Super IO Appendix

May 12, 2005

Contents

1 Verilog Source Code

1.1 labkit.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Apr-29: Change history started
```

```
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
//////////////////////////////////////////////////////////////////////////////

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
       ac97_bit_clock,

       vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
       vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
       vga_out_vsync,

       tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
       tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
       tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

       tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
       tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
       tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
       tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

       ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
       ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

       ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
       ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

       clock_feedback_out, clock_feedback_in,

       flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
       flash_reset_b, flash_sts, flash_byte_b,

       rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

       mouse_clock, mouse_data, keyboard_clock, keyboard_data,

       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,

    daughtercard,

    systemace_data, systemace_address, systemace_ce_b,
    systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

    analyzer1_data, analyzer1_clock,
      analyzer2_data, analyzer2_clock,
      analyzer3_data, analyzer3_clock,
      analyzer4_data, analyzer4_clock);

 output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
 input  ac97_bit_clock, ac97_sdata_in;

 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;

 output [9:0] tv_out_ycrcb;
 output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
tv_out_subcar_reset;

 input  [19:0] tv_in_ycrcb;
 input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
tv_in_hff, tv_in_aff;
 output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
tv_in_reset_b, tv_in_clock;
 inout  tv_in_i2c_data;

 inout  [35:0] ram0_data;
 output [18:0] ram0_address;
 output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
 output [3:0] ram0_bwe_b;

 inout  [35:0] ram1_data;
 output [18:0] ram1_address;
 output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
 output [3:0] ram1_bwe_b;

 input  clock_feedback_in;
 output clock_feedback_out;
```

```
  inout  [15:0] flash_data;
  output [23:0] flash_address;
  output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
  input  flash_sts;

  output rs232_txd, rs232_rts;
  input  rs232_rxd, rs232_cts;

  input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

  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;

  inout [43:0] daughtercard;

  inout  [15:0] systemace_data;
  output [6:0]  systemace_address;
  output systemace_ce_b, systemace_we_b, systemace_oe_b;
  input  systemace_irq, systemace_mpbrdy;

  output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
analyzer4_data;
  output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

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

  // Audio Input and Output
  assign beep= 1'b0;
  assign audio_reset_b = 1'b0;
  assign ac97_synch = 1'b0;
  // ac97_sdata_out and ac97_sdata_out are inputs;

  // 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;

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input
```

```
// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
//assign rs232_txd = 1'b1;
//assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
// disp_data_out is an input

// Buttons, Switches, and Individual LEDs
//assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
```

```
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;


    wire ready;
    wire eop;
    wire idle;
wire [7:0] rdata;

wire global_reset;
wire [7:0] bus_data;
wire [7:0] bus_addr;
wire bus_wr, bus_busy, bus_go;

    // motor 0 wiring
    wire bus_busy0;
    wire [7:0] dMot0;
    wire [2:0] aMot0;
    wire eMot0;
    wire bMot0;
    wire wMot0;
    wire m0en;
    wire m0dir;
    // motor 1 wiring
    wire bus_busy1;
    wire [7:0] dMot1;
    wire [2:0] aMot1;
    wire eMot1;
    wire bMot1;
    wire wMot1;
    wire m1en;
    wire m1dir;

    // digital in 0 wiring
    wire bus_busy2;
    wire [7:0] dDigin0;
    wire [2:0] aDigin0;
    wire eDigin0;
    wire bDigin0;
    wire wDigin0;
    wire [7:0] diginPort0;

    // lcd wiring
    wire bus_busy3;
    wire [7:0] dLCD;
    wire [2:0] aLCD;
    wire wLCD;
    wire eLCD;
    wire bLCD;
    wire [7:0] lcdData;
```

```
wire lcdRS;
wire lcdRW;
wire lcdEn;

// servo wiring
wire bus_busy4;
wire [7:0] dServ0;
wire [2:0] aServ0;
wire eServ0;
wire bServ0;
wire wServ0;
wire [7:0] servo;

// encoder 0 wiring
wire bus_busy5;
wire [7:0] dEnc0;
wire [2:0] aEnc0;
wire eEnc0;
wire bEnc0;
wire wEnc0;
wire [7:0] enc0vel;
wire encoder0;

// encoder 1 wiring
wire bus_busy6;
wire [7:0] dEnc1;
wire [2:0] aEnc1;
wire eEnc1;
wire bEnc1;
wire wEnc1;
wire [7:0] enc1vel;
wire encoder1;

// analog 0 wiring
wire bus_busy7;
wire [7:0] dAna0;
wire [2:0] aAna0;
wire eAna0;
wire bAna0;
wire wAna0;
wire [3:0] ana0mux;
wire [7:0] ana0data;
wire ana0ce, ana0cs, ana0rw, ana0stat;

// digital out 0 wiring
wire bus_busy8;
wire [7:0] dDigout0;
wire [2:0] aDigout0;
wire eDigout0;
wire bDigout0;
```

```
    wire wDigout0;
    wire [7:0] digoutPort0;

    wire [7:0] templeds;

    assign global_reset = switch[0];

SerialBusController bc(clock_27mhz, global_reset, rs232_rxd, rs232_txd, bus_data, bus_addr
    assign bus_busy = bus_busy0 | bus_busy1 | bus_busy2 | bus_busy3 | bus_busy4 | bus_busy

    // motor controller 0
    RegController8  rcM0(clock_27mhz, global_reset, 5'h00, bus_data, bus_addr, bus_wr, bus
    ModuleMotor ModuleM0(clock_27mhz, global_reset, aMot0, dMot0, wMot0, eMot0, bMot0, enc
    assign user1[0] = m0en;
    assign user1[1] = m0dir;
    assign user1[2] = ~m0dir;

    // motor controller 1
    RegController8  rcM1(clock_27mhz, global_reset, 5'h01, bus_data, bus_addr, bus_wr, bus
    ModuleMotor ModuleM1(clock_27mhz, global_reset, aMot1, dMot1, wMot1, eMot1, bMot1, enc
    assign user1[3] = m1en;
    assign user1[4] = m1dir;
    assign user1[5] = ~m1dir;

    // digital in 0
    RegController8  rcDI0(clock_27mhz, global_reset, 5'h02, bus_data, bus_addr, bus_wr, bu
    ModuleDigitalIn ModuleDI0(clock_27mhz, global_reset, aDigin0, dDigin0, wDigin0, eDigin
    Debouncer db0(clock_27mhz, global_reset, diginPort0[0], button_up);
    Debouncer db1(clock_27mhz, global_reset, diginPort0[1], button_down);
    Debouncer db2(clock_27mhz, global_reset, diginPort0[2], button_left);
    Debouncer db3(clock_27mhz, global_reset, diginPort0[3], button_right);
    Debouncer db4(clock_27mhz, global_reset, diginPort0[4], button_enter);
    Debouncer db5(clock_27mhz, global_reset, diginPort0[5], button0);
    Debouncer db6(clock_27mhz, global_reset, diginPort0[6], button1);
    Debouncer db7(clock_27mhz, global_reset, diginPort0[7], button2);

    // lcd
    RegController8  rcLCD(clock_27mhz, global_reset, 5'h03, bus_data, bus_addr, bus_wr, bu
    ModuleCharacterLCD lcd(clock_27mhz, global_reset, aLCD, dLCD, wLCD, eLCD, bLCD, lcdDat
    assign user2[14:7]  = lcdData[7:0];
    assign user2[6] = lcdEn;
    assign user2[5] = lcdRW;
    assign user2[4] = lcdRS;

    // servo controller
    RegController8  rcS0(clock_27mhz, global_reset, 5'h04, bus_data, bus_addr, bus_wr, bus
    ModuleServo ModuleS0(clock_27mhz, global_reset, aServ0, dServ0, wServ0, eServ0, bServ0
    assign user1[31] = servo[0];
    assign user1[30] = servo[1];
```

```
    // encoder controller 0
    RegController8  rcEnc0(clock_27mhz, global_reset, 5'h05, bus_data, bus_addr, bus_wr, b
    ModuleEncoder ModuleEnc0(clock_27mhz, global_reset, aEnc0, dEnc0, wEnc0, eEnc0, bEnc0,
    assign encoder0 = user1[6];

    // encoder controller 1
    RegController8  rcEnc1(clock_27mhz, global_reset, 5'h06, bus_data, bus_addr, bus_wr, b
    ModuleEncoder ModuleEnc1(clock_27mhz, global_reset, aEnc1, dEnc1, wEnc1, eEnc1, bEnc1,
    assign led = ~enc1vel;
    assign encoder1 = user1[7];

    // analog in 0
    RegController8  rcAna0(clock_27mhz, global_reset, 5'h07, bus_data, bus_addr, bus_wr, b
    ModuleAnalogIn ModuleAna0(clock_27mhz, global_reset, aAna0, dAna0, wAna0, eAna0, bAna0
    assign ana0data[7:0] = user4[27:20];
    assign user4[28] = ana0ce;
    assign user4[29] = ana0ce;
    assign user4[30] = ana0rw;
    assign ana0stat = user4[31];

    // mux (digital out) 0
    RegController8  rcDigout0(clock_27mhz, global_reset, 5'h08, bus_data, bus_addr, bus_wr
    ModuleDigitalOut ModuleDigout0(clock_27mhz, global_reset, aDigout0, dDigout0, wDigout0
    assign user4[19:17] = digoutPort0[2:0];

endmodule
```

1.2 SerialBusController.v

```
module SerialBusController(clk,reset,rxd,txd,data,addr,write,go,busy,leds);
    input clk;
    input reset;
    input rxd;
    output txd;
    inout [7:0] data;
    output [7:0] addr;
    reg [7:0] addr;
    output write;
    output go;
    reg go;
    input busy;

    output [7:0] leds;
    reg [7:0] leds;

    reg [7:0] dataOut;

    wire rxd_ready, rxd_eop, rxd_idle;
    wire [7:0] rxd_data;

    reg txd_start;
    wire txd_busy;
    reg [7:0] txd_data;

    async_receiver rx(clk, rxd, rxd_ready, rxd_data, rxd_eop, rxd_idle);
    async_transmitter tx(clk, txd_start, txd_data, txd, txd_busy);

    reg dataWr;

    reg [4:0] state;

    parameter INFO_VERSION_MAJOR    = 0;
    parameter INFO_VERSION_MINOR    = 1;
    parameter INFO_BUILD_DAY        = 10;
    parameter INFO_BUILD_MONTH      = 5;
    parameter INFO_BUILD_YEAR       = 5;

    parameter S_IDLE    = 0;
    parameter S_GETREG  = 1;
    parameter S_GETVAL  = 2;
    parameter S_BUSYWAIT= 3;
    parameter S_WRITE   = 4;
    parameter S_READ1   = 5;
    parameter S_READ2   = 6;
    parameter S_READ3   = 7;
    parameter S_READWAIT= 8;
```

```
assign data = dataWr ? dataOut : 8'hz;
assign write = dataWr;

always @ (posedge clk)
begin
    if (reset)
    begin
        state <= S_IDLE;
        txd_data <= 0;
        txd_start <= 0;
        dataWr <= 1;
        dataOut <= 0;
        addr <= 0;
        go <= 0;

        leds <= 0;
    end
    else
    begin
        case (state)
            S_IDLE:
            begin
                go <= 0;
                if (rxd_ready && (rxd_data[7:1]==7'b0100001))
                begin
                    dataWr <= rxd_data[0];
                    state <= S_GETREG;
                end
            end

            S_GETREG:
            begin
                if (rxd_ready)
                begin
                    addr <= rxd_data;
                    if (dataWr)
                        state <= S_GETVAL;
                    else
                    begin
                        go <= 1;
                        state <= S_READ1;
                    end
                end
            end

            S_GETVAL:
            begin
                if (rxd_ready)
                begin
```

```
        dataOut <= rxd_data;
        state <= S_BUSYWAIT;
    end
end

S_BUSYWAIT:
begin
    if (!busy)
        state <= S_WRITE;
end

S_WRITE:
begin
    go <= 1;
    state <= S_IDLE;
end

S_READ1:
begin
    state <= S_READ2;
end

S_READ2:
begin
    state <= S_READ3;
end

S_READ3:
begin
    leds <= data;
    case (addr)
        248:    txd_data <= INFO_VERSION_MAJOR;
        249:    txd_data <= INFO_VERSION_MINOR;
        250:    txd_data <= INFO_BUILD_DAY;
        251:    txd_data <= INFO_BUILD_MONTH;
        252:    txd_data <= INFO_BUILD_YEAR;
        default:txd_data <= data;
    endcase

    txd_start <= 1;
    if (txd_busy)
        state <= S_READWAIT;
end

S_READWAIT:
begin
    txd_start <= 0;
    if (!txd_busy)
        state <= S_IDLE;
end
```

```
            endcase
        end
    end

endmodule
```

1.3 rs232rx.v

```
// RS232 Serial UART Reciever - www.fpga4fun.com
module async_receiver(clk, RxD, RxD_data_ready, RxD_data, RxD_endofpacket, RxD_idle);
input clk, RxD;
output RxD_data_ready;  // onc clock pulse when RxD_data is valid
output [7:0] RxD_data;

parameter ClkFrequency = 27000000;
parameter Baud = 115200;

// We also detect if a gap occurs in the received stream of characters
// That can be useful if multiple characters are sent in burst
//  so that multiple characters can be treated as a "packet"
output RxD_endofpacket;  // one clock pulse, when no more data is received (RxD_idle is go
output RxD_idle;  // no data is being received

// Baud generator (we use 8 times oversampling)
parameter Baud8 = Baud*8;
parameter Baud8GeneratorAccWidth = 16;
parameter Baud8GeneratorInc = ((Baud8<<(Baud8GeneratorAccWidth-7))+(ClkFrequency>>8))/(Clk
reg [Baud8GeneratorAccWidth:0] Baud8GeneratorAcc;
always @(posedge clk)
Baud8GeneratorAcc <= Baud8GeneratorAcc[Baud8GeneratorAccWidth-1:0] + Baud8GeneratorInc;
wire Baud8Tick = Baud8GeneratorAcc[Baud8GeneratorAccWidth];

////////////////////////////
reg [1:0] RxD_sync_inv;
always @(posedge clk)
if(Baud8Tick) RxD_sync_inv <= {RxD_sync_inv[0], ~RxD};
// we invert RxD, so that the idle becomes "0", to prevent a phantom character to be recei

reg [1:0] RxD_cnt_inv;
reg RxD_bit_inv;

always @(posedge clk)
if(Baud8Tick)
begin
  if( RxD_sync_inv[1] && RxD_cnt_inv!=2'b11) RxD_cnt_inv <= RxD_cnt_inv + 1;
  else
  if(~RxD_sync_inv[1] && 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

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 clk)
if(state==0)
  bit_spacing <= 0;
else
if(Baud8Tick)
  bit_spacing <= {bit_spacing[2:0] + 1} | {bit_spacing[3], 3'b000};

always @(posedge clk)
if(Baud8Tick)
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

reg [7:0] RxD_data;
always @(posedge clk)
if(Baud8Tick && next_bit && state[3]) RxD_data <= {~RxD_bit_inv, RxD_data[7:1]};

reg RxD_data_ready, RxD_data_error;
always @(posedge clk)
begin
  RxD_data_ready <= (Baud8Tick && next_bit && state==4'b0001 && ~RxD_bit_inv);  // ready o
  RxD_data_error <= (Baud8Tick && next_bit && state==4'b0001 &&  RxD_bit_inv);  // error i
end

reg [4:0] gap_count;
always @(posedge clk) if (state!=0) gap_count<=0; else if(Baud8Tick & ~gap_count[4]) gap_c
assign RxD_idle = gap_count[4];
reg RxD_endofpacket; always @(posedge clk) RxD_endofpacket <= Baud8Tick & (gap_count==15);

endmodule
```

1.4 rs232tx.v

```
// RS232 Serial UART Transmitter - www.fpga4fun.com
module async_transmitter(clk, TxD_start, TxD_data, TxD, TxD_busy);
    input clk, TxD_start;
    input [7:0] TxD_data;
    output TxD, TxD_busy;

    parameter ClkFrequency = 27000000;
    parameter Baud = 115200;

    // Baud generator
    parameter BaudGeneratorAccWidth = 16;
    parameter BaudGeneratorInc = ((Baud<<(BaudGeneratorAccWidth-4))+(ClkFrequency>>5))/(Cl
    reg [BaudGeneratorAccWidth:0] BaudGeneratorAcc;
    wire BaudTick = BaudGeneratorAcc[BaudGeneratorAccWidth];
    wire TxD_busy;
    always @(posedge clk) if(TxD_busy) BaudGeneratorAcc <= BaudGeneratorAcc[BaudGeneratorA

    // Transmitter state machine
    reg [3:0] state;
    assign TxD_busy = (state!=0);

    always @(posedge clk)
    case(state)
      4'b0000: if(TxD_start) state <= 4'b0100;
      4'b0100: if(BaudTick) state <= 4'b1000;  // start
      4'b1000: if(BaudTick) state <= 4'b1001;  // bit 0
      4'b1001: if(BaudTick) state <= 4'b1010;  // bit 1
      4'b1010: if(BaudTick) state <= 4'b1011;  // bit 2
      4'b1011: if(BaudTick) state <= 4'b1100;  // bit 3
      4'b1100: if(BaudTick) state <= 4'b1101;  // bit 4
      4'b1101: if(BaudTick) state <= 4'b1110;  // bit 5
      4'b1110: if(BaudTick) state <= 4'b1111;  // bit 6
      4'b1111: if(BaudTick) state <= 4'b0001;  // bit 7
      4'b0001: if(BaudTick) state <= 4'b0010;  // stop1
      4'b0010: if(BaudTick) state <= 4'b0000;  // stop2
      default: if(BaudTick) state <= 4'b0000;
    endcase

    // Output mux
    reg muxbit;
    always @(state[2:0] or TxD_data)
    case(state[2:0])
      0: muxbit <= TxD_data[0];
      1: muxbit <= TxD_data[1];
      2: muxbit <= TxD_data[2];
      3: muxbit <= TxD_data[3];
      4: muxbit <= TxD_data[4];
```

```
      5: muxbit <= TxD_data[5];
      6: muxbit <= TxD_data[6];
      7: muxbit <= TxD_data[7];
    endcase

    // Put together the start, data and stop bits
    reg TxD;
    always @(posedge clk) TxD <= (state<4) | (state[3] & muxbit);  // register the output

endmodule
```

1.5 I2CSlave.v

```
module I2CSlave(reset, clk, sda, sdaoutm, scl, sckcopy, I2CAddr, data, addr, write, dataEn

    // reset
    input reset;
    // my i2c address
    input [6:0] I2CAddr;
    // global clock
    input clk;
    // i2c lines
    input scl;
    //inout sda;
    input sda;
    // data bus
    inout [7:0] data;
    reg [7:0] dataOut;
    reg [7:0] dataRead;
    output [7:0] addr;
    reg [7:0] addr;
    output write;
    reg write;
    output dataEnOut;
    reg dataEnOut;
    output sdaoutm;
    output sckcopy;

    assign sckcopy = scl;

    reg dataEn ,dataEns1, dataEns2;

    reg [7:0] dataWrite;

    reg [7:0] dataIn;

    // internal status stuff
    reg start,stop;
    reg startOld;
    wire gotByte;
    wire addrGood;
    reg i2c_read;
    output [7:0] shift;
    reg [7:0] shift;
    output [2:0] counter;
    reg [2:0] counter;
    reg counterReset;
    reg sdaOut;

output [3:0] stated;
```

```
reg [3:0] stated;

    reg [3:0] state;

// state machine
    parameter S_IDLE        = 4'h0;
    parameter S_ADDR_ACK    = 4'h1;
    parameter S_REG         = 4'h2;
    parameter S_REG_ACK     = 4'h3;
    parameter S_DATA        = 4'h4;
    parameter S_DATA_ACK    = 4'h5;

    // check if we've recieved 8 bits
    assign gotByte = (counter==0);

    // check if cur addr = our addr
    assign addrGood = (shift[7:1]==I2CAddr);

    // check for start bit
    always @ (negedge sda or posedge sda)
    begin
if (!sda) begin
       if (scl)
       begin
           $display("[I2CSLAVE] start bit");
           start <= 1;
           stop <= 0;
       end
        else start <= 0;
end
else begin
if (scl)
        begin
           $display("[I2CSLAVE] stop bit");
           stop <= 1;
           start <= 0;
          end
        else stop <= 0;
end
    end

    // wierd i2c pullup tristating.
    assign sdaoutm = sdaOut;

    // tristate parallel data port
    assign data = write ? dataWrite : 8'hz;

    always @ (posedge clk)
    begin
        if (reset)
```

```
        begin
            dataWrite <= 0;
            dataIn <= 0;
        end
        else
        begin
            dataWrite <= dataOut;
            dataIn <= data;
        end
    end


    always @ (negedge scl or posedge scl)
    begin
if (!scl)
begin
       if (reset)
       begin
           state <= S_IDLE;
           counterReset <= 1;
           sdaOut <= 1;
           dataOut <= 0;
           dataRead <= 0;
           addr <= 0;
           write <= 1;
           i2c_read <= 0;
           dataEn <= 0;
           dataEns1 <= 0;
           dataEns2 <= 0;
           dataEnOut <= 0;
stated <= 0;
       end
       else
       if (stop || (start && !startOld))
       begin
           // reset fsm on start or stop
           counterReset <= 1;
           state <= S_IDLE;
       end
       else
       begin
           // run
           counterReset <= 0;

           case (state)
               S_IDLE:
               begin
                   // wait for I2C address and check it
    stated <= stated | 1;
    if (gotByte)
```

```
 dataEnOut <= 1;
                if (/*addrGood && */gotByte)
                begin
                    state <= S_ADDR_ACK;
                    i2c_read <= shift[0];
                    write <= !shift[0];
                    if (shift[0])
                       dataEn <= 1;
                    sdaOut <= 0;
                    $display("[I2CSLAVE] got good i2c addr");
                end
            end

            S_ADDR_ACK:
            begin
stated <= stated | 2;
                // ack the I2C address
                $display("[I2CSLAVE] sending ack (read=%d)",i2c_read);
                if (i2c_read)
                begin
                    state <= S_DATA;
                    dataRead <= dataIn;
                    sdaOut <= dataIn[7];
                end
                else
                begin
                    sdaOut <= 1;
                    state <= S_REG;
                end
                counterReset <= 1;
                dataEn <= 0;
            end

            S_REG:
            begin
stated <= stated | 4;
                // wait for register byte
                if (gotByte)
                begin
                    $display("[I2CSLAVE] got register address (0x%X)",shift);
                    addr <= shift;
                    sdaOut <= 0;
                    state <= S_REG_ACK;
                end
            end

            S_REG_ACK:
            begin
stated <= stated | 8;
                // ack register byte
```

```
    $display("[I2CSLAVE] sending ack");
    sdaOut <= 1;
    state <= S_DATA;
    counterReset <= 1;
end

S_DATA:
begin
    // wait for data byte / send data byte
    if (i2c_read)
       sdaOut <= dataRead[7];

    if (gotByte)
    begin
        sdaOut <= i2c_read;
        state <= S_DATA_ACK;

        if (i2c_read)
        begin
            $display("[I2CSLAVE] reading");
        end
        else
        begin
            $display("[I2CSLAVE] writing (0x%X)",shift);
            dataOut <= shift;
            dataEn <= 1;
        end
    end
end

S_DATA_ACK:
begin
    dataEn <= 0;
    // ack data byte
    counterReset <= 1;
    if (i2c_read)
    begin
        if (sda)
        begin
            state <= S_IDLE;
            sdaOut <= 1;
        end
        else
        begin
            state <= S_DATA;
            sdaOut <= dataRead[7];
        end
    end
    else
    begin
```

```
                        state <= S_DATA;
                        sdaOut <= 1;
                    end
                end
            endcase
        end
end
else
begin
  shift <= {shift[6:0], sda};
        if (counterReset)
           counter <= -1;
        else
           counter <= counter - 1;

        startOld <= start;
  if (!gotByte && i2c_read)
            dataRead <= {dataRead[6:0], 1'h1};
end
    end
endmodule
```

1.6 RegController8.v

```
module RegController8(clk, reset, index, databus, addrb, wrbb,  enableb, busyb, datamod, a
    input [7:0] addrb;
    inout [7:0] databus;
    inout [7:0] datamod;
    input [2:0] addrm;
    input [4:0] index;
    input wrbb, clk, enableb, enablem, wrbm, reset;
    output busyb, busym;

    reg [7:0] doutb;
    reg [7:0] doutm;
    reg busyb, busym;

    output [7:0] reg1;
    reg [7:0] reg1;
    reg [7:0] reg2;
    reg [7:0] reg3;
    reg [7:0] reg4;
    reg [7:0] reg5;
    reg [7:0] reg6;
    reg [7:0] reg7;
    reg [7:0] reg8;

    reg [2:0] state;
    reg [7:0] counter;

    parameter RESET = 0;
    parameter IDLE = 1;
    parameter READB = 2;
    parameter WRITEB = 3;
    parameter READM = 4;
    parameter WRITEM = 5;

    assign databus = busym ? doutb : 8'bZ;
    assign datamod = busyb ? doutm : 8'bZ;

    always @ (posedge clk) begin
        case (state)

            RESET:
            begin
                busyb <= 0;
                busym <= 0;
                reg1 <= 0;
                reg2 <= 0;
                reg3 <= 0;
                reg4 <= 0;
```

```
                reg5 <= 0;
                reg6 <= 0;
                reg7 <= 0;
                reg8 <= 0;
                state <= IDLE;
                counter <= 0;
            end

            IDLE:
            begin
                busyb <= 0;
                busym <= 0;
                counter <= 0;
                if (enableb && addrb[7:3] == index)
                begin
                    if (wrbb)
                        state <= WRITEB;
                    else
                        state <= READB;
                end
                else if (enablem)
                begin
                    if (wrbm)
                        state <= WRITEM;
                    else
                        state <= READM;
                end
            end

            READB:
            begin
                busym <= 1;
//                doutb <= regs[addrb[2:0]];
                if (addrb[2:0] == 0) doutb <= reg1;
                else if (addrb[2:0] == 1) doutb <= reg2;
                else if (addrb[2:0] == 2) doutb <= reg3;
                else if (addrb[2:0] == 3) doutb <= reg4;
                else if (addrb[2:0] == 4) doutb <= reg5;
                else if (addrb[2:0] == 5) doutb <= reg6;
                else if (addrb[2:0] == 6) doutb <= reg7;
                else if (addrb[2:0] == 7) doutb <= reg8;
                if (counter >= 3) state <= IDLE;
                else
                    counter <= counter + 1;
            end

            WRITEB:
            begin
                busym <= 1;
//                regs[addrb[2:0]] <= databus;
```

```
                if (addrb[2:0] == 0) reg1 <= databus;
                else if (addrb[2:0] == 1) reg2 <= databus;
                else if (addrb[2:0] == 2) reg3 <= databus;
                else if (addrb[2:0] == 3) reg4 <= databus;
                else if (addrb[2:0] == 4) reg5 <= databus;
                else if (addrb[2:0] == 5) reg6 <= databus;
                else if (addrb[2:0] == 6) reg7 <= databus;
                else if (addrb[2:0] == 7) reg8 <= databus;
                state <= IDLE;
            end

            READM:
            begin
                busyb <= 1;
//                doutm <= regs[addrm];
                if (addrm == 0) doutm <= reg1;
                else if (addrm == 1) doutm <= reg2;
                else if (addrm == 2) doutm <= reg3;
                else if (addrm == 3) doutm <= reg4;
                else if (addrm == 4) doutm <= reg5;
                else if (addrm == 5) doutm <= reg6;
                else if (addrm == 6) doutm <= reg7;
                else if (addrm == 7) doutm <= reg8;
//                state <= IDLE;
                if (counter >= 3) state <= IDLE;
                else
                    counter <= counter + 1;
            end

            WRITEM:
            begin
                busyb <= 1;
//                regs[addrm] <= datamod;
                if (addrm == 0) reg1 <= datamod;
                else if (addrm == 1) reg2 <= datamod;
                else if (addrm == 2) reg3 <= datamod;
                else if (addrm == 3) reg4 <= datamod;
                else if (addrm == 4) reg5 <= datamod;
                else if (addrm == 5) reg6 <= datamod;
                else if (addrm == 6) reg7 <= datamod;
                else if (addrm == 7) reg8 <= datamod;
                state <= IDLE;
            end

            default: state <= IDLE;

        endcase

        if (reset) state <= RESET;
```

```
    end
endmodule
```

1.7 ModuleSkeleton.v

```
module ModuleSkeleton(reset, clk, addr, data, write, enable, busy);
input reset;
input clk;
// default bus wires
output [2:0] addr;
inout [7:0] data;
reg [7:0] dataOut;
reg dataEn;
output write;
output enable;
input busy;
reg oldBusy;

reg enable;
reg [2:0] addr;
reg write;

// local copies of registers
// !!! Change names !!!
reg [7:0] curReg0;
reg [7:0] curReg1;
reg [7:0] curReg2;

// state
reg [3:0] state;

// reg parameters
// !!! Change these names to correct ones for your module !!!
parameter REG_0   = 0;
parameter REG_1   = 1;
parameter REG_2   = 2;

// states
// !!! Same here !!!
parameter S_IDLE    = 0;
parameter S_GETREG0 = 1;
parameter S_GETREG1 = 2;
parameter S_GETREG2 = 3;
parameter S_RESET   = 4;

assign data = dataEn ? dataOut : 8'hz;

always @ (posedge clk)
begin
// do updates here
// !!! do any io update you need here !!!
// !!! change the state updater code as fit !!!
```

```
case (state)

S_IDLE:
begin
// negedge busy, update our local reg copy
enable <= 0;
if (oldBusy && !busy)
begin
state <= S_GETREG0;
addr <= REG_0;
enable <= 1;
end
end

S_GETREG0:
begin
// reg0
curReg0 <= data;
state <= S_GETREG1;
addr <= REG_1;
end

S_GETREG1:
begin
// reg1
curReg1 <= data;
state <= S_GETREG2;
addr <= REG_2;
end

S_GETREG2:
begin
// reg2
curReg2 <= data;
state <= S_IDLE;
end

S_RESET:
begin
dataOut <= 0;
dataEn <= 0;
curReg0 <= 0;
curReg1 <= 0;
curReg2 <= 0;
state <= S_IDLE;
write <= 0;
enable <= 0;
end

default: state <= S_IDLE;
```

```
endcase
oldBusy <= busy;
if (reset) state <= S_RESET;
end

endmodule
```

1.8 ModuleMotor.v

```
module ModuleMotor(clk, reset, addr, data, write, enable, busy, encVel, motDir, motEn);

    input reset;
    input clk;

//    output [7:0] leds;
//    reg [7:0] leds;

    output [2:0] addr;
    reg [2:0] addr;
    inout [7:0] data;
    reg [7:0] dataOut;
    reg dataEn;
    output write;
    reg write;
    output enable;
    reg enable;
    input busy;
    input [7:0] encVel;

    output motDir;
    reg motDir;
    output motEn;
    reg motEn;

    reg oldBusy;

    reg [7:0] pwmDiv;
    reg [7:0] pwmCount;
    reg [7:0] curDir;
    reg [7:0] curVel;

    reg [7:0] outDir;
    reg [7:0] outVel;

    reg [4:0] state;
    reg [31:0] feedbackUpdate;
    reg feedbackEn;

    parameter PWM_DIV   = 128;

    parameter REG_VEL   = 0;
    parameter REG_DIR   = 1;
    parameter REG_CUR   = 2;
    parameter REG_FBACK = 3;

    parameter S_IDLE    = 0;
```

```
    parameter S_GETVEL1 = 1;
    parameter S_GETVEL2 = 2;
    parameter S_GETVEL3 = 3;
    parameter S_CREG    = 4;
    parameter S_GETDIR1 = 5;
    parameter S_GETDIR2 = 6;
    parameter S_GETDIR3 = 7;
    parameter S_CREG1   = 8;
    parameter S_GETFBK1 = 9;
    parameter S_GETFBK2 = 10;
    parameter S_GETFBK3 = 11;

    assign data = dataEn ? dataOut : 8'hz;


    always @ (posedge clk)
    begin
        if (reset)
        begin
            dataOut <= 0;
            dataEn <= 0;
            motDir <= 0;
            motEn <= 0;
            curDir <= 0;
            curVel <= 0;
            pwmCount <= 0;
            pwmDiv <= 0;
            state <= S_IDLE;
            enable <= 0;
            write <= 0;
            addr <= 0;
//            leds <= 0;
            oldBusy <= 0;
            feedbackEn <= 0;
            outDir <= 0;
            outVel <= 0;
            feedbackUpdate <= 0;
        end
        else
        begin
//            leds <= curVel;


            if (feedbackEn)
            begin
                if (feedbackUpdate>10000000)
                begin
                    feedbackUpdate <= 0;
                    outDir <= curDir;
                    if ((encVel<curVel) && (outVel<255))
```

```
                        outVel <= outVel + 1;
                    else
                    if ((encVel>curVel) && (outVel>0))
                        outVel <= outVel - 1;
                end
                else
                   feedbackUpdate <= feedbackUpdate + 1;
            end
            else
            begin
                outDir <= curDir;
                outVel <= curVel;
            end

            pwmDiv <= pwmDiv + 1;
            if (pwmDiv==PWM_DIV)
            begin
                pwmCount <= pwmCount + 1;
                pwmDiv <= 0;
            end
            motEn <= (pwmCount<outVel);
            motDir <= outDir[0];

            case (state)
                S_IDLE:
                begin
enable <= 0;
                    if (oldBusy && (!busy))
                    begin
                        state <= S_GETVEL1;
                        addr <= REG_VEL;
                        enable <= 1;
                    end
                end

                S_GETVEL1:
                begin
                    state <= S_GETVEL2;
                end

                S_GETVEL2:
                begin
                    state <= S_GETVEL3;
                end

                S_GETVEL3:
                begin
                    curVel <= data;
                    state <= S_CREG;
                end
```

```
            S_CREG:
            begin
                addr <= REG_DIR;
                state <= S_GETDIR1;
            end

            S_GETDIR1:
            begin
                state <= S_GETDIR2;
            end

            S_GETDIR2:
            begin
                curDir <= data;
                state <= S_GETDIR3;
            end

            S_GETDIR3:
            begin
                state <= S_CREG1;
            end

            S_CREG1:
            begin
                addr <= REG_FBACK;
                state <= S_GETFBK1;
            end

            S_GETFBK1:
            begin
                state <= S_GETFBK2;
            end

            S_GETFBK2:
            begin
                feedbackEn <= data[0];
                state <= S_GETFBK3;
            end

            S_GETFBK3:
            begin
                state <= S_IDLE;
            end
        endcase

        oldBusy <= busy;
    end
end
```

```
endmodule
```

1.9 ModuleEncoder.v

```
module ModuleEncoder(clk, reset, addr, data, write, enable, busy, encVel, encoder);

    input reset;
    input clk;

//    output [7:0] leds;
//    reg [7:0] leds;

    output [2:0] addr;
    reg [2:0] addr;
    inout [7:0] data;
    reg [7:0] dataOut;
    reg dataEn;
    output write;
    reg write;
    output enable;
    reg enable;
    input busy;

    input encoder;
    output encVel;

    reg oldBusy;

    reg [15:0] encCount;
    reg [15:0] oldEncCount;
    reg [15:0] encWriteCount;
    reg [31:0] encWriteVel;
    reg [7:0] encVel;

    reg [4:0] state;

    parameter PWM_DIV   = 128;

    parameter REG_ENCHI = 0;
    parameter REG_ENCLO = 1;
    parameter REG_VEL   = 2;

    parameter S_IDLE    = 0;
    parameter S_SETENCHI= 1;
    parameter S_CREG0   = 2;
    parameter S_SETENCLO= 3;
    parameter S_CREG1   = 4;
    parameter S_SETVEL  = 5;

    assign data = write ? dataOut : 8'hz;
```

```
    always @ (posedge encoder or posedge reset)
    begin
        if (reset)
            encCount <= 0;
        else
            encCount <= encCount + 1;
    end

    always @ (posedge clk)
    begin
        if (reset)
        begin
            dataOut <= 0;
            dataEn <= 0;
            state <= S_IDLE;
            enable <= 0;
            write <= 0;
            addr <= 0;
            encWriteCount <= 0;
            oldBusy <= 0;
            encVel <= 0;
            oldEncCount <= 0;
            encWriteVel <= 0;
        end
        else
        begin

            encWriteCount <= encWriteCount + 1;
            encWriteVel <= encWriteVel + 1;
            if (encWriteVel==14000000)
            begin
                encVel <= ((encCount - oldEncCount) >> 4);
                oldEncCount <= encCount;
                encWriteVel <= 0;
            end

            case (state)
                S_IDLE:
                begin
enable <= 0;
                    write <= 0;
                    if ((!busy) && (oldBusy | (encWriteCount > 8192)))
                    begin
                        encWriteCount <= 0;
                        state <= S_SETENCHI;
                        addr <= REG_ENCHI;
                        write <= 1;
                        enable <= 1;
                    end
                end
```

```
                S_SETENCHI:
                begin
                    dataOut <= encCount[15:8];
                    state <= S_CREG0;
                end

                S_CREG0:
                begin
                    state <= S_SETENCLO;
                    addr <= REG_ENCLO;
                end

                S_SETENCLO:
                begin
                    dataOut <= encCount[7:0];
                    state <= S_CREG1;
                end

                S_CREG1:
                begin
                    state <= S_SETVEL;
                    addr <= REG_VEL;
                end

                S_SETVEL:
                begin
                    dataOut <= encVel;
                    state <= S_IDLE;
                end

            endcase

            oldBusy <= busy;
        end
    end


endmodule
```

1.10 ModuleServo.v

```
module ModuleServo(clk, reset, addr, data, write, enable, busy, servo);

    input reset;
    input clk;

//    output [7:0] leds;
//    reg [7:0] leds;

    output [2:0] addr;
    reg [2:0] addr;
    inout [7:0] data;
    reg [7:0] dataOut;
    reg dataEn;
    output write;
    reg write;
    output enable;
    reg enable;
    input busy;

    output [7:0] servo;
    reg [7:0] servo;

    reg oldBusy;

    reg [15:0] pwmDiv;
    reg [7:0] pwmCount;

    reg [7:0] curVel0;
    reg [7:0] curVel1;

    reg [4:0] state;

    parameter PWM_DIV   = 2048;

    parameter REG_VEL0  = 0;
    parameter REG_VEL1  = 1;

    parameter S_IDLE    = 0;
    parameter S_GETVEL01= 1;
    parameter S_GETVEL02= 2;
    parameter S_GETVEL03= 3;
    parameter S_CREG0   = 4;
    parameter S_GETVEL11= 5;
    parameter S_GETVEL12= 6;
    parameter S_GETVEL13= 7;
    parameter S_CREG1   = 8;
```

```
    assign data = dataEn ? dataOut : 8'hz;


    always @ (posedge clk)
    begin
        if (reset)
        begin
            dataOut <= 0;
            dataEn <= 0;
            servo <= 0;
            curVel0 <= 16;
            curVel1 <= 16;
            pwmCount <= 0;
            pwmDiv <= 0;
            state <= S_IDLE;
            enable <= 0;
            write <= 0;
            addr <= 0;
//            leds <= 0;
            oldBusy <= 0;
        end
        else
        begin
//             leds <= curVel;
            pwmDiv <= pwmDiv + 1;
            if (pwmDiv==PWM_DIV)
            begin
                pwmCount <= pwmCount + 1;
                pwmDiv <= 0;
            end
            servo[0] <= (pwmCount<curVel0);
            servo[1] <= (pwmCount<curVel1);

            case (state)
                S_IDLE:
                begin
enable <= 0;
                    if (oldBusy && (!busy))
                    begin
                        state <= S_GETVEL01;
                        addr <= REG_VEL0;
                        enable <= 1;
                    end
                end

                S_GETVEL01:
                begin
                    state <= S_GETVEL02;
                end
```

```
                S_GETVEL02:
                begin
                    state <= S_GETVEL03;
                end

                S_GETVEL03:
                begin
                    curVel0 <= data;
                    state <= S_CREG0;
                end

                S_CREG0:
                begin
//                      state <= S_IDLE;
                    state <= S_GETVEL11;
                    addr <= REG_VEL1;
                end

                S_GETVEL11:
                begin
                    state <= S_GETVEL12;
                end

                S_GETVEL12:
                begin
                    curVel1 <= data;
                    state <= S_GETVEL13;
                end

                S_GETVEL13:
                begin
                    state <= S_CREG1;
                end

                S_CREG1:
                begin
                    state <= S_IDLE;
                end

            endcase

            oldBusy <= busy;
        end
    end


endmodule
```

1.11 ModuleDigitalIn.v

```
module ModuleDigitalIn(clk, reset, addr, data, write, enable, busy, digitalPort);

    input reset;
    input clk;

    output [2:0] addr;
    inout [7:0] data;
    reg [7:0] dataOut;
    reg dataEn;
    output write;
    output enable;
    input busy;
    reg oldBusy;
    reg [2:0] addr;
    reg write;
    reg enable;


    input [7:0] digitalPort;
    reg [7:0] oldDigitalPort;
    reg [15:0] counter;

    reg [3:0] state;

    parameter REG_VAL   = 0;

    parameter S_IDLE    = 0;
    parameter S_SETVAL  = 1;
    parameter S_RESET   = 2;

    assign data = write ? dataOut : 8'hz;

    always @ (posedge clk)
    begin
        case (state)

            S_IDLE:
            begin
                write <= 0;
                enable <= 0;

                if (!busy && (digitalPort!=oldDigitalPort || counter==65000))
                begin
                counter <= 0;
                state <= S_SETVAL;
                addr <= REG_VAL;
                write <= 1;
```

```
                enable <= 1;
                end
            end

            S_SETVAL:
            begin
                dataOut <= digitalPort;
                state <= S_IDLE;
            end

            S_RESET:
            begin
                dataOut <= 0;
                dataEn <= 0;
                state <= S_IDLE;
                write <= 0;
                enable <= 0;
                oldDigitalPort <= 0;
                counter <= 0;
            end

            default: state <= S_IDLE;

        endcase

        oldDigitalPort <= digitalPort;

        if (reset) state <= S_RESET;
    end


endmodule
```

1.12 Debouncer.v

```
//////////////////////////////////////////////////////////////////////
////                                                              ////
//// debouncer.v                                                  ////
////                                                              ////
//// This file is part of the boundaries opencores effort.        ////
//// <http://www.opencores.org/cores/boundaries/>                 ////
////                                                              ////
//// Module Description:                                          ////
//// Debounce a mechanical switch or contact.                     ////
////                                                              ////
//// To Do:                                                       ////
//// Verify in silicon.                                           ////
////                                                              ////
//// Author(s):                                                   ////
//// - Shannon Hill                                               ////
////                                                              ////
//////////////////////////////////////////////////////////////////////
////                                                              ////
//// Copyright (C) 2004 Shannon Hill and OPENCORES.ORG             ////
////                                                              ////
//// This source file may be used and distributed without         ////
//// restriction provided that this copyright statement is not    ////
//// removed from the file and that any derivative work contains  ////
//// the original copyright notice and the associated disclaimer. ////
////                                                              ////
//// This source file is free software; you can redistribute it   ////
//// and/or modify it under the terms of the GNU Lesser General   ////
//// Public License as published by the Free Software Foundation; ////
//// either version 2.1 of the License, or (at your option) any   ////
//// later version.                                               ////
////                                                              ////
//// This source is distributed in the hope that it will be       ////
//// useful, but WITHOUT ANY WARRANTY; without even the implied   ////
//// warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR      ////
//// PURPOSE. See the GNU Lesser General Public License for more  ////
//// details.                                                     ////
////                                                              ////
//// You should have received a copy of the GNU Lesser General    ////
//// Public License along with this source; if not, download it   ////
//// from <http://www.opencores.org/lgpl.shtml>                   ////
////                                                              ////
//////////////////////////////////////////////////////////////////////
//
// $Id: debouncer.v,v 1.1 2004/07/07 12:41:17 esquehill Exp $
//
// CVS Revision History
//
```

```
// $Log: debouncer.v,v $
// Revision 1.1  2004/07/07 12:41:17  esquehill
// Initial.
//
//
//
module Debouncer(clk_i, rst_i, button_o, button_i);

    parameter CW = 8;

    input        rst_i;
    input        clk_i;     // 1us period for a (1<<CW)us debounce interval
    input        button_i;
    output       button_o;

    reg          button_1;
    reg          button_2;
    reg [CW-1:0] count;
    reg          button_o;

    wire         changed  =   button_2 ^ button_o;

    always @( posedge clk_i or posedge rst_i )
    if( rst_i )
    begin
        button_1 <= 1'b0;
        button_2 <= 1'b0;
        count    <= {CW{1'b0}};
        button_o <= 1'b0;
    end
    else
    begin
        button_1 <= button_i;       // async input
        button_2 <= button_1;
        count    <= count + 1'b1;
        casex( { changed, &count } )
            2'b0x:  count    <= {CW{1'b0}};    // output == input; reset counter
            2'b10: ;                           // output != input; wait for debounce timeo
            2'b11:  button_o <= button_2;      // copy input to output
            default: ;
        endcase
    end

endmodule
```

1.13 ModuleDigitalOut.v

```
module ModuleDigitalOut(clk, reset, addr, data, write, enable, busy, port);

    input reset;
    input clk;

    output [2:0] addr;
    reg [2:0] addr;
    inout [7:0] data;
    reg [7:0] dataOut;
    reg dataEn;
    output write;
    reg write;
    output enable;
    reg enable;
    input busy;

    output [7:0] port;
    reg [7:0] port;

    reg oldBusy;

    reg [7:0] curVal;

    reg [4:0] state;

    parameter REG_VEL0  = 0;

    parameter S_IDLE    = 0;
    parameter S_GETVEL01= 1;
    parameter S_GETVEL02= 2;
    parameter S_GETVEL03= 3;
    parameter S_CREG0   = 4;

    assign data = dataEn ? dataOut : 8'hz;


    always @ (posedge clk)
    begin
        if (reset)
        begin
            dataOut <= 0;
            dataEn <= 0;
            port <= 0;
            curVal <= 16;
            state <= S_IDLE;
            enable <= 0;
            write <= 0;
```

```
            addr <= 0;
//            leds <= 0;
            oldBusy <= 0;
        end
        else
            port <= curVal;

            case (state)
                S_IDLE:
                begin
enable <= 0;
                    if (oldBusy && (!busy))
                    begin
                        state <= S_GETVEL01;
                        addr <= REG_VEL0;
                        enable <= 1;
                    end
                end

                S_GETVEL01:
                begin
                    state <= S_GETVEL02;
                end

                S_GETVEL02:
                begin
                    state <= S_GETVEL03;
                end

                S_GETVEL03:
                begin
                    curVal <= data;
                    state <= S_CREG0;
                end

                S_CREG0:
                begin
                      state <= S_IDLE;
                end

            endcase

            oldBusy <= busy;
        end


endmodule
```

1.14 ModuleAnalogIn.v

```
module ModuleAnalogIn(clk, reset, addr, data, write, enable, busy, mux, adc_data, adc_ce,
input reset;
input clk;
// default bus wires
output [2:0] addr;
inout [7:0] data;
reg [7:0] dataOut;
reg dataEn;
output write;
output enable;
input busy;
reg oldBusy;

reg enable;
reg [2:0] addr;
reg write;

output [3:0] mux;
reg [3:0] mux;

input [7:0] adc_data;
input adc_stat;
output adc_ce, adc_rw;
reg adc_ce, adc_rw;

   reg [7:0] counter;

// local copies of registers
reg [7:0] regMux;
reg [7:0] regVal;

// state
reg [3:0] state;

// reg parameters
parameter REG_VAL = 0;

// states
parameter S_IDLE    = 0;
parameter S_A2D_START = 1;
parameter S_A2D_STATUS_HI = 2;
parameter S_A2D_STATUS_LO = 3;
parameter S_A2D_READ_WAIT = 4;
parameter S_A2D_READ = 5;
parameter S_RESET   = 6;
    parameter S_WRITE_VAL = 7;
```

```
assign data = write ? dataOut : 8'hz;

always @ (posedge clk)
begin
// do updates here
case (state)

S_IDLE:
begin
// negedge busy, update our local reg copy
counter <= 0;
enable <= 0;
write <= 0;
dataEn <= 0;
                if (!busy && oldBusy) //
begin
state <= S_A2D_START;
     adc_rw <= 1;
          adc_ce <= 1;
                    write <= 1;
enable <= 1;   //Don't need to check this yet
end
end

        S_A2D_START:
            begin
adc_rw <= 0;
     adc_ce <= 0;
                if (counter<48)
                    counter <= counter + 1;
                else
          state <= S_A2D_STATUS_HI;
            end

        S_A2D_STATUS_HI:
            begin
adc_rw <= 1;
     if (adc_stat) state <= S_A2D_STATUS_LO;
            end

        S_A2D_STATUS_LO:
            begin
                counter <= 0;
     if (!adc_stat) state <= S_A2D_READ_WAIT;
            end

        S_A2D_READ_WAIT:
            begin
                addr <= REG_VAL;
                if (counter<48)
```

```
                    counter <= counter + 1;
                else
                    state <= S_A2D_READ;
            end

        S_A2D_READ:
            begin
dataOut <= adc_data;
state <= S_IDLE;
            end

S_RESET:
begin
dataOut <= 0;
dataEn <= 0;
mux <= 0;
regMux <= 0;
regVal <= 0;
adc_rw <= 1;  // should default to write, maybe?
adc_ce <= 1;  // Should default to off...
oldBusy <= 0;
state <= S_IDLE;
write <= 0;
enable <= 0;
                counter <= 0;
end

default: state <= S_IDLE;
endcase
oldBusy <= busy;
if (reset) state <= S_RESET;
end

endmodule
```

1.15 ModuleCharacterLCD.v

```
module ModuleCharacterLCD(clk, reset, addr, data, write, enable, busy, db, rs, rw, enableL
input reset;
input clk;
// default bus wires
output [2:0] addr;
inout [7:0] data;
reg [7:0] dataOut;
reg dataEn;
output write;
output enable;
input busy;
reg oldBusy;
output [7:0] db;
output rs, rw, enableLCD;

reg [7:0] db;
reg rs, enableLCD;
reg [15:0] counter;
reg [4:0] newline;
assign rw = 0;  // only writing to the LCD

reg enable;
reg [2:0] addr;
reg write;

reg [7:0] characterReg;

// state
reg [4:0] state;

// reg Parameters
parameter CHAR_REG   = 0;


// states
parameter S_IDLE    = 0;
parameter S_GETREG = 1;
parameter S_GETCHAR = 2;
parameter S_ENABLEC  = 3;
parameter S_CLEAR    = 4;
parameter S_ENABLEW  = 5;
parameter S_WRITE    = 6;
parameter S_RESET0   = 7;
parameter S_RESET1   = 8;
parameter S_RESET2   = 9;
parameter S_RESET3   = 10;
   parameter S_RESET4   = 11;
```

```
   parameter S_RESET5   = 12;
   parameter S_RESET6   = 13;
   parameter S_RESET7   = 14;
   parameter S_RESET8   = 15;
   parameter S_NEWLN1   = 16;

assign data = dataEn ? dataOut : 8'hz;

always @ (posedge clk)
begin

case (state)

S_IDLE:
begin
// negedge busy, update our local reg copy
counter <= 0;
enableLCD <= 0;
rs <= 0;
if (newline >= 16) begin
   newline <= 0;
   state <= S_NEWLN1;
   end
else if (oldBusy && !busy)
begin
state <= S_GETREG;
addr <= CHAR_REG;
enable <= 1;
end
   else enable <= 0;
end

S_GETREG:
begin
if (counter >= 2) begin
enable <= 0;
counter <= 0;
characterReg <= data;
state <= S_GETCHAR;
end
else counter <= counter + 1;
end

S_GETCHAR:
begin

      if (characterReg == 0) begin
          rs <= 0;
          if (counter == 1) begin
             newline <= 0;
```

```
          state <= S_ENABLEC;
          counter <= 0;
          end
       else counter <= counter + 1;
       end
   else begin
       rs <= 1;
       if (counter == 1) begin
           newline <= newline + 1;
           state <= S_ENABLEW;
           counter <= 0;
           end
       else counter <= counter + 1;
       end
end

S_ENABLEC:
begin
    enableLCD <= 1;
    db <= 8'b00000001;
    if (counter >= 7) begin
       enableLCD <= 0;
       state <= S_CLEAR;
       counter <= 0;
    end
    else counter <= counter + 1;
end

S_CLEAR:
begin
    if (counter >= 44329) state <= S_IDLE;
    else counter <= counter + 1;
end

S_ENABLEW:
begin
    enableLCD <= 1;
    db <= characterReg;
          if (counter >= 7) begin
             enableLCD <= 0;
             state <= S_CLEAR;
             counter <= 0;
          end
          else counter <= counter + 1;
      end

      S_WRITE:
      begin
      if (counter >= 1246) state <= S_IDLE;
         else counter <= counter + 1;
```

```
        end

S_RESET0:
begin
newline <= 0;
dataOut <= 0;
dataEn <= 0;
write <= 0;
enable <= 0;
enableLCD <= 0;
      counter <= 0;
   rs <= 0;
   state <= S_RESET1;
end

S_RESET1:
                      state <= S_RESET2;

        S_RESET2:
        begin
            if (counter < 7) enableLCD <= 1;
            else enableLCD <= 0;
            db <= 8'b00000001;
            if (counter >= 44324) begin
            state <= S_RESET3;
            counter <= 0;
            end
            else counter <= counter + 1;
        end




        S_RESET3:
        begin
            if (counter < 7) enableLCD <= 1;
            else enableLCD <= 0;
            db <= 8'b00111111;
            if (counter >= 1082) state <= S_RESET4;
            else counter <= counter + 1;
        end

        S_RESET4:
        begin
            enableLCD <= 0;
            counter <= 0;
            state <= S_RESET5;
        end

        S_RESET5:
```

```
            begin
                if (counter < 7) enableLCD <= 1;
                else enableLCD <= 0;
                db <= 8'b00001100;
                if (counter >= 1082) state <= S_RESET6;
                else counter <= counter + 1;
            end

            S_RESET6:
            begin
                enableLCD <= 0;
                counter <= 0;
                state <= S_RESET7;
            end

            S_RESET7:
            begin
                if (counter < 7) enableLCD <= 1;
                else enableLCD <= 0;
                db <= 8'b00000110;
                if (counter >= 1082) state <= S_RESET8;
                else counter <= counter + 1;
            end

            S_RESET8:
            begin
               enableLCD <= 0;
               db <= 8'b00000000;
               counter <= 0;
               state <= S_IDLE;
           end

            S_NEWLN1:
            begin
              enableLCD <= 1;
              db <= 8'b11000000;
              if (counter >= 7) begin
                 enableLCD <= 0;
                 state <= S_WRITE;
                 counter <= 0;
                 end
             else counter <= counter + 1;
             end


default: state <= S_IDLE;
endcase
oldBusy <= busy;
if (reset) state <= S_RESET0;
end
```

```
endmodule
```