ECE 551
HW5

- Due Weds Nov 20^{th} by class time
- Work Individually on Problems 1
- Work Individually or as team of 2 on Problem 2
- Rest of HW can be done as a team
  - These problems are submitted via dropbox
  - **Please**…only **one** person submit all problems for the team!

HW5 Problem 1 (This problem is **individual** basis)

Individual problem and not covered by exercise. You should have it done by Thurs Nov 14th because we APR it at that time.

1. (20pts) Post Synthesis Simulation of **telemetry.vg**

Posted on the class webpage (under the tutorials section) is a file about Post Synthesis Validation (simulation). Read it carefully.

As part of HW4 (and Ex14) you created a synthesis script to synthesize **telemetry** and produce the flattened gate level netlist **telemetry.vg** Now incorporate that gate level netlist into your test bench for **telemetry** and prove your post synthesis netlist works.

Submit to the dropbox:

a) Your **telemetry_tb.sv** Verilog test bench used to validate **telemetry.vg**
b) Simulation results proving success. Waveforms or results from the self-checking testbench. Yes…you could turn in something here that fooled me into thinking you did this when you actually didn't. However, you would be hurting yourself, because you will need post synthesis simulation to work for the class project, and you can't fool me for that.

(**Note:** you may want to try post synthesis simulation of a few of your other blocks: mtr_drv,brushless, … to ensure they are ready for the project)

HW5 Problem 2 (This problem as individual or team of 2)

This problem will be started as Ex18 on Fri Nov 15th

ABCDEFGHIJKLMNOPQRSTUVWXYZ
00000000011111111112222222
12345678901234567890123456

Add up the "value" of the first letter of each persons last (family) name. If Wangnan Zhong and I were working together that would be H + Z = 8 + 26 = 34

modulo	0	1	2	3	4
Aspect ratio	.32	.45	.64	.9	1.28

Then modulo that result by 5 to get a number between 0 and 4. This will determine the aspect ratio you are to use for your APR block. For Wangnan and I that would be 4

2. (20pts) APR of **telemetry.vg**

Posted on the class webpage (under the tutorials section) is a video on IC Compiler (APR how to video). There is also an accompanying .pdf with slides. Watch the video and follow along in the .pdf

Perform APR of your **telemetry.vg** netlist with your determined aspect ratio. **Submit a video** of the screen of your linux terminal with the result of your finished APR block. Include a head shot of yourself(s) in the video.

HW5 Problem 3 (A2D_intf.sv)

This problem will be started as Ex17 on Fri Nov 8th

3. (30pts) A2D Interface

In HW3 you produced a SPI master (**SPI_mstr.sv**). We are now going to use that block to make an interface to the A2D converter on the DE0-Nano board that will read the torque sensor, the motor current level and a few other things.

The A2D converter has 8 analog channels it can convert, so obviously the channel is specified by a 3-bit field. We will use four of the channels. This module will perform "round robin" conversions on 4 A2D channels we are using.

A2D Channel:	Slide Pot:
000	Read battery voltage
001	Current the motor is drawing
011	Brake lever position (even though we get an analog reading we use this in a digital manner).
100	Crank spindle torque sensor. (how much "force" is rider putting into the pedals)

HW5 Problem 3 (A2D_intf.sv)

You will be producing a module called **A2D_intf.sv** with the following interface:

Signal:	Dir:	Description:
clk, rst_n	in	clock and asynch active low reset
batt[11:0]	out	Battery voltage result (channel 0)
curr[11:0]	out	Current motor is consuming (channel 1)
brake[11:0]	out	Brake lever position (channel 3).
torque[11:0]	out	Crank spindle torque sensor (channel 4)
SS_n	out	Active low slave select (to A2D)
SCLK	out	Serial clock to the A2D
MOSI	out	Master Out Slave In (serial data to the A2D)
MISO	in	Master In Slave Out (serial data from the A2D)

This unit will be instantiating a copy of your **SPI_mstr.sv** to interface with the A2D which is a SPI based peripheral

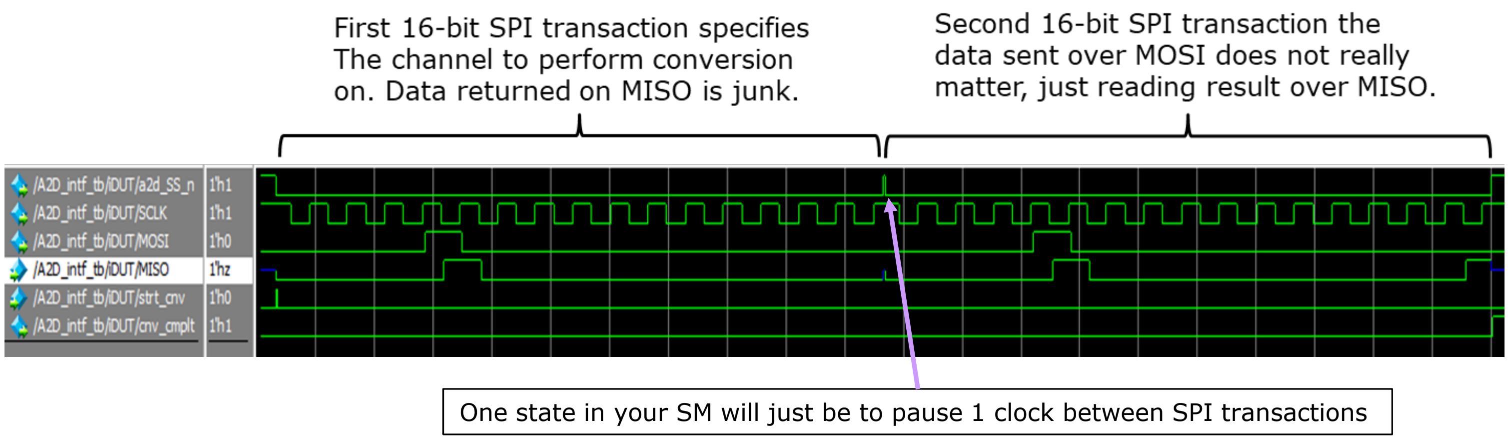
HW5 Problem 3 (A2D_intf.sv)

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 0x0800 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 of 8 A2D channels to convert, and the lower 11-bits of the command are zero. Therefore, the 0x0800 in this example represents a request for channel 1 (motor current)

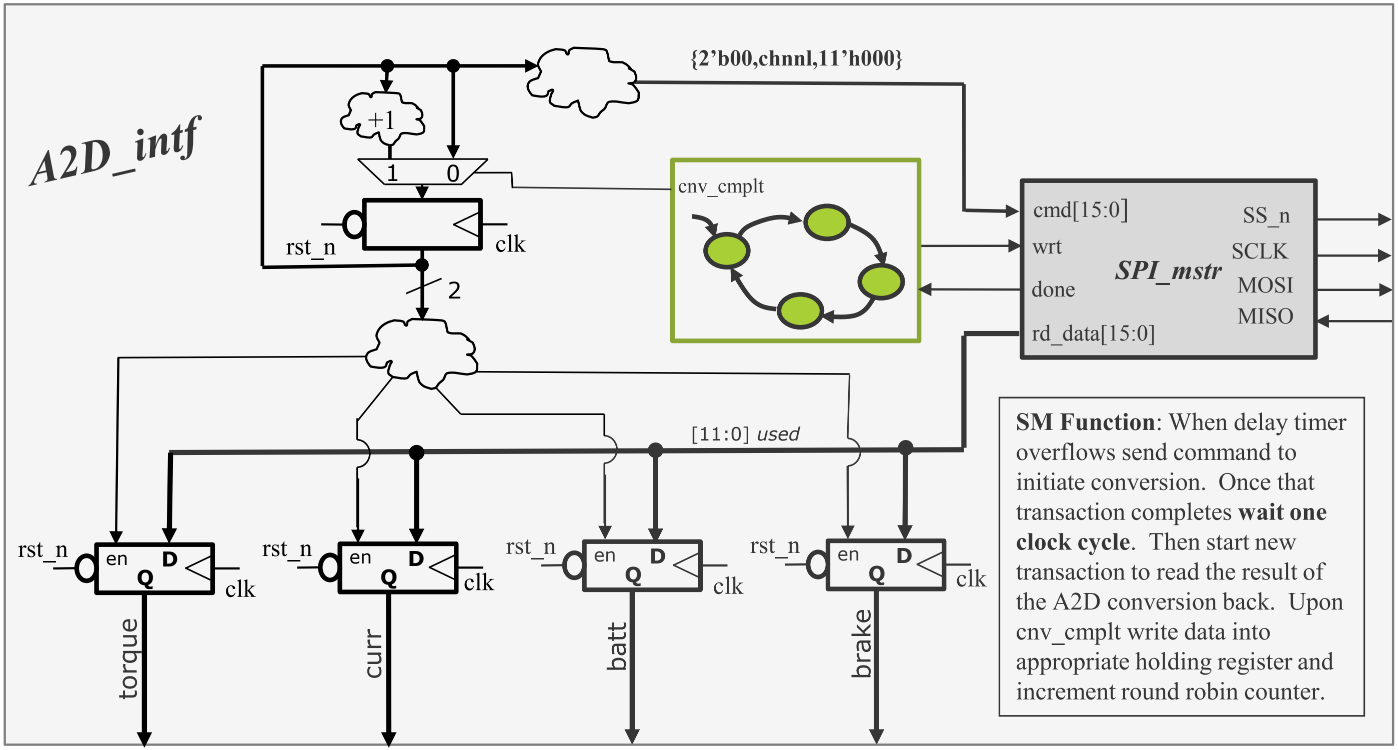
For the next 16-bit transaction the data sent over MOSI to the A2D does not matter that much. We are really just trying to get the data back from the A2D over the MISO line.

NOTE: you need at least a 1 clock cycle pause between the completion of the first SPI transaction and the initiation of the second.



HW5 Problem 3 (A2D_intf.sv)

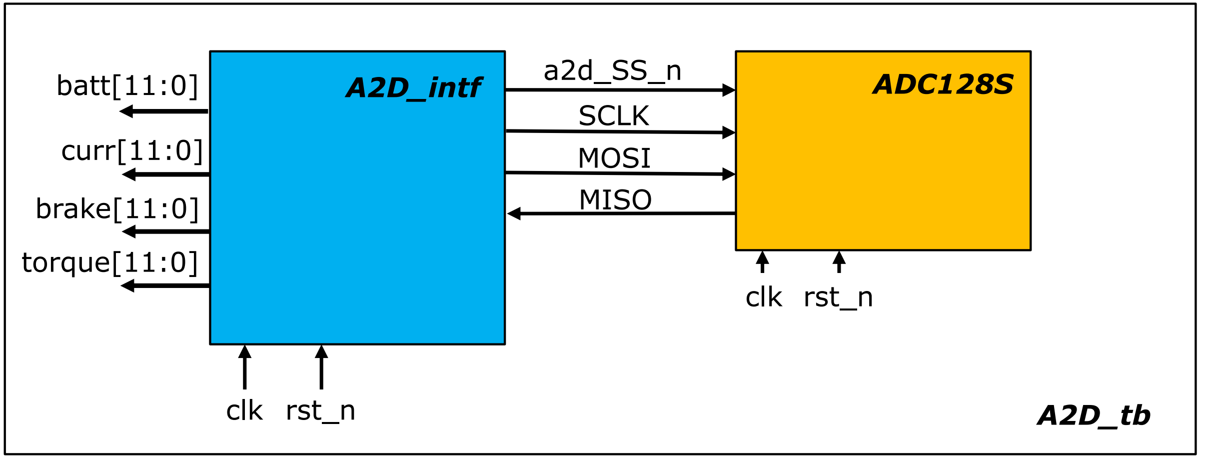
Not shown here is a 14-bit counter. The SM only kicks off a conversion once every 328µs. There are 4 conversions to the round. So the period for any given sensor is 1.3ms



SM Function: When delay timer overflows send command to initiate conversion. Once that transaction completes **wait one clock cycle**. Then start new transaction to read the result of the A2D conversion back. Upon cnv_cmplt write data into appropriate holding register and increment round robin counter.

HW5 Problem 3 (A2D_intf.sv) (Testing)

A model of the A2D converter is provided on the course website (**ADC128S.sv**). Download this and make a test bench that incorporates your A2D_intf and ADC128S.

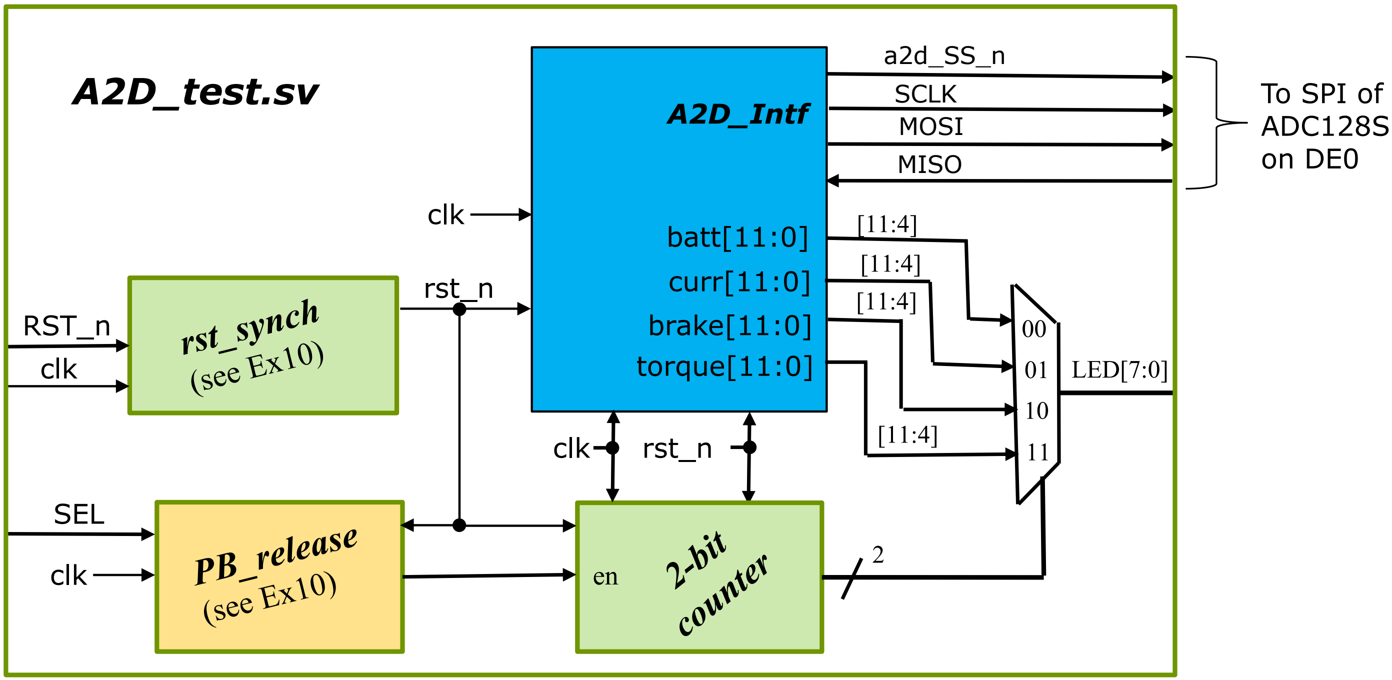


NOTE: The general return for ADC128S is:

0xC00 – 0x010*num_readings + 0x00X where X is the channel requested. Since A2D_intf reads channels 0,1,3,4 we would expect the first 4 values to be 0xC00, 0xBF1, 0xBE3, and 0xBD4. Then the next round would give 0xBC0, 0xBB1, 0xBA3 and 0xB94

HW5 Problem 3 (A2D_intf.sv) (Testing by Mapping to the "real thing")

It is possible to write Verilog for the **SPI_mstr** that works in ModelSim with the provided **ADC128S** model, but still not have it work in "real life". Now you will map your A2D_intf design to the DE0-Nano and test it on the test platform boards.

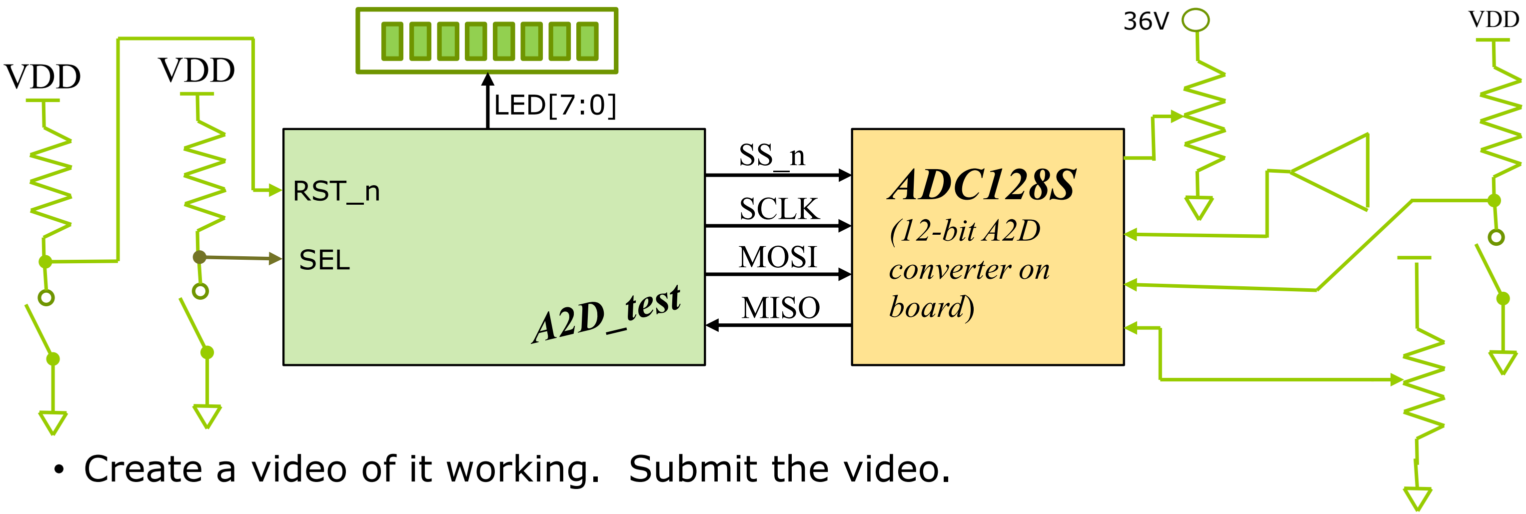


A2D_test.sv, A2D_test.qpf, and A2D_test.qsf are all provided.

HW5 Problem 3 (A2D_intf.sv) (Testing by Mapping to the "real thing")

Selecting:	Expected reading:
batt	Somewhere in the 0xAC to 0xB8 vicinity
curr	Low…the motor is not running so perhaps in the 0x00 to 0x04 region
brake	0xFF normally. Should go low 0x00 when you push the brake button
torque	There is a slide pot connected to this one. When you vary the slide pot you should see the value vary.

- Use Quartus and map A2D_test to a DE0-Nano on the test platform

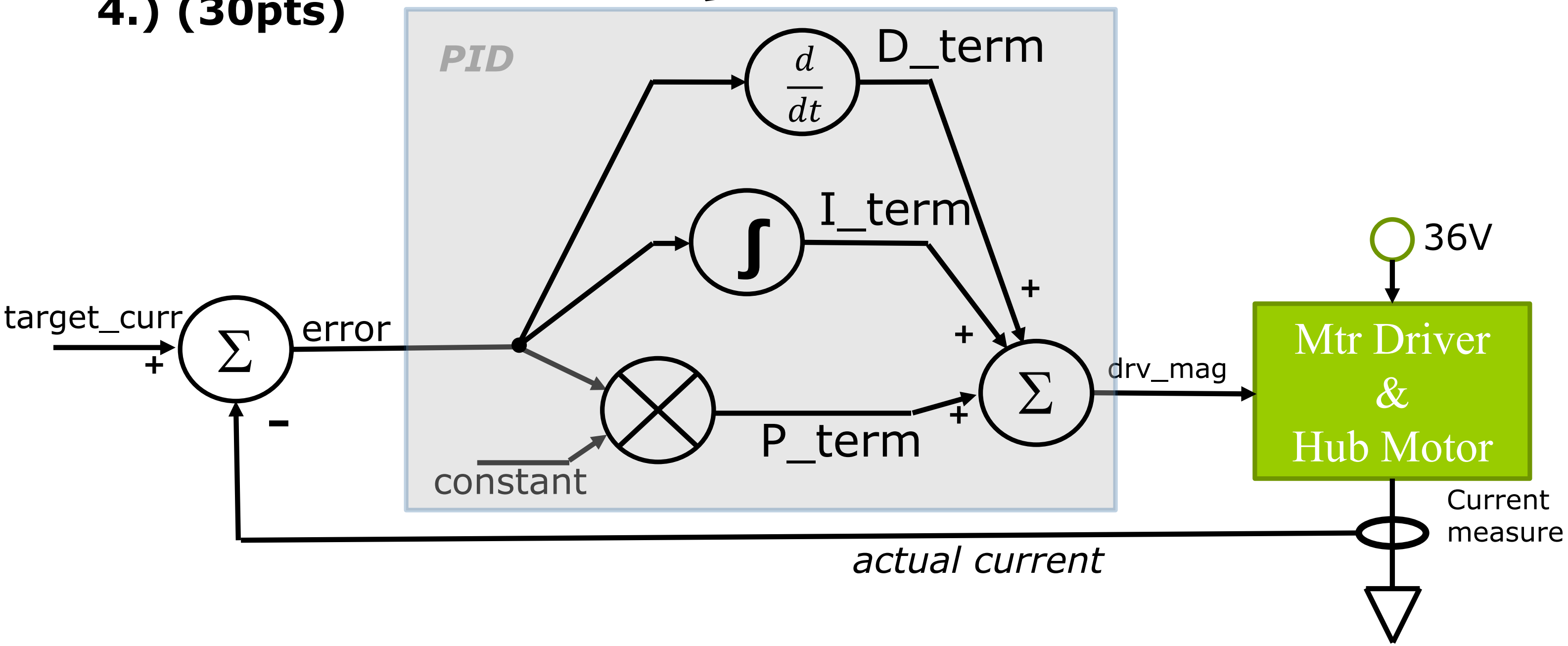


- Create a video of it working. Submit the video.

HW5 Problem 4 (PID.sv)

This problem will be started as Ex17 on Fri Nov 8th

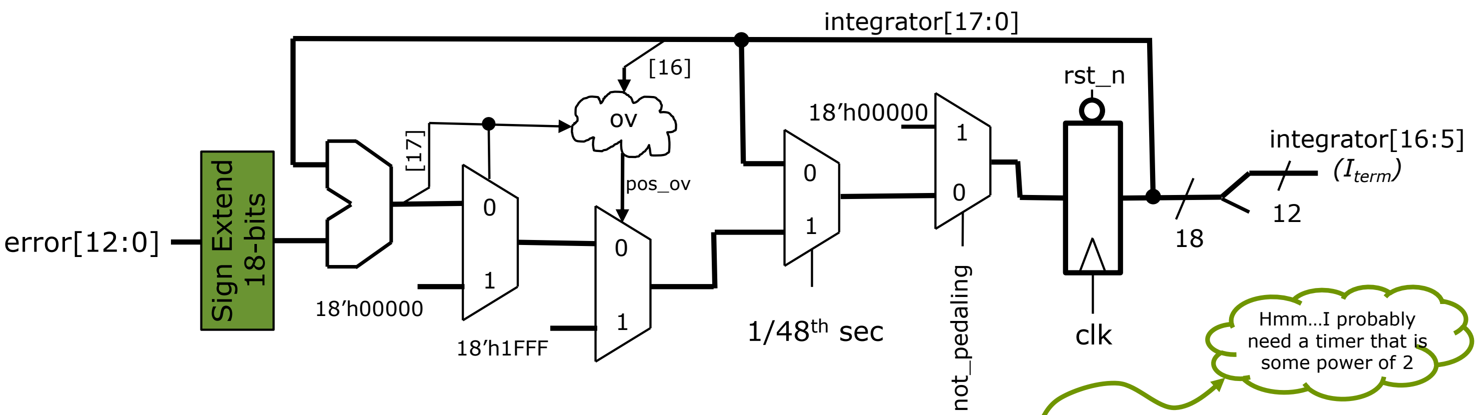
4.) (30pts)



- Integration is nothing more than summing over time, so this is implemented with an accumulator
- A derivative can be approximated by how much a value changed over a given period of time, so this can be implemented by keeping track of previous values of **error**, and subtracting them from the current value of **error**.

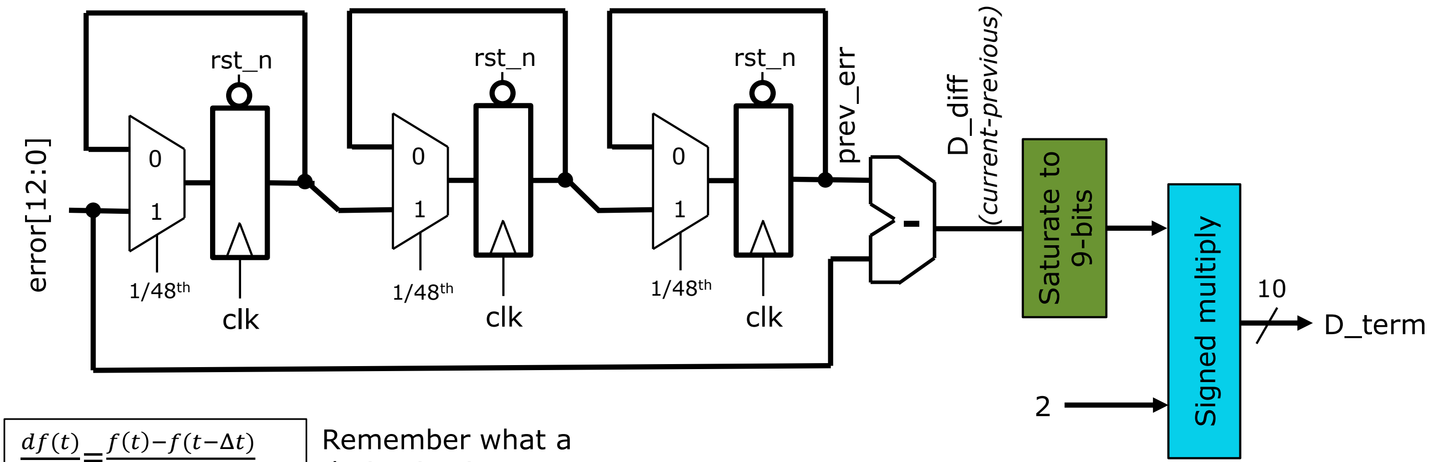
HW5 Problem 4 (PID.sv) (P_term & I_term)

- The ***P_term*** is the ***error*** times a constant. It turns out our constant is 1.
- For the addition of P,I,&D we need 14-bit quantities so ***P_term*** can simply be assigned to be a 14-bit sign extended version of ***error.***
- The integrator is simply an accumulator accumulating error over time.



Integrating accumulator is 18-bits wide and is not allowed to go negative. If new value would be negative we clip it to zero.	Using bit 17 of the adder and bit 16 of the current accumulator we can determine if positive overflow is occurring. If so we saturate to 0x1FFFF.	The integrator cannot be allowed to integrate much faster than the system can respond. We only allow accumulation 48 times a second.	When the rider stops pedaling there is a decent chance the I_term was "wound up". When they resume pedaling we cannot start with a wound up integrator. It must be cleared.

HW5 Problem 4 (PID.sv) (D_term)



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

Remember what a derivative is

For us Δt is 3/48th of a second. So our derivative is simply proportional to the current reading of **error** minus a stored sample from three readings ago (sampled at 1/48th intervals). 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 9-bit number and then multiplied by a factor of 2. *(Remember what you learned in 352 about multiplying by a power of 2)*

HW5 Problem 4 (PID.sv)

Signal:	Dir:	Description:
clk	In	50MHz clock
rst_n	In	Active low asynch reset
error[12:0]	In	13-bit signed error
not_pedaling	In	Asserted if rider is not pedaling
drv_mag[11:0]	Out	Unsigned output that determines motor drive

Implement **PID.sv** with this interface and the functionality outlined in the previous slides.

NOTE: The integrator and the previous D_terms are not updated every clock. In fact at 1/48th of a second there is a little over a million clocks between their updates. This concept of updating a sample only one out of many clocks is often referred to as decimation. So…call the counter you use for this **decimator.**

When you are done with your first pass at implementing PID advance to the next slide to see a modification needed, and how to test it.

HW5 Problem 4 (PID.sv) (Simulation & Testing)

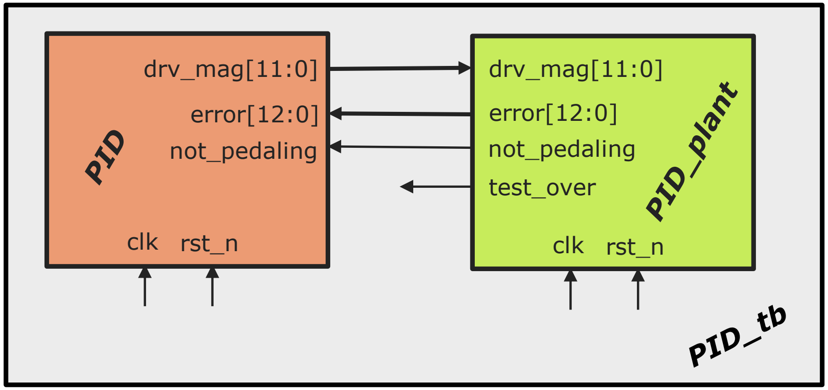
NOTE: OK…so you should have figured out by now that the **decimator** (1/48th second) counter had to be 20-bits wide. So the integrator and D_terms are only updating once every million clocks. In real life a million clocks happens quick (1/48th of a second). But in simulation a million clocks takes a while. To simulate PID loop closure we need to speed this up or we will be waiting for ModelSim forever.

We will make the width of **decimator** a parameter that is passed into PID. The parameter should default to 20, but for simulation purposes we will override it to 15-bits wide so simulation can occur 32X faster.

Go ahead now and modify your code to add a parameter called **DECIMATOR_WIDTH** to **PID.sv** and have it default to 20. You need to ensure your code is all "generic" enough such that it can handle a variable width definition of **decimator**. For instance if you determine when to update integrator and previous D_terms using &**decimator** then that is generic and decimator could be any width.

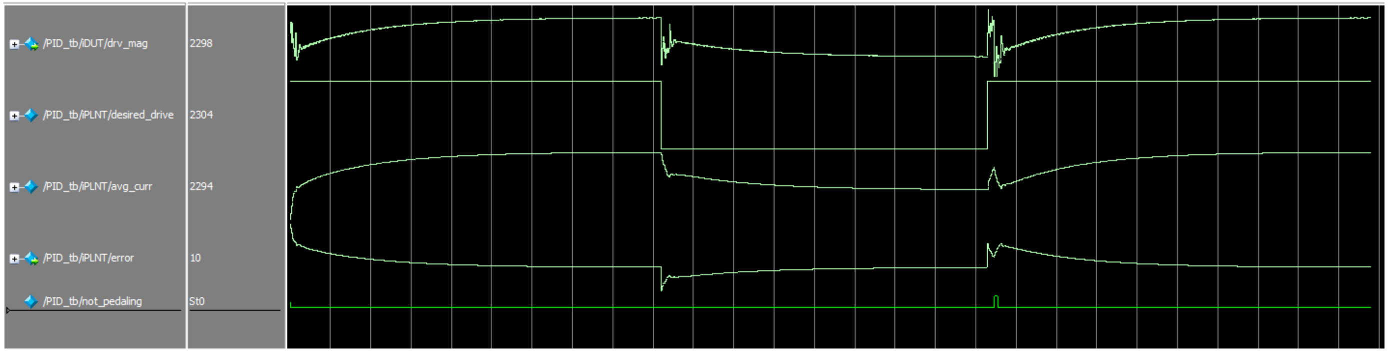
Now you are ready to simulate. Since I am a really nice guy I provided a model of the "plant" you can use to simulate and test your PID. Download **plant_PID.sv** from the canvas site.

HW5 Problem 4 (PID.sv) (Simulation & Testing)



plant_PID is a simple model of that produces an **error** signal into your PID DUT and adjusts **error** over time from **drv_mag**. It also produces a **not_pedaling** signal, and a signal for when the test is over. In short it is darn near magical.

All you have to do is hook it up as shown in a test bench that simply applies clock and reset, and waits for **test_over** to assert.



If you plot the above signals as analog they should look as shown. **drv_mag**, **desired_drv** (inside plant), and **avg_curr** (also inside plant) are unsigned. **error** is signed.

Submit: PID.sv, PID_tb.sv, and an image showing your waveforms looked similar to this.