ECEN 749: Microprocessor Systems Design

Lab 7: IR remote device Driver

Manav Gurumoorthy

830000011

Section 603

Table of Contents

Introduction

The purpose of this lab was to create and test a hardware circuit that receives and decodes Infrared signals from a TV remote. The hardware detection circuit consists of a photodiode as a receiver. The photodiode input is given to a op-amp configured as a comparator, it is compared to a threshold level to generate the pulses. On the FPGA a circuit was designed to demodulate the pulses using a software routine.

Procedure

Configuration

The receiver circuit was built in hardware. A block diagram was generated that contained all the necessary functional units.

Part 1: Hardware Circuit

The circuit consists of a photodiode as a receiver and an op-amp as a comparator. The input signal is compared to a threshold voltage. In this case the threshold voltage was 1.65 V, which is the halfway point of the input signal, this is in order to detect the edges of the signal. The signal was viewed on the oscilloscope and the timing for the start, 0 and 1 pulses was recorded. The duration of the start pulse was 2.4 ms, the '1' pulse was 1.2 ms and the '0' pulse was 0.6 ms. This information was needed in the demodulation software.

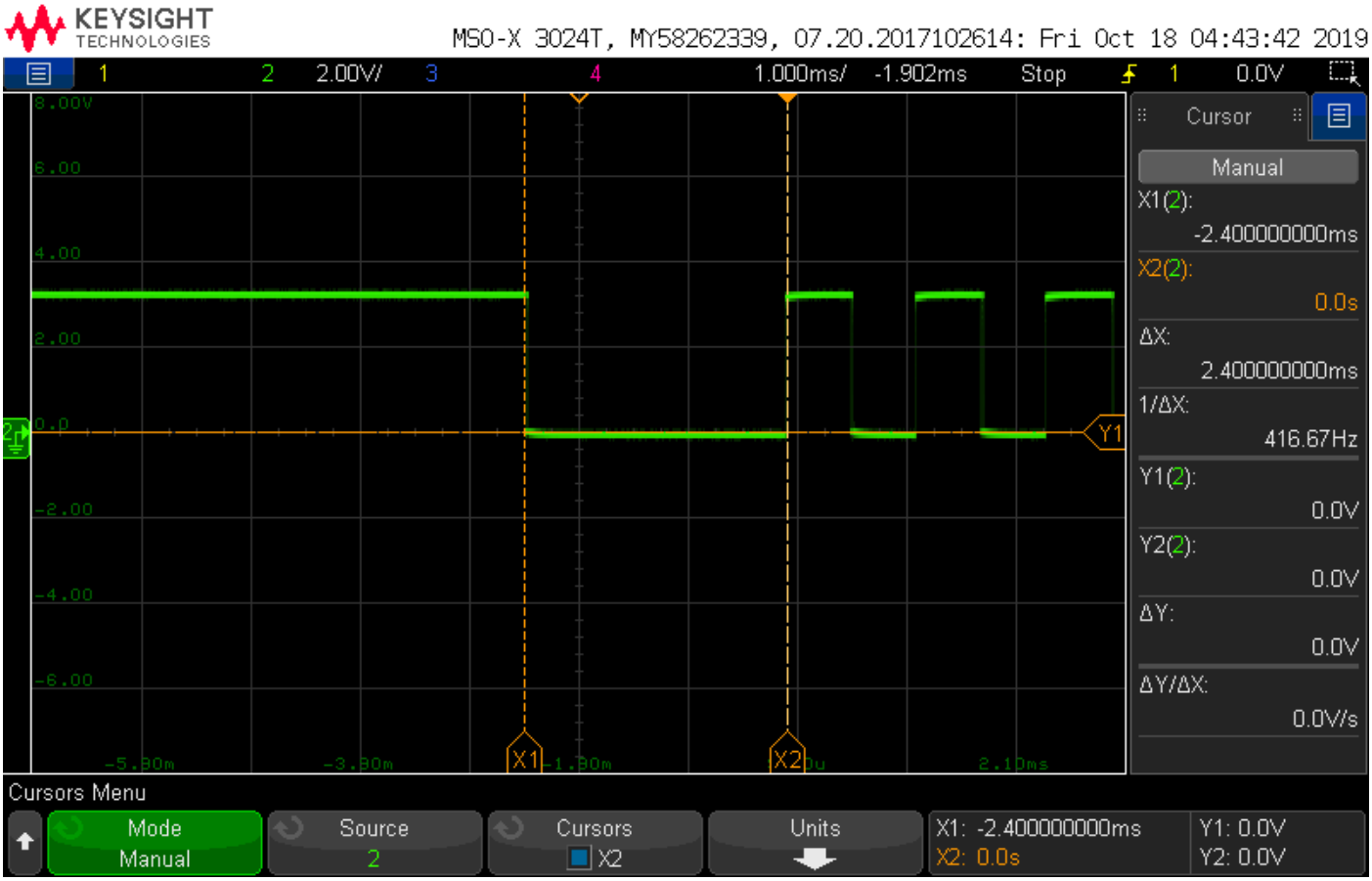


Figure 1: Duration of the start pulse

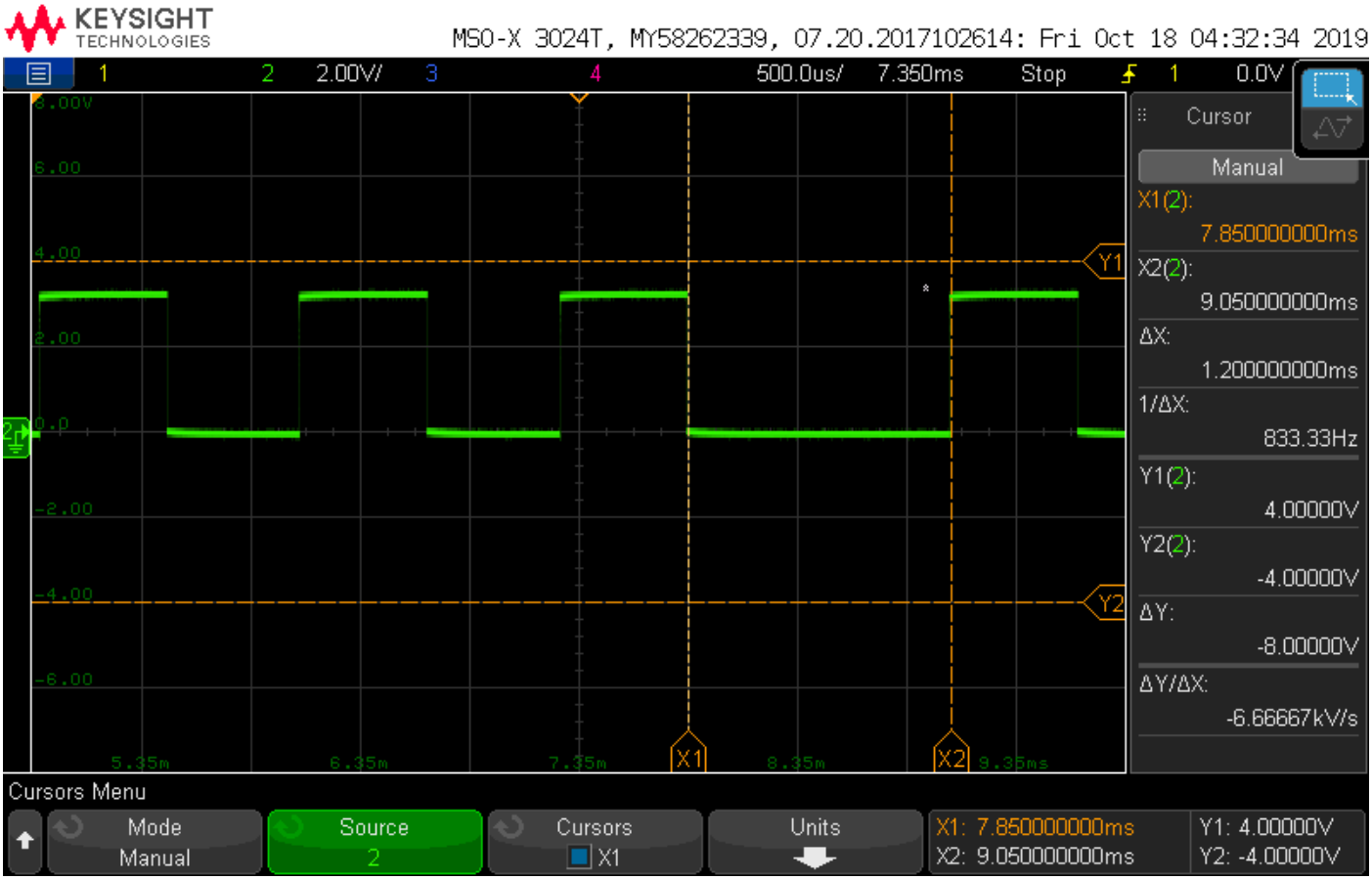


Figure 2: Duration of the '1' pulse

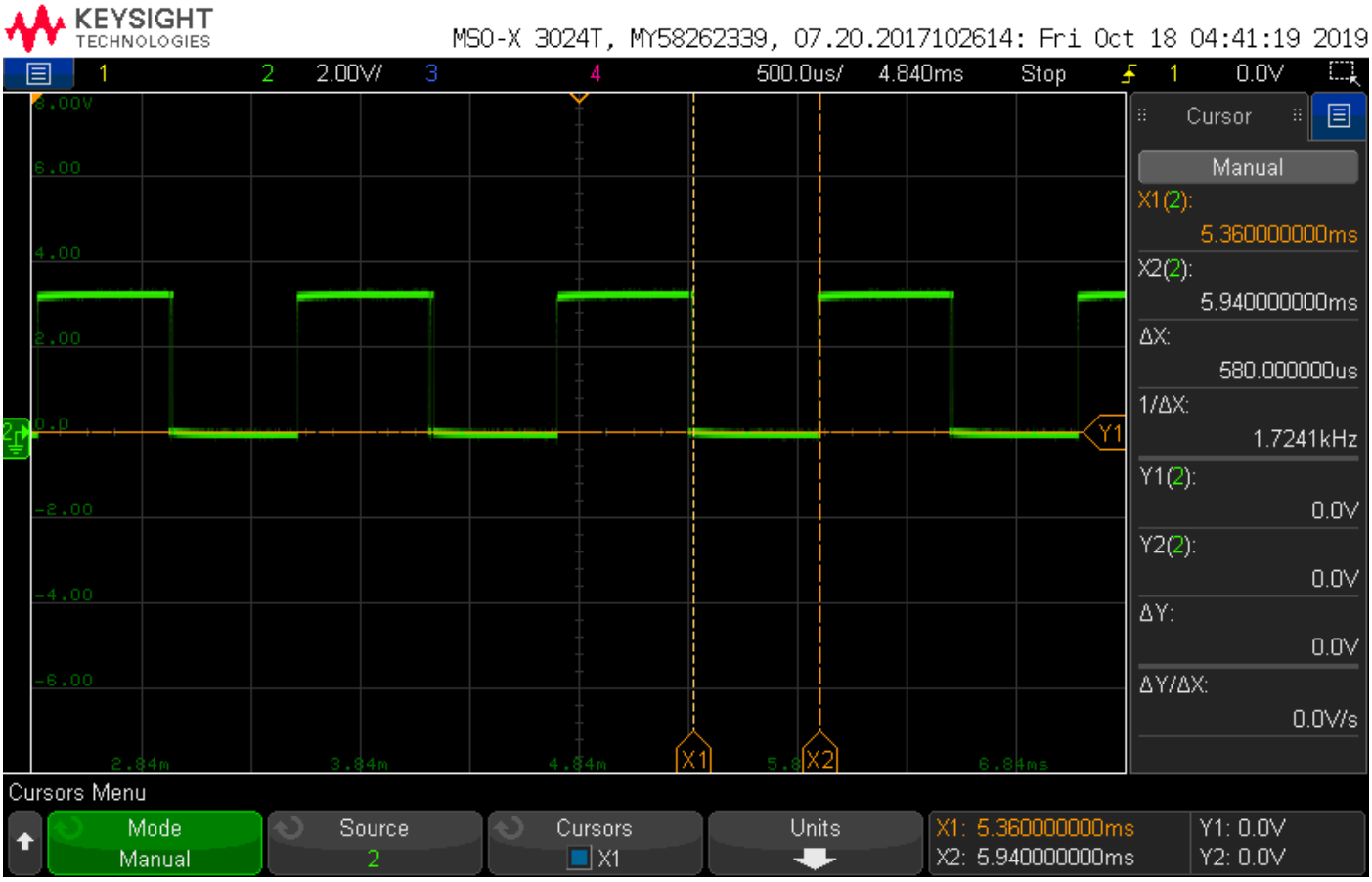


Figure 3: Duration of the '0' pulse

Part 2: FPGA Software

The FPGA software was written to take the modulated input signal and detect the individual pulses. The code is written such that it waits for a start pulse and then counts 12 bits. Using the timing information from part (a) the value of the recorded bit was discerned. The picocom output is shown below.

The software was written in such a way that it uses a divided clock, that is at one thousandth the speed of the actual clock. At every positive edge of this slowed down clock, a routine is triggered that performs the detection of the pulses being transmitted.

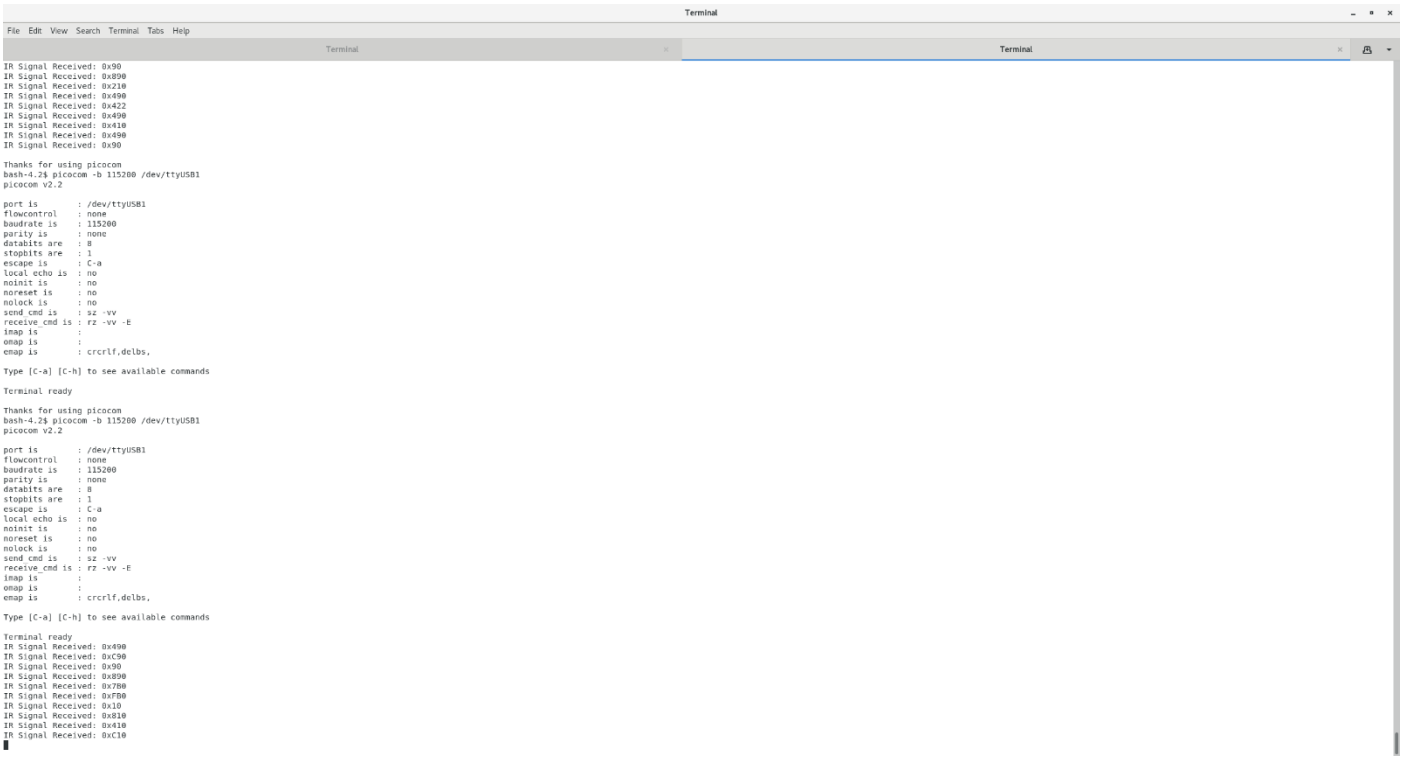
The hardware output is an open collector , hence the output is pulled down whenever a pulse is seen at the input, so even in software it was necessary to track the negative edges to detect the start of the pulse. The program does this by waiting for the *IR_signal* to go low, while a reg holding the previous signal is still high, at this point the program begins counting. The program counts upto 12 bits. The 12 bits together form the entire message being transmitted. Every time a bit is recorded the timing details are used to discern the value of the bit. (start: 2.4ms, 0: 0.6ms , 1: 1.2ms). To ensure the 1, 0s were correctly identified, the code waits for the halfway point between 0,1 to accurately detect the value being received. In the code if the 'start' sequence is detected it is just discarded.

User Logic for the ir_demod IP:

```
reg [31:0] clockTics;   // to count the timing of the IR_signal
      reg clock;
      assign reset = ~S_AXI_ARESETN; //S_AXI_ARESETN is active low
      assign fastClock = S_AXI_ACLK;
      /* Dividing clock by 1,000 */
      always@(posedge fastClock) begin
            if (clockTics == 1000 && ~reset) begin
                  clock <= 1;
                  clockTics <= 0;
            end

            else if (reset) begin
                  clockTics <= 0;
                  clock <= 0;
            end

            else begin
                  clockTics <= clockTics + 1;
                  clock <= 0;
            end
      end

      reg prevSignal;/* Old Signal used for edge detection */
      reg [31:0] counter;/* Counter to count time of ~IR_signal */
      reg bit_value;
      reg [11:0] demodulatedMessage;
      reg startSignal;
      reg [31:0] bitCount;
      reg keepCounting;

      always@(posedge clock) begin
            prevSignal <= IR_signal;
            if (prevSignal && ~IR_signal) begin
                  /* We start counting on the negative edge of the clock */
                  keepCounting <= 1'b1;
            end

            else if (~prevSignal && IR_signal) begin //if we just resolve a bit
                  /* We stop counting when we reach the positive edge again */
                  bitCount <= bitCount + 1; //increament bitCount (we just read a
bit)
                  keepCounting <= 1'b0; //stop counting
                  counter <= 0; //reset counter
```

```
                //store the bit we just read into current message

                if (startSignal && bitCount >= 12) begin //if we have received a
full message
                      slv_reg0 <= demodulatedMessage;
                      slv_reg1 <= slv_reg1 + 1;
                      bitCount <= 0;
                      startSignal <= 0;
                end

                else if (startSignal && bitCount < 12 && bitCount != 0) begin
                      demodulatedMessage[11 - (bitCount - 1)] <= bit_value;
                end
          end

          /* IR Signal is Active Low */
          if (keepCounting && ~IR_signal) begin
                counter <= counter + 1;

                      if (counter >= 20 && counter <= 68) begin
                            bit_value <=0;
                      end

                      else if (counter >= 69 && counter <= 134) begin
                            bit_value <= 1;
                      end

                      else if (counter >= 135 && counter <= 250) begin
                                  startSignal <= 1;
                                  /* Initialize bit count to 0 */
                                  bitCount <= 0;
                      end

                      else begin

                            bit_value <= 0;
                      end

          end


     end
```

Results



Conclusion

Lab 6 describes the way the user space and kernel space interact with each other through device drivers.

Questions

1.
    a. Volume up: 0x490
    b. Volume dn: 0xC90
    c. Channel up: 0x90
    d. Channel dn: 0x890
    e. Stop: 0x7B0
    f. Play: 0xFb0
    g. 1: 0x10
    h. 2: 0x810
    i. 3: 0x410
    j. 4: 0xC10
2. When a button is pressed multiple copies are sent this is to ensure that if any errors occur during transmission the receiver can recover code it may have missed. On average my observation on the oscilloscope was that the message was transmitted 3 to 5 times. (or is it because of polling and absence of the interrupt.)
3. The signal being described is an interrupt. The signal can go high whenever a message is received. My code uses two always blocks. One block was to divide the clock. The other block contained the logic to evaluate the messages being received. I would consider using a second signal to the processor that could trigger the second always block whenever it would go high. This could be used by the processor as an interrupt trigger.

```
// VERILOG code snippet.
output reg irq;
if ( newMessage)
        irq<=1;
else
        irq <=0;

if( irq) {
*insert routine to demod
}
```