3D MOTION CONTROLLER

RedSox: Anundson, Chen, Ghosh, Lin, Yang
Concept
Goal and Motivation
Specification and Functionality

1. **Gesture capture**
   - Interface with gyro and accelerometer via SPI interface
   - Interpret data from the gyro and accelerometer to be used to control different demo components
     - Basic control using raw gyro or accelerometer data
     - Filter and combine accelerometer and gyro data into a set of orientation coordinates

2. **2D maze to demonstrate gesture control**
   - Display maze
   - Implement collision detection

3. **Simple 3D rendering**
   - Draw a line on a VGA display between two arbitrary points
   - Calculate 2D translations of corners of a cube moving in a 3D space
High-Level Design 2

- Gyro
- Accelerometer
- VGA Monitor
- SPI Interface
- SPI Interface
- Orientation
- Orientation Calculations
- Wireframe cube demo
- Acceleration

FPGA Design
Task 1: Interface with gyro and accelerometer via SPI
Gyro and Accelerometer Interfacing Pt 1

- SPI
  - Master-slave
  - Master generates clock signal
  - Master to slave via MOSI
  - Slave to master via MISO (Full duplex)
  - Slave select to “wake up” slave
ACL Interfacing

```verilog
wire doneConfigure;
wire done;
wire transmitt;

// Implementation

// Generates a 1MHz Data Transfer Request Signal
// -------------------------------
ClkBtv_1Mhz genStart(
    .CLK(CLK),
    .RST(RST),
    .CLKOUT(start)
);

// Controls SPI Interface, Stores Received Data, and Controls Data to Send
// -----------------------------------------------
SPImaster CO(
    .rst(RST),
    .start(start),
    .clk(CLK),
    .transmit(transmit),
    .txdata(TxBuffer),
    .rxdata(RxBuffer),
    .done(done),
    .x_axis_data(xAxis),
    .y_axis_data(yAxis),
    .z_axis_data(zAxis)
);

// Produces Timing Signal, Reads ACL Data, and Writes Data to ACL
// -------------------------------------------------------------
SPIInterface CI(
    .sd(SDI),
    .sdo(SDO),
    .rst(RST),
    .clk(CLK),
    .sclk(SCL),
    .txbuffer(TxBuffer),
    .rxbuffer(RxBuffer),
    .done_out(done),
    .transmit(transmit)
);

// Enables/Disables PmodACL Communication
// ----------------------------------------
slaveSelect C2(
    .clk(CLK),
    .ss(SS),
    .done(done),
    .transmit(transmit),
    .rst(RST)
);
```

GYRO Interfacing

```verilog
output [15:0] x_axis_data;
output [15:0] y_axis_data;
output [15:0] z_axis_data;
input mosi;
output sclk;
output slaveSelect;
wire begin_transmission;
wire end_transmission;
wire [7:0] send_data;
wire [7:0] received_data;
wire [7:0] temp_data;

// Serial Port Interface Controller
// ----------------------------------
master_interface gyro_master(
    .begin_transmission(begin_transmission),
    .end_transmission(end_transmission),
    .send_data(send_data),
    .received_data(received_data),
    .clk(clk),
    .rst(rst),
    .slave_select(slaveSelect),
    .start(start),
    .temp_data(temp_data),
    .x_axis_data(x_axis_data),
    .y_axis_data(y_axis_data),
    .z_axis_data(z_axis_data)
);

// Serial Port Interface
// ----------------------
spi_interface gyro_spi(
    .begin_transmission(begin_transmission),
    .slave_select(slaveSelect),
    .send_data(send_data),
    .received_data(received_data),
    .mosi(mosi),
    .clk(clk),
    .rst(rst),
    .end_transmission(end_transmission),
    .mosi(mosi),
    .clk(clk),
    .sclk(sclk)
);
```
Task 2: Combine gyro and accelerometer data into orientation using a complementary filter
Gyro and Accelerometer Interfacing Pt 2
Gyro and Accelerometer Interfacing Pt 2

Gyroscope
- SPI Interface
- SPI Interface Controller

Accelerometer
- SPI Interface
- SPI Interface Controller

CompFilter
- 16-bit xyz gyro data
  - angular velocity
- 10-bit xyz acc data
  - gravity projection

\[ \text{gyro\_degree} \leftarrow \text{gyro\_degree} + \int \text{gyro\_data} \, \text{data} \cdot \text{dt} \]

\[ \text{acc\_degree} \leftarrow \arctan(\text{acc\_x}, \text{acc\_y}) \]

E.g. \( x\_\text{degree} = 0.98 \cdot x\_\text{gyro} + 0.02 \cdot x\_\text{acc} \)

16-bit xyz orientation data
Arbitrary Line Drawing

- Bresenham’s line algorithm
  - Calculates which pixels to illuminate between two points
  - Can be implemented using integer math
- Dual port RAM used as VGA display buffer
  - Utilizes the BRAM resources in the FPGA
- Module uses FIFOs to store multiple sets of line coordinates to be drawn for each screen refresh
Arbitrary Line Drawing

Diagram:
- VGA Display Logic
- Display State Machine
- FIFOs for x0_coord, x1_coord, y0_coord, y1_coord
- Bresenham
- Clear Buffer
- Refresh screen
- VGA Buffer
- VCOUNTER
- Hcounter
- Pixel_on
Arbitrary Line Drawing

- Algorithm works very well for a hardware application
  - Integer math
  - Simple state machine

- Trade-offs
  - Uses a large buffer to store screen buffer
    - Design successfully implements line drawing at 640x480 resolution
    - Consumes a large amount of area
    - Can only store “on” or “off” so cannot store different color lines
    - Possible alternatives for future development
      - Use a larger FPGA device
      - Use off-chip memory to store screen buffer
Verilog Example: Bresenham

module bresenham(
    input  i_clk,
    input  i_reset,
    input  [P_XCOORD_W-1:0] i_x0,
    input  [P_XCOORD_W-1:0] i_x1,
    input  [P_YCOORD_W-1:0] i_y0,
    input  [P_YCOORD_W-1:0] i_y1,
    input  i_load_vals,
    output reg  [P_XCOORD_W-1:0] o_x_val,
    output reg  [P_YCOORD_W-1:0] o_y_val,
    output     o_vals_rdy,
    output     o_waiting
);

// Parameters
parameter P_XCOORD_W = 11;
parameter P_YCOORD_W = 11;
Verilog Example: Bresenham

```verilog
// Local parameters
localparam P_ERROR_W = (P_X_COORD_W>P_Y_COORD_W) ? P_X_COORD_W : P_Y_COORD_W;
localparam STATE_WAITING = 0;
localparam STATE_SETUP_IS_STEEP = 1;
localparam STATE_SETUP_REV_COORDS = 2;
localparam STATE_SETUP_ERROR_AND_STEP = 3;
localparam STATE_DRAWING = 4;

// State variables
reg [2:0] curr_state, next_state;

// X and Y registers
reg signed [P_X_COORD_W-1:0] x, x0, x1;
reg signed [P_Y_COORD_W-1:0] y, y0, y1;
// X and Y change registers
reg signed [P_X_COORD_W-1:0] delta_x;
reg signed [P_XCOORD_W-1:0] delta_y;
// Error register
reg signed [P_ERROR_W-1:0] error;
// Step value for Y coordinate of line
reg signed [1:0] ystep;
// Control signals
reg steep;
reg vals_rdy;
reg waiting;
```
Verilog Example: Bresenham

```verilog
// Generate output values
always @(posedge i_clk)
begin
    if (i_reset) begin
        x <= 'b0;
x0 <= 'b0;
x1 <= 'b0;
y <= 'b0;
y0 <= 'b0;
y1 <= 'b0;
error <= 'b0;
curr_state <= STATE_WAITING;
    end else begin
        vals_rdy <= 0;
x0 <= x0;
x1 <= x1;
y0 <= y0;
y1 <= y1;
error <= error;
curr_state <= next_state;
    end

    case (curr_state)
        STATE_WAITING : begin
            // Load input coordinates into registers
            if (i_load_vals) begin
                x0 <= i_x0;
x1 <= i_x1;
y0 <= i_y0;
y1 <= i_y1;
error <= 'b0;
            end
        end
end
```
Verilog Example: Bresenham

STATE_SETUP_IS_STEEP : begin
  // Determine if the slope of the line is steep
  // (i.e. abs(slope) > 1)
  if (y1 > y0) begin
    if (x1 > x0) begin
      steep <= (y1-y0) > (x1-x0);
    end else begin
      steep <= (y1-y0) > (x0-x1);
    end
  end else begin
    if (x1 > x0) begin
      steep <= (y0-y1) > (x1-x0);
    end else begin
      steep <= (y0-y1) > (x0-x1);
    end
  end
end
Verilog Example: Bresenham

STATE_SETUP_REV_COORDS : begin
  // Adjust the coordinates to draw line with
  // a slope that is positive and less than 1
  if (steep) begin
    if (y0 > y1) begin
      x0 <= y1;
      x1 <= y0;
      y0 <= x1;
      y1 <= x0;
      delta_x <= y0-y1;
      delta_y <= (x1-x0 ? x1-x0 : x0-x1);
    end else begin
      x0 <= y0;
      x1 <= y1;
      y0 <= x0;
      y1 <= x1;
      delta_x <= y1-y0;
      delta_y <= (x1-x0 ? x1-x0 : x0-x1);
    end
  end else begin
    if (x0 > x1) begin
      y0 <= y1;
      y1 <= y0;
      delta_x <= x0-x1;
      delta_y <= (y1-y0 ? y1-y0 : y0-y1);
    end else begin
      delta_x <= x1-x0;
      delta_y <= (y1-y0 ? y1-y0 : y0-y1);
    end
  end
end
Verilog Example: Bresenham

```verilog
STATE__SETUP_ERROR_AND_STEP : begin
    // Set up the error, step values and initial values of x,y
    error <= delta_x >>> 1;
    ystep <= (y0<y1 ? 1 : -1);
    x   <= x0;
    y   <= y0;
end

STATE__DRAWING : begin
    // Write coordinates to output
    if (step) begin
        o_x_val <= y;
        o_y_val <= x;
    end else begin
        o_x_val <= x;
        o_y_val <= y;
    end

    x <= x+1;

    if (error-delta_y < 0) begin
        error <= error-delta_y + delta_x;
        y    <= y+ystep;
    end else begin
        error <= error-delta_y;
    end

    vals_rdy <= 1;
end
```
Verilog Example: Bresenham

// Next state logic
always @(curr_state, i_load_vals, x, x1)
begin
  next_state = curr_state;
  case(curr_state)
    STATE_WAITING : begin
      if (i_load_vals) begin
        next_state = STATE_SETUP_IS_STEEP;
      end
    end
    STATE_SETUP_IS_STEEP : begin
      next_state = STATE_SETUP_REVCOORDS;
    end
    STATE_SETUP_REVCOORDS : begin
      next_state = STATE_SETUP_ERROR_AND_STEP;
    end
    STATE_SETUP_ERROR_AND_STEP : begin
      next_state = STATE_DRAWING;
    end
    STATE_DRAWING : begin
      if (x == x1) begin
        next_state = STATE_WAITING;
      end
    end
  endcase
end
Wireframe Cube Display

- **Functionality**
  - Uses Gyro And Accelerometer Data as input to manipulate 3D cube size and position
  - Continually update display screen with image of cube
  - Simulate matrix multiplication calculations for cube rotations in 3D space
  - Values are pushed into line drawing buffer
  - Use 4 fold axis rotation over 3 or 2 fold rotation due to simplified calculation
Wireframe Cube Display

- Gyro
- Accelerometer
- Clock
- Sin & Cosine lookup table
- Cube Start Coordinates
- Draw Line Module
- Cos & Sin of X, Y, Z
- Matrix Multiplication broken down through incremental calculations cycles
- Output Modified Coordinates

Matrix Rotation Formulas:

\[
R_x(\theta) = \begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \theta & -\sin \theta \\
0 & \sin \theta & \cos \theta
\end{bmatrix}
\]

\[
R_y(\theta) = \begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{bmatrix}
\]

\[
R_z(\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
Wireframe Cube Display

Look up table for Values. The accuracy of values was limited to accuracy to the hundredth decimal place due to limits of the onboard memory.

```hier
if (calcbreak == 5) begin
  case (LineNum)
    4'b0000: begin
      PP[0] <= 10;
      PP[1] <= 10;
      PP[2] <= -10;
    JJ[0] <= 10;
    JJ[1] <= -10;
    JJ[2] <= -10;
  end

  4'b0001: begin
    assign outX = cos[degreesX];
    assign outY = cos[degreesY];
    assign outZ = cos[degreesZ];

  end
end
```

Insertion of Cube coordinates at starting position.
Calculations are cascaded through a series of intermediate steps. This was put in due to the limits of the FPGA. We could not squeeze in all the calculations needed in one cycle due to limits in DSP48A1s Slices.

Outputs of the lookup table and coordinates are fed into the calculations.
**2D Maze**

- **Functionality**
  - Takes the data input from the gyro
  - The VGA_Display module converts the data to the dimensions of the ball that moves through the maze
  - A 2D maze is displayed on the VGA
  - Collision Detection is implemented by the inputs taken from the gyro
  - The aim of the game is to move the ball to the end of the maze
2D Maze

- **Features of the Maze**
- A switch is provided which when turned on enables the maze to blink.
- Different portions of the maze appear at regular intervals hence making it tougher for the ball to complete the game.
- The color of the maze keeps changing throughout the game giving it a visual appeal.
2D Maze Game

- Gyro
- Accelerometer
- Switches
- VGA Monitor

Top:
- Data Select
- SPI Component

Task3:
- Maze and Ball Logic
  - Maze Portion Files
  - Ball Logic
  - Cursor ROM (Baseball sprite)
- VGA Controller
2D Maze Game

Verilog Code Snippet showing the Maze Display

```verilog
always @ (posedge clk) begin
  if (balls) begin
    // Define color codes
    if (mode) begin
      // Maze 1
      if (maze_eq_pos[1]) begin
        (red, green, blue) = 8'h0010100;
      end else begin
        (red, green, blue) = 8'h0000000;
      end
      // Maze 2
      if (maze_eq_pos[2]) begin
        (red, green, blue) = 8'h1000100;
      end else begin
        (red, green, blue) = 8'h0011000;
      end
      // Maze 3
      if (maze_eq_pos[3]) begin
        (red, green, blue) = 8'h0010011;
      end else begin
        (red, green, blue) = 8'h0001001;
      end
      // Maze 4
      if (maze_eq_pos[4]) begin
        (red, green, blue) = 8'h0010111;
      end else begin
        (red, green, blue) = 8'h0100001;
      end
      // Maze 5
      if (maze_eq_pos[5]) begin
        (red, green, blue) = 8'h0100010;
      end else begin
        (red, green, blue) = 8'h0110000;
      end
      // Maze 6
      if (maze_eq_pos[6]) begin
        (red, green, blue) = 8'h0110100;
      end else begin
        (red, green, blue) = 8'h1110000;
      end
      // Maze 7
      if (maze_eq_pos[7]) begin
        (red, green, blue) = 8'h1101000;
      end else begin
        (red, green, blue) = 8'h1110001;
      end
      // Maze 8
      if (maze_eq_pos[8]) begin
        (red, green, blue) = 8'h1111000;
      end else begin
        (red, green, blue) = 8'h1101001;
      end
      // Maze 9
      if (maze_eq_pos[9]) begin
        (red, green, blue) = 8'h1101001;
      end else begin
        (red, green, blue) = 8'h1110001;
      end
    end
  end
end
```

- **mode** refers to the switch that is responsible for the blinking maze.
- This part of the code is responsible for displaying the different colors of the maze, depending upon the range of the counter.
2D Maze Game

- Verilog Code Showing Collision Detection

```verilog
assign n1 = (hcounter == 0) && vcounter < 40; // hcounter < 40 && vcounter < 400;
assign n2 = (hcounter == 20 && vcounter < 20); // hcounter < 20 && vcounter < 200;
assign n3 = (hcounter == 20 && vcounter < 60); // hcounter < 20 && vcounter < 600;
assign n4 = (hcounter == 40 && vcounter < 148); // hcounter < 40 && vcounter < 148;
assign n5 = (hcounter == 104 && vcounter < 114); // hcounter < 104 && vcounter < 114;
assign n6 = (hcounter == 20 && vcounter < 388); // hcounter < 20 && vcounter < 388;
assign n7 = (hcounter == 230 && vcounter < 260); // hcounter < 230 && vcounter < 260;
assign n8 = (hcounter == 40 && vcounter < 28); // hcounter < 40 && vcounter < 28;
assign n9 = (hcounter == 801 && vcounter < 371); // hcounter < 801 && vcounter < 371;
assign n10 = (hcounter == 461 && vcounter < 491); // hcounter < 461 && vcounter < 491;
assign n11 = (hcounter == 391 && vcounter < 491); // hcounter < 391 && vcounter < 491;
assign n12 = (hcounter == 491 && vcounter < 491); // hcounter < 491 && vcounter < 491;
assign n13 = (hcounter == 470 && vcounter < 491); // hcounter < 470 && vcounter < 491;
assign n14 = (hcounter == 491 && vcounter < 491); // hcounter < 491 && vcounter < 491;
assign n15 = (hcounter == 1490 && vcounter < 590); // hcounter < 1490 && vcounter < 590;
assign n16 = (hcounter == 1490 && vcounter < 490); // hcounter < 1490 && vcounter < 490;
assign n17 = (hcounter == 520 && vcounter < 530); // hcounter < 520 && vcounter < 530;
assign n18 = (hcounter == 490 && vcounter < 490); // hcounter < 490 && vcounter < 490;
assign n19 = (hcounter == 1490 && vcounter < 490); // hcounter < 1490 && vcounter < 490;
assign n20 = (hcounter == 490 && vcounter < 19); // hcounter < 490 && vcounter < 19;
assign n21 = (hcounter == 490 && vcounter < 19); // hcounter < 490 && vcounter < 19;
assign n22 = (hcounter == 95 && vcounter < 63); // hcounter < 95 && vcounter < 63;
assign n23 = (hcounter == 95 && vcounter < 420); // hcounter < 95 && vcounter < 420;
assign enable = n1 || n2 || n3 || n4 || n5 || n7 || n8 || n9 || n10 || n11 || n12 || n13 || n14 || n15 || n17 || n18;
assign collision = enable;
```

// Collision detection
reg collision;
always @(w_ball, y_ball, ball_width)
begin
    stop_right = 0;
    stop_left = 0;
    stop_up = 0;
    stop_down = 0;

    if((x_ball + ball_width == 20) && (y_ball > 20-ball_width && y_ball < 460-1)) // n1
        if((x_ball + ball_width == 20) && (y_ball > 102-ball_width && y_ball < 460)) // n2
            if((x_ball + ball_width == 20) && (y_ball > 18-ball_width && y_ball < 460)) // n3
                if((x_ball + ball_width == 20) && (y_ball > 38-ball_width && y_ball < 460)) // n4
                    if((x_ball + ball_width == 20) && (y_ball > 94-ball_width && y_ball < 460)) // n5
                        if((x_ball + ball_width == 20) && (y_ball > 150-ball_width && y_ball < 460)) // n6
                            if((x_ball + ball_width == 20) && (y_ball > 460-ball_width && y_ball < 460)) // n7
                                if((x_ball + ball_width == 20) && (y_ball > 50-ball_width && y_ball < 460)) // n8
                                    if((x_ball + ball_width == 20) && (y_ball > 104-ball_width && y_ball < 460)) // n9
                                        if((x_ball + ball_width == 20) && (y_ball > 200-ball_width && y_ball < 460)) // n10
                                            if((x_ball + ball_width == 20) && (y_ball > 280-ball_width && y_ball < 460)) // n11
                                                if((x_ball + ball_width == 20) && (y_ball > 388-ball_width && y_ball < 460)) // n12
                                                    if((x_ball + ball_width == 20) && (y_ball > 491-ball_width && y_ball < 460)) // n13
                                                        if((x_ball + ball_width == 20) && (y_ball > 590-ball_width && y_ball < 460)) // n14
                                                            if((x_ball + ball_width == 20) && (y_ball > 690-ball_width && y_ball < 460)) // n15
                                                                if((x_ball + ball_width == 20) && (y_ball > 790-ball_width && y_ball < 460)) // n16
                                                                    if((x_ball + ball_width == 20) && (y_ball > 890-ball_width && y_ball < 460)) // n17
                                                                        if((x_ball + ball_width == 20) && (y_ball > 990-ball_width && y_ball < 460)) // n18
                                                                            if((x_ball + ball_width == 20) && (y_ball > 1090-ball_width && y_ball < 460)) // n19
                                                                                if((x_ball + ball_width == 20) && (y_ball > 1190-ball_width && y_ball < 460)) // n20
                                                                                    if((x_ball + ball_width == 20) && (y_ball > 1290-ball_width && y_ball < 460)) // n21
                                                                                        if((x_ball + ball_width == 20) && (y_ball > 1390-ball_width && y_ball < 460)) // n22
                                                                                            if((x_ball + ball_width == 20) && (y_ball > 1490-ball_width && y_ball < 460)) // n23
                                                                                                                                                    stop_right = 1;
```
Project Reflections: Successes and Future Work Pt 1

- **Gyro + ACL Interfacing**
  - Interfacing was relatively simple, since the SPI interface code was already written
  - Would have been nice to write SPI interface modules (Or some custom protocol..?)

- **Complimentary Filter**
  - Data scaling is tricky
  - More accurate (expensive) pmod components would be better
  - Could write SPI interface ourselves and use wider sensing range

- **Line Drawing (Bresenham’s)**
  - Might be nice to store screen buffer off-chip (Latency issues)
  - Possibly use larger FPGA so screen buffer does not consumer all BRAM resources
Project Reflections: Successes and Future Work Pt 2

- **2D Maze**
  - Successfully integrating the Maze and the movement of the ball with the Gyro on one board.
  - Would have been better to have implemented the 3D Maze, although it requires more memory.

- **Wireframe Cube**
  - Memory and limits in calculation blocks were a constant ceiling we kept hitting. Need a beefier board.
  - Initial concept was to form a cube. But realized it would have been more interesting to draw insides of a room and used gyro to “look around” the room.
  - Timing errors in calculations were not caught by the program and as a result gave a lot of misleading readings that required many hours of debugging.
Questions?