State University of New York at New Paltz

Department of Computer and Electrical Engineering

EGC493-01
System-On-Chip

Group Members	Department
Mike Facelle	CE
Nick Falco	CE

Dr. Rajeev Narayanan

December 16, 2013

1 Objective

The objective of this project is to design a multiprocessor system on the Altera DE2-115 FPGA board. The project uses two processors with a shared memory resource to simulate a two conveyor belt system. The system allows for allows for variable speed control, conveyor belt direction control, and an emergency cutoff switch. Each of the two conveyor belts is controlled by one of two NIOs-II processors. Programs are installed into the sytem memory using the Eclipse software where either processor is selected. A third program is then uploaded and run on a third processor which allows the two processors to work together as a single system.

2 Introduction

For the design of this system we utilized several different software tools. These include the QSYS System Builder, Quartus II IDE, and Eclipse IDE. QSYS allowed for system components to be instanciated and interconnected. After generating a QSYS system file, the Quartus II IDE is used to assign the boards peripherals to the system using verilog and to program the system onto the FPGA board. Once the board has been programmed, the Eclipse IDE is used to write hardware specific C programs that control the behavior of the system.

3 Design Procedure

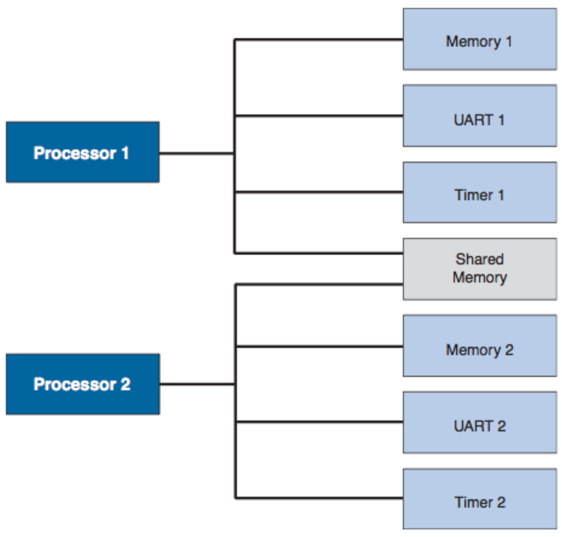


Figure 1: The layout of two processors with shared memory (from [2])

Figure 1 contains the layout of a simple multiprocessor system with shared memory. Each processor has its own JTAG UART, timer, and memory device, but includes a bridge to allow it to communicate with an external main memory.

3.1 Qsys

In order to design a multiprocessor system with shared memory, first the individual processor systems must be created. This was done by combining a Nios II processor with On-Chip memory, a JTAG UART device, a mutex, a timer, any input/output peripherals required (in this case, an 8-bit LED output and 4-bit switch input), Avalon Memory-Mapped Pipeline Bridges for shared memory input/output, and a SystemID peripheral to set the specific ID of each individual CPU, allowing access through the mutex.

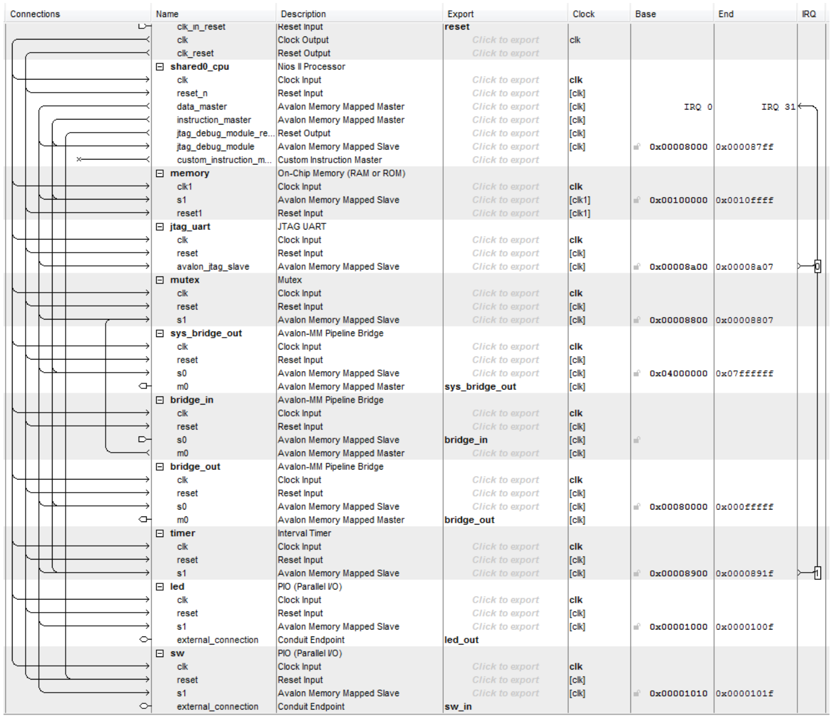
Using the command "Assign Base Addresses" will not work, because it will set addresses that cannot be reached by the processor. It is important to note that the addresses for reset and exception vectors used are "absolute," and are at the base address of the master output bridge (giving access to the main, shared, memory). These memory locations can be found in Table 1. Addresses were manually assigned based on the addresses used in [2], but with additional addresses used for individual memory and input/output peripherals. Figure 2 contains the layout of an individual Qsys shared-memory processor (shared_zero), with the memory locations used in this project, and Table 2 contains the memory addresses for each component. It should be noted that sys_bridge_out has an address with of 26, while bridge_out has a width of 19 and bridge_in has a width of 16. Bridge_in does not receive memory locations as it is an input.

Table 1: Reset and Exception Vector Memory Locations

	main cpu	**shared0_cpu**	**shared1_cpu**
Memory	main_memory.s1	Absolute	Absolute
Reset Vector Offset	0x0000a000	0x04050000	0x04050000
Reset Vector	0x0005a000	0x04050000	0x04050000
Exception Vector Offset	0x0000a020	0x04050020	0x04050020
Exception Vector	0x0005a020	0x04050020	0x04050020

Table 2: Memory Locations for shared_zero.qsys System

Component	start	end
shared0_cpu	0x00008000	0x000087ff
memory	0x00100000	0x0010ffff
jtag_uart	0x0005a000	0x04050000
mutex	0x00008800	0x00008807
sys_bridge_out	0x04000000	0x07ffffff
bridge_in	-	-
bridge_out	0x00080000	0x000fffff
timer	0x00008900	0x0000891f
led	0x00001000	0x0000100f
sw	0x00001010	0x0000101f



Connections	Name	Description	Export	Clock	Base	End	IRQ
	clk_in_reset	Reset Input	**reset**				
	clk	Clock Output	Click to export	clk			
	clk_reset	Reset Output	Click to export				
	⊟ **shared0_cpu**	Nios II Processor					
	clk	Clock Input	Click to export	**clk**			
	reset_n	Reset Input	Click to export	[clk]			
	data_master	Avalon Memory Mapped Master	Click to export	[clk]	IRQ 0	IRQ 31	
	instruction_master	Avalon Memory Mapped Master	Click to export	[clk]			
	jtag_debug_module_re...	Reset Output	Click to export	[clk]			
	jtag_debug_module	Avalon Memory Mapped Slave	Click to export	[clk]	0x00008000	0x000087ff	
	custom_instruction_m...	Custom Instruction Master	Click to export				
	⊟ **memory**	On-Chip Memory (RAM or ROM)					
	clk1	Clock Input	Click to export	**clk**			
	s1	Avalon Memory Mapped Slave	Click to export	[clk1]	0x00100000	0x0010ffff	
	reset1	Reset Input	Click to export	[clk1]			
	⊟ **jtag_uart**	JTAG UART					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	Click to export	[clk]	0x00008a00	0x00008a07	0
	⊟ **mutex**	Mutex					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Click to export	[clk]	0x00008800	0x00008807	
	⊟ **sys_bridge_out**	Avalon-MM Pipeline Bridge					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s0	Avalon Memory Mapped Slave	Click to export	[clk]	0x04000000	0x07ffffff	
	m0	Avalon Memory Mapped Master	**sys_bridge_out**	[clk]			
	⊟ **bridge_in**	Avalon-MM Pipeline Bridge					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s0	Avalon Memory Mapped Slave	**bridge_in**	[clk]			
	m0	Avalon Memory Mapped Master	Click to export	[clk]			
	⊟ **bridge_out**	Avalon-MM Pipeline Bridge					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s0	Avalon Memory Mapped Slave	Click to export	[clk]	0x00080000	0x000fffff	
	m0	Avalon Memory Mapped Master	**bridge_out**	[clk]			
	⊟ **timer**	Interval Timer					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Click to export	[clk]	0x00008900	0x0000891f	1
	⊟ **led**	PIO (Parallel I/O)					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Click to export	[clk]	0x00001000	0x0000100f	
	external_connection	Conduit Endpoint	**led_out**				
	⊟ **sw**	PIO (Parallel I/O)					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Click to export	[clk]	0x00001010	0x0000101f	
	external_connection	Conduit Endpoint	**sw_in**				

Figure 2: Screenshot of shared_zero.qsys, label for clock module cut off at top

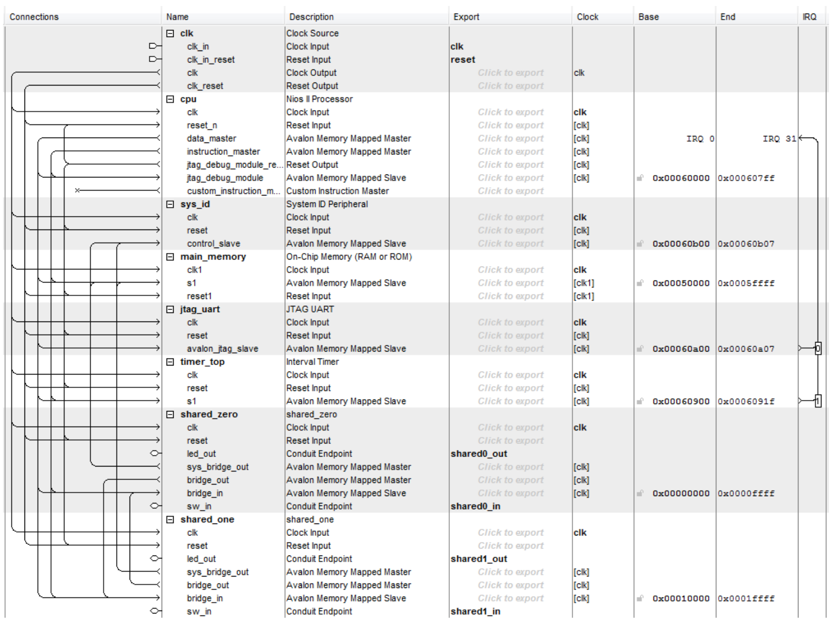
After assembling the individual processor systems needed for this project, the top-level system was created. Included in it are the same components for individual processors, such as timer, Nios II processor, JTAG UART device, and On-chip memory, as well as the individual processor systems created earlier: shared_zero and shared_one. Addresses were assigned using "Assign Base Addresses" again, but then manually input for each individual processor system. Since each system has a range of 0x00000-0x0ffff, each system was offset by

Table 3: Memory Locations for project.qsys System (top-level)

Component	start	end
cpu	0x00060000	0x000607ff
sys_id	0x00060b00	0x00060b07
main_memory	0x00050000	0x0005ffff
jtag_uart	0x00060a00	0x00060a07
timer_top	0x00060900	0x0006091f
shared_zero	0x00000000	0x0000ffff
shared_one	0x00010000	0x0001ffff

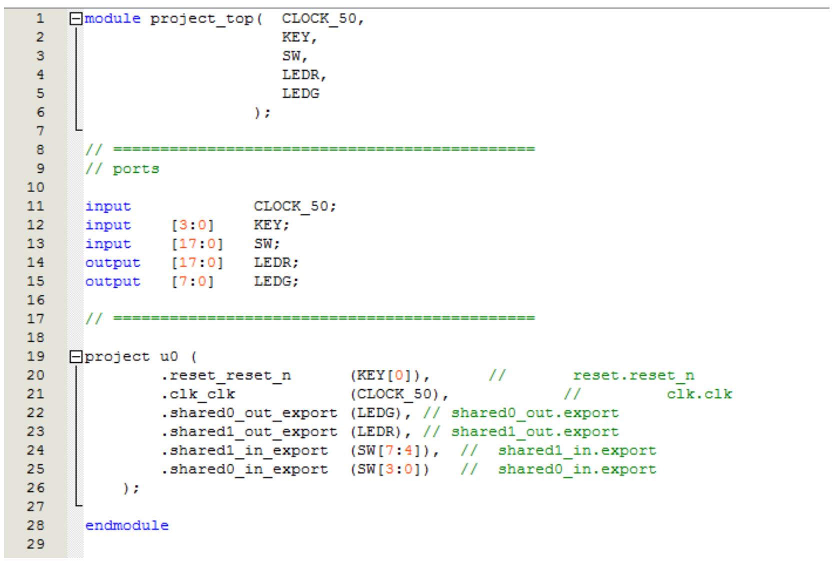
0x10000. Figure 3 contains the top-level Qsys system with addresses assigned and Table 3 contains the memory locations for each component.

In order to run the system, each component was generated and the top-level "HDL Example" was copied and put into the main Verilog module.



Connections	Name	Description	Export	Clock	Base	End	IRQ
	⊟ **clk**	Clock Source					
	clk_in	Clock Input	**clk**				
	clk_in_reset	Reset Input	**reset**				
	clk	Clock Output	Click to export	clk			
	clk_reset	Reset Output	Click to export				
	⊟ **cpu**	Nios II Processor					
	clk	Clock Input	Click to export	**clk**			
	reset_n	Reset Input	Click to export	[clk]			
	data_master	Avalon Memory Mapped Master	Click to export	[clk]	IRQ 0	IRQ 31	
	instruction_master	Avalon Memory Mapped Master	Click to export	[clk]			
	jtag_debug_module_re...	Reset Output	Click to export	[clk]			
	jtag_debug_module	Avalon Memory Mapped Slave	Click to export	[clk]	0x00060000	0x000607ff	
	custom_instruction_m...	Custom Instruction Master	Click to export				
	⊟ **sys_id**	System ID Peripheral					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	control_slave	Avalon Memory Mapped Slave	Click to export	[clk]	0x00060b00	0x00060b07	
	⊟ **main_memory**	On-Chip Memory (RAM or ROM)					
	clk1	Clock Input	Click to export	**clk**			
	s1	Avalon Memory Mapped Slave	Click to export	[clk1]	0x00050000	0x0005ffff	
	reset1	Reset Input	Click to export	[clk1]			
	⊟ **jtag_uart**	JTAG UART					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	Click to export	[clk]	0x00060a00	0x00060a07	0
	⊟ **timer_top**	Interval Timer					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Click to export	[clk]	0x00060900	0x0006091f	1
	⊟ **shared_zero**	shared_zero					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export				
	led_out	Conduit Endpoint	**shared0_out**				
	sys_bridge_out	Avalon Memory Mapped Master	Click to export	[clk]			
	bridge_out	Avalon Memory Mapped Master	Click to export	[clk]			
	bridge_in	Avalon Memory Mapped Slave	Click to export	[clk]	0x00000000	0x0000ffff	
	sw_in	Conduit Endpoint	**shared0_in**				
	⊟ **shared_one**	shared_one					
	clk	Clock Input	Click to export	**clk**			
	reset	Reset Input	Click to export				
	led_out	Conduit Endpoint	**shared1_out**				
	sys_bridge_out	Avalon Memory Mapped Master	Click to export	[clk]			
	bridge_out	Avalon Memory Mapped Master	Click to export	[clk]			
	bridge_in	Avalon Memory Mapped Slave	Click to export	[clk]	0x00010000	0x0001ffff	
	sw_in	Conduit Endpoint	**shared1_in**				

Figure 3: Screenshot of project.qsys, the top level Qsys system



```verilog
module project_top(  CLOCK_50,
                     KEY,
                     SW,
                     LEDR,
                     LEDG
                  );

// ============================================
// ports

input               CLOCK_50;
input      [3:0]    KEY;
input      [17:0]   SW;
output     [17:0]   LEDR;
output     [7:0]    LEDG;

// ============================================

project u0 (
        .reset_reset_n      (KEY[0]),      //       reset.reset_n
        .clk_clk            (CLOCK_50),            //         clk.clk
        .shared0_out_export (LEDG), // shared0_out.export
        .shared1_out_export (LEDR), // shared1_out.export
        .shared1_in_export  (SW[7:4]),  //  shared1_in.export
        .shared0_in_export  (SW[3:0])   //  shared0_in.export
    );

endmodule
```

Figure 4: Screenshot of project_top.v, the top level Verilog file for this system

3.2 Quartus II

Figure 4 contains the Verilog code for the top-level module, which instantiates the Qsys system talked about earlier. It requires the external signals CLOCK_50, KEY, SW, LEDG, and LEDR. KEY[0] is used at the reset input to the Qsys system (which seems to be a requirement of all systems developed in this manner), and CLOCK_50 is used as the global CLK input. The switches (SW) are used for speed, direction, and emergency signal control for each set of LEDs.

The Quartus II project is then compiled and programmed onto the DE2-115 board. Following this, Eclipse is started to allow each processor to be programmed.

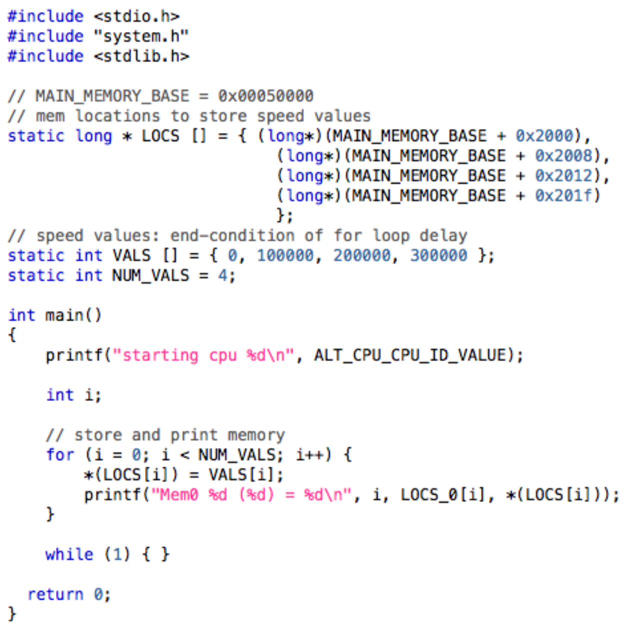
3.3 Nios II Programming with Eclipse

Since this is a multiprocessor system, each processor must have its own Eclipse project and its own C Code to control it. This is a time-consuming process, but allows everything to run individually and if an issue is encountered in just a single system, only that one system needs to be recompiled and rerun.

The main issue encountered in this project is the question of how to write to memory. From an example found in [1], it was discovered that simple pointers to locations in memory could be used. As can be seen in Figure 5, using macros defined in each system's system.h file, a specific address could be referenced that would work for each individual system, regardless of the base address. This is important because the base address is different for the main processor than for the individual ones, due to the use of a bridge. The MAIN_MEMORY_BASE_ADDRESS macro in the main processor is 0x50000, but for the individual processors, shared_one and shared_zero, this address is 0x4050000, which is the starting address for the Avalon pipelined bridge. These are identical to the addresses chosen when building the Qsys system, plus the main memory offset.

The value chosen to begin writing to memory was the base address plus 0x02000. Since long int pointers were used, they were assumed to be 64-bit, so each slot in memory was offset by 0x00008 (8 bytes). Once memory pointers are created, writing to them or reading to them is simple: just dereference the pointer and either make an assignment to/from it to write/read at that location.

Writing the actual programs is quite simple once the memory issue has been dealt with. The main processor's program simply writes the speed values to the specified memory locations, stored in an array. These speed values are simply numbers used as the control condition in for-loops used to delay the LED display systems. The main loop then enters an empty while(1) loop to keep it turned on. The code for the main processor, main.c, can be found in Figure 5.



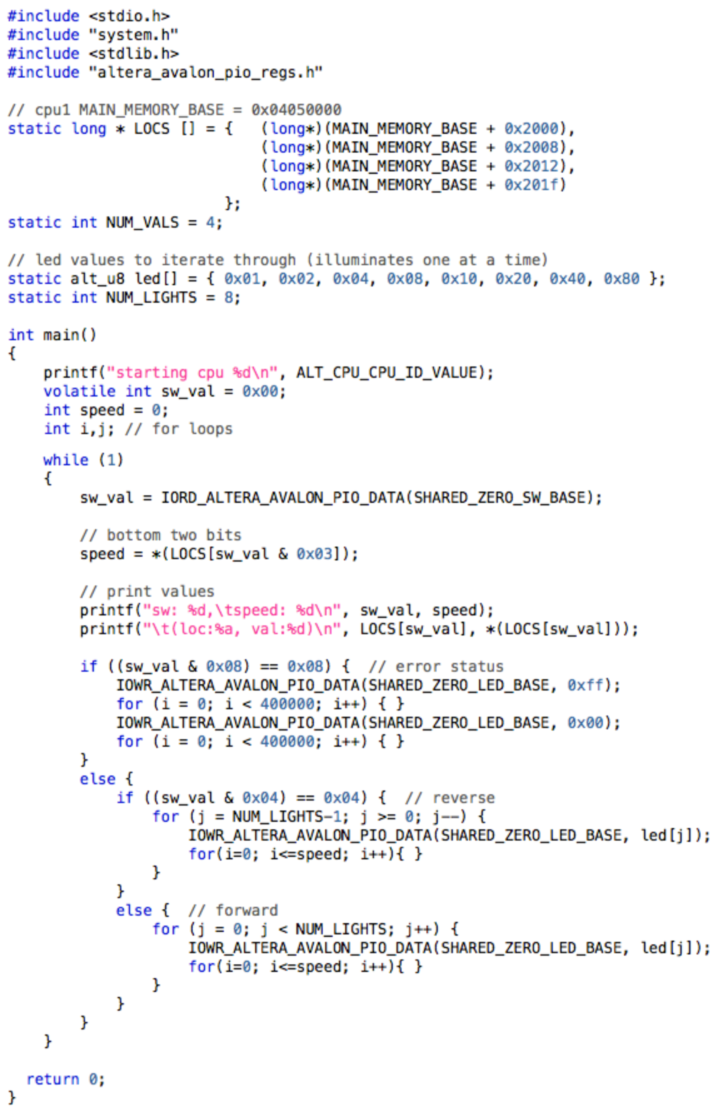
```c
#include <stdio.h>
#include "system.h"
#include <stdlib.h>

// MAIN_MEMORY_BASE = 0x00050000
// mem locations to store speed values
static long * LOCS [] = { (long*)(MAIN_MEMORY_BASE + 0x2000),
                          (long*)(MAIN_MEMORY_BASE + 0x2008),
                          (long*)(MAIN_MEMORY_BASE + 0x2012),
                          (long*)(MAIN_MEMORY_BASE + 0x201f)
                          };
// speed values: end-condition of for loop delay
static int VALS [] = { 0, 100000, 200000, 300000 };
static int NUM_VALS = 4;

int main()
{
    printf("starting cpu %d\n", ALT_CPU_CPU_ID_VALUE);

    int i;

    // store and print memory
    for (i = 0; i < NUM_VALS; i++) {
        *(LOCS[i]) = VALS[i];
        printf("Mem0 %d (%d) = %d\n", i, LOCS_0[i], *(LOCS[i]));
    }

    while (1) { }

  return 0;
}
```

Figure 5: main.c file

Each individual system simply enters a while(1) loop, which reads the value of the switches SW[0:1] and then finds the speed value in memory which they correspond to. It also determines in which direction to display the LEDs, based on SW[2] and whether or not to blink every light in an "emergency" conition with SW[3]. Selecting individual switches is done by a bitwise AND operation with the desired switches to isolate. For example, if trying to get the value of the lowest two switches, the following is done in an if-statement:

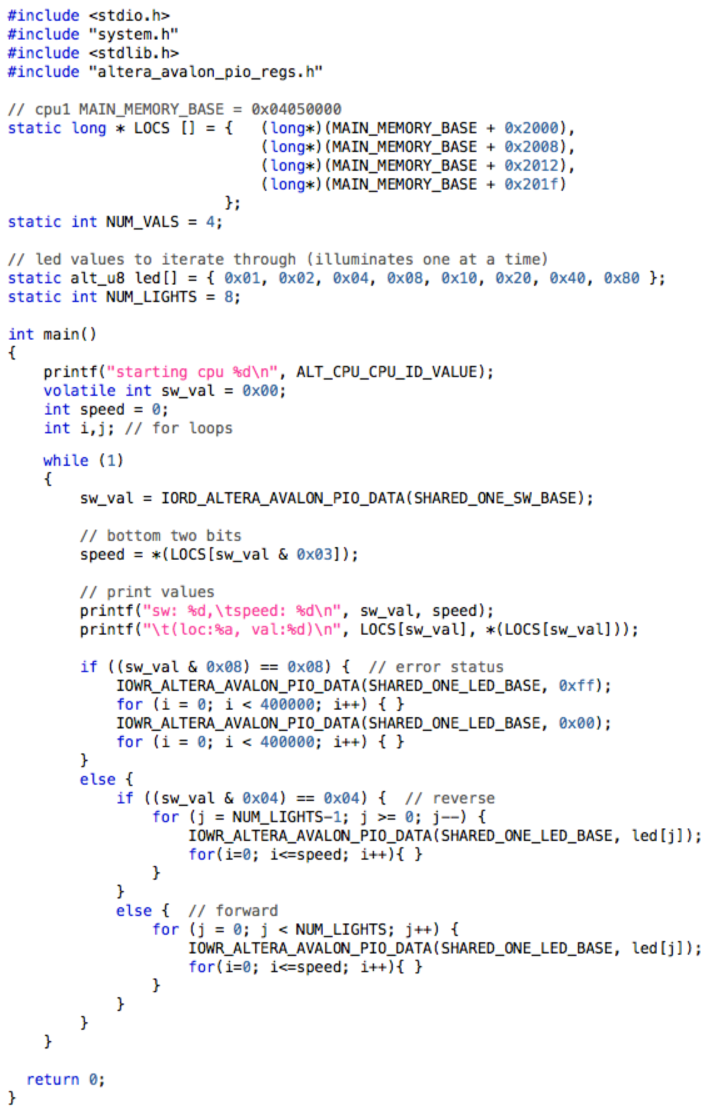
$$(sw\_val\ \&\ 0x03)\ ==\ 0x03 \tag{1}$$

A mutex should have been utilized, but since this system only initializes and then reads data from memory, there was no real risk of two systems simultaneously reading/writing from memory. Figures 6 and 7 contain the c code for the files cpu0.c and cpu1.c, respectively.



```c
#include <stdio.h>
#include "system.h"
#include <stdlib.h>
#include "altera_avalon_pio_regs.h"

// cpu1 MAIN_MEMORY_BASE = 0x04050000
static long * LOCS [] = {   (long*)(MAIN_MEMORY_BASE + 0x2000),
                            (long*)(MAIN_MEMORY_BASE + 0x2008),
                            (long*)(MAIN_MEMORY_BASE + 0x2012),
                            (long*)(MAIN_MEMORY_BASE + 0x201f)
                        };
static int NUM_VALS = 4;

// led values to iterate through (illuminates one at a time)
static alt_u8 led[] = { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };
static int NUM_LIGHTS = 8;

int main()
{
    printf("starting cpu %d\n", ALT_CPU_CPU_ID_VALUE);
    volatile int sw_val = 0x00;
    int speed = 0;
    int i,j; // for loops

    while (1)
    {
        sw_val = IORD_ALTERA_AVALON_PIO_DATA(SHARED_ZERO_SW_BASE);

        // bottom two bits
        speed = *(LOCS[sw_val & 0x03]);

        // print values
        printf("sw: %d,\tspeed: %d\n", sw_val, speed);
        printf("\t(loc:%a, val:%d)\n", LOCS[sw_val], *(LOCS[sw_val]));

        if ((sw_val & 0x08) == 0x08) {  // error status
            IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ZERO_LED_BASE, 0xff);
            for (i = 0; i < 400000; i++) { }
            IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ZERO_LED_BASE, 0x00);
            for (i = 0; i < 400000; i++) { }
        }
        else {
            if ((sw_val & 0x04) == 0x04) {  // reverse
                for (j = NUM_LIGHTS-1; j >= 0; j--) {
                    IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ZERO_LED_BASE, led[j]);
                    for(i=0; i<=speed; i++){ }
                }
            }
            else {  // forward
                for (j = 0; j < NUM_LIGHTS; j++) {
                    IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ZERO_LED_BASE, led[j]);
                    for(i=0; i<=speed; i++){ }
                }
            }
        }
    }

  return 0;
}
```

Figure 6: cpu0.c file



```c
#include <stdio.h>
#include "system.h"
#include <stdlib.h>
#include "altera_avalon_pio_regs.h"

// cpu1 MAIN_MEMORY_BASE = 0x04050000
static long * LOCS [] = {   (long*)(MAIN_MEMORY_BASE + 0x2000),
                            (long*)(MAIN_MEMORY_BASE + 0x2008),
                            (long*)(MAIN_MEMORY_BASE + 0x2012),
                            (long*)(MAIN_MEMORY_BASE + 0x201f)
                        };
static int NUM_VALS = 4;

// led values to iterate through (illuminates one at a time)
static alt_u8 led[] = { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };
static int NUM_LIGHTS = 8;

int main()
{
    printf("starting cpu %d\n", ALT_CPU_CPU_ID_VALUE);
    volatile int sw_val = 0x00;
    int speed = 0;
    int i,j; // for loops

    while (1)
    {
        sw_val = IORD_ALTERA_AVALON_PIO_DATA(SHARED_ONE_SW_BASE);

        // bottom two bits
        speed = *(LOCS[sw_val & 0x03]);

        // print values
        printf("sw: %d,\tspeed: %d\n", sw_val, speed);
        printf("\t(loc:%a, val:%d)\n", LOCS[sw_val], *(LOCS[sw_val]));

        if ((sw_val & 0x08) == 0x08) {  // error status
            IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ONE_LED_BASE, 0xff);
            for (i = 0; i < 400000; i++) { }
            IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ONE_LED_BASE, 0x00);
            for (i = 0; i < 400000; i++) { }
        }
        else {
            if ((sw_val & 0x04) == 0x04) {  // reverse
                for (j = NUM_LIGHTS-1; j >= 0; j--) {
                    IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ONE_LED_BASE, led[j]);
                    for(i=0; i<=speed; i++){ }
                }
            }
            else {  // forward
                for (j = 0; j < NUM_LIGHTS; j++) {
                    IOWR_ALTERA_AVALON_PIO_DATA(SHARED_ONE_LED_BASE, led[j]);
                    for(i=0; i<=speed; i++){ }
                }
            }
        }
    }

  return 0;
}
```

Figure 7: cpu1.c file

4 Conclusion

This project demonstrated the use of a simple multiprocessor system with shared memory. The most challenging part of the project was resolving problems with reading data from memory. While writing the program code, we at first did not have memory locations completely adjacent to one another. As a result, unpredictable values were returned when data was read from memory.

One possible improvement to this project would be to include software mutexes to make it possible to add more processors to the system. Software mutexes are program objects that allow multiple processors in the system share the same memory resource efficiently. These processors could perform actions such as displaying the selected speed value onto a 7-segment hexadecimal display or character LCD display. The system we created did not have mutex related issues because both processors were only reading data from memory.

Another prossible improvement to the this project would be to use the FPGAs GPIO ports to connect and control an actual conveyor belt. This would involve designing the necessary electronic components required to control the conveyor belt, as well as modifying the exsisting FPGA system.

References

[1] Basic Computer System for the Altera DE2-115 Board. The Altera Corporation. p 8. July 2010.

[2] Creating Multiprocessor Nios II Systems Tutorial. The Altera Corporation. June 2011.