ECE 551
Project Spec

Fall ‘25

“Segway” like device

Except it is made of wood and old bicycle wheels.

Grading Criteria: (Project is 22% of final grade)

- Project Grading Criteria:
  - Quantitative Element 15%

  (yes this could result in extra credit)

$$Quantitative = \frac{Our\_Project Area}{Your Synthesized Area}$$

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

  - Project Demo (85%)
    - ✓ Code Review (12.5%)
    - ✓ Testbench Method/Completeness (15%)
    - ✓ Synthesis Script review (7.5%)
    - ✓ Post-synthesis Test run results (10%)
    - ✓ Results when placed in our Testbench (22.5%)
    - ✓ Test of your code mapped to the actual "Segway" and tested. (10%)
    - ✓ Teammates judgement of your contribution (7.5%)

My design was about XXXXX square microns

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 coverage number and get another 1% 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 another 1% extra credit.

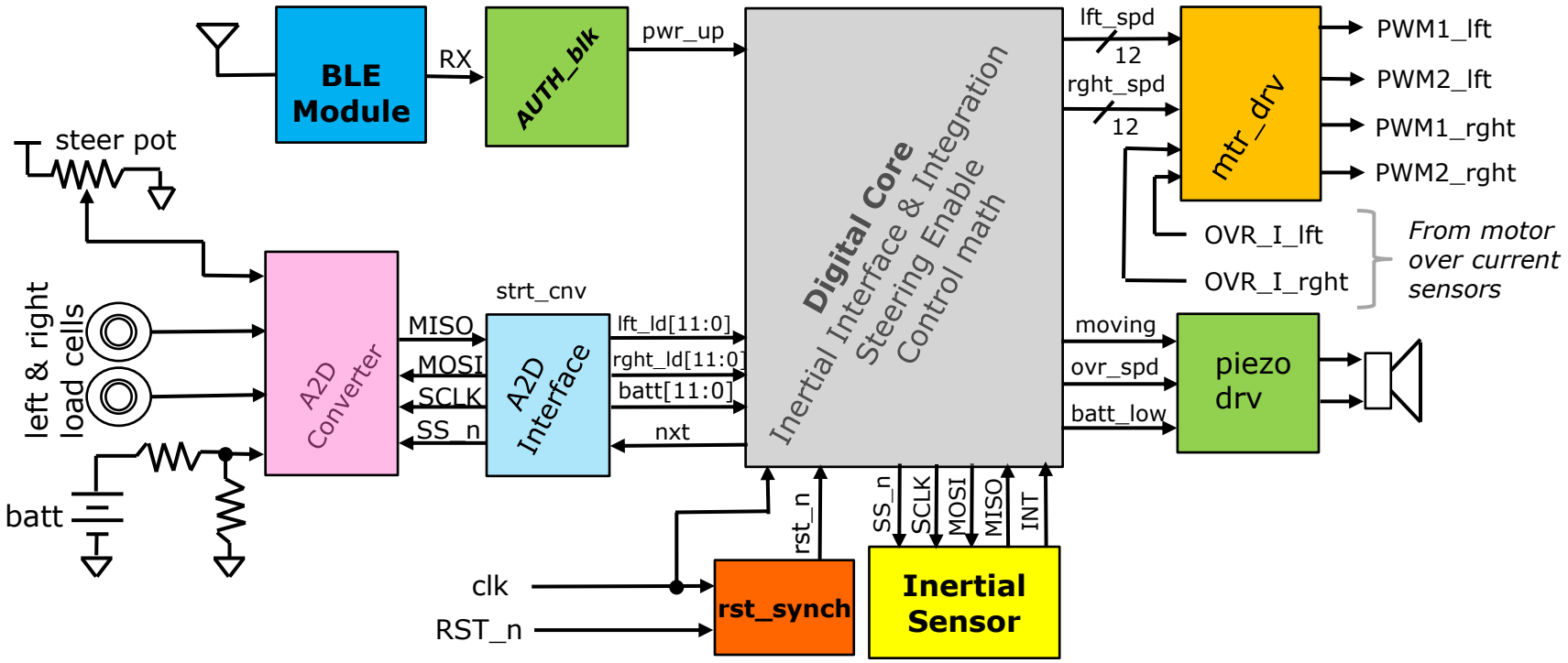
- Also some extra credit for submitting early

Project Due Date

- Project Demos will be held in B555:
  - Fri (12/05/25) from 3:00PM till evening. (2.5% bonus)
  - Sun (12/7/25) from 3:00PM till evening. (1.75% bonus)
  - Mon (12/8/25) from 1:45PM till evening. (1.25% bonus)
  - Tues (12/9/25) from 1:00PM till evening. (0.75 bonus)
  - Weds (12/10/25) from 1:45PM till evening. (due date)
  - Thurs (12/11/25) from 10:45PM till early afternoon. (1% penalty)

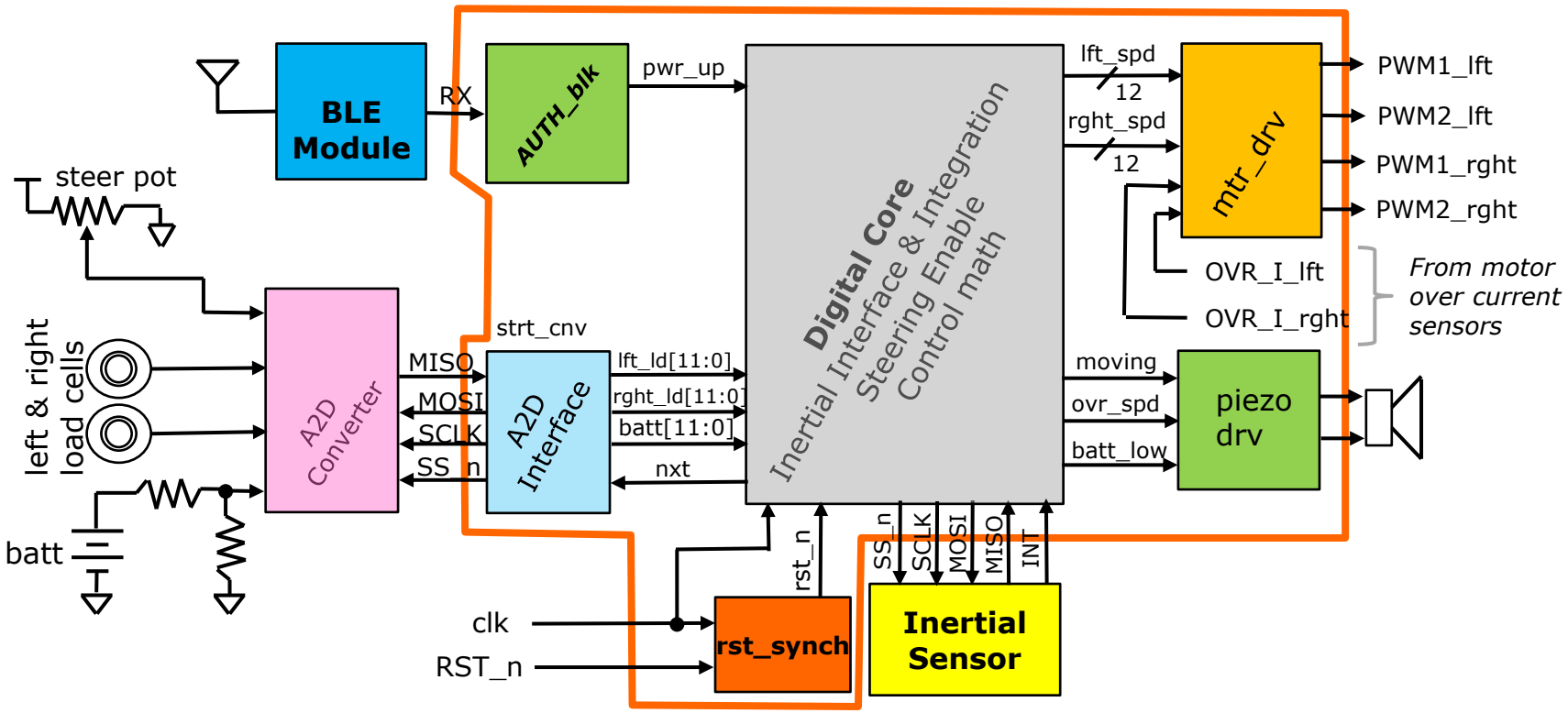
- Project Demo Involves:
  - ✓ Code Review
  - ✓ Testbench Method/Completeness
  - ✓ Synthesis Script & Results review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in our testbench
  - ✓ Results when tested on demo platform (“Segway” like device)

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 wheels independently forward or reverse at various speeds. An inertial sensor is used to obtain the pitch of the platform. If the 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 presence and side to side balance of the rider. A slide potentiometer on the handle of the device is used to effect 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.

You Must have a block called **Segway.sv** 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 0x47 ('G') over its UART TX line. This will in turn cause the **pwr_up** signal to be asserted

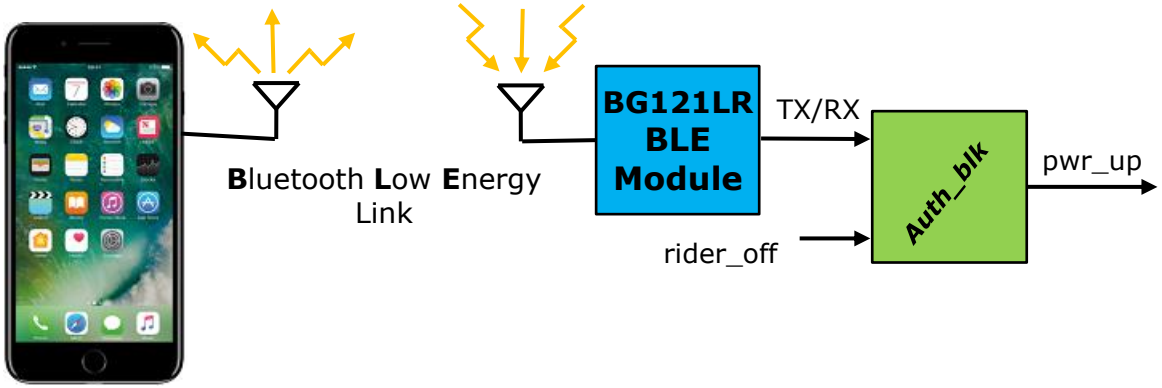
- When the phone app deliberately disconnects, or is disconnected due to range the BLE121LR module sends out 0x53 ('S') over its UART TX line . The "Segway then shuts down (if the weight on the platform no longer exceeds MIN_RIDER_WEIGHT (**rider_off** signal from ***en_steer*** block).

- The UART will send at 19200 using 8N1 variant.

MID_RIDER_WEIGHT = 12'h200;

- 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.

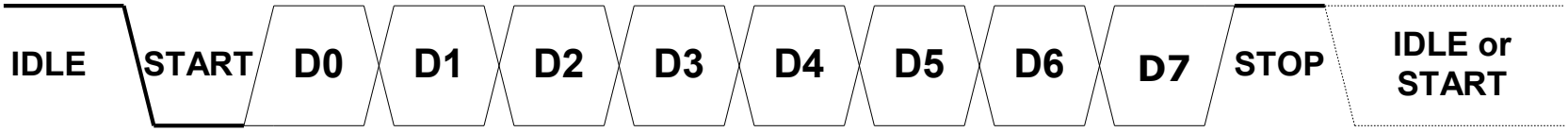


What is UART (RS-232)

- 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

$\frac{1}{Baud}$

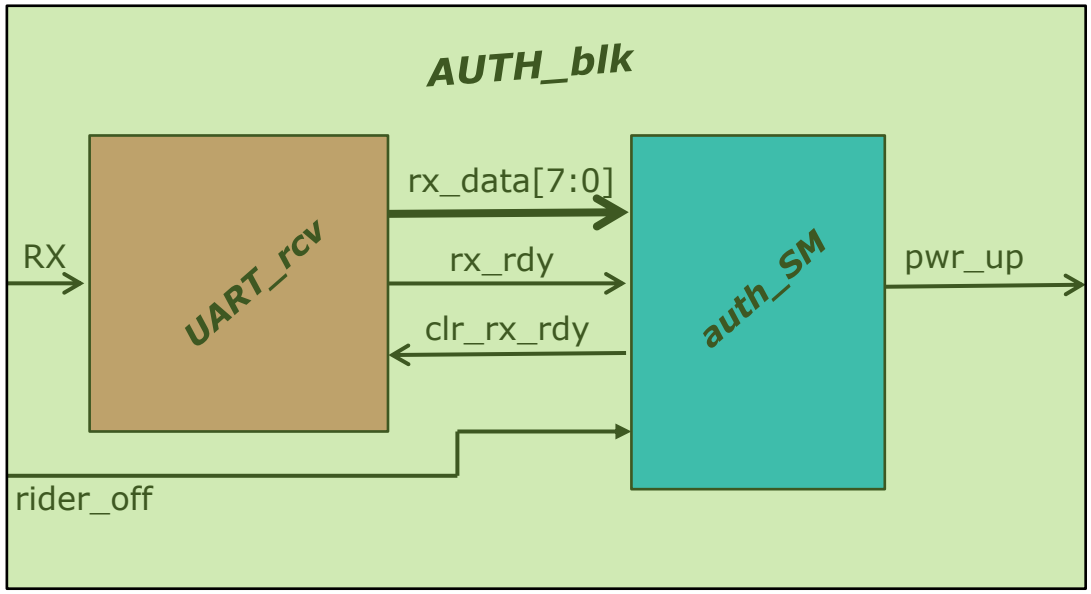
Baud rate will be 19200 with 50MHz clock



- 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 0x47, 0x53, and monitoring the **rider_off** signal.

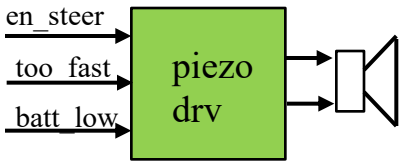


- **pwr_up** is asserted upon reception of 'G' (0x47). It is deasserted after the last reception was 'S' and the rider_off signal is high.

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

piezo_drv

- 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
- The tune for **en_steer** *(normal rider on and riding case)* should occur once every 3 seconds and will be *charge fanfare*
- Tone for **too_fast** should be the first 3 notes of charge fanfare played continuously. **too_fast** tone has priority over all others as it is indicating a dangerous condition.
- If **batt_low** is asserted *(and it is not **too_fast**)* then *charge fanfare* should be played backwards once every 3 seconds, regardless the state of **en_steer**.
- If no input signals are asserted the piezo should be silent.

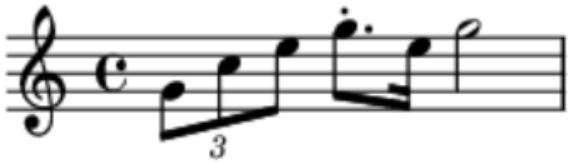
piezo_drv (Charge! Fanfare)

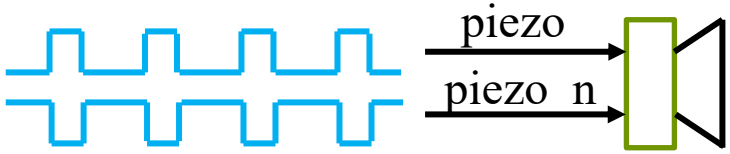
Charge Fanfare as frequencies & durations:

Note:	Freq:	Duration:
G6 (G in octave 6)	1568	2^{23} clocks
C7 (C in octave 7)	2093	2^{23} clocks
E7	2637	2^{23} clocks
G7	3136	$2^{23} + 2^{22}$ clocks
E7	2637	2^{22} clocks
G7	3136	2^{25} clocks

piezo_drv.sv Interface:

Signal:	Dir:	Description:
clk,rst_n	in	50MHz clk
en_steer	in	"normal" operation
too_fast	in	priority over other inputs
batt_low	in	*Charge* backwards
piezo, piezo_n	out	Differential piezo drive





A piezo bender is a "speaker" that can be driven with the GPIO's of our FPGA. We simply drive with a square wave of the frequency we want to generate a tone for.

The duty cycle is not so important (anything from 20% to 80% will do). We will drive differentially for increased sound amplitude.

The interface of the block should be as shown in the table.

The implementation will require a couple of counter/timers *(frequency & duration)* and a controlling SM.

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

piezo_drv *(parameter **fast_sim**)*

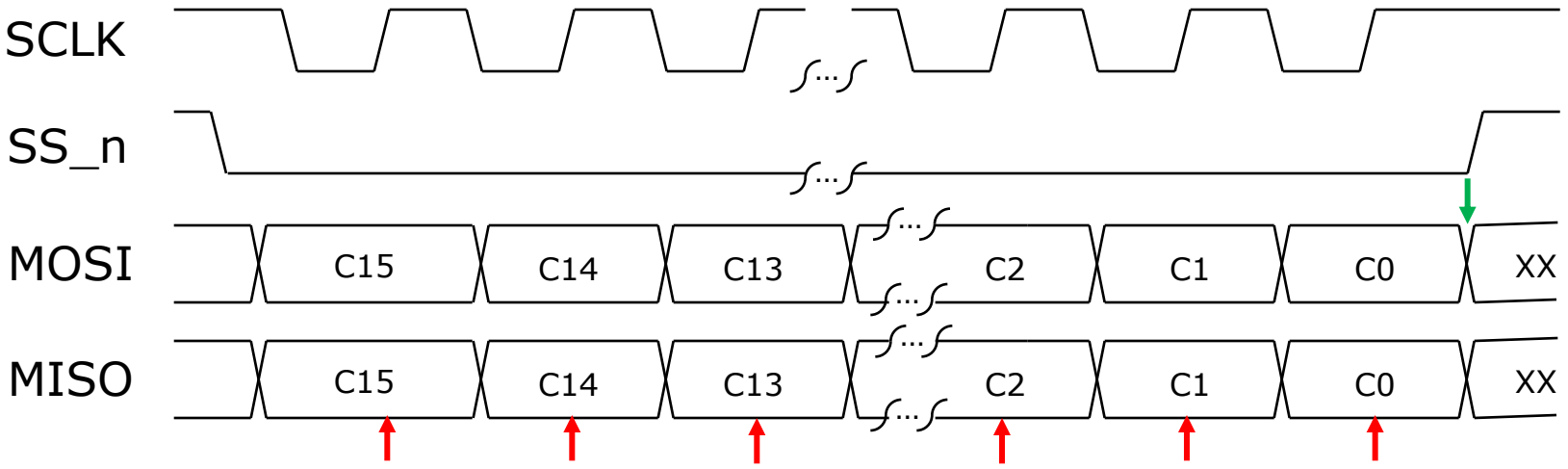
- The duration of the notes (2^{23} clock cycles) is quite long for simulation purposes *(especially when we get to fullchip simulations where **piezo_drv.sv** will be one of many blocks being simulated).*

- We will need a method to speed up simulations. ***piezo_drv.sv*** should employ a parameter (called **fast_sim**). **fast_sim** should be defaulted true. When **fast_sim** is passed a 0 then durations and note frequencies should be as specified. When **fast_sim** is true then the duration of notes should be 1/64 their normal length, and all frequencies should be 64X their normal. Durations of the 3sec timer should also be 1/64 of normal. **HINT**: use a **generate if** statement to create an amount by which you increment your various counter2. Increment by 1 or by 64 depending on **fast_sim**.

- Create a simple testbench (***piezo_drv_tb.sv***) that simply instantiates the DUT, applies clock and reset and asserts the inputs in order. Note **piezo** and **piezo_n** should not be toggling before any of the inputs are asserted.

- Once your testbench is passing visual inspection of **piezo/piezo_n** move on to the next portion (testing with DE0).

SPI Packets



Shown above is a 16-bit SPI packet. The master is changing (shifting) **MOSI** on the falling edge of **SCLK**. The serf 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).

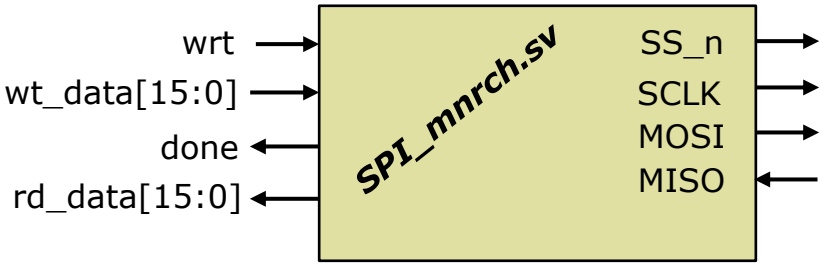
The sampled version of **MISO** in turn gets shifted into our 16-bit shift register. (on **SCLK** fall)

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 serf.

SPI Unit for Inertial Interface & A2D

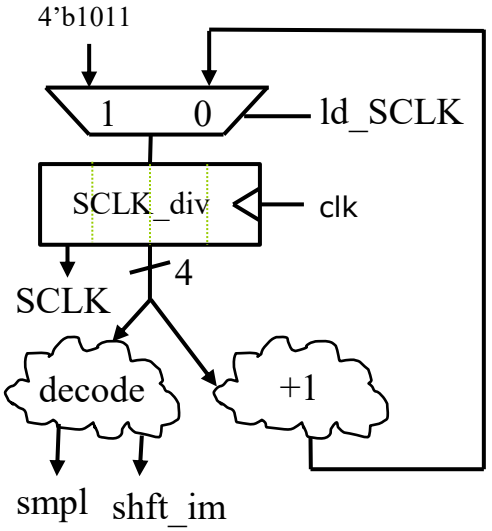
- You will implement **SPI_mnrch.sv** with the interface shown.
- SCLK frequency will be 1/16 of the 50MHz clock (i.e. it comes from the MSB of a 4-bit counter running off **clk**)
- I had better not see any ***always*** blocks triggered directly on **SCLK**. We only use **clk** when inferring flops.
- Remember you are producing **SCLK** from the MSB of a 4-bit counter. So for example, when that 4-bit counter equals 4'b0111 you know **SCLK** rise happens on the next **clk** *(sample MISO (see red arrow))*. Likewise when that 4-bit counter equals 4'b1111 you know SCLK fall happens next *(enable your shift register)*.



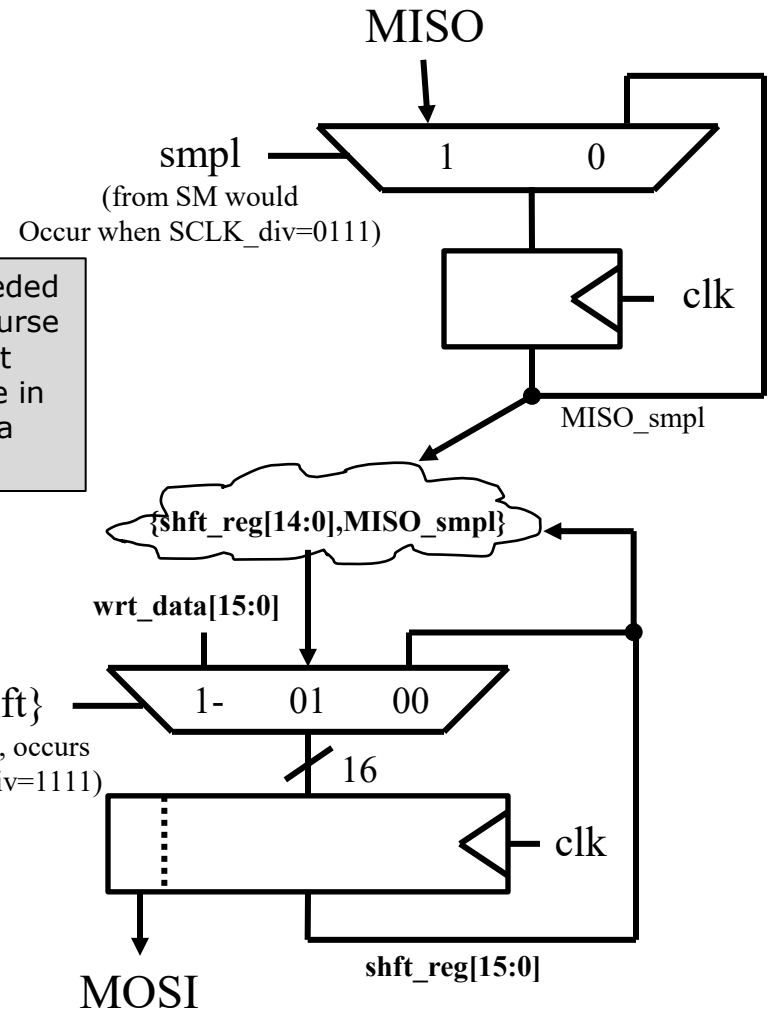
Signal:	Dir:	Description:
clk, rst_n	in	50MHz system clock and reset
SS_n, SCLK, MOSI,MISO	3-out 1-in	SPI protocol signals outlined above
wrt	in	A high for 1 clock period would initiate a SPI transaction
wt_data[15:0]	in	Data (command) being sent to inertial sensor.
done	out	Asserted when SPI transaction is complete. Should stay asserted till next **wrt**
rd_data[15:0]	out	Data from SPI serf. For inertial sensor we will only ever use [7:0]

SPI Implementation

"Heart of you SPI monarch is a 16-bit shift register. The MSB of this shift register forms MOSI. A version of MISO sampled at "SCLK rise" is shifted in as the LSB. The register is shifted during a SPI transaction at "SCLK fall".



This is a picture of the needed datapath elements. Of course you are going to need a bit counter too to know where in the 16-bits you are. Plus a SM to control all this.



shft_im => shift imminent *(about to occur)*

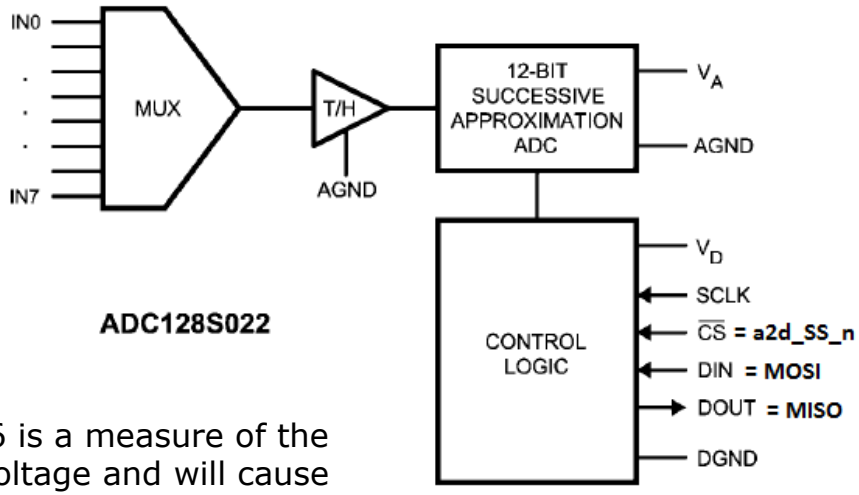
A synchronous reset of **SCLK_div** to a value like 1011 can help with the creation of a "front porch" (see previous slide). You are the monarch, you generate **SCLK**, so you know exactly when rise/falls are going to occur. If **sclk_div** = 0111 then a rise of **SCLK** is going to occur on the next **clk** edge.

A2D Converter (National Semi ADC128S022)

The DE0 Nano board contains a 12-bit eight channel A2D converter. We will make use of 3 channels.

Channels 0 and 4 are used to measure the left and right load cells respectively. These readings are used to determine rider weight/presence and side to side balance.

Channel 5 reads a slide potentiometer mounted on the handle the the rider uses for steering



ADC128S022

ADC Channel:	IR Sensor:
000 = chnnl	Left load cell
100 = chnnl	Right load cell
101 = chnnl	Steering Potentiometer
110 = chnnl	Battery voltage

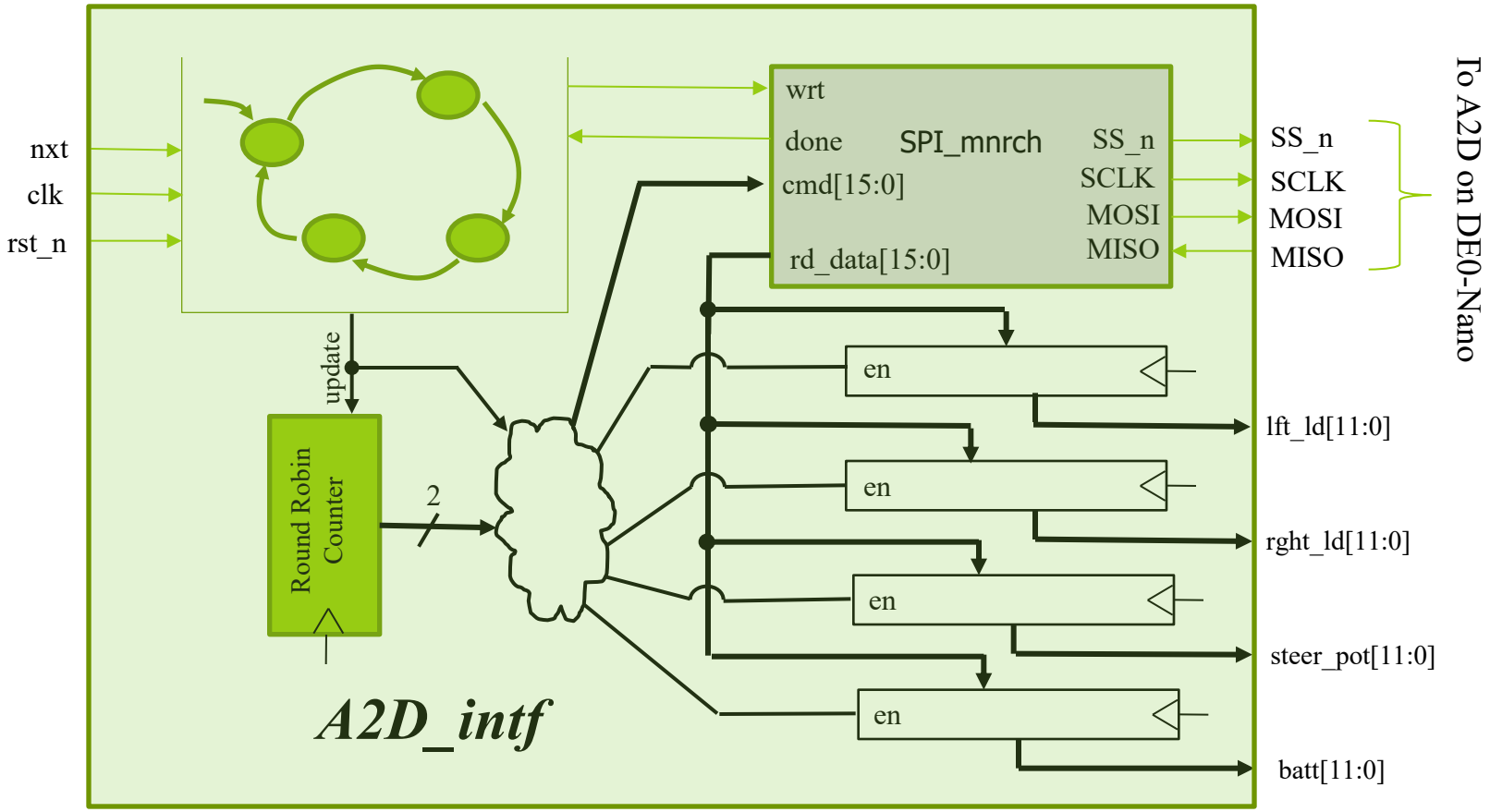
Channel 6 is a measure of the Battery voltage and will cause an audible tone change in piezo_drv if too low.

To read the A2D converter one sends the 16-bit packet {2'b00,chnnl,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 four channels. One reading will be requested every update from the inertial sensor.

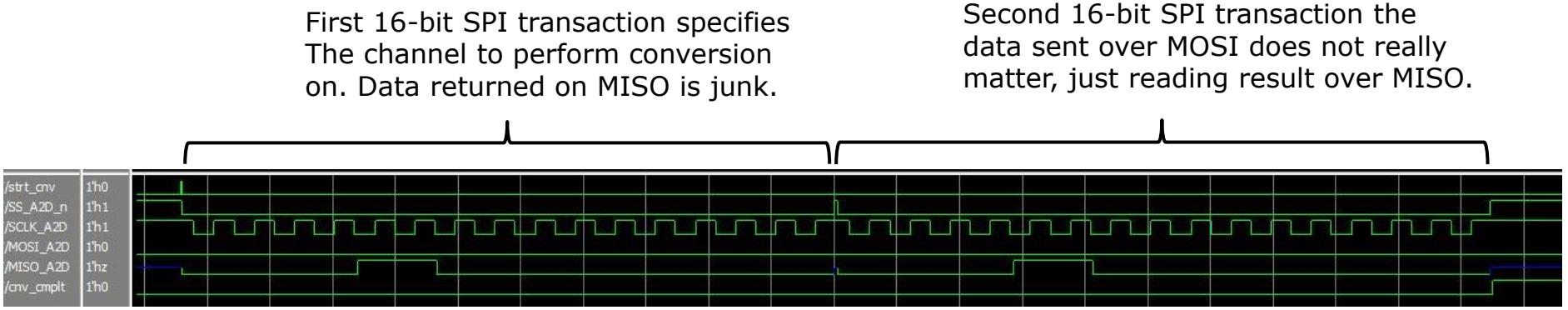
A2D Intf Design



The SM will sit idle till it is told to perform a conversion (**nxt** asserted). Then it will kick off two SPI transactions via **SPI_mnrch**. The first SPI transaction determines what channel to convert, and the second SPI transaction reads the result for that channel. The round robin counter is then incremented, and on the **nxt** request it will convert the next channel in the sequence.

You also need 4 holding registers to hold the respective results. The round robin counter determines where to store the results as well.

Exercise 21: A2D Intf Design and Test Bench



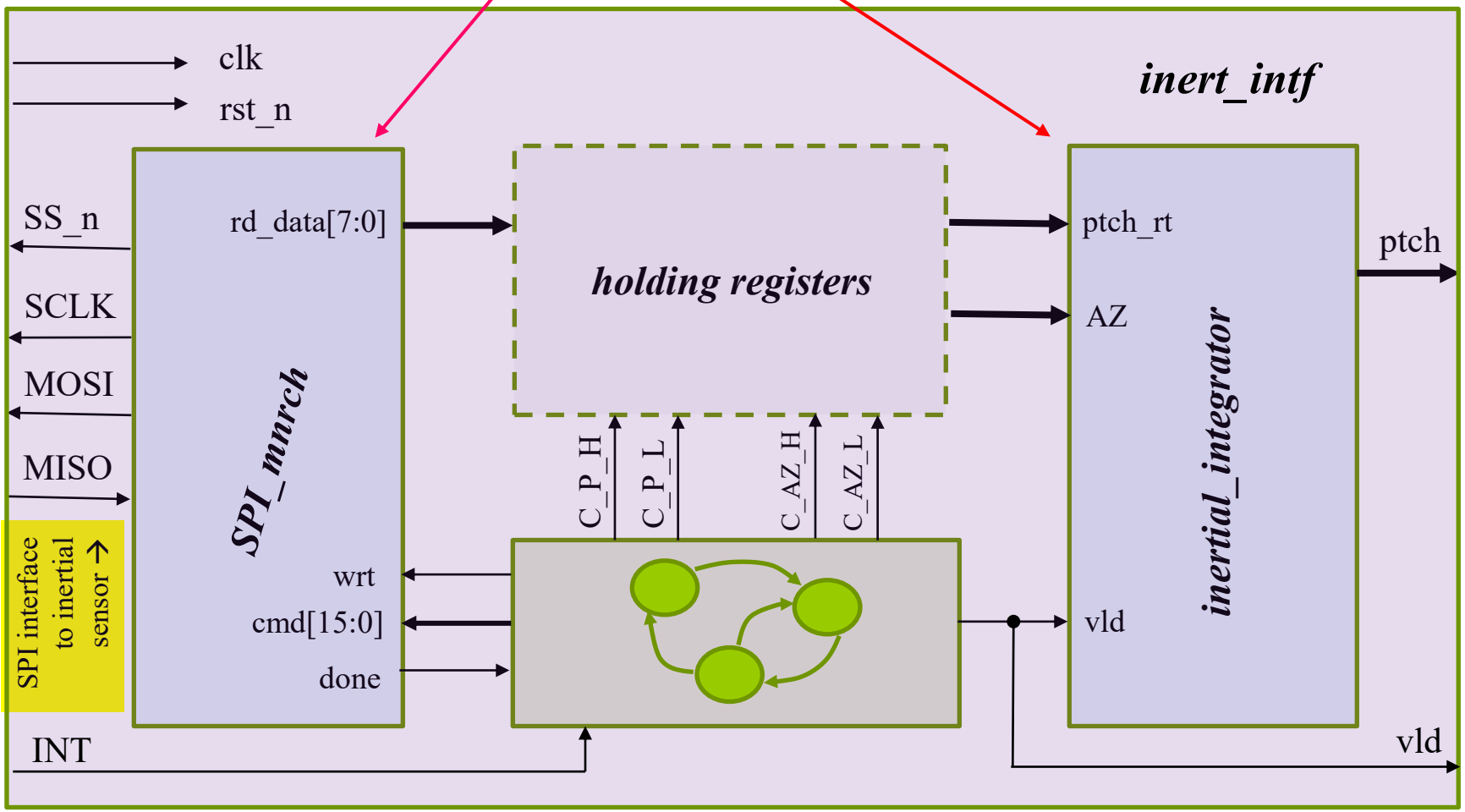
Our use of the A2D converter will involve two 16-bit SPI transactions nearly back to back (separated by 1 system clock cycle).

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 in our application).

For the next 16-bit transaction the data sent over MOSI to the A2D does not matter. 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 meaningful.

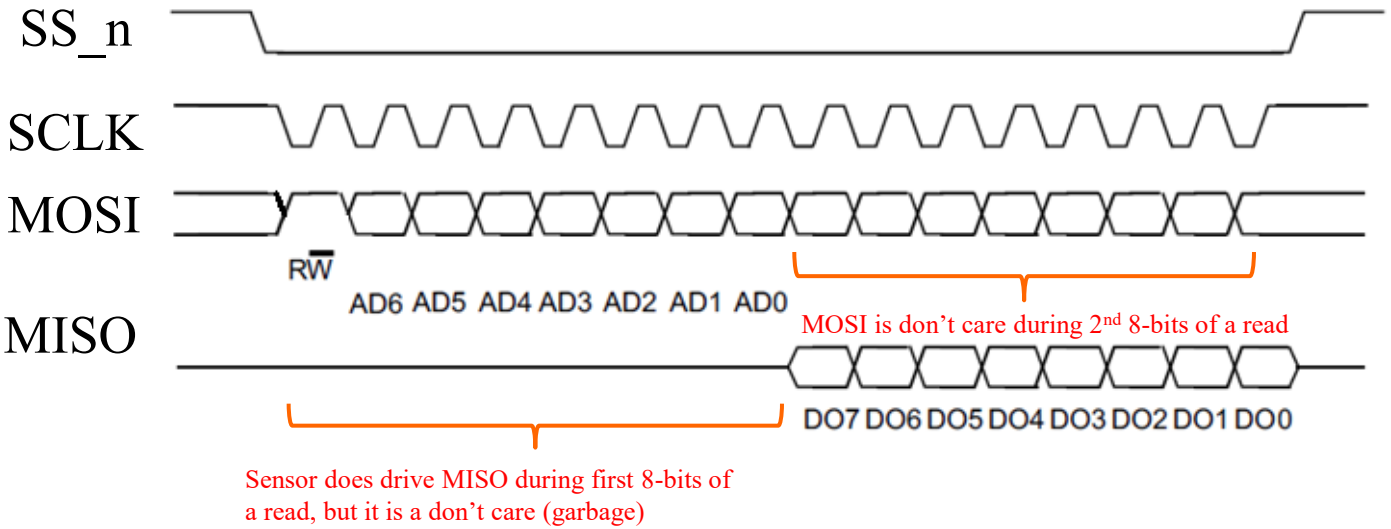
Inertial Sensor Interface

You provide ***SPI_mnrch*** & ***inertial_integrator***. **inert_intf.sv** will be provided.



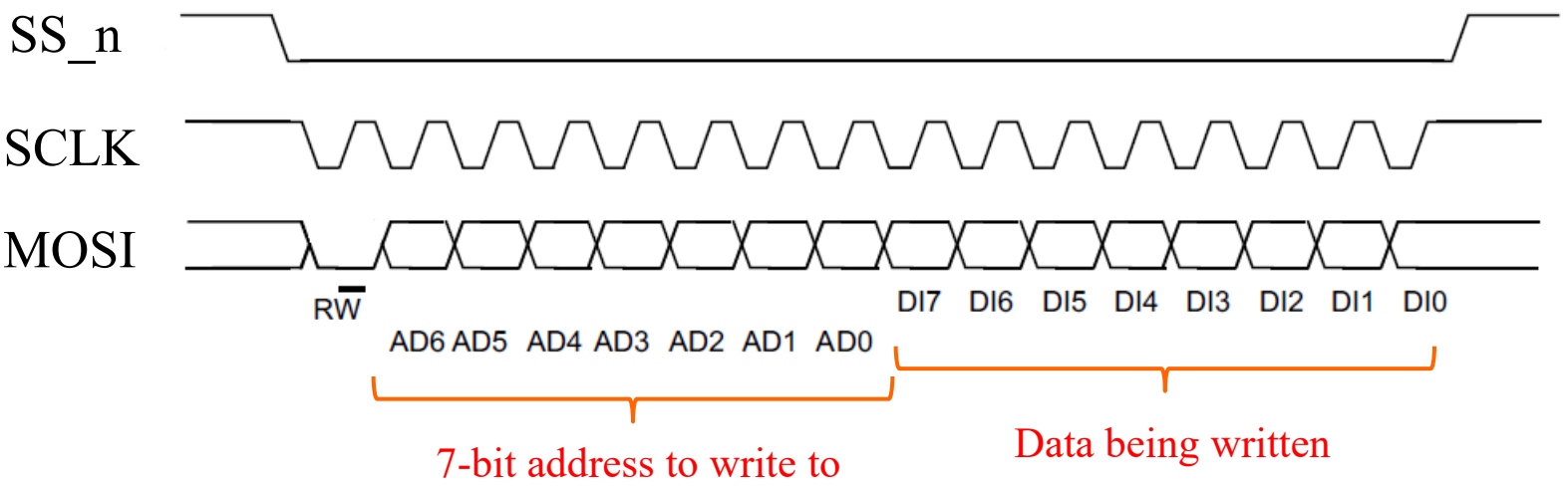
Inertial Sensor Interface

- One of the primary functions of the inertial interface is to drive the SPI interface to configure and perform reads of the inertial sensor.
  - 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

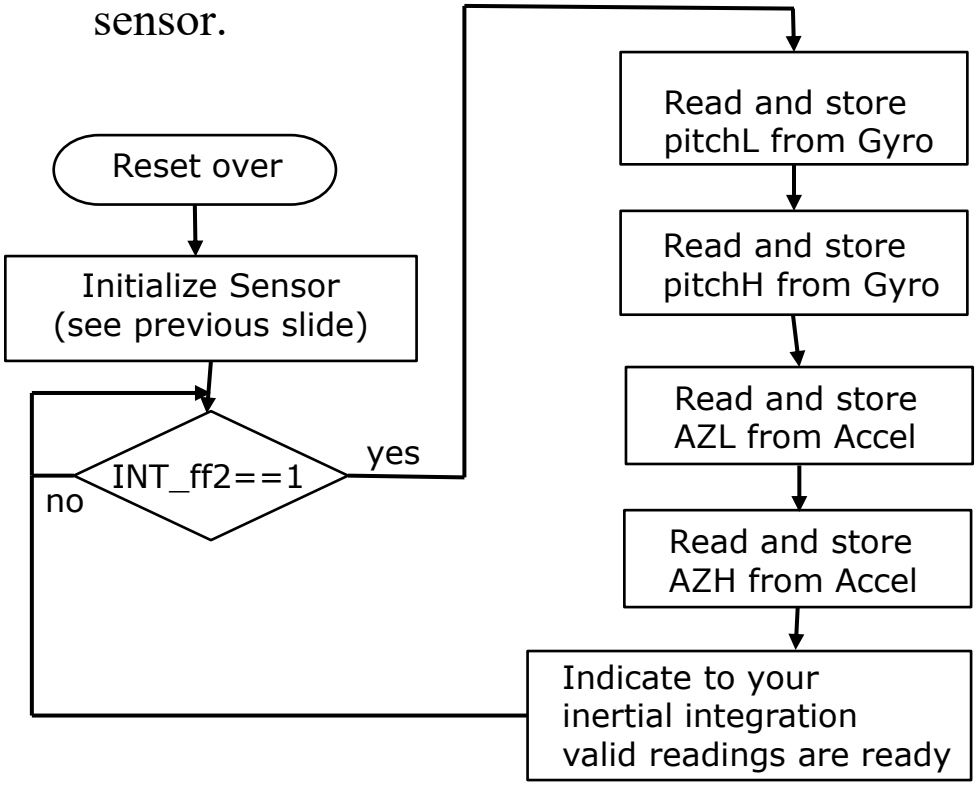
- After reset de-asserts the system must write to some registers to configure the inertial sensor to operate in the mode we wish. The table below specifies the writes to perform.

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 readings we obtain from the gyro are 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 gyro readings with accelerometer readings (AZ) to correct for this drift.

Inertial Integrator

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 (acceleration in Z direction)
ptch[15:0]	out	Fully compensated and "fused" 16-bit signed pitch.

inertial_integrator internal register:

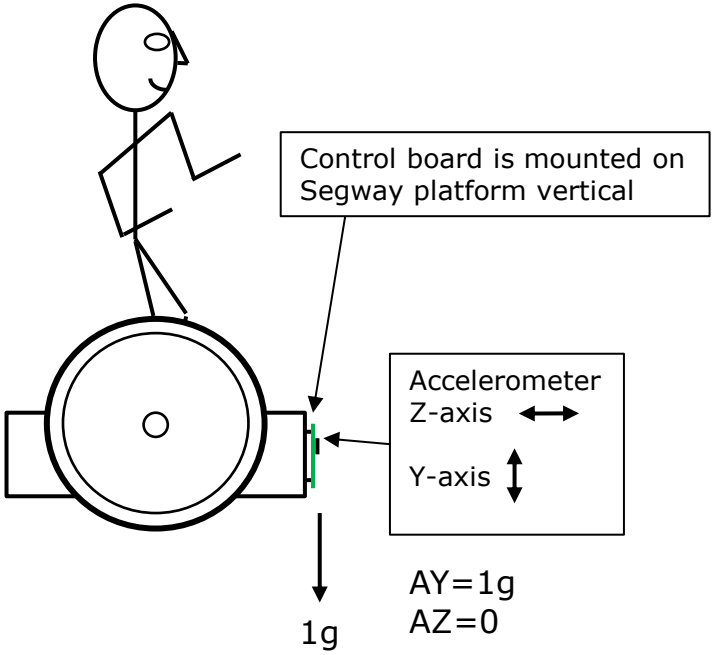
Register:	Purpose:
ptch_int[26:0]	Pitch integrating accumulator. This is the register ***ptch_rt*** is summed into

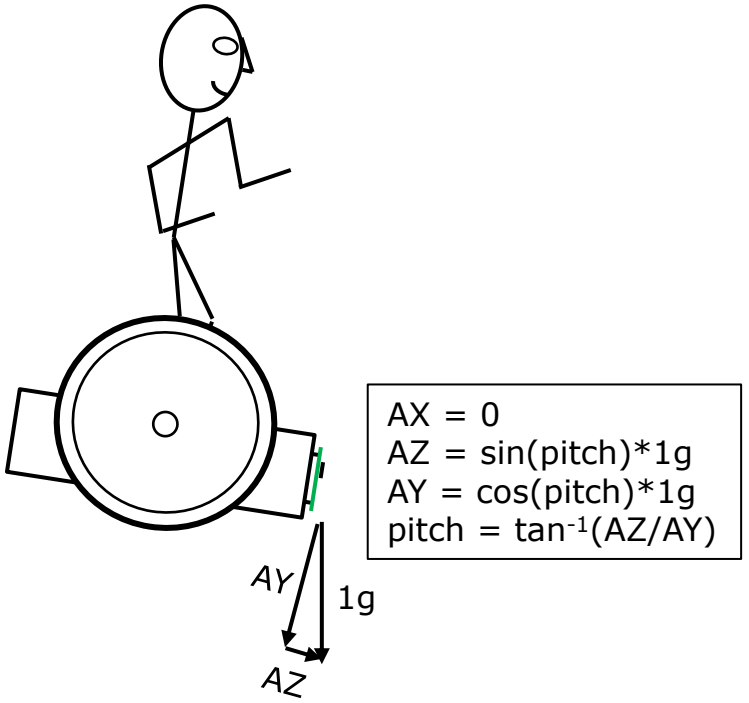
- On every **vld** pulse this unit will integrate **ptch_rt_comp** signal into **ptch_int[26:0]** (we integrate the **negative** of **ptch_rt_comp** due to orientation of sensor mount)
- Bits [26:11] of **ptch_int** form **ptch[15:0]** (i.e. divide by 2^11). This scaling factor was derived by trial and error with the actual Segway.
- 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'h0050)**

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.





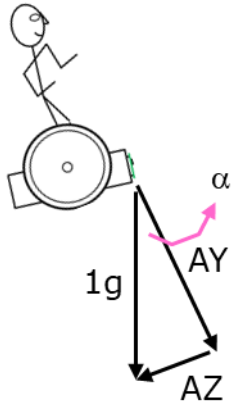
So why get pitch and roll from the gyro?
We can just use AY and AZ

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

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 = +1024;
  else
    fusion_ptch_offset = -1024;
```

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

AZ_comp = **AZ** - **AZ_OFFSET** where **AZ_OFFSET** is a localparam we will assign to 16'h00A0

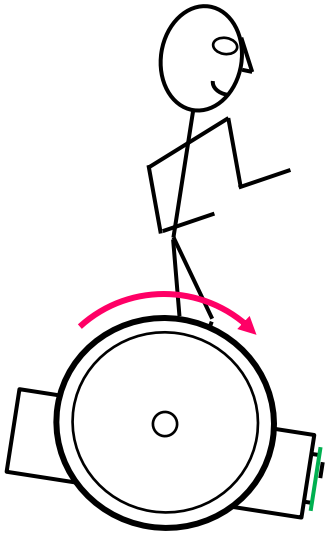
Fusion Calcs…litte more explicit

- 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.

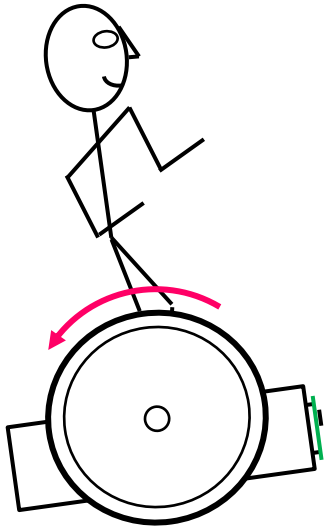
  - ```
    ptch_int <= ptch_int - {{11{ptch_rt_comp[15]}},ptch_rt_comp} +
                fusion_ptch_offset;
    ```

- During normal operation **fusion_ptch_offset** will be +1024 if **ptch_acc > ptch** and will be -1024 if **ptch_acc < ptch.**
    ```

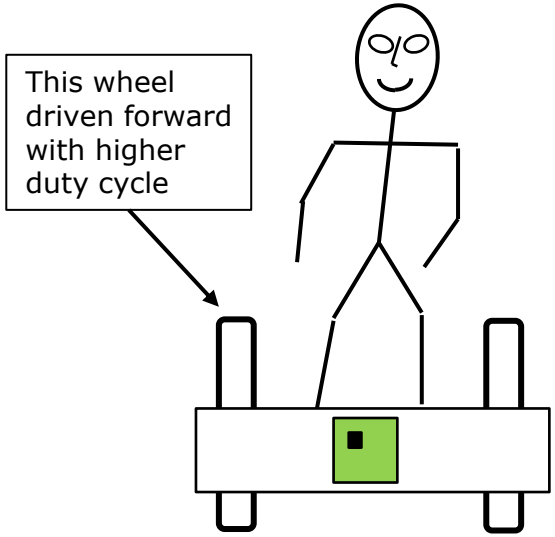
Balance & Steering Control



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

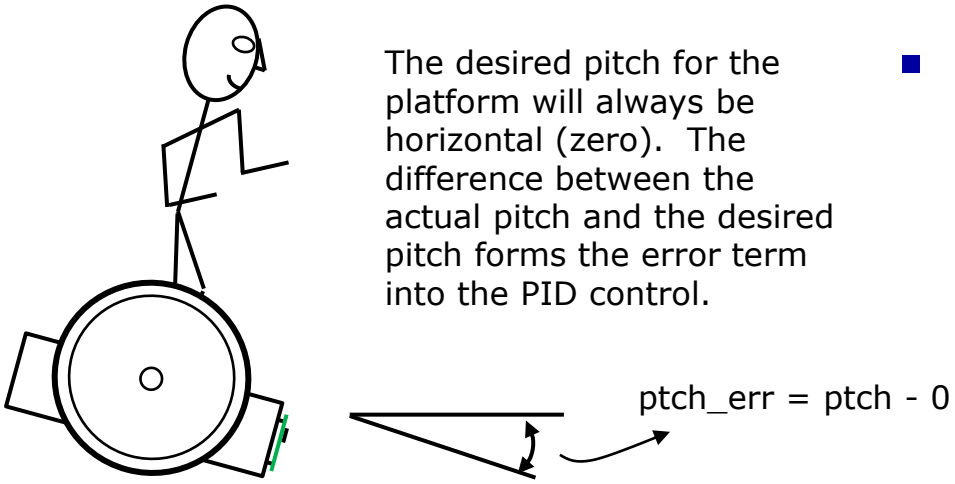


Rider leans backwards the control has to drive the wheels reverse to correct the pitch of the platform.



Load sensors in the floor of the platform will detect rider leaning to one side. If they are reasonably balanced then steering is enabled. Steering is affected by a slide potentiometer on the handle.

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**

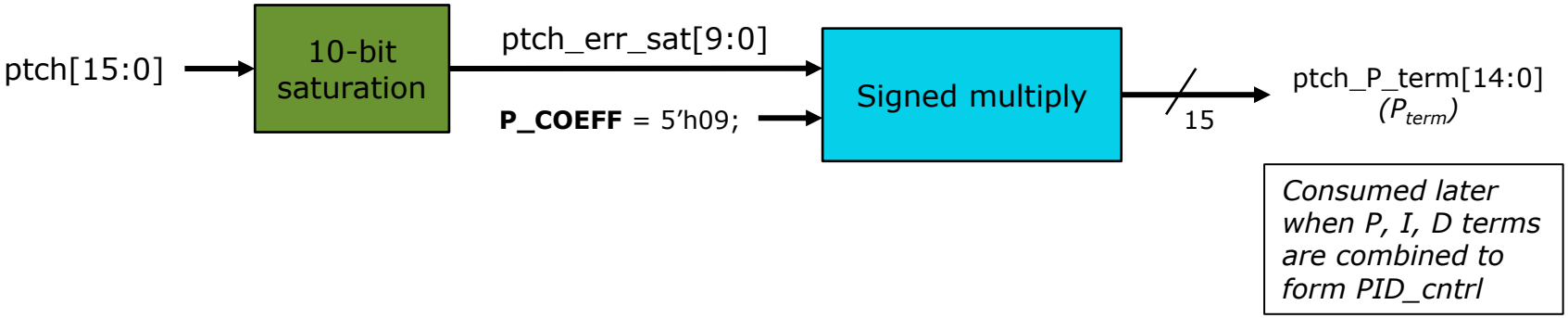
- Systems with very little momentum or lag in their response don't need a **D** term. Most control systems for power electronics will operate with just **PI** control since the power electronic circuit outputs respond quickly and with no "momentum" to changes in the control input.

- 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.

PID Math (more detail…P_{term})



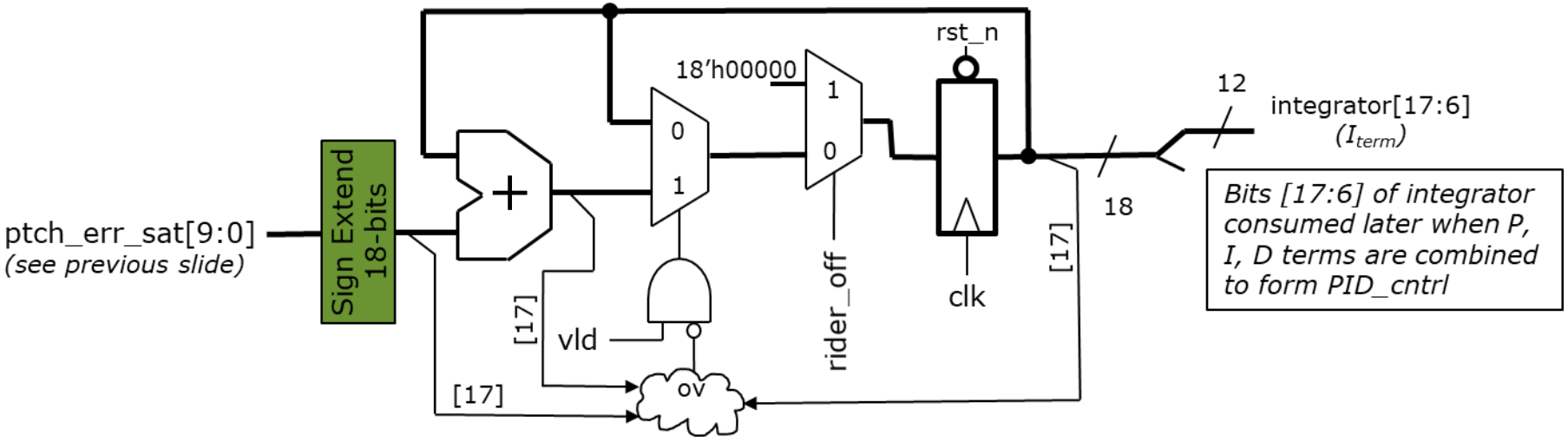
The P_{term} is the simplest of the terms to generate. All you need to do is saturate **ptch** into a 10-bit signed number and multiply it by P_COEFF. You need to ensure you infer a signed multiplier, which means both operands need to be signed ($**signed(P_COEFF)**) and the result it is assigned into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

The 18-bit integrator used to be an input. Now we need to form the integrator accumulator as an internal register.

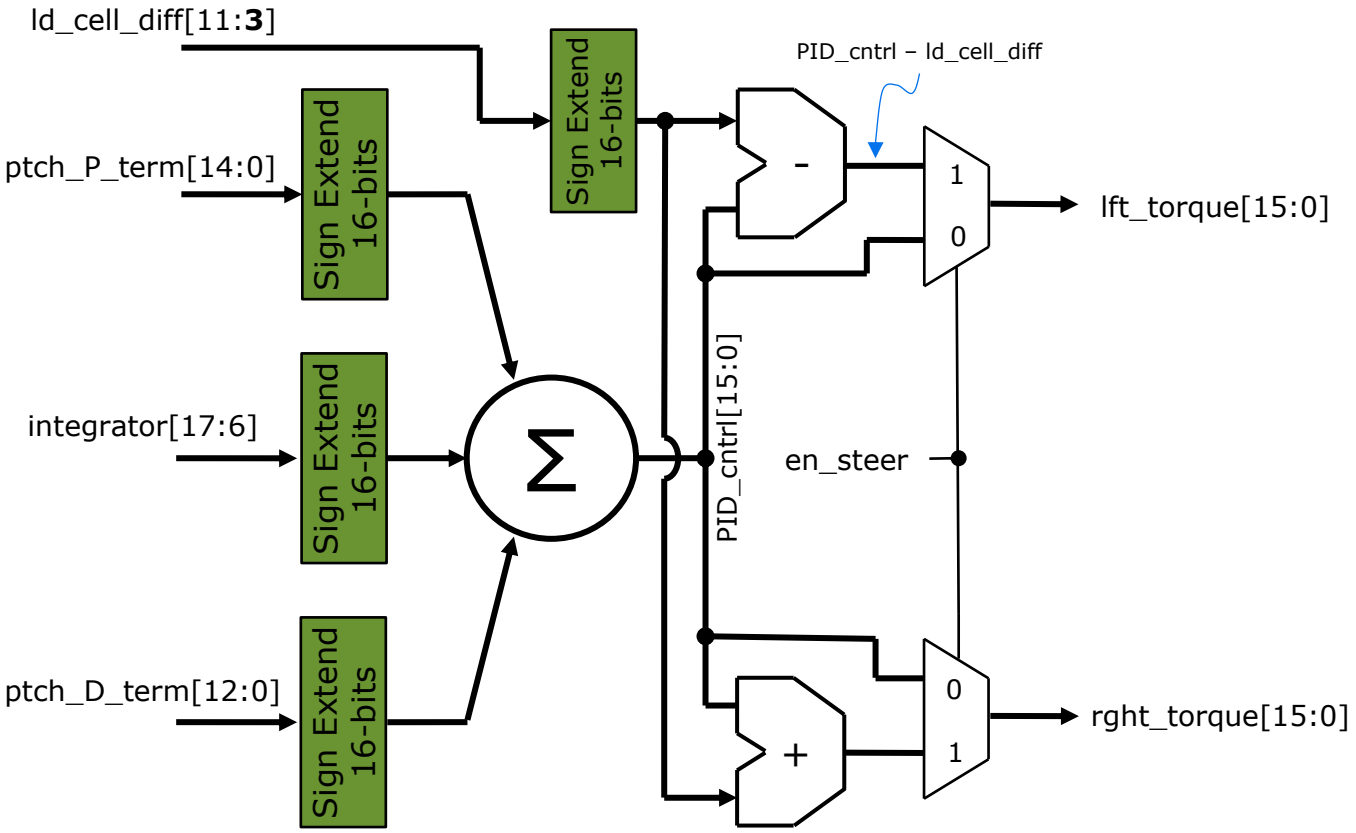


The primary function of the I_{term} is quite simple. On every **vld** reading from the inertial sensor the saturated version of pitch error (**ptch_err_sat**) is accumulated into an 18-bit accumulator register. We then use the upper bits ([17:6]) of this accumulator to form our I_{term} that summed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



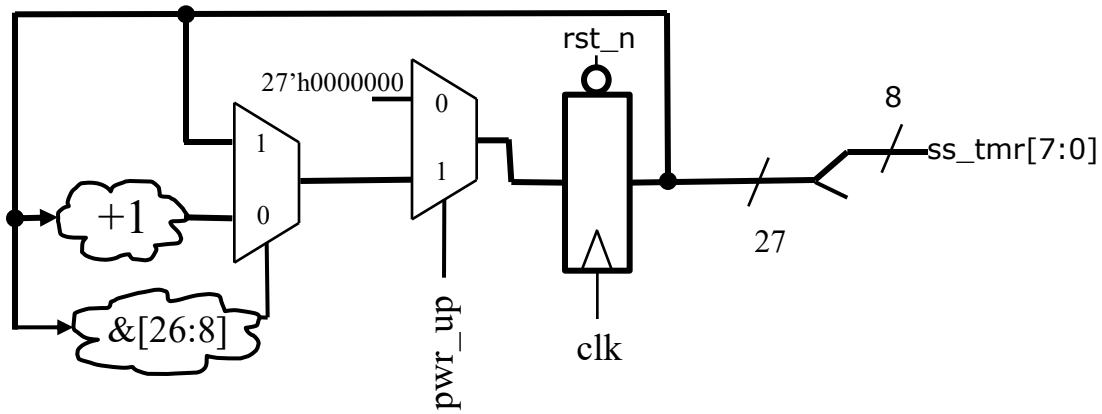
We sum the 3 terms together to form **PID_cntrl**[15:0]. Then if steering is enabled then 1/8 the difference between left and right load cells is subtracted/added to form a differential drive.

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.

PID (Soft Start Timer)

- The ***PID*** unit will also generate the soft start timer (**ss_tmr[7:0]**) that is used in ***SegwayMath*** to ensure on power up the Segway does not jerk to a start.
- We want this ramp of ss_tmr to be in the 2-3 second range. Which given our clock speed (50MHz) takes quite a wide timer.
- **ss_tmr[7:0]** will be formed from the upper 8-bits of a 27-bit timer.
- The FPGA of the Segway is powered and running initially, but it is waiting for a code from the ***auth_blk*** (authorization block) to actually enable the balancing feature and allow a rider to ride it. When this code comes in via a Bluetooth link the **pwr_up** signal will be asserted.
- The counter that forms **ss_tmr** should be held in a zero state until **pwr_up** is asserted.
- See the next slide for more detail on **ss_tmr** implementation.

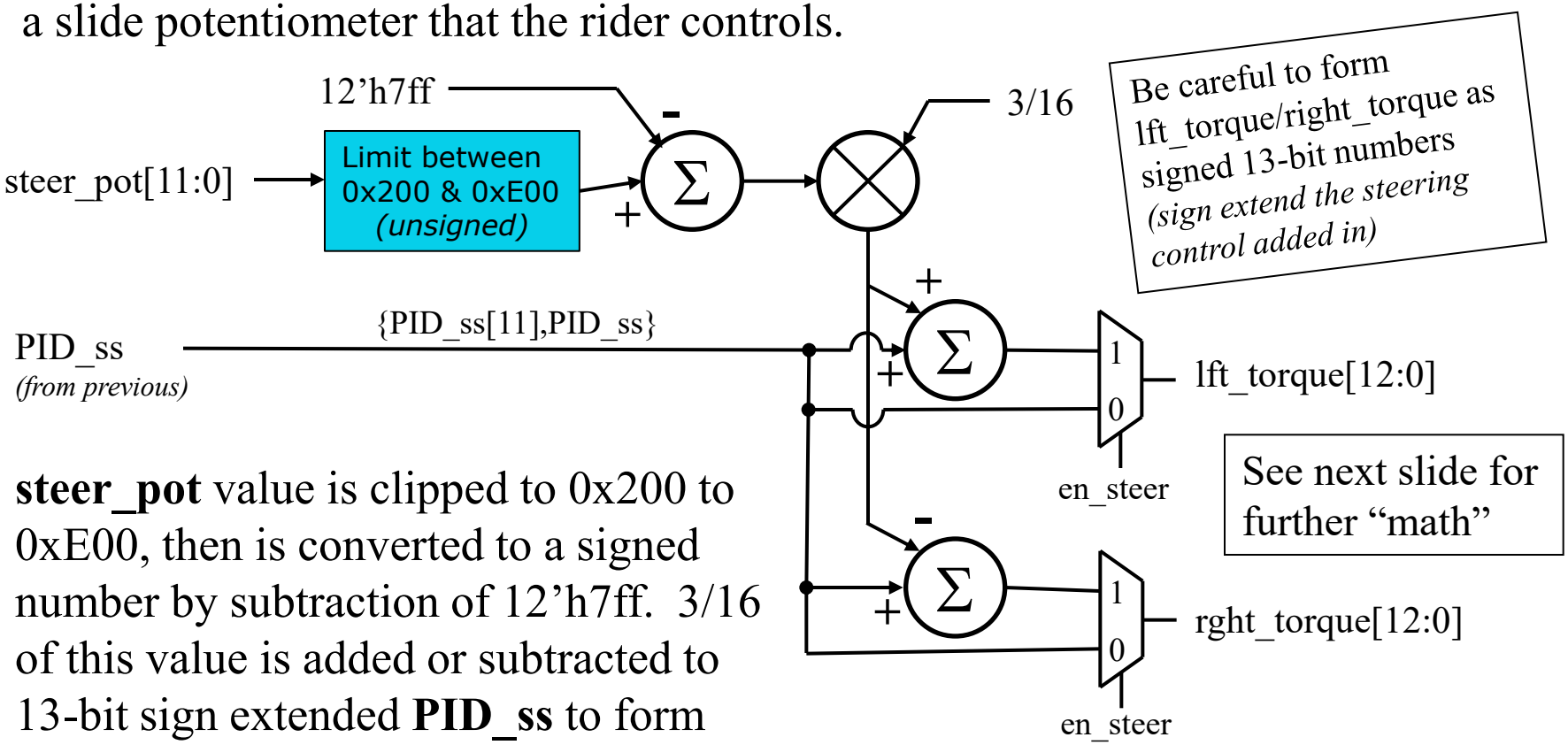
PID (Soft Start Timer)(continued)



- When **pwr_up** is deasserted the soft start timer should remain zero.
- It is a one shot timer. Once it starts counting and gets near full *(bits [26:8] set)* it freezes.

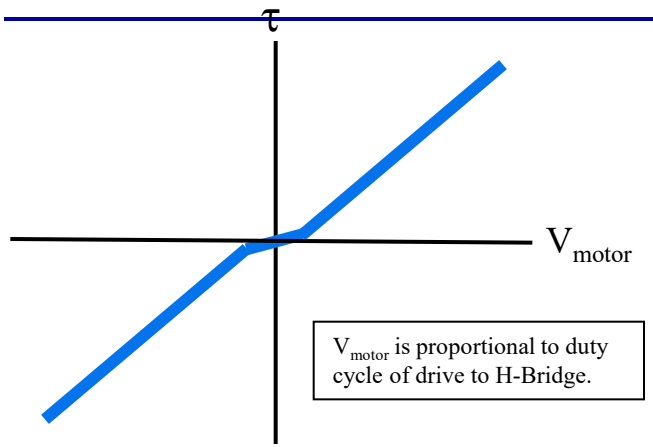
Exercise 8 Segway Math: (steering input)

The PID controller ensures the over all forward/reverse drive of the motors to keep the platform balanced. However, to effect steering a differential signal is added/subtracted to the left/right motors. This steering signal comes from a slide potentiometer that the rider controls.



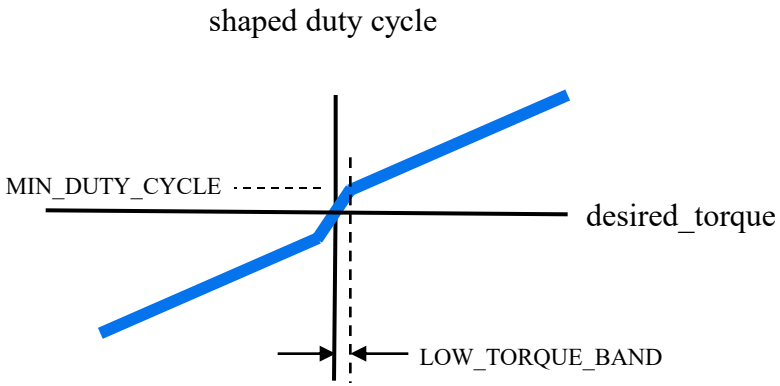
steer_pot value is clipped to 0x200 to 0xE00, then is converted to a signed number by subtraction of 12'h7ff. 3/16 of this value is added or subtracted to 13-bit sign extended **PID_ss** to form **lft_torque/rght_torque** if steering is enabled.

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 30V motors, but they cannot overcome their own internal friction from -1.4V to +1.4V. With a voltage magnitude that exceeds 1.4V the torque & speed increases linearly with voltage.

This deadband, though small, causes some havoc when small corrections are needed to keep the platform level. We need to compensate for it.

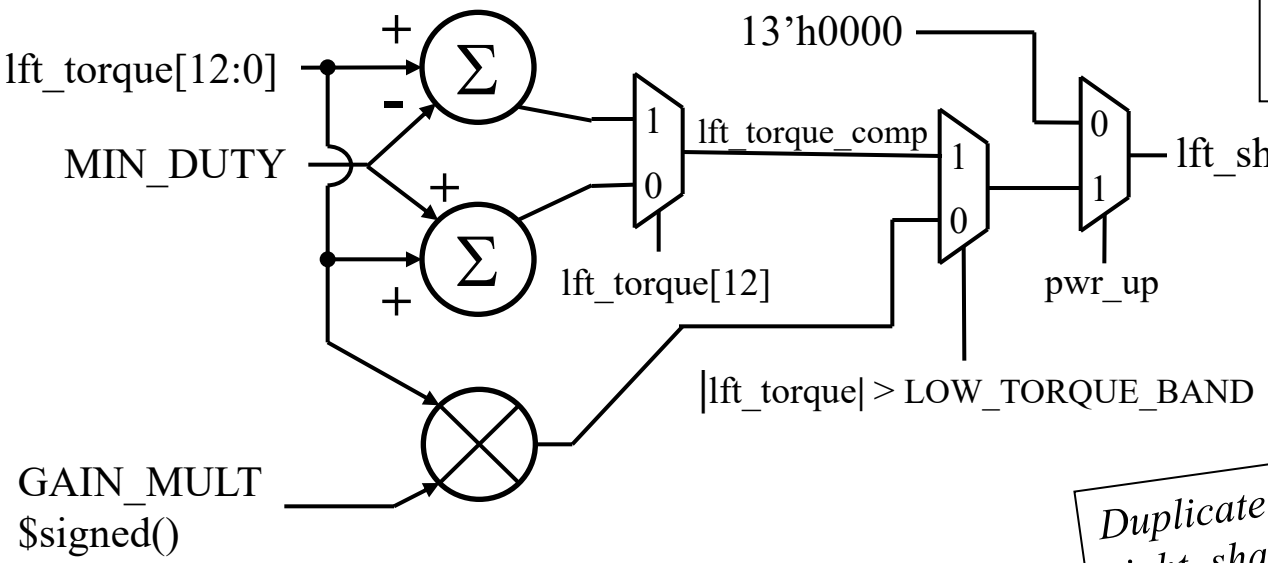


We will scale/shape our desired torque to compensate for the DC motors dead zone and compress it into a small range of desired torques: **(-LOW_TORQUE_BAND,LOW_TORQUE_BAND).**

When: **|desired_torque| < LOW_TORQUE_BAND** we are in the steep part of the compensation and will scale the **desired_torque** by **GAIN_MULTIPLIER** (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.

Exercise 8 Segway Math: (deadzone shaping)

localparams:
MIN_DUTY = 13'h0A8
LOW_TORQUE_BAND = 7'h2A
GAIN_MULT = 4'h4



See next slide for further "math"

Duplicate these calculations to form right_shaped from right_torque

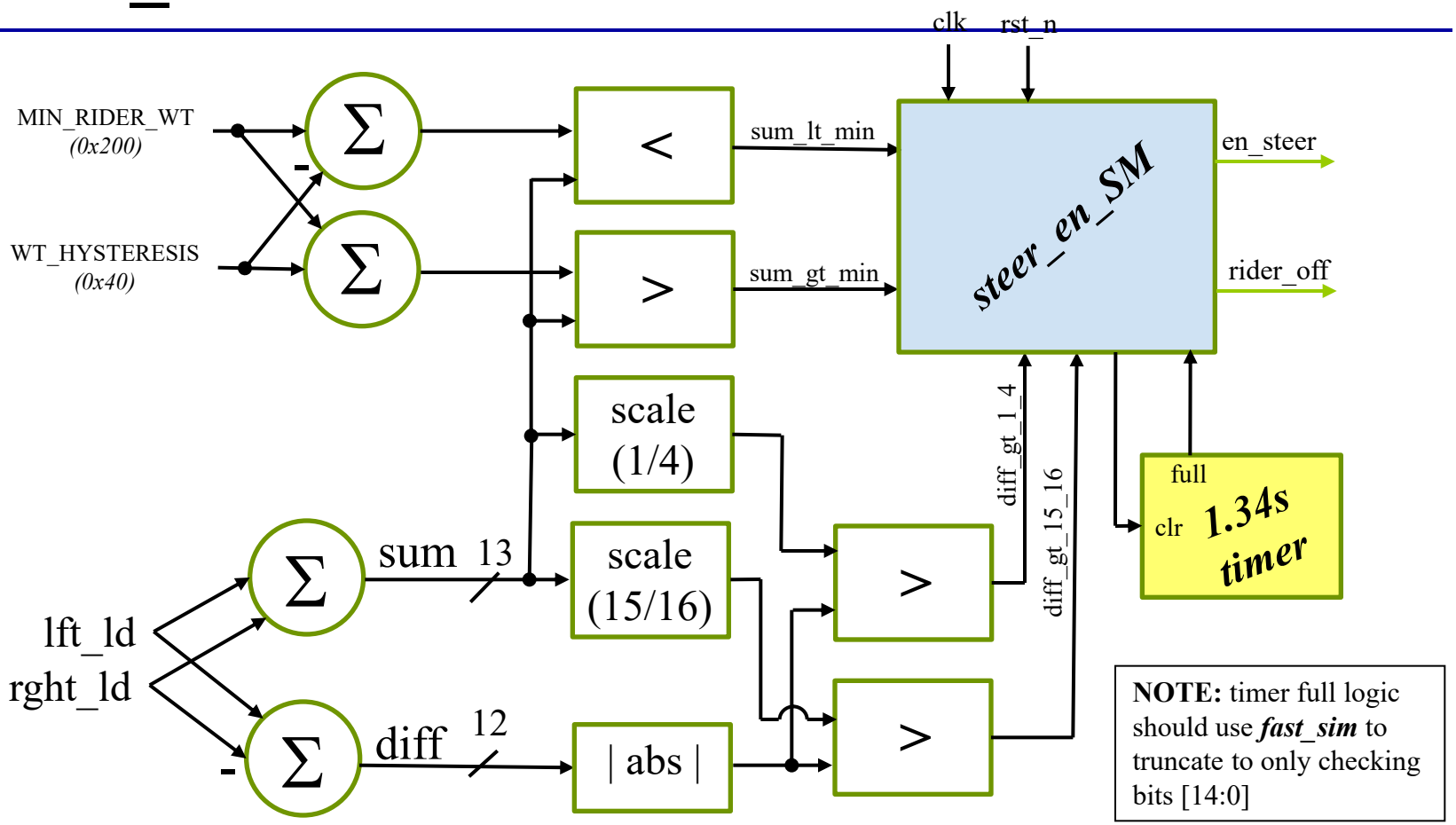
If **lft_toque** is negative (MSB set) then the compensated torque is: **lft_torque** - MIN_DUTY, otherwise it is **lft_torque** + MIN_DUTY. If the abs(**lft_torque**) < LOW_TORQUE_BAND *(i.e. we are inside the deadzone)* we need to use **lft_torque***$signed(GAIN_MULT) to form the shaped torque. Finaly if the Segway is not powered up we set the shaped torque to zero.

steer_en

- In HW3 you created a statemachine **steer_en_SM.sv**

- In this exercise you will wrap some supporting logic around it and test it.
  - You will need to generate the following signals:
    - sum_gt_min
    - sum_lt_min
    - diff_gt_1_4
    - diff_gt_15_16
  - You will need to infer the 1.34sec timer the SM uses
  - You will need to have a ***fast_sim*** parameter to speed up the timer by only looking at bits [14:0] when ***fast_sim*** is set.

steer_en



en_steer block diagram

Make it and test it. No need to turn anything in…you just need it.

mtr_drv

- Download **mtr_drv.sv** and study it.
- This code must not be changed.
- It serves 4 main functions.
    1. Synch **lft_spd/rght_spd** from ***balance_cntrl*** to the PWM cycle
    2. Convert signed **lft_spd/rght_spd** to magnitude to drive PWM.
    3. Monitor over current signals (**OVR_I_lft**/**OVR_I_right**) to perform shut down if over current occurring too frequently
    4. Distribute PWM signals to H-Bridges that drive the motors.\
- It uses two instantiations of your ***PWM11***.

- What your testing should accomplish:

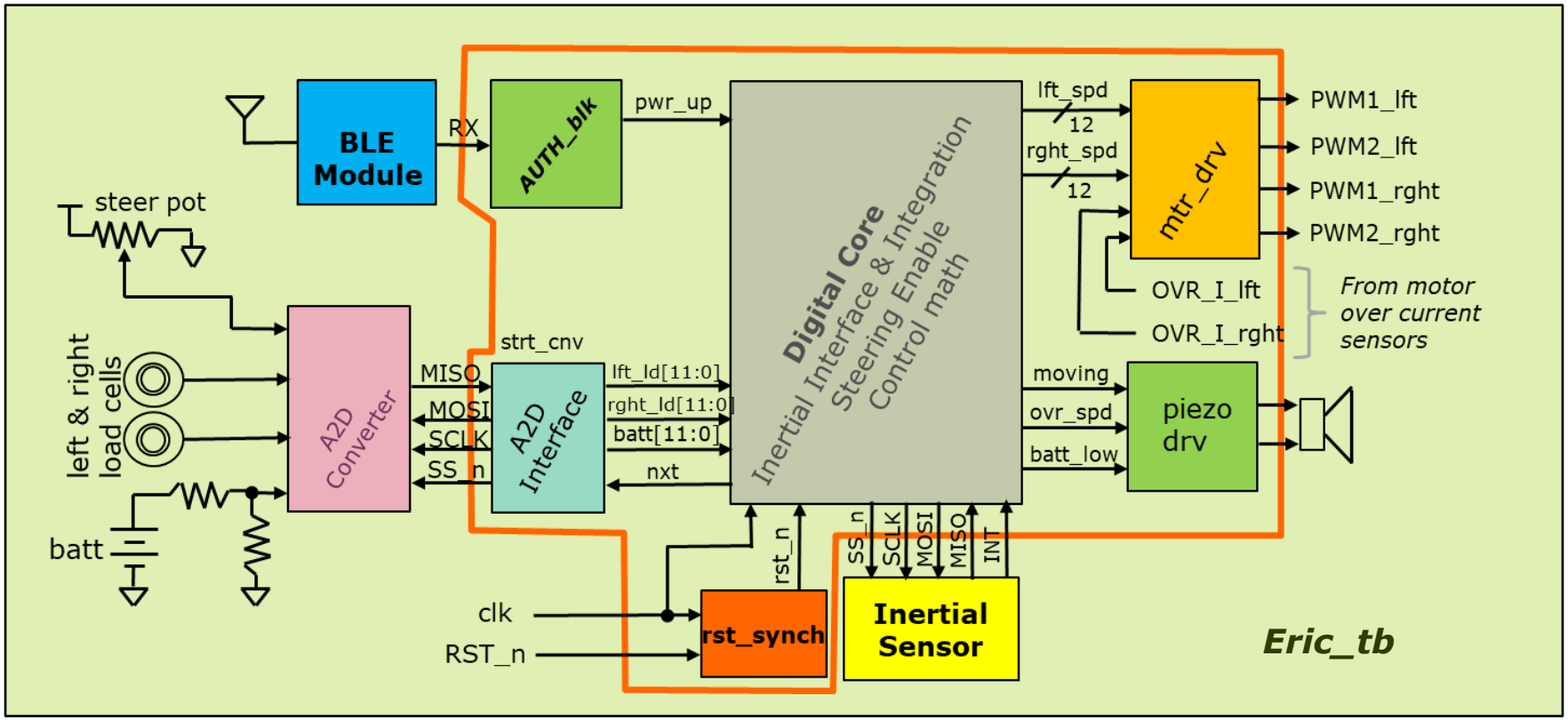
    1. Test that 40 or more **OVR_I_lft** or **OVR_I_right** pulses that occur within a blanking period (128 clocks after **PWM_synch**) do not cause a shut down (**OVR_I_shtdwn**) to assert.
    2. That 40 or fewer **OVR_I_lft** or **OVR_I_right** pulses occurring outside the blanking period will cause **OVR_I_shtdwn** to assert.

mtr_drv interface

Signal:	Dir:	Description:
clk, rst_n	in	50MHz clock, and active low asynch reset
lft_spd[11:0]	in	Signed left motor speed
rght_spd[11:0]	in	Signed right motor speed
OVR_I_lft, OVR_I_rght	in	Instantaneous over current signals left/right
PWM1_lft, PWM2_lft	out	Controls H-Bridge driving left motor
PWM1_rght, PWM2_rght	out	Controls H-Bridge driving right motor

- An 11-bit PWM module was done as part of an in class exercise/HW3.
- Two copies of PWM11 are instantiated in the provided **mtr_drv.sv** block

Required Hierarchy & Interface



Your design will be placed in our testbench to validate its functionality. It must have a block called **Segway.sv** which is top level of what will be the synthesized DUT.
The interface of **Segway.sv must match exactly** to our specified **Segway.sv** interface
Please download Segway.sv (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
INERT_INT	in	INT signal from intertial sensor (rising edge indicates new data ready)
RX	in	UART input from Bluetooth link
PWM1_lft, PWM2_lft, PWM1_rght, PWM2_rght	out	H-Bridge control 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:
OVR_I_lft, OVR_I_rght	in	Instantaneous overcurrent signals. Go to mtr_drv and if occur too frequently can result in shut down of drive to motors.
piezo, piezo_n	out	To piezo buzzer. Used to indicate to rider that battery is low, they are going to fast, or to outside world that they are on and driving (steering enabled).

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

File Name:	Description:
Segway_tb.sv	**Optional** testbench template file.
Segway.sv	**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_FC.v	Model of A2D on DE0-Nano used for full chip testing
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 (333MHz for clk) is pretty easy to make. Your main objective is to minimize area.

My design was about 11900 square microns

Synthesis Constraints:

Contraint:	Value:
Clock frequency	333MHz for clk (3ns period)
Input delay	0.25ns after clk for all inputs
Output delay	0.35ns prior to next clk rise for all outputs
Drive strength of inputs	Equivalent to a NAND2X2 gate from our library
Output load	50fF on all outputs
Wireload model	16000 square micron model
Max transition time	0.10ns
Clock uncertainty	0.15ns

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

NOTE: compile_ultra cannot be used