1. If counting the reserved registers, there are in total 123 registers in the LSM303DLHC module, with 64 registers for Linear acceleration sensor and 59 registers for Magnetic field sensor. If not counting the reserved registers, there are 45 registers in total, with 30 register for Linear acceleration sensor and 15 registers for Magnetic field sensor.
2. Acceleration: x_acceleration at 0xA8 and 0xA9, y_acceleration at 0xAA and 0xAB, z_acceleration at 0xAC and 0xAD. (Acceleration sensor has slave address 0x32)
Magnetic: x_mag at 0x03 and 0x04, y_mag at 0x07 and 0x08, and z_mag at 0x05 and 0x06. (Magnetic sensor has slave address 0x3C)
3. For all of the output data, there are 16 bits per channel, which is why we need to read two bytes for each channel output.
4. I think we can use capacitor such that one surface is fixed while the another surface will move when there are acceleration, so with different acceleration the distance between the surfaces will be different and the capacitance will be different. By measuring the capacitance, we can measure the acceleration. To measure acceleration in different axis, just position the capacitors in different orientations.

CP1 question:
Yes, the sensor data make sense. When we do not move the board, it has 1g reading on the x-axis because of gravity and around 0g reading on other axis. While turn it by 90 degree, the y-axis has 1g reading while other axis readings are 0. When I move it in a direction, there will be higher reading in the corresponding axis and the faster I move it, the higher is the reading.

Python code:
```python
# -*- coding: utf-8 -*-


#%%
# import various libraries necessary to run your Python code
import time   # time related library
import sys,os    # system related library
ok_sdk_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\Python\\x64"
ok_dll_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\lib\\x64"
```

```python
sys.path.append(ok_sdk_loc)   # add the path of the OK library
os.add_dll_directory(ok_dll_loc)
```

```python
import ok     # OpalKelly library
#%%
def write_to_device(slave_addr, reg_addr, value):
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()
    dev.SetWireInValue(0x01, slave_addr)
    dev.SetWireInValue(0x02, reg_addr)
    dev.SetWireInValue(0x03, value)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.5)
    dev.SetWireInValue(0x00, 1) # Write trigger
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.5)
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()  # Update the WireIns

#%%
def read_from_device(slave_addr, reg_addr):
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.2)
    dev.SetWireInValue(0x01, slave_addr)
    dev.SetWireInValue(0x02, reg_addr)
    dev.SetWireInValue(0x00, 2)  # Read trigger
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.2)
    dev.UpdateWireOuts()
    read = dev.GetWireOutValue(0x20)
    if slave_addr == 0x3C:
        m_L = read // 2**8
        m_H = read - (m_L * 2**8)
        read =  m_H * 2**8 + m_L
    if read >= 2**15:
        read = read - 2**16 # deal with 2's complement
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()
    return read
#%%
# Define FrontPanel device variable, open USB communication and
# load the bit file in the FPGA
dev = ok.okCFrontPanel()  # define a device for FrontPanel communication
SerialStatus=dev.OpenBySerial("")      # open USB communication with the OK board
# We will NOT load the bit file because it will be loaded using JTAG interface from Vivado
```

```
# Check if FrontPanel is initialized correctly and if the bit file is loaded.
# Otherwise terminate the program
print("----------------------------------------------------")
if SerialStatus == 0:
    print ("FrontPanel host interface was successfully initialized.")
else:
    print ("FrontPanel host interface not detected. The error code number is:" + str(int(SerialStatus)))
    print("Exiting the program.")
    sys.exit ()
```

```
#%% Reg and value constants
ctrl_reg_1_addr = 0x20
ctrl_reg_1_value = 0x37
mr_reg_m_addr = 0x02
mr_reg_m_value = 0x00
accel_slave_addr = 0x32
magnet_slave_addr = 0x3C
x_a_reg_addr = 0xA8
y_a_reg_addr = 0xAA
z_a_reg_addr = 0xAC
x_m_reg_addr = 0x03
y_m_reg_addr = 0x07
z_m_reg_addr = 0x05
#%%
# Define the two variables that will send data to the FPGA
# We will use WireIn instructions to send data to the FPGA
write_to_device(accel_slave_addr, ctrl_reg_1_addr, ctrl_reg_1_value)  # Enable output
write_to_device(magnet_slave_addr, mr_reg_m_addr, mr_reg_m_value)  # Continuous-conversion mode
while True:
```

```
    print("Send GO signal to the FSM")
    x_a_read = read_from_device(accel_slave_addr, x_a_reg_addr)
    print("x-acceleration read is " + str(x_a_read / 16000) + " g")
    #input()
    y_a_read = read_from_device(accel_slave_addr, y_a_reg_addr)
    print("y-acceleration read is " + str(y_a_read / 16000) + " g")
    #input()
    z_a_read = read_from_device(accel_slave_addr, z_a_reg_addr)
    print("z-acceleration read is " + str(z_a_read / 16000) + " g")
    #input()
    x_m_read = read_from_device(magnet_slave_addr, x_m_reg_addr)
    print("x-magnetic read is " + str(x_m_read))
    #input()
    y_m_read = read_from_device(magnet_slave_addr, y_m_reg_addr)
    print("y-magnetic read is " + str(y_m_read))
    #input()
    z_m_read = read_from_device(magnet_slave_addr, z_m_reg_addr)
    print("z-magnetic read is " + str(z_m_read))
    #input()
```

```
dev.Close
```

[redacted]

Verilog code:
main.v:

```
`timescale 1ns / 1ps

module Main(
	output [7:0] led,
	input sys_clkn,
	input sys_clkp,
	output ADT7420_A0,
	output ADT7420_A1,
	output I2C_SCL_1,
	inout I2C_SDA_1,
	input	[4:0] okUH,
	output [2:0] okHU,
	inout	[31:0] okUHU,
	inout	okAA
);




	//                                                                    Clock
generation/////////////////////////////////////////////////////////////////
	reg ILA_Clk;
	wire clk;
	reg [23:0] ClkDivILA = 24'd0;
	IBUFGDS osc_clk(
		.O(clk),
		.I(sys_clkp),
		.IB(sys_clkn)
	);
	always @(posedge clk) begin
		if (ClkDivILA == 10) begin
			ILA_Clk <= !ILA_Clk;
			ClkDivILA <= 0;
		end else begin
			ClkDivILA <= ClkDivILA + 1'b1;
		end
	end
	//                         Clock                         generation;
/////////////////////////////////////////////////////////////////
```

```
	//PC
communication/////////////////////////////////////////////////////////////////
	// TODO verify OK communication function
	wire [31:0]		PC_rx;
	wire [31:0]		PC_tx;
	wire [31:0]		PC_slave_addr;
	wire [31:0]		PC_addr;
	wire [31:0]		PC_val;
	wire [112:0]		okHE;
	wire [64:0]		okEH;
	localparam	endPt_count = 2;
	wire [endPt_count*65-1:0] okEHx;
	okWireOR # (.N(endPt_count)) wireOR (okEH, okEHx);

	okHost hostIF (
		.okUH(okUH),
		.okHU(okHU),
		.okUHU(okUHU),
		.okClk(okClk),
		.okAA(okAA),
		.okHE(okHE),
		.okEH(okEH)
	);
	okWireIn wire10 (	.okHE(okHE),
				.ep_addr(8'h00),
				.ep_dataout(PC_rx));
	okWireIn wire11 (	.okHE(okHE),
				.ep_addr(8'h01),
				.ep_dataout(PC_slave_addr));
	okWireIn wire12 (	.okHE(okHE),
				.ep_addr(8'h02),
				.ep_dataout(PC_addr));
	okWireIn wire13 (	.okHE(okHE),
				.ep_addr(8'h03),
				.ep_dataout(PC_val));
	okWireOut wire20 (	.okHE(okHE),
				.okEH(okEHx[ 0*65 +: 65 ]),
				.ep_addr(8'h20),
				.ep_datain(PC_tx));

	//								PC
communication////////////////////////////////////////////////////////////////
```

```
	//I2C
SERDES/////////////////////////////////////////////////////////////////////
	wire SCL, SDA,ACK;
	wire [5:0] State;
	wire [7:0] tx_byte,rx_byte;
	wire [1:0] next_step;
	wire ready;
	wire busy;
	I2C_driver I2C_SERDES (
		.busy(busy),

		.led(led),
		.clk(clk),
		.ADT7420_A0(ADT7420_A0),
		.ADT7420_A1(ADT7420_A1),
		.I2C_SCL_0(I2C_SCL_1),
		.I2C_SDA_0(I2C_SDA_1),

		.ACK(ACK),
		.SCL(SCL),
		.SDA(SDA),
		.State(State),

		.tx_byte(tx_byte),
		.rx_byte(rx_byte),
		.next_step(next_step),
		.ready(ready)
		);
	//                                  I2C                                  SERDES
//////////////////////////////////////////////////////////////////////
	wire [9:0] cur_state;
	wire [31:0] PC_rx_reg1;
	wire [31:0] PC_rx_reg2;
	//Sensor
Controller/////////////////////////////////////////////////////////////////
	TS_controller TS_controller(
		.clk(clk),
		.PC_rx(PC_rx),
		.PC_tx(PC_tx),
		.PC_slave_addr(PC_slave_addr),
		.PC_addr(PC_addr),
		.PC_val(PC_val),
		.next_step(next_step),
		.tx_byte(tx_byte),
		.rx_byte(rx_byte),
		.cur_state(cur_state),
		.PC_rx_reg1(PC_rx_reg1),
		.PC_rx_reg2(PC_rx_reg2),
		.ready(ready)
	);
	//Sensor
Controller//////////////////////////////////////////////////////////////////////

	//Instantiate the ILA module
	ila_0 ila_sample12 (
		.clk(clk),
		.probe0({State, SDA, SCL, busy}),
		.probe1(PC_rx_reg1),
		.probe2(PC_rx_reg2),
		.probe3(cur_state));
endmodule
```

TS_controller.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2024/09/22 14:34:42
// Design Name:
// Module Name: TS_controller
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module TS_controller(
	input clk,
        input wire [31:0] PC_rx,
        input wire [31:0] PC_slave_addr,
        input wire [31:0] PC_addr,
        input wire [31:0] PC_val,
        output reg [31:0] PC_tx,
        output reg [1:0] next_step,
        output reg [7:0] tx_byte,
        output reg [9:0] cur_state,
        output reg [7:0] PC_rx_reg1,
        output reg [7:0] PC_rx_reg2,
        input wire [7:0] rx_byte,
        input wire ready
        );

        reg ready_reg;
        reg [7:0] tx_byte_reg;
        reg [7:0] rx_byte_reg;
//        reg [9:0] cur_state;
//        reg [31:0] PC_rx_reg;
        reg [31:0] PC_tx_reg;
        reg tx_flag;
        reg byte2_flag;

        localparam idle_        = 9'b000000001;
        localparam start_wr     = 9'b000000010;
        localparam tx_wr        = 9'b000000100;
        localparam end_wr       = 9'b000001000;
        localparam start_rt     = 9'b000010000;
        localparam tx_rt        = 9'b000100000;
        localparam rstart_rt = 9'b001000000;
        localparam rx_rt        = 9'b010000000;
        localparam end_rt       = 9'b100000000;

        localparam ns_start     = 2'b01;
        localparam ns_tx        = 2'b10;
        localparam ns_rx        = 2'b11;
        localparam ns_end       = 2'b00;

        initial begin
                cur_state <= idle_;
                next_step <= ns_end;
                PC_rx_reg1 <= 0;
                PC_rx_reg2 <= 0;
                PC_tx_reg <= 0;
                tx_byte_reg <= 0;
                rx_byte_reg <= 0;
                tx_flag <= 1'b0;
                byte2_flag    <= 1'b0;
                ready_reg <= 1'b1;
        end

        integer i;
        always @(posedge clk) begin
                for (i=0; i<8; i=i+1) begin
                        tx_byte[i] <= tx_byte_reg[7-i];
                        rx_byte_reg[i] <= rx_byte[7-i];
                end
        end


        always @(posedge clk) begin
                case (cur_state)
                        idle_ : begin
                                PC_rx_reg1 <= PC_rx;
                                PC_rx_reg2 <= PC_rx_reg1;
                                if (PC_rx_reg2[0] == 1'b0 && PC_rx_reg1[0] == 1'b1) begin
                                        cur_state <= start_wr;
                                end
                                if (PC_rx_reg2[1] == 1'b0 && PC_rx_reg1[1] == 1'b1) begin
                                        cur_state <= start_rt;
                                end
                        end
                        //Write single byte
                        start_wr: begin
                                ready_reg <= ready;
                                tx_byte_reg <= PC_slave_addr[7:0];
                                next_step <= ns_start;
                                if (ready_reg == 1'b0 && ready == 1'b1) begin
                                        cur_state <= tx_wr;
                                end
                        end
                        tx_wr: begin
                                case (tx_flag)
                                        1'b0: begin
                                                ready_reg <= ready;
                                                tx_byte_reg <= PC_addr[7:0];
                                                next_step <= ns_tx;
                                                if(ready_reg == 1'b0 && ready == 1'b1) begin
                        tx_flag <= 1'b1;
                    end
                end
                1'b1: begin
                    ready_reg <= ready;
                    tx_byte_reg <= PC_val[7:0];
                    next_step <= ns_tx;
                    if(ready_reg == 1'b0 && ready == 1'b1) begin
                        cur_state <= end_wr;
                        tx_flag <= 1'b0;
                    end
                end
            endcase
        end
        end_wr : begin
            tx_byte_reg <= {8{1'b0}};
            next_step <= ns_end;
            cur_state <= idle_;
        end
        //Read two byte
        start_rt: begin
            ready_reg <= ready;
            tx_byte_reg <= PC_slave_addr[7:0];
            next_step <= ns_start;
            if (ready_reg == 1'b0 && ready == 1'b1) begin
                cur_state <= tx_rt;
            end
        end
        tx_rt: begin
            ready_reg <= ready;
            tx_byte_reg <= PC_addr[7:0];
            next_step <= ns_tx;
            if(ready_reg == 1'b0 && ready == 1'b1) begin
                cur_state <= rstart_rt;
            end
        end
        rstart_rt : begin
            ready_reg <= ready;
            tx_byte_reg <= (PC_slave_addr[7:0] + 1);
            next_step <= ns_start;
            if (ready_reg == 1'b0 && ready == 1'b1) begin
                cur_state <= rx_rt;
            end
        end
```

```
            rx_rt : begin
                ready_reg <= ready;
                tx_byte_reg <= {8{byte2_flag}};
                next_step <= ns_rx;
                if (ready_reg == 1'b0 && ready == 1'b1) begin
                    case(byte2_flag)
                        1'b0: begin
                            byte2_flag <= 1'b1;
                            for (i=0; i<8; i =i+1) begin
                                PC_tx_reg[7-i] <= rx_byte_reg[7-i];
                            end
                        end
                        1'b1: begin
                            byte2_flag <= 1'b0;
                            for (i=0; i<8; i = i+1) begin
                                PC_tx_reg[15-i] <= rx_byte_reg[7-i];
                            end
                            cur_state <= end_rt;
                        end
                    endcase
                end
            end
            end_rt : begin
                tx_byte_reg <= {8{1'b0}};
                next_step <= ns_end;
                cur_state <= idle_;
                PC_tx <= PC_tx_reg;
                PC_tx_reg <= 0;
            end
            default : begin
                tx_byte_reg <= {8{1'b0}};
                next_step <= ns_end;
                cur_state <= idle_;
            end
        endcase
    end


endmodule
```

I2C.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
```

```
// Engineer:
//
// Create Date: 2024/09/22 02:36:21
// Design Name:
// Module Name: I2C
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module I2C_driver(
	output [7:0] led,
	input    clk,
	output ADT7420_A0,
	output ADT7420_A1,
	output I2C_SCL_0,
	inout    I2C_SDA_0,

	output reg ACK,
	output reg SCL,
	output reg SDA,
	output reg busy,

	output reg [5:0] State,
	input    wire [7:0] tx_byte,
	output reg [7:0] rx_byte,
	input    wire [1:0] next_step,
	output reg ready
	);

	localparam idle_      = 6'b000000;
	localparam start_    = 6'b000001;
	localparam tx           = 6'b000010;
	localparam tx_ack    = 6'b000100;
	localparam rx           = 6'b001000;
localparam rx_ack    = 6'b010000;
localparam end_       = 6'b100000;
localparam error_    = 6'b111111;

reg [2:0] bit_counter;
reg [9:0] clk_counter;

reg [7:0] rx_byte_reg;
reg [7:0] tx_byte_reg;

reg error;

assign led[7] = ACK;
assign led[6] = SCL;
assign led[5] = SDA;
assign led[4:0] = {5{error}};
assign I2C_SCL_0 = SCL;
assign I2C_SDA_0 = SDA;
assign ADT7420_A0 = 1'b0;
assign ADT7420_A1 = 1'b0;

initial begin
	SCL = 1'b1;
	SDA = 1'b1;
	ACK = 1'b1;
	error = 1'b0;
	ready = 1'b1;
	State = idle_;
	rx_byte = 8'b00000000;
	rx_byte_reg = 0;
	tx_byte_reg = 0;
end

always @(posedge clk) begin
	case (State)
		idle_ : begin
			busy <= 1'b0;
			if (next_step == 2'b01)begin
				busy <= 1'b1;
				State <= start_;
				clk_counter <= 10'd400;
				bit_counter <= 0;
			end
		end
```

```
start_: begin
        case (clk_counter)
                10'd0      : begin
                        SCL <= 1'b0;
                        SDA <= 1'bz;
                        clk_counter <= clk_counter + 1;
                end
                10'd400 : begin
                        SCL <= 1'b1;
                        clk_counter <= clk_counter + 1;
                end
                10'd600 : begin
                        SCL <= 1'b1;
                        SDA <= 1'b0;
                        clk_counter <= clk_counter + 1;
                end
                10'd799 : begin
                        State <= tx;
                        tx_byte_reg <= tx_byte;
                        clk_counter <= 10'd0;
                end
                default : begin
                        clk_counter <= clk_counter + 1;
                end
        endcase
end

tx: begin
        case (clk_counter)
                10'd0 : begin
                        SCL <= 1'b0;
                        clk_counter <= clk_counter + 1;
                end
                10'd200 : begin
                        SDA <= tx_byte_reg[bit_counter];
                        SCL <= 1'b0;
                        clk_counter <= clk_counter + 1;
                end
                10'd400 : begin
                        SCL <= 1'b1;
                        clk_counter <= clk_counter + 1;
                end
                10'd799 : begin
                        if (bit_counter == 3'd7) begin
                rx_byte <= rx_byte_reg;
                State <= tx_ack;
                bit_counter <= 3'd0;
                end
            else begin
                bit_counter <= bit_counter + 1;
            end
            clk_counter <= 10'd0;
        end
        default : begin
            clk_counter <= clk_counter + 1;
        end
    endcase
end

tx_ack : begin
    case (clk_counter)
        10'd0 : begin
            SCL <= 1'b0;
            SDA <= 1'bz;
            clk_counter <= clk_counter + 1;
        end
        10'd400 : begin
            SCL <= 1'b1;
            ACK <= SDA;
            clk_counter <= clk_counter + 1;
        end
        10'd799 : begin
            tx_byte_reg <= tx_byte;
            clk_counter <= 10'd0;
            case (next_step)
                2'b00: begin
                    State <= end_;
                end
                2'b01: begin
                    State <= start_;
                end
                2'b10: begin
                    State <= tx;
                end
                2'b11: begin
                    State <= rx;
                end
            endcase
```

```
		end
		default : begin
			clk_counter <= clk_counter + 1;
		end
	endcase
end

rx: begin
	case (clk_counter)
		10'd0 : begin
			SCL <= 1'b0;
			SDA <= 1'bz;
			clk_counter <= clk_counter + 1;
		end
		10'd400 : begin
			SCL <= 1'b1;
			clk_counter <= clk_counter + 1;
		end
		10'd500 : begin
			rx_byte_reg[bit_counter] <= SDA;
			clk_counter <= clk_counter + 1;
		end
		10'd799 : begin
			if (bit_counter == 3'd7) begin
				rx_byte <= rx_byte_reg;
				State <= rx_ack;
				bit_counter <= 3'd0;
			end else begin
				bit_counter <= bit_counter + 1;
			end
			clk_counter <= 10'd0;
		end
		default : begin
			clk_counter <= clk_counter + 1;
		end
	endcase
end

rx_ack : begin
	case (clk_counter)
		10'd0 : begin
			SCL <= 1'b0;
			clk_counter <= clk_counter + 1;
		end
```

```
                10'd200: begin
                    SDA <= tx_byte_reg[0];
                    clk_counter <= clk_counter + 1;
                end
                10'd400 : begin
                    SCL <= 1'b1;
                    clk_counter <= clk_counter + 1;
                end
                10'd799 : begin
                    clk_counter <= 10'd0;
                    tx_byte_reg <= tx_byte;
                    case (next_step)
                            2'b00: begin
                                State <= end_;
                            end
                            2'b01: begin
                                State <= start_;
                            end
                            2'b10: begin
                                State <= tx;
                            end
                            2'b11: begin
                                State <= rx;
                            end
                    endcase
                end
                default : begin
                    clk_counter <= clk_counter + 1;
                end
            endcase
        end

        end_ : begin
            case (clk_counter)
                10'd0: begin
                    SCL <= 1'b0;
                    SDA <= 1'b0;
                    clk_counter <= clk_counter + 1;
                end
                10'd400: begin
                    SCL <= 1'b1;
                    SDA <= 1'b0;
                    clk_counter <= clk_counter + 1;
                end
```

```
                    10'd600 : begin
                        SCL <= 1'b1;
                        SDA <= 1'b1;
                        clk_counter <= 10'd0;
                        State <= idle_;
                    end
                    default : begin
                        clk_counter <= clk_counter + 1;
                    end
                endcase
            end
        default : begin
            error <= 1'b1;
        end
      endcase
    end

    always @(posedge clk) begin
        case (State)
            tx : ready <= 1'b0;
            rx : ready <= 1'b0;
            default : ready <=1'b1;
         endcase
    end


endmodule
```