Signed Operations in Verilog:

- Addition & Subtraction are agnostic operations.
  - An adder to add two signed numbers is no different than an adder to add two unsigned numbers.

- Multiplication is not agnostic
  - A multiplier for signed numbers is a little different than a multiplier for unsigned numbers.

- Magnitude comparison is not agnostic

- Right shifting is not agnostic *(arithmetic vs logical)*

Verilog is stubborn…it wants to be unsigned:

- To infer a signed multiplier all three signals must be declared as type signed. *(dest signal and both operands)*
  - **assign** product = operand1 * operand2;

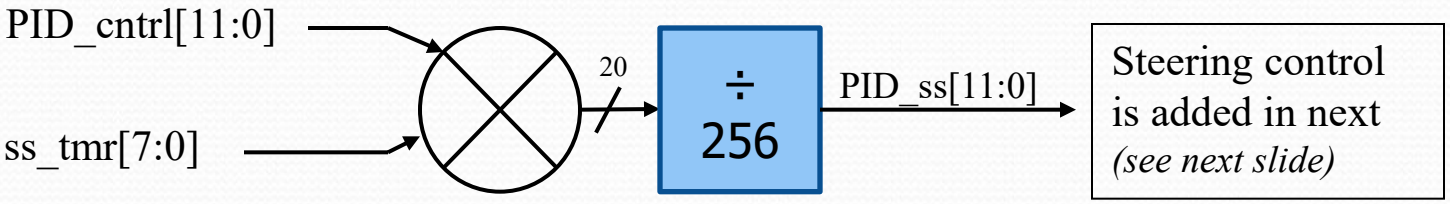
- There is also a $signed() casting operator that can be used:
  assign product = value * **$signed**(6);

- If both numbers are **signed** then a magnitude comparison is signed. Otherwise you could used **$signed()** as well to force a signed magnitude comparison.

- If you want to use the >>> operator you need to ensure both the dest signal and operand being arithmetically right shifted are declared as type **signed**.

Exercise 8 Segway Math: (scaling with soft start)

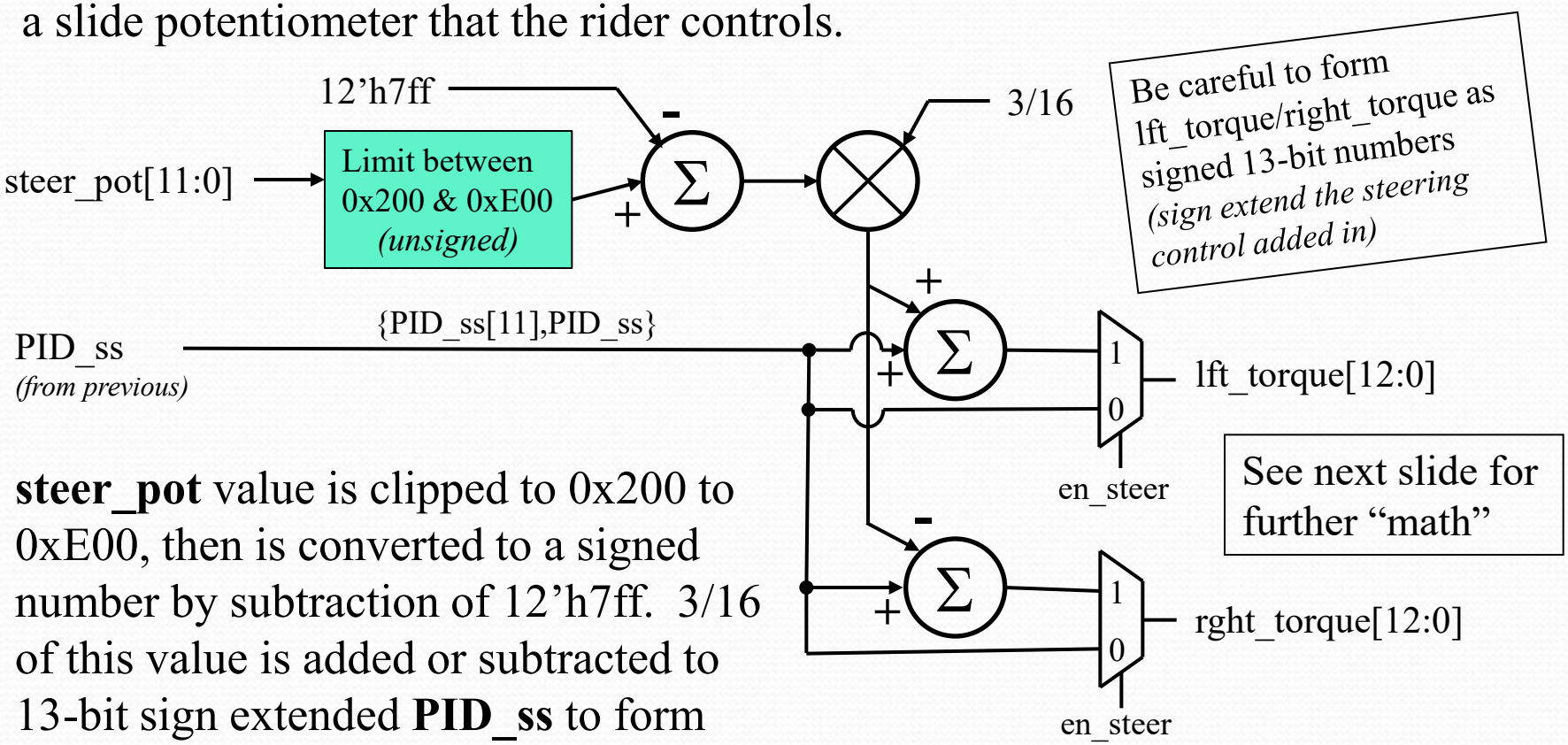
The PID control signal should first be scaled by **ss_tmr**. This ensures a smooth start to the machine so it does not "jerk" the platform level, but rather smoothly drives it to level on power up. **ss_tmr** slowly ramps up to a full 8-bit value after power up of the Segway.



PID_cntrl is signed, and the multiply should be signed. Zero extend **ss_tmr** to form a 9-bit quantity that is multiplied by **PID_cntrl** to form a 20-bit product. You must still cast the zero extended **ss_tmr** to signed to infer a signed multiply ($**signed**({1'b0,**ss_tmr**}) should be multiplied by **PID_cntrl**)

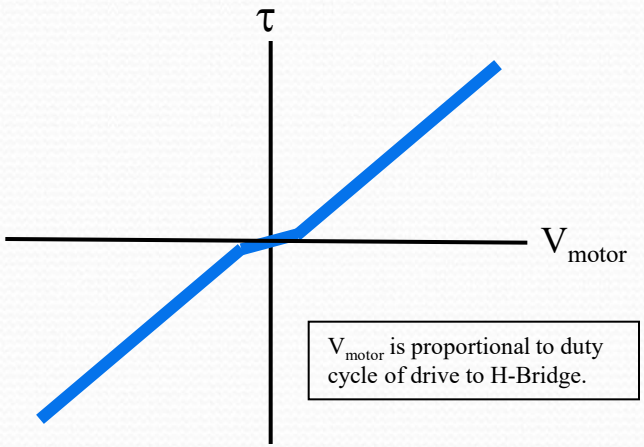
Exercise 8 Segway Math: (steering input)

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.



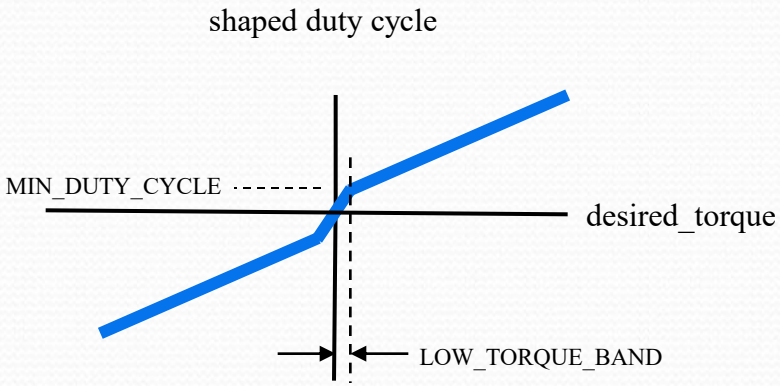
steer_pot value is clipped to 0x200 to 0xE00, then is converted to a signed number by subtraction of 12'h7ff. 3/16 of this value is added or subtracted to 13-bit sign extended **PID_ss** to form **lft_torque/rght_torque** if steering is enabled.

DC Motor Dead Band (need for MIN_DUTY_CYCLE & High Gain 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.

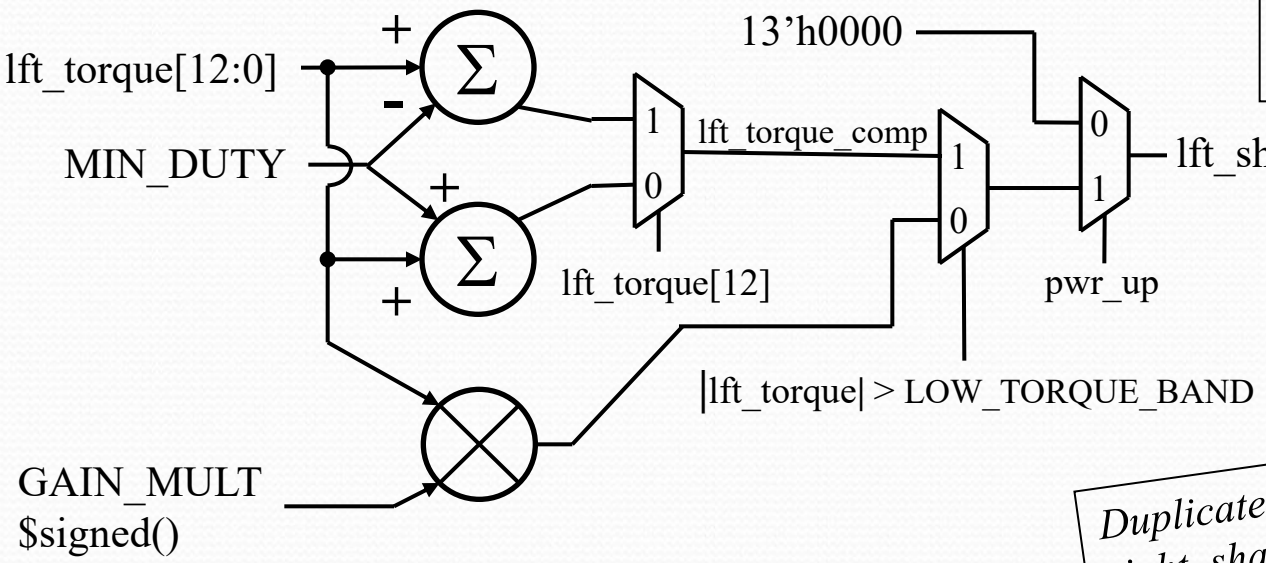
This deadband, though small, causes some havoc when small corrections are needed to keep the platform level. We need to compensate for it.



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)



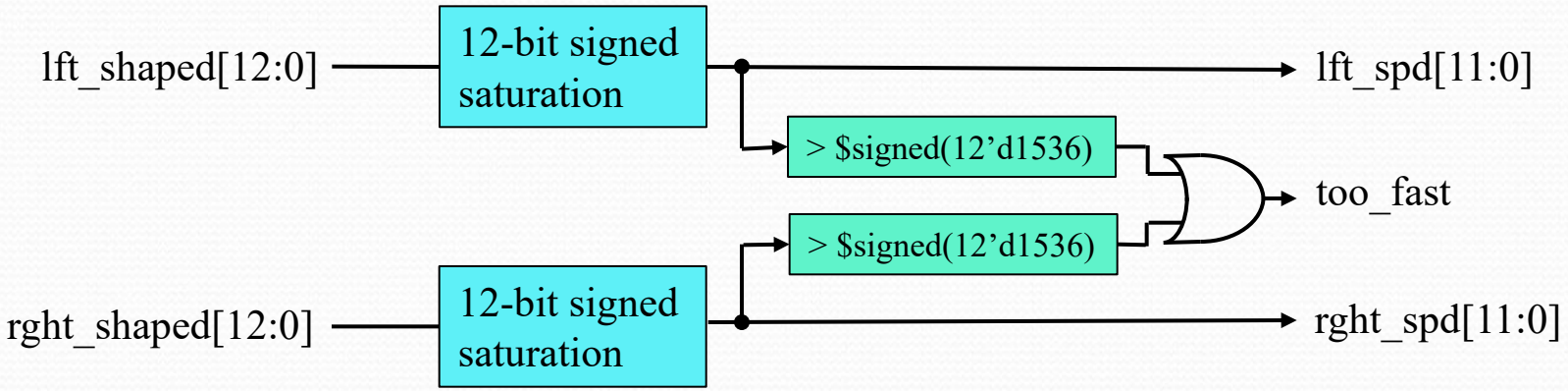
localparams:
MIN_DUTY = 13'h0A8
LOW_TORQUE_BAND = 7'h2A
GAIN_MULT = 4'h4

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.

Exercise 8 Segway Math: (final saturation & over speed detect)



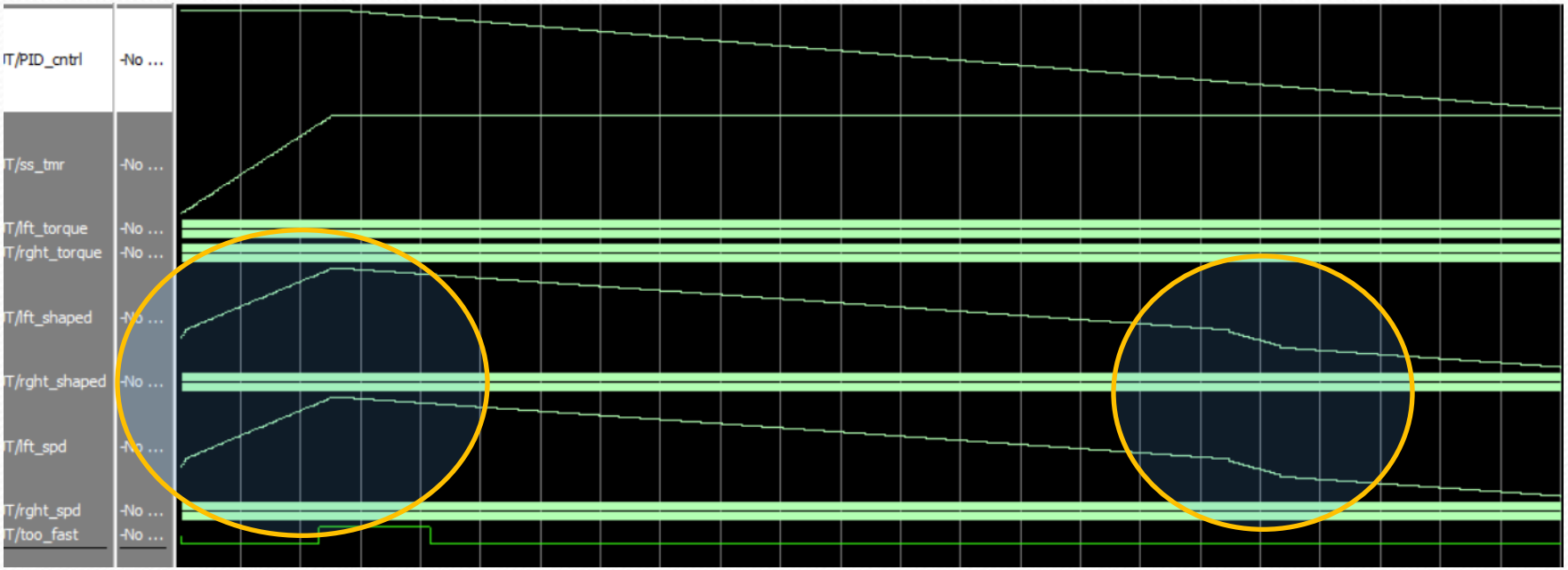
Finally the signed 12-bit signals **lft_spd** and **rght_spd** are formed by signed 12-bit saturation of **lft_shaped**/**right_shaped**. In addition the signal **too_fast** is formed to inform the rider they are going forward too fast and pushing the control limits of the Segway.

Exercise 8 Segway Math: (SegwayMath.sv interface)

Signal:	Dir:	Description:
PID_cntrl[11:0]	in	Signed 12-bit control from PID that dictates frwrd/rev drive of motors to maintain platform balance
ss_tmr[7:0]	in	Unsigned 8-bit scaling quantity used to provide a soft-start to control loop. **PID_cntrl** is scaled by this timer that ramps up slowly from power on.
steer_pot[11:0]	in	12-bit unsigned measure of steering potentiometer. Comes from ***A2D_intf***. Limited and converted to signed version internal to ***Segway_math***
en_steer	In	Indicates steering has been enabled. Enabled by rider having equal weight distribution on load cells.
pwr_up	in	If ~**pwr_up** then both **lft_spd** & **rght_spd** are forced to zero
lft_spd/rght_spd [11:0]	out	Desired output speed/torque for each of left/right motors. 12-bit signed quantity.
too_fast	out	If either lft_spd or right_spd exceed 12'd1792 then this signal is asserted. Used to warn rider of approaching control limits.

Exercise 8 Segway Math: (Testing it) (recommended test 1)

This is a simulation where **PID_cntrl** starts at 12'h5FF, and **ss_tmr** initially ramps from 0 to 8'hFF. Then **PID_cntrl** ramps from 12'h5FF (+1535) to 12'hE00 (-512). In this simulation **steer_en** = 0 and **pwr_up** = 1.

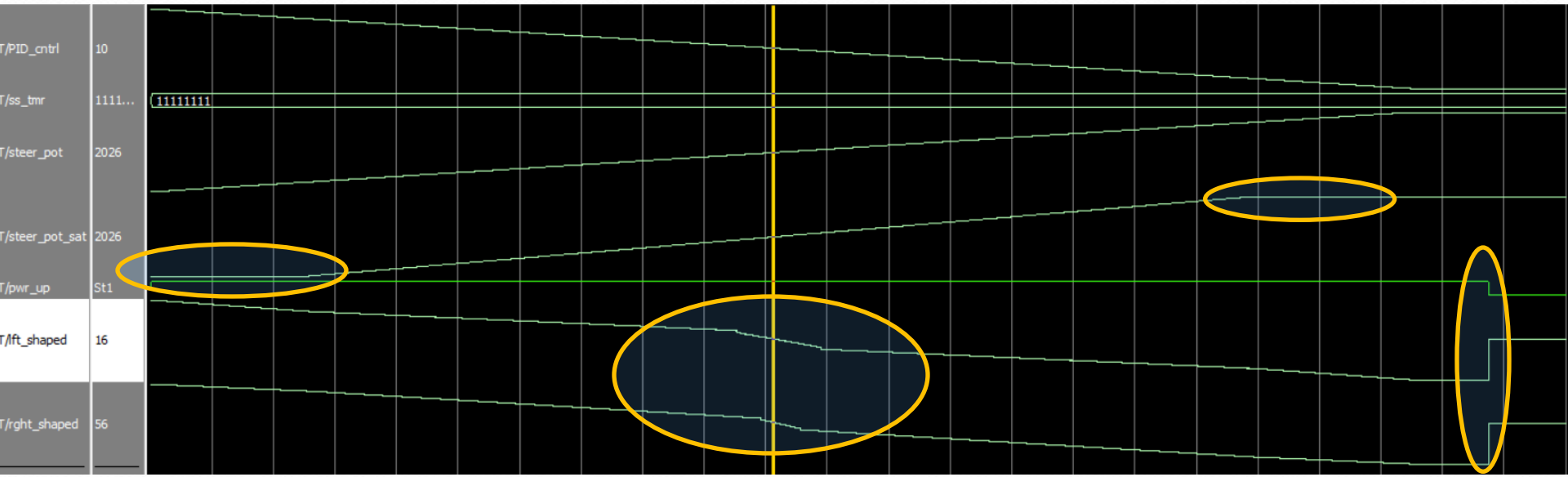


During initial ramp up of **ss_tmr** we see **lft_shaped/right_shaped** ramp up too. Initially in high gain region. We also see **too_fast** asserted after **ss_tmr** is near full, and until **PID_cntrl** falls far enough.

As **PID_cntrl** crosses through LOW_TORQUE_BAND we see the shaped signal transition through the high gain region.

Exercise 8 Segway Math: (Testing it) (recommended test 2)

In this simulation **PID_cntrl** starts at 12'h3FF *(1023)* and is ramped down to 12'hC00 *(-1024)*, but **ss_tmr** starts and stays at 8'hFF for the entire simulation. **steer_pot** is ramped from 12'h000 to 12'hFFE. In this simulation **steer_en** = 1, and **pwr_up** falls at the very end of the simulation.



Note the saturation/clipping of **steer_pot_sat** to between 0x200 and 0xE00.

NOTE: *you may have a different signal name than* **steer_pot_sat**.

In this simulation we see difference between **lft** & **rght** because **en_steer=1** and **steer_pot** is being swept.

When **pwr_up** falls both lft/right spds go to zero

Exercise 8 Segway Math: (Testing it) (more testing in the future)

- These two tests are obviously not sufficient to prove your ***SegwayMath*** is good.
- However, if you can duplicate similar results you are probably pretty close.
- We will revisit validation of this unit in the future when we learn more about testing with test vectors read from a file.
- For now try to validate to the above two tests.

Submit both **SegwayMath.sv** and **SegwayMath_tb.sv** even if not finished by end of class (Weds 9/24)