Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

Instructors: Gim Hom and Joe Steinmyer
Project Supervisor: Mike Wang
Fall 2019

6.111 Digital Systems Laboratory

FPGA RFID Utility – Final Report

Hannah Field and Miles Dai

December 11, 2019

Contents

1 Overview

In the fields of corporate and building security, contactless smartcards and proximity cards are the dominant form of access control. Indeed, Radio Frequency Identification (RFID) is the cornerstone of MIT's access control system. In recent years, non-contact forms of payment using Near-Field Communication (NFC) have also been growing in popularity. In our project, we would like to explore the security of this system by investigating the signals transmitted from these devices and attempting to replicate them.

2 Background

RFID is a subset of more general non-contact credential systems. In particular, MIT primarily uses passive RFID in the low frequency band, with card readers broadcasting 125kHz signals. Passive here refers to the fact that the signal sent by the card reader is sufficient to power the onboard circuitry that is used to transmit the ID data.

RFID cards come in a variety of flavors. The onboard IC can be read-only, read-write, or write-once, read-many (WORM). Read-only cards have the ID number baked into the circuitry. Read-write cards allow for a card reader to edit the information on the card. WORM cards allow the end user to write to the card once, after which it becomes read-only.

2.1 Data Transmission

Embedded within each ID card is a wire coil connected to an integrated circuit. This wire loop picks up the AC signal emitted by the card reader and rectifies it to provide power to the IC. The purpose of the IC is to effectively modulate the impedance across the ends of the coil. Because the coil in the card acts as the secondary of a transformer (with the card reader being the primary), the impedance changes in the card are reflected across the air gap and can be detected by the card reader.

The particular brand of ID card used by MIT uses binary phase-shift keying (BPSK) to encode data. The card reader provides the 125 kHz carrier frequency which the onboard IC in the card uses to superimpose a 62.5kHz data signal. This data signal is then phase shifted to send information.

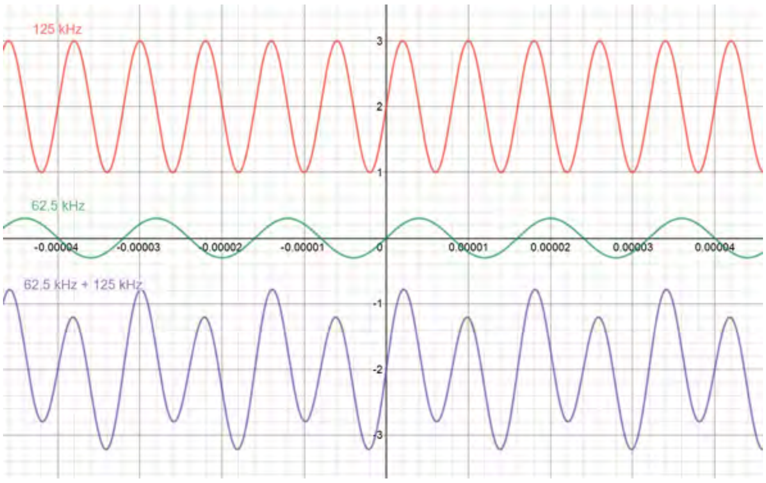

Figure 1: A 62.5kHz signal (green) superimposed on top of a 125kHz carrier (red). The result (purple) is alternating "high" and "low" peaks.

The ICs on the MIT ID cards send one bit every 16 cycles of the 62.5 kHz wave. If the current bit is different from the previous bit, then the IC will cause a phase shift in the 62.5 kHz signal. This looks like two consecutive high or low peaks in the carrier waveform.

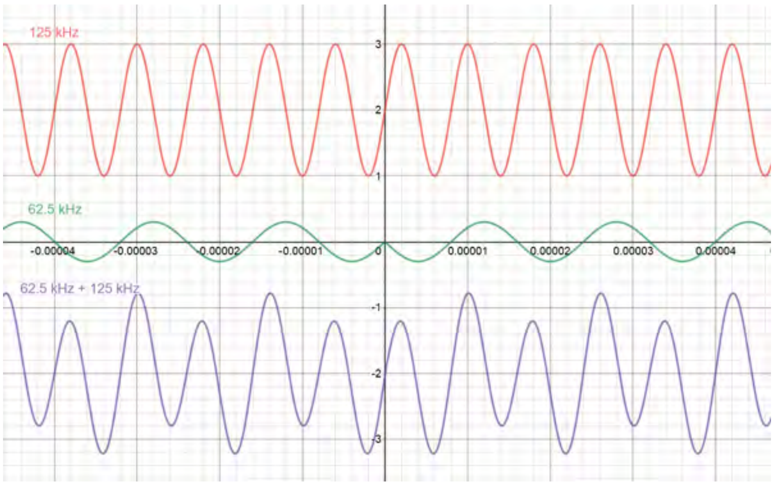

Figure 2: A phase-shifted signal (green) superimposed on the 125kHz carrier (red) to produce the purple waveform. The result (purple) is two high or low peaks at the location of phase shift.

There are essentially three ways to emit a spoofed signal. The first is to record the coil's response to an ID card, and store some number of samples using an ADC. The downsides to this approach are that it requires (1) a lot of memory, (2) a faster ADC than that on the Nexys 4 DDR, and (3) inability to spoof ID cards not already in the system.

A second approach is to recreate the 62.5 kHz signal. This method circumvents all of the issues exhibited by the previous approach. Unfortunately, generating this discontinuous signal would prove difficult, and questions of timing synchronization between the incoming and spoofed signal arise.

The last method and, in particular, the approach we take in this project is to spoof the superposition signal. In this approach, we modulate the natural response of a receive coil to the incoming 125 kHz signal. This method provides a clean solution to the problem of time synchronization; the circuitry is discussed in section 4.

2.2 Data Packet Structure

The data packet sent by the MIT ID is of a constant structure. Because the password is static and there is no acknowledgement protocol with the reader, it is relatively simple to reverse engineer the structure of the data packet that is sent. In the table below, the bits are sent from LSB to MSB. That is, the 30 zeros are sent first followed by bit 30 all the way to 224.

Bits	**Length**	**Description**
[29:0]	30	All 0's; synchronization
[51:30]	22	MIT constant bits
[84:52]	33	Personal bits
[224:85]	139	Constant bits

Based on examining multiple ID cards, it is possible to determine that only bits 52 through 84 change between cards. These, we believe are the bits that are being modified to produce unique identifiers.

3 Goals

3.1 Commitment

- Read module: this module will take raw analog input from the card reading coil and translate that into bits. Proof of goal: display bits on VGA screen.
- Spoof module: this module will simulate the PSK 62.5kHz signal generated by the ID card. This can be demonstrated by viewing the resulting waveform on an oscilloscope and compared to the response of an actual ID card. Proof of goal: View phase shifted waveform and incoming waveform on oscilloscope.
- Basic Deliverable: Upload code to two FPGA's, and show the spoofed signal is correctly read.

3.2 Goals

- SD Card Interface: Each time a new ID is read, offer the option to save it to an SD card. Then, select ID numbers from the SD card to playback as a spoofed signal. Index into the SD card via switches.
- VGA Interface: Add visual navigation for the SD card. Display a list of ID numbers stored on the SD card. Scroll through them with the up and down buttons. Use the center button to select an ID to playback.

3.3 Stretch Goals

- Program Cards: Reverse engineer the programming protocol by reading the bits from the Arduino programmer.
- All Systems Go: Read Joe or Jim's ID card onto the SD card, and spoof their ID to open the 38-6 lab doors.

4 Spoofing (Hannah)

The spoofed signal is achieved by modulating a receive coil's impedance via the use of a mosfet shorted accross the coil. In this manner, the superposition of the 125 kHz card reader signal and the 62.5 kHz data signal is simulated by consecutively turning on and off the mosfet. The key aspect of the spoofing system is that a phase-locked pulse is generated from the incoming signal. This phase-locked pulse is effectively used as the clock for turning on and off the mosfet, allowing the spoofed signal to phase align with the incoming signal from the card reader. We suspect this is a necessary feature for our spoofed signal to be recognized by MIT card readers.

4.1 Analog Frontend

4.1.1 Inductors

The inductors are hand-wound with approximately 20-turns and encompass a roughly 2-inch diameter. These inductors had inductances in the tens-of-micro-Henry range. We found that we had the best results using standard lab wire. One test with a 1000x larger milli-Henry range inductor made from magnet wire performed worse as a receive coil, which we believe is

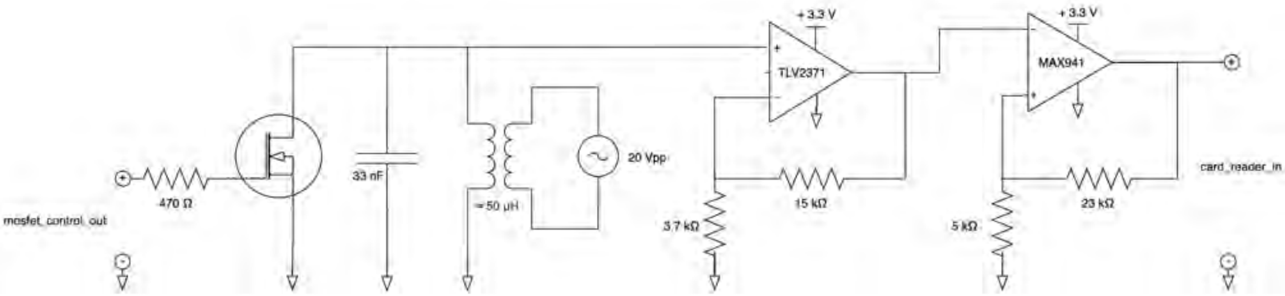

Figure 3: Analog frontend for the spoofer module. System consists of (1) a mosfet (IRF740) for modulating the incoming signal (represented by the 20 Vpp oscillatory source) (2) op-amp (TLV2371) to ensure the signal will sizable relative to the 3.3 V rails (3) comparator with hysteresis (MAX941) to generate a phase-locked pulse.

due to it's higher resistance. Based on measurements, MIT card readers appear to output a 20 Vpp signal. This transfers to roughly 1-2 Vpp on our inductor coils.

4.1.2 Modulation

The modulation is accomplished by an IRF740 N-Channel mosfet in parallel with the receive coil. An additional capacitor is added in parallel to act as a low-pass filter. A rough estimate of the capacitor value can be made to ensure that the cutoff frequency $\frac{1}{2\pi\sqrt{LC}} \approx 1$ MHz. This value of cutoff frequency is above the 125 kHz necessary for the signal to exist without too much damping but is also low enough to remove the rapid periodic high-frequency oscillation in the system. Experimentally, 33 nF worked well.

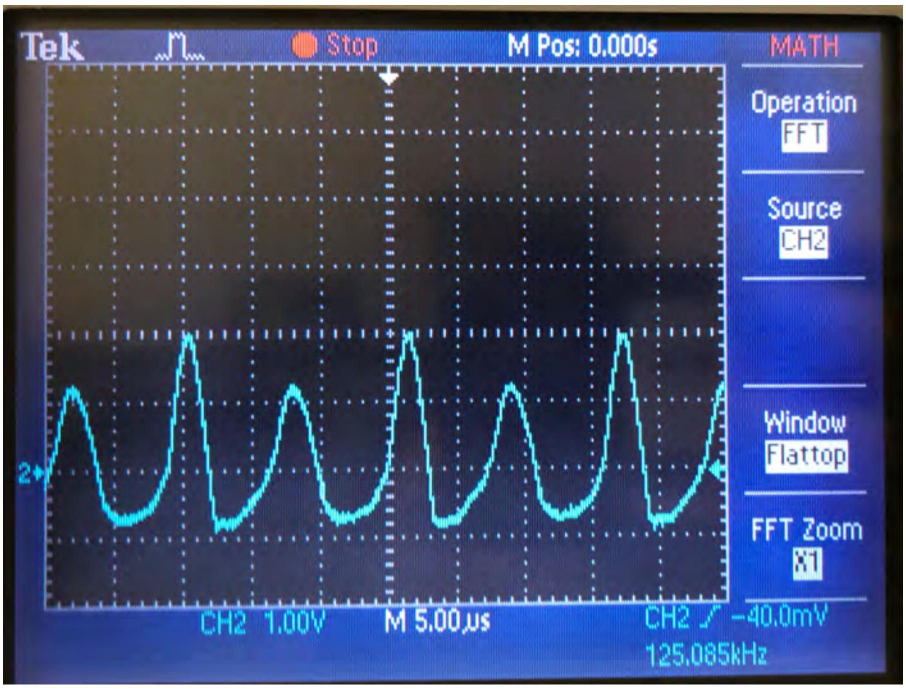

Figure 4: Clean spoofed signal. The filtering capacitor removes all high-frequency content.

4.1.3 Phase-locked Pulse

The phase-locked pulse is computed with an op-amp and comparator acting upon the receive-side coil voltage. The first iteration of the analog setup incorporated a potentiometer to set the hysteresis of the comparator. This effectively allowed for analog control of the relative phase at which the mosfet would be turned on or off. Interestingly, the spoofed signal with the greatest voltage differentiation between peaks was generated by switching the mosfet at approximately 270 degrees before the expected maximum in the signal. Figure 5 depicts all the analog signals for the basic system, which effectively corresponds to transmitting a single bit.

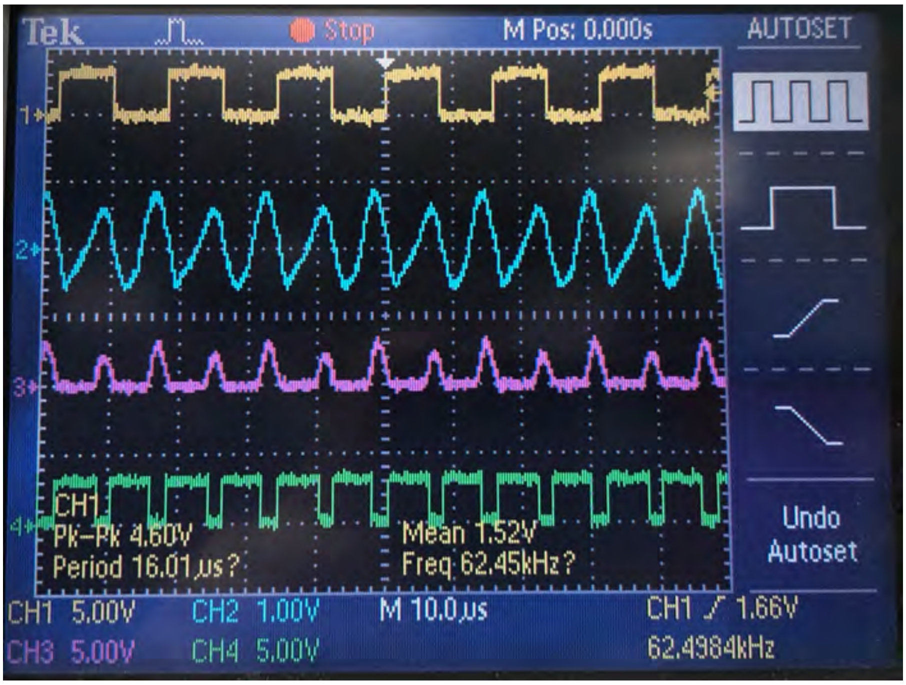

Figure 5: Oscilloscope output of all analog front-end signals. Yellow: mosfet control. Blue: spoofed signal out. Pink: op-amp output. Green: phase-locked pulse.

4.2 Spoofing Module

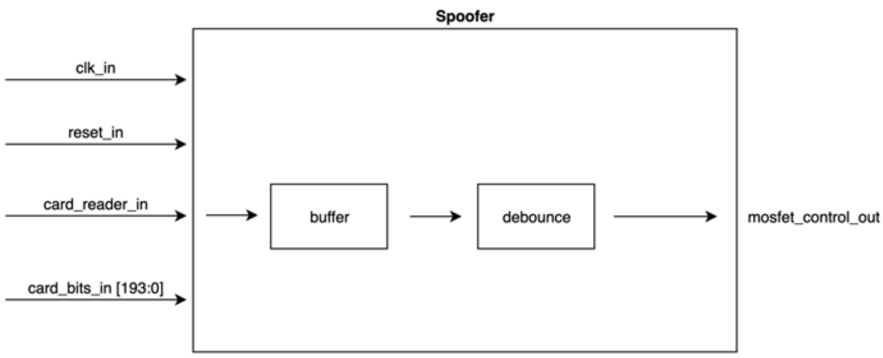

Figure 6: Spoof module block diagram.

The spoofing module accomplishes two tasks: (1) buffering and debouncing the `card_reader_in` signal so that it is reliable and (2) computing the state of `mosfet_control_out`.

4.2.1 Phase-Locked Clock

The main difficulty for the spoof module is to generate a reliable signal for `card_reader_in` so that the mosfet may use that signal as a pseudo-clock. If any edge of the analog `card_reader_in` signal is missed, the mosfet state will remain unchanged, causing the entire rest of the transmission to be faulty.

The most significant improvement in error rate came from a 3 wide buffer in of the `card_reader_in` signal. The remaining errors were improved by debouncing. With the module's actual logic clock of 100 MHz, each clock cycle is 10 ns. Thus, an individual high or low peak of the 125 kHz signal lasts for 400 clock cycles. Experimentally, debouncing over 5 clock cycles led to a near-zero error rate in the signal. Anything much more than that was too stringent of a condition and resulted in missed edges.

Debugging of this module with the ILA was crucial: (1) the testbench simulations were correct without either of these features implemented and (2) the oscilloscope could not pick up on the instabilities in the FPGA from the noisy signal that caused it to fail. Of great significance was the ability to select a number of frames for the ILA to trigger on (at the small cost of fewer data points per window).

4.2.2 Mosfet Control Logic

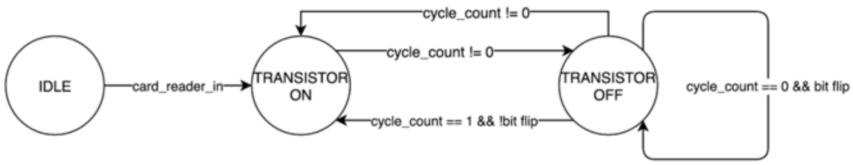

Figure 7: FSM for spoof module logic.

The mosfet control logic first alternates the mosfet on and off for 32 cycles of a 125 kHz wave. Since the spoof module should cyclically spoof the given bit stream, a pointer is used to maintain the location of the currently transmitted bit. After the 32 cycles, the module checks whether a bit shift occurs in the bit stream and flips the mosfet state accordingly. If no bit shift occurs then the on off behavior continues as before. Otherwise, the bit shift is represented by maintaining the current state of the mosfet for an additional cycle.

5 Reading (Miles)

Reading the stored data on the card requires first energizing the ID card with a 125 kHz signal. This is picked up by a coil embedded in the card which is attached to an integrated circuit that is able to change the impedance across the coil. In order to create a reader, we use a signal generator connected to a transmit coil made up of 15-20 turns of 22 AWG wire in an approximately 3-inch diameter loop to simulate a card reader. Another identical coil was made to act as the receive coil. The voltage across this shows the signal being sent from the card.

5.1 Analog Frontend

The most straightforward way of capturing the changing voltage on the receive coil is to feed the signal into the on-board ADC. However, the ADC has a maximum sampling rate of 1 Mbps. The incoming carrier wave is at 125 kHz which corresponds to 8 samples per cycle. This is insufficient to determine the small differences in amplitude required to decode the modulation.

Instead, an analog frontend is used to preprocess this signal. First, a notch filter tuned to 125 kHz increases the amplitude difference between consecutive peaks. This signal is then amplified, and a comparator is set to trigger on each high peak. The comparator pulses can be polled by a digital pin at 100 MHz and phase shifts can be determined by the timing between the pulses.

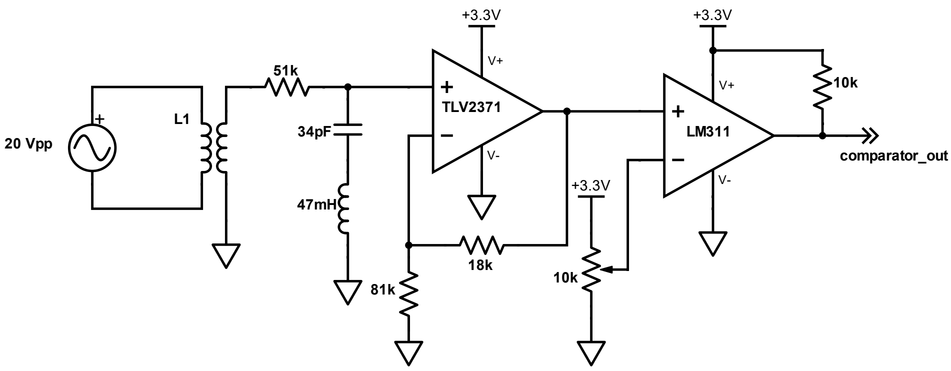

Figure 8: Analog frontend for the reader module. The transmit coil is represented by the primary of L1 and the receive coil is the secondary.

Because we are also using a hand-wound coil to spoof the signal, there is very different coupling between the spoof coil and the coil in the ID card. As a result, we need to adjust the potentiometer to change the comparator threshold since the voltage on the recieve coil will vary depending on the coupling.

5.2 Reading Module

The job of the Reading Module is to accept the raw input from the analog frontend and output the 194-bit ID number that is stored on the card. The reader should signal to the next module when the ID is ready to be read with a flag.

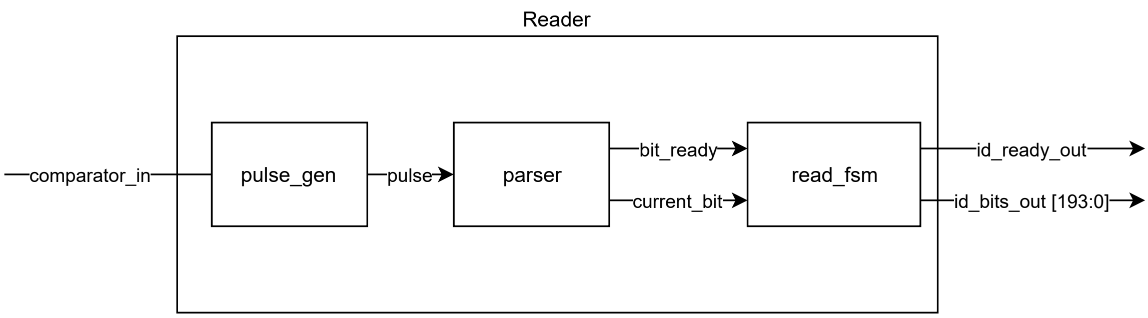

Figure 9: Reader module block diagram.

5.2.1 Pulse Gen Module

The goal of the pulse_gen module is to synchronize the pulses from the LM311 comparator and convert them into pulses of one clock cycle for the downstream modules.

Because this is the first module to receive the raw input, it needs to make special considerations for the analog nature of the signal. Even though the comparator is fast, it still has a fall time[1] the of about 100 ns. With the FPGA's 100 MHz clock, this is still around 10 clock cycles in which the comparator is at an intermediate voltage. As a result, it is necessary for this module to prevent metastability problems. This is done by shifting the data through a four-stage shift register. The pulse_gen module only outputs a pulse when it detects that all four values in the shift register agree on the logic level.

5.2.2 Parser Module

The role of the parser module is to decode the pulses received from the comparator into bits. The integrated circuit in the ID card transmits one bit every 16 cycles of the 62.5 kHz wave. If the current bit to be sent differs from the last bit (i.e. there is a bit flip), then the IC will cause a phase shift in the 62.5 kHz signal which looks like two consecutive high or low peaks in the 125 kHz carrier. The parser counts the clock cycles between pulses to determine if a phase shift has occurred (i.e. the duration between the pulses is either too high or too low). If so, it flips the value on `current_bit`. Every 16 cycles, it raises the `bit_ready` line to signal that a bit has been sent. Subsequent modules then know that a valid bit is visible on `current_bit`.

5.2.3 Read FSM Module

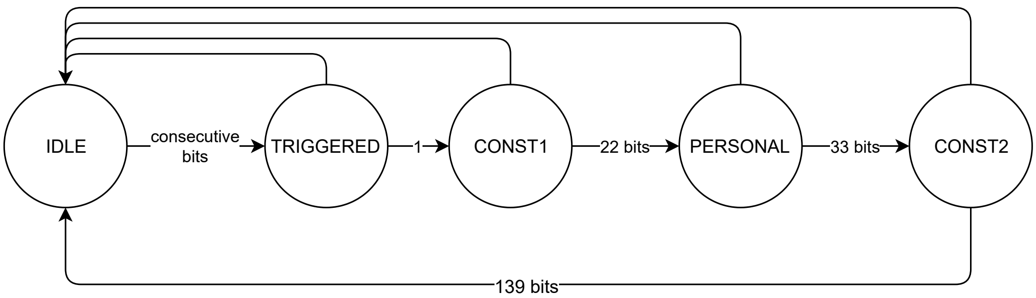

Figure 10: ID Bits decode FSM.

The read FSM represents the highest level of abstraction in decoding the ID number. This module effectively receives a stream of bits from the parser and needs to figure out where in the transmitted ID it is. A finite state machine accepts the incoming bits and uses the known structure of the data packet to parse the bits.

In IDLE state, the module waits until it detects a consecutive sequence of 27 transmissions of the same bit. This matches up with the 30 zeros sent out at the beginning and indicates to the FSM that an ID transition is about to begin. After 27 zeros or ones are detected, the system waits for the first one. The first one indicates the start of the first group of constant bits. The state machine starts recording bits at this point and raises the `id_ready_out` flag once it transitions from CONST2 back to IDLE.

Another benefit of knowing the structure of the data is that it allows the FSM to check the incoming bits for known segments of data. For example, it is known that the first 22 bits sent by every MIT card is 1000001110011000010110. We can use this in the state machine to reject data packets that do not match this sequence since noise in the system can cause spurious bit flips. Only after the entire 194-bit bus is filled does the FSM raise the `id_ready_out` flag to signal to the downstream module that a valid ID has been loaded.

[1]We use the falling edge of the comparator because the LM311 is an open collector device. Thus it relies on the 10k pullup resistor shown in Figure 8 to create the rising edge which makes it much slower than the falling edge.

6 Graphical User Interface (Hannah)

The goal of the GUI module is to provide a way for users to interact with the RFID system. The user may switch between SPOOF and READ mode via the use of a sw[15].

6.1 Spoof GUI

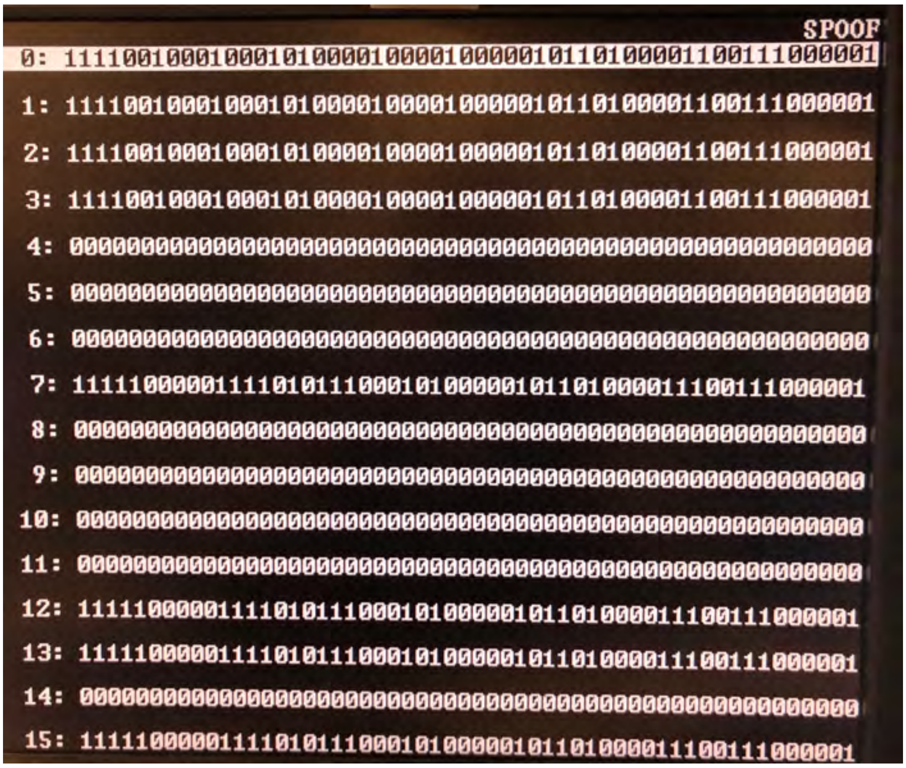

Figure 11: GUI display for the spoof module.

In SPOOF mode, users are provided with a glance of the first 55 bits (personal + MIT bits) stored in the 16 BRAM locations shared between the READ and SPOOF modules. Using the up and down buttons, they may scroll through the contents. The selected entry is automatically spoofed.

6.2 Read GUI

In READ mode, users are provided with a live update of the most recent bits read from the Reading FSM. Furthermore, since the read system requires tuning of a physical potentiometer, this mode provides feedback such as "MIT ID" or "ID Not Recognized" so that the user may be able to tune the hardware without an external oscilloscope. This feedback is given by checking whether the first 22 bits match the 22 MIT bits hardcoded into the module itself. Finally, users may use the sw[4:0] to select a location in BRAM to which they may save the entire 194 long sequence of bits.

6.3 Implementation Details

The main ability to display text on the screen is provided by the module `cstringdisplay.v` written by I. Chuang and C. Terman. This module takes in the ascii code for up to 64 characters to be displayed in a single line on the display. The font is provided by a COE file.

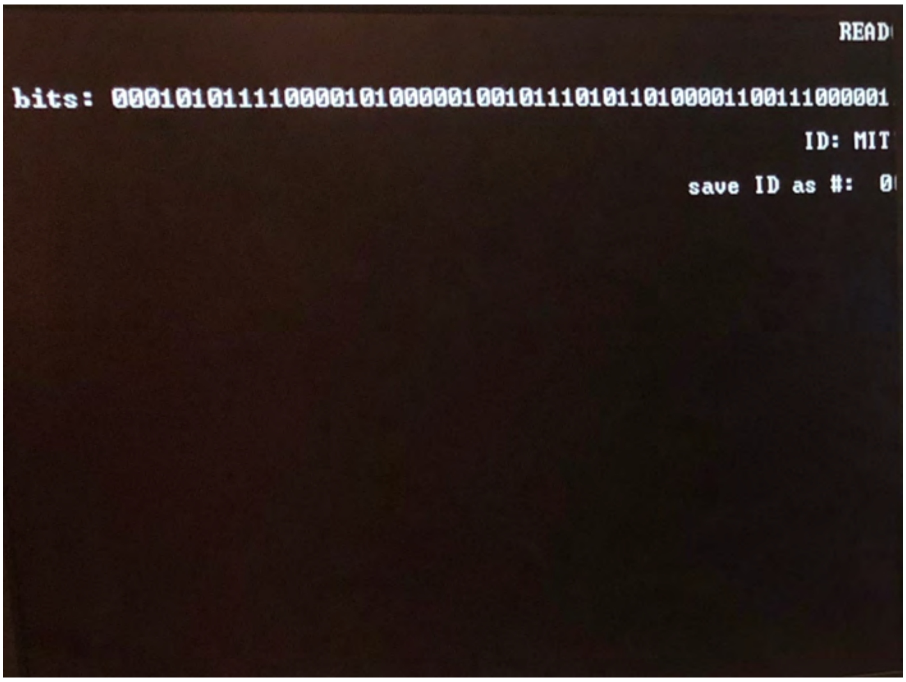

Figure 12: GUI display for the read module.

The module `bits_to_ascii.sv` adds the ability to convert arbitrary sequences of bits to ascii code so that it may be displayed on the screen in the READ and SPOOF module. Other text on the screen uses hard-coded ascii valuese.

7 Portable System (Miles)

One of the goals for this project was to create a device that could be held up to a card reader to spoof an RFID card. This required portability which was not easy with the Nexys A7 DDR boards. Instead, we switched to a CMOD A7 board from Digilent which contains an Artix-7 FPGA in a more portable form factor. This allowed us to power the entire system from a portable power bank and bring the device to a wall-mounted card reader.

Most of the code from the main project was transferred directly to the CMOD A7 with the exception of the graphics handlers. The only other major modification made was to change some of the constants since the Artix-7 uses a 20 MHz clock. There are fewer clock cycles between comparator pulses, so some counting constants need to be changed.

7.1 User Interface

The CMOD A7 board is much more limited in user interface components. For our project, in order to allow the user to select between the read and spoof modes of operation, we installed a row of DIP switches with pullup resistors which could be connected to the GPIO pins. We also took advantage of the two onboard LED lights and pushbuttons to indicate when to record an id number and when the ID number read matched the MIT constant.

7.2 Flash Memory

The volatile nature of the FPGA causes challenges when making this device portable. Because the programmed bitstream disappears on each reboot, it is necessary to write the desired bitstream into non-volatile flash memory so that it can be loaded into the FPGA during start up. We were able to take advantage of the CMOD A7's onboard Quad SPI Flash memory. We were able to follow detailed instructions[2] to generate a binary file that could be written to the flash memory. Then, on each power up, the FPGA looks first for stored bitstreams in the flash memory.

8 Challenges

8.1 Debugging

Because this project relies on interfacing between the digital logic and the analog frontend, the integrated logic analyzer (ILA) proved invaluable. However, one challenge was the difficulty in debugging some of the larger state machines. The ILA has limited memory and the data packets takes tens of milliseconds to send. There was a lot of dead time between sending every two bits that was taking up ILA memory depth. We were able to use more complex triggering and the multiple sample features to trigger every time a bit is sent and and to only capture a few samples after each trigger since it only takes one clock cycle to send a bit.

At a higher level, another challenge with debugging was the ability to verify correctness of the bits that we were reading off the cards. Because the ID card is a black box system, we had no way of knowing if we were reading or spoofing the bits correctly. We had to carefully verify each module we built and trust that it was working correctly to debug downstream modules. The only confirmation we would get is when we were able to use the system to open a door successfully.

9 Future Work

9.1 Security

Since its use as an authentication mechanism, RFID has received criticism for its security. Some additional security concerns specific to the MIT system reside in the generation of the personal bits. Additional analysis can be done to see if there are any patterns or ranges that the bits span. This would potentially narrow the scope for a brute force attack on the card readers.

Because the RFID cards use static passwords, they are vulnerable to the exact kind of spoofing replay attack we have demonstrated in this project. One improvement we can use to mitigate this attack vector is the use of a one-time password (OTP). Because the card is a passive system, it would first have to communicate with the card reader to generate the OTP. With our system, we can implement a handshake and acknowledgement protocol and have the card FPGA calculate a hash function with a private key along with the acknowledgement data to generate a unique password per session.

10 Appendix

10.1 Top Level

10.1.1 top_level.sv

```
module top_level(
    input clk_100mhz,
```

[2]https://reference.digilentinc.com/learn/programmable-logic/tutorials/cmod-a7-programming-guide/start

```
    input [1:0] ja,
    output logic[0:0] jb,
    input [15:0] sw,
    logic btnc, // reset
    logic btnd, // record
    logic btnu, btnl, btnr, // unassigned
    output[3:0] vga_r,
    output[3:0] vga_b,
    output[3:0] vga_g,
    output vga_hs,
    output vga_vs,
    output logic[15:0] led
    );

    logic record_btn;
    assign record_btn = btnd;

    // Declare BRAM
    logic [3:0] addr;
    logic [193:0] data_to_bram;
    logic [193:0] data_from_bram;
    logic bram_write;
    blk_mem_gen_0 bit_bram(.addra(addr), .clka(clk_100mhz),
              .dina(data_to_bram),
              .douta(data_from_bram),
              .ena(1), .wea(bram_write));

    // Reader
    logic id_ready;
    logic [193:0] id_bits;
    reader card_reader(.comparator_in(ja[0]),
                       .clk_in(clk_100mhz),
                       .reset_in(sw[15]), // activate reader when sw[15] is
   low
                       .id_bits_out(id_bits),
                       .id_ready_out(id_ready));

//     record id_recorder(.addr(addr),
//                         .record_in(record_btn),
//                         .id_bits_in(id_bits),
//                         .id_ready_in(id_ready),
//                         .clk_in(clk_100mhz),
//                         .reset_in(sw[15]),
//                         .data_to_bram_out(data_to_bram),
//                         .bram_write_out(bram_write));
    // Spoofer
    logic [193:0] bits_to_spoof;
    spoofer id_spoofer( .card_reader_in(ja[1]),
                        .card_bits_in(bits_to_spoof),
                        .clk_in(clk_100mhz),
                        .reset_in(!sw[15]), // activate spoofer when sw[15]
   is high
                        .mosfet_control_out(jb[0]));
    //////////////////////////////////////////////////////////
    // GUI
    // create 65mhz system clock, happens to match 1024 x 768 XVGA timing
    wire clk_65mhz;
    clk_wiz_0 clkdivider(.clk_in1(clk_100mhz), .clk_out1(clk_65mhz), .reset
   (0));

```

```
wire [10:0] hcount;    // pixel on current line
wire [9:0] vcount;     // line number
wire hsync, vsync;
wire [11:0] pixel;
reg [11:0] rgb;
wire blank;
xvga xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
      .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));


// btnc button is user reset
logic reset;
debounce_65mhz db1(.reset_in(0),.clock_in(clk_65mhz),.noisy_in(btnc),.clean_out(reset));

// UP and DOWN and LEFT and RIGHT for menu interface
wire up,down,left,right;
debounce_65mhz db2(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(btnu),.clean_out(up));
debounce_65mhz db3(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(btnd),.clean_out(down));
debounce_65mhz db4(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(btnl),.clean_out(left));
debounce_65mhz db5(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(btnr),.clean_out(right));

wire spoof_switch;
debounce_65mhz db6(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(sw[15]),.clean_out(spoof_switch));

wire phsync,pvsync,pblank;
rfid_gui gui(.vclock_in(clk_65mhz),.reset_in(reset),
             .up_in(up),.down_in(down),.left_in(left), .right_in(right),.is_spoof_display(spoof_switch),
             .hcount_in(hcount),.vcount_in(vcount),
             .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
             .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_out(pixel),
             .save_addr(sw[3:0]),
             .read_module_bits(id_bits),
             .data_to_bram(data_to_bram),
             .data_from_bram(data_from_bram),
             .addr(addr),
             .write_to_bram(bram_write),
             .bits_to_spoof_out(bits_to_spoof));

reg b,hs,vs;
always_ff @(posedge clk_65mhz) begin
     hs <= phsync;
     vs <= pvsync;
     b <= pblank;
     rgb <= pixel;
end

// the following lines are required for the Nexys4 VGA circuit - do not change
assign vga_r = ~b ? rgb[11:8]: 0;
assign vga_g = ~b ? rgb[7:4] : 0;
assign vga_b = ~b ? rgb[3:0] : 0;
```

```
    assign vga_hs = ~hs;
    assign vga_vs = ~vs;

    // Debug
    assign led[15:0] = sw[14] ? data_from_bram[38:23] : bits_to_spoof
   [45:30];

endmodule
```

10.2 GUI

10.2.1 rfid_gui.sv

```
//
   ///////////////////////////////////////////////////////////////////////////////
//
// RFID GUI
//
// sw[15] == 1 for SPOOF; 0 for READ
//
   ///////////////////////////////////////////////////////////////////////////////

module rfid_gui (
   input vclock_in,        // 65MHz clock
   input reset_in,         // 1 to initialize module
   input up_in,            //
   input down_in,          //
   input left_in,
   input right_in,
   input is_spoof_display,  //connected to sw[15]
   input [3:0] save_addr,         //connected to sw[3:0], location to save
   bits to in bram
   input [193:0] read_module_bits,

   input [10:0] hcount_in, // horizontal index of current pixel (0..1023)
   input [9:0]  vcount_in, // vertical index of current pixel (0..767)
   input hsync_in,         // XVGA horizontal sync signal (active low)
   input vsync_in,         // XVGA vertical sync signal (active low)
   input blank_in,         // XVGA blanking (1 means output black pixel)

   output phsync_out,       // pong game's horizontal sync
   output pvsync_out,       // pong game's vertical sync
   output pblank_out,       // pong game's blanking
   output logic [11:0] pixel_out,  // pong game's pixel  // r=11:8, g=7:4, b
   =3:0
   input [193:0] data_from_bram,
   output logic [193:0] data_to_bram,
   output logic [3:0] addr,
   output logic write_to_bram,
   output logic [193:0] bits_to_spoof_out
   );

   assign phsync_out = hsync_in;
   assign pvsync_out = vsync_in;
   assign pblank_out = blank_in;

   /////// BIT CODE THINGS /////////
   parameter MIT_BITS = 22'b0110100001100111000001;
```

```
    ///////
    /////// Pulse UI Buttons //////////
    logic up_pulse;
    pulse_65mhz my_up_pulse (.clock_in(vclock_in), .signal_in(up_in), .
    pulse_out(up_pulse));
    logic down_pulse;
    pulse_65mhz my_down_pulse (.clock_in(vclock_in), .signal_in(down_in), .
    pulse_out(down_pulse));
    logic left_pulse;
    pulse_65mhz my_left_pulse (.clock_in(vclock_in), .signal_in(left_in), .
    pulse_out(left_pulse));
    logic right_pulse;
    pulse_65mhz my_right_pulse (.clock_in(vclock_in), .signal_in(right_in), .
    pulse_out(right_pulse));
    /////////////////////////////////////////

     parameter BITS_IN_BRAM = 194;
     logic [BITS_IN_BRAM-1:0] ascii_module_in;
     logic [8*BITS_IN_BRAM-1: 0] ascii_module_out;
     bits_to_ascii my_bits_to_ascii(.bits_in(ascii_module_in), .ascii_out(
    ascii_module_out));


    ///// SWITCHES TO NUMBER ////
    logic [15:0] save_addr_ascii;

    ////// FONT MODULE ///////
    logic [64*8-1:0] char_string;
    logic [10:0] string_start_x;
    logic [9:0] string_start_y;
    logic [6:0] line_number; //can fit 32 lines on the screen but line_number
     will count up to the blank vsync interval
    assign line_number = vcount_in/24; //text heigh is 24 pixels so line
    number is module 24
    assign string_start_y = line_number * 24;
    logic coe_pixel_out;
    parameter CHARS_PER_LINE = 64; //at most 64 characters per line
    char_string_display read_spoof_display(.vclock(vclock_in), .hcount(
    hcount_in), .vcount(vcount_in), .pixel(coe_pixel_out), .cstring(
    char_string), .cx(string_start_x), .cy(string_start_y));
/////////////////////////////////////////////////////////////////

///////////////////////////
////// ASCII
/////////////////////////////////
     parameter SPOOF_ASCII = 40'b0101001101010000010011110100111101000110;
     parameter READ_ASCII = 32'b01010010010001010100000101000100;
     parameter ASCII_0 = 8'b00110000;
     parameter ASCII_1 = 8'b00110001;
     parameter ASCII_SPACE = 8'b00100000;
     parameter ASCII_COLON = 8'b00111010;
     parameter ASCII_BITS_TEXT = 48'
    b011000100110100101110100011100110011101000100000; // "bits: "
     parameter ASCII_ID = 32'b01001001010001000011101000100000;//"ID: "
     parameter ASCII_NOT_REC = 112'
    b0110111001101111011101000010000001110010011001010110001101101111011001110110111001
    ; //"not recognized"
     parameter ASCII_MIT = 24'b010011010100100101010100;// "MIT"
     parameter ASCII_POUND = 16'b0010001100100000; //"# "
```

```
    parameter ASCII_EMPTY = 40'b0110010101101101011100000111010001111001; //
   "empty"
    parameter ASCII_SAVE_AS = 112'
   b0111001101100001011101100110010100100000010010010100010000100000011000010111001100
   ; //"save ID as #: "
/////////////////////////////////

    //logic old_hsync_in;

    logic [3:0] selected_id; // 16 IDs in bram to select from using up and
   down arrow keys
    logic [3:0] displayed_id;
    logic verbose_mode;

    always_ff @ (posedge vclock_in) begin

    /// CLOCKED SPOOF LOGIC
    if (is_spoof_display == 1) begin
            write_to_bram <= 0;
            ascii_module_in <= data_from_bram;
            addr <= displayed_id; //will change as hcount and vcount change

            if (displayed_id == selected_id) begin
                bits_to_spoof_out <= data_from_bram;
            end
            if (reset_in) begin
                string_start_x <= 0;
                selected_id <= 0;
                verbose_mode <= 0;

            // keep track of state of selected ID
            end else if (up_pulse) begin
                if (selected_id > 0) selected_id <= selected_id - 1;
            end else if (down_pulse) begin
                if (selected_id < 15) selected_id <= selected_id + 1;

            end else if (right_pulse) begin
                verbose_mode <= 1;
            end else if (left_pulse) begin
                verbose_mode <= 0;
            end

            if (line_number == 0) begin
                pixel_out = 12'hFFF*coe_pixel_out;
            end else if ((2*selected_id+1) == line_number) begin
                pixel_out = ~(12'hFFF*coe_pixel_out);
            end
            else begin
                pixel_out = 12'hFFF*coe_pixel_out;
            end

      ///// CLOCKED READ LOGIC

      end else begin
           ascii_module_in <= read_module_bits;
           addr <= save_addr;
           //save_addr switches to ascii
           case(save_addr)
                4'b0000: save_addr_ascii <= ASCII_0+8'd0;
                4'b0001: save_addr_ascii <= ASCII_0+8'd1;
```

```
                4'b0010: save_addr_ascii <= ASCII_0+8'd2;
                4'b0011: save_addr_ascii <= ASCII_0+8'd3;
                4'b0100: save_addr_ascii <= ASCII_0+8'd4;
                4'b0101: save_addr_ascii <= ASCII_0+8'd5;
                4'b0110: save_addr_ascii <= ASCII_0+8'd6;
                4'b0111: save_addr_ascii <= ASCII_0+8'd7;
                4'b1000: save_addr_ascii <= ASCII_0+8'd8;
                4'b1001: save_addr_ascii <= ASCII_0+8'd9;
                4'b1010: save_addr_ascii <= {ASCII_1,ASCII_0+8'd0};
                4'b1011: save_addr_ascii <= {ASCII_1,ASCII_0+8'd1};
                4'b1100: save_addr_ascii <= {ASCII_1,ASCII_0+8'd2};
                4'b1101: save_addr_ascii <= {ASCII_1,ASCII_0+8'd3};
                4'b1110: save_addr_ascii <= {ASCII_1,ASCII_0+8'd4};
                4'b1111: save_addr_ascii <= {ASCII_1,ASCII_0+8'd5};
            endcase

          if (right_pulse == 1) begin //write to bram
               write_to_bram <= 1;
          end else begin //don't write to bram, just display
               write_to_bram <= 0;
              if ((line_number == 7) & (right_in == 1)) begin
                           pixel_out = ~(12'hFFF*coe_pixel_out);
              end else begin
                 pixel_out = 12'hFFF*coe_pixel_out;
              end
         end

     end

 end

 // combinational logic determines
 // 1) based on line number, what text should be,
 // 2) what the color of pixel_out should be by feeding char_string through cstringdisplay.v to produce coe_pixel_out
 // 3) pixel_out based on coe_pixel_out
 // uses combinational logic on hcount_in and vcount_into determine what the element to be displayed is

 always_comb begin
     ////////// //SPOOF DISPLAY /////////////
     if (is_spoof_display == 1) begin
         //generate ascii for each line -- display rom text on odd lines in non verbose mode
         displayed_id = (line_number - 1) >> 1; //which ID is currently displayed


         //ascii_module

         //ascii string to display
         case (line_number)
             0: char_string = SPOOF_ASCII;
             1: char_string  = {ASCII_0,ASCII_COLON,ASCII_SPACE, ascii_module_out[55*8-1:0]};
             3: char_string  = {ASCII_0+8'd1,ASCII_COLON,ASCII_SPACE, ascii_module_out[55*8-1:0]};
             5: char_string  = {ASCII_0+8'd2,ASCII_COLON,ASCII_SPACE, ascii_module_out[55*8-1:0]};
             7: char_string  = {ASCII_0+8'd3,ASCII_COLON,ASCII_SPACE,
```

```
ascii_module_out[55*8-1:0]};
                9: char_string  = {ASCII_0+8'd4,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                11: char_string  = {ASCII_0+8'd5,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                13: char_string  = {ASCII_0+8'd6,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                15: char_string  = {ASCII_0+8'd7,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                17: char_string  = {ASCII_0+8'd8,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                19: char_string  = {ASCII_0+8'd9,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                21: char_string  = {ASCII_1,ASCII_0,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                23: char_string  = {ASCII_1,ASCII_0+8'd1,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                25: char_string  = {ASCII_1,ASCII_0+8'd2,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                27: char_string  = {ASCII_1,ASCII_0+8'd3,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                29: char_string  = {ASCII_1,ASCII_0+8'd4,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                31: char_string  = {ASCII_1,ASCII_0+8'd5,ASCII_COLON,ASCII_SPACE,ascii_module_out[55*8-1:0]};
                default: char_string  = 0;
            endcase

        end


        ////////is_spoof_display==0 for READ DISPLAY//////////
        else begin

            //ascii

            data_to_bram = read_module_bits;
        //// display
             case (line_number)
                0: char_string = READ_ASCII;
                3: char_string = {ASCII_BITS_TEXT, ascii_module_out[55*8-1:0]}; //print out 22 MIT and 33 personal from left to right
                5: begin
                    if (read_module_bits[21:0] == MIT_BITS) begin
                        char_string = {ASCII_ID, ASCII_MIT}; //add information about whether the ID is mit, unidentified or stored in the bram
                    end else begin
                        char_string = {ASCII_ID, ASCII_NOT_REC};
                    end
                end
                7: char_string = {ASCII_SAVE_AS, save_addr_ascii};
                default: char_string = 0;
            endcase
        end


    end


```

```
    //ila_0 myila(.clk(vclock_in), .probe0(hsync_in), .probe1(vsync_in), .
    probe2(hcount_in), .probe3(pixel_out), .probe4(puck_center_x), .probe5(
    puck_center_y));

    endmodule

    module synchronize #(parameter NSYNC = 3)  // number of sync flops.
    must be >= 2
                       (input clk,in,
                        output reg out);

    reg [NSYNC-2:0] sync;

    always_ff @ (posedge clk)
    begin
        {out,sync} <= {sync[NSYNC-2:0],in};
    end
endmodule

//
    ///////////////////////////////////////////////////////////////////////////////

//
// Rising Edge Pulse
//
//
    ///////////////////////////////////////////////////////////////////////////////

module pulse_65mhz (input clock_in, input signal_in, output pulse_out);

    logic old_signal_in;
    assign pulse_out = (old_signal_in == 0) & (signal_in == 1);
    always_ff @(posedge clock_in) begin
        old_signal_in <= signal_in;
    end
endmodule



//
    ///////////////////////////////////////////////////////////////////////////////

//
// Pushbutton Debounce Module (video version - 24 bits)
//
//
    ///////////////////////////////////////////////////////////////////////////////

module debounce_65mhz (input reset_in, clock_in, noisy_in,
                    output reg clean_out);

   reg [19:0] count;
   reg new_input;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
```

```
        count <= 0; end
      else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
      else if (count == 1000000) clean_out <= new_input;
      else count <= count+1;


endmodule

// ///////////////////////////////////////////////////////////////////////////////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                               ---- HORIZONTAL -----     ------VERTICAL -----
//                               Active                    Active
//                     Freq      Video   FP  Sync   BP      Video   FP  Sync  BP
//   640x480, 60Hz     25.175    640     16    96   48       480    11    2   31
//   800x600, 60Hz     40.000    800     40   128   88       600     1    4   23
//   1024x768, 60Hz    65.000    1024    24   136  160       768     3    6   29
//   1280x1024, 60Hz   108.00    1280    48   112  248       768     1    3   38
//   1280x720p 60Hz    75.25     1280    72    80  216       720     3    5   30
//   1920x1080 60Hz    148.5     1920    88    44  148      1080     4    5   36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
// ///////////////////////////////////////////////////////////////////////////////

module xvga(input vclock_in,
            output reg [10:0] hcount_out,    // pixel number on current line
            output reg [9:0] vcount_out,     // line number
            output reg vsync_out, hsync_out,
            output reg blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

```

```
    // horizontal: 1344 pixels total
    // display 1024 pixels per line
    reg hblank,vblank;
    wire hsyncon,hsyncoff,hreset,hblankon;
    assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
    assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
    assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE -
    1));  // 1183
    assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE +
    H_BP - 1));  //1343

    // vertical: 806 lines total
    // display 768 lines
    wire vsyncon,vsyncoff,vreset,vblankon;
    assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
    assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));
    // 771
    assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
    V_SYNC_PULSE - 1));  // 777
    assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
    V_SYNC_PULSE + V_BP - 1)); // 805

    // sync and blanking
    wire next_hblank,next_vblank;
    assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
    assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
    always_ff @(posedge vclock_in) begin
       hcount_out <= hreset ? 0 : hcount_out + 1;
       hblank <= next_hblank;
       hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

       vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
       vblank <= next_vblank;
       vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

       blank_out <= next_vblank | (next_hblank & ~hreset);
    end
endmodule
```

10.2.2 bits_to_ascii.sv

```
//converts entire 194 bits into ascii. later on, can index in
module bits_to_ascii(
    input  logic [193:0] bits_in,
    output logic [8*194-1:0] ascii_out
    );

    parameter ASCII_0 = 8'b00110000;
    parameter ASCII_1 = 8'b00110001;
    parameter MAX_BITS = 194; //at most 64 characters per line

    always @ (*) begin
        for (int n=0 ; n< MAX_BITS ; n++) begin
            ascii_out[8*n +: 8] <= (bits_in[n] == 1) ? ASCII_1 : ASCII_0;
         end
    end

endmodule
```

10.2.3 cstringdisplay.v

```
//
// File:   cstringdisp.v
// Date:   24-Oct-05
// Author: I. Chuang, C. Terman
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//   vclock       - video pixel clock
//   hcount       - horizontal (x) location of current pixel
//   vcount       - vertical (y) location of current pixel
//   cstring      - character string to display (8 bit ASCII for each char)
//   cx,cy        - pixel location (upper left corner) to display string at
//
// OUTPUT:
//
//   pixel        - video pixel value to display at current location
//
// PARAMETERS:
//
//   NCHAR        - number of characters in string to display
//   NCHAR_BITS   - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.


// ////////////////////////////////////////////////////////////////////////////
//
// video character string display
//
// ////////////////////////////////////////////////////////////////////////////

module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy);

   parameter NCHAR = 64;  // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 6; // number of bits in NCHAR

   input vclock;  // 65MHz clock
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   output [2:0] pixel;  // char display's pixel
   input [NCHAR*8-1:0] cstring; // character string to display
   input [10:0] cx;
   input [9:0]  cy;
```

```
  // 1 line x 8 character display (8 x 12 pixel-sized characters)

  wire [10:0]  hoff = (hcount+2)-1-cx; //"prefetch" 2 clock cycles
  wire [9:0]   voff = vcount-cy;
  wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // <
  NCHAR
  wire [2:0]   h = hoff[3:1];              // 0 .. 7
  wire [3:0]   v = voff[4:1];        // 0 .. 11

  // look up character to display (from character string)
  reg [7:0]  char;
  integer  n;
  always @(*)
    for (n=0 ; n<8 ; n = n+1 )   // 8 bits per character (ASCII)
      char[n] <= cstring[column*8+n];

  // look up raster row from font rom
  wire reverse = char[7];
  wire [10:0] font_addr = char[6:0]*12 + v;    // 12 bytes per character
  wire [7:0]  font_byte;
  font_rom f(
       .clka(vclock),
       .addra(font_addr),
       .douta(font_byte));

  // generate character pixel if we're in the right h,v area
  wire [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
  wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
       & (vcount < cy + 24));
  wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule
```

10.3 Spoofing

10.3.1 spoofer.sv

```
module spoofer(
        input clk_in,
        input reset_in, //center button for reset
        input card_reader_in, // ready-signal from incoming 125kHz wave
        input [193:0] card_bits_in, // data bits to spoof ([193:0])
        output logic mosfet_control_out // mosfet state
    );

    //mosfet output signal
    logic spoof_out; // mosfet state
    assign mosfet_control_out = spoof_out;

    parameter NUM_BITS = 224;
    logic[NUM_BITS-1: 0] data_in;
    assign data_in = {30'b0, card_bits_in}; // prepend 30 zeros to the front
    of the data bits
    logic[NUM_BITS-1:0] cyclic_data_in;

    logic [4:0] cycles_per_bit_count; //5 bits so 32 cycles per bit

    logic [3:0] card_reader_buffer; //incoming data
    logic card_reader_noisy; //buffered
    logic card_reader_clean; //buffered and debounced
```

```
    debounce card_reader_noisy_debounce (.reset_in(reset_in), .clock_in(clk_in), .noisy_in(card_reader_noisy) ,
                  .clean_out(card_reader_clean));
    logic card_reader_pulse; //true on rising edge
    pulse my_card_reader_pulse (.clock(clk_in), .signal(card_reader_clean), .pulsed_signal(card_reader_pulse)); //pulse the clean debounced signal

    logic [7:0] current_bit_loc;
    parameter MAX_LOC = 223;
    logic currentBit; //MSB of cyclic_data_in
    logic previousBit; //previous MSB of cyclic_data_in
    assign currentBit = data_in[current_bit_loc];

    always_ff @(posedge clk_in) begin
        if (reset_in) begin //reset and initialize
            spoof_out <= 0;
            cycles_per_bit_count <= 1;
            previousBit <= currentBit;
            card_reader_buffer <= 4'b0;
            card_reader_noisy <= 0;
            current_bit_loc <= 0;
        end

        else begin
            if (card_reader_pulse) begin //determine spoof_out
                if (cycles_per_bit_count == 0) begin //move to next bit

                    //if bit flip from previous to current, phase shift implies spoof_out remains the same
                    spoof_out <= (previousBit != currentBit) ? spoof_out : !spoof_out;

                    //get the next bit and bit shift cyclic_data_in
                    previousBit <= currentBit;
                    current_bit_loc <= (current_bit_loc == MAX_LOC) ? 0: current_bit_loc + 1; //bit shifting cyclic_data_in will pop the MSB
                end

                else begin
                    spoof_out <= !spoof_out;
                end

                cycles_per_bit_count <= cycles_per_bit_count + 1;
            end

            card_reader_noisy <= (card_reader_buffer >> 3);
            card_reader_buffer <= (card_reader_buffer << 1) + card_reader_in;
        end
    end

endmodule


//module spoof_module(
//        input clk_100mhz,
//        input logic btnc, //center button for reset
//        input logic [0:0] ja, //card_reader_in:  ready-signal from incoming 125kHz wave
//        output logic [0:0] jb //spoof_out:  mosfet state
```

```
//     );


//     //mosfet output signal
//     logic spoof_out; // mosfet state
//     assign jb = spoof_out;

//     logic[29:0] consecutive_bits;
//     assign consecutive_bits = 30'b0;

//     logic[21:0] constant_bits;
//     assign constant_bits = 22'b1000001110011000010110;

//     logic[32:0] personal_bits;
//     assign personal_bits = 33'b100000100001000010100010001001111; //hannah
////     // 33'b10101011010100001011100001110100 //miles

//     logic[138:0] trash_bits;
//     assign trash_bits = 139'
    b01011000101011010100000110010111000101101110011000110101010111010111100100100010000
    ;

//     parameter NUM_BITS = 224;
//     logic[NUM_BITS-1: 0] data_in;
//     assign data_in = {consecutive_bits, constant_bits, personal_bits,
    trash_bits};
//     logic[NUM_BITS-1:0] cyclic_data_in;

//     logic [2:0] cycles_per_bit_count; //5 bits so 32 cycles per bit

//     logic [3:0] card_reader_buffer; //incoming data
//     logic card_reader_noisy; //buffered
//     logic card_reader_clean; //buffered and debounced
//     debounce card_reader_noisy_debounce (.reset_in(btnc), .clock_in(
    clk_100mhz), .noisy_in(card_reader_noisy) ,
//                 .clean_out(card_reader_clean));
//     logic card_reader_pulse; //true on rising edge
//     pulse my_card_reader_pulse (.clock(clk_100mhz), .signal(
    card_reader_clean), .pulsed_signal(card_reader_pulse)); //pulse the clean
     debounced signal

//     logic currentBit; //MSB of cyclic_data_in
//     logic previousBit; //previous MSB of cyclic_data_in
//     assign currentBit = btnc ? (data_in >> (NUM_BITS - 1)) : (
    cyclic_data_in >> (NUM_BITS - 1));

//     always_ff @(posedge clk_100mhz) begin
//         if (btnc) begin //reset and initialize
//             cyclic_data_in <= data_in;
//             spoof_out <= 0;
//             cycles_per_bit_count <= 1;
//             previousBit <= currentBit;
//             card_reader_buffer <= 4'b0;
//             card_reader_noisy <= 0;
//         end

//         else begin
//             if (card_reader_pulse) begin //determine spoof_out
//                 if (cycles_per_bit_count == 0) begin //move to next bit

```

```
//                      //if bit flip from previous to current, phase shift implies spoof_out remains the same
//                      spoof_out <= (previousBit != currentBit) ? spoof_out : !spoof_out;

//                      //get the next bit and bit shift cyclic_data_in
//                      previousBit <= currentBit;
//                      cyclic_data_in <= (cyclic_data_in << 1) + currentBit; //bit shifting cyclic_data_in will pop the MSB
//                  end

//                  else begin
//                      spoof_out <= !spoof_out;
//                  end

//                  cycles_per_bit_count <= cycles_per_bit_count + 1;
//              end

//              card_reader_noisy <= (card_reader_buffer >> 3);
//              card_reader_buffer <= (card_reader_buffer << 1) + ja;
//          end
//      end

//endmodule
```

10.3.2 debounce.sv

```
module pulse(
    input clock,
    input signal,
    output pulsed_signal
    );

    logic old_signal;
    assign pulsed_signal = (old_signal == 0) & (signal == 1);

    always_ff @ (posedge clock) begin
        old_signal <= signal;
    end

endmodule



module debounce (input reset_in, clock_in, noisy_in,
                 output logic clean_out);

   logic [4:0] count;
   logic new_input;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
     else if (count >= 5) clean_out <= new_input;
     else count <= count+1;

endmodule
```

10.4 Reading

```
module reader(
    input comparator_in,
    input clk_in,
    input reset_in,
    input [3:0] addr_in,
    input record_in,
    output logic [193:0] id_bits_out,
    output logic id_ready_out
);
    logic pulse;
    logic bit_ready;
    logic current_bit;
    pulse_gen comparator_cleanup(.comparator_in(comparator_in),
                                 .clk_in(clk_in),
                                 .reset_in(reset_in),
                                 .pulse_out(pulse));
    parser comparator_parser(.pulse_in(pulse),
                             .clk_in(clk_in),
                             .reset_in(reset_in),
                             .bit_ready_out(bit_ready),
                             .current_bit_out(current_bit));
    read_fsm fsm(.bit_ready_in(bit_ready),
                 .sent_bit_in(current_bit),
                 .reset_in(reset_in),
                 .clk_in(clk_in),
                 .id_out(id_bits_out),
                 .id_ready_out(id_ready_out));

endmodule

module record(
    input [3:0] addr,
    input record_in, // Signal to record
    input [193:0] id_bits_in, // id number to record
    input id_ready_in, // id number ready signal from the reader module
    input clk_in,
    input reset_in,
    output logic [193:0] data_to_bram_out,
    output logic bram_write_out
);
    // Continuously stores IDs being emitted from the reader module and
   stores in internal register
    // This allows for instantaneous recording when the button is pressed.
    parameter S_IDLE = 0;
    parameter S_RECORD = 1;

    logic state = S_IDLE;
    logic [193:0] last_valid_id_num = 0;

    always_ff @(posedge clk_in) begin
        case(state)
            S_IDLE: begin
                bram_write_out <= 0; // ensure one cycle pulse
                if(record_in) begin
                    state <= S_RECORD;
                end
                if(id_ready_in) begin
                    last_valid_id_num <= id_bits_in;
```

```
                end
            end
            S_RECORD: begin
                data_to_bram_out <= last_valid_id_num;
                bram_write_out <= 1;
                state <= S_IDLE;
            end
            default: begin
                state <= S_IDLE;
                bram_write_out <= 0;
                last_valid_id_num <= 0;
            end
        endcase
    end
endmodule

/*
parser receives the raw pulse data from the comparator and determines if a bit has been sent
*/
module parser(
    input pulse_in, // Assumed to be a single clock cycle pulse
    input clk_in,
    input reset_in,
    output logic bit_ready_out,
    output logic current_bit_out
);
    parameter RFID_FREQ = 125000; // This needs to be tuned depending on the coil for optimal performance

    parameter PULSE_PER_BIT = 16;
//    parameter CYCLES_PER_PULSE = 2 * 100000000 / RFID_FREQ; //1600; // 1 / 62.5kHz * 100MHz = 1600 clock cycles per pulse
    parameter CYCLES_PER_PULSE = 1600;
    parameter CYCLE_COUNT_ERROR = 100; // allowable tolerance on the period
    logic [4:0] pulse_count; // count the number of pulses detected
    logic [11:0] cycle_count; // longest expected duration between pulses is 2400 cycles (1.5 * period)

    // Every 16 pulses, output one bit
    always_ff @(posedge clk_in) begin
        if(bit_ready_out) begin
            bit_ready_out <= 0; // ensure bit_ready_out is one pulse wide
        end
        if(pulse_in) begin
            // Check for phase shift
            if(cycle_count > CYCLES_PER_PULSE + CYCLE_COUNT_ERROR ||
                cycle_count < CYCLES_PER_PULSE - CYCLE_COUNT_ERROR) begin
                current_bit_out <= current_bit_out ^ 1'b1; // toggle current_bit_out
            end
            // Check if a bit has been sent
            if(pulse_count == PULSE_PER_BIT - 1) begin
                pulse_count <= 0;
                bit_ready_out <= 1; // tell next module that a bit is ready to be read
            end else begin
                pulse_count <= pulse_count + 1;
            end
            cycle_count <= 0;
```

```
        end else begin
            cycle_count <= cycle_count + 1;
        end

        if(reset_in) begin
            pulse_count <= 0;
            current_bit_out <= 0;
            bit_ready_out <= 0;
        end
    end

endmodule

/* Receives the output from the comparator. Needs to create a sharp transistion and an output pulse with one clock cycle width */
module pulse_gen(
    input comparator_in,
    input clk_in,
    input reset_in,
    output logic pulse_out
);
    logic prev_input;
    logic [4:0] input_buffer;
    always_ff @(posedge clk_in) begin
        if(pulse_out) begin
            pulse_out <= 0; // Guarantee one-cycle long pulse
        end
        input_buffer <= {comparator_in, input_buffer[4:1]};
        // Wait until the entire buffer agrees before accepting the bit since the comparator is slow
        if(input_buffer == 5'b11111 || input_buffer == 0) begin
            prev_input <= input_buffer[0];
            // Check for falling edge
            if(input_buffer[0] == 0 && prev_input == 1) begin
                pulse_out <= 1;
            end
        end
        if(reset_in) begin
            pulse_out <= 0;
            prev_input <= 0;
        end
    end
endmodule

module read_fsm(input bit_ready_in,
                input sent_bit_in,
                input clk_in,
                input reset_in,
                output logic [193:0] id_out,
                output logic id_ready_out);

    // Hyperparameters
    parameter CONSEC_BIT_THRESHOLD = 25; // Detect 25 consecutive bits before transitioning to triggered
    parameter NUM_CONST_1 = 22;
    parameter NUM_PERSONAL = 33;
    parameter NUM_CONST_2 = 139;

    // States
    parameter S_IDLE = 0;
```

```
    parameter S_TRIGGERED = 1;
    parameter S_CONSTANT_1 = 2;
    parameter S_PERSONAL = 3;
    parameter S_CONSTANT_2 = 4;

    logic [2:0] state = S_IDLE;
    logic parity = 0; // There's a chance all the bits are flipped. XOR inputs with this parity bit to fix this
    logic input_bit;
    assign input_bit = parity ^ sent_bit_in;
    logic prev_bit;
    logic [7:0] bit_count;

    always_ff @(posedge clk_in) begin
        if(reset_in) begin
            state <= S_IDLE;
            prev_bit <= 0;
            bit_count <= 0;
            parity <= 0;
            id_ready_out <= 0;
            id_out <= 0;
        end else if(bit_ready_in) begin
            case(state)
                S_IDLE: begin
                    // Look for consecutive string of same bit
                    id_ready_out <= 0; // Clear the bit if already set from a previous run
                    bit_count <= input_bit == prev_bit ? bit_count + 1 : 0;
                    prev_bit <= input_bit;
                    if(bit_count > CONSEC_BIT_THRESHOLD) begin
                        state <= S_TRIGGERED;
                        if(input_bit == 1) begin
                            // If a string of ones is detected, flip parity bit (we are backwards)
                            parity <= 1;
                        end
                    end
                end
                S_TRIGGERED: begin
                    // Wait for first one
                    if(input_bit == 1) begin
                        id_out[0] <= 1;
                        bit_count <= 1;
                        state <= S_CONSTANT_1;
                    end
                end
                S_CONSTANT_1: begin
                    if(bit_count == 11 && id_out[9:0] != 10'b0111000001) begin // invalid, reject
                        state <= S_IDLE;
                        bit_count <= 0;
                        parity <= 0;
                        prev_bit <= 0;
                    end
                    id_out[bit_count] <= input_bit;
                    bit_count <= bit_count + 1;
                    if(bit_count == NUM_CONST_1 - 1) begin
                        state <= S_PERSONAL;
                    end
                end
```

```
                S_PERSONAL: begin
                    id_out[bit_count] <= input_bit;
                    bit_count <= bit_count + 1;
                    if(bit_count == NUM_PERSONAL + NUM_CONST_1 - 1) begin
                        state <= S_CONSTANT_2;
                    end
                end
                S_CONSTANT_2: begin
                    id_out[bit_count] <= input_bit;
                    bit_count <= bit_count + 1;
                    if(bit_count == NUM_PERSONAL + NUM_CONST_1 + NUM_CONST_2
    - 1) begin
                        state <= S_IDLE;
                        id_ready_out <= 1;
                        bit_count <= 0;
                        parity <= 0;
                        prev_bit <= 0;
                    end
                end
                default:
                    state <= S_IDLE;
            endcase
        end
    end
endmodule
```