ECE 551

HW4 *(100 pts)*

- Due Weds Nov 7th @ class
- Work Individually
- Use descriptive signal names
- Comment & indent your code
- Code will be judged on coding style

HW4 Problems 1&2 **(10pts) + (5pts)**

1. **(10pts)** Complete the Synopsys Design Vision tutorial. Sign below, (preferably in blood).

I, ____ completed the Synopsys Design Vision tutorial. If I had any problems with it, I discussed them with the TA or Instructor, either in person, or through email.

2. **(5pts)** Project Team Formation

Form a 3 or 4 person project team, **Come up with a team name**, and fill in the table below:

Team Name:	
Person1:	
Person2:	
Person3:	
Person4:	

Print and turn in a paper copy of this sheet.
The rest of HW4 is submitted via dropbox

HW4 Problem 3 **(20pts)** Synthesize your mtr_drv *(from HW3)*

- In HW3 you produced **mtr_drv.sv** which took in a magnitude and direction for the motor drive and produced PWM signals. Recall this block instantiated two copies of **PWM11.sv**
- In this HW you will synthesize **mtr_drv.sv** using Synopsys on the CAE linux machines.
- Write a synthesis script to synthesize your **mtr_drv.sv.** The script should perform the following:
  - Defines a clock of 500MHz frequency and sources it to clock
  - Performs a set don't touch on the clock network
  - Defines input delays of 0.7 ns on all inputs other than **clk**
  - Defines a drive strength equivalent to a 2-input nand of size 2 from the TSMC library (ND2D2BWP) for all inputs except **clk** and **rst_n**
  - Defines an output delay of 0.55ns on all outputs.
  - Defines a 0.15pf load on all outputs.
  - Sets a max transition time of 0.10ns on all nodes.
  - Employ the TSMC32K_Lowk_Conservative wire load model.
  - Produces a min_delay report
  - Produces a max_delay report
  - Produces an area report
  - Writes out the gate level verilog netlist (UART.vg)
- Submit to the dropbox.
  - Your synthesis scripts (**mtr_drv.dc**)
  - The output reports for area (**mtr_drv_area.txt**)
  - The gate level verilog netlist (**mtr_drv.vg**)

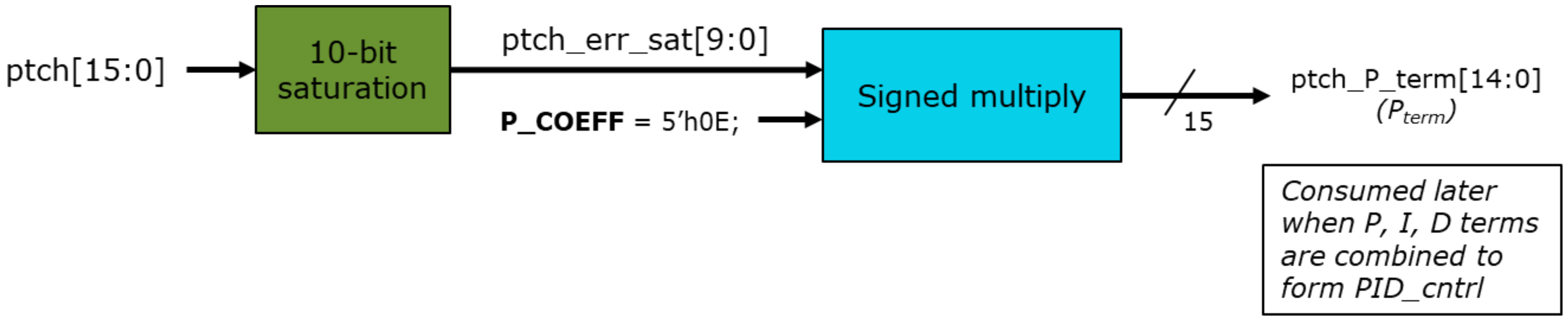
You did something very similar in Exercise13 for your UART.
Leverage that for completing this problem.

We start this problem in Exercise14

HW4 Problem 4 (**30pts)** Balance Control Math & Testing

- 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.
- 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))

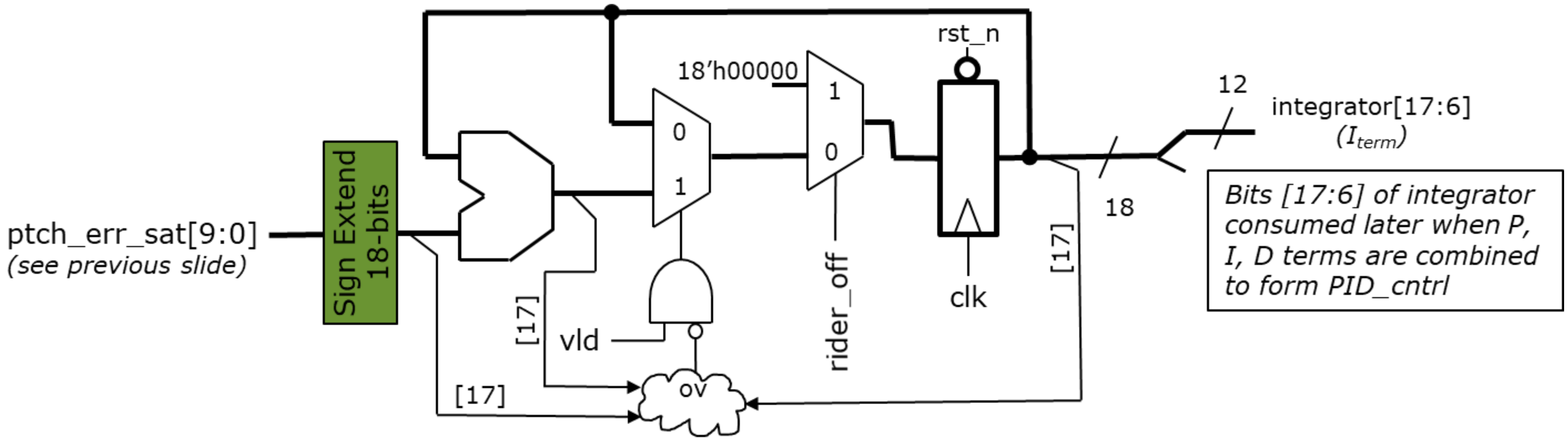


The P_{term} is the simplest of the terms to generate. All you need to do is saturate **ptch** into a 10-bit signed number and multiply it by P_COEFF. You need to ensure you infer a signed multiplier, which means both operands need to be signed ($**signed(P_COEFF)**) and the result it is assigned into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



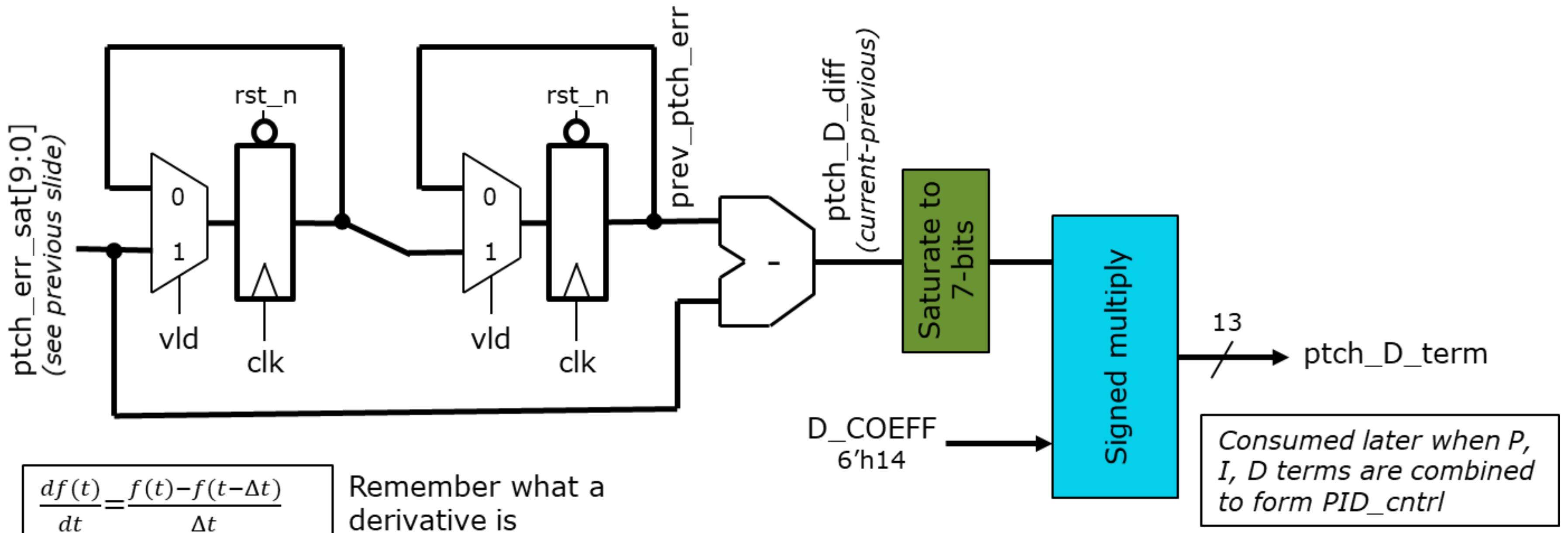
Bits [17:6] of integrator consumed later when P, I, D terms are combined to form PID_cntrl

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))



Consumed later when P, I, D terms are combined to form PID_cntrl

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

Remember what a derivative is

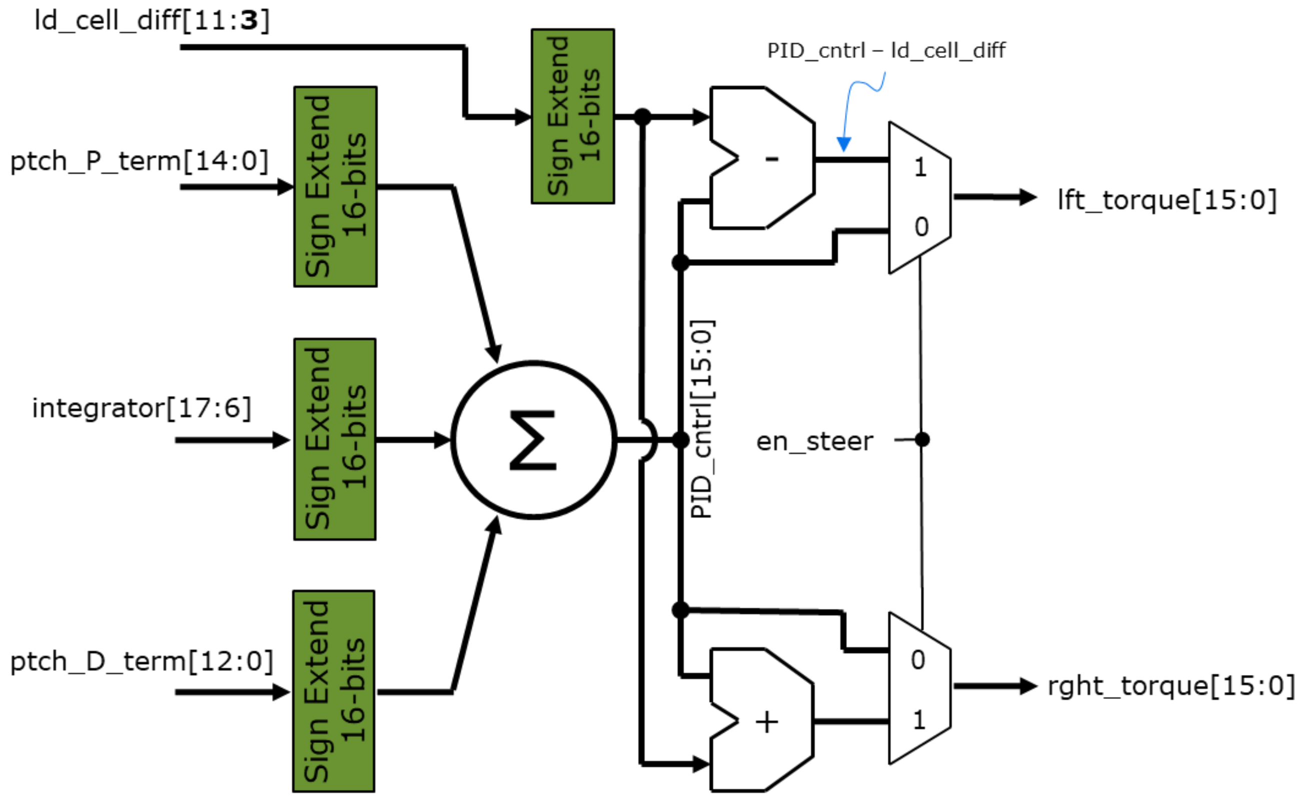
For us Δt is related to the sample rate of the inertial sensor (two **vld** readings). So our derivative is simply proportional to the current reading of **ptch_err_sat** minus a stored sample from two readings ago. There is no reason to divide by Δt since it is a constant and we need to scale the number 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 it all together)

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**.

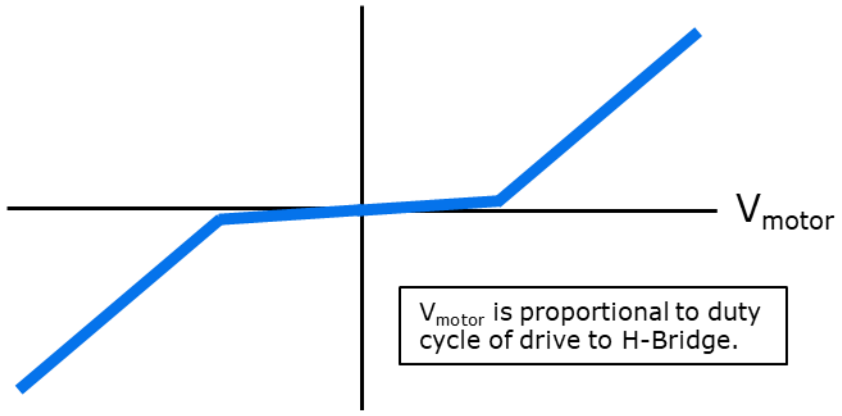
PID Math (Putting it all together)



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.

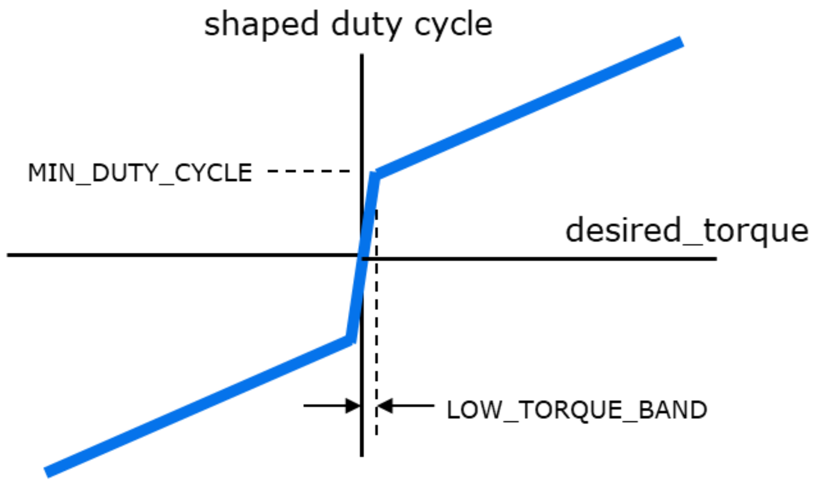
The resulting terms (**lft_torque/rght_torque**) represent the torque we desire for each motor to apply to the wheels. Next we need to compensate for the nasty deadband DC motors have in their torque vs applied voltage.

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.

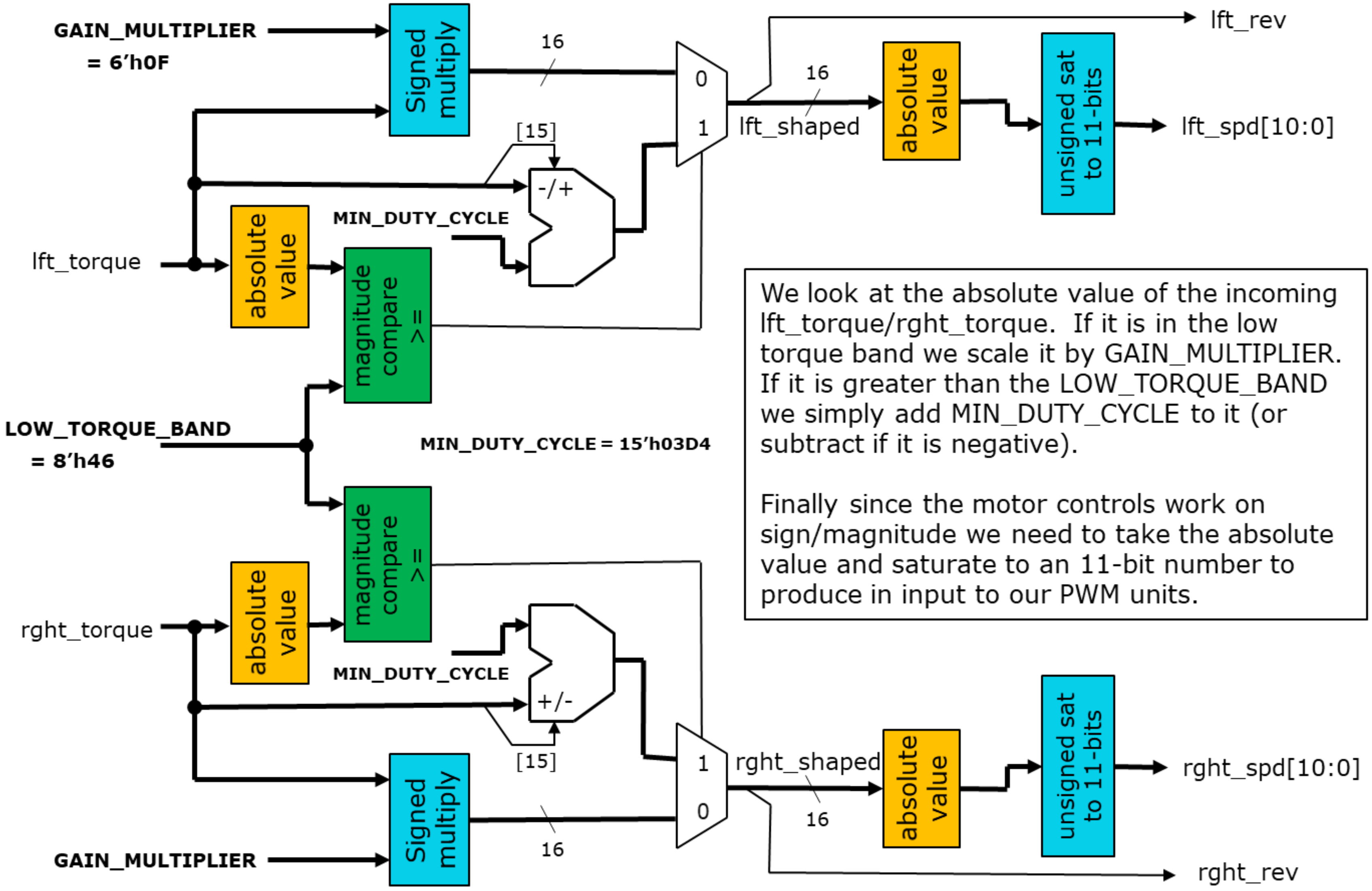
Imagine the havoc this deadband is going to cause for our control system if we do not compensate for it.



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.

HW4 Problem 4 (**30pts**) Balance Control Math & Testing

(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.

HW4 Problem 4 (**30pts)** Balance Control Math & 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.

HW4 Problem 4 **(30pts)** Flight Control Math & Testing

- 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 **flght_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

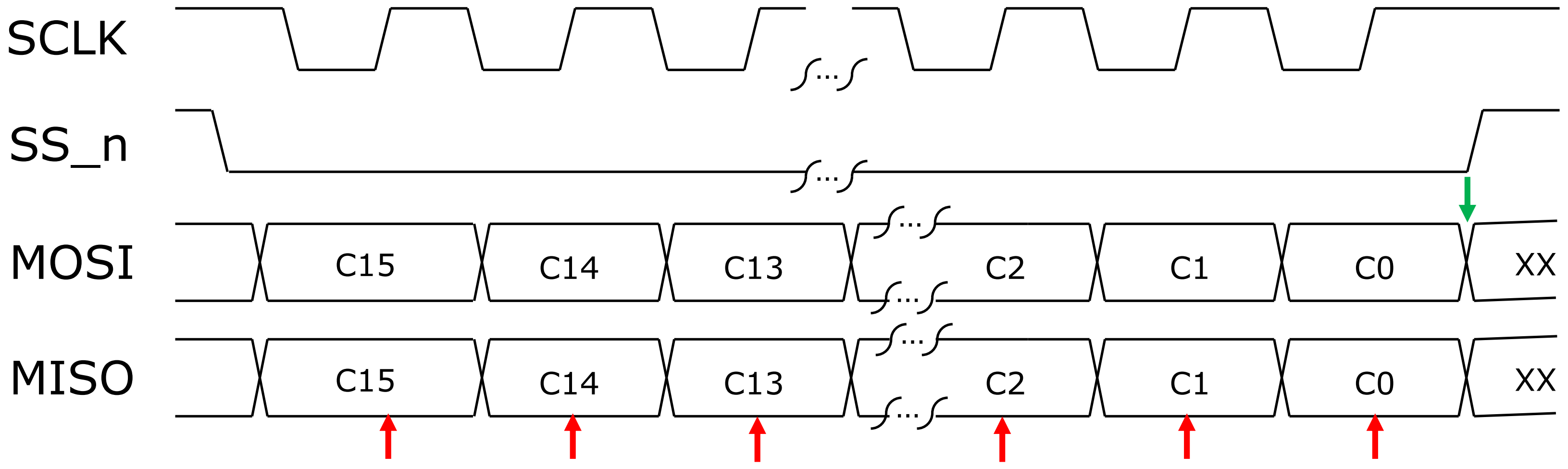
We start this problem in Exercise15

HW4 Problem 5 **(35pts)** SPI Tranceiver

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) changes **MISO** on the falling edge too. We sample **MISO** on the rising edge (see red arrows).

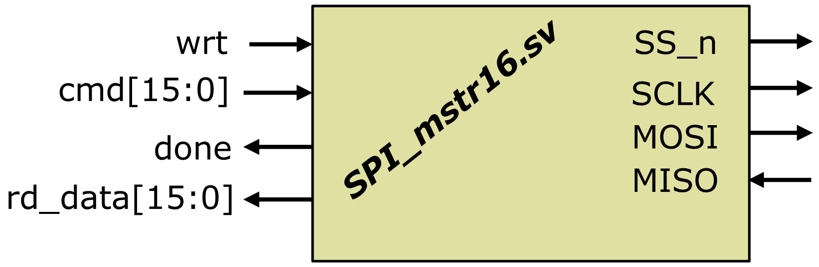
The sampled version of **MISO** in turn gets shifted into our 16-bit shift register. (on **SCLK** fall)

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 thing 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]

HW4 Problem 5 (**35pts)** SPI master for Inertial and A2D Interface

- Create **SPI_mstr16.sv** block

- Download **ADC128S.sv** (model of A2D converter on DE0-Nano, and a SPI slave)

- Also download **SPI_ADC128S.sv** (child of ADC128S.sv that you need)

- Create a testbench (**SPI_mstr16_tb.sv**) in which the **SPI_mstr16.sv** drives the **ADC128S**. Test and debug. **NOTE:** ADC128S.sv only gives data when reading channel 0 of the ADC. Values will start at 0xC00 and decrement by 0x10 every 2nd reads.

- Submit:
    - **SPI_mstr16.sv**
    - Your testbench (**SPI_mstr16_tb.sv**)
    - Output from your self checking test bench proving you ran it successfully (name this file **SPI_mstr16_proof.jpg/png**)