv
- chu_mmio_controller.sv
- baud_gen.sv
- chu_uart.sv
- uart.sv
- uart_rx.sv
- uart_tx.sv
- chu_io_map.sv
- chu_mcs_bridge.sv
- mmio_sys_sampler.sv
- mcs_top_sampler.sv

Several new files were needed to be added in order to make the FPGA programmable in software needed for this project. Those additional files were:

- chu_xadc_core.sv
- chu_fpro.sv
- chu_io_pwm_core.sv
- chu_led_mux_core.sv
- led_mux8.sv

With these additional files, they needed to be placed in the correct slots in the mmio_sys_vanilla. This was learned by viewing the other instantiations and changing them as needed for each of the modules individuals needs for input and output.

Listing 1: Slot Instantiations

```
chu_xadc_core xadc_slot5
   (.clk(clk),
   .reset(reset),
   .cs(cs_array[`S5_XDAC]),
   .read(mem_rd_array[`S5_XDAC]),
   .write(mem_wr_array[`S5_XDAC]),
   .addr(reg_addr_array[`S5_XDAC]),
   .rd_data(rd_data_array[`S5_XDAC]),
   .wr_data(wr_data_array[`S5_XDAC]),
   .adc_p(adc_p),
   .adc_n(adc_n)
   );

chu_io_pwm_core #(.W(W), .R(R)) gpi_slot6
   (.clk(clk),
   .reset(reset),
   .cs(cs_array[`S6_PWM]),
   .read(mem_rd_array[`S6_PWM]),
   .write(mem_wr_array[`S6_PWM]),
   .addr(reg_addr_array[`S6_PWM]),
   .rd_data(rd_data_array[`S6_PWM]),
   .wr_data(wr_data_array[`S6_PWM]),
   .pwm_out(pwm_out)
   );

chu_led_mux_core sseg_slot8
   (.clk(clk),
   .reset(reset),
   .cs(cs_array[`S8_SSEG]),
   .read(mem_rd_array[`S8_SSEG]),
   .write(mem_wr_array[`S8_SSEG]),
   .addr(reg_addr_array[`S8_SSEG]),
   .rd_data(rd_data_array[`S8_SSEG]),
   .wr_data(wr_data_array[`S8_SSEG]),
   .sseg(sseg),
   .an(an)
   );
```

There were several outputs/inputs that needed to be added to both the mmio and the top files. Those were the adc_p, adc_n, pwm_out, sseg, and an. In the top-level file, all the new outputs/inputs were created using the constraint file that Dr. Chu created for the Neyxs 4 DDR board. Here was a listing of all in outputs and inputs:

Listing 2: Top Level

```
module mcs_top_vanilla
#(parameter BRG_BASE = 32'hc000_0000)
   (
      input logic clk,
      input logic reset_n,
      // switches and LEDs
      input logic [15:0] sw,
      output logic [15:0] led,
      // uart
      input logic rx,
      output logic tx,
      // XADC
      input logic [3:0] adc_p,
      input logic [3:0] adc_n,
      // SSEG
      output logic [7:0] sseg,
      output logic [7:0] an,
      // pwm rgb LEDs
      output logic [2:0] rgb_led1,
      output logic [2:0] rgb_led2,
      //i2c
      output tri tmp_i2c_scl,
      inout tri  tmp_i2c_sda
   );
```

At this point, all the hardware was completed and a bit stream could be generated. Once generated, it was all exported into Vitis where software could be written.

2.2 Software

The first step in software was to add some of the files provided by Dr. Pong Chu. These files were:

- chu_init
- chu_init.h
- chu_io_map.h
- chu_io_rw.h
- gpio_cores.cpp
- gpio_cores.h
- timer_core.cpp
- timer_core.h
- uart_core.cpp
- uart_core.h
- sseg_core.cpp
- sseg_core.h
- xadc_core.cpp
- xadc_core.h
- main_vanilla_test.cpp

The final project utilized several pre-made functions as well as a plethora of newly created functions. The following listing shows the main function as well as the declaration of each of the classes used.

```
// instantiate switch, led
GpoCore led(get_slot_addr(BRIDGE_BASE, S2_LED));
GpiCore sw(get_slot_addr(BRIDGE_BASE, S3_SW));

//instantiate my new classes
XadcCore vcc(get_slot_addr(BRIDGE_BASE, S5_XDAC));
XadcCore temp(get_slot_addr(BRIDGE_BASE, S5_XDAC));
SsegCore seg(get_slot_addr(BRIDGE_BASE, S8_SSEG));
PwmCore rgb(get_slot_addr(BRIDGE_BASE, S6_PWM));

int main() {
  while (1) {
    bool switch0 = sw.read(0);
    bool switch1 = sw.read(1);
    bool switch2 = sw.read(2);

    bool isFahr = 0;

    if (switch0 && !switch1 && !switch2) {
      writeVCC(&vcc, &seg);
      voltageLED(&vcc, &led);
      colorRG(&rgb, &sw);
    }
    else if (!switch0 && switch1 && !switch2) {
      isFahr = 0;
      writeTemp(&temp, &seg, isFahr);
      tempLED(&temp, &led, isFahr);
      colorGB(&rgb, &sw);
    }
    else if (!switch0 && !switch1 && switch2) {
      isFahr = 1;
      writeTemp(&temp, &seg, isFahr);
      tempLED(&temp, &led, isFahr);
      colorBR(&rgb, &sw);
    }
    else {
      errorLED(&led);
      writeZero(&seg);
    }
  }
}
```

Listing 3: Declarations and Main

The general concept here is that the system checks to see which switch is flipped. If it's the first one, the system needs to go in voltage-mode, meaning that it writes the voltage on the chip to the seven-segment, lights up the appropriate amount of LEDs, and turns on an RGB light that switches from red to green. If the second switch is on, it goes into Celsius-Temperature mode, meaning it writes the temperature in Celsius to the seven-segment display, lights up the appropriate amount of LEDs, and turns on two lights that change from green to blue. Finally if the third switch is flipped, it goes into Fahrenheit-temp mode, writing the temperature in Fahrenheit to the seven-segment display, lighting up the appropriate amount of LEDs, and turning on two lights that change from blue to red. If there happens to be an error (such as a wrong switch being high or multiple switches being on), the code will turn everything off until it is fixed. This is meant to act like a safety feature. Although it isn't necessary in a project with this small of a scope, it is always good practice to have a default safety mode. (There is be more detail with each of these functions).

Three pre-made functions that was invaluable when reading the voltage on the chip was one the 'read_fpga_vcc', 'read_adc_in', and 'read_raw'. They all were functions of the Xadc class and already built in.

```
uint16_t XadcCore::read_raw(int n) {
  uint16_t rd_data;

  rd_data = (uint16_t) io_read(base_addr, n) & 0x0000ffff;
  return (rd_data);
}

double XadcCore::read_adc_in(int n) {
  uint16_t raw;
  raw = read_raw(n) >> 4;
  return ((double) raw / 4096.0);
}

// input source 5 is connected to vcc reading
double XadcCore::read_fpga_vcc() {
  return (read_adc_in(VCC_REG) * 3.0);
}
```

Listing 4: Xadc functions

The first function was writing a value to the seven segment display. (There might be an easier way to write to the LEDs, but this was what was figured out independently). The 'writeVCC' function took in an Xadc pointer and a sseg pointer as arguments. After reading the current voltage on the chip, this function would manipulate the value so that each digit would be displayed individually on the seven-segment. It would write a hexidecimal value to the display (one of the pre-made function). Finally, a decimal point was also included in the proper placement.

```
void writeVCC(XadcCore *vcc, SsegCore *seg) {
  double voltage;
  voltage = vcc -> read_fpga_vcc();
  voltage *= 1000;

  for (int i = 0; i < 8; i++) {
    seg->write_1ptn(0xff, i);
  }

  seg -> set_dp(0x00);

  uint8_t d1 = voltage / 1000;
  uint8_t d2 =(voltage - d1 * 1000) / 100;
  uint8_t d3 = (voltage - d1 * 1000 - d2 * 100) / 10;
  uint8_t d4 = (voltage - d1 * 1000 - d2 * 100 - d3 * 10);

  uart.disp(d3);
  uart.disp("\n\r");

  seg -> write_1ptn(seg -> h2s(d1), 3);
  seg -> write_1ptn(seg -> h2s(d2), 2);
  seg -> write_1ptn(seg -> h2s(d3), 1);
  seg -> write_1ptn(seg -> h2s(d4), 0);
  seg -> set_dp(8);
  sleep_ms(50);
}
```

Listing 5: Writing Voltage to 7-seg

The other function which is a counterpart of this one is the 'writeTemp' function. This is essentially the same exact thing as the 'writeVCC' function, the only difference is that is takes in a bool as an argument that will change the value from Fahrenheit fo Celsius as input by the user.

In order to understand the number level being presented on the seven-segment display, a level of LEDs were also included to light up based on a scale. Since there were 16 LEDs, a fraction of them were lit up in proportion to the current level (which could be either voltage or temperature). For example, the voltage was placed on a scale between 0 and 3.3V. So when the function would read a value around 3V, a total of 13 lights were light up. Both Celsius and Fahrenheit were put on a scale from 0 to 100 (granted it could get above and below that, but it would be damaging to the board at that point). These lights were light up one at a time like a fuel gauge. The following function shows how this was accomplished with the voltage level (the temperature function is essentially the same function).

```
void voltageLED(XadcCore *vcc, GpoCore *led_p) {

  double voltage;
  voltage = vcc -> read_fpga_vcc();

  double num = voltage / 3.3 * 16;

  for (int i = 0; i < num - 1; i++) {
    led_p->write(1, i);
    sleep_ms(60);
  }
}
```

Listing 6: Volage LED Light-up Function

To indicate to the user what type of value is being displayed on the board (whether it is voltage, temperature in Celsius, or temperature in Fahrenheit), a light switching from different colors of the rainbow is showing an active status. This is similar to a rainbow-LED project created in class report 4, but rather than switching all three colors, only two colors at a time are changed. This causes one color to need to ramp up in value while the other one is ramping down.

```
void colorGB(PwmCore *rgb, GpiCore *sw) {
  double bright1 = 1.0;
  double bright2 = 1.0;
  double duty1;
  double duty2;
  double P = 1.2589;
  double percent = 1.0;

  const int length = 4 * 20;

  int blue = 0;
  int green = 1;
  int red = 2;

  bool on;
  rgb -> set_freq(50);

  rgb -> set_duty(0, green);
  rgb -> set_duty(0, green + 3);
  rgb -> set_duty(0, red);
  rgb -> set_duty(0, red + 3);
  rgb -> set_duty(0, blue);
  rgb -> set_duty(0, blue + 3);

  for (int i = 0; i < length; i++) {
    on = sw -> read(1);

    if (!on) {
      rgb -> set_duty(0, green);
      rgb -> set_duty(0, green + 3);
      rgb -> set_duty(0, blue);
      rgb -> set_duty(0, blue + 3);
      rgb -> set_duty(0, red);
      rgb -> set_duty(0, red + 3);
      break;
    }

    if (i <= 20) {
      rgb -> set_duty(percent * 1.0, blue);
      rgb -> set_duty(percent * 1.0, blue + 3);

      rgb -> set_duty(0.0, green);
      rgb -> set_duty(0.0, green + 3);

      sleep_ms(5);
    }
    else if (i <= 40) {
      bright1 *= P;
      duty1 = bright1 / 100.0;

      bright2 /= P;
      duty2 = bright2;

      rgb -> set_duty(percent * duty1, green);
      rgb -> set_duty(percent * duty1, green + 3);
      rgb -> set_duty(percent * duty2, blue);
      rgb -> set_duty(percent * duty2, blue + 3);

      sleep_ms(100);
    }
    else if (i <= 60) {
      rgb -> set_duty(0.0, blue);
      rgb -> set_duty(0.0, blue + 3);

      rgb -> set_duty(percent * 1.0, green);
      rgb -> set_duty(percent * 1.0, green + 3);

      sleep_ms(5);
    }
    else if (i <=80) {
      bright1 /= P;
      duty1 = bright1 / 100.0;

      bright2 *= P;
      duty2 = bright2;

      rgb -> set_duty(percent * duty2, blue);
      rgb -> set_duty(percent * duty2, blue + 3);
      rgb -> set_duty(percent * duty1, green);
      rgb -> set_duty(percent * duty1, green + 3);

      sleep_ms(100);
    }
  }
```

```
}
```

Listing 7: Color changing Function

With all of these areas combined in main, a system that would display the different statuses of the chip were able to be displayed as well as given visual queues that would help a user understand what values that are being displayed mean.

3 Tests

This project was completed incrementally. The first and most important step was receiving the data from each of the functions that would communicate the voltage and the temperature. To verify that these were working properly, the functions were read and understood in order to debug. Then the values being calculated were output in Putty to verify that they were within expected parameters. After this step, it was simply a matter of displaying the important data and including visual factors that would aid a user in understanding the number. After a numerical value of data was received, it was broken into parts to be displayed on the seven-segment using the functions included. The lights were then lit and the number of LEDs being lit up were verified manually using a calculator. Finally when testing the RGB color schemes, each light was displayed and dimmed as expected, being tested by visual observation. Finally all these components were put together and due to each incremental test, they all worked in conjunction.

4 Results

The result of this project was a system that would check the status of a chip and see if it was functioning properly. This was first done by measuring the voltage reading on the chip and seeing if it was between 0 and 3.3V. It was measured to be around 3.0V consistently. Next, the temperature was read and was found to be slightly above room temperature. This is expected because the chip was generating a small amount of heat, and it would be illogical for the chip to be below the ambient temperautre. It was all displayed on a seven-segment display and when reading temperature, the user could deicde bewteen Fahrenheit and Celsius. The 16-LEDs also lit up in proportion to the displayed value. The two RGB LEDs had a specific pattern to indicate what value was being displayed in order to have a user understand exactly what value they were reading on the display.

5 Conclusion

This project sucessfully demonstrated a chip-status checker. It was important to read the voltage on a chip and verify that it was between 0 and 3.3V. If it were 0V, this should indicate to a user that the chip is no longer functioning and if it were getting close to the maximum value of 3.3V, this would indicate to the user that it was reaching a critical max. In between these two values would verify proper functioning. The temperature would also be important to know. At certain values that are too hot, a chip could be damaged, so it would be important for a user to know that it is within safe parameters. Finally, it was designed to be user friendly by adding lights to translate the numbers better. For example if the user was not an engineer that had all the important parameters measured, these lights could show how close it was to unsafe values. All in all, this was a very helpful project learning in how to always keep a chip safe by monitoring several values.