Mauro Lopez

LAB 8 Report

ECEN-449

Section 502

November 15, 2018

Introduction:

In this lab we are going to keep learning more on how to create our own drivers and implement them into our Zybo board. In this lab we are going to continue and add an interrupt signal to our lab 7 hardware.

Procedure:

We will begin by creating a copy of lab 7 hardware and copy it into a new directory called lab8. Once we have a new directory, we need to open the project and update the settings. One of the things we need to do is change the path to our IP directory. We do this in vivado, we also need to delete our IP in our block diagram and add our new updated version to it.

Once we have a copy of our lab7 we can begin to edit our IR_Mod. We right click and select the edit and repackage ip option.

Here we will need to add a new output signal called “IR_interrupt” that will be an output signal. We need to update our logic so that this signal goes high whenever we get a new message. This will help us eliminate some of noise that happens when reading the signal.

Its important to note that we also need a register that will server as a “status/control” register. The lab suggest using the third register which is called “slv_reg2”. Since we are going to be writing to the upper 16 bits we need to go back and uncomment our function that allows to write to register two.

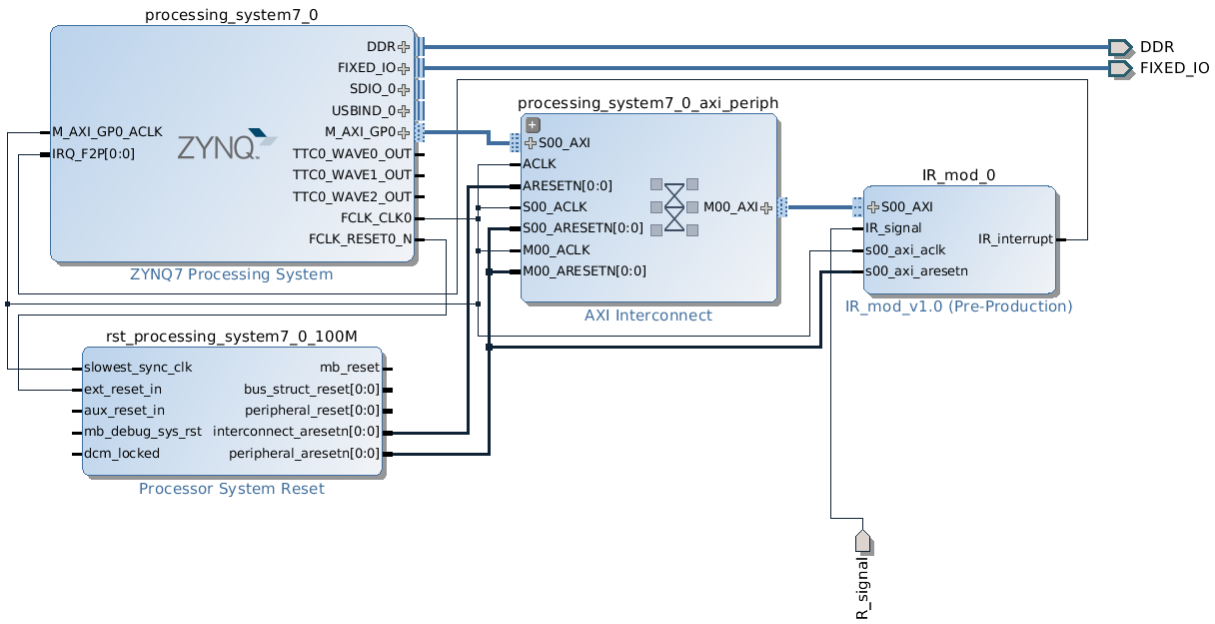
Once we have enable our register 2 to have the ability to read and write we can begin to implement our code. We should end up with something similar to this:

```
1.  // Add user logic here
2.  /*
3.  upper part of the code that gets edited.
4.  reg control_reset;
5.      always @( posedge S_AXI_ACLK )
6.      begin
7.       if ( S_AXI_ARESETN == 1'b0 || (control_reset == 1'b1))
8.          begin
9.            //slv_reg0 <= 0;
10.           //slv_reg1 <= 0;
11.           slv_reg2[31:16] <= 0;
12.           // the left part is going to be the control/reset
13.           //slv_reg3 <= 0;
14.         end
15. */
16. reg previous_Signal;
17. reg [31:0] counter;
18. reg possible_bit;// holds the possible bit
19. reg [11:0] remote_signal; // decoded signal from remote.
20. reg startSignal; // new message from remote
21. reg [31:0] index; // the place of the bit
22. reg keepCounting; // updates our counter.
23. assign IR_interrupt = slv_reg2[0];
24. // our interrupt will be set by the right most bit
25. // makes it easier to view.
```

```
always@(posedge clock) begin

    if (slv_reg2[31]) begin // if control is on
      slv_reg2[15:0] <= 0; // reset
      control_Reset <= 1'b1;// reset is high
    end
    else if (control_Reset==1'b1) begin
      control_Reset <= 1'b0;// reset is set back to low
    end

    else begin
    previous_Signal <= IR_signal;
    if ( (previous_Signal==1'b1) && (IR_signal==1'b0) begin // neg edge.

        keepCounting <= 1'b1;
    end

    else if ( (previous_Signal==1'b0) && (IR_signal==1'b1)) begin //posedge
        index <=index-1; // updates our index
        keepCounting <= 1'b0; // no counts on high signal
        counter <= 0; // resets our counter back to 0

        if (startSignal && index < 0) begin
        /*if our index is less than 0 means we have more than 12 bits.
        */
            slv_reg0 <= remote_signal;
            slv_reg1 <= slv_reg1 + 1;
            slv_reg2[0] <= 1;
            index <= 11;
            startSignal <= 0;
        end

        else if (startSignal && index >= 0 ) begin
        /*updates the bits by index
        */
            remote_signal[index] <= possible_bit;
        end
    end


    if (keepCounting && ~IR_signal) begin
    /*we are only counting on a (1'b0) signal and if occurs
     *keepcounting flag is on
    */
        counter <= counter + 1; // updates counter
            // our ranges for our giving value
            if (counter >= 20000 && counter <= 65000) begin
                possible_bit <=0;
            end

            else if (counter >= 64000 && counter <= 130000) begin
                possible_bit <= 1;
            end

            else if (counter >= 135000) begin
                    startSignal <= 1;
                    index <= 11;
                    //new signal, set index back to 11
            end


```

```
87.     end
88.     end
89.
90.
91. end
92.
93.
94. // User logic ends
```

Once we have our IR_Mod updated we need to also update our block diagram and end up something like this. NOTE: This block diagram is after we test our part1 of the lab.



Once we have our code ready we need to create and appropriate .xdc file that will be used to test our interrupt signal. For this we use the U20 pin in our zybo board.

Once we have completed this, we are ready to start to generate our bitstream.

Once we have successfully generated our bit stream is time to export our bitstream into SDK and launch SDK.

Once we are in SDK we need to test our progress and see that everything works accordingly. Below is the code I used to test this part of the lab:

```
1.  #include <stdio.h>
2.  #include "platform.h"
3.  #include "xparameters.h"
4.  #include "IR_mod.h"
5.
6.  #define BASE_ADDRESS XPAR_IR_MOD_0_S00_AXI_BASEADDR
7.
8.  void print(char *str);
9.
10. int main()
11. {
12.
```

```
    print("Hello World\n\r");
    init_platform();
    unsigned long slv_reg0;
    unsigned long slv_reg1;
    unsigned long slv_reg2;
    unsigned long slv_reg3;
    unsigned long oldreg;
    unsigned long interrupt;

    while(1){
        slv_reg2 = IR_MOD_mReadReg(BASE_ADDRESS, 8);
        interrupt = (IR_MOD_mReadReg(BASE_ADDRESS, 8) & 0x00000001);
        /* To see the value of interrupt we use a mask and "AND"
         * all of its value except for the right most bit. This
         * returns our interrupt signal.
        */
        if(interrupt){
        //To prevent from constantly having values, we only need to see
        // when we have a new signal
        slv_reg0 = IR_MOD_mReadReg(BASE_ADDRESS, 0);
        slv_reg1 = IR_MOD_mReadReg(BASE_ADDRESS, 4);

        printf("slv_reg0(message) = %d \n",slv_reg0);
        printf("slv_reg1(count)  = %d \n",slv_reg1);
        printf("slv_reg2(control/status)  = %d \n",slv_reg2);
        // this helps see what values we have so far.
        }
        IR_MOD_mWriteReg(BASE_ADDRESS, 8, 0x80000000);
        // write to register2 and set our reset signal high
        // should set our interrupt back to zero.
        printf("Resetting Interrupt...\n");


    }


    cleanup_platform();
    return 0;
}
```

Once we have successfully completed updating the IR_interrupt signal, we can begin to create our driver file for our Zybo board.

For this part we are given a .C and an .h file that we use as a guideline to complete the lab.

Once completed our code should look similar to this :

```
/*  irq_test.c - Simple character device module
*
*  Demonstrates interrupt driven character device.  Note: Assumption
*  here is some hardware will strobe a given hard coded IRQ number
*  (200 in this case).  This hardware is not implemented, hence reads
*  will block forever, consider this a non-working example.  Could be
*  tied to some device to make it work as expected.
*
*  (Adapted from various example modules including those found in the
*  Linux Kernel Programming Guide, Linux Device Drivers book and
*  FSM's device driver tutorial)
```

```
*/

/* Moved all prototypes and includes into the headerfile */
#include "irq_test.h"


/* This structure defines the function pointers to our functions for
opening, closing, reading and writing the device file.  There are
lots of other pointers in this structure which we are not using,
see the whole definition in linux/fs.h */
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
typedef struct node {
    //we split the message into two signal
    char byte0;
    char byte1;
} MESSAGE; // allias name
void* virt_addr; //virtual address pointing to ir peripheral

                /*
                * This function is called when the module is loaded and registers a
                * device for the driver to use.
                */
int my_init(void)
{
    printk(KERN_INFO "Mapping virutal address...\n");
    init_waitqueue_head(&queue);    /* initialize the wait queue */

                                    /* Initialize the semaphor we will use to protect a
gainst multiple
                                    users opening the device  */

    virt_addr = ioremap(PHY_ADDR, MEMSIZE);
    printk("Physical Address: 0x%x\n", PHY_ADDR);
    printk("Virtual Address: 0x%x\n", virt_addr);

    sema_init(&sem, 1);

    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if (Major < 0) {
        printk(KERN_ALERT "Registering char device failed with %d\n", Major);
        return Major;
    }

    printk(KERN_INFO "Registered a device with dynamic Major number of %d\n", Major);
    printk(KERN_INFO "Create a device file for this device with this command:\n'mknod /
dev/%s c %d 0'.\n", DEVICE_NAME, Major);

    return 0;       /* success */
}

/*
* This function is called when the module is unloaded, it releases
* the device file.
*/
void my_cleanup(void)
```

```
71. {
72.     /*
73.     * Unregister the device
74.     */
75.     unregister_chrdev(Major, DEVICE_NAME);
76.     printk(KERN_ALERT "unmapping virtual address space...\n");
77.     iounmap((void*)virt_addr); // unmapping address
78. }
79.
80.
81. /*
82. * Called when a process tries to open the device file, like "cat
83. * /dev/irq_test".  Link to this function placed in file operations
84. * structure for our device file.
85. */
86. static int device_open(struct inode *inode, struct file *file)
87. {
88.     int irq_ret;
89.
90.     if (down_interruptible(&sem))
91.         return -ERESTARTSYS;
92.
93.     /* We are only allowing one process to hold the device file open at
94.     a time. */
95.     if (Device_Open) {
96.         up(&sem);
97.         return -EBUSY;
98.     }
99.     Device_Open++;
100.
101.          /* OK we are now past the critical section, we can release the
102.          semaphore and all will be well */
103.          up(&sem);
104.
105.          /* request a fast IRQ and set handler */
106.          irq_ret = request_irq(IRQ_NUM, irq_handler, 0 /*flags*/, DEVICE_NAME, NULL);
107.          if (irq_ret < 0) {       /* handle errors */
108.              printk(KERN_ALERT "Registering IRQ failed with %d\n", irq_ret);
109.              return irq_ret;
110.          }
111.
112.          try_module_get(THIS_MODULE);    /* increment the module use count
113.                                          (make sure this is accurate or you
114.                                          won't be able to remove the module
115.                                          later. */
116.
117.          msg_Ptr = NULL;
118.          printk("Device has been opened\n");
119.
120.          //allocating messageQueue with enough bytes to store 100 of MESSAGE
121.          messageQueue = (MESSAGE*)kmalloc(100 * sizeof(MESSAGE), GFP_KERNEL);
122.
123.          return 0;
124.      }
125.
126.      /*
127.      * Called when a process closes the device file.
128.      */
129.      static int device_release(struct inode *inode, struct file *file)
130.      {
```

```
131.            Device_Open--;      /* We're now ready for our next caller */
132.
133.            free_irq(IRQ_NUM, NULL);
134.
135.            /*
136.            * Decrement the usage count, or else once you opened the file,
137.            * you'll never get get rid of the module.
138.            */
139.            module_put(THIS_MODULE);
140.            printk("Cloing the device... \n");
141.            return 0;
142.        }
143.
144.        /*
145.        * Called when a process, which already opened the dev file, attempts to
146.        * read from it.
147.        */
148.        static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
149.            char *buffer,   /* buffer to fill with data */
150.            size_t length,  /* length of the buffer     */
151.            loff_t * offset)
152.        {
153.            int bytes_read = 0;
154.
155.            int i = 0;
156.
157.
158.            length = writeIndex * 2;
159.
160.            printk("Read %d messages since last checked...\n", length);
161.            writeIndex = 0;
162.
163.            msg_Ptr = (char*)messageQueue;
164.            for (i = 0; i < length; i++) {
165.                /*
166.                * The buffer is in the user data segment, not the kernel segment
167.                * so "*" assignment won't work.  We have to use put_user which
168.                * copies data from the kernel data segment to the user data
169.                * segment.
170.                */
171.                put_user(*(msg_Ptr++), buffer++); /* one char at a time... */
172.                bytes_read++;
173.            }
174.
175.            /* completed interrupt servicing reset
176.            pointer to wait for another
177.            interrupt */
178.            msg_Ptr = NULL;
179.
180.            /*
181.            * Most read functions return the number of bytes put into the buffer
182.            */
183.            return bytes_read;
184.        }
185.
186.        /*
187.        * Called when a process writes to dev file: echo "hi" > /dev/hello
188.        * Next time we'll make this one do something interesting.
189.        */
190.        static ssize_t
191.        device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
```

```
192.        {
193.
194.            /* not allowing writes for now, just printing a message in the
195.            kernel logs. */
196.            printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
197.            return -EINVAL;     /* Fail */
198.        }
199.
200.        irqreturn_t irq_handler(int irq, void *dev_id) {
201.            sprintf(msg, "IRQ Num %d called, interrupt signal processed %d times\n", irq
    , counter++);
202.            printk("%d...\n", counter);
203.            msg_Ptr = (char*)messageQueue; //pointer array to the start of the queue
204.            message = ioread32(virt_addr + 0); // read from register 0;
205.
206.            if (writeIndex >= 100) {
207.                // once we have 100 messages we need to reset
208.                writeIndex = 0;
209.            }
210.            /*using our class node to access its public members
211.             *and updating its indivial bites.
212.            */
213.            messageQueue[writeIndex].byte0 = byteBuff[0];
214.            messageQueue[writeIndex].byte1 = byteBuff[1];
215.            writeIndex++; // messages we have written
216.
217.            iowrite32(0x80000000, virt_addr + 8);
218.            //clear the interrupt signal in our slv_reg2 by setting our
219.            // 31st bit high.
220.
221.            return IRQ_HANDLED;
222.        }
223.
224.
225.
226.        /* These define info that can be displayed by modinfo */
227.        MODULE_LICENSE("GPL");
228.        MODULE_AUTHOR("Mauro Lopez (and others)");
229.        MODULE_DESCRIPTION("Module which creates a character device and allows user inte
    raction with it");
230.
231.        /* Here we define which functions we want to use for initialization
232.        and cleanup */
233.        module_init(my_init);
234.        module_exit(my_cleanup);
```

Once we have completed this step, we need to create our boot.bin file just like we did in lab4. We do this by exporting our new generated bitstream into SDK and creating a new project for a Zynq images. We use the uboot files from our lab 4 directory to avoid copying new files. Once we have created the image we can exit out.

Next we need to add the following to our .dts file. we have done this in the past, we will need to include the following to our file:

ir_demod{

compatible = “ecen449,ir_demod”;

interrupt-parent = <&ps7_scugic_0>;

interrupts = <0 61 1>;

reg = <0x43C00000 0x10000> ;

}

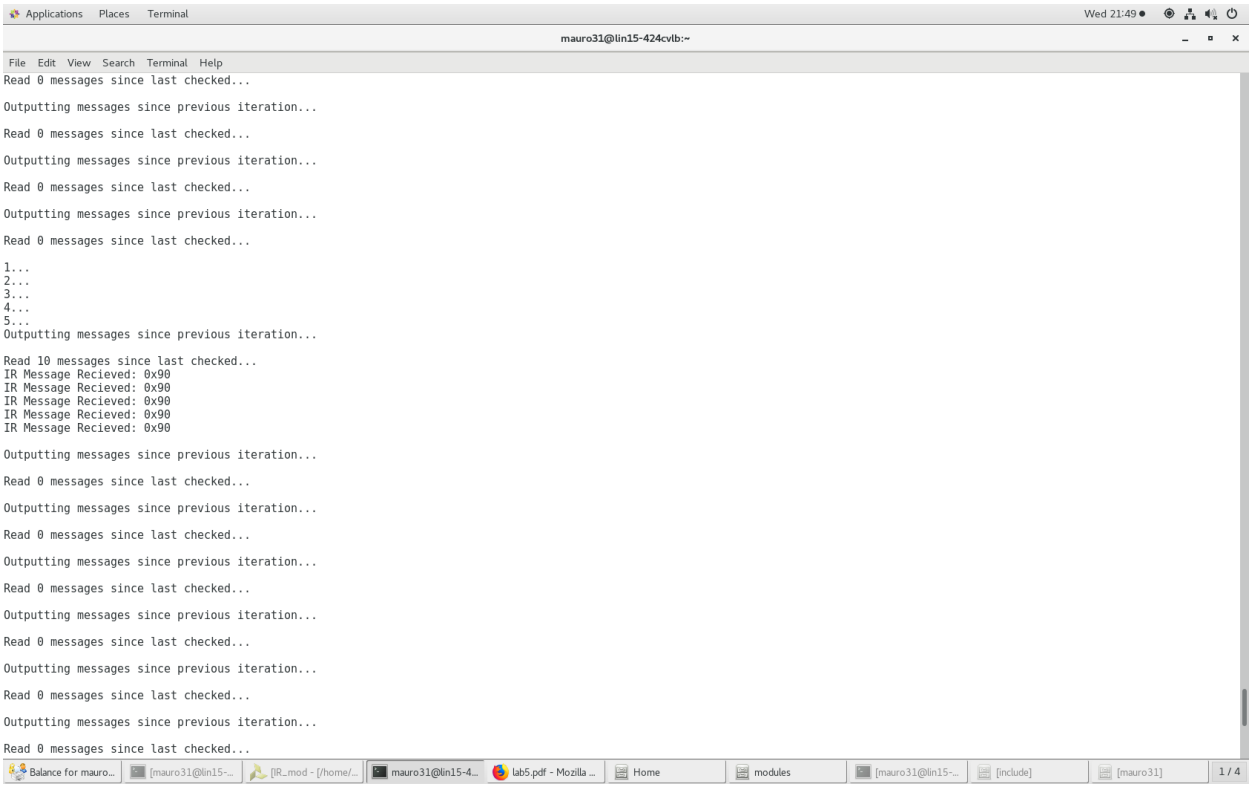
Once we have this, we can generate our new devicetree.dtb.

We now need to create an executable file that we can use to run our interrupt driver we created.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    printf("IR Remote Device \n");
    unsigned int IR_message;
    char* msg_Ptr = (char*)&IR_message; //used to write 1 byte at a time to IR_message

    int f_d; //file descriptor
    char input = 0;
    int size = 200;
    char* outputBuffer = (char*)malloc(size  * sizeof(char));
    // create space for our buffer, should be able to hold 100 mesages.

    f_d = open("/dev/irq_test", O_RDWR);
    //checking our opening command

    //handle error opening file
    if (fd == -1) {
        printf("Failed to open device file!\n");
        return -1;
    }
    int i ;
    while(true){
        printf("Outputting IR_messages since previous iteration...\n\n");
        int num_bytes_read = read(fd, outputBuffer, size);

        for (i = 0; i < num_bytes_read/2 ; i+2) {
            /* we are going to read our values from our buffer
             * and print them out
            */
             Index = i*2;
            msg_Ptr[0] = outputBuffer[i ];
            msg_Ptr[1] = outputBuffer[ i+1];

            printf("IR Message Recieved: 0x%x\n", IR_message);
        }
        printf("\n\n");
        sleep(1);
```

```
43.     }
44.
45.     close(fd); /// close the driver filer.
46.     free(outputBuffer);// release memory to avoid memory leaks and
47.     // possibilities of chrasing the program.
48.     return 0;
49. }
```

Result:



Conclusion:

over all the lab was a good source of practice when it comes to working with both software and hardware. I was having trouble with creating the Verilog code at first because we had to implement the interrupt signal. The lab became very tedious at the end since it involved going back to the IP editor and all the way back to our zybo boards. There was also times where SDK would crash for no reason and everything became corrupted. There was lots of unstable things that came into play in this lab, and that's something that made this lab less enjoyable than other labs.

Questions:

a)

Contrast the use of an interrupt-based device driver with the polling method used in the previous lab.

The main difference between the two method is that this method seemed more inefficient since it didn't require to constantly keep checking on the signal. The interrupt signal served as a way to save resources.

b)

Are there any race conditions that your device driver does not address? If so, what are they and how would you fix them?

some of the race condition that can occur are when im reading the ir signal, sometimes I miss point and we can get a different value and that will give the wrong output.

c)

If you register your interrupt handler as a 'fast' interrupt (i.e. with the SA INTERRUPT flag set), what precautions must you take when developing your interrupt handler routine? Why is this so? Taking this into consideration, what modifications would you make to your existing IR-remote device driver?

We want to perform our actions faster, meaning we need to keep a minimal use of statements that require computational power. To create a fast interrupt in our driver, we would have to disable most other commands that require computational power such as our prink statements and try to minimize using logic that would result in a bigger runtime.

d)

What would happen if you specified an incorrect IRQ number when registering your interrupt handler? Would your system still function properly? Why or why not?

We can potentially receive an error. We are going to give the wrong IR interrupt to a different signal, meaning when we print out the statements we can potentially print out values that are incorrect and will lead to incorrect commands.