Interfacing to External Devices

Notes and/or Reference

6.111 October 18, 2016

Huge Amount of Self-Contained Devices

- Sensors
- A-to-D converters
- D-to-A
- Memory
- Microcontrollers
- Etc…

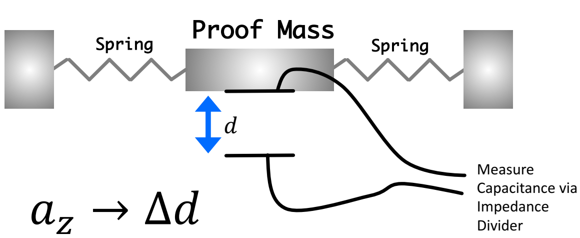
- We need ability/fluency to extract info from and work with them

Case Study

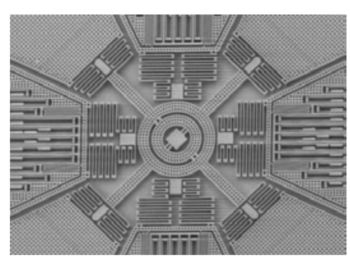
- 9 axis IMU (Inertial Measurement Unit)
  - Accelerometer
  - Gyroscope
  - Magnetometer
- One of the only real MEMS (MicroElectroMechanical Systems) applications that has gone full-scale (others might be TI's DMD, gyroscopes, microphones, some microfluidics, Si resonators, Piezoelectrics from Inkjets, etc…)

Accelerometers

- First MEMS accelerometer: 1979
- Position of a proof mass is capacitively sensed and decoded to provide acceleration data



$a_z \rightarrow \Delta d$



SEM of two-axis accelerometer

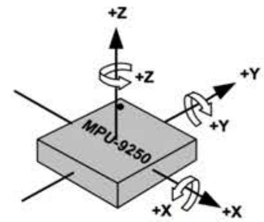
Uses of Acceleration Measurements:

- Acceleration can be used to detect motion
  - (pedometer, drop detection):

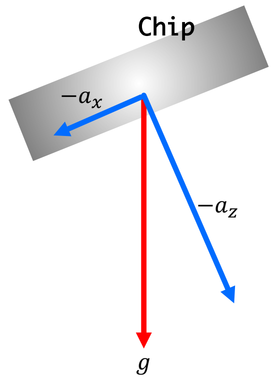
$$a_T = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

- Use gravity and trig to find orientation:

$$\theta_y = \tan^{-1}\left(\frac{a_z}{a_x}\right)$$



Accelerometer directions
+X, +Y, +Z



Problems

- Accelerometers have huge amounts of high-frequency noise
- To fix, usually Low Pass Filter the raw signal
- This cuts down on frequency response though ☹

$$\theta_y[n] = \theta_y[n-1]\beta \; + (1-\beta)\tan^{-1}\left(\frac{a_z[n-1]}{a_x[n-1]}\right)$$

a_x X acceleration

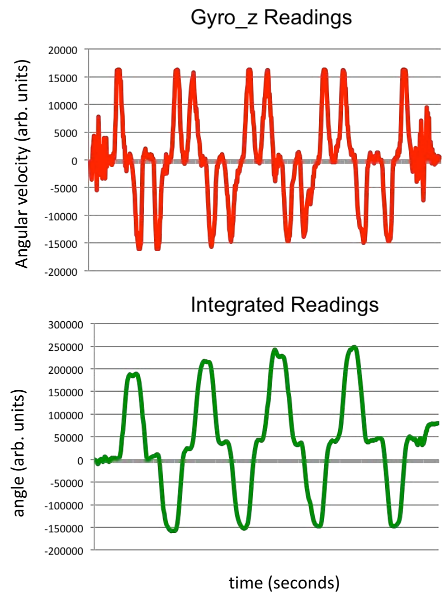
a_z z acceleration

$0 < \beta < 1$ Filter Coefficient

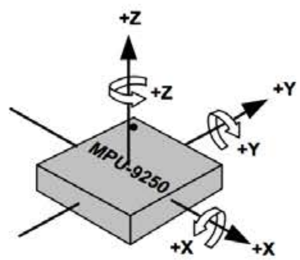
θ_y Angle estimate around y axis

Bring in Gyroscopes

- Provide Direct **Angular Velocity** which we can integrate to get angle
- Very little high-frequency noise, but lots of low frequency noise (Gyros drift like crazy)

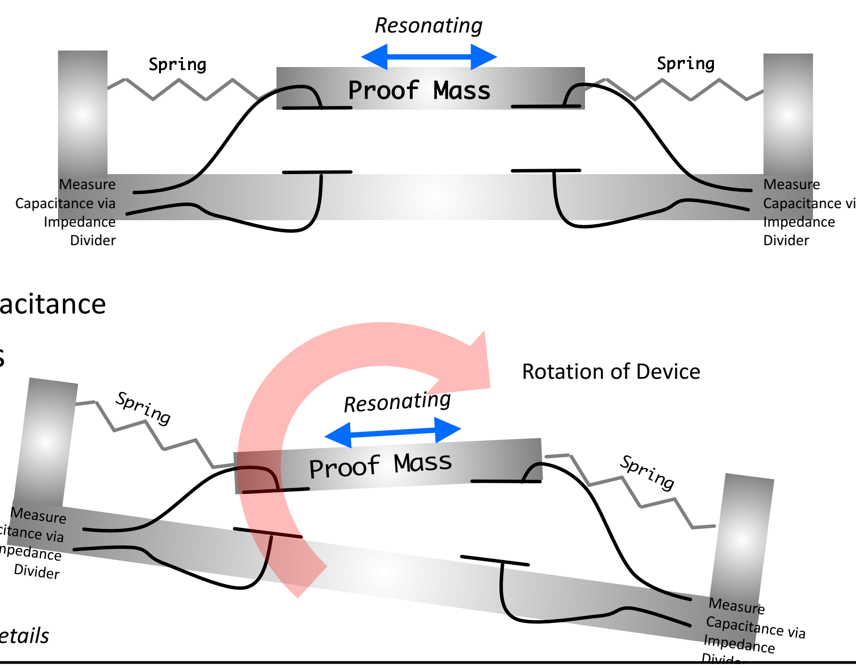


Gyro readings are "around" the axis they refer to (use right-hand rule):



Gyro Operation

- Resonating Proof Mass
  - Electrostatic Drive
  - Piezoelectric Drive
- Turning out-of-plane:
  - Proof-mass fights turn
  - Detect deviation via capacitance
- Do this for all three axes



Changes in capacitance measured at different points

Scale not accurate/nor design details

How to use Gyro Readings:

- Because of Drift (low frequency noise/offset) you want to avoid doing much long-term integration
- Having beta less than unity ensures any angle that comes from gyro reading will eventually disappear, but in short term it will dominate
- Depending on time step:

$$\theta_y[n] = \beta\theta_y[n-1] + Tg_y[n-1]$$

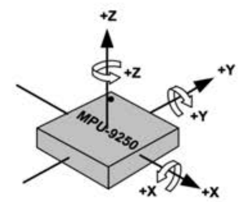
$0 < \beta < 1$ Filter Coefficient g_y Gyro y reading

$\beta \approx 0.95$ starting point T Time Step

What to do?

- Using only accelerometer, leaves us blind to motion/change in the short term but fine in the long-term
- Using only gyroscope, leaves us blind in the long term, but good in the short term
- What to do?

Merge the signals



- **Complementary Filter:**

$$\theta_y[n] = \beta\left(\theta_y[n-1] + T g_y[n-1]\right) + (1-\beta)\tan^{-1}\left(\frac{a_z[n-1]}{a_x[n-1]}\right)$$

$0 < \beta < 1$ Filter Coefficient

g_y Gyro y reading

a_x X acceleration

T Time Step

$\beta \approx 0.95$ good starting point

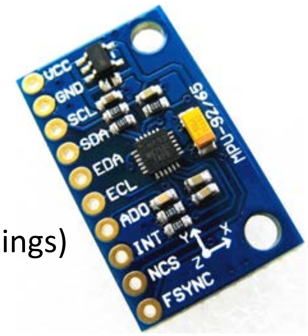
a_z z acceleration

- Could also do Kalman Filter (LQE) if desired (or others)

How to get Access to the signals in first place?

- Some accelerometers are analog out (can therefore read them with an A-to-D converter) (ADXL335, for example)
- These have limited functionality…and also it is analog so there's the whole noise issue....which is not nice
- Most flavors of sensors are digital

MPU-9250



Board: $8.00 from Ebay
Chip: $5.00 in bulk

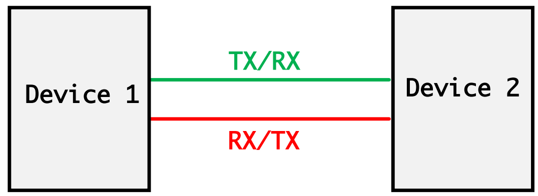
- 3-axis Accelerometer (16-bit readings)
- 3-axis Gyroscope (16-bit readings)
- 3-axis Magnetic Hall Effect Sensor (Compass) (16 bit readings)
- SPI or I2C communication (!)…no analog out
- On-chip Filters (programmable)
- On-chip programmable offsets
- On-chip programmable scale!
- On-chip sensor fusion possible (with quaternion output)!
- Interrupt-out (for low-power applications!)
- On-chip sensor fusion and other calculations (can do orientation math on-chip or pedometry even)
- So cheap they usually aren't even counterfeited! ☺

Common Device-Device Communication Protocols

- Parallel (not so much anymore)
- Serial (UART) (still common in some communication and GPS devices)
- SPI (Serial Peripheral Interface) very common
- I2C (Inter-Integrated Circuit Communication) very common

Serial (UART)

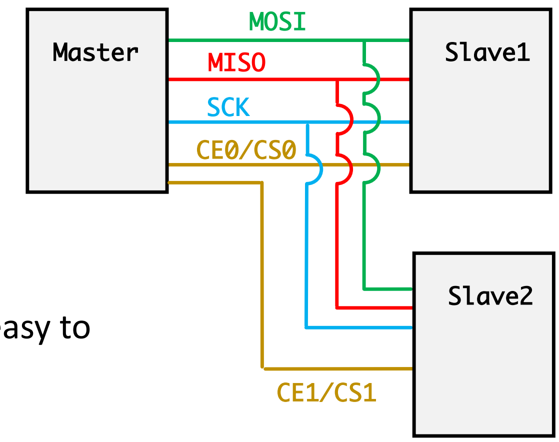
- Stands for Universal Asynchronous Receiver Transmitter
- Requires agreement ahead-of-time between devices regarding things like clock rate (BAUD), etc…
- Two wire communication
- Cannot really share
  - (every pair of devices needs own pair of lines)
- Data rate really < 115.2Kbps



SPI

- Stands for Serial-Peripheral Interface
- Four Wires:
  - MOSI: Master-Out-Slave-In
  - MISO: Master-In-Slave-Out
  - SCK: Clock
  - CE/CS (Chip Enable or Chip Select)
- SCK removes need to agree ahead of time on data rate (from UART)
- High Data Rates: (1MHz up to ~70 MHz clock (bits))

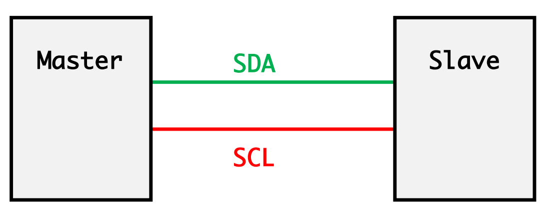
SPI



- Can share MOSI/MISO Bus
- Addition of multiple slaves requires additional select wires
- Hardware/firmware for SPI is pretty easy to implement:
  - Wires are uni-directional
  - Classic “duh” sort of approach to digital communication, but very robust.

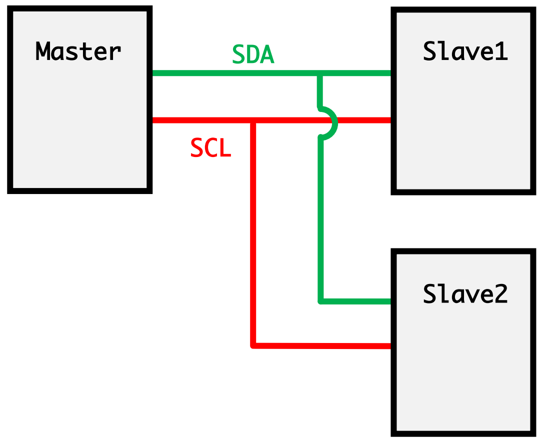
i2C

- Stands for Inter-Integrated Circuit communication
- Invented in 1980s
- Two Wire, One for Clock, one for data (both directions)
- Usually 100kHz or 400 kHz clock (newer versions go to 3.4 MHz)



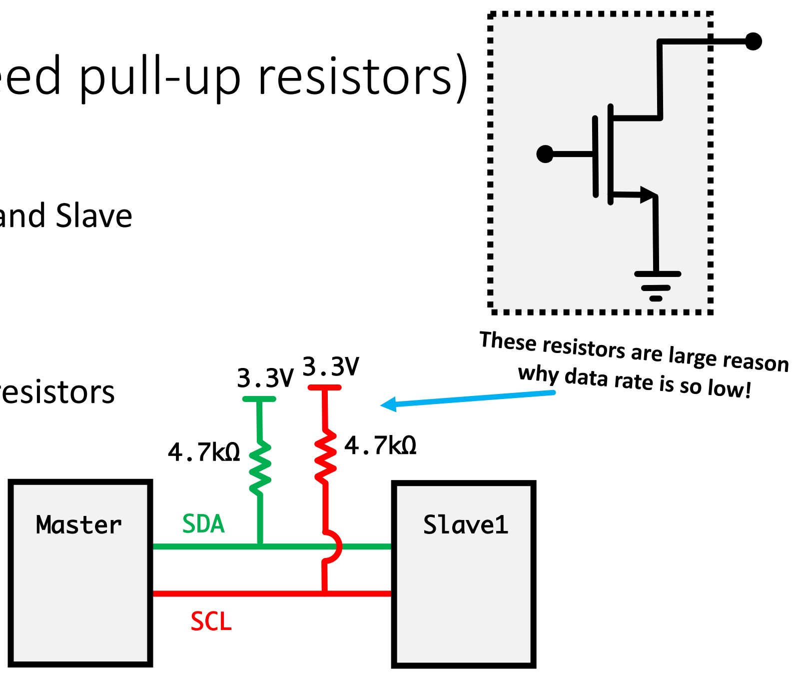
On i2C Multiple Devices Require Same # of Wires

- Devices come with their own ID numbers (originally a 7 bit value but more modern ones have 10 bits)…allows potentially up to 2^7 devices or 2^10 on a bus (theoretically anyways)
- ID's are specified at build, usually several to choose from and you select them by pulling external pins HI or LOW



More to story (need pull-up resistors)

- i2C uses an open drain
- Meaning both Master and Slave are either:
  - LOW
  - High-Impedance
- Need external pull-up resistors



Tri-State

- inout cannot be a reg ever, ever…it is closer to a wire…usual way to work with them is the following:

In verilog…

```
inout sda;

reg sda_val;

assign thing = sda_val? 1'bz: 1'b0;
```

As a result:

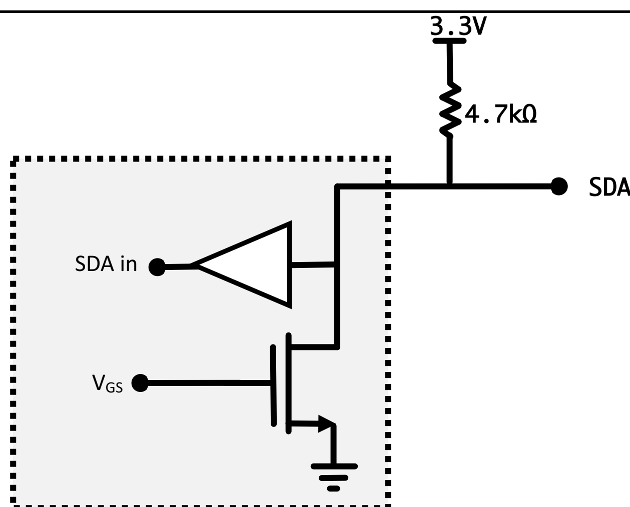
```
inout sda;
reg sda_val;
assign sda = sda_val? 1'bz: 1'b0;
```

Wanna write to SDA?

```
sda_val <= 0; //or 1 if desired :wq
```

Wanna read to SDA?

```
sda_val <= 1;
//wait clock cycle…
some_reg <= sda; //read from input
```

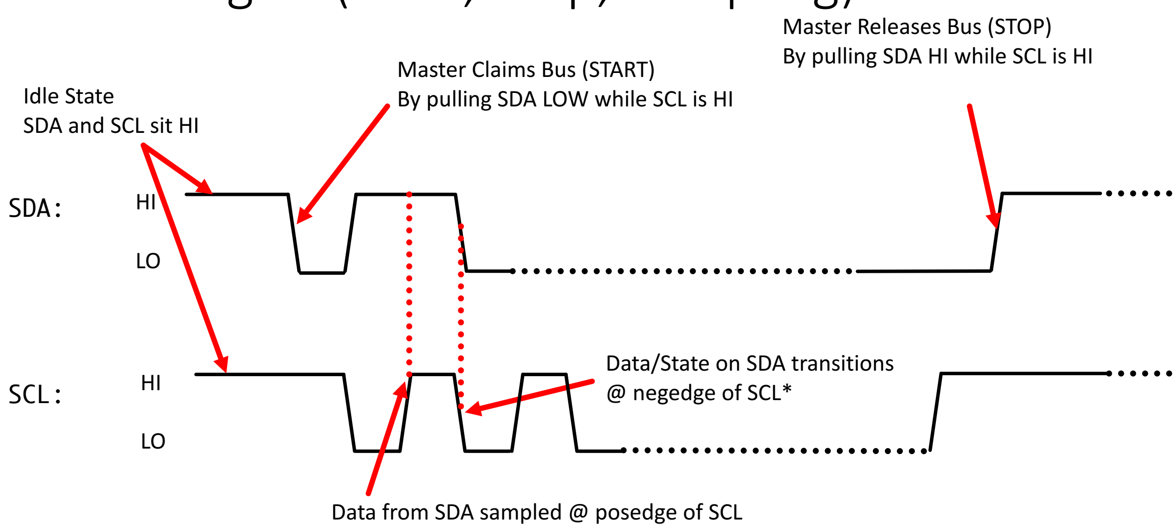


Mode	Master	Slave
Master Transmit	HiZ (HI) or LOW	HiZ (listening)
Slave ACK/NACK	HiZ (listening)	HiZ (HI) or LOW
Slave Transmit	HiZ (listening)	HiZ (HI) or LOW
Master ACK/NACK	HiZ (HI) or LOW	HiZ (listening)

i2C Operation

- Data is conveyed on SDA (Either from Master or Slave depending on point during communication)
- SCL is 50% duty cycle
- SDA generally changes on falling edge of SCL (isn't required)
- SDA sampled at rising edge of SCL
- Master is in charge of setting SCL frequency and driving it

Meanings I: (Start, Stop, Sampling)



*not specified but probably easiest spot to do

Meanings II Address

- First thing sent by Master is 7 bit address (10 bit in more modern i2C…has some leading 11111's in it..don't worry about that)
- If a device on the bus possesses that address, it acknowledges (ACK/NACK=0) and it becomes the slave
- All other devices (other than Master/Slave) will ignore until STOP signal appears later on.

Meanings III (Read/Write Bit)

- After sending address, a Read/Write Bit is specified by Master on SDA:
  - If Write (0) is specified, the next byte will be a register to write to, and following bytes will be information to write into that register
  - If Read (1) is specified, the Slave will start sending data out, with the Master acknowledging after every byte (until it wants data to not be sent anymore)

Meanings IV (ACK/NACK)

- After every 8 bits, it is the listener's job to acknowledge or not acknowledge the data just sent (called an ACK/NACK)
- Transmitter pulls SDA HI and listens for next reading (@posedge of SCL):
  - If LOW, then receiver acknowledges data
  - If remains HI, no acknowledgement
- Transmitter/Receiver act accordingly

Meanings V

- For Master to write to Slave:
  - START
  - Send Device Address (with Write bit)
  - Send register you want to write to
  - Send data…until you're satisfied
  - STOP
- For Master to read from Slave:
  - START
  - Send Device Address (with Write bit)
  - Send register you want to read from
  - ReSTART communication
  - Send Device Address (With Read bit)
  - Read in bits
  - After every 8 bits, it is Master's job to acknowledge Slave…continued acknowledgement leads to continued data out by Slave.
  - Not-Acknowledge says "no more data to Slave"
  - STOP leads to Master ceasing all communication

Implementing i2C on FPGA with MPU9250:

- Made master i2C controller in Verilog
- Used MPU9250 Data sheet: 42 pages (basic functionality, timing requirements, etc…)
- MPU9250 Register Map: 55 pages

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F
35	53	I2C_SLV4_DI	R
36	54	I2C_MST_STATUS	R
37	55	INT_PIN_CFG	R/W
38	56	INT_ENABLE	R/W
3A	58	INT_STATUS	R
3B	59	ACCEL_XOUT_H	R
3C	60	ACCEL_XOUT_L	R
3D	61	ACCEL_YOUT_H	R
3E	62	ACCEL_YOUT_L	R
3F	63	ACCEL_ZOUT_H	R
40	64	ACCEL_ZOUT_L	R
41	65	TEMP_OUT_H	R
42	66	TEMP_OUT_L	R
43	67	GYRO_XOUT_H	R
44	68	GYRO_XOUT_L	R
45	69	GYRO_YOUT_H	R
46	70	GYRO_YOUT_L	R
47	71	GYRO_ZOUT_H	R
48	72	GYRO_ZOUT_L	R

State-Machine Implementation of i2C Master

- Continuously reads 2 bytes starting at the 0x3B register (X accelerometer data)
- Print out value in hex in LEDs
- 34 States
- Clocked at 200kHz, and creates 100 kHz SCL
- Change SDA on falling edge of SCL
- Sample SDA on rising edge of SCL

```
module i2c_master(input clock,
    input reset,
    output reg [15:0] reading,
    inout sda,
    inout scl,
    output [4:0] state_out,
    output  sys_clock);

    localparam IDLE = 6'd0; //Idle/initial state (SDA= 1, SCL=1)
    localparam START1 = 6'd1; //FPGA claims bus by pulling SDA LOW while SCL is HI
    localparam ADDRESS1A = 6'd2; //send 7 bits of device address (7'h68)
    localparam ADDRESS1B = 6'd3; //send 7 bits of device address
    localparam READWRITE1A = 6'd4; //set read/write bit (write here) (a 0)
    localparam READWRITE1B = 6'd5; //set read/write bit (write here)
    localparam ACKNACK1A = 6'd6; //pull SDA HI while SCL ->LOW
    localparam ACKNACK1B = 6'd7; //pull SCL back HI
    localparam ACKNACK1C = 6'd8; //Is SDA LOW (slave Acknowledge)? if so, move on, else go back to IDLE
    localparam REGISTER1A = 6'd9; //write MPU9250 register we want to read from (8'h3b)
    localparam REGISTER1B = 6'd10; //write MPU9250 register we want to read from
    localparam ACKNACK2A = 6'd11; //pull SDA HI while SCL -> LOW
    localparam ACKNACK2B = 6'd12; //pull SCL back HI
    localparam ACKNACK2C = 6'd13; //Is SDA LOW (slave Ack?) If so move one, else go to idle
    localparam START2A = 6'd14; //SCL -> HI
    localparam START2B = 6'd15; //SDA -> HI
    localparam START2C = 6'd16; //SDA -> LOW (restarts)
    localparam ADDRESS2A = 6'd17; //Address again (7'h68)
    localparam ADDRESS2B = 6'd18; //Address again
    localparam READWRITE2A = 6'd19; //readwrite bit...this time read (1)
    localparam READWRITE2B = 6'd20; //readwrite bit...this time read (1)
    localparam ACKNACK3A = 6'd21; //like other acknacks...wait for MPU to respond
    localparam ACKNACK3B = 6'd22; //else go back to IDLE
    localparam ACKNACK3C = 6'd23; //"""""
    localparam READ1A = 6'd24; //start reading in data from device
    localparam READ1B = 6'd25; //this data is 8MSB of x accelerometer reading
    localparam ACKNACK4A = 6'd26; //Master (FPGA) assets acknowledgement to Slave
    localparam ACKNACK4B = 6'd27; //Effectively asking for more data
    localparam READ2A = 6'd28; //start reading next 8 bits (8LSB)
    localparam READ2B = 6'd29; //assign to lower half of 16 bit register
    localparam NACK = 6'd30; //Fail to acknowledge Slave this time (way to say "I'm done so slave doesn't
    localparam STOP1A = 6'd31; //Stop/Release line
    localparam STOP1B = 6'd32; //FPGA master does this by pulling SCL HI while SDA LOW
    localparam STOP1C = 6'd33; //Then pulling SDA HI while SCL remains HI
```

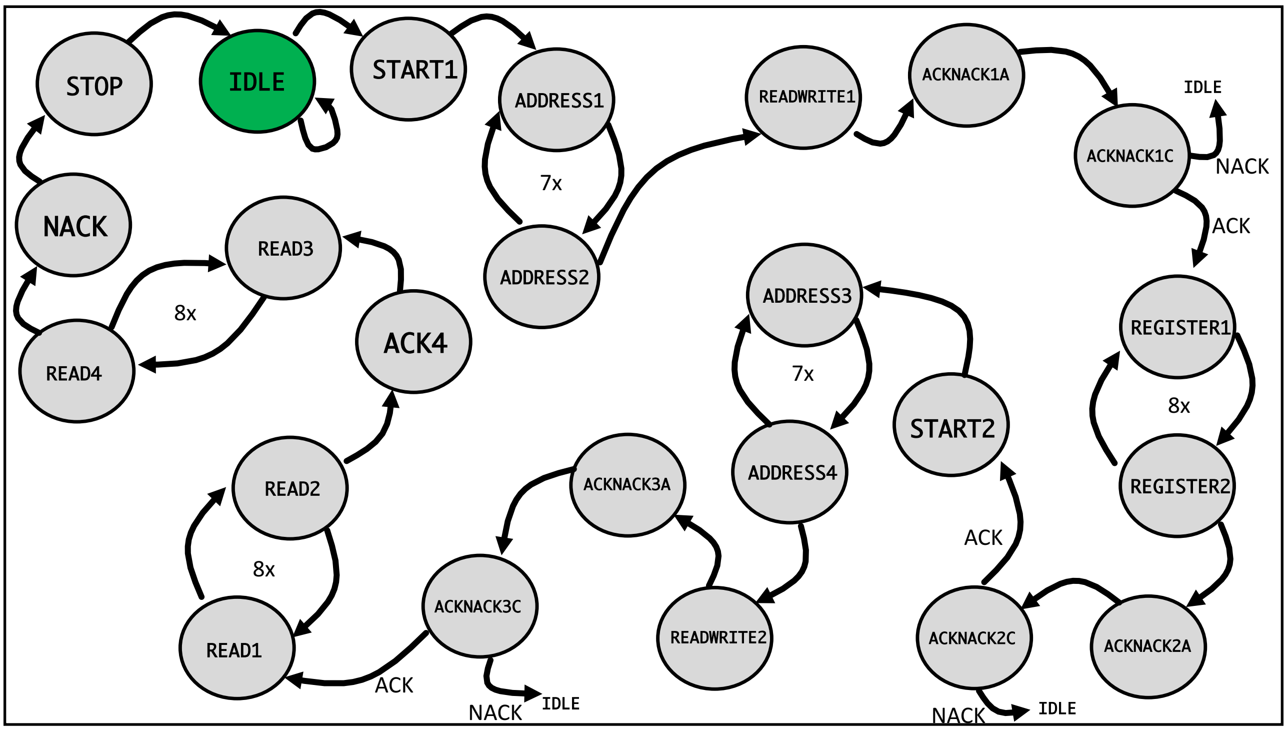
State-Machine Implementation of i2C Master

- Redundant states (repeated READ/WRITE, ADDRESS, ACK/NACK, etc…)
- ARM manual describes ~20 state FSM
- Included code on site for reference/starting point
- Diagram: on next page for reference

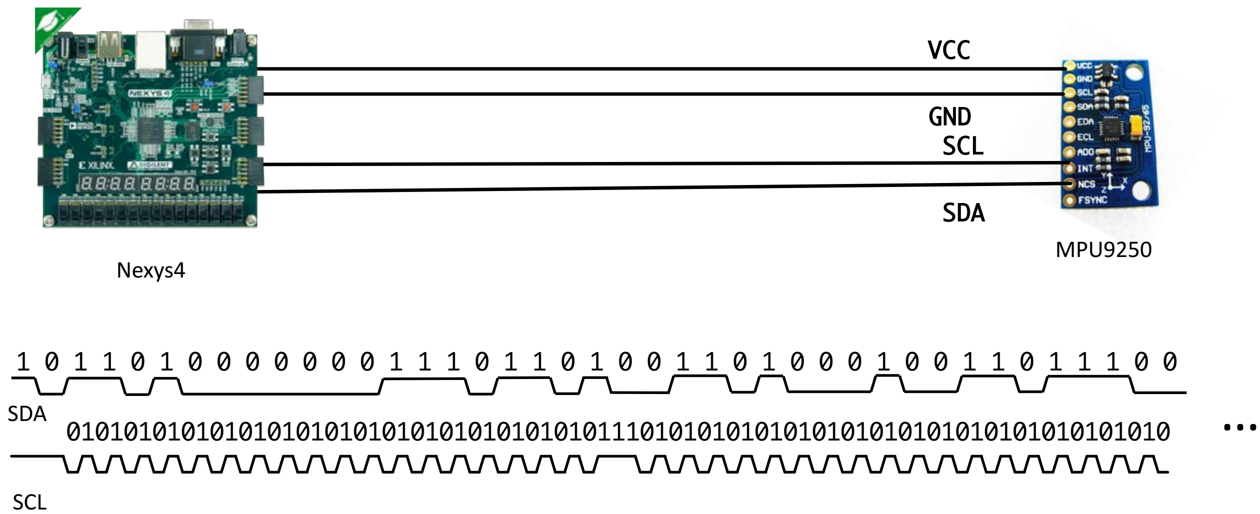
```
case (state)
    IDLE: begin
        if (reset) state <= IDLE;
        else if (count == 60)begin
            state <= START1;
            count <=0;
        end
        count <= count +1;
        sda_val <=1;
        scl_val <=1;

    end
    START1: begin
        sda_val <= 0; //pull SDA low
        scl_val <=1;
        state <=ADDRESS1A;
        count <= 6;
    end
    ADDRESS1A: begin
        scl_val<=0;
        sda_val <= device_address[count];
        state <= ADDRESS1B;
    end
    ADDRESS1B: begin
        scl_val <=1;
        if (count >= 1) begin
            count <= count -1;
            state <= ADDRESS1A;
        end else begin
            state <= READWRITE1A;
        end
    end
    READWRITE1A: begin
        scl_val <=0;
        sda_val <=0;//write address
        state <= READWRITE1B;
    end
    READWRITE1B: begin
        scl_val <=1;
        state <= ACKNACK1A;
    end
    ACKNACK1A: begin
        scl_val <=0;
        sda_val <=1; //float sda for listening next time
```

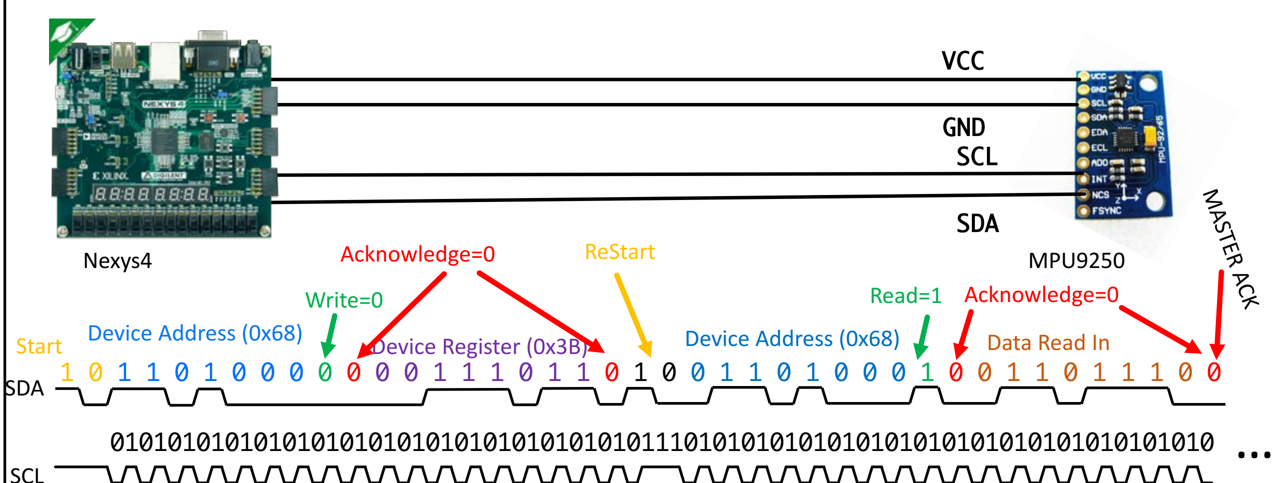
…200 more lines



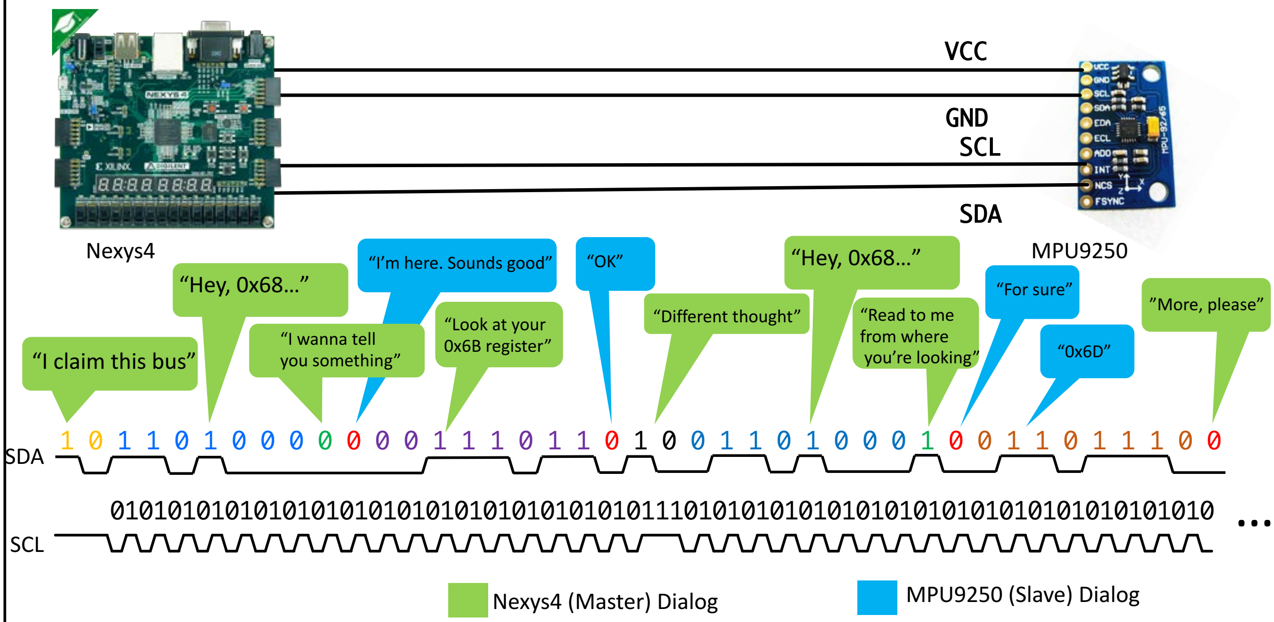
Communication Part



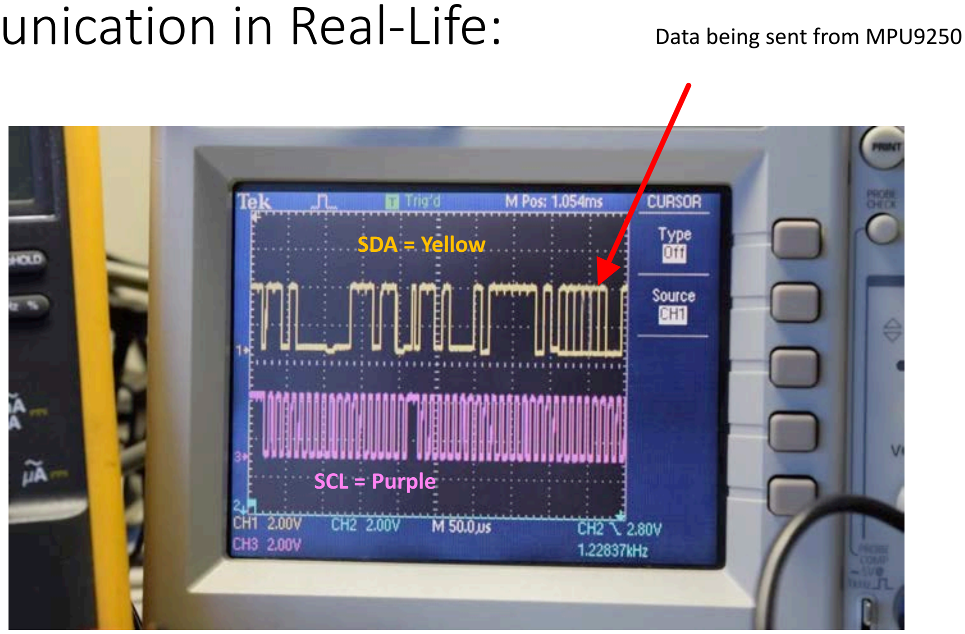
Communication Part



Communication Part

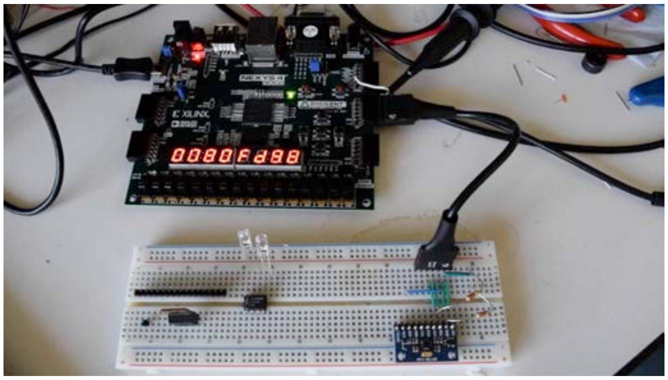


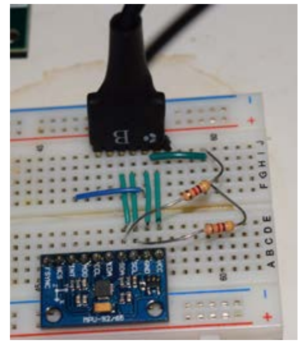
Communication in Real-Life:



Triggered on leaving IDLE state

Running and reading X acceleration:





HOOKUP

Horizontal:

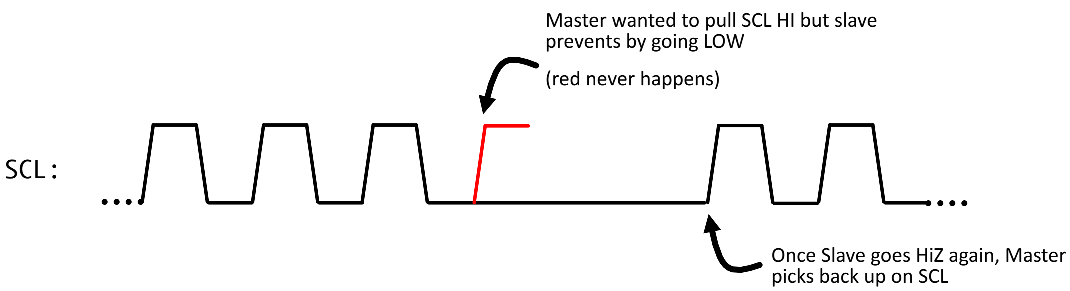
16'hFD88 = 16'b1111_1101_1000_1000 (2's complement)
Flip bits to get magnitude: 16'b0000_0010_0111_0111
=-315
Full-scale (default +/- 2g)
-315/(2**15)*2g = -0.02g ☺ makes sense

Vertical:

16'h4088 = 16'b0100_0000_1000_1000 (2's complement)
Leave bits to get magnitude: 16'b0100_0000_1000_1000
=+16520
Full-scale (default +/- 2g)
-16520/(2**15)*2 = +1.01g ☺ makes sense!

Clock-Stretching (Cool part of i2C!!!)

- Normally Master drives SCL, but since Master drives SCL high by going hiZ, it leaves the option open for Slave to step in and prevent SCL from going high by setting SCL LOW



- Allows Slave a way to buy time/slow down things (if it requires multiple clock cycles to process incoming data and/or generate output)

Final Thoughts…What about SPI or Serial?

- If you can implement i2C, the others are easier.
- SPI is also a little less standardized
- Generally with communication protocols, the more wires, the easier the protocol/less overhead
- SPI (four wires)
- Serial TX/RX (little bit more complicated, but not too bad)

- Check out the example i2C code from this lecture…see if you can add clock-stretching! (not required)