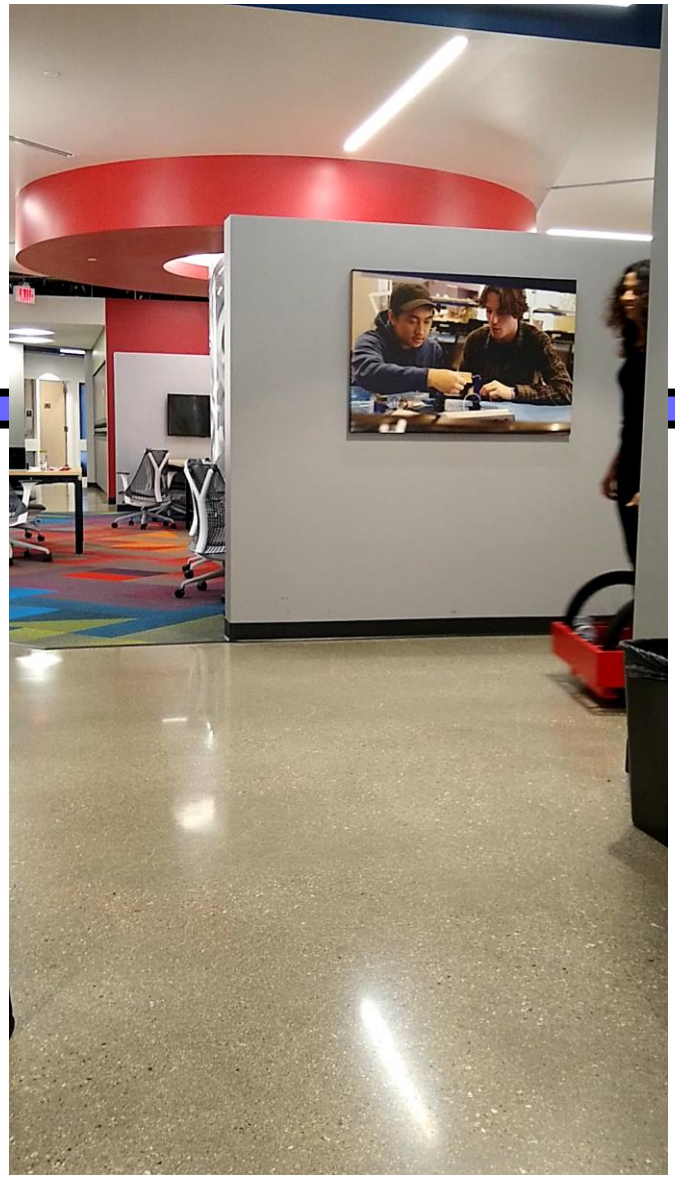
ECE 551
Project Spec

Fall ‘18

“Segway” like device



Grading Criteria: (Project is 28% of final grade)

- Project Grading Criteria:
  - Quantitative Element 15%

  (yes this could result in extra credit)

$$Quantitative = \frac{Eric\_Project Area}{YourSynthesizedArea}$$

Note: The design has to be functionally correct for this to apply

  - Project Demo (85%)
    - ✓ Code Review (10%)
    - ✓ Testbench Method/Completeness (15%)
    - ✓ Synthesis Script review (7.5%)
    - ✓ Post-synthesis Test run results (12.5%)
    - ✓ Results when placed in EricFegos Testbench (25%) (this is quantitative too: (number of tests passed)/(number of tests)
    - ✓ Test of your code mapped to the actual "Segway" and tested. (15%)

Extra Credit Opportunity:

Appendix C of ModelSim tutorial instructs you how to run code coverage

- Run code coverage on a single test and get 1% extra credit
- Run code coverage across your test suite and get a cumulative number and get 2% extra credit.
- Run code coverage across your test suite and give concrete example of how you used the results to improve your test suite and get 2.5% extra credit.

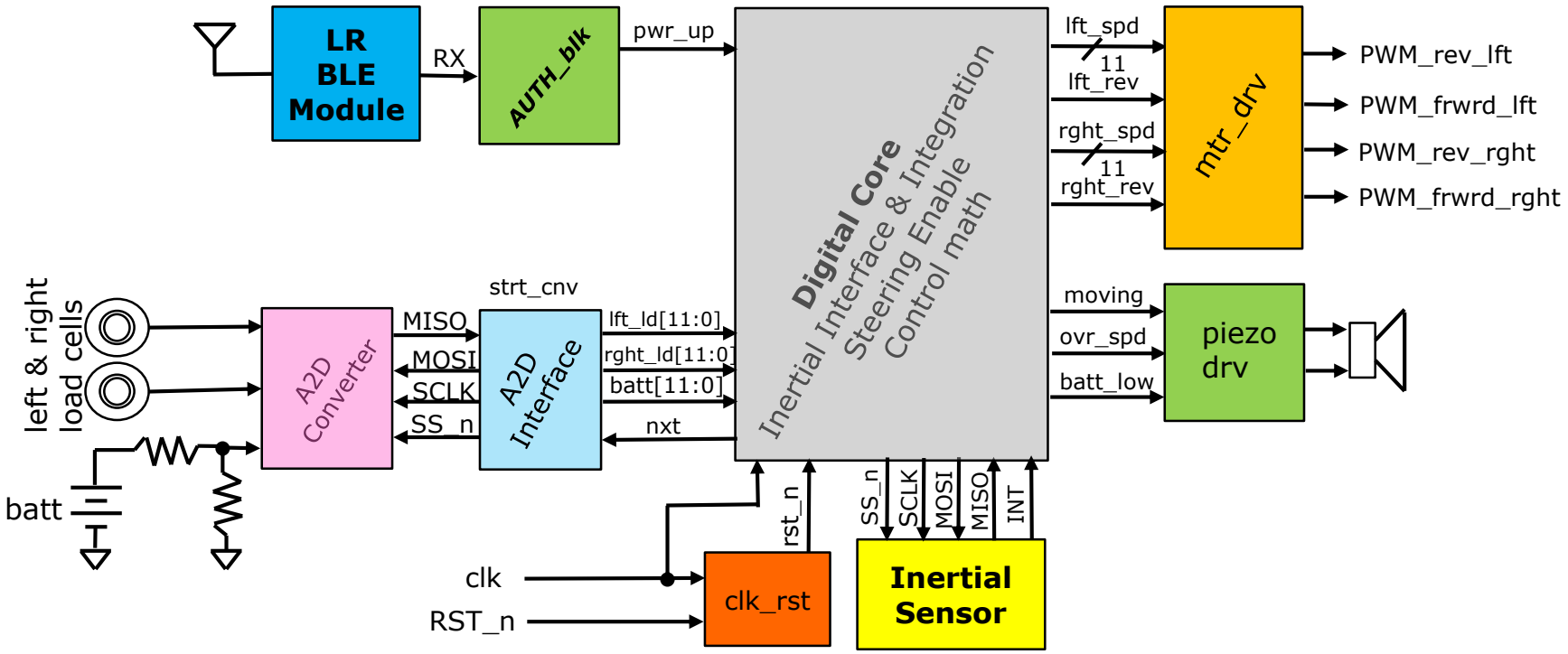
My design was about 5500 square microns

Project Due Date

- Project Demos will be held in B555:
  - Monday (12/10/18) from 2:30PM till evening. (1.5% bonus)
  - Wednesday (12/12/18) from 2:30PM till evening.
  - Thursday (12/13/18) from 1:00PM till evening. (0.75% penalty)

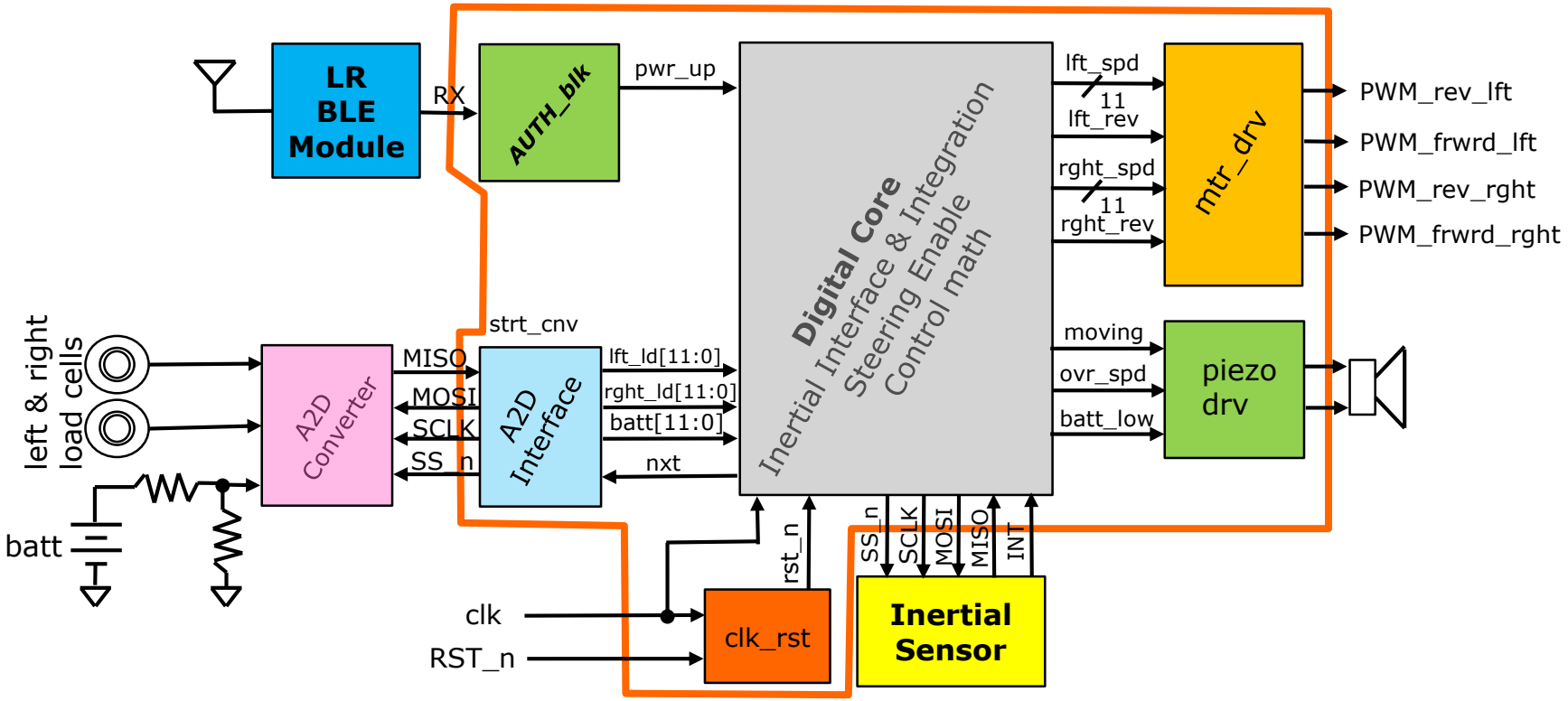
- Project Demo Involves:
  - ✓ Code Review
  - ✓ Testbench Method/Completeness
  - ✓ Synthesis Script & Results review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in EricFegos testbench
  - ✓ Results when tested on demo platform (“Segway” like device)

Block Diagram



We will be implementing the controls of a "Segway" like device. The device has two motors that can drive the left and right motors independently forward or reverse at various speeds. An inertial sensor is used to obtain the pitch of the platform of the device. If platform is pitching forward the motors are driven forward to correct the balance. If the platform is pitching backwards the motors are driven in reverse. There are load cells in the floor to determine the side to side leaning of the rider. This information is used to differentially drive left vs right motors to achieve steering. Battery voltage and PWM duty cycle are monitored and used to give audible warnings via a piezo element. Finally authentication of the rider is achieved via entering a code with their phone via **B**luetooth **L**ow **E**nergy

What is synthesized DUT vs modeled?



The blocks outlined in red above are pure digital blocks, and will be coded with the intent of being synthesized & "APRed".

You Must have a block called **Segway.v** which is top level of what will be the synthesized. (what is outlined in red).

Auth_blk (Authorized Rider)

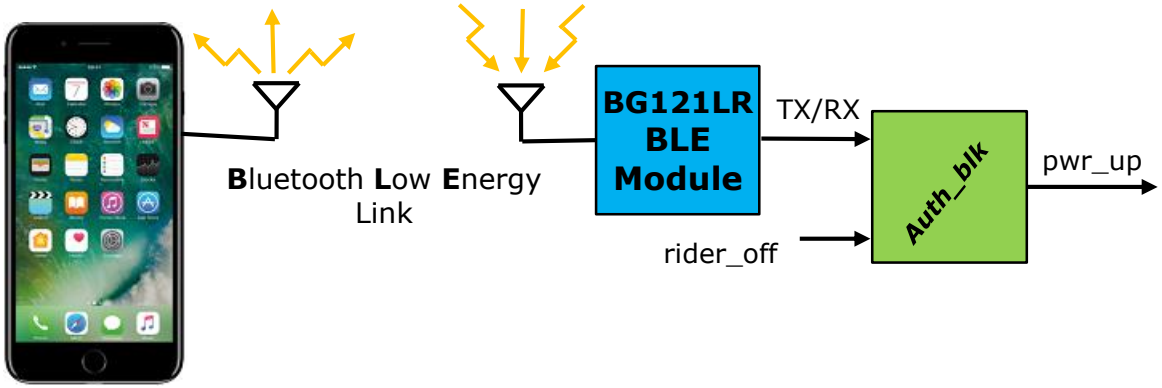
- An authorized rider will carry a phone with an app that sends the proper authorization code to a BLE module on the "Segway" device. The Segway control board has a BLE121LR module on that will be advertising a "Segway" service. When the users phone scans the service, connects, and sends the appropriate authorization code it will cause the BLE121LR module to send out 0x67 ('g') over its UART TX line. This will in turn cause the **pwr_up** signal to be asserted

- When the phone app deliberately disconnects, or is disconnected due to range the BLE121LR module sends out 0x73 ('s') over its UART TX line . The "Segway then shuts down (if the weight on the platform no longer exceeds MIN_RIDER_WEIGHT (**rider_off** signal from ***en_steer*** block).

- The UART will send at 9600 using 8N1 variant.

- Of course once the "Segway" is enabled it will stay enabled as long as there is a rider on the device. We wouldn't want it to power down and throw the rider just because the phone went dead or the BLE link was interrupted. We have a signal (**rider_off**) that indicates the weight on the platform does not exceed the minimum allowed rider weight.

- **pwr_up** goes to the ***balance_cntrl*** unit to enable it.

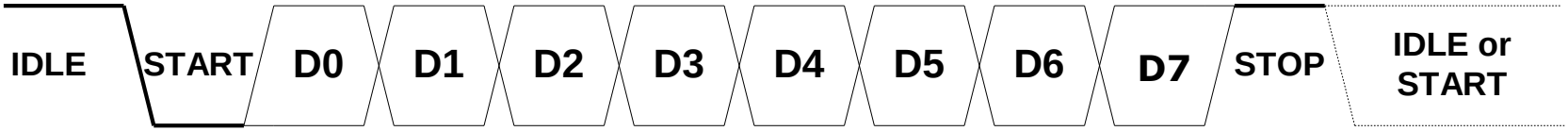


What is UART (RS-232)

- RS-232 signal phases
  - Idle
  - Start bit
  - Data (8-data for our project)
  - Parity (no parity for our project)
  - Stop bit – channel returns to idle condition
  - Idle or Start next frame

$\frac{1}{Baud}$

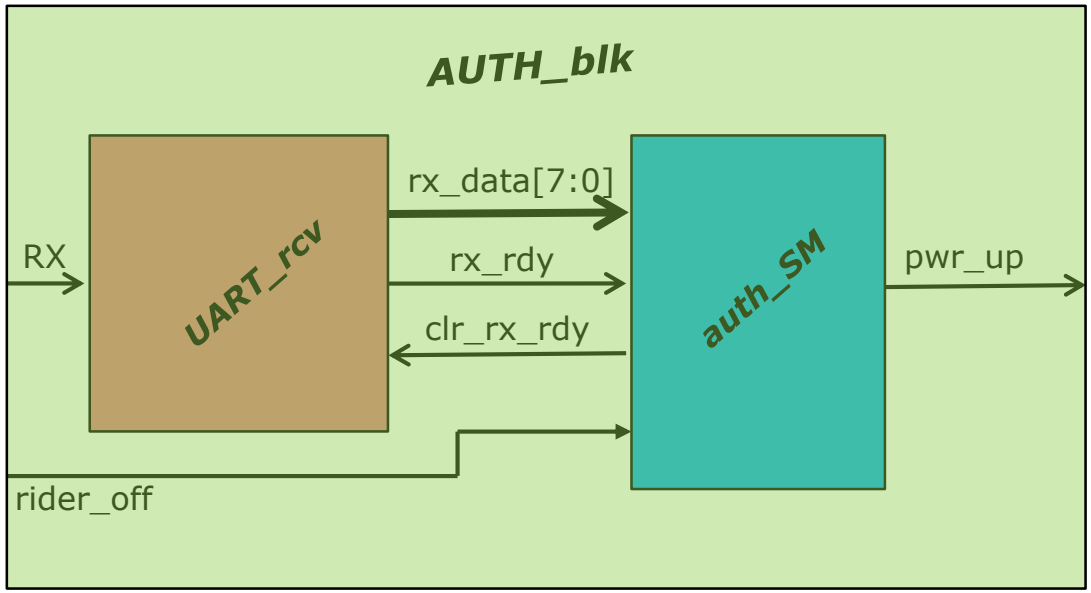
Baud rate will be 9600 with 50MHz clock



- Receiver monitors for falling edge of Start bit. Counts off 1.5 bit times and starts shifting (right shifting since LSB is first) data into a register.
- Transmitter sits idle till told to transmit. Then will shift out a 9-bit (start bit appended) register at the baud rate interval.

AUTH_blk

- **AUTH_blk** will be constructed from a UART receiver (UART_rcv) and a simple state machine comparing the receptions to 0x67, 0x73, and monitoring the **rider_off** signal.

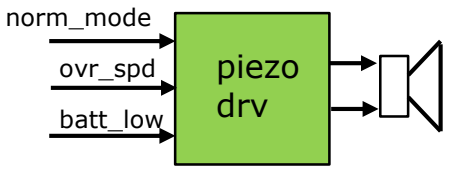


- **pwr_up** is asserted upon reception of ‘g’ (0x67). It is deasserted after the last reception was ‘s’ and the rider_off signal is high.

- **UART_rcv** will be developed as part of HW3

Piezo Driver

- The Piezo element is used to provide warnings.
  - Warning to people in vicinity when rider on
  - Warn rider when going too fast
  - Warn rider when battery is getting low



- 3 Signals come to piezo to inform of various conditions.

- Drive to piezo is simply digital square wave, and its complement.
  - Piezo will respond to signals in the 300Hz to 7kHz range

- Tone for norm_mode should occur at least once every 2 seconds, and not be too obnoxious (keep it a short burst) *(less obnoxious than school bus backing up)*

- Tone for **ovr_spd** should be alarming. Going too fast is not just dangerous because you are going fast. You are also pushing the limits of balance control. To ensure balance, the system always has to be able to drive the motors harder yet forward.

Batt Low Threshold is 0x800

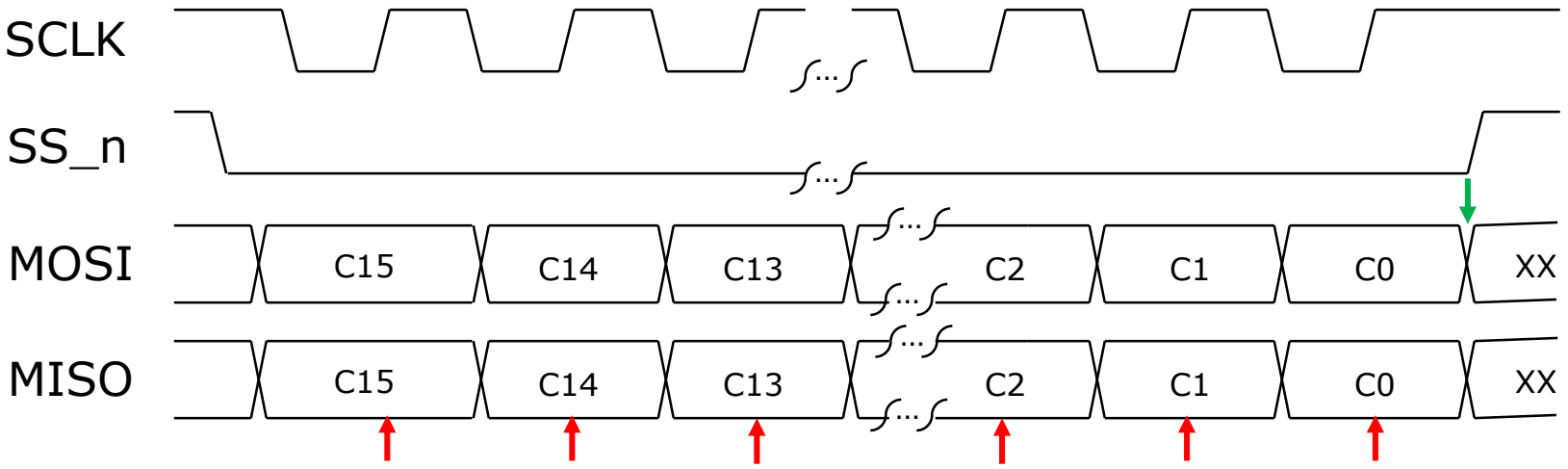
- Tone for **ovr_spd** and **batt_low** should be able to occur at same time.

- Have fun implementing the sounds, yet don't get too crazy, remember a portion of your grade is based on area.

What is SPI?

- Simple uni-directional serial interface (Motorola long long ago)
  - **S**erial **P**eripheral **I**nterconnect (very popular physical interface)
  - 4-wires for full duplex
    - ✓ MOSI (Master Out Slave In) (digital core will drive this to AFE)
    - ✓ MISO (Master In Slave Out) (not used in connection to AFE digital pots, only EEP)
    - ✓ SCLK (Serial Clock)
    - ✓ SS_n (Active low Slave Select) (Our system has 4 individual slave selects to address the 4 dual potentiometers, and a fifth to address the EEPROM)
  - There are many different variants
    - ✓ MOSI shifted on SCLK rise vs fall, MISO sampled on SCLK rise vs fall
    - ✓ SCLK normally high vs normally low
    - ✓ Widths of packets can vary from application to applications
    - ✓ Really is a very loose standard (barely a standard at all)
  - We will stick with:
    - ✓ SCLK normally high, 16-bit packets only
    - ✓ MOSI shifted on SCLK fall
    - ✓ MISO sampled on SCLK rise

SPI Packets



Shown above is a 16-bit SPI packet. The master is changing (shifting) **MOSI** on the falling edge of **SCLK**. The slave device (6-axis inertial sensor or A2D converter) changes **MISO** on the falling edge too. We sample **MISO** on the rising edge (see red arrows).

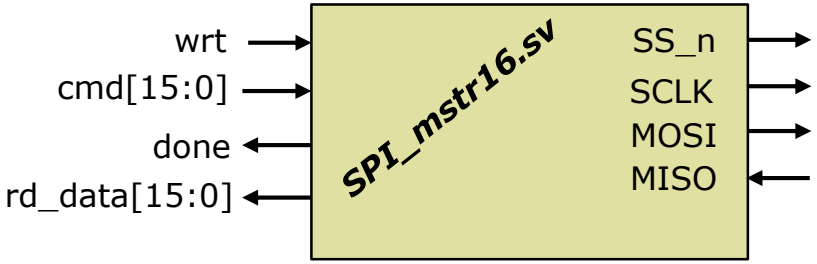
The sampled version of **MISO** in turn gets shifted into our 16-bit shift register. (on **SCLK** fall)

When **SS_n** first goes low there is a bit of a period before **SCLK** goes low. Our 16-bit shift register does not shift on the first fall of **SCLK**. This is called the "front porch".

At the end of the transaction C0 from the slave (on **MISO**) is sampled on the last rise of **SCLK**. Then there is a bit of a "back porch" before **SS_n** returns high. When **SS_n** returns high we shift our 16-bit shift register one last time (see green arrow) so "C0" captured on **SCLK** rise (last red arrow) is shifted into our shift register and we have received 16-bits from the slave.

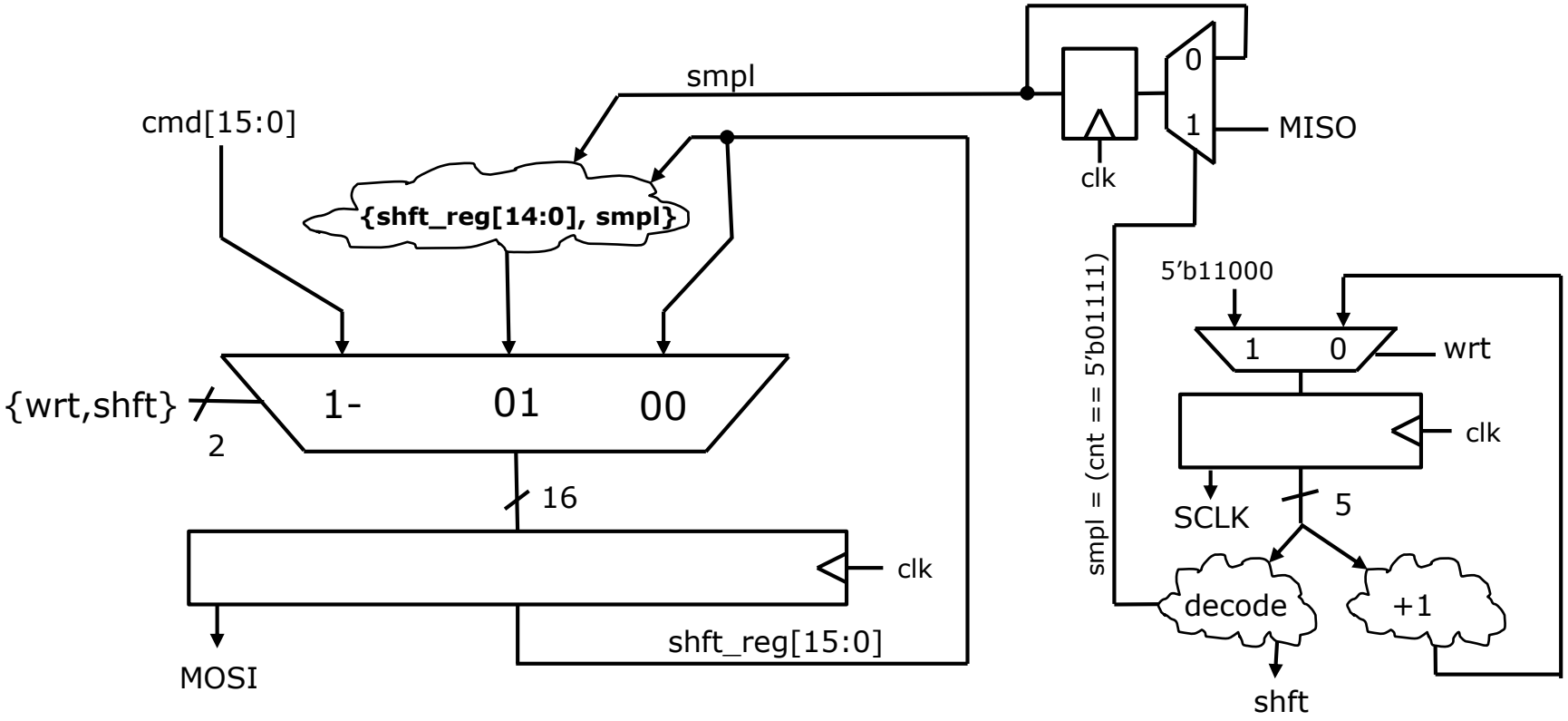
SPI Unit for Inertial Interface & A2D

- Both the 6-axis inertial sensor, and the A2D on the DE-0 Nano board can be read with a SPI master that implements the 16-bit SPI transaction mentioned above.
- You will implement **SPI_mstr16.sv** with the interface shown.
- SCLK frequency will be 1/32 of the 50MHz clock (i.e. it comes from the MSB of a 5-bit counter running off clk)
- Although the description says things like: "the shift register is shifted on **SCLK** fall" and "**MISO** is sampled on **SCLK** rise". I had better not see any ***always*** blocks triggered directly on **SCLK**. We only use **clk** when inferring flops.
- Remember you are producing **SCLK** from the MSB of a 5-bit counter. So for example, when that 5-bit counter equals 5'b01111 you know **SCLK** rise happens on the next clk, so you can enable a sample of **MISO** then. Similar logic is used for when to shift the main 16-bit shift register.



Signal:	Dir:	Description:
clk, rst_n	in	50MHz system clock and reset
SS_n, SCLK, MOSI	in	SPI protocol signals outlined above
wrt	in	A high for 1 clock period would initiate a SPI transaction
cmd[15:0]	in	Data (command) being sent to inertial sensor or A2D converter.
done	out	Asserted when SPI transaction is complete. Should stay asserted till next **wrt**
rd_data[15:0]	out	Data from SPI slave. For inertial sensor we will only ever use [7:0] for A2D converter we will use bits [11:0]

SPI Implementation

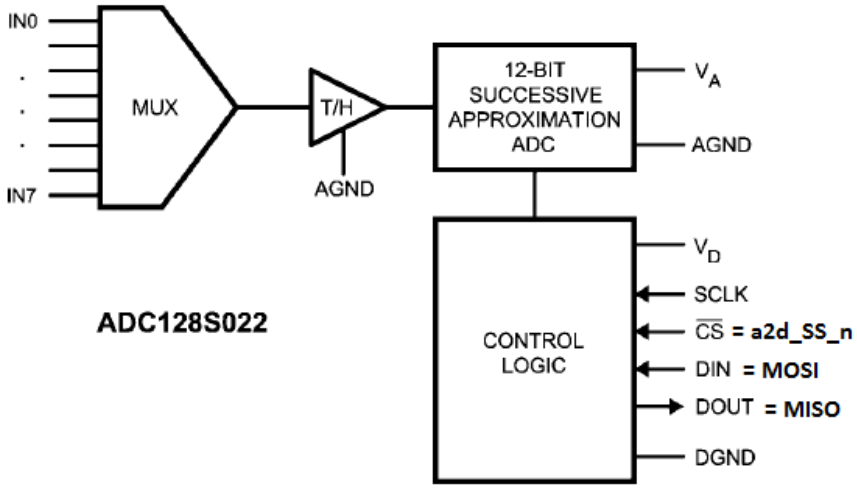


Heart of a SPI unit is a shift register and a counter. The MSB of the counter forms SCLK. The MSB of the shift register forms MOSI. A sampled version of MISO feeds into the LSB position of the shift register. Sampling and shifting are based on the value of the count. EVERYTHING works off clock, not SCLK. Of course a small state machine is needed to coordinate control of this SPI datapath to make it chooch.

A2D Converter (National Semi ADC128S022)

The DE0 Nano board contains a 12-bit eight channel A2D converter. We will make use of 3 channels. Channel 5 will be used to measure the battery voltage. Once the voltage falls below a threshold we will start warning the rider through an audible tone on the piezo buzzer.

Channels 0 and 4 are used to measure the left and right load cells respectively. These readings are used to add differential drive to the left vs right motors to enable steering.



ADC Channel:	IR Sensor:
000 = addr	Left load cell
100 = addr	Right load cell
101 = addr	Battery voltage

To read the A2D converter one sends the 16-bit packet {2'b00,addr,11'b000} twice via the SPI. There needs to be a 1 system clock cycle pause between the first SPI transaction completing, and the second one starting

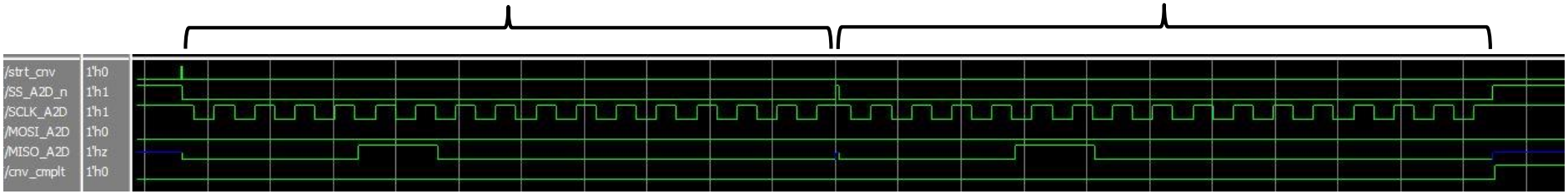
During the first SPI transaction the value returned over MISO will be ignored. The first 16-bits are really setting up the channel we wish the A2D to convert. During the 2nd SPI transaction the data returned on MISO will be the result of the conversion. Only the lower 12-bits are meaningful since it is a 12-bit A2D.

The digital core will request A2D readings in a round robin fashion on the three channels. One reading will be requested every update from the inertial sensor.

A2D Converter (Example SPI Read)

First 16-bit SPI transaction specifies The channel to perform conversion on. Data returned on MISO is junk.

Second 16-bit SPI transaction the data sent over MOSI does not really matter, just reading result over MISO.



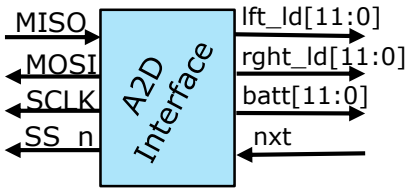
Our use of the A2D converter will involve two 16-bit SPI transactions nearly back to back (separated by 1 system clock cycle).

The first transaction here is sending a 0x0000 to the A2D over MISO. The command to request a conversion is {2'b00,channel[2:0],11'h000}. The upper 2-bits are always zero, the next 3-bits specify 1:8 A2D channels to convert, and the lower 11-bits of the command are zero. Therefore, the 0x0000 in this example represents a request for channel 0 conversion (channel 0 is the left load cell).

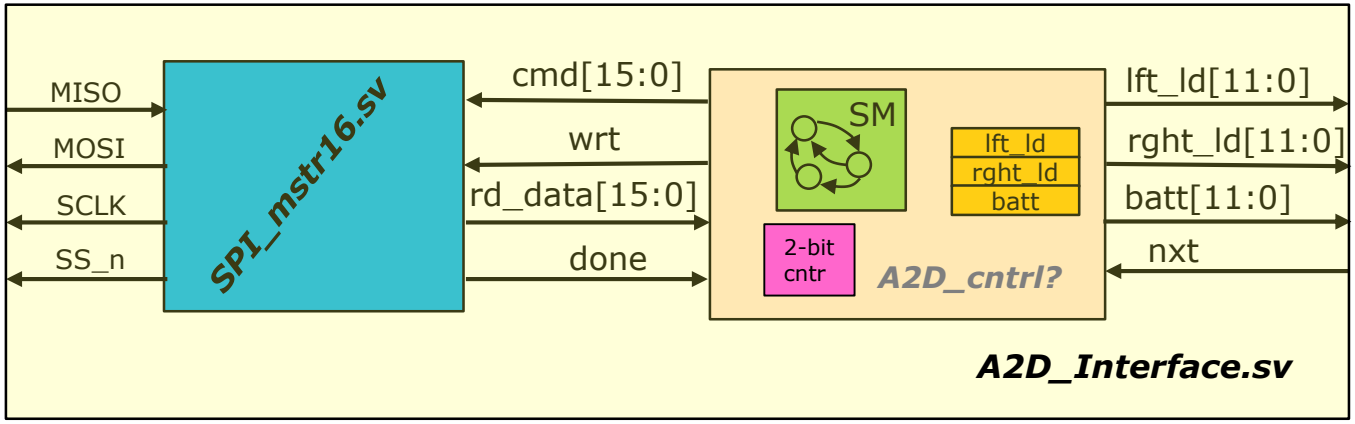
For the next 16-bit transaction the data sent over MOSI to the A2D does not matter that much. We are really just trying to get the data back from the A2D over the MISO line. The data we get back in this example is 16'h0C00. Of course since it is a 12-bit converter only the lower 12-bits (12'hC00) is valid.

Note the timing of data vs SCLK edges. Note the behavior of SS_n. The same SPI transceiver we use to read the inertial sensor also works to read the A2D. Difference is we need to perform two 16-bit SPI transactions to read a single result from the A2D.

A2D Interface

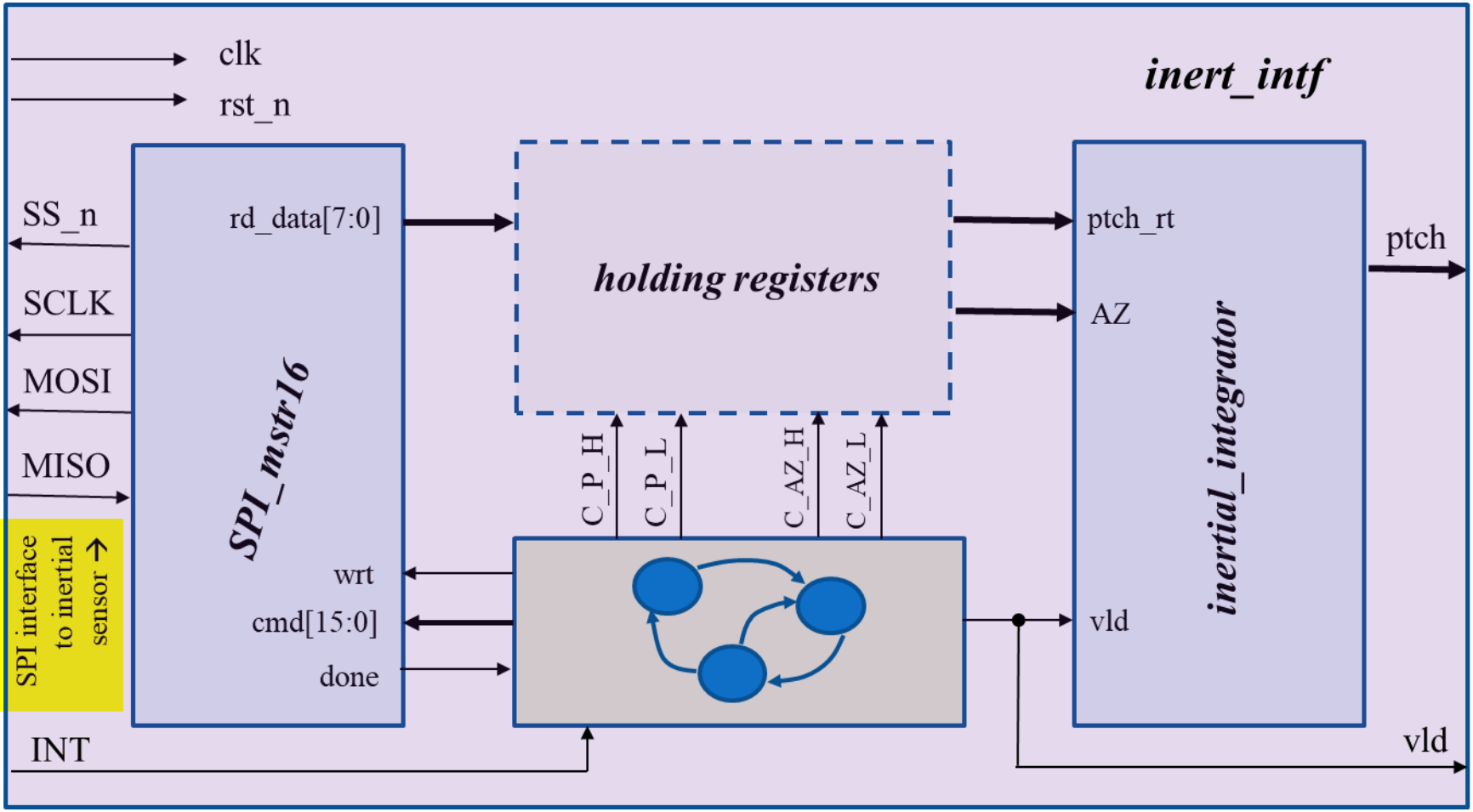


- You will build a block called A2D_Interface.sv
- It will perform round robin conversions on channels (0, 4, 5) (left load cell, right load cell, battery voltage)
- It will receive a signal (**nxt**) to initiate the next conversion. It maintains an internal 2-bit counter to know which measurand is next to be converted.



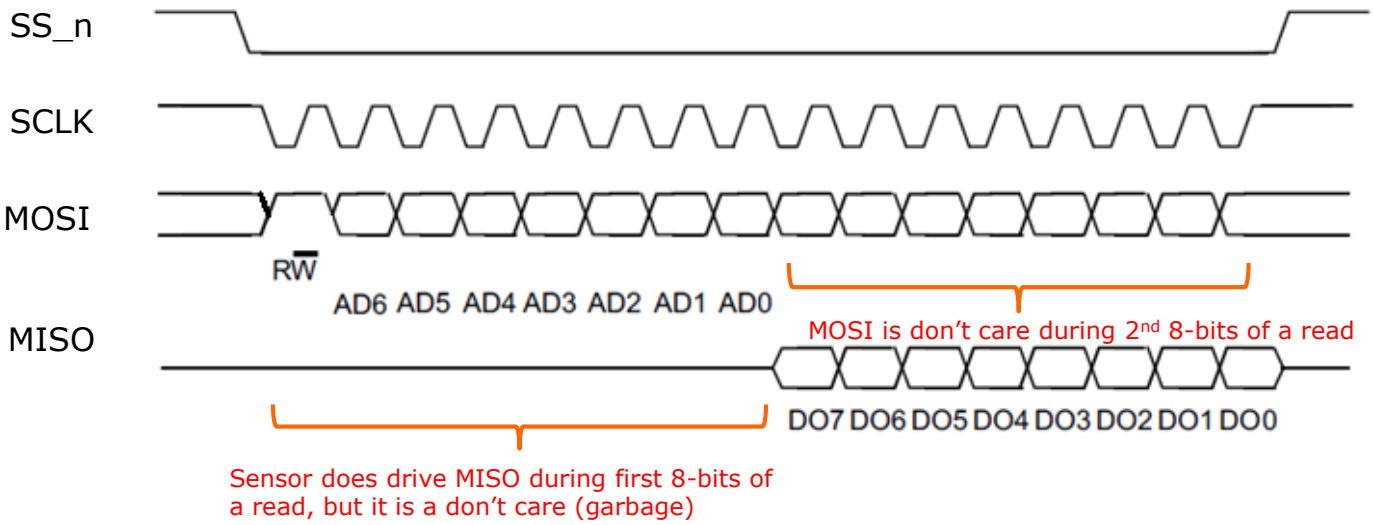
- You will make use of **SPI_mstr16**, and add control logic (State machine, some holding registers, and a counter) to produce **A2D_Interface.sv**. Whether you wrap this control logic in a level of hierarchy (**A2D_cntrl**) or code it flat at the **A2D_Interface** level is up to you.

Inertial Sensor Interface



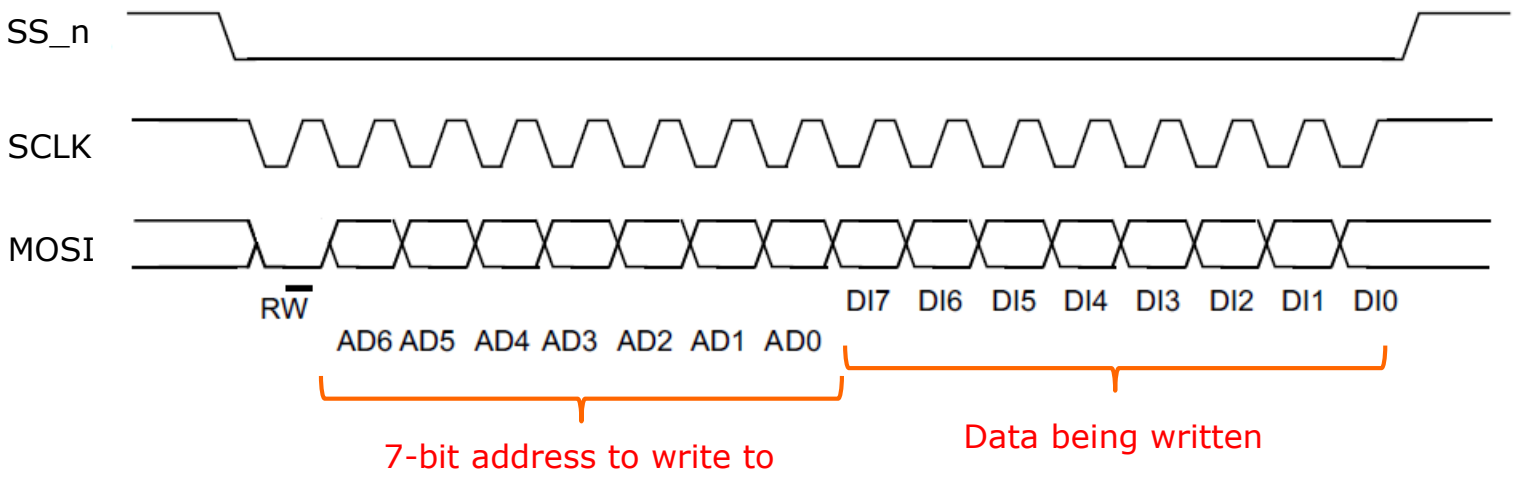
Inertial Sensor Interface

- The inertial Sensor is configured and read via a SPI interface
  - Unlike the A2D which requires two 16-bit transactions to complete a single conversion with the inertial sensor reads/writes are accomplished with single 16-bit transaction.
  - For the first 8-bits of the SPI transaction, the sensor is looking at MOSI to see what register is being read/written. The MSB is a R/$\overline{W}$ bit, and the next 7-bits comprise the address of the register being read or written.
  - If it is a read the data at the requested register will be returned on MISO during the 2nd 8-bits of the SPI transaction (see waveforms below for read)



Inertial Sensor Interface (write)

- During a write to the inertial sensor the first 8-bits specify it is a write and the address of the register being written. The 2nd 8-bits specify the data being written. (see diagram below)



- Of course the sensor is returning data on MISO during this transaction, but this data is garbage and can be ignored.

Initializing Inertial Sensor

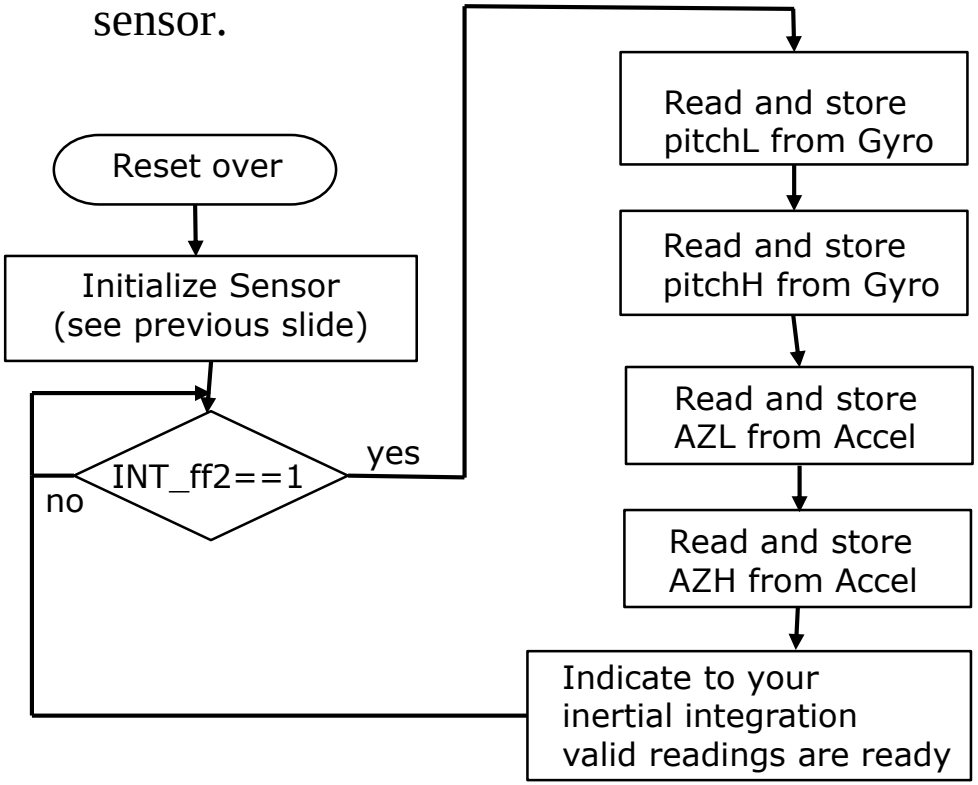
- After reset de-asserts the system must write to some registers to configure the inertial sensor to operate in the mode we wish. The table below specifies the writes to perform.

Addr/Data to write:	Description
0x0D02	Enable interrupt upon data ready
0x1053	Setup accel for 208Hz data rate, +/- 2g accel range, 50Hz LPF
0x1150	Setup gyro for 208Hz data rate, +/- 245°/sec range.
0x1460	Turn rounding on for both accel and gyro

- You will need a state-machine to control communications with the inertial sensor. Obviously we are also reading the inertial sensor constantly during normal operation. The initialization table above just specifies what some of your first states in that state-machine are doing.

Reading Inertial Sensor

- After initialization of the inertial sensor is complete the inertial interface state-machine should go into an infinite loop of reading gyro and accel data.
- The sensor provides an active high interrupt (INT) that tells when new data is ready. Double flop that signal (for meta-stability reasons) and use the double flopped version to initiate a sequence of reads (4 reads in all) from the inertial sensor.



- You will have four 8-bit flops to store the 4 needed reading from the inertial sensor. These are: pitchL, pitchH, AZL, AZH. We are only interested in determining pitch (forward/backward lean) of the platform. Since the sensor is mounted at 90° the AZ component can be used to determine pitch.

Reading Inertial Sensor (continued)

- The table below specifies the addresses you need to use to read inertial data. Recall for a read the lower byte of the 16-bit packet is a don't care.

Addr/Data:	Description:
0xA2xx	pitchL → pitch rate low from gyro
0xA3xx	pitchH → pitch rate high from gyro
0xACxx	AZL → Acceleration in Z low byte
0xADxx	AZH → Acceleration in Z high byte

Gyro Reading Integration (and drift)

- The Pitch reading we obtain from the gyro is not an angle, but rather an angular rate (we get °/sec). Therefore we have to integrate to get degrees of rotation.
- Don't worry, integration simply means summing over time. So we simply need an accumulator register that sums in the signed angular rate readings we are getting. (**ptch_int** = **ptch_int** – **ptch_rt**) (OK…subtracting rate due to orientation of the sensor mount on the platform).
- Imagine the "Segway" device was just maintaining level, and not pitched forward or back. The pitch angular rate readings would be small negative and small positive numbers, and would on average sum to zero.
- As awesome as the inertial sensor is (and it is pretty freaking amazing) it is not that good. Just the act of soldering an inertial sensor to a board will affect the offsets of its gyro readings. So even if it was perfectly factory compensated, it will not be once it is in your application.
- The small offset will be integrated over time and the reading obtained purely by integrating the gyro pitch readings will drift. We need to "fuse" the accelerometer readings (AZ) with the gyro integration to correct for this drift.

inertial_integrator.sv (at least that's what I called this unit)

inertial_integrator signal interface:

Signal:	Dir:	Description:
clk, rst_n	in	clock & reset
vld	in	High for a single clock cycle when new inertial readings are valid.
ptch_rt[15:0]	in	16-bit signed raw pitch rate from inertial sensor
AZ [15:0]	in	Will be used for sensor fusion
ptch[15:0]	out	Fully compensated and "fused" 16-bit signed pitch.

inertial_integrator internal register:

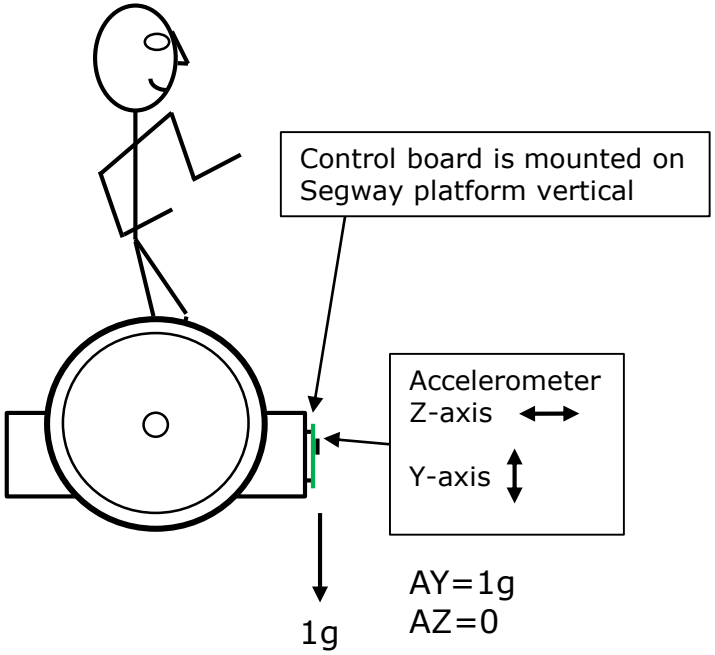
Register:	Purpose:
ptch_int[26:0]	Pitch integrating accumulator. This is the register ***ptch_rt*** is summed into

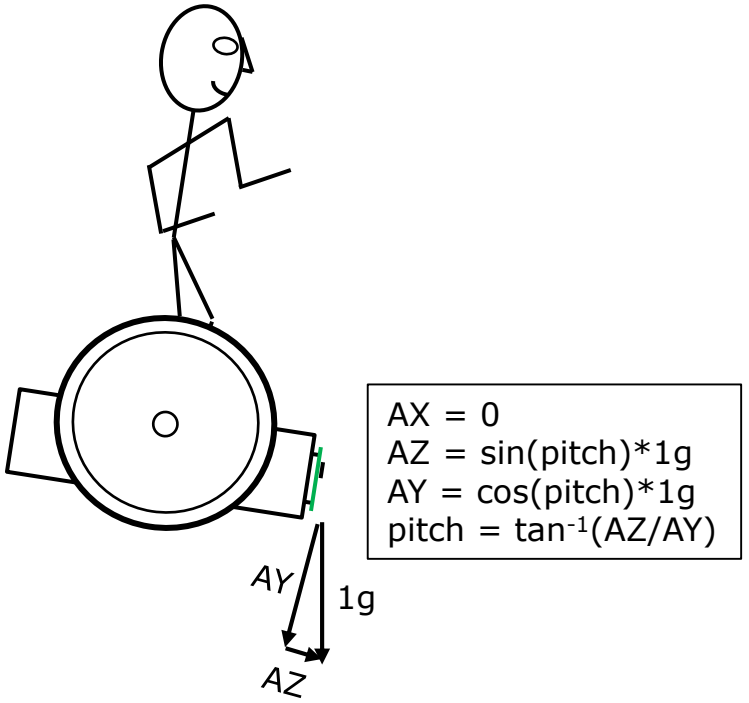
- On every **vld** pulse this unit will integrate **ptch_rt_comp** signal into **ptch_int[26:0]** (we integrate the **negative** of **ptch_rt** due to orientation of sensor mount)
- Bits [26:11] of **ptch_int** form **ptch[15:0]** (i.e. divide by 2^11). This was somewhat of an arbitrary scaling factor, just go with it, don't question it.
- There is also a fusion correction term (**fusion_ptch_offset**) that is summed into **ptch_int**. This will be discussed more later.

ptch_rt_comp = ptch_rt – PTCH_RT_OFFSET (PTCH_RT_OFFSET=16'h03C2)

What is Sensor Fusion?

- One can get pitch and roll from an accelerometer. The earth's gravity always provides one **g** of acceleration straight down.





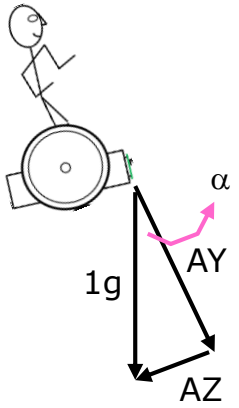
So why get pitch and roll from the gyro?
We can just use AY and AZ

Accelerometers give noisy readings. Vibrations plus acceleration in linear direction come into play.

What is Sensor Fusion?

- Accelerometers are noisy, and Gyros are subject to long term drift.
  - Accelerometer readings are noisy (vibrations and linear acceleration)
  - Gyro readings are amazingly quiet (even in presence of vibration) but since we are integrating, any remaining offset error will result in a long term drift.
- Sensor fusion is an attempt to take the best from each to create a low noise, no drift version of pitch.
  - The fusion data is dominated by the integrated gyro readings, thus it stays low noise.
  - However, pitch is also calculated via accel readings, and the long term average of the integrated gyro readings is "leaked" toward the accelerometer results. In electrical engineering terms the accelerometer gets to establish the DC operating point, but the gyro readings determine the transient response.
  - **AZ [15:0]** will be used to calculate the pitch as seen by the accelerometer (**ptch_acc**). If **ptch_acc>ptch** then **ptch_int** will have a constant added to it. If **ptch_acc<ptch** then **ptch_int** will have a constant subtracted from it.

Trigonomic Simplifications



- Remember this one? *(you should have learned it in a math class somewhere along the line)*
  - For small angles (α) the $\tan^{-1}$(opposite/adjacent) ≈ opposite/adjacent
  - Taking it one step further for small angles: adjacent ≈ hypotenuse = unity
  - So… for us pitch is simply proportional to AZ.
  - This is only true for small angles, but the "Segway" platform better be at a small angle from horizontal or the rider is in serious trouble

AZ_comp = AZ – AZ_OFFSET (AZ_OFFSET=16'hFE80)
(all sensors ever made require offset compensation)

Perform this math inside ***inertial_integrator.sv***

```
ptch_acc_product = AZ_comp * $signed(327);                    // 327 is fudge factor
ptch_acc = {{3{ptch_acc_product[25]}},ptch_acc_product[25:13]};  // pitch angle calculated from accel only

if (ptch_acc>ptch)                // if pitch calculated from accel > pitch calculated from gyro
    fusion_ptch_offset = +512;
  else
    fusion_ptch_offset = -512;
```

fusion_ptch_offset is added into next integration of **ptch_int** on every valid reading from the inertial sensor. **fusion_ptch_offset** is a 27-bit wide constant (same width as **ptch_int**)

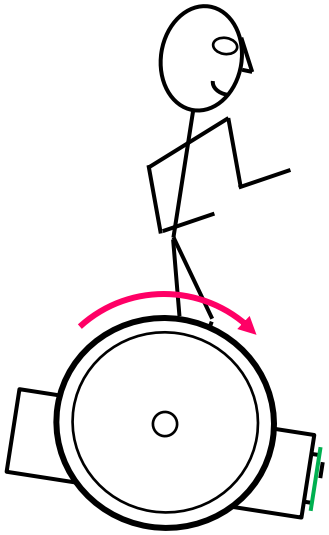
Fusion Calcs…litte more explicit

- During normal operation the pitch integrator (**ptch_int**) is summing the compensated rate signal (**ptch_rt_comp**) (OK actually the negative of it), but we are also going to sum in this **fusion_ptch_offset** term to “leak” the gyro angular measurement to agree with the accelerometer gyro measurement.

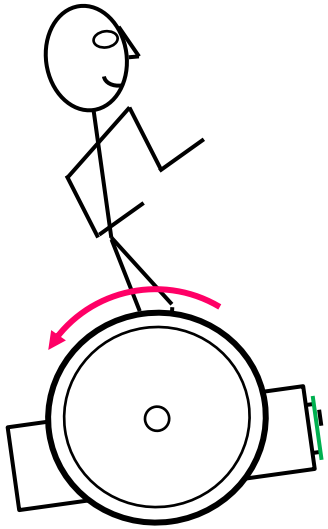
  - ```
    ptch_int <= ptch_int - {{11{ptch_rt_comp[15]}},ptch_rt_comp} +
                fusion_ptch_offset;
    ```

- During normal operation **fusion_ptch_offset** will be +512 if **ptch_acc > ptch** and will be -512 if **ptch_acc < ptch.**
    ```

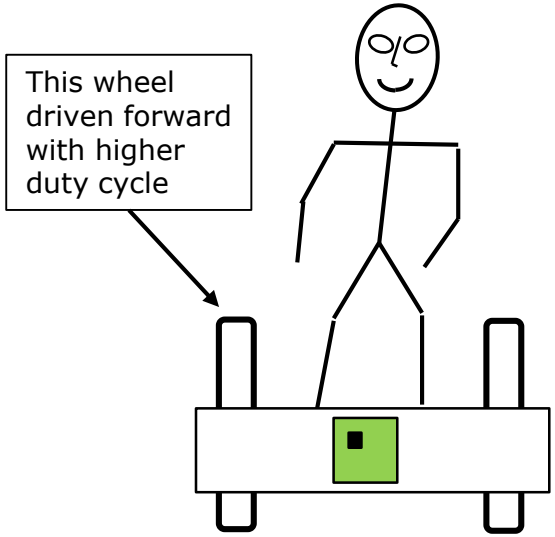
Balance & Steering Control



Rider leans forward the control has to drive the wheels forward to correct the pitch of the platform.

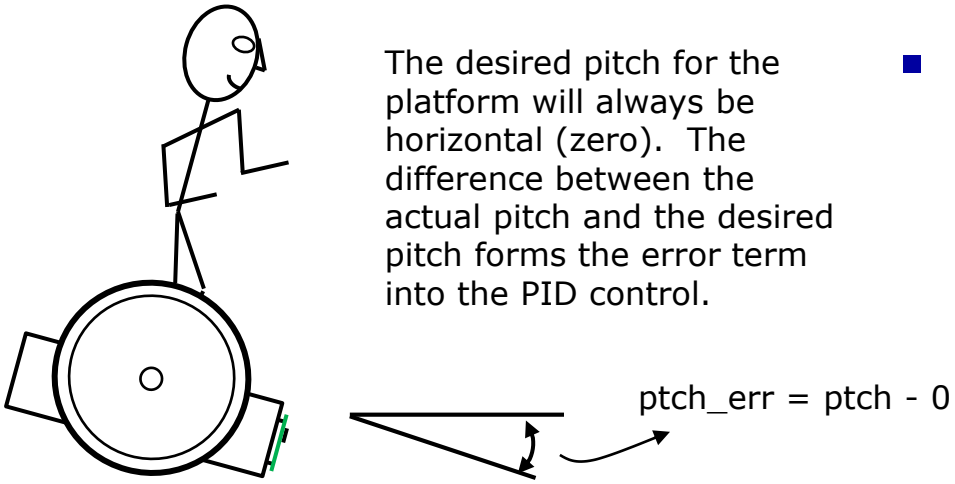


Rider leans backwards the control has to drive the wheels reverse to correct the pitch of the platform.



Load sensors in the floor of the platform will detect rider leaning to one side or the other. In the case shown the wheel on the left would be driven harder than the wheel on the right.

Balance Control PID



The desired pitch for the platform will always be horizontal (zero). The difference between the actual pitch and the desired pitch forms the error term into the PID control.

- PID is a very common control scheme. With PID control your control input is based on an error term, where the error is the actual measurement minus the desired set point

- The control input (motor duty cycle and direction in our case) is based on a combination of terms. One term **P**roportional to the error. One term that is the **I**ntegral of the error over time, and one term that is proportional to the **D**erivative of the error term.

$$PID\_cntrl = \underbrace{P_K * ptch\_err}_{\mathbf{P}} + * \underbrace{I_K \int ptch\_err}_{\mathbf{I}} + \underbrace{D_K * \frac{dptch\_err}{dt}}_{\mathbf{D}}$$

Balance Control PID (continued)

- Don't panic. Integration is simply summing. So the integral term is simply done with an accumulator register that accumulates the **ptch_err**

- Systems with very little momentum or lag in their response don't need a **D** term. Most control systems for power electronics will operate with just **PI** control since the power electronic circuit outputs respond quickly and with no "momentum" to changes in the control input.

- However, we have definite inertia/momentum in our system. Once you start pitching the platform it has angular momentum. As you are approaching flat you need to back off (or even reverse) your control input so the angular momentum does not carry you past the desired point.

- The derivative will simply be done by keeping track of the previous error terms in a queue and creating an error proportional to ***err – prev_err*** where ***prev_err*** is the oldest value in our queue**.** It is not a "true" derivative but works fine.

- The amount of **P** vs **I** vs **D** to mix together, and the depth of the ***prev_err*** queue used for **D** term were found through much trial and error. A large part of my engineering career has been built on guess and check (I am a real good guesser). The math will be specified in excruciating detail as part of HW2 & HW4. Pay very close attention to it and implement it exactly as specified.

Steering Control

- Load sensors in the floor are measuring the side to side loading of the platform. As the rider leans to the right we want the "Segway" device to turn to the right. Meaning we want the left motor driven harder, and the right motor driven softer.

- Your design will be performing A2D measurements of the load sensors, and computing the difference. If steering is enabled then 1/8 of this difference is subtracted/added to left vs right motor duty cycles.

$$\text{load\_diff} = \text{load\_cell\_left} - \text{load\_cell\_right};$$

$$\text{lft\_torque} = (\text{en\_steer}) \; ? \; (\text{PID\_cntrl} - \text{sign\_extend}_{16}(\text{load\_diff}[11:3]) \; ? \; \text{PID\_cntrl};$$

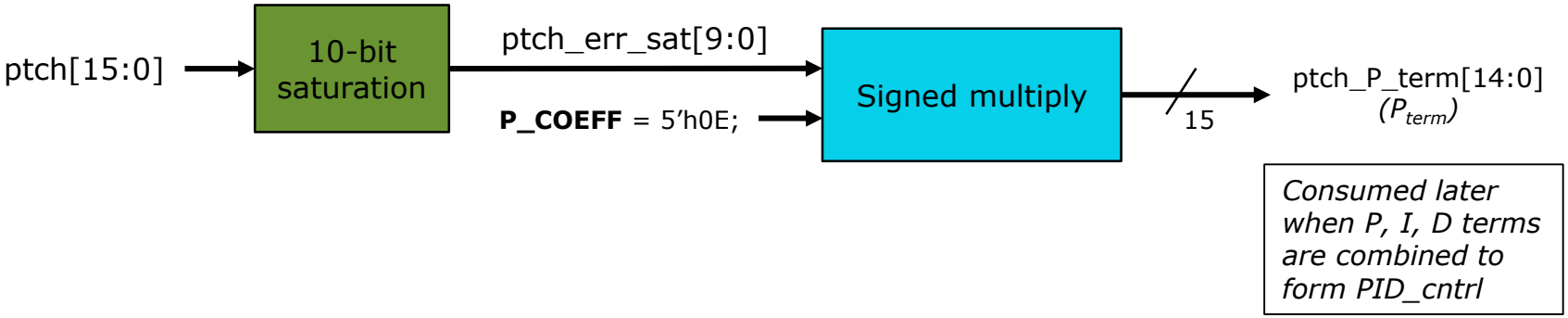
$$\text{rght\_torque} = (\text{en\_steer}) \; ? \; (\text{PID\_cntrl} + \text{sign\_extend}_{16}(\text{load\_diff}[11:3]) \; ? \; \text{PID\_cntrl};$$

- The signals above (**lft_torque/rght_torque**) represent the desired torque each motor should apply. Unfortunately brushed DC motors have a nasty deadband in their torque vs applied voltage, so next we need compensate for this next.

- The torque deadband compensation is discussed later

- Finally the actual motor control is done with sign/magnitude (direction & PWM duty cycle) not with signed 2's complement numbers. Therefore we need to take the absolute value to eventually drive into our lft/rght PWM units.

PID Math (more detail…P_{term})

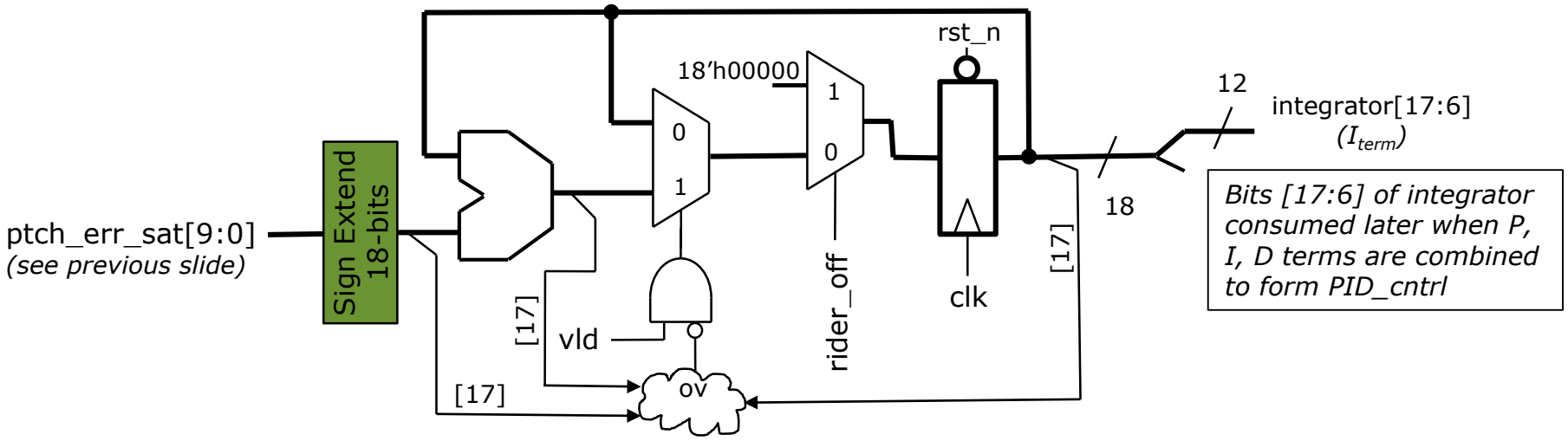


The P_{term} is the simplest of the terms to generate. All you need to do is saturate **ptch** into a 10-bit signed number and multiply it by P_COEFF. You need to ensure you infer a signed multiplier, which means both operands need to be signed ($**signed(P_COEFF)**) and the result it is assigned into (**ptch_P_term**) must be of type signed.

P_COEFF should be a localparam so it could be easily changed.

The saturated error term (**ptch_err_sat**) will also be used in both the **I** and **D** portions of the calculations.

PID Math (more detail…I_{term})

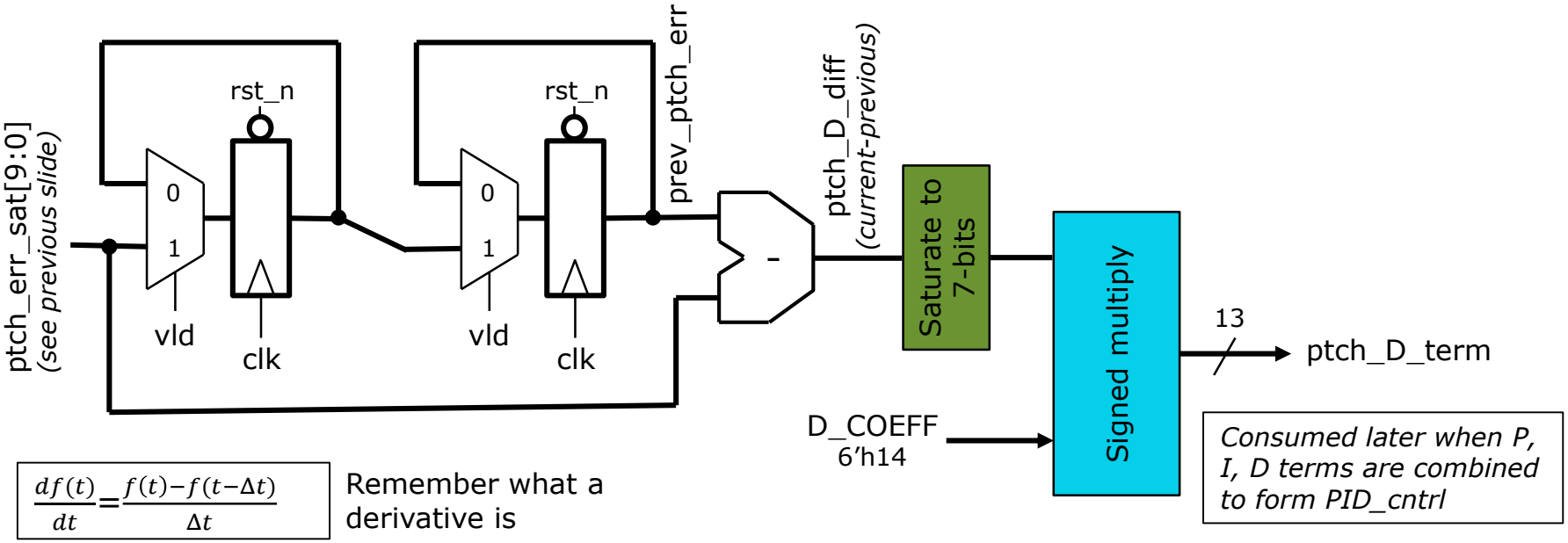


The primary function of the I_{term} is quite simple. On every **vld** reading from the inertial sensor the saturated version of pitch error (**ptch_err_sat**) is accumulated into an 18-bit accumulator register. We then use the upper bits ([17:6]) of this accumulator to form our I_{term} that summed with our P_{term} and D_{term} to form PID_cntrl.

When the rider steps off the "Segway" it is possible the I_{term} was kind of "wound up" We don't want the "Segway" ramming them or running away after they step off, so we clear the integrator on **rider_off**.

We don't want to let the integrator roll over (either beyond its most positive number or it most negative number. The easiest way to accomplish this is two inspect the MSBs of the two numbers being added, if they match, yet do not match the result of the addition then overflow occurred. If overflow occurs we simply freeze the integrator at the value it was at last which must have been pretty close to a full positive or negative number.

PID Math (more detail…D_{term})

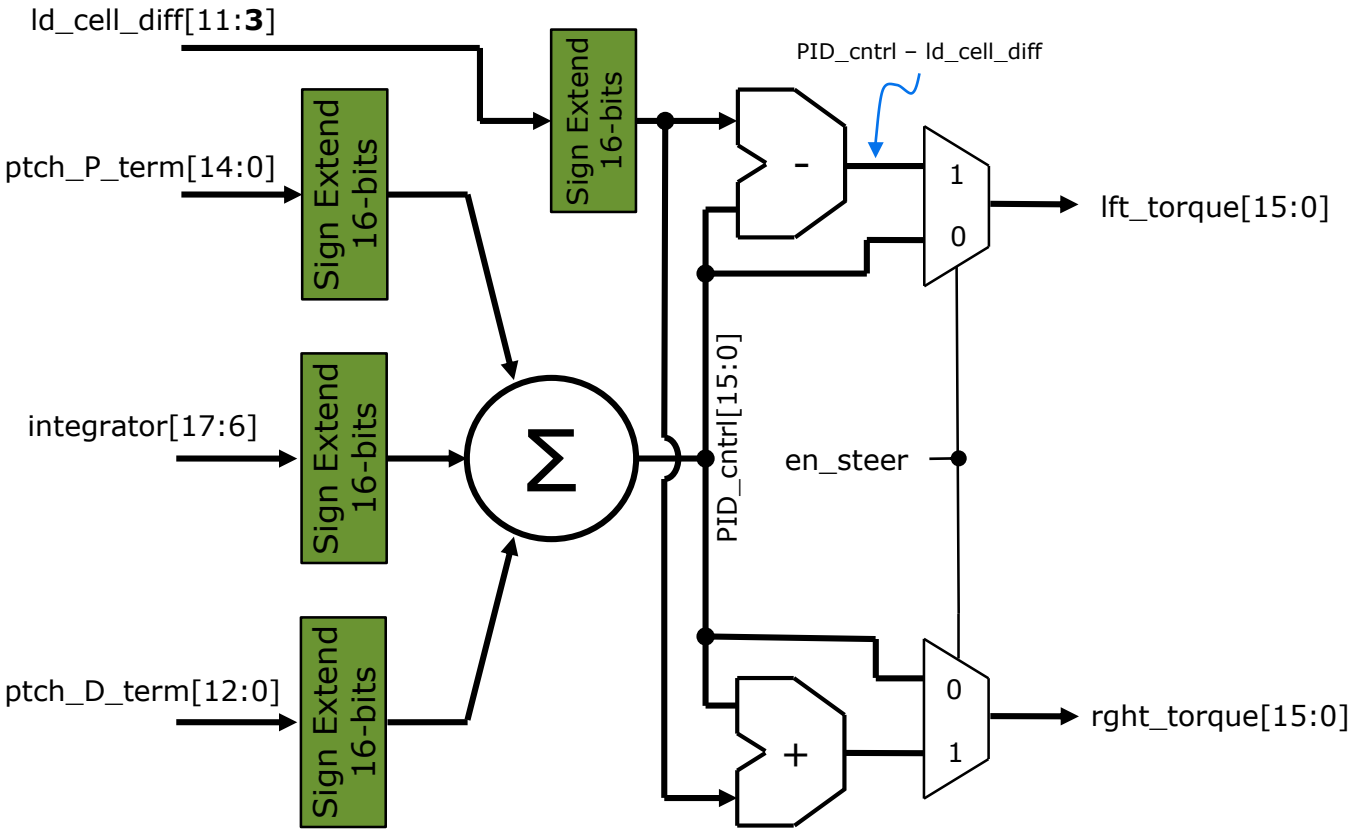


$$\frac{df(t)}{dt} = \frac{f(t) - f(t-\Delta t)}{\Delta t}$$

Remember what a derivative is

For us Δt is related to the sample rate of the inertial sensor (two **vld** readings). So our derivative is simply proportional to the current reading of **ptch_err_sat** minus a stored sample from two readings ago. There is no reason to divide by Δt since it is a constant and we need to scale the number anyway. This result is saturated to a 7-bit number and then scaled by a coefficient (D_COEFF) implemented as a ***localparam*** that we will set to 6'h14 for now.

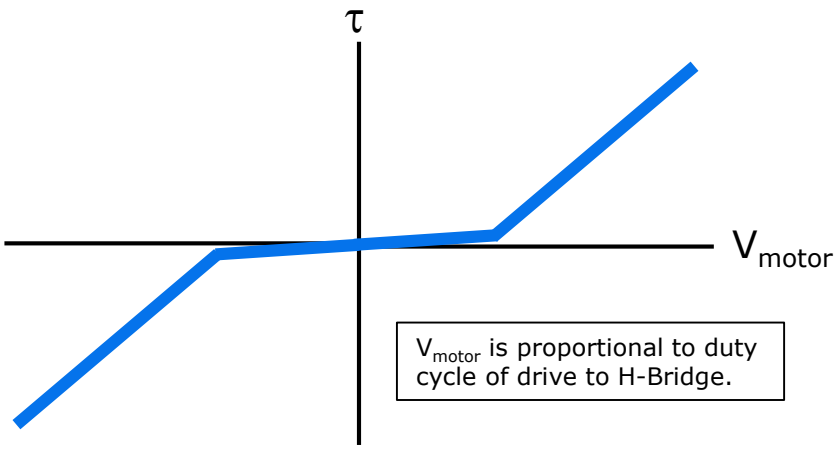
PID Math (Putting **PID** together, and steering)



We sum the 3 terms together to form **PID_cntrl**[15:0]. Then if steering is enabled then 1/8 the difference between left and right load cells is subtracted/added to form a differential drive.

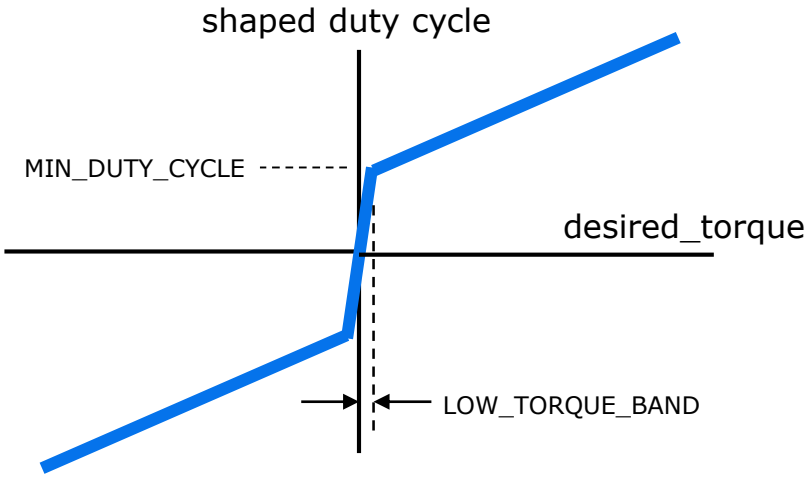
The resulting terms (**lft_torque/rght_torque**) represent the torque we desire for each motor to apply to the wheels. Next we need to compensate for the nasty deadband DC motors have in their torque vs applied voltage.

DC Motor Dead Band (need for MIN_DUTY_CYCLE & **H**igh **G**ain zone)



Shown is a graph that represents the torque of a DC motor (τ) vs the voltage applied to the terminals of the motor (V_{motor}). Notice the "dead band" in the middle? For example, we are using 24V motors, but they cannot overcome their own internal friction from -11.5V to +11.5V. With a voltage magnitude that exceeds 11.5V the torque & speed increases linearly with voltage.

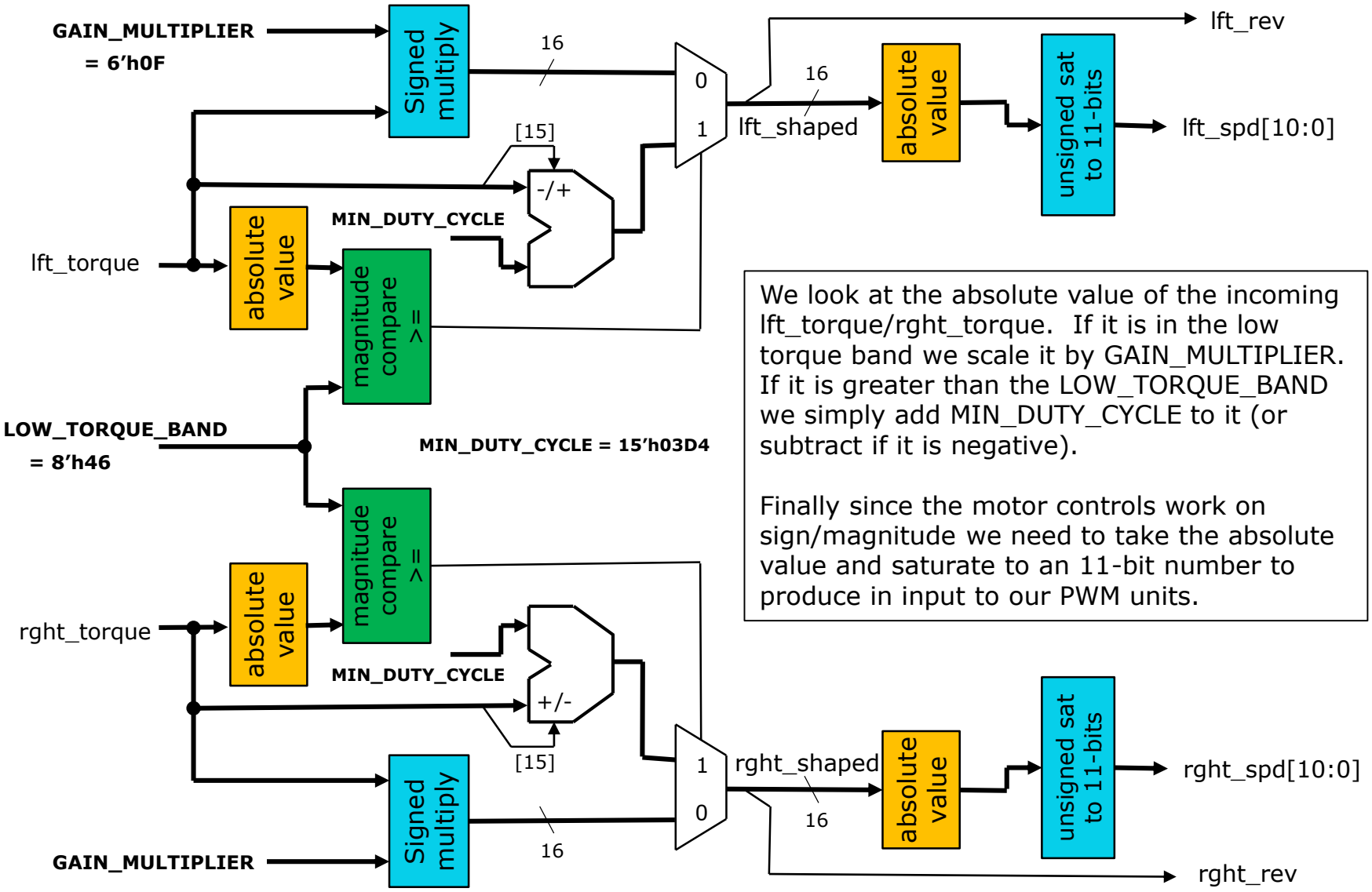
Imagine the havoc this deadband is going to cause for our control system if we do not compensate for it.



We will scale/shape our desired torque to compensate for the DC motors dead zone and compress it into a small range of desired torques: **(-LOW_TORQUE_BAND,LOW_TORQUE_BAND).**

When: |**desired_torque**| < **LOW_TORQUE_BAND** we are in the steep part of the compensation and will scale the **desired_torque** by **GAIN_MULTIPLIER** (much greater than unity). When |**desired_torque**| ≥ **LOW_TORQUE_BAND** **desired_torque** will not be scaled up, but will have **MIN_DUTY_CYCLE** added to it if it is positive, or subtracted if it is negative.

Shaping Desired Torque to form Duty

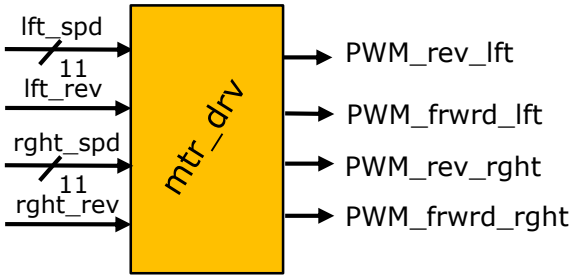


We look at the absolute value of the incoming lft_torque/rght_torque. If it is in the low torque band we scale it by GAIN_MULTIPLIER. If it is greater than the LOW_TORQUE_BAND we simply add MIN_DUTY_CYCLE to it (or subtract if it is negative).

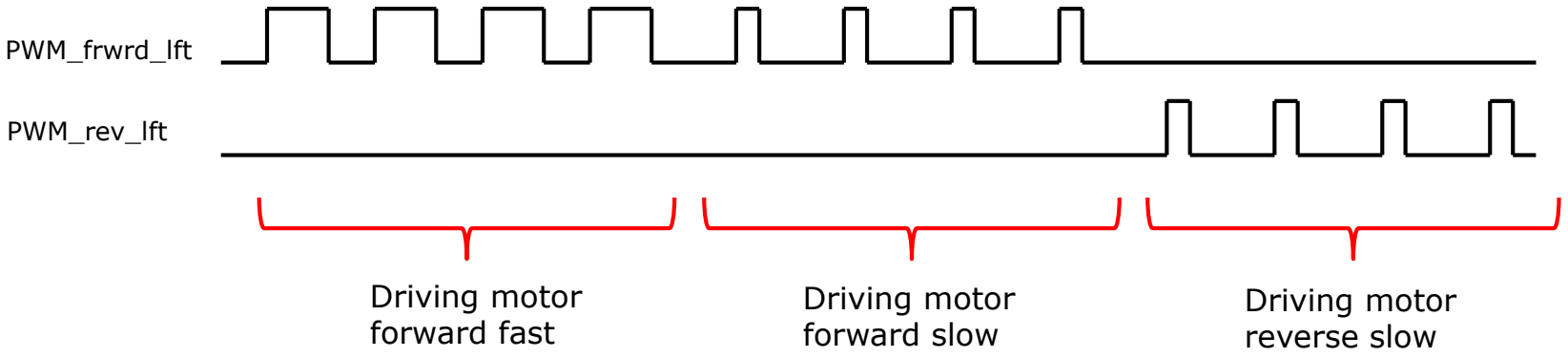
Finally since the motor controls work on sign/magnitude we need to take the absolute value and saturate to an 11-bit number to produce in input to our PWM units.

Motor Drive (mtr_drv)

- The digital core will produce a motor drive magnitude and direction for both the left and right motors.
- The magnitude of drive will be an 11-bit unsigned number.



- The DC motors are driven through an H-bridge. An H-bridge uses a PWM signal and can drive current in either direction through a load proportional to the duty cycle of the incoming PWM signal. The example waveforms below illustrate how the H-bridge driver chip we are using works:



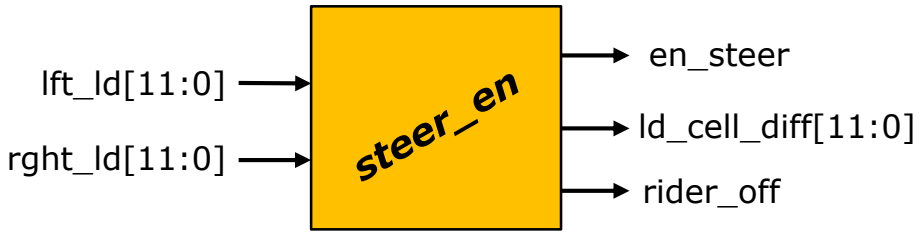
mtr_drv interface

Signal:	Dir:	Description:
clk, rst_n	in	50MHz clock, and active low asynch reset
lft_spd[10:0]	in	Left motor duty cycle
lft_rev	in	If high left motor should be driven in reverse
PWM_rev_lft PWM_frwrd_lft	in	11-bit PWM signal (2048 divisions/period) Go to H-Bridge controller chip
rght_spd[10:0]	in	Left motor duty cycle
rght_rev	in	If high left motor should be driven in reverse
PWM_rev_rght PWM_frwrd_rght	in	11-bit PWM signal (2048 divisions/period) Go to H-Bridge controller chip

- An 11-bit PWM module was done as part of an in class exercise.
- You are just using two copies of that (one for left and one for right) along with some simple ANDing logic to make this unit.

Enabling Steering Input (steer_en) (this block is inside what is shown as the digital core)

- When powered the "Segway" device is always in balance mode, but steering via the load cells is not immediately enabled.
- You have to step onto the platform one foot at a time. If steering was enabled immediately the device would be very hard to mount.
- The **steer_en** block looks at the left and right load cell values and only enables steering once both load cells have exceeded a threshold and have been within 25% of each other for 1.3+ seconds.
- Steering is kept enabled until one of the load cells is less than 6.25% of the other (or the total weight dips below a threshold). At this point it is assumed the rider is stepping off the platform (or fell off).
- This block needs to calculate the difference between left and right load cells. The **balance_cntrl** block uses this difference to affect steering. So this block will forward the difference to **balance_cntrl** so it is not computed twice.



- The **rider_off** output will be discussed later.

Steering Enable (continued)

- Immediately after reset the **steer_en** block will be looking for:

(**lft_ld** + **rght_ld**)>=MIN_RIDER_WEIGHT; // where MIN_RIDER_WEIGHT is a localparam

- Once minimum weight has been exceeded the next criteria that has to be satisfied is:

abs(**lft_ld** – **rght_ld**)<(**lft_ld** + **rght_ld**)/4

- This criteria has to be met for a full 26-bit count (1.34 sec). If at anytime during the 1.34 seconds the absolute value of the difference exceeds 25% of the total weight then the 1.34 sec period should restart.

- Once steering is enabled it will stay enabled until:

abs(**lft_ld** – **rght_ld**)>15*(**lft_ld** + **rght_ld**)/16 OR until:

(**lft_ld** + **rght_ld**)<MIN_RIDER_WEIGHT;

- Make an effort to explicitly share arithmetic blocks (make intermediate signals)

- Of course you have a state machine here as well *(you did a bubble diagram in HW1)*

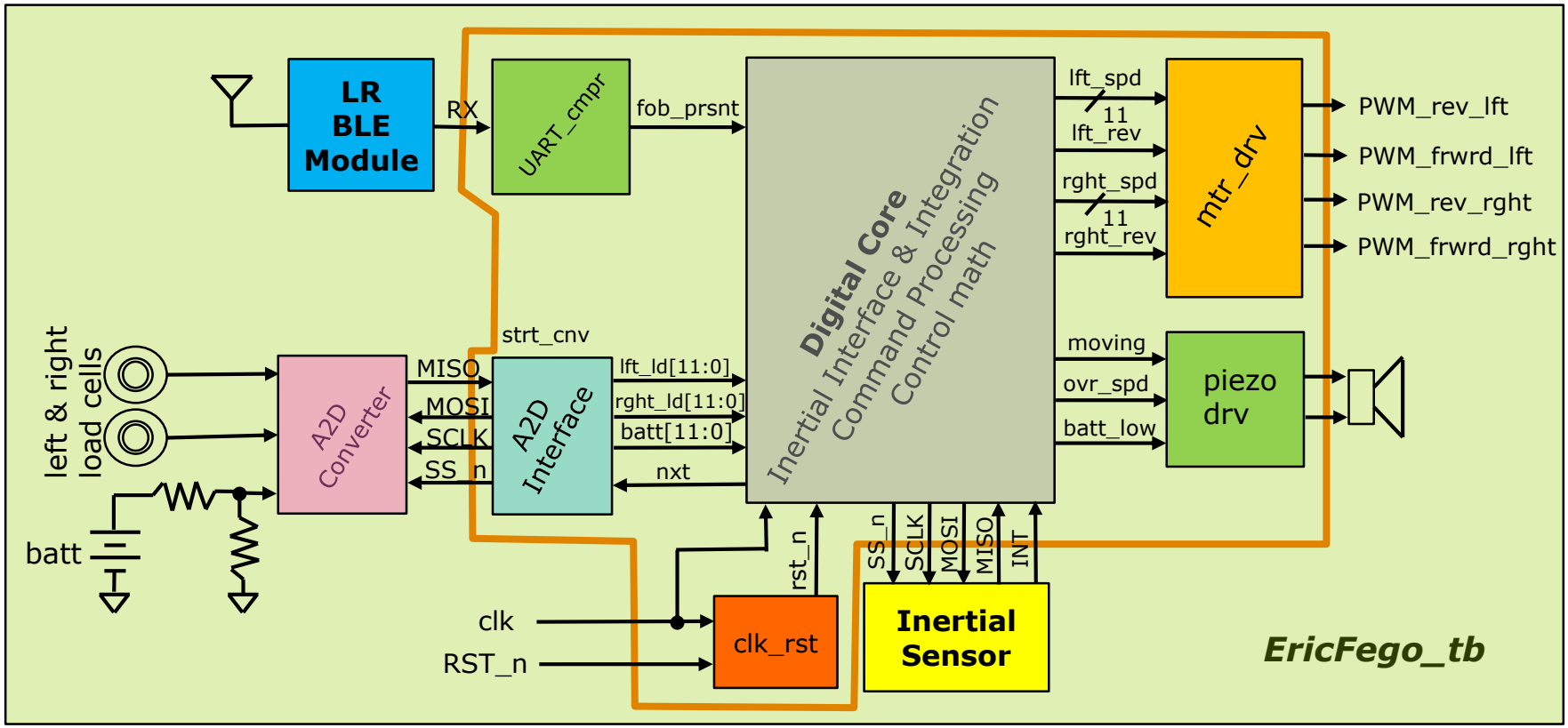
Steering Enable (continued)

- The output **en_steer** should be asserted whenever the state machine determines we are in normal mode of operation with steering enabled. This signal will go to **balance_cntrl** block and the piezo buzzer control.

- The signal **rider_off** should be asserted whenever the rider weight does not exceed the minimum threshold (i.e. when (lft_ld + rght_ld)<MIN_RIDER_WEIGHT).
  - There is a possibility the rider falls off the Segway all at once. Meaning there are two ways the state machine can exit normal mode. If the load cell difference exceeds 15/16 of the sum, or if all of a sudden the load cell sum is less than MIN_RIDER_WEIGHT (rider_off). Your normal mode case needs to check for both.
  - If the difference exceeded 15/16 of the sum you enter a state where steering is disabled and you are waiting for the rider to completely step off.
  - If they fell off all at once then you transition from normal mode to the initial state (waiting for sum to exceed MIN_RIDER_WEIGHT)

- **rider_off** goes to the balance controller to clear the integral term of the PID loop. (Imagine you were riding along on your Segway and the integral term was at a high value. Then you got "clotheslined" off your Seqway. You would want your device to stop, not keep going and run away from you).

Required Hierarchy & Interface



Your design will be placed in an "EricFego" testbench to validate its functionality. It must have a block called **Segway.v** which is top level of what will be the synthesized DUT.
The interface of **Segway.v must match exactly** to our specified **Segway.v** interface
Please download Segway.v (interface skeleton) from the class webpage.

The hierarchy/partitioning of your design below Segway is up to your team.
The hierarchy of your testbench above Segway is up to your team.

Segway Interface

Signal Name:	Dir:	Description:
clk	in	50MHz clock
RST_n	in	Unsynchronized reset (from push button) goes through ***clk_rst_smpl*** block to get synchronized and form **rst_n** (global reset to all other blocks).
INERT_SS_n	out	Active low slave select to inertial sensor
INERT_SCLK	out	SCLK of SPI interface to inertial sensor
INERT_MOSI	out	Master Out Slave In to inertial sensor
INERT_MISO	in	Master In Slave Out from intertial sensor
INT	in	INT signal from intertial sensor (rising edge indicates new data ready)
RX	in	UART input from Bluetooth link
PWM_frwrd_lft, PWM_rev_lft, PWM_frwrd_rght, PWM_ref_lft	out	Motor speed PWM signals left/right motors
A2D_SS_n	out	Active low slave select to A2D
A2D_SCLK	out	SCLK of SPI interface to inertial sensor
A2D_MOSI	out	Master Out Slave In to A2D
A2D_MISO	in	Master In Slave Out from A2D

Continued next page

Segway Interface (continued)

Signal Name:	Dir:	Description:
piezo, piezo_n	out	To piezo buzzer. Used to indicate to rider that battery is low, or they are going to fast.

Provided Modules & Files: (available on website under: Project as Collateral.zip)

File Name:	Description:
Segway_tb.v	**Optional** testbench template file.
Segway.v	**Requried** interface skeleton verilog file. **Copy this** you can change internals if you wish, but toplevel interface signals must remain as is.
SegwayModel.sv	Model of Inertial sensor and overall physics of the Segway. You need this to make a "complete" testbench.
ADC128S.v	Model of A2D on DE0-Nano
SPI_ADC128S.sv	SPI slave model used in ADC128S model

Synthesis:

- You have to be able to synthesize your design at the Segway level of hierarchy.
- You should have a synthesis script. It will be reviewed.

- Your synthesis script should write out a gate level netlist of Segway (**Segway.vg**).
- You should be able to demonstrate at least one of your tests running on this post synthesis netlist successfully.
- Timing (400MHz for clk) is pretty easy to make. Your main objective is to minimize area.

My design was about 5500 square microns

Synthesis Constraints:

Contraint:	Value:
Clock frequency	400MHz for clk (2.5ns period)
Input delay	0.25ns after clk for all inputs
Output delay	0.5ns prior to next clk rise for all outputs
Drive strength of inputs	Equivalent to a ND2D2BWP gate from our library
Output load	0.1pF on all outputs
Wireload model	TSMC32K_Lowk_Conservativve
Max transition time	0.10ns
Clock uncertainty	0.15ns

NOTE: Area should be taken after all hierarchy in the design has been smashed.