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)

1. 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.
2. 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:

# -\*- 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"

sys.path.append(ok\_sdk\_loc)   # add the path of the OK library

os.add\_dll\_directory(ok\_dll\_loc)

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

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