EE 371

Lab 4: Building and Working with a Simple Microprocessor

Sean Leisle #1332740
Joe Jimenez #1242579
Keegan Griffee #1432621

The following report and its contents are solely our work.

_____   2/24/2017

_____   2/24/2017

_____   2/24/2017

Table of Contents

I.   Abstract

The goal of this lab was to integrate a processor with the onboard FPGA of the DE1-SoC and have them communicate using both verilog and C. We incorporated a microprocessor that ran compiled C code along with the gate arrays we programmed with verilog. After creating a few basic programs that utilized this interface of microprocessor and gate arrays, we then integrated our scanner subsystem from the previous lab to run commands from the processor. The tools used in this lab included iVerilog, Quartus II IDE, Nios II eclipse IDE, Qsys, the SignalTap II Logic Analyzer, PuTTY client for communication with our board, and the Linux operating system.

II.   Introduction

The following lab report covers our implementation of several C programs that utilize the ARM Cortex-A9 processor that exists on the De1-SoC Cyclone V FPGA board used. In addition, we also used the Nios II soft processor that we designed in Qsys and programmed onto the DE1-SoC. The C programs we wrote exercised console input and output, controlling the DE1-SoC hardware from within programs, and interacting with gate arrays on the FPGA with the ARM and Nios II processors. It also covers the various paths we took in trying to accomplish the goal of getting the ARM microprocessor to communicate with the hardware as well as the Nios II processor. Finally, we discuss how we adjusted our previous implementation of the scanner subsystem from the previous lab to interact with the Nios II processor so that we could control the scanner subsystem from a program we wrote in C. We do not explicitly describe the implementation of this scanner subsystem in this lab report. Refer to the previous lab: Lab 3: Designing an Underwater Digital Scanning System for understanding this system. This report focuses on the process used for integrating the FPGA portion of the scanner system with the microprocessors along with the additional applications wrote in C to run on the DE1-SoC.

III. Process Used for Initializing the FPGA for Running C Applications

Baremetal Approach (Failed):

When we first set out to accomplish integrating a processor to control the FPGA we attempted it with the ARM processor. We attempted to enable the ARM Cortex-A9 processor and program using the 'baremetal' approach. 'Baremetal' refers to programming a device without an operating system supporting it. The purpose of this approach was to use the Altera SoC EDS (Embedded Design Suite) software to target the Cyclone V FPGA to compile the C code we wrote and run it on the DE1-SoC. However, we attempted the tutorials for this process on three different computers and ran into the same roadblock each time when we tried to deploy the compiled code.
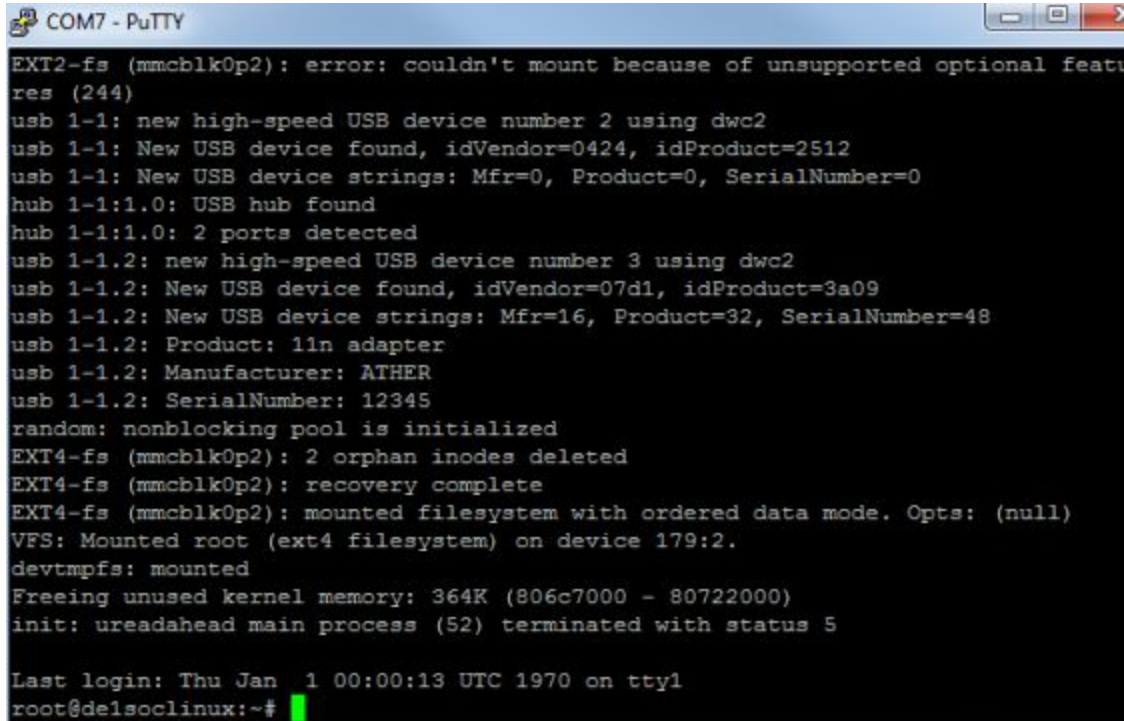
Linux Approach (Used for all non-scanner related C applications):

At this point we decided to switch to another promising approach of booting Linux on the FPGA and running C applications on Linux that would access the FPGA's hardware. In order to do this approach, we had to reconfigure the settings of the MSEL switches of the DE1-SoC board.
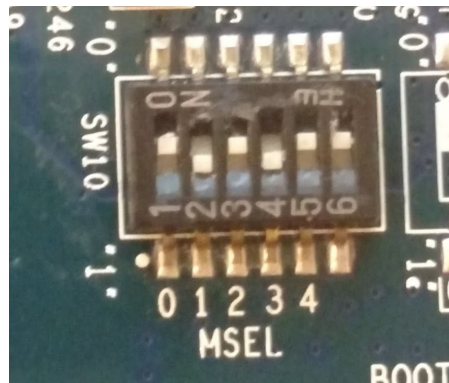
This configuration allows the ARM processor to program the DE1-SoC which is necessary for the Linux image to program the FPGA and enable access to the board's hardware on the DE1-SoC during bootup. We then flashed a compatible version of Linux onto a microSD card. We then inserted the microSD card into the DE1-SoC's microSD card reader so that when it booted it would boot from the microSD and run Linux. In order to access the Linux console on the DE1-SoC and be able to run our application we used a micro USB cable from the UART TO USB port on the DE1-SoC to our PC. This allowed us to run PuTTY on our PC to serial connect to the board and access the Linux command line. At this point we had to configure PuTTY to use the

COM serial line specified by the PC in the Device Manager tools. We then had to change the baud rate to 115200 bits per second, data bits set to 8 bits, stop bits set to 1 bit, and parity and flow control both set to none in device manager for the COM port the DE1-SoC was using our PC and set these same settings in PuTTY. Once these settings were configured we were able to run PuTTY and get the following window:

This verified we had successfully booted into Linux on the DE1-SoC and we could now issue commands through the terminal to run applications on our board using the ARM processor.

At this point we could now write C programs on our computers then cross compile them for ARM Linux (since compiling them on x86 machines results in unusable code on ARM processors) so that the resulting executable applications could be run on the DE1-SoC. We used the arm-linux-gnueabihf- toolchain provided when we downloaded the SoC EDS suite to compile our C programs and create executables to run on the DE1-SoC. In order to get our executable applications onto the DE1-SoC we just had to put them onto the microSD card which had a partition that was reachable by our Linux operating system even when we booted Linux on the DE1-SoC.

The difference when writing C programs for Linux vs. for 'baremetal' was that Linux uses virtual addresses, therefore we couldn't just deference a memory address to a peripheral like the LED's or Hex display from the DE1-SoC documentation. Instead, we had to call a function at the start of our C programs to get the offset caused by Linux and add the baremetal address of the peripheral to this offset, then we could properly

access the register that controlled the peripheral. This was our process for writing all of our C programs that controlled the DE1-SoC hardware.

However, we hit a final roadblock when we tried to program the FPGA with our scanning system and control it from a C program at the same time. We were able to program the FPGA with our control station during the bootup sequence of Linux using a raw binary file (.rbf) from our scanning control system top level design. Once we tried to run any C programs that utilized the hardware along with the FPGA it would throw a segmentation fault. We tried various ways to program the FPGA to get around this but came to the conclusion we needed to instead go back to the Nios II approach for this portion of the lab.

Nios II Approach (Used for scanner system processor integration):

After our many failed attempts at integrating the ARM processor with the scanning system, we quickly learned how to generate a soft processor (specifically the Nios II) from using the Quartus tool Qsys. We manually chose the input/output signals for our processor in order to control our scanner system. These signals included the ready to transfer signal for signalling the control station that the scanner was ready to transmit its data buffer, a start scanning signal to tell the scanner to start scanning, a transfer signal to actually initiate the transfer of the scanner's data buffer and finally a reset signal to reset the state of both the FPGA and the processor to return the system to a default starting state.

IV. Scanning System with the Nios II Processor Integration Verilog Implementation

```verilog
module lab4TopLevel (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, GPIO_0);
    input wire CLOCK_50; // 50MHz System Clock
    input wire [9:0] SW; // Used for demo for flushing
    input wire [3:0] KEY; // Buttons
    inout [31:0] GPIO_0; // AC18 = dataOut, Y17 = clkOut, AD17 = readyToTransmitOut, AK16 = dataIn, AK18 = clkIn, AK19 = readyToTransmitIn
    output wire [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5; // Seven Segment Display
    output wire [9:0] LEDR; // LEDs

    wire [7:0] byteIn;
    wire [1:0] outFromTransfer, stationToScanner;
    wire [3:0] dataBufferScanner;
    wire [7:0] dataBufferTransfer;
    wire [1:0] ps;
    wire [2:0] byteCounter;
    wire [7:0] commandBuffer, shiftReg;
    wire readyToTransfer;
    wire startScanning;
    wire transfer;
    wire clkOut, rst;
    wire [31:0] clkMain;
    parameter whichClock = 23;
    assign rst = ~KEY[0];
    // Initialize clock divider
    assign LEDR[9] = clkOut;
    assign LEDR[8] = rst;
    assign LEDR[7] = stationToScanner[1];
    assign LEDR[6] = stationToScanner[0];
    assign clkOut = /*CLOCK_50; // CHANGE WHEN IT IS FINISHED!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! --->*/ clkMain[whichClock];
    clock_divider cdiv (CLOCK_50, clkMain);
    // DEBUG to see the signals actively working from the processor
    assign LEDR[3] = readyToTransfer;
    assign LEDR[4] = startScanning;
    assign stationToScanner[1] = outFromTransfer[1] & SW[7]; // Demo for flushing the buffer
    assign stationToScanner[0] = outFromTransfer[0] | startScanning;
    assign readyToTransfer = GPIO_0[0]; // Looks for the ready to transfer signal
    // Initialize internals
    seg7 h1 (dataBufferScanner, HEX1);
    seg7 h2 (dataBufferTransfer[3:0], HEX2);
    seg7 h0 (4'b0, HEX0);
    seg7 h3 (4'b0, HEX3);
    seg7 h4 (commandBuffer[3:0], HEX4);
    seg7 h5 (shiftReg[3:0], HEX5);

    // The Nios II processor with integration to receive the readyToTransfer signal
    // and transmit the startScanning signal, or start the transfer with the transfer signal
    // Also has a reset signal to reset the state of the processor
    nios_systemv3 u0 (
        .clk_clk               (CLOCK_50),      //            clk.clk
        .readytotransfer_export (readyToTransfer), // readytotransfer.export
        .reset_reset_n         (~rst),          //          reset.reset_n
        .startscanning_export  (startScanning), //   startscanning.export
        .transfer_export       (transfer)       //        transfer.export
    );

    scanner localScanner (clkOut, rst, GPIO_0[6], stationToScanner, GPIO_0[1], GPIO_0[0], dataBufferScanner); // CHANGE 5 to 1 and 4 to 0 and 2 to 6
    // Integration with the processors signals
    transferCenterNew localTransfer (byteCounter, GPIO_0[5], rst, GPIO_0[4], transfer, GPIO_0[2], outFromTransfer, dataBufferTransfer, commandBuffer, shiftReg);

endmodule
```

The top level implementation of the underwater digital scanning system. This handles assigning I/O for the FPGA and connecting our local scanner to the local control station. The outputs include a seven-segment display for the buffer level of the local scanner system on our board as well as GPIO pins assigned as outputs to send the necessary data and clock signals to a remote scanner from our control station. For inputs we used our newly integrated processor to control the system. The Nios II now can send the start scanning signal from the control station to a scanner as well as a transfer signal in order to transmit the data buffer from the scanner to the control station. It can also tell when the scanner is ready to transfer its data with the readyToTransfer signal or can receive a reset signal to set the system back to its starting state . We also have GPIO pins assigned as inputs to receive a clock and data from a remote scanner system not on our board. Finally we have the hex display as output for the various buffer levels in the system.

V.   Scanning System Control Application

```c
/*
 *      Author: Keegan, Joe, Sean
 *      scannercontrol.c - Used to control the
 *      scanning system by command line utilizing
 *      a processor
 */

#include "sys/alt_stdio.h"
#include "sys/unistd.h"

#define readyToTransfer (volatile char *) 0x0009020
#define transfer (volatile char *) 0x0009000
#define startScanning (volatile char *) 0x0009010

int main() {
    alt_putstr("Scanner Control Ready \n");
    char input = 'i'; // Idle/Don't print further output
    char nextState = 'i';

    while (1) {
        // Always set scanning back to zero to prevent repeat scanning
        *startScanning = 0;
        // Always set transfer back to zero to prevent repeat transfering
        *transfer = 0;

        // Check if the scanner is ready to transfer it's data buffer
        if((*readyToTransfer &= 0x1) == 0x1) {
            alt_putstr("\n ready to transfer!\n");
        }

        // Get the command from the user
        input = alt_getchar();

        // Use the command to determine the next state
        nextState = input;

        // Tell scanner to begin scanning
        if(nextState == 's') {
            // Prevent further scanning
            nextState = 'i';
            *startScanning = 1;
            // Reset the ready to transfer signal
            *readyToTransfer = 0;
            alt_putstr("\n Start scanning...\n");
        }

        // Tell the scanner to transfer it's data
        if(nextState == 't') {
            // Return to idle
            nextState = 'i';
            // Transfer the data
            *transfer = 1;
            alt_putstr("\n Transfering data...\n");
        }
    }
    return 0;
}
```

   C program used to control our scanner from console input. Can start the scanner, or tell the scanner to transmit it's data and informs us when the scanner is ready to transfer its data.

VI. The Nios II Processor Qsys Schematic

ab4\nios_systemv3.qsys)

Use	Connections	Name	Description	Export	Clock	Base	End
☑		⊟ **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_qsys_0**	Nios II Processor				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset_n	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]		
		d_irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ
		jtag_debug_module_r...	Reset Output	*Double-click to export*	[clk]		
		jtag_debug_module	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x8800	0x8fff
		custom_instruction_m...	Custom Instruction Master	*Double-click to export*			
☑		⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)				
		clk1	Clock Input	*Double-click to export*	**clk_0**		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x4000	0x7fff
		reset1	Reset Input	*Double-click to export*	[clk1]		
☑		⊟ **readyToTransfer**	PIO (Parallel I/O)				
		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]	0x9020	0x902f
		external_connection	Conduit	**readytotransfer**			
☑		⊟ **jtag_uart_0**	JTAG UART				
		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]	0x9030	0x9037
		irq	Interrupt Sender	*Double-click to export*	[clk]		
☑		⊟ **transfer**	PIO (Parallel I/O)				
		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]	0x9000	0x900f
		external_connection	Conduit	**transfer**			
☑		⊟ **startScanning**	PIO (Parallel I/O)				
		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]	0x9010	0x901f
		external_connection	Conduit	**startscanning**			

Parameters

nios_systemv3 > clk_0

Clock Source
clock_source

▼ Parameters
Clock frequency: 50000000
☑ Clock frequency is known
Reset synchronous edges: None ▾

Messages

Type	Path	Message
⊟ ⓘ	2 Info Messages	
ⓘ	**nios_systemv3.nios2_qsys_0**	Please note that for early evaluation, preview versions of new Nios II Gen2 Processors are available with this release.
ⓘ	**nios_systemv3.transfer**	PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.

Processor Description:

-Operates on a 50 MHz clock

-Has 16384 bytes of memory

I/O Signals Used By the Processor:

startScanning - Output signal that activates the scanner to begin actively scanning

readyToTransfer - Input signal that tells us that the scanner is ready to transfer data

transfer - Output signal that tells the scanner to begin transferring its data

reset - Input signal that will reset the state of the processor

VII. Scanning System with Processor Integration SignalTap Analysis
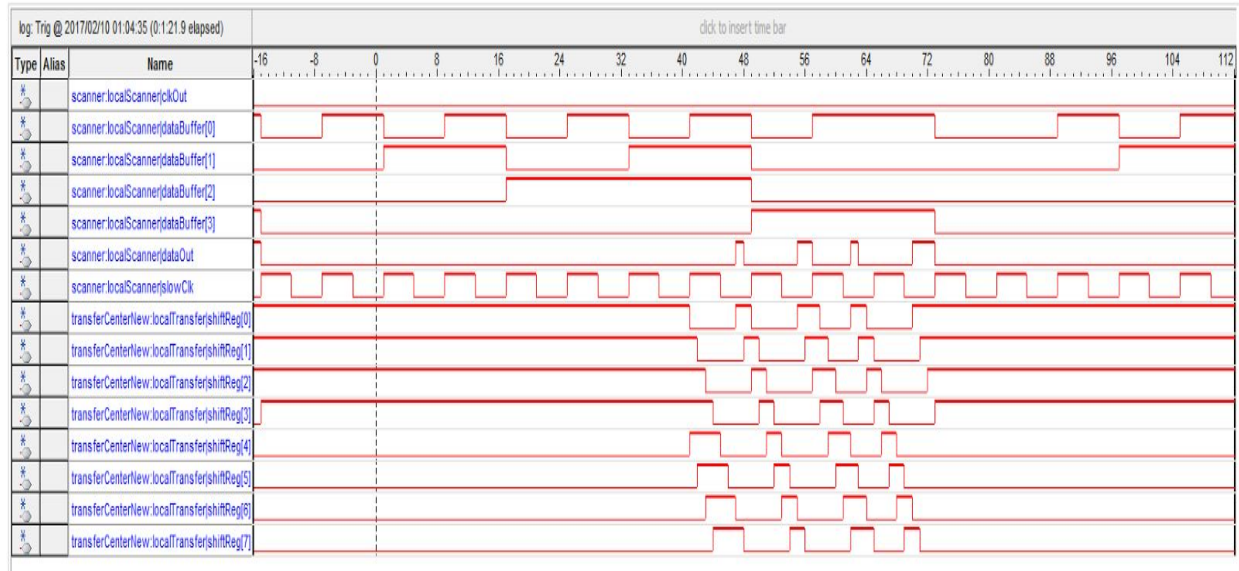
Above is a capture of the SignalTap Analysis of the working control system. The following are the controls and outputs for the system.

LED8 - Reset indicator
LED9 - Internal clock of scanner
Key 0 – Reset
AC18 - Data Out
Y17 - Clock Out
AD17 - Ready to transmit out
AK16 - Data In
AK18 - Clock In
AK19 - Ready to transmit in

Now Controlled by Processor
Activate scanner signal
Toggles the ability to flush the buffer (demo only)
Ready to Transfer Out
Begin transfer

VIII. C Applications Developed for the FPGA
A. Count Binary

Program that counts up by one repeatedly on both the FPGA's LEDs and on the 7-segment hex displays until it rolls over back to zero. The LED's represent the current count in binary and the hex displays the current count in decimal.
Hex Drivers

```
1   #ifndef _HEX_HEADER_
2   #define _HEX_HEADER_
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <unistd.h>
7   #include <fcntl.h>
8   #include <sys/mman.h>
9   #include <inttypes.h>
10
11  #define HW_REGS_BASE ( 0xff200000 )
12  #define HW_REGS_SPAN ( 0x00200000 )
13  #define HW_REGS_MASK ( HW_REGS_SPAN - 1 )
14  #define HEX_BASE 0x20
15  #define HEX_BASE_TWO 0x30
16
17  // Struct that represents one of the 7 segment hex displays
18  struct hex {
19          uint32_t *address;
20          uint8_t value;
21          uint8_t index;
22  };
23
24  typedef struct hex *hex_t;
25
26  // Create new hex with given index and virtualBase address
27  hex_t allocateHex(uint8_t index, void *virtualBase);
28
29  // Set value of the hex
30  void setHexValue(hex_t hex, uint8_t value);
31
32  // Return value of the hex
33  uint8_t getHexValue(hex_t hex);
34
35  #endif // _HEX_HEADER_
```

```c
#include "hex.h"

hex_t allocateHex(uint8_t index, void *virtualBase) {
        hex_t hex = (hex_t)malloc(sizeof(struct hex));
        if (hex == NULL) {
                return NULL;
        }
        hex->index = index;
        // Index < 4 has different register than index >= 4, gets the virtual address of register
        if (index < 4) {
                uint32_t *hexBase = (unsigned int *)(virtualBase + (( HEX_BASE ) & ( HW_REGS_MASK ) ));
                hex->address = hexBase;
        } else {
                uint32_t *hexBase = (unsigned int *)(virtualBase + (( HEX_BASE_TWO ) & ( HW_REGS_MASK ) ));
                hex->address = hexBase;
        }
        setHexValue(hex, 0);
        return hex;
}

void setHexValue(hex_t hex, uint8_t value) {
        hex->value = value;
        uint32_t bits = 0;
        // Choses correct bit pattern to make the number on the led display
        switch (value) {
                case 0: bits = 0b00111111; break;
                case 1: bits = 0b00000110; break;
                case 2: bits = 0b01011011; break;
                case 3: bits = 0b01001111; break;
                case 4: bits = 0b01100110; break;
                case 5: bits = 0b01101101; break;
                case 6: bits = 0b01111101; break;
                case 7: bits = 0b00000111; break;
                case 8: bits = 0b01111111; break;
                case 9: bits = 0b01101111; break;
                default: bits = 0b00111111; break;
        }
        // Calculate offset in the register
        int offset = (hex->index) % 4;
        // Shift bits to correct offset
        bits = bits << (offset * 8);
        // Clear the old value at that offset
        uint32_t clearer = 0x000000FF;
        clearer = clearer << (offset * 8);
        clearer = ~clearer;
        *(hex->address) = *(hex->address) & clearer;
        // Add in the bits of the value
        *(hex->address) = *(hex->address) | bits;
}

uint8_t getHexValue(hex_t hex) {
        return hex->value;
}
```

Actual Counting program

```c
1    #include "hex.h"
2
3    #define LED_BASE 0x0
4    #define DELAY 10000000
5    #define MAX_NUM 421
6
7    int main(int argc, char **argv) {
8            void *virtual_base;
9            uint32_t *leds;
10           int fd;
11
12           // Open /dev/mem
13           if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
14                   printf( "ERROR: could not open \"/dev/mem\"...\n" );
15                   return( 1 );
16           }
17
18           // Get virtual base address that maps to physical
19           virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );
20           if( virtual_base == MAP_FAILED ) {
21                   printf( "ERROR: mmap() failed...\n" );
22                   close( fd );
23                   return(1);
24           }
25
26           // Get address of the LED register
27           leds = (unsigned int *)(virtual_base + (( LED_BASE ) & ( HW_REGS_MASK ) ));
28
29           // Initialize hexes
30           hex_t hexes[6];
31           int i;
32           for (i = 0; i < 6; i++) {
33                   hexes[i] = allocateHex(i, virtual_base);
34                   if (hexes[i] == NULL) {
35                           return 1;
36                   }
37           }
```

```
38
39          int number = 0;
40          int counter = 0;
41
42          // Keeps counting in binary on the LEDs, and in decimal on the hex displays
43          while (number < MAX_NUM) {
44                  counter = (counter + 1) % DELAY;
45                  if (counter == 0) {
46                          number = (number + 1) % MAX_NUM;
47                          setHexValue(hexes[0], number % 10);
48                          setHexValue(hexes[1], (number / 10) % 10);
49                          setHexValue(hexes[2], (number / 100) % 10);
50                          *leds = number;
51                  }
52          }
53
54          if( munmap( virtual_base, HW_REGS_SPAN ) != 0 ) {
55                  printf( "ERROR: munmap() failed...\n" );
56                  close( fd );
57                  return( 1 );
58          }
59
60          // Free the hex structs
61          for (i = 0; i < 6; i++) {
62                  free(hexes[i]);
63          }
64
65          close( fd );
66          return 0;
67  }
```

B. Lights and Switches

Program that controls the FPGA's LEDs based on the position of the switches on
the FPGA. When a switch is flipped up then the LED corresponding to that switch
becomes lit and when the switch is flipped down the LED corresponding to that
switch turns off. However, if any of the FPGA's keys are pressed the system
sees this as a reset and instead all LED's are turned off until the key is released.

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

#define HW_REGS_BASE ( 0xff200000 )
#define HW_REGS_SPAN ( 0x00200000 )
#define HW_REGS_MASK ( HW_REGS_SPAN - 1 )
#define LED_BASE 0x0
#define SWITCH_BASE 0x40
#define KEY_BASE 0x50

int main(void) {
        /* Declare volatile pointers to I/O registers
        (volatile means that the locations will not be cached, * even in registers)
        */
        volatile unsigned int * LED_ptr = NULL;  // red LED address
        volatile unsigned int * SW_switch_ptr = NULL;  // SW slider switch address // pushbutton KEY address
        volatile unsigned int * KEY_ptr = NULL;  // Pushbutton Key Address
        void *virtual_base;
        int SW_value;
        int fd;
        volatile int delay_count;

        // Open /dev/mem
        if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
                printf( "ERROR: could not open \"/dev/mem\"...\n" );
                return( 1 );
        }

    // get virtual addr that maps to physical
        virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );
        if( virtual_base == MAP_FAILED ) {
                printf( "ERROR: mmap() failed...\n" );
                close( fd );
                return(1);
        }
```

```
39          // Get the address that maps to the LEDs and other peripherals
40              LED_ptr = (unsigned int *)(virtual_base + (( LED_BASE ) & ( HW_REGS_MASK ) ));
41              SW_switch_ptr = (unsigned int *)(virtual_base + (( SWITCH_BASE ) & ( HW_REGS_MASK ) ));
42              KEY_ptr = (unsigned int *)(virtual_base + (( KEY_BASE ) & ( HW_REGS_MASK ) ));
43
44              while (1) {
45                      // Get value of switches
46                      SW_value = *(SW_switch_ptr);
47                      // Reset on key press
48                      if (*KEY_ptr != 0) {
49                              *(LED_ptr) = 0;
50                              while (*KEY_ptr != 0);
51                      }
52                      // Set LEDS to switch values
53                      *(LED_ptr) = SW_value;
54              }
55
56          if( munmap( virtual_base, HW_REGS_SPAN ) != 0 ) {
57                  printf( "ERROR: munmap() failed...\n" );
58                  close( fd );
59                  return( 1 );
60          }
61          close( fd );
62          return 0;
63      }
```

C. Basic Hello World (Basic Console Output)

This program simply prints 'Hello World' to the console.

```
#include <stdio.h>

int main(int argc, char **argv) {
        printf("Hello, World!\n");
        return 0;
}
```

D. Advanced Hello World (Console Input/Output)

This program first prints 'Hello World' to the console, then asks the user to input a 'G'. Once the user inputs a 'G' the FPGA hardware is activated so that when a switch is flipped up (from SW1 to SW9) the corresponding LED becomes lit up and doesn't light up if the switch is down. However, if SW0 is flipped up LED0 will

16

turn on and now switches that are in the downright position will cause their corresponding LED to light up and upright switches cause their LED to turn off.

```c
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <fcntl.h>
4   #include <sys/mman.h>
5
6   #define HW_REGS_BASE ( 0xff200000 )
7   #define HW_REGS_SPAN ( 0x00200000 )
8   #define HW_REGS_MASK ( HW_REGS_SPAN - 1 )
9   #define LED_BASE 0x0
10  #define SWITCH_BASE 0x40
11
12  int main(void) {
13          /* Declare volatile pointers to I/O registers
14          (volatile means that the locations will not be cached, * even in registers)
15          */
16          volatile unsigned int * LED_ptr = NULL;  // red LED address
17          volatile unsigned int * SW_switch_ptr = NULL;  // SW slider switch address // pushbutton KEY address
18          void *virtual_base;
19          int SW_value;
20          int fd;
21          volatile int delay_count;
22
23          // Open /dev/mem
24          if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
25                  printf( "ERROR: could not open \"/dev/mem\"...\n" );
26                  return( 1 );
27          }
28
29      // get virtual addr that maps to physical
30          virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );
31          if( virtual_base == MAP_FAILED ) {
32                  printf( "ERROR: mmap() failed...\n" );
33                  close( fd );
34                  return(1);
35          }
```

```
36
37        // Get the address that maps to the LEDs and other peripherals
38            LED_ptr = (unsigned int *)(virtual_base + (( LED_BASE ) & ( HW_REGS_MASK ) ));
39            SW_switch_ptr = (unsigned int *)(virtual_base + (( SWITCH_BASE ) & ( HW_REGS_MASK ) ));
40
41            // Scan for input, G to continue
42            char input;
43            printf("hello world...\n");
44            printf("To continue enter \'G\': ");
45            scanf("%c", &input);
46            getchar();
47            while (input != 'G') {
48                    printf("To continue enter \'G\': ");
49                    scanf("%c", &input);
50                    getchar();
51            }
52
53            while (1) {
54                    // Get value of switches
55                    SW_value = *(SW_switch_ptr);
56                    // Invert if first switch is on
57                    if (SW_value % 2 == 1) {
58                            SW_value = ~SW_value + 1;
59                    }
60                    // Set LEDS to the value
61                    *(LED_ptr) = SW_value;
62            }
63
64            if( munmap( virtual_base, HW_REGS_SPAN ) != 0 ) {
65                    printf( "ERROR: munmap() failed...\n" );
66                    close( fd );
67                    return( 1 );
68            }
69            close( fd );
70            return 0;
71    }
```

IX.   Additional Subjects of Importance

None came to mind when writing and explaining the process we used in setting up the FPGA for the various programs and integration with the scanner system. One thing we unanimously agreed upon was a better set of tutorials need to be done in order to proceed with the ARM method of programming the FPGA. We almost got it to entirely work with ARM, however we ran into an eventual roadblock we couldn't overcome and made a last minute switch to using the Nios II cpu instead.

X.   Anything the Lab Specification Didn't Think of

We did not think of anything additional for this lab report that was not specified.

XI.   Discussion of Problems During Development

We ran into many problems during this lab involving the ARM processor approach. The first problem we ran into was when we attempted to program the DE1-SoC using the 'baremetal' approach using the SoC FPGA Embedded Design Suite (SoC EDS). Whenever we tried to deploy the code to our board and debug the application, the board was unrecognized by the development environment and we couldn't work around this problem.

We instead decided to utilize the other possible approach which was booting Linux and instead of writing baremetal C programs, we wrote programs that would go through the Linux operating system to get to the registers and peripherals we needed in order to access and control the DE1-SoC's hardware's behavior. This approach worked smoothly for our basic applications, however integrating our scanner control system onto the board with the processor interacting with it often halted our entire Linux operating system. After searching for another way to interface with the ARM processor, we found a tutorial on how to build a bridge between the ARM processor and the FPGA using Qsys. This approach also failed, however, so we decided to try a completely different approach.

We then decided to set up the Nios II soft processor on the FPGA and connect it to our scanner program. We managed to succeed with enabling the processor and sending control signals from a console to the processor which then controlled our scanner. However, we still found the processor was glitchy and resulted in some faulty behavior.

Due to time constraints and the amount of effort we put into trying to integrate the ARM processor we were unable to iron out all of the bugs the Nios II processor introduced which were present in our demo when certain sequences of commands to the console were issued. This also seemed to exacerbate the issues we had from our previous lab design and some of the failed state transitions.

In hindsight, we all agree that it would have been better to have just started with the Nios II approach and spent more time debugging our original scanner system when integrated with the processor. However, the insight we gained from learning new tools such as the SoC EDS, cross-compiling our C code for running on ARM, learning how to boot Linux onto the DE1-SoC while deciphering the DE1-SoC manual, using Qsys to design hardware bridges, and the process for writing Linux applications to control underlying hardware were invaluable and taught us more about general embedded system design than this lab originally entailed.

XII.   Summary and Conclusion

The product of this lab was a scanner control system that is now integrated with a Nios II soft microprocessor. We also developed various C programs that successfully ran on the DE1-SoC on the ARM processor and Linux operating system and ranged from printing simple console output, interacting with the user, determining the behavior of the hardware, and reacting to peripherals on the DE1-SoC. We also got to experience a variety of methods for programming and accessing the ARM processor on the DE1-SoC. This even involved learning how to boot and run Linux on the DE1-SoC. We learned both the 'baremetal' approach using no operating system and the Linux operating system approach. This taught us how to program hardware even with the abstraction of an operating system. This taught us valuable lessons in embedded design and perseverance when one route doesn't work. Overall, we understand even

more about the hardware on the FPGA and the process behind how software runs on it and affects its behavior.