ELC 4396 02
Class Report 3

Jordan Cook

November 1, 2021

1 Introduction and Problem Statement

The purpose of this coding project was to create a blinking LED core. This was the first project that included hardware that was programmed with Vivado and software that was written in Vitis. Hardware needed to be written to create an LED core. This was plugged into one of the slots on an FPGA written also in Vivado. From there, code was written to change the rate at which four of the LED's on the Nexys 4 DDR board would blink.

Link to my Vivado Project: https://github.com/jordanstarr/System-on-Chip/blob/master/FPGA_vanilla.xpr
Link to the Vitis Project: https://github.com/jordanstarr/System-on-Chip
**Video Clip included in submission

2 Method and Approach

2.1 Hardware

The first step, as done in class, was to create the FPGA itself. In order to do this, 16 files were downloaded from the textbook author's website (Dr. Pong Chu), as well as the constraint file necessary to program the Nexys board. Those 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_vanilla.sv
- mcs_top_vanilla.sv

With these files now included in the project, a CPU from the IP catalog was added to the project to ensure that everything would be working with any system updates.

Starting from the very start, the first thing that needed to be done was to create hardware that could be hardcoded with a desired blink rate to verify the functionalty.

The first file was a simple timer module. Since the board has a clock rate of 100 MB (which is too fast to be easily programmable), there needed to be something to create a ”tick” whenever one millisecond had passed. This was implemented with the following code:

Listing 1: Millisecond Timer Module Code

```
‘timescale 1ns / 1ps
//tic should only be up one clock cycle
//counts in 10 nano second period
// assign tick = (count == count value)
// assign n_count = (count == count value)? 0:count  + 1

module timer_ms(
   input logic clk,
   input logic rst,
   output logic tick
   );

   logic [31:0] count, n_count;
   parameter counts_per_ms = 20'b00011000011010100000; // this is 100,000
      in binary

   always_ff @(posedge clk, posedge rst)
      if(rst) begin
         count <= 0;
      end
         else begin
      count <= n_count;
   end

   assign tick = (count == counts_per_ms);
   assign n_count = (count == counts_per_ms)? 0 : count + 1;


endmodule
```

The ”tick” output would then be input into an LED blinker. This new module had an input which would set the period (in milliseconds) of the blinking rate. In the code, it checks to see if half the time has passed of the desired input. If it has not, the LED is on. If it has, the LED turns off. This essentially creates a square wave with the LED toggling back and forth.

Listing 2: LED Blinker Module Code

```
‘timescale 1ns / 1ps
//needs a 16 bit number to set freuency (comes from outside the whole
   thing)
// if count is less than half of num, turn on. if great than half, turn
   off.
// if count == num, restart count
// to divide by 2, cut off the last bit

module LED_blinker(
   input logic clk,
   input logic rst,
   input logic tick,
   input logic [15:0] num,
   output logic led
   );

   logic n_led;
   logic [15:0] ms_count;

   initial begin
   ms_count = 0;
   end

   always_ff @(posedge clk, posedge rst) begin
      if (rst) begin
         led <= 0;
      end
      else begin
         led <= n_led ;
      end
   end

   always @(posedge tick) begin
      ms_count = ms_count + 1;
      if (ms_count >= num) begin
         ms_count = 0;
      end
      else if (ms_count > num[15:1]) begin
         n_led = 0;
      end
      else if (ms_count <= num[15:1]) begin
         n_led = 1;
      end
   end

endmodule
```

From here, an upper module was created to plug the output from the timer to the input of the blinker.

Listing 3: Single LED blinker module

```
‘timescale 1ns / 1ps

module top(
   input clk,
   input rst,
   input [15:0] num,
   output led
   );

   logic tick;
   timer_ms timer(
      .clk(clk),
      .rst(rst),
      .tick(tick)
      );

   LED_blinker blink1(
      .clk(clk),
      .rst(rst),
      .tick(tick),
      .num(num),
      .led(led)
      );

endmodule
```

This module would only turn on one LED at a time. Therefore, another upper module was created with four instantiations a four separate number inputs to get an 4-bit output that correlates to 4 LED's on the board blinking at four different rates.

Listing 4: Set of 4 LED blinkers

```
‘timescale 1ns / 1ps

module Wrapper(
   input clk,
   input reset_n,
   input [15:0] num0,
   input [15:0] num1,
   input [15:0] num2,
   input [15:0] num3,
   output [15:0] led
   );

   top led0(
      .clk(clk),
      .rst(!reset_n),
      .num(num0),
      .led(led[0])
      );

   top led1(
      .clk(clk),
      .rst(!reset_n),
      .num(num1),
      .led(led[1])
      );

   top led2(
      .clk(clk),
      .rst(!reset_n),
      .num(num2),
      .led(led[2])
      );
   top led3(
      .clk(clk),
      .rst(!reset_n),
      .num(num3),
      .led(led[3])
      );
endmodule
```

With all this created, its wrapping circuit could now be derived. Since there were four values that needed to be input to the LED core, four registers needed to be created. The rest of the wrapper was made similarly to other plug-ins in the FPGA slots. Since the hardware needed four numbers' values but there was only one one write port, there needed to be four different buffer registers that could hold the values that were being written to. It only depended on what the input address which of the four buffer registers the write data would be written to.

Listing 5: LED Core Wrapper

```
‘timescale 1ns / 1ps
// typical in and outs
// need four registers for the num data
// outputs are the led’s


module LED_core(
   input logic clk,
   input logic reset,
   // slot interface
   input logic cs,
   input logic read,
   input logic write, //I think this needs to be changed to a 2 bit value
      because there are 4 options to write to
   input logic [4:0] addr,
   input logic [31:0] wr_data,
   output logic [31:0] rd_data,
   // external port
   output logic [3:0] dout
   );

   logic [15:0] num_reg0, buf_reg0;
   logic [15:0] num_reg1, buf_reg1;
   logic [15:0] num_reg2, buf_reg2;
   logic [15:0] num_reg3, buf_reg3;

   logic enable;

   Wrapper ledblinkie(
      .clk(clk),
      .reset_n(!reset),
      // connecting numbers to registers
      .num0(num_reg0),
      .num1(num_reg1),
      .num2(num_reg2),
      .num3(num_reg3),
      .led(dout)
      );

   always_ff @(posedge clk, posedge reset) begin
      if (reset) begin
         buf_reg0 <= 0;
         buf_reg1 <= 0;
         buf_reg2 <= 0;
         buf_reg3 <= 0;
      end
      else begin
         if (enable && (addr == 0)) begin
         buf_reg0 <= wr_data[15:0];
      end
         if (enable && (addr == 1)) begin
         buf_reg1 <= wr_data[15:0];
      end
         if (enable && (addr == 2)) begin
         buf_reg2 <= wr_data[15:0];
      end
         if (enable && (addr == 3)) begin
         buf_reg3 <= wr_data[15:0];
      end
      end
   end

   assign enable = cs && write;
   assign rd_data = 0;

   assign num0 = buf_reg0;
   assign num1 = buf_reg1;
   assign num2 = buf_reg2;
   assign num3 = buf_reg3;

endmodule
```

This wrapper was plugged into slot 4 of the MMIO module. In addition to this new module, another thing that was changed was an input to the module titled ”myLED”. The GPO module already was utilizing all 16 of the LED’s on the board. I simply changed that slot to only require 12 LED’s and the new Blinker Core would require 4 LED’s. (This is explained further in the top module). The following code shows how the new wrapper was plugged into the MMIO:

Listing 6: LED-core Instantiation

```
LED_core slot_4
   (.clk(clk),
   .reset(reset),
   .cs(cs_array[‘S4_USER]),
   .read(mem_rd_array[‘S4_USER]),
   .write(mem_wr_array[‘S4_USER]),
   .addr(reg_addr_array[‘S4_USER]),
   .rd_data(rd_data_array[‘S4_USER]),
   .wr_data(wr_data_array[‘S4_USER]),
   .dout(myLED)
   );
```

The very last piece of hardware to change was the Top level module. The instantiation of the MMIO module needed to be changed to accomodate the GPO's LED's and the Blinker's LED's. The Blinker was given the top 4 LED's and the GPO was given the rest. The following shows how this was done:

Listing 7: MMIO Instantiation with changes

```
mmio_sys_vanilla #(.N_SW(16),.N_LED(12)) mmio_unit (
   .clk(clk),
   .reset(reset_sys),
   .mmio_cs(fp_mmio_cs),
   .mmio_wr(fp_wr),
   .mmio_rd(fp_rd),
   .mmio_addr(fp_addr),
   .mmio_wr_data(fp_wr_data),
   .mmio_rd_data(fp_rd_data),
   .sw(sw),
   .led(led[11:0]),
   .myLED(led[15:12]),
   .rx(rx),
   .tx(tx)
   );
```

2.2 Software

From this point, an XSA file was exported and opened in the Vitis IDE. After going through the proper sections and creating a platform, certain files from Dr. Chu were needed to be added. 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
- main_vanilla_test.cpp

The main source of inspiration for new code written for this project was the gpio_cores.h and gpio_cores.cpp, especially the GPO functions. The goal with the new files was essentially write to four registers. The first file created was a Blinker.h file. The BlinkerCore class was simply a copy of the GPOCore class with a few simple changes. In the enumeration, 3 more data registers were added. The constructor and destructor were exact copies. Lastly, 4 new functions were created to write data to those four different registers.

```
// Jordan Cook
// November 1

#include "chu_init.h"

class BlinkerCore {
```

```
public:
  enum {
    DATA_REG0 = 0, /**< output data register */
    DATA_REG1 = 1,
    DATA_REG2 = 2,
    DATA_REG3 = 3
    };
  /**
  * constructor.
  *
  */
  BlinkerCore(uint32_t core_base_addr);
  ~BlinkerCore();                    // not used

  /**
  * write a 32-bit word
  * @param data 32-bit data
  *
  */
  void write(uint32_t data);

  void write_reg0(uint32_t data);
  void write_reg1(uint32_t data);
  void write_reg2(uint32_t data);
  void write_reg3(uint32_t data);

private:
uint32_t base_addr;
uint32_t wr_data;      // same as GPO core data reg
};
```

Listing 8: Blinker.h

The next file to create was Blinker.cpp. Once again, the majority of the code written here was a copy (with renaming) of source material from GPOCore.cpp. The four different write_reg functions would simply pass a number in and have it written to a register that could be accessed at that specific address.

```
// Jordan Cook
// November 1, 2021

#include "blinker.h"

BlinkerCore::BlinkerCore(uint32_t core_base_addr) {
  base_addr = core_base_addr;
  wr_data = 0;
}
BlinkerCore::~BlinkerCore() {
}

void BlinkerCore::write_reg0(uint32_t data) {
  wr_data = data;
  io_write(base_addr, DATA_REG0, wr_data);
}

void BlinkerCore::write_reg1(uint32_t data) {
  wr_data = data;
  io_write(base_addr, DATA_REG1, wr_data);
}
```

```

void BlinkerCore::write_reg2(uint32_t data) {
  wr_data = data;
  io_write(base_addr, DATA_REG2, wr_data);
}

void BlinkerCore::write_reg3(uint32_t data) {
  wr_data = data;
  io_write(base_addr, DATA_REG3, wr_data);
}
```

Listing 9: Blinker.cpp

The last step in software was to go into main_vanilla_test.cpp and update the code as necessary. A variable of type BlinkerCore (as created in Blinker.h) was created following the example of a GpoCore.

```
BlinkerCore blinkie(get_slot_addr(BRIDGE_BASE, S4_USER));
```

Next, four separate functions were that would call on the Blinker.cpp file to write a value to each of the four registers. An integer would be passed to the function which corresponds to speed of the blinking.

```
void set_blinker0(BlinkerCore *led_p, int s) {
  led_p->write_reg0(s);
}
void set_blinker1(BlinkerCore *led_p, int s) {
  led_p->write_reg1(s);
}
void set_blinker2(BlinkerCore *led_p, int s) {
  led_p->write_reg2(s);
}
void set_blinker3(BlinkerCore *led_p, int s) {
  led_p->write_reg3(s);
```

Finally, the while(1) loop was updated to work with the new code and the functions that had been written.

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


// stuff I wrote
  set_blinker0(&blinkie, 1000);
  set_blinker1(&blinkie, 500);
  set_blinker2(&blinkie, 250);
  set_blinker3(&blinkie, 125);
}
```

3 Testing

3.1 Hardware Tests

The first test was to see if the hardware had been created properly in order to blink the LED on and off at a desired rate. The modules utilized were the millisecond timer, the LED blinker, the top level blinker, and the top level module which contained 4 separate blinkers. In the inputs of the highest module, instead of connecting the numbers to some other input, they were hard coded. This allowed the board to display four separate LED's which had different rates of blinking lights. With this successful test, the LED blinker could now be put into a wrapping circuit and plugged into the FPGA.

3.2 Software Tests

Since the hardware had been proven to work, there was not significant testing in the software side aside from the finished product. Once the code was completed, different values were input to the functions to see if the code would work with any time that was being input.

4 Results

The result from this project was a Nexys4 DDR board that had been hardware that could take an input from software and affect the rate at which the LED's could blink.

5 Conclusion

In conclusion, this project demonstrated the use of hardware and software in conjunction. It was learned that hardware is ideal when switching or taking input and simply completing a task. Software is ideal to set values to registers and easily change variables (such as the rate that the LED's would switch on and off). Many skills were gained, such as learning how to interact with an FPGA by connecting new hardware into the slots and using code to write to registers that would be accessed by hardware.