

RFID Tag Reader System

Katonio Butler and Akua Nti

6.111: Introductory Digital Systems Laboratory

December 14, 2005

Abstract

Radio Frequency Identification (RFID) technology holds much promise as applied to areas such as remote sensing and access control. The RFID reader system constructed in this project was designed to demonstrate access control through the use of low-frequency ID cards. These cards contained identification data that is read by the reader, sent to a database where it is compared to stored values, and implemented depending upon the level of access associated with it. If the card's identification data is in the system, a stored image as well as the name of the user associated with the ID card is displayed. If the data is not in the system, it notifies the user with the message "ACCESS DENIED!" on a display screen. The reader was successfully constructed using Verilog hardware description language in an FPGA.

Contents

1	Introduction	4
2	RFID Tag Reader System Overview	4
3	Tag Reader System Module Descriptions	5
3.1	RFID Reader Chip (Akua)	5
3.2	Asynchronous Receiver (Akua)	7
3.3	Database (Akua)	7
3.4	Display (Katonio)	7
3.4.1	VGA ROM Display (Katonio)	8
3.4.2	VGA Synchronizer (Katonio)	8
3.4.3	Character Display (Katonio)	9
4	Design and Testing	9
4.1	Design Flow	9
4.2	Testing and Debugging	10
5	Conclusions	11
A	Verilog Code	12
A.1	Top-Level RFID Module	13
A.2	Asynchronous Receiver	21
A.3	Database	26
A.4	Display	31
B	Miscellaneous	37
B.1	PGM-to-COE Conversion Code	37
B.2	Card Numbers	39

List of Figures

1	RFID Tag Reader System Block Diagram	6
---	--	---

List of Tables

1	Card ID Number Format	6
2	Card ID Numbers	39

1 Introduction

Radio Frequency Identification (RFID) is a remote sensing and control technology that can be used to identify and track people or objects. Applications of the technology include logistics, inventory management access control.

An RFID system typically consists of a tag, or transponder, and a reader. The RFID tag contains a digital memory chip with a unique ID, and an antenna to send data to and receive data from a reader. The RFID tag reader consists of a transceiver and decoder. The reader transmits a continuous interrogation signal which is received by a tag when it is within range. Upon receiving and decoding this signal, the tag sends the reader its identification data, which is typically a stream of bits.

When the reader receives proper identification data from the card, it can then send command signals to the tag. Responding to commands from the reader, the tag sends out encoded data. This data, which is sent out on an amplitude-modulated frequency, is received and decoded by the reader. The reader then sends it to a host computer to be processed.

Our RFID tag reader system combines a tag reader module with a VGA display to demonstrate the use of RFID for access control. It displays messages and images that depend on the ID number of the card and the level of access associated with that number.

2 RFID Tag Reader System Overview

A principal application of the RFID tag reader system is access control. Only people with cards whose ID numbers are in the database can unlock a door or have some other access to a facility. Additionally, if the system is used at a place where a human is checking identification, the guard can make sure the person holding the card is the person associated with the card by viewing the picture.

Typical use of the tag reader system is as follows: A user holds up a card in front of the receiver, parallel to the antenna. The value on the card is compared to values in a database, and a picture is displayed based on the card's level of access. If the ID is in the database, the corresponding picture and name is displayed on the screen. Otherwise, "ACCESS DENIED!" is displayed on the screen.

The system has the following modules: the RFID reader chip, an asynchronous receiver module, a database module, a character display module, and a VGA display module. The transmitter is connected to ground, so it is always transmitting. The receiver demodulates the signal sent from the card and outputs it serially, with 8 data bits, no parity, 1 stop bit (8N1). The asynchronous receiver decodes the data and outputs it byte-by-byte to the database. The database combines the serial bytes from the asynchronous receiver into a single 80-bit ID number. It then checks a database to see if the ID number is present. If it is, it outputs a name (160-bit ASCII string) and raises the unlock signal. Otherwise, it sets unlock to zero and outputs “ACCESS DENIED!”

There is a character display module and a VGA ROM display module, which both output to the VGA display. The character display module prints the ASCII name string sent to it from the database if the card’s ID number is in the database; otherwise it prints “ACCESS DENIED!”

The following sections of this report describes each module in more detail, discusses the design process and testing, and offers concluding considerations on the successful design and implementation of this project.

3 Tag Reader System Module Descriptions

The RFID tag reader system is made up of the following modules: an RFID reader circuit, an asynchronous receiver module, a database module, a character display module, and a VGA ROM display module. The information gathered by the reader chip flows from the chip to the asynchronous receiver to the database module, where it is used to determine which string and/or picture should be displayed on the screen. Figure 1 shows a block diagram of the RFID tag reader system.

3.1 RFID Reader Chip (Akua)

The first block in the system is an RFID reader module that we purchased from Parallax, Inc. (<http://www.parallax.com>). Along with the reader module (#28140), we also purchased five 54mm x 85mm cards with tags in them (#28141). Both the reader and the tags are designed to operate at a frequency of 170 kHz.

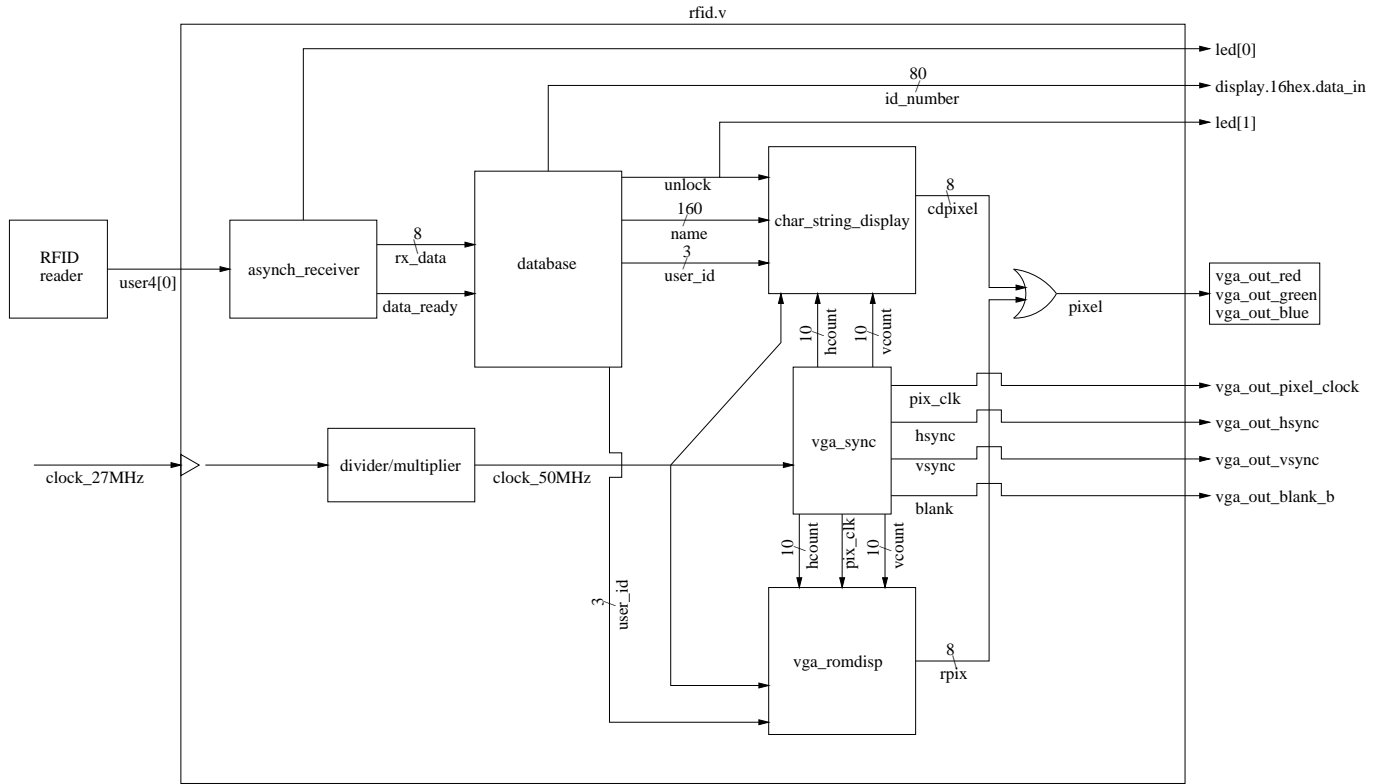


Figure 1: RFID Tag Reader System Block Diagram

The reader takes a 5V DC input and a voltage on its /ENABLE pin (negative logic) to control it, then outputs data serially at 2400 bps on its SOUT pin. The data is formatted with 8 data bits, no parity, 1 stop bit (8N1), with the least significant bit first, and sent using the RS-232 protocol. The output is a 12 byte ASCII string, of which the first byte is a newline (0A), the last byte is a carriage return (0D), and the middle ten bytes are the digits of the ID number. Table 1 shows some example card data.

Table 1: Card ID Number Format

hex	0A	30	46	30	32	37	38	32	42	43	32	0D
ASCII	newline	0	F	0	2	7	8	2	B	C	2	carriage return

3.2 Asynchronous Receiver (Akua)

The asynchronous receiver receives data serially from the reader module with the RS-232 protocol and outputs it byte-by-byte. Its input is a serial stream of bits from `user4[0]`, and its outputs are the received bytes (`rx_data`) and a `data_ready` signal.

In the RS-232 protocol, data is sent byte-by-byte, with varying idle periods in between the data. The idle period is indicated by a long period of a high signal, and the start of a new data byte is indicated by a logic 0, or the start bit. Next, the data is sent, followed by the stop bit(s).

The asynchronous receiver module essentially waits until the data line stays constant for a while and then looks for the start bit to determine when data is going to be sent. It collects the bits into the `rx_data` registers and raises `data_ready` when it has all 8 bits. To control the timing of all this, the receiver takes as a parameter the baud rate (in this case, 2400) and generates a `serial_clk` signal at that rate.

3.3 Database (Akua)

The database module converts data from the receiver from byte-serial to parallel. Its inputs are `rx_data` (8 bits) and `data_ready`, and its outputs are `unlock`, `name` (160 bits), `id_number` (80 bits), and `user` (3 bits). It first looks at `rx_data` until its value is equal to that of the start byte (0A). Then, at each assertion of `data_ready`, the database puts the data from `rx_data` into the appropriate slot of `id_number`, until all ten bytes are filled.

The next thing the database module does is to look up the ID number in a ROM to see if the number corresponds to a trusted user. Based on the ID number, the database module outputs an ASCII string on the `name` line, and a value for `user_id`. If the ID number corresponds to a trusted user, that user's name is output on `name`, and the `unlock` signal is raised. Otherwise, `unlock` gets zero, and the string "ACCESS DENIED!" is output on `name`.

3.4 Display (Katonio)

The `cdpixel` and `rpix` outputs of the `char_string_display.v` and `vga_romdisp.v` modules are OR'ed together into a combined `pixel` output in the top-level `rfid.v` module. This combined output is sent out through the VGA RGB output wires `vga_out_red`, `vga_out_green` and `vga_out_blue` to the 640x480 display, which is utilizing the `pix_clk` signal over `vga_out_pixel_clock` to operate

at a 60 Hz refresh rate.

3.4.1 VGA ROM Display (Katonio)

The `vga_romdisp.v` module displays an image from a single-port BROM module onto a VGA display. It operates using a 50 MHz `clk` input. Additionally, it has 10-bit inputs `hcount` and `vcount`, and `pix_clk` from the `vga_sync.v` module. It takes a 3-bit user input from the `database.v` module. It uses the `hcount` and `vcount` signals to set up addressing for the pixel output locations for the contents of the image BROMs, which are instantiated in this module. Four single-port BROMs are created to store four different 8-bit 320x240 `.pgm` (portable graymap) files which correspond to the user images. These `.pgm` files are converted to into an initial value (`.coe`) file for use in the BROM modules via a Python script named `pgm2coe.py` (see Appendix B.1). Employing the user input as a selector, a multiplexer was established which, based on the value of user, would select the appropriate BROM image output out of the four established to send to the module's 8-bit `pixel` output every cycle of the input clock value `pix_clk`.

3.4.2 VGA Synchronizer (Katonio)

The `vga_sync.v` module produces synchronous signals and blanking for a 640x480 VGA screen display. It takes a 50 MHz `clk` input and divides it to produce a 25 MHz `pix_clk` output for the 60 Hz VGA output pixel clock. In addition, it outputs two counters: `hcount` and `vcount`. The counter `hcount` counts pixels in a horizontal line. Values 0 through 639 are the 640 displayed pixels, while values 640 through 793 time the interval between the end of one scan line and the start of the next. Values 652 through 746 in the latter interval are decoded in this module to time the beginning and end of the `hsync` signal output.

The counter `vcount` counts scan lines in a video frame. Values 0 through 479 are displayed, while values 480 through 527 time the interval between the end of one frame and the beginning of the next. As with `hcount`, values in the latter interval are decoded to time the beginning and end of the `vsync` signal output (492 through 494). Both the `hcount` and `vcount` values, which store the (x,y) position of displayed pixels, are stored in a 10-bit frame buffer. Lastly, the module generates a `blank` output, which is a signal that equals 0 when a pixel is displayed on the VGA screen and 1 when a pixel would be off of the screen.

3.4.3 Character Display (Katonio)

The `char_string_display.v` module displays an ASCII-encoded character string onto a VGA screen at a specified location. The module is driven by a 50 MHz `vclock`, and takes the `hcount` and `vcount` inputs from the `vga_sync.v` module.

In order to display character string information, it is useful to program inputs to capture the character string and its location on the display; an input `cstring` is written to hold the character string, and inputs `cx` and `cy` are written to hold the (x,y) position of each pixel used to generate the character string output. Two parameter values are established to store character information: `NCHAR`, which is the value of the number of characters that will be displayed in a string, and `NCHAR.BITS`, which is the value of the number of bits that have to be used to specify `NCHAR`.

The output of this module is the value `pixel`, which contains the video pixel value that is displayed at the location specified by the inputs `hcount` and `vcount`. In this module, each character (using 8-bit ASCII format) is designed to be 8 bits x 12 bytes in size. The individual character font information is provided via a `font_rom` ROM module; this allows for versatility in the selection of fonts to be used to display the character strings. In addition, the displayed characters are magnified 2x due to the doubling of the `pixel` output. In this module, a maximum value of `NCHAR = 20` 8-bit characters per string is used to display the necessary information, with the corresponding `NCHAR.BIT` value set to 5 bits.

4 Design and Testing

4.1 Design Flow

Our design has evolved over the weeks into something quite different from what we originally envisioned. In general, the trend has been from a more analog design to a more digital design.

At the beginning, we were going to build our own transmitter and receiver to operate with the MIT ID cards. However, we soon found that we didn't know enough about RF and analog design. There were many issues (impedance matching, decoding the output, etc.) that we did not really understand, and would have taken us a lot of time to figure out. We eventually decided that there were too many complicating factors, so we decided to look into purchasing an RFID reader integrated circuit. In the end, we purchased an entire module that output data serially. We had to

abandon the MIT ID card.

With a new RFID module, we decided to focus more on decoding the data, comparing it with known values in a database, and using the results to display an image on a VGA screen. Our new design consisted of the reader, a module to gather the data from the reader, a module with the database and comparing, and a module to display pictures to the VGA screen. We eventually realized that we did not have things well specified, and that we needed to have more communication between the modules. We also realized that we did not know how to deal with the ID numbers, which are quite large.

4.2 Testing and Debugging

We quickly realized that the most effective way to verify that our modules were working as expected was to wire the various outputs to things we could look at—the oscilloscope, the logic analyzer, and the hex display and LEDs on the labkit.

To test the reader chip we purchased, we wired a simple circuit with `enable` wired to ground and VCC and ground wired appropriately. We then used an oscilloscope to look at the signal on SOUT. This gave us an idea of the way the data was organized on the cards, and verified that the chip worked as expected.

We were able to get code for an RS-232 asynchronous receiver module from the staff, which made things simpler. However, we needed to verify that it worked correctly. We did this by viewing the `rx_data` output on the logic analyzer. By doing so, we were able to see the data shifting every (serial) clock cycle and holding for a while after. When we looked at the `data_ready` output, however, we realized that it was never being asserted. Through trial and error, and by looking at the data we wanted on the analyzer, we were able to modify the code to make sure it was asserted at exactly the right moments.

To test the database module, we connected various outputs to the LEDs and the logic analyzer. For example, we output a bit of `rx_data` (or SOUT) to an led so we could tell when the chip was receiving data from the card. We used the logic analyzer to make sure that the values in the buffer were what we expected them to be, as well as to check on the values of intermediate signals like the counter.

Eventually, we began displaying the values of `buffer` and `id_number` on the hex display,

because the ID numbers were too large to be seen well on the logic analyzer.

A persistent issue that occurred during testing was the constant retracing of the ROM image associated with an ID card when it was held up to the RFID transceiver. As the ROM image was retraced on the screen, there was a slight delay that resulted in the assertion of the “ACCESS DENIED!” string. It is believed that this issue was due to the decision to ground the RFID transceiver’s /ENABLE pin, which left the transceiver in a state of perpetual transmission/reception. As a result, the ID number data was constantly being re-read and processed by the system.

5 Conclusions

The processes associated with the design and implementation of this system have been complicated. We have learned that it is often difficult to understand how involved something will be before actually attempting it. Despite the differences between the result and our original visions, we are pleased with the final result. We successfully completed a project that performed to our revised specifications, and we managed to do it without much stress. We learned that investing time and energy in planning in the initial design stages of a project can make the implementation phase much easier to complete.

A Verilog Code

This section contains the Verilog code used to implement the RFID tag reader system. It includes excerpts of a top-level file in which the modules are connected, the asynchronous receiver module, the database module, and the VGA ROM and character display modules.

A.1 Top-Level RFID Module

```
////////////////////////////////////  
// File:   rfid.v  
// Date:   14 December 2005  
// Authors: Akua Nti <afriyie@mit.edu>  
//         Katonio A. Butler <kabutler@mit.edu>  
//  
// modules: aysnc_receiver.v, baud_gen.v, char_string_display.v, database.v  
// rfid_rom.v, display_16hex.v, vga_romdisp.v, vga_sync.v  
//  
// input:  
// user4[0] - demodulated bitstream from RFID transceiver  
//  
// output:  
// pixel - video pixel value to display on screen  
//  
// Code for the MIT 6.111 FPGA Labkit, demonstrating the operation of a  
// radio-frequency ID (RFID) transceiver/reader unit. Upon enabling the  
// transceiver to broadcast an ID interrogation signal, an identification code  
// is received from the ID tag at 2400 bits per second, decoded, parsed and  
// output on the labkit's hex display. Images and user information stored in  
// ROMs are activated by the ID information. This data and the ID access level  
// are displayed to a vga monitor.  
//  
////////////////////////////////////
```

```

////////////////////////////////////
// Top-level display module (rfid)
// Date Modified: 12 December 2005
// Authors: Akua Nti, Katonio Butler
//
// This module wires each of the modules together and to produce the desired
// outputs on the hex display, leds and vga display.
//
////////////////////////////////////
module rfid(
    beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
    ac97_bit_clock,

    ...

    clock_27mhz, clock1, clock2,

    disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
    disp_reset_b, disp_data_in,

    button0, button1, button2, button3, button_enter, button_right,
    button_left, button_down, button_up,

    switch,

    led,

    user1, user2, user3, user4,

);

```

...

```
output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
vga_out_hsync, vga_out_vsync;
```

...

```
input  clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output disp_data_out;

input  button0, button1, button2, button3, button_enter, button_right,
       button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

inout [31:0] user1, user2, user3, user4;
```

...

```
////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////
```


...

```
// VGA Output
// assign vga_out_red = 10'h0;
// assign vga_out_green = 10'h0;
// assign vga_out_blue = 10'h0;
assign vga_out_sync_b = 1'b1;
// assign vga_out_blank_b = 1'b1;
// assign vga_out_pixel_clock = 1'b0;
// assign vga_out_hsync = 1'b0;
// assign vga_out_vsync = 1'b0;
```

...

```
// use FPGA's digital clock manager to produce a 50 Mhz clock from 27 Mhz
// actual frequency: 49.85 MHz
wire clock_50mhz_unbuf,clock_50MHz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 13
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_50MHz),.I(clock_50mhz_unbuf));
```

```
////////////////////////////////////
```

```
//
```

```
// Testing rfid module
```

```
//
```

```
// The last 8 bytes of the id number get printed to the hex display, led[1]
```

```
// lights when a card's id is in the database, led[0] flashes to show
```

```

// activity.
//
////////////////////////////////////////////////////////////////

wire [63:0] data;
wire [7:0] rx_data;
wire data_ready;
wire unlock;
wire [159:0] name;
wire [3:0] count;
wire [79:0] id_number;
wire [2:0] user_id;

// hex display
display_16hex hex(.reset(~button0), .clock_27mhz(clock_27mhz),
    .data_in(data), .disp_blank(disp_blank),
    .disp_clock(disp_clock), .disp_rs(disp_rs),
    .disp_ce_b(disp_ce_b), .disp_reset_b(disp_reset_b),
    .disp_data_out(disp_data_out));

// asynchronous receiver
async_receiver rfid_test(.clock_27mhz(clock_27mhz), .rxd(user4[0]),
    .data_ready(data_ready), .rx_data(rx_data));

// database
database db(.clock(clock_27mhz), .ready(data_ready), .data(rx_data),
    .unlock(unlock), .name(name), .id_number(id_number), .user(user_id));

//***** assign data to hex display *****

```

```

assign data = id_number[63:0];

//***** assign leds *****
assign led[0] = user4[0];
assign led[1] = ~unlock;

//***** assign logic analyzer *****
// assign analyzer1_data[15:0] = data[15:0];
// assign analyzer1_data[7:0] = rx_data;

////////////////////////////////////
//
// VGA image ROM display implementation
//
////////////////////////////////////

wire clk = clock_50MHz; // wired 50 MHz clock to drive vga output modules below

// 640x480 VGA display
wire [7:0] pixel;
wire blank;
wire pix_clk;
wire [9:0] hcount;
wire [9:0] vcount;

// vga signal synch module
vga_sync vga1(.clk(clk),.hsync(vga_out_hsync),.vsync(vga_out_vsync),

```

```

        .hcount(hcount), .vcount(vcount), .pix_clk(pix_clk),
        .blank(blank));

// wire attached to pixel output from vga ROM image display module
wire [7:0] rpix;

// vga ROM image display module
vga_romdisp vr(.clk(clk), .hcount(hcount), .vcount(vcount), .pix_clk(pix_clk),
               .user(user_id), .pixel(rpix));

/////////////////////////////////////////////////////////////////
//
// Character display implementation
//
/////////////////////////////////////////////////////////////////

// character display module: sample string in middle of screen
wire [159:0] cstring;
wire [2:0] cdpixel;
wire [10:0] cx;
wire [9:0] cy;

// assign (x,y) values for display of characters on display screen
assign cx = unlock ? 11'd210 : 11'd120;
assign cy = 10'd240;

// character string display module
char_string_display cd(.vclock(clk), .hcount({1'b0, hcount}),
                      .vcount(vcount), .pixel(cdpixel), .cstring(name), .cx(cx), .cy(cy));

```

```

// Assign the vga and character display signals to the pixel output
assign vga_out_red = pixel | {5{cdpixel[2]}}; // B+W display, blue chars
assign vga_out_green = pixel | {2{cdpixel[1]}};
assign vga_out_blue = pixel | {8{cdpixel[0]}};
assign vga_out_blank_b = ~blank;
assign vga_out_pixel_clock = pix_clk; // vga pixel clock @ 50 MHz for vga screen

// produces pixel output by ORing the character pixel output and image ROM
// pixel output
assign pixel = cdpixel | ((hcount > 320 | vcount < 120 | ~unlock) ? 8'b0 : rpix);
endmodule

```

A.2 Asynchronous Receiver

```
////////////////////////////////////
// RECEIVER MODULE
//
// modified source from
// RS-232 RX module
// (c) fpga4fun.com KNJN LLC - 2003, 2004, 2005
////////////////////////////////////

module async_receiver(clock_27mhz, rxd, data_ready, rx_data);
    input clock_27mhz, rxd;
    output data_ready; // one clock pulse when rx_data is valid
    output [7:0] rx_data;

    //connect baud generator
    wire serial_clk;
    baud_gen rx_clk(.clock_27mhz(clock_27mhz), .serial_clk(serial_clk),
        .enable(1'b0), .rx_mode(1'b1) );

    //synchronize (& invert) rxd input
    reg sync, rxd_inv;

    always @(posedge clock_27mhz)
        begin
            if(serial_clk)
                {rxd_inv, sync} <= {sync, ~rxd};
        end

    //////////////////////////////////////
    reg [1:0] rxd_cnt_inv;
```

```

reg rxd_bit_inv;

always @ (posedge clock_27mhz)
begin
    if(serial_clk) begin
        if( rxd_inv && rxd_cnt_inv!=2'b11)
            rxd_cnt_inv <= rxd_cnt_inv + 1;
        else if(~rxd_inv && rxd_cnt_inv!=2'b00)
            rxd_cnt_inv <= rxd_cnt_inv - 1;

        if(rxd_cnt_inv==2'b00)
            rxd_bit_inv <= 0;
        else if(rxd_cnt_inv==2'b11)
            rxd_bit_inv <= 1;
    end
end

////////////////////////////////////
reg [7:0] rx_data;
reg data_ready;
reg [3:0] state;
reg [3:0] bit_spacing;

// "next_bit" controls when the data sampling occurs
// depending on how noisy the rxd is, different values might work better
// with a clean connection, values from 8 to 11 work
wire next_bit = (bit_spacing==10);

always @ (posedge clock_27mhz)

```

```

begin
    if(state==0)
        bit_spacing <= 0;
    else if(serial_clk)
        bit_spacing <= {bit_spacing[2:0] + 1} | {bit_spacing[3], 3'b000};
    end

always @ (posedge clock_27mhz)
begin
    if(serial_clk)
        case(state)
            4'b0000: if(rxd_bit_inv) state <= 4'b1000; // start bit found?
            4'b1000: if(next_bit) state <= 4'b1001; // bit 0
            4'b1001: if(next_bit) state <= 4'b1010; // bit 1
            4'b1010: if(next_bit) state <= 4'b1011; // bit 2
            4'b1011: if(next_bit) state <= 4'b1100; // bit 3
            4'b1100: if(next_bit) state <= 4'b1101; // bit 4
            4'b1101: if(next_bit) state <= 4'b1110; // bit 5
            4'b1110: if(next_bit) state <= 4'b1111; // bit 6
            4'b1111: if(next_bit) state <= 4'b0001; // bit 7
            4'b0001: if(next_bit) state <= 4'b0000; // stop bit
            default: state <= 4'b0000;
        endcase

        data_ready <= (next_bit && state==4'b0001); // ready when there is a new byte

        if (serial_clk && next_bit && state[3])
            rx_data <= {~rxd_bit_inv, rx_data[7:1]};
    end
end

```



```
endmodule
```

```
////////////////////////////////////////////////////////////////
```

```
// BAUD GENERATOR
```

```
// modified source from
```

```
// RS-232 RX module
```

```
// (c) fpga4fun.com KNJN LLC - 2003, 2004, 2005
```

```
////////////////////////////////////////////////////////////////
```

```
module baud_gen(clock_27mhz, serial_clk, enable, rx_mode);
```

```
    input clock_27mhz, enable, rx_mode;
```

```
    output serial_clk;
```

```
    parameter clkfreq = 27000000; // 27 MHz
```

```
    parameter baudrate = 2400; // 11520
```

```
    parameter baudrate_8 = 2400*8; // 8 times over sampling (when receiving)
```

```
    // Baud generator
```

```
    parameter acc_width = 16;
```

```
    reg [acc_width:0] accumulator;
```

```
    parameter inc = ((baudrate<<(acc_width-4))+(clkfreq>>5))/(clkfreq>>4);
```

```
    parameter inc8 = ((baudrate_8<<(acc_width-7))+(clkfreq>>8))/(clkfreq>>7);
```

```
    assign serial_clk = accumulator[acc_width];
```

```
    always @ (posedge clock_27mhz)
```

```
        begin
```

```
            if(enable)
```

```
                accumulator <= accumulator[acc_width-1:0] + inc;
```

```
    else if (rx_mode)
        accumulator <= accumulator[acc_width-1:0] + inc8;
    end

endmodule
```

A.3 Database

```
////////////////////////////////////
//
// database
//
// database.v modified 12-9-05
// Author: Akua Nti
//
// This module contains the ROM with the stored information about trusted users.
// It takes in data and compares the values to those in the database.  If there
// is a match, it outputs a name and raises unlock.  Otherwise, it lowers unlock.
//
////////////////////////////////////

module database(clock,ready,data,unlock,name,id_number,user);
    input clock;
    input ready;
    input [7:0] data;
    output unlock;
    output [159:0] name;
    output [79:0] id_number; // tag id
    output [2:0] user; // user id

    /***** data extraction *****/
    Here we extract the id number bytes coming in and put them into a register.
    The module looks for the start byte in the incoming stream.  Once it finds
    that, it puts the next 10 bytes into the appropriate locations in id_number.

    *****/
```

```

reg old_ready;
wire new_byte;

// level-to-pulse conversion of ready
always @ (posedge clock) old_ready <= ready;
assign new_byte = ready & ~old_ready;

parameter [7:0] START = 8'h0A;
//parameter [7:0] STOP = 8'h0D;

reg [7:0] buffer = 8'h0A;
wire start_byte;
//wire stop_byte;

// if ready, put data into buffer, otherwise leave alone
always @ (posedge clock)
begin
    buffer <= new_byte ? data : buffer;
end

// look for start byte
assign start_byte = (buffer == START);
//assign stop_byte = (buffer == STOP);

// if start byte found, then put next 10 bytes into a RAM
reg [79:0] id_number;
reg [3:0] i;
//reg reading = 0;

```

```

parameter [3:0] ID_LENGTH = 10;

// counter goes up to length of id number (in bytes) and wraps around to 1
always @ (posedge clock)
begin
    i <= new_byte ? (start_byte ? 0 : ((i == ID_LENGTH) ? 1 : (i + 1))) : i;
end

// fill bytes of id_number from buffer
// least significant byte corresponds to last digit of id number
always @ (posedge clock)
begin
    case(i)
        9: id_number[7:0] <= buffer;
        8: id_number[15:8] <= buffer;
        7: id_number[23:16] <= buffer;
        6: id_number[31:24] <= buffer;
        5: id_number[39:32] <= buffer;
        4: id_number[47:40] <= buffer;
        3: id_number[55:48] <= buffer;
        2: id_number[63:56] <= buffer;
        1: id_number[71:64] <= buffer;
        0: id_number[79:72] <= buffer;
        default: id_number <= id_number;
    endcase // case(i)
end // always @ (posedge clock)

...

// instantiate ROM

```

```

        rfid_rom rom(.clock(clock), .addr(id_number), .data(name), .allow(unlock),
                    .user(user));

endmodule // database

//////////////////////////////////// ROM //////////////////////////////////////
// A simple simulated ROM.
//
// Compares the value of id_number with ROM values.  Outputs name (ASCII
// string) if id is in ROM.  Outputs "ACCESS DENIED!" if id is not in ROM.
// If an id is in the ROM, raises allow.
//
////////////////////////////////////

module rfid_rom (clock, addr, data, allow, user);
    input clock;
    input [79:0] addr;
    output [2:0] user;
    output [159:0] data; // 20 bytes (20 ascii chars) for the name
    output allow;
    reg [159:0] data;
    reg [2:0] user; // simplified user id

    always @ (posedge clock)
        begin
            case (addr)
                80'h30463032373832424332:
                    begin
                        data <= "Dolemite";

```

```

        user <= 3'b010;
    end

    80'h30463032373832303638:
    begin
        data <= "John Shaft";
        user <= 3'b011;
    end

    80'h30463032373833353443:
    begin
        data <= "Foxy Brown";
        user <= 3'b100;
    end

    80'h30463032373833354133:
    begin
        data <= "Cleo Jones";
        user <= 3'b110;
    end

    //      80'h30463032373833344633: data <= "person 2";

    default:
        data <= "ACCESS DENIED!";
    endcase
end // always @ (posedge clock)

    assign allow = (data != "ACCESS DENIED!");

endmodule

```

A.4 Display

```
////////////////////////////////////  
//  
// vga_sync.v  
// Verilog code to produce VGA sync signals (and blanking) for 640x480 screen  
// Date Modified: 11 December 2005  
// Author: Katonio Butler  
//  
////////////////////////////////////  
  
module vga_sync(clk,hsync,vsync,hcount,vcount,pix_clk,blank);  
    input clk;    // 50Mhz driven from 27 MHz  
    output hsync;  
    output vsync;  
    output [9:0] hcount, vcount;  
    output pix_clk;  
    output blank;  
  
    // pixel clock: 25Mhz = 40ns (clk/2)  
    reg pcount; // used to generate pixel clock  
    wire en = (pcount == 0);  
    always @ (posedge clk) pcount <= ~pcount;  
    wire pix_clk = ~en;  
  
    //*****  
    //*****  
    //***  
    //*** Sync and Blanking Signals  
    //***  
    //*****
```



```

//*****

reg hsync,vsync,hblank,vblank;
reg [9:0] hcount; // pixel number on current line
reg [9:0] vcount; // line number

// horizontal: 794 pixels = 31.76us
// display 640 pixels per line
wire hsyncon,hsyncoff,hreset,hblankon;
assign hblankon = en & (hcount == 639);
assign hsyncon = en & (hcount == 652);
assign hsyncoff = en & (hcount == 746);
assign hreset = en & (hcount == 793);

wire blank = (vblank | (hblank & ~hreset)); // blanking => black
//wire blank = vblank | hblank;

// vertical: 528 lines = 16.77us
// display 480 lines
wire vsyncon,vsyncoff,vreset,vblankon;
assign vblankon = hreset & (vcount == 479);
assign vsyncon = hreset & (vcount == 492);
assign vsyncoff = hreset & (vcount == 494);
assign vreset = hreset & (vcount == 527);

// sync and blanking
always @(posedge clk)
begin
    hcount <= en ? (hreset ? 0 : hcount + 1) : hcount;
    hblank <= hreset ? 0 : hblankon ? 1 : hblank;
end

```

```

        hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;    // hsync is active low
        vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
        vblank <= vreset ? 0 : vblankon ? 1 : vblank;
        vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;    // vsync is active low
    end

endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// vga_romdisp.v
// Displays an image from a BROM module
// Date Modified: 12 December 2005
// Author: Katonio Butler
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module vga_romdisp(clk,hcount,vcount,pix_clk,user,pixel);
    input clk; // video clock at 50 MHz
    input [9:0] hcount; // current x,y location of pixel
    input [9:0] vcount;
    input pix_clk; // pixel clock
    input [2:0] user; // user id number from database
    output [7:0] pixel; // pixel value output

    // the memory address is hcount/2 + vcount/2 * 320
    // (4 pixels per memory location, since image is 320x240, and
    // display is 640x480).

    reg [14:0] raddr;
    reg [7:0] pixel;

```

```

// implement wires for a mux to take in the outputs from the ROMs and
// send a single pixel output
wire [7:0] out_0;
wire [7:0] out_1;
wire [7:0] out_2;
wire [7:0] out_3;

always @ (posedge clk)
begin
    raddr <= (hcount==0 & vcount==120) ? 0
        : (hcount==0 & pix_clk & ~vcount[0]) ? raddr + 160 : raddr;
end

// row addressing for pixel outputs
wire [14:0] addr = {6'b0,hcount[9:1]} + raddr[14:0];

always @ (posedge clk)
begin
    if (pix_clk)
begin
        case(user) // based on user ID from database module,
                // select appropriate ROM pixel output
            3'b010 : pixel <= out_0; // Dolemite
            3'b011 : pixel <= out_1; // John Shaft
            3'b100 : pixel <= out_2; // Foxy Brown
            3'b110 : pixel <= out_3; // Cleo Jones
            default : pixel <= 8'b0;
        endcase
    end
end

```

```

end

// instantiate the image BROMs for each user
dolemite dolemite1(.addr(addr[14:0]),.clk(clk),.dout(out_0)); // Dolemite
shaft shaft1(.addr(addr[14:0]),.clk(clk),.dout(out_1)); // Shaft
foxy_brown foxy_brown1(.addr(addr[14:0]),.clk(clk),.dout(out_2)); // Foxy Brown
cleo_jones cleo_jones1(.addr(addr[14:0]),.clk(clk),.dout(out_3)); // Cleo Jones

endmodule // vga_romdisp

////////////////////////////////////
// video character string display
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// Date Modified: 12 December 2005
// Author: Katonio Butler
//
////////////////////////////////////
module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy);
    // increase paramater values to display more characters on screen
    parameter NCHAR = 20; // number of 8-bit characters in cstring
    parameter NCHAR_BITS = 5; // number of bits in NCHAR

    input vclock; // 27 MHz 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 20 character display (20 x 12 pixel-sized characters)
wire [10:0] hoff = hcount-1-cx;
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(font_addr,vclock,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

```

B Miscellaneous

B.1 PGM-to-COE Conversion Code

```
#!/usr/bin/python
#
# File:   pgm2coe.py
# Modified date:   11-Dec-05
# Author: I. Chuang <ichuang@mit.edu>
#
# Q&D conversion of PGM format image file to COE file for Xiling FPGA memory
# initialization.

import string,re,sys,os,struct

fn = "shaft"

fp = open('%s.pgm' % fn)

s = fp.readline() # P5

s = fp.readline()
m = re.compile('([0-9]+) ([0-9]+)').search(s)
nx = string.atoi(m.group(1))
ny = string.atoi(m.group(2))

print "nx = %d, ny=%d" % (nx,ny)

s = fp.readline() # maxval

ndat = nx*ny
```

```
imgdat = fp.read(ndat)

imgbytes = struct.unpack('%dB' % ndat, imgdat)

print "size = %d" % len(imgbytes)

fp.close()

fp = open('%s.coe' % fn, 'w')

fp.write("memory_initialization_radix=16;\nmemory_initialization_vector=\n")

for j in range(ny):
    for k in range(nx):
        fp.write("%02x," % imgbytes[k+j*nx])
        fp.write("\n");

fp.close()
```

B.2 Card Numbers

For future reference, Table 2 shows a list of the card numbers we read from the cards.

Table 2: Card ID Numbers

card 1	30	46	30	32	37	38	32	42	43	32
card 2	30	46	30	32	37	38	32	30	36	38
card 3	30	46	30	32	37	38	33	35	34	43
card 4	30	46	30	32	37	38	33	35	41	33
card 5	30	46	30	32	37	38	33	34	46	33