Exercise 14 (Balance Control Math)(HW4 Prob 4):

- Do as team of 2 **(both people submit to dropbox, but include names in code as comment)**
- You will not get it done this period (no way)
- You **need to finish the implementation over weekend**
  - There is no video quiz Monday so use that time
- Monday we will verify via “golden” test vectors for stimulus and response

Exercise 14 (Balance Control Math)(HW4 Prob 4):

- In HW2 you did a couple of problems (saturate.v and duty.v) that relate to the calculations necessary for balance control. We will complete those calculations in a block called **balance_cntrl.sv.**

- The lion's share of this block is implementing the **PID** math to control balance, but it also incorporates the difference in the load cell readings to effect steering, and some shaping of the desired torque to compensate for deadband of DC motor.

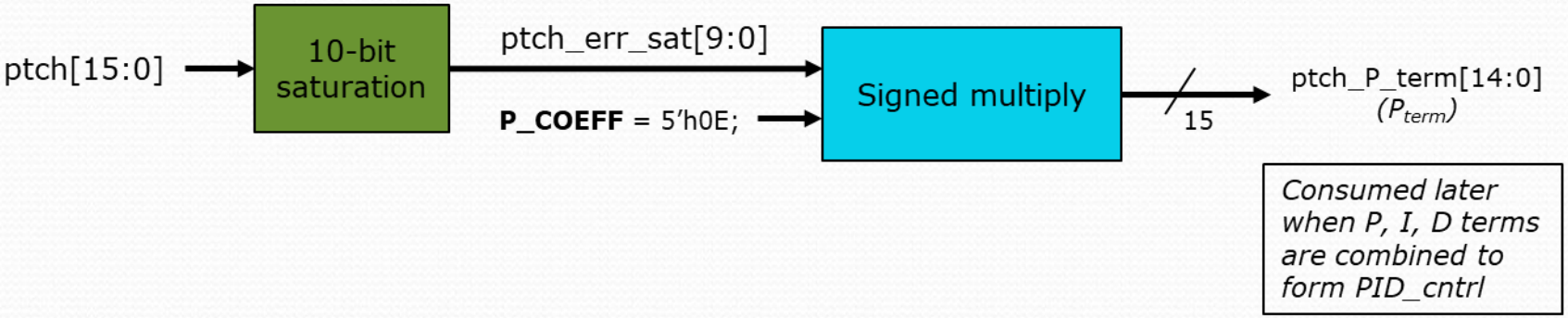
- The next 6 slides describe the math in detail, 5 of them pictorially and one verbally.

- You should code it as you see it, however, it should be coded flat in a single file using dataflow Verilog.

- There is no need for hierarchy or sub-blocks in **balance_cntrl.sv**. It is shown as 5 slides because you can't wedge that much information into a single .ppt slide.

- A shell **balance_cntrl.sv** is provided on the webpage. Start with that!

PID Math (The P of PID (Proportional))

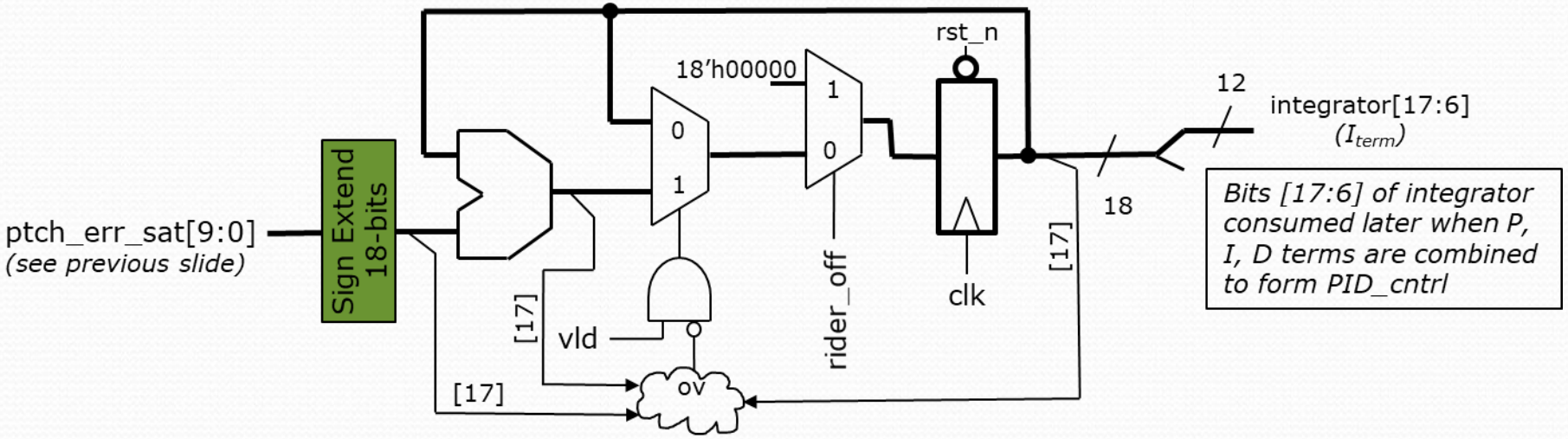


The P_{term} is the simplest of the terms to generate. All you need to do is saturate **ptch** into a 10-bit signed number and multiply it by P_COEFF. You need to ensure you infer a signed multiplier, which means both operands need to be signed ($**signed(P_COEFF)**) and the result it is assigned into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

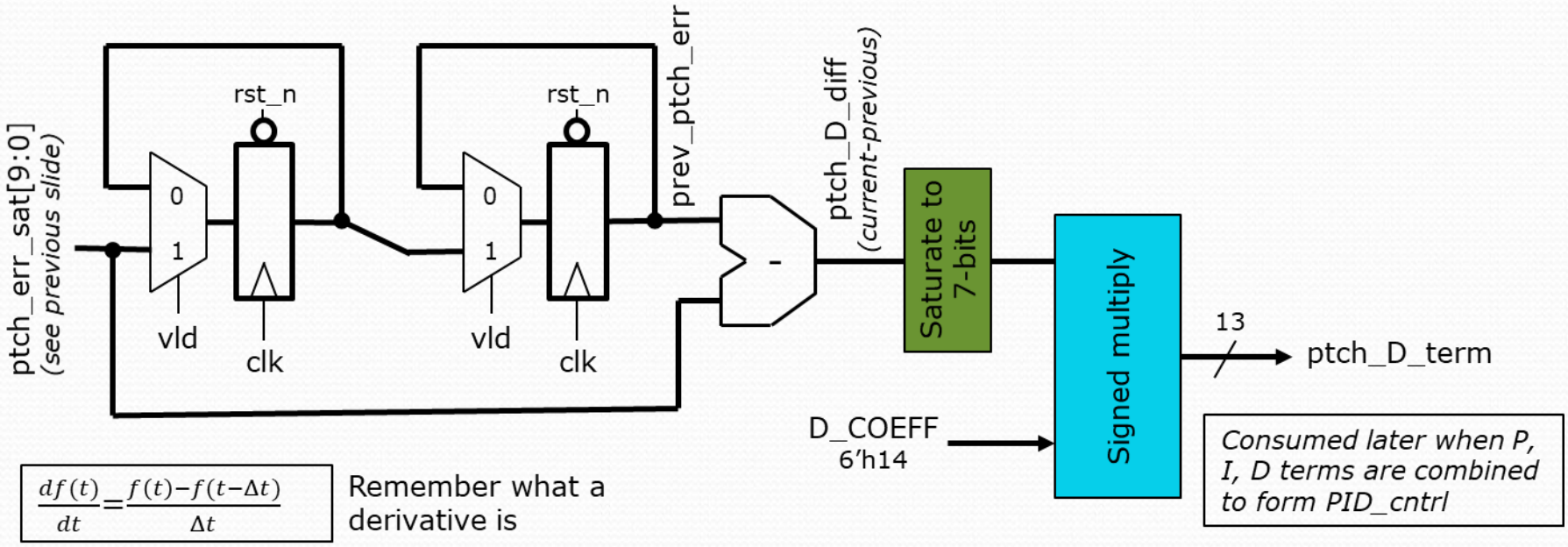


The primary function of the I_{term} is quite simple. On every **vld** reading from the inertial sensor the saturated version of pitch error (**ptch_err_sat**) is accumulated into an 18-bit accumulator register. We then use the upper bits ([17:6]) of this accumulator to form our I_{term} that summed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



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

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.

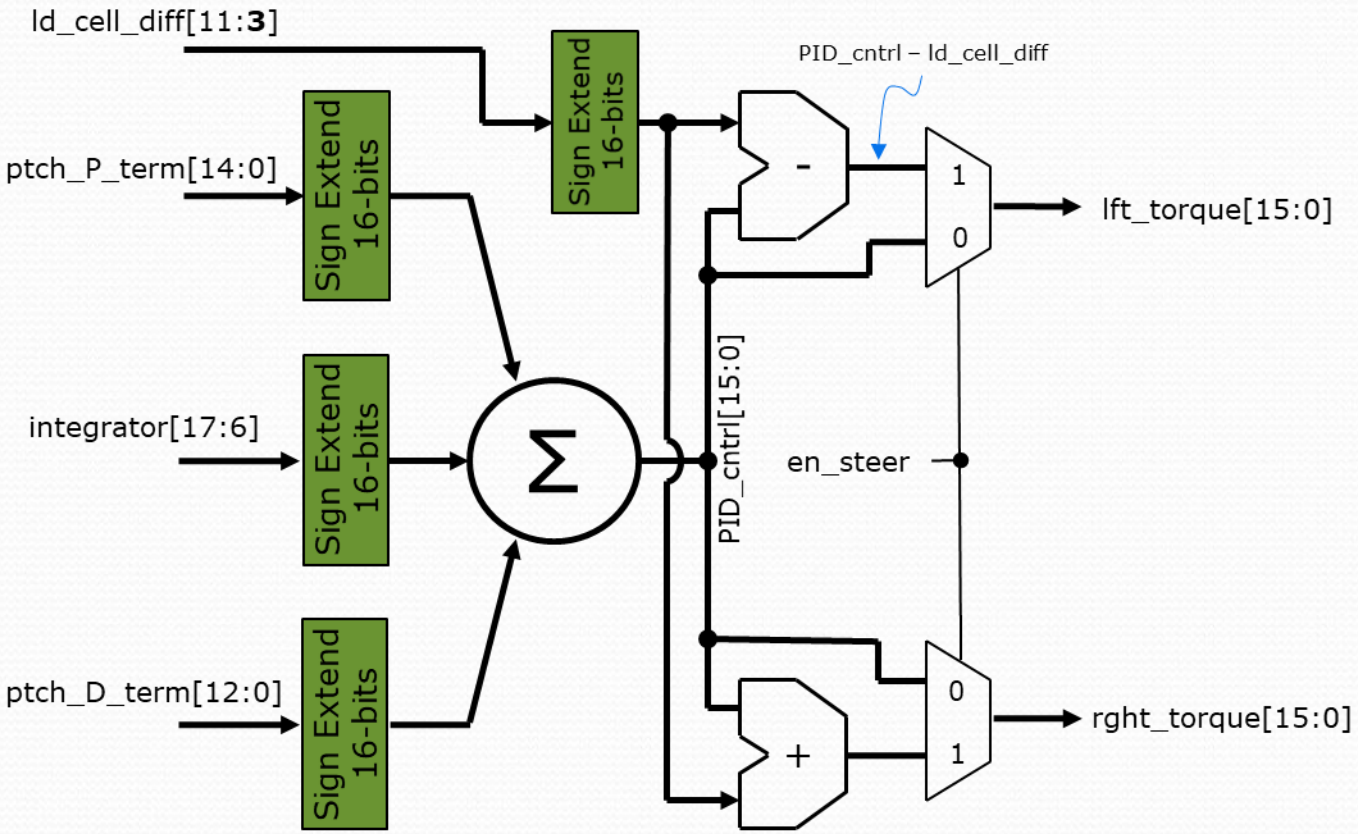
Exercise 14 (Balance Control Math)(HW4 Prob 4):

The following list is a verbal step by step of what is presented in the previous slides. Compare it directly to the dataflow diagrams of the previous three slides and see if it "jives" in your mind.

1. Saturate the incoming signed 16-bit ptch measurement to a 10-bit signed number, call it **ptch_err_sat**
2. Infer a signed multiplier to produce a 15-bit product called **ptch_P_term**

3. Infer an 18-bit wide asynch cleared register called **integrator**.
4. This register will by synch cleared by a signal called **rider_off**.
5. Most of the time this register will recirculate its contents, but under the case of a **vld** reading from the inertial sensor, and no overflow it will update to **integrator** + SE_{18}(**ptch_err_sat**).
6. In the above step SE_{18}() referred to sign extending to 18-bits. Overflow referred to an overflow in the addition.
7. Bits [17:6] of integrator will be used as the I_{term}
8. By inferring two registers make a 2-deep queue that creates a delayed version of **ptch_err_sat**. This queue should "shift" on **vld** readings from the inertial sensor. Call the output of the 2nd delay register **prev_ptch_err**
9. Subtract **prev_ptch_err** from the current value **ptch_err_sat** to form **ptch_D_diff**
10. Saturate this 10-bit signed number to form a 7-bit number.
11. Multiply (signed) this 7-bit number by a localparam called D_COEFF to form a 13-bit signed number called **ptch_D_term**.

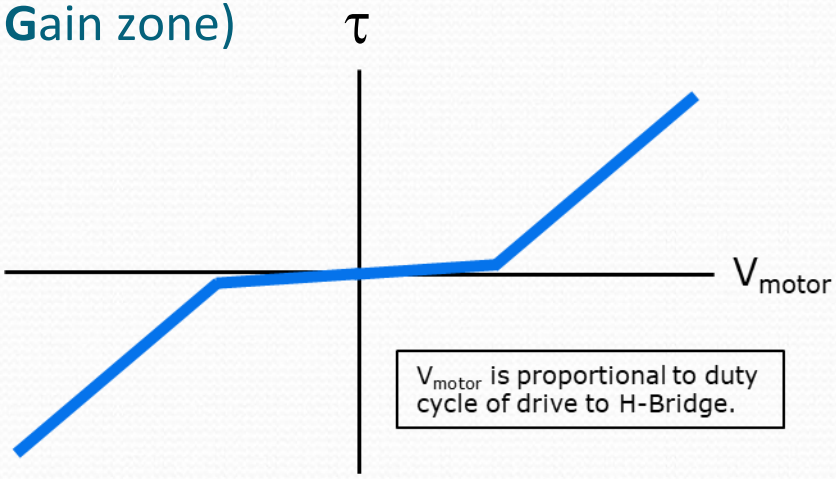
PID Math (Putting it all together)



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.

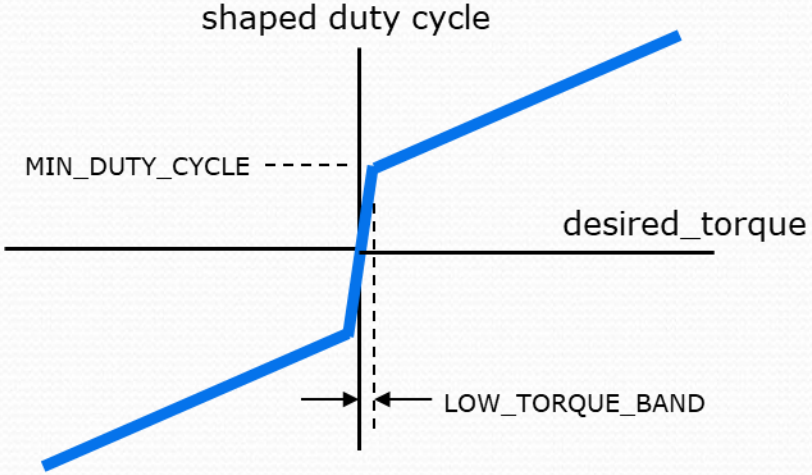
The resulting terms (**lft_torque/rght_torque**) represent the torque we desire for each motor to apply to the wheels. Next we need to compensate for the nasty deadband DC motors have in their torque vs applied voltage.

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.

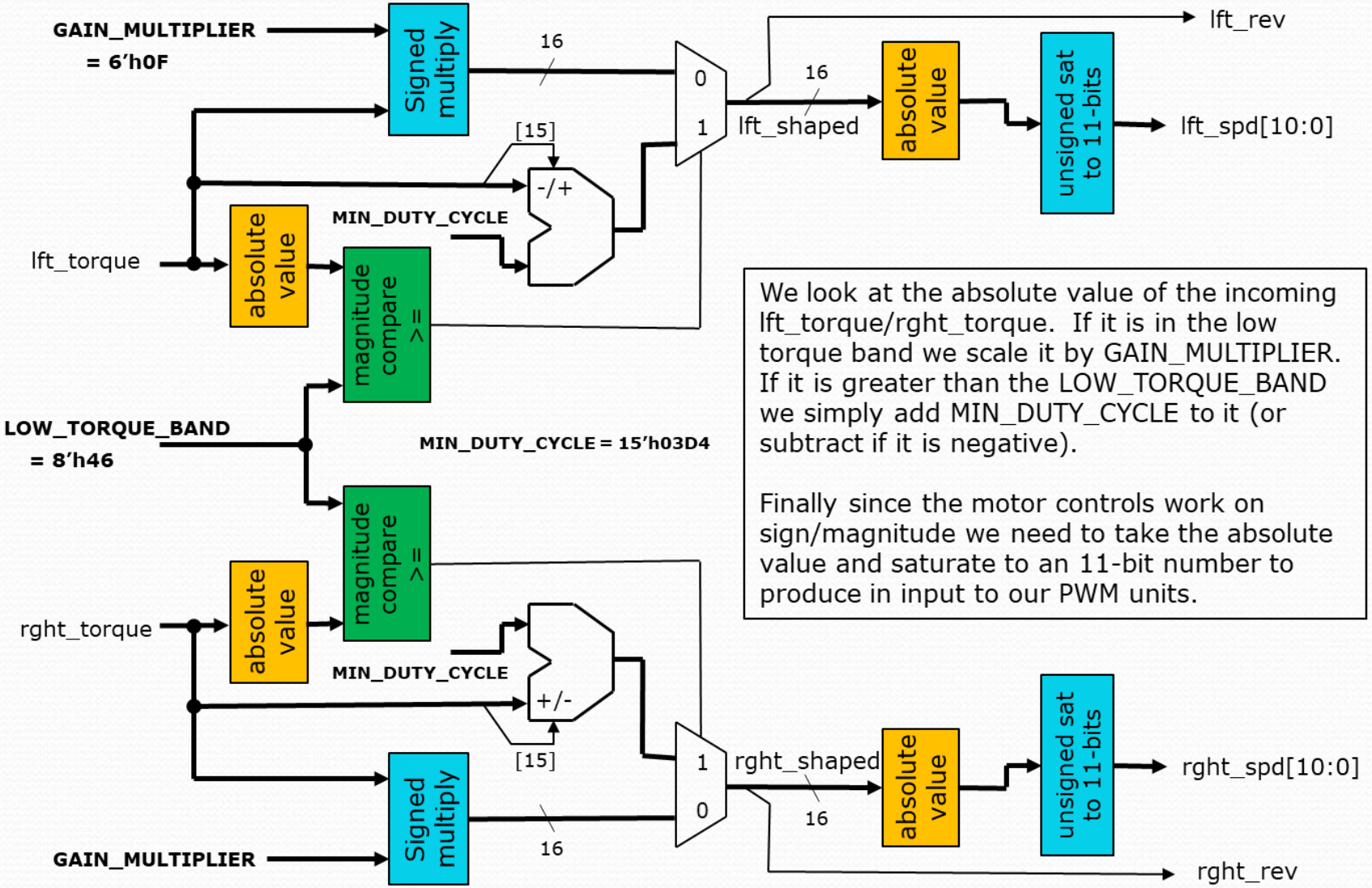
Imagine the havoc this deadband is going to cause for our control system if we do not compensate for it.



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.

Exercise 14 (Balance Control Math)(HW4 Prob 4):

(A skeleton **balance_cntrl.sv** is given in exercise folder)

Signal Name:	Width:	Dir:	Description:
clk, rst_n	1	in	Clock and active low asynch reset.
vld	1	in	Indicates when new inertial reading is valid
ptch	[15:0]	in	Actual pitch (front/back tilt) of the platform. Desired pitch is always zero (flat) so this actually represent error.
ld_cell_diff	[11:0]	in	Signed difference (left – right) of load cells. This determines steering input.
lft_spd, rght_spd	[10:0]	out	Unsigned motor speed for left and right motors
lft_rev, rght_rev	1	out	Direction motor is to run. 1 ➔ reverse
rider_off	1	in	Pulsed high for one clock cycle when rider steps off. This clears the integrator of the PID.
en_steer	1	In	Enable steering when high by subtracting/adding ld_cell_diff in left/right motor speeds

- Implement **balance_cntrl.sv** with the above signal interface. The next few slides discuss testing it.

Exercise 14 (Balance Control Math)(Initial Testing):

- We will perform two tests on balance_cntrl math.
  - The first test (**balance_cntrl_dbg_tb.v**) is a functional test that is easier to debug, but not thorough.
  - The second test will apply 1000 vectors of random stimulus. It is more thorough but very difficult to debug if an error is found.
- The first testbench (**balance_cntrl_dbg_tb.v**) is provided for you. Run your DUT against this testbench and debug in preparations for Monday's class.
- For the 2nd testbench stimulus and "golden" response vectors are provided, but you will make the testbench. This is Monday's exercise.
- **balance_cntrl_dbg_tb.v** code is commented fairly well with what it is trying to test. Look at the comments when debugging your code. This code running perfectly does not ensure your DUT is good.

Exercise14 Part II Next:

Random Testing

Exercise 14 (Balance Control Testing)(HW4 Prob 4):

- You will test your **balance_cntrl.sv** unit using stimulus and expected response read from a file. In the HW4 folder on the website you will find **balance_cntrl_stim.hex** and **balance_cntrl_resp.hex**. These represent stimulus and expected response.

- For **balance_cntrl_stim.hex** the vector is 32-bits wide and is assigned as follows:

Stimulus Bit Range:	Signal Assignment:
stim[31]	rst_n
stim[30]	vld
stim[29:14]	ptch
stim[13:2]	ld_cell_diff
stim[1]	rider_off
stim[0]	en_steer

- There are 1000 vectors of stimulus and response. Read each file into a separate memory using **$readmemh**.

- For **balance_cntrl_resp.hex** the vector is 24 bits wide and is assigned as follows:

Stimulus Bit Range:	Signal Assignment:
resp[23]	lft_rev
resp[22:12]	lft_spd
resp[11]	rght_rev
resp[10:0]	rght_spd

- Create a testbench to apply the stimulus and check the results. Call it **balance_cntrl_chk_tb.v**

- Loop through the 1000 vectors and apply the stimulus vectors to the inputs as specified. Then wait till #1 time unit after the rise of **clk** and compare the DUT outputs to the response vector (self check). Do all 1000 vectors match?

Submit **balance_cntrl.sv**, **balance_cntrl_chk_tb.v** to the dropbox

Exercise 14 balance_cntrl_chk_tb.v Hints:

- Instantiate DUT (balance_cntrl.sv) connecting all the inputs to the respective bits of a 32-bit wide vector of type **reg**.
- Declare a “memory” of type **reg** that is 32-bits wide and has 1000 entries. This is your stimulus memory
- Declare a “memory” of type reg that is 24-bits wide and has 1000 entries. This is your expected response memory
- Inside the main “initial” block of your testbench do a **$readmemh** of the provided .hex files into the respective “memories”
- Start **clk** at zero and toggle it every #5 time units the way we often do
- In a **for** loop going over 1000 entries assign an entry of the stim memory to the stim vector that drives the DUT inputs
- Wait for @(posedge clk), then wait for #1 more time unit. Now check, do DUT outputs match the respective bits of the response vector?

Loading Memory Data From Files

- This is very useful (memory modeling & testbenches)
  - $readmemb("<file_name>",<memory>);
  - $readmemb("<file_name>",<memory>,<start_addr>,<finish_addr>);
  - $readmemh("<file_name>",<memory>);
  - $readmemh("<file_name>",<memory>,<start_addr>,<finish_addr>);
- **$readmemh** ➔ Hex data…**$readmemb** ➔ binary data
  - But they are reading ASCII files either way (just how numbers are represented)

```
// addr     data
@0000 10100010
@0001 10111001
@0002 00100011
```

example "binary" file

```
// addr     data
@0000   A2
@0001   B9
@0002   23
```

example "hex" file

```
//data
A2
B9
23
```

address is optional for the lazy

Example of **$readmemh**

```
module rom(input clk; input [7:0] addr; output [15:0] dout);

reg [15:0] mem[0:255];      // 16-bit wide 256 entry ROM
reg [15:0] dout;

initial
   $readmemh("constants",mem);

always @(negedge clk) begin
   ////////////////////////////////////////////////////////
   // ROM presents data on clock low //
   ////////////////////////////////////////////////////////
   dout <= mem[addr];
end

endmodule
```