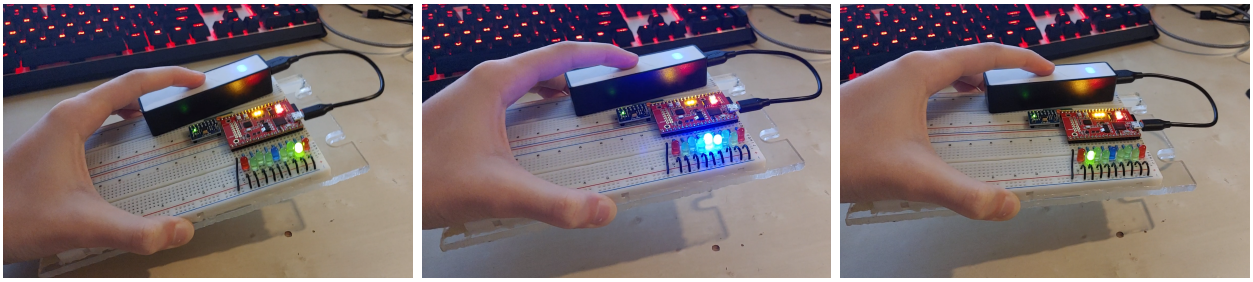
Digital Bubble Level

Vladislav Pomogaev - 26951160

September 27, 2021

1 Introduction

This device forms a digital bubble level; also called a spirit level. It can be helpful in levelling things horizontally.



(a) (b) (c)

Figure 1: Photo stills of the device in action. When you tilt the device to the left as in a), the LEDs to the right light up. As you tilt the device more to the right, the lit LED moves from right to left as in b) and c). When level, the blue center LED lights up.

This project consists of a synthesised Verilog design running on an Efinix Xyloni FPGA development board. An accelerometer (MPU-6050) and 9 LEDs are connected to this board.

The Verilog design consists of an FSM (Finite State Machine), and an I^2C controller IP block from Efinix. The FSM can send commands to the accelerometer through the I^2C controller, which acts as the master. Upon reset, the FSM wakes the accelerometer by writing to a register. Then, in a loop, the FSM requests the acceleration measured in one axis from the accelerometer, and converts the acceleration to a grey-code-like signal which is then output to the LEDs. All of this is done while implementing the communication protocol of the I^2C controller. The FSM must request data in the correct format, provide slave address, data, command byte, and number of data bits, wait for response, and check for errors during every read and write command.

2 Overview

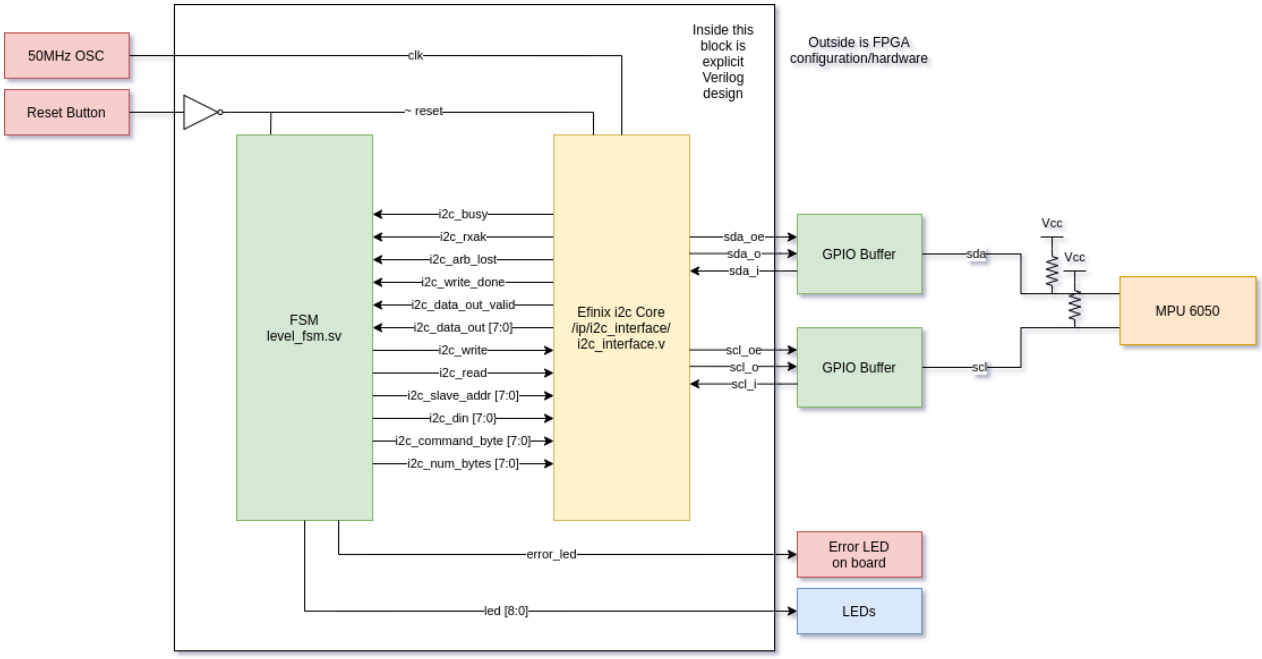


Figure 2: Block diagram of the FSM and how it forms the bubble-level system. The level FSM talks to the I2C core, which communicates to the MPU via a set of buffered GPIO blocks on the FPGA. A reset button resets all FSMs to their initial state. The design runs is compiled for a 50MHz clock constraint. The output to outside of the FPGA is buffered through the GPIO logic blocks, which was the reason I used IP specific to the FPGA.

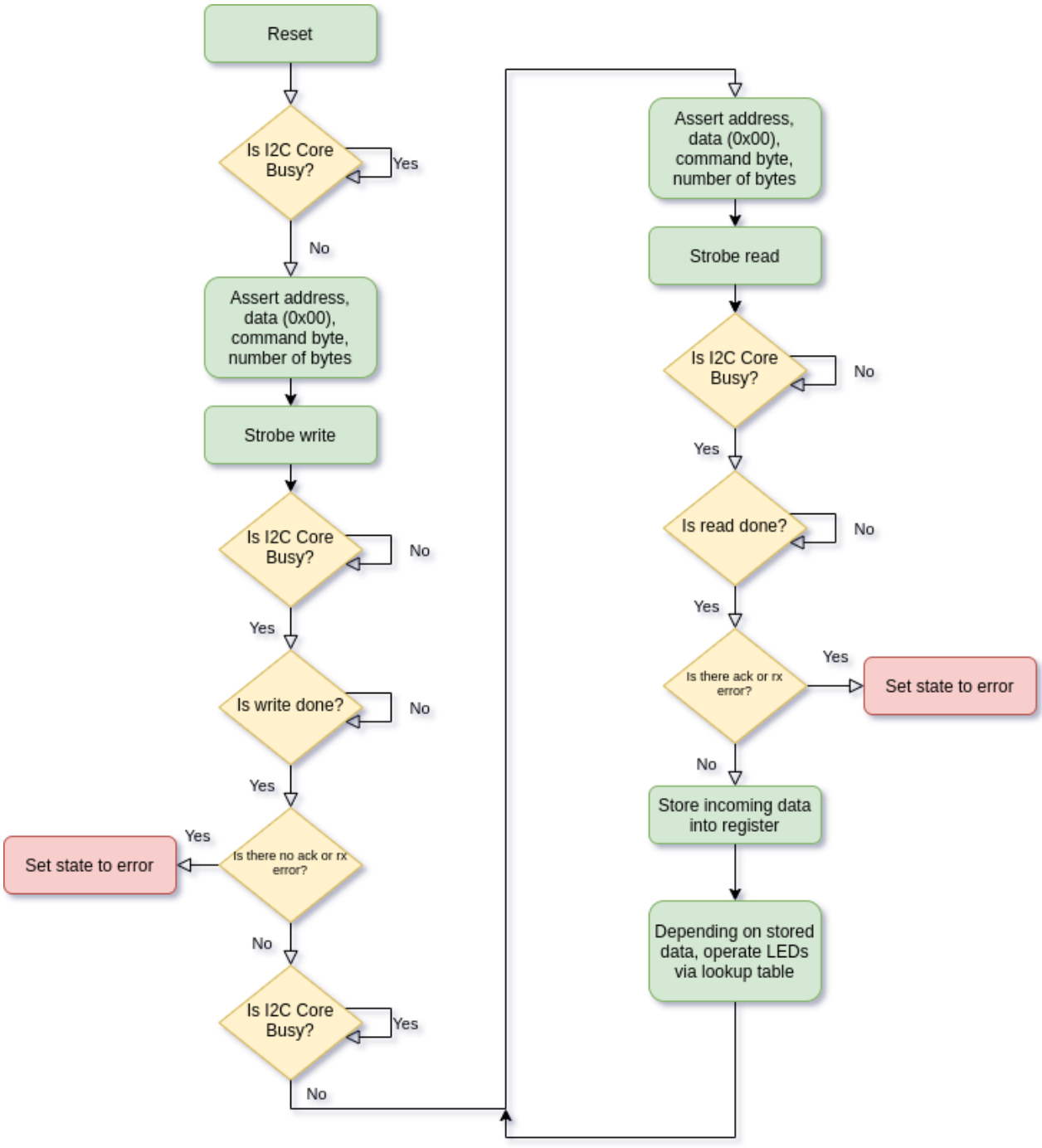


Figure 3: Flow diagram of the internal functioning of the FSM. The assertion of the addresses, data, command byte is done even though it is slightly redundant in this situation to allow for easy expansion of functionality by changing those lines to be tri-state and allowing other modules to write to them.

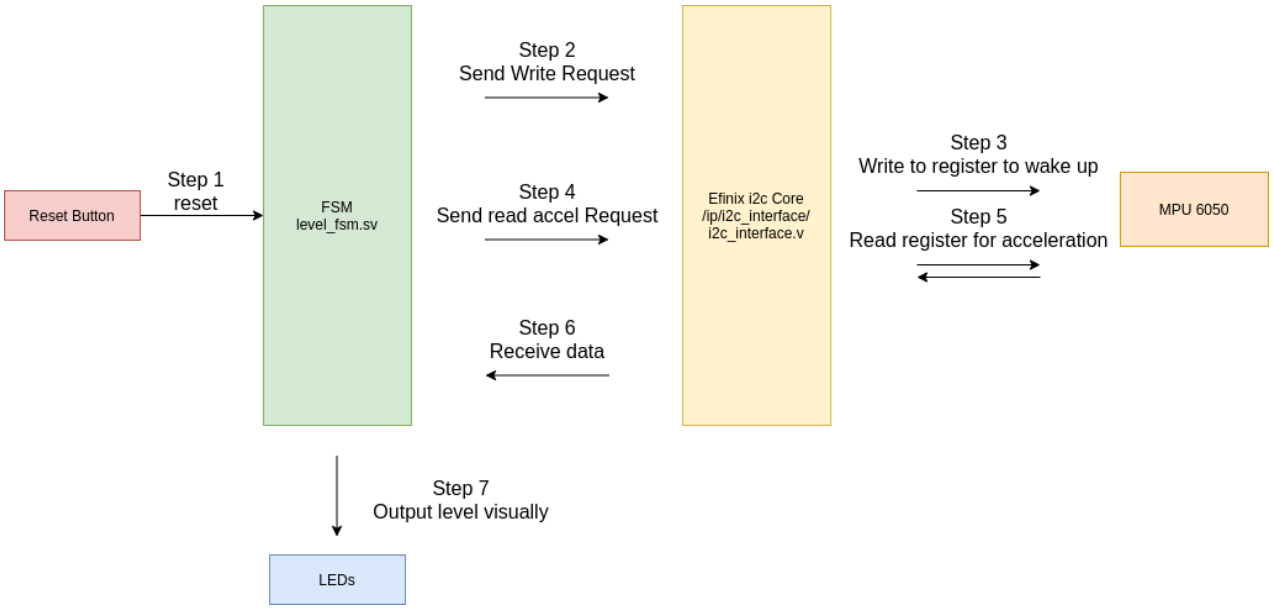


Figure 4: Data flow diagram for the FSM.

2.1 States

The states for the FSM and their purposes are as follows:

```
typedef enum logic [6:0] {
  ENSURE_BUSY_1 =       7'b0000_000, // wait for I2C controller to not be busy
  ASSIGN_WRITE_2 =      7'b0001_000, // tell I2C controller which registers to write what to
  ASSERT_WRITE_3 =      7'b0010_001, // tell I2C controller to write
  WAIT_FOR_BUSY_4 =     7'b0011_000, // wait for controller to be busy
  WAIT_FOR_DONE_5 =     7'b0100_000, // wait for controller to finish
  VERIFY_6 =            7'b0101_000, // verify that there were no errors writing register
  ENSURE_BUSY_7 =       7'b0110_000, // verify that the controller is not busy
  ASSIGN_WRITE_8 =      7'b0111_000, // tell I2C controller which registers to read
  ASSERT_READ_9 =       7'b1000_010, // tell I2C controller to read register
  WAIT_FOR_BUSY_10 =    7'b1001_000, // wait for controller to be busy
  WAIT_FOR_VALID_11 =   7'b1010_000, // wait for controller to finish read
  VERIFY_12 =           7'b1011_000, // verify that no errors occured, register info
  LED_OPERATION_13 =    7'b1100_000, // output LED pattern
  ERROR =               7'b1101_100
} state_e;
```

3 Testing

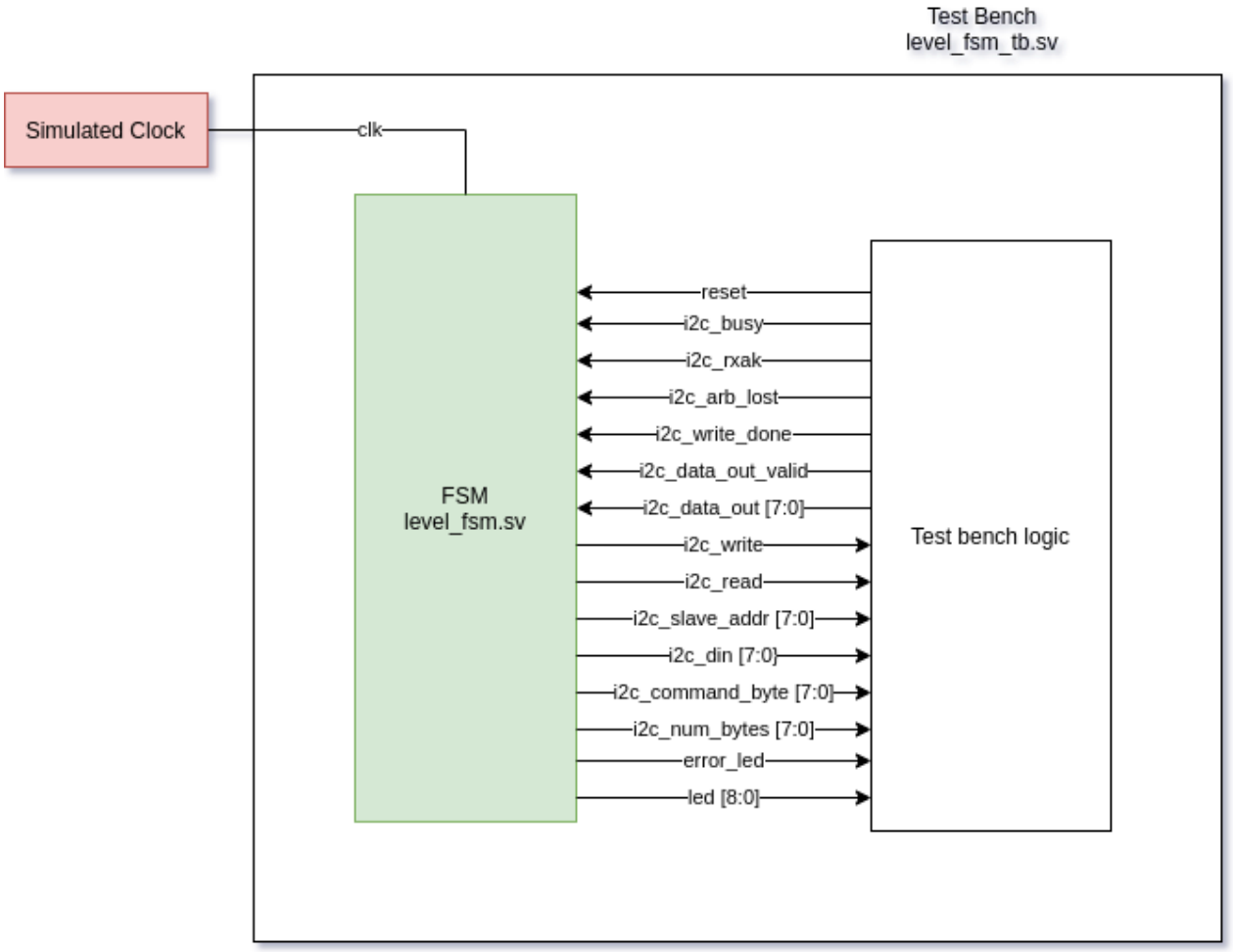


Figure 5: Block diagram of how the level FSM (not the Efinix i2c core) is connected to the test bench. The test bench is a simple piece of logic that provides timed inputs to the level FSM and asserts that the outputs of the FSM are what are expected. When a new test is run by the test bench logic, the instantiated DUT is reset. Asserts are used to check correct functionality.

The I2C core provided by Efinix was autogenerated and came with it's own test-bench. Since I did not write this IP or the test bench, I do not include the source code for this report nor diagrams for it. I'm taking it at face-value and trusting that it works.

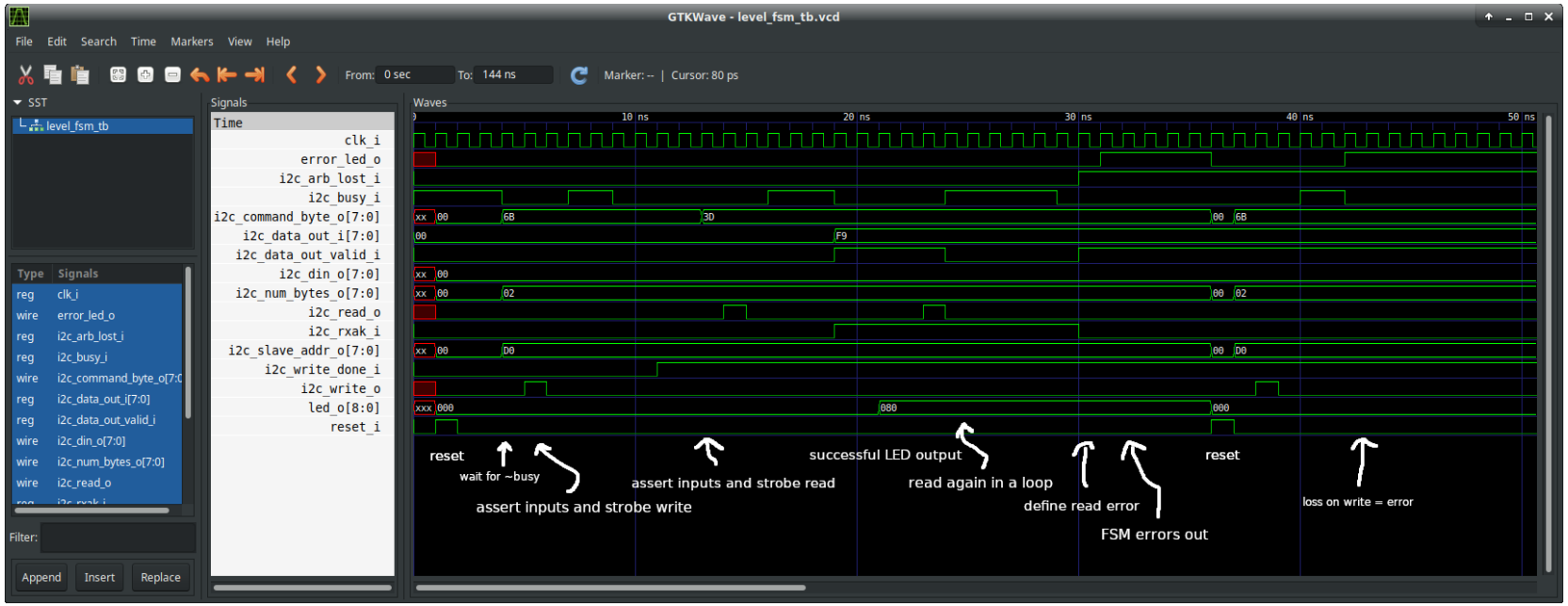


Figure 6: Waveform diagram from FSM test. The FSM was tested under three scenarios: perfectly functioning I2C controller and MPU, an error on the read command, and an error on the write command. All delays were verified to work (waiting for busy, write and read finish).

4 Code (also included in separate file)

level_fsm.sv

```
/*
Description:
  An FSM which implements the functionality of a "bubble level" or "spirit level".
  Reads single-axis accelection data from the MPU-6050 by
  interfacing with an I2C Core IP from Efinix. Wakes accelerometer up first by writing to register,
  then reads register in a loop.
  Updates the status of the 9 LED's based on the acceleration measured. Contains error LED as well.

  LED Encoding table looks like so:

  Tilt      Accel.  MSB  LSB    MSB  MSB          Led Led Led Led Led Led Led Led Led   Decimal
  Degrees   (g)     Sign              Bins         8   7   6   5   4   3   2   1   0     LED Encoding
  -8        -0.139  1    -2280  -9   1111111000   1   0   0   0   0   0   0   0   0     256
  -7        -0.122  1    -1997  -8   1111111001   1   1   0   0   0   0   0   0   0     384
  -6        -0.105  1    -1713  -7   1111111010   0   1   0   0   0   0   0   0   0     128
  -5        -0.087  1    -1428  -6   1111111011   0   1   1   0   0   0   0   0   0     192
  -4        -0.070  1    -1143  -4   1111111100   0   0   1   0   0   0   0   0   0      64
  -3        -0.052  1     -857  -3   1111111101   0   0   1   1   0   0   0   0   0      96
  -2        -0.035  1     -572  -2   1111111110   0   0   0   1   0   0   0   0   0      32
  -1        -0.017  1     -286  -1   1111111111   0   0   0   1   1   0   0   0   0      48
   0         0.000  0        0   0   0            0   0   0   0   1   0   0   0   0      16
   1         0.017  0      286   1   1            0   0   0   0   1   1   0   0   0      24
   2         0.035  0      572   2   10           0   0   0   0   0   1   0   0   0       8
   3         0.052  0      857   3   11           0   0   0   0   0   1   1   0   0      12
   4         0.070  0     1143   4   100          0   0   0   0   0   0   1   0   0       4
   5         0.087  0     1428   6   101          0   0   0   0   0   0   1   1   0       6
   6         0.105  0     1713   7   110          0   0   0   0   0   0   0   1   0       2
   7         0.122  0     1997   8   111          0   0   0   0   0   0   0   1   1       3
   8         0.139  0     2280   9   1000         0   0   0   0   0   0   0   0   1       1

  The accelerometer stores each axis measurement as 16 bit. Registers are 8 bit wide.
  We only access the MSB because we don't need the resolution.
  See page 29 of https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

Inputs:
  clk_i                   - Clock
  reset_i                 - Active high reset
  i2c_busy_i              - Logic high indicates that the I2C bus is busy
  i2c_rxak_i              - Logic low indicates that the I2C slave devicereceived and
                            acknowledged the I2C transfer
  i2c_arb_lost_i          - Logic high indicates that there is arbitration lost in the I2C transfer
  i2c_write_done_i        - Logic high indicates that I2C master write data is
                            sent and ready to accept by I2C slave device
  i2c_data_out_valid_i    - Logic high indicates that I2C master read data is valid and ready to read
  i2c_data_out_i          - Read data output from the I2C core


Outputs:
  i2c_write_o             - Write to slave strobe high
  i2c_read_o              - Read from slave strobe high
  i2c_slave_addr_o        - Address of slave in 8bit format. Add trailing zero to use 7-bit addressing.
  i2c_din_o               - Data read from slave
  i2c_command_byte_o      - Command byte sent to slave. (Register to read from)
  i2c_num_bytes_o         - Number of bytes of data to write or read. Includes the
                            command byte (if sending one byte, i2c_num_bytes_o = 'd2)
  error_led_o             - Active high if an error has occured in state machine; lost bytes,
                            error writing to slave or reading
  led_o                   - 9 bit bus that lights up based on encoding above
*/

module level_fsm (
  input wire          clk_i,
  input wire          reset_i,

  input wire          i2c_busy_i,
  input wire          i2c_rxak_i,
  input wire          i2c_arb_lost_i,
  input wire          i2c_write_done_i,
  input wire          i2c_data_out_valid_i,
```

```
70    input wire  [7:0]   i2c_data_out_i,
71
72    output wire         i2c_write_o,
73    output wire         i2c_read_o,
74    output logic [7:0]  i2c_slave_addr_o,
75    output logic [7:0]  i2c_din_o,
76    output logic [7:0]  i2c_command_byte_o,
77    output logic [7:0]  i2c_num_bytes_o,
78
79    output wire         error_led_o,
80    output logic [8:0]  led_o
81  );
82
83  logic signed [7:0] raw_buffer;
84  localparam slave_address = {7'h68, 1'b0};
85  localparam accel_register = 8'h3D;
86  localparam wake_register = 8'h6b;
87
88  typedef enum logic [6:0] {
89    ENSURE_BUSY_1 =       7'b0000_000, // wait for I2C controller to not be busy
90    ASSIGN_WRITE_2 =      7'b0001_000, // tell I2C controller which registers to write what to
91    ASSERT_WRITE_3 =      7'b0010_001, // tell I2C controller to write
92    WAIT_FOR_BUSY_4 =     7'b0011_000, // wait for controller to be busy
93    WAIT_FOR_DONE_5 =     7'b0100_000, // wait for controller to finish
94    VERIFY_6 =            7'b0101_000, // verify that there were no errors writing register
95    ENSURE_BUSY_7 =       7'b0110_000, // verify that the controller is not busy
96    ASSIGN_WRITE_8 =      7'b0111_000, // tell I2C controller which registers to read
97    ASSERT_READ_9 =       7'b1000_010, // tell I2C controller to read register
98    WAIT_FOR_BUSY_10 =    7'b1001_000, // wait for controller to be busy
99    WAIT_FOR_VALID_11 =   7'b1010_000, // wait for controller to finish read
100   VERIFY_12 =           7'b1011_000, // verify that no errors occured, register info
101   LED_OPERATION_13 =    7'b1100_000, // output LED pattern
102   ERROR =               7'b1101_100
103 } state_e;
104
105 state_e state;
106
107 // Glitch free state outputs based on last bits of current state
108 assign i2c_write_o = state[0];
109 assign i2c_read_o = state[1];
110 assign error_led_o = state[2];
111
112 always @(posedge clk_i) begin
113   if (reset_i) begin
114     state <= ENSURE_BUSY_1;
115     led_o <= 9'b0;
116     raw_buffer <= 8'b0;
117     i2c_din_o <= 8'b0;
118     i2c_command_byte_o <= 8'h0;
119     i2c_slave_addr_o <= 8'b0;
120     i2c_num_bytes_o <= 8'd0;
121   end
122   else begin
123     case(state)
124       ENSURE_BUSY_1: begin
125         if (i2c_busy_i === 1) state <= ENSURE_BUSY_1;
126         else begin
127           state <= ASSIGN_WRITE_2;
128           //assign DIN, command_byte, slave_addr, and num_bytes here
129           //wake accelerometer by writing to wake_register 0x00
130           i2c_din_o <= 8'b0;
131           i2c_command_byte_o <= wake_register;
132           i2c_slave_addr_o <= slave_address;
133           i2c_num_bytes_o <= 8'd2;
134         end
135       end
136       ASSIGN_WRITE_2: state <= ASSERT_WRITE_3; //assign inputs first before flashing write strobe
137       ASSERT_WRITE_3: state <= WAIT_FOR_BUSY_4; //write strobe
138       WAIT_FOR_BUSY_4: begin
139         if (i2c_busy_i === 1) state <= WAIT_FOR_DONE_5; //wait for busy
140         else state <= WAIT_FOR_BUSY_4;
141       end
142       WAIT_FOR_DONE_5: begin //wait for write done
143         if (i2c_write_done_i === 1) state <= VERIFY_6;
```

```
144           else state <= WAIT_FOR_DONE_5;
145         end
146         VERIFY_6: begin
147           if (i2c_arb_lost_i === 0 && i2c_rxak_i === 0) state <= ENSURE_BUSY_7;
148           //check that slave has acknowledged and no bits were lost
149           else state <= ERROR;
150         end
151         ENSURE_BUSY_7: begin // wait for busy to not be high
152           if (i2c_busy_i === 1) state <= ENSURE_BUSY_7;
153           else begin
154             state <= ASSIGN_WRITE_8;
155             //assign command_byte, slave_addr, and num_bytes here
156             //read the MSB of accelerometer axis measurement
157             i2c_command_byte_o <= accel_register;
158             i2c_slave_addr_o <= slave_address;
159             i2c_num_bytes_o <= 8'd2;
160           end
161         end
162         ASSIGN_WRITE_8: state <= ASSERT_READ_9; //wait for writing of data
163         ASSERT_READ_9: state <= WAIT_FOR_BUSY_10; //flash read strobe
164         WAIT_FOR_BUSY_10: begin
165           if (i2c_busy_i === 1) state <= WAIT_FOR_VALID_11; //wait for busy
166           else state <= WAIT_FOR_BUSY_10;
167         end
168         WAIT_FOR_VALID_11: begin
169           if (i2c_data_out_valid_i === 1) state <= VERIFY_12; // wait for data rx
170           else state <= WAIT_FOR_VALID_11;
171         end
172         VERIFY_12: begin
173           if (i2c_arb_lost_i === 0 && i2c_rxak_i === 1) begin //check that no bits lost and no slave ack
174             state <= LED_OPERATION_13;
175             raw_buffer <= i2c_data_out_i; // register the data
176           end
177           else state <= ERROR;
178         end
179         LED_OPERATION_13: begin
180           // enter the calculation that associates the LEDs with their respective tilt levels here
181           if (raw_buffer < 8'sb11111000)
182             led_o <= 9'd256;
183           else if (raw_buffer < 8'sb11111001)
184             led_o <= 9'd384;
185           else if (raw_buffer < 8'sb11111010)
186             led_o <= 9'd128;
187           else if (raw_buffer < 8'sb11111011)
188             led_o <= 9'd192;
189           else if (raw_buffer < 8'sb11111100)
190             led_o <= 9'd64;
191           else if (raw_buffer < 8'sb11111101)
192             led_o <= 9'd96;
193           else if (raw_buffer < 8'sb11111110)
194             led_o <= 9'd32;
195           else if (raw_buffer < 8'sb11111111)
196             led_o <= 9'd48;
197           else if (raw_buffer < 8'sb00000000)
198             led_o <= 9'd16;
199           else if (raw_buffer < 8'sb00000001)
200             led_o <= 9'd24;
201           else if (raw_buffer < 8'sb00000010)
202             led_o <= 9'd8;
203           else if (raw_buffer < 8'sb00000011)
204             led_o <= 9'd12;
205           else if (raw_buffer < 8'sb00000100)
206             led_o <= 9'd4;
207           else if (raw_buffer < 8'sb00000101)
208             led_o <= 9'd6;
209           else if (raw_buffer < 8'sb00000110)
210             led_o <= 9'd2;
211           else if (raw_buffer < 8'sb00000111)
212             led_o <= 9'd3;
213           else if (raw_buffer < 8'sb00001000)
214             led_o <= 9'd1;
215           else
216             led_o <= 9'd1;
217           state <= ENSURE_BUSY_7;
```

```
        end
        ERROR: begin
          state <= ERROR;
        end
        default:  state <= ENSURE_BUSY_1;
      endcase
    end
  end

  endmodule
```

level_fsm_tb.sv

```
  `timescale 1ns/100ps
  /*
  Description:
    An FSM testbench for the bubble level FSM. Emulates the functionality of three different states of I2C
    controller:
      A perfectly working controller
      A controller that errors on the read
      A controller that errors on the write

    Test bench tests:
       All delays
       All rx_arb errors checks
       One write state from the accelerometer to the LEDs


  DUT Inputs:
    clk_i                   - Clock
    reset_i                 - Active high reset
    i2c_busy_i              - Logic high indicates that the I2C bus is busy
    i2c_rxak_i              - Logic low indicates that the I2C slave devicereceived and
                              acknowledged the I2C transfer
    i2c_arb_lost_i          - Logic high indicates that there is arbitration lost in the I2C transfer
    i2c_write_done_i        - Logic high indicates that I2C master write data is
                              sent and ready to accept by I2C slave device
    i2c_data_out_valid_i    - Logic high indicates that I2C master read data is valid and ready to read
    i2c_data_out_i          - Read data output from the I2C core


  DUT Outputs:
    i2c_write_o             - Write to slave strobe high
    i2c_read_o              - Read from slave strobe high
    i2c_slave_addr_o        - Address of slave in 8bit format. Add trailing zero to use 7-bit addressing.
    i2c_din_o               - Data read from slave
    i2c_command_byte_o      - Command byte sent to slave. (Register to read from)
    i2c_num_bytes_o         - Number of bytes of data to write or read. Includes the
                              command byte (if sending one byte, i2c_num_bytes_o = 'd2)
    error_led_o             - Active high if an error has occured in state machine; lost bytes,
                              error writing to slave or reading
    led_o                   - 9 bit bus that lights up based on encoding above
  */


  module level_fsm_tb;

    logic         clk_i = 1;
    logic         reset_i;
    logic         i2c_busy_i;
    logic         i2c_rxak_i;
    logic         i2c_arb_lost_i;
    logic         i2c_write_done_i;
    logic         i2c_data_out_valid_i;
    logic [7:0]   i2c_data_out_i;

    logic         i2c_write_o;
    logic         i2c_read_o;
    logic [7:0]   i2c_slave_addr_o;
    logic [7:0]   i2c_din_o;
    logic [7:0]   i2c_command_byte_o;
    logic [7:0]   i2c_num_bytes_o;

    logic         error_led_o;
```

```
62      logic [8:0]  led_o;
63
64      level_fsm dut (
65        .clk_i,
66        .reset_i,
67        .i2c_busy_i,
68        .i2c_rxak_i,
69        .i2c_arb_lost_i,
70        .i2c_write_done_i,
71        .i2c_data_out_valid_i,
72        .i2c_data_out_i,
73        .i2c_write_o,
74        .i2c_read_o,
75        .i2c_slave_addr_o,
76        .i2c_din_o,
77        .i2c_command_byte_o,
78        .i2c_num_bytes_o,
79        .error_led_o,
80        .led_o
81      );
82
83      initial
84      begin
85        $dumpfile("level_fsm_tb.vcd");
86        $dumpvars(0,clk_i);
87        $dumpvars(1,reset_i);
88        $dumpvars(2,i2c_busy_i);
89        $dumpvars(4,i2c_rxak_i);
90        $dumpvars(5,i2c_arb_lost_i);
91        $dumpvars(6,i2c_write_done_i);
92        $dumpvars(7,i2c_data_out_valid_i);
93        $dumpvars(8,i2c_data_out_i);
94        $dumpvars(9,i2c_write_o);
95        $dumpvars(10,i2c_read_o);
96        $dumpvars(11,i2c_slave_addr_o);
97        $dumpvars(12,i2c_din_o);
98        $dumpvars(13,i2c_command_byte_o);
99        $dumpvars(14,i2c_num_bytes_o);
100       $dumpvars(15,error_led_o);
101       $dumpvars(16,led_o);
102     end
103
104     always #0.5 clk_i = ~clk_i;
105
106     initial begin
107       ////////////////////////////
108       //////   Reset/Begin   //////
109       ////////////////////////////
110       // full reset with busy line
111       reset_i = 1'b0;
112       i2c_busy_i = 1'b1;
113       i2c_rxak_i = 1'b0;
114       i2c_arb_lost_i = 1'b0;
115       i2c_write_done_i = 1'b0;
116       i2c_data_out_valid_i = 1'b0;
117       i2c_data_out_i = 8'b0;
118       #1;
119       reset_i = 1'b1;
120       #1;
121       reset_i = 1'b0;
122       #1;
123       ////////////////////////////
124       ///   Normal operation   ////
125       ////////////////////////////
126       // in this section we test the normal operation of the I2C controller
127       // that no errors are thrown, read, write, delays work
128       // correct LED pattern shows up
129       // and that we loop reading
130
131       // make sure writing has not started
132       assert(i2c_din_o == 8'b0);
133       assert(i2c_command_byte_o == 8'b0);
134       assert(i2c_slave_addr_o == 8'b0);
135       assert(i2c_num_bytes_o == 8'b0);
```

```
136       #1;
137       i2c_busy_i = 1'b0; // now I2C is free
138       #1;
139       assert(i2c_din_o == 8'b0); // assert that correct data is going to be written
140       assert(i2c_command_byte_o == 8'h6b);
141       assert(i2c_slave_addr_o == {7'h68, 1'b0});
142       assert(i2c_num_bytes_o == 8'd2);
143       #1;
144       assert(i2c_write_o == 1'b1); // assert write stobe
145       #1;
146       i2c_busy_i = 1'b1; // assert that we are waiting for chip to be not busy
147       assert(i2c_write_o == 1'b0);
148       assert(error_led_o == 1'b0);
149       #1;
150       i2c_busy_i = 1'b1; // assert that we are waiting for chip to be not busy
151       assert(i2c_write_o == 1'b0);
152       assert(error_led_o == 1'b0);
153       #1;
154       i2c_busy_i = 1'b0; // make sure we have not errored out yet because we have an ack and no errors
155       assert(error_led_o == 1'b0);
156       #1;
157       assert(error_led_o == 1'b0);
158       #1;
159       i2c_write_done_i = 1'b1;
160       #1; //wait for write to ack
161       #1; //verify arb_lost and rxak low
162       #1; //ensure busy is low
163       assert(i2c_command_byte_o == 8'h3D);
164       assert(i2c_slave_addr_o == {7'h68, 1'b0});
165       assert(i2c_num_bytes_o == 8'd2);
166       #1;
167       assert(i2c_read_o == 1'd1);
168       #1;
169       i2c_busy_i = 1'b1;
170       #1;
171       #1;
172       #1;
173       i2c_busy_i = 1'b0;
174       i2c_data_out_valid_i = 1'b1;
175       i2c_data_out_i = 8'b11111001;
176       i2c_arb_lost_i = 1'b0;
177       i2c_rxak_i = 1'b1;
178       //wait for valid data and busy low
179       #1; //wait for busy to go low
180       #1; //verify data out is valid
181       #1; //verify no lost and rx did not ack, register the data
182       assert(led_o == 9'd128); // assert that led output is correct
183       #1; //finished test under standard conditions. since busy is low, we assert the lines again
184
185       ////////////////////////////
186       ////   Error on read     ////
187       ////////////////////////////
188       // in this section we confirm that upon a read error the FSM goes into error state
189
190       assert(i2c_command_byte_o == 8'h3D);
191       assert(i2c_slave_addr_o == {7'h68, 1'b0});
192       assert(i2c_num_bytes_o == 8'd2);
193       #1;
194       assert(i2c_read_o == 1'd1); // we strobe read again
195       i2c_busy_i = 1'b1;
196       i2c_data_out_valid_i = 1'b0;
197       #1;
198       #1;// we are now waiting for busy
199       #1;
200       #1;
201       #1;
202       i2c_busy_i = 1'b0;
203       #1;
204       i2c_data_out_valid_i = 1'b1;
205       i2c_arb_lost_i = 1'b1;
206       i2c_rxak_i = 1'b0;
207       #1; // verify data out valid
208       #1; // verify errors; we should get out because we lost bits
209       assert(error_led_o == 1'd1); // we should get an error
```

```
210       #1;
211       #1;
212       #1;
213       #1;
214       reset_i = 1'b1;
215       #1;
216       reset_i = 1'b0;
217
218       ////////////////////////////
219       ////   Error on write    ////
220       ////////////////////////////
221       // in this section we confirm that upon a write error the FSM goes into error state
222       assert(i2c_din_o == 8'b0);
223       assert(i2c_command_byte_o == 8'b0);
224       assert(i2c_slave_addr_o == 8'b0);
225       assert(i2c_num_bytes_o == 8'b0);
226       #1;
227       assert(i2c_din_o == 8'b0); // assert that correct data is going to be written
228       assert(i2c_command_byte_o == 8'h6b);
229       assert(i2c_slave_addr_o == {7'h68, 1'b0});
230       assert(i2c_num_bytes_o == 8'd2);
231       #1;
232       assert(i2c_write_o == 1'b1); // assert write stobe
233       #1;
234       i2c_busy_i = 1'b1; // assert that we are waiting for chip to be not busy
235       assert(i2c_write_o == 1'b0);
236       assert(error_led_o == 1'b0);
237       #1;
238       i2c_busy_i = 1'b1; // assert that we are waiting for chip to be not busy
239       assert(i2c_write_o == 1'b0);
240       assert(error_led_o == 1'b0);
241       #1;
242       i2c_busy_i = 1'b0; // make sure we have not errored out yet because we have an ack and no errors
243       assert(error_led_o == 1'b0);
244       i2c_write_done_i = 1'b1;
245       i2c_arb_lost_i = 1'b1;
246       #1; // verify write done
247       #1; // verify errors; we should get out because we lost bits
248       assert(error_led_o == 1'd1); // we should get an error
249
250
251       #100 $finish;
252     end
253
254   endmodule
```