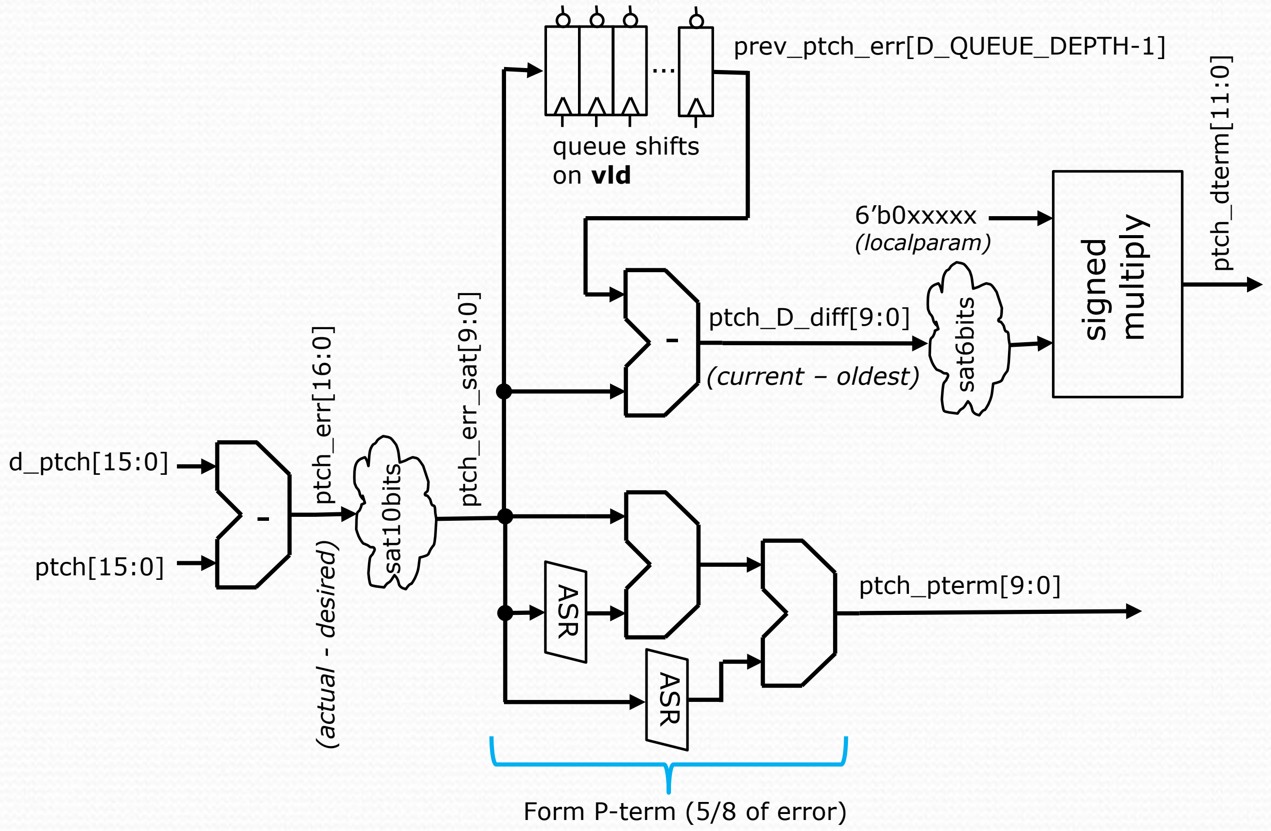
Exercise 14 **(Flight Control Math)(HW4 Prob 4)**:

- 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 (Flight Control Math)(HW4 Prob 4):

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



- Shown is a datapath for calculating the **P** and **D** terms for pitch. Roll and Yaw have identical math
- The derivative term requires a delayed version of the error term
- Implement a queue that is reset to zero on reset, and shifts on new valid inertial data. The depth of the queue should be set by parameter. This would be a good spot for a **for** loop.

- *pterms* and *dterms* from ptch, roll, and yaw will be combined with thrust level and a constant (MIN_RUN_SPEED) to form the final motor speed value.

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

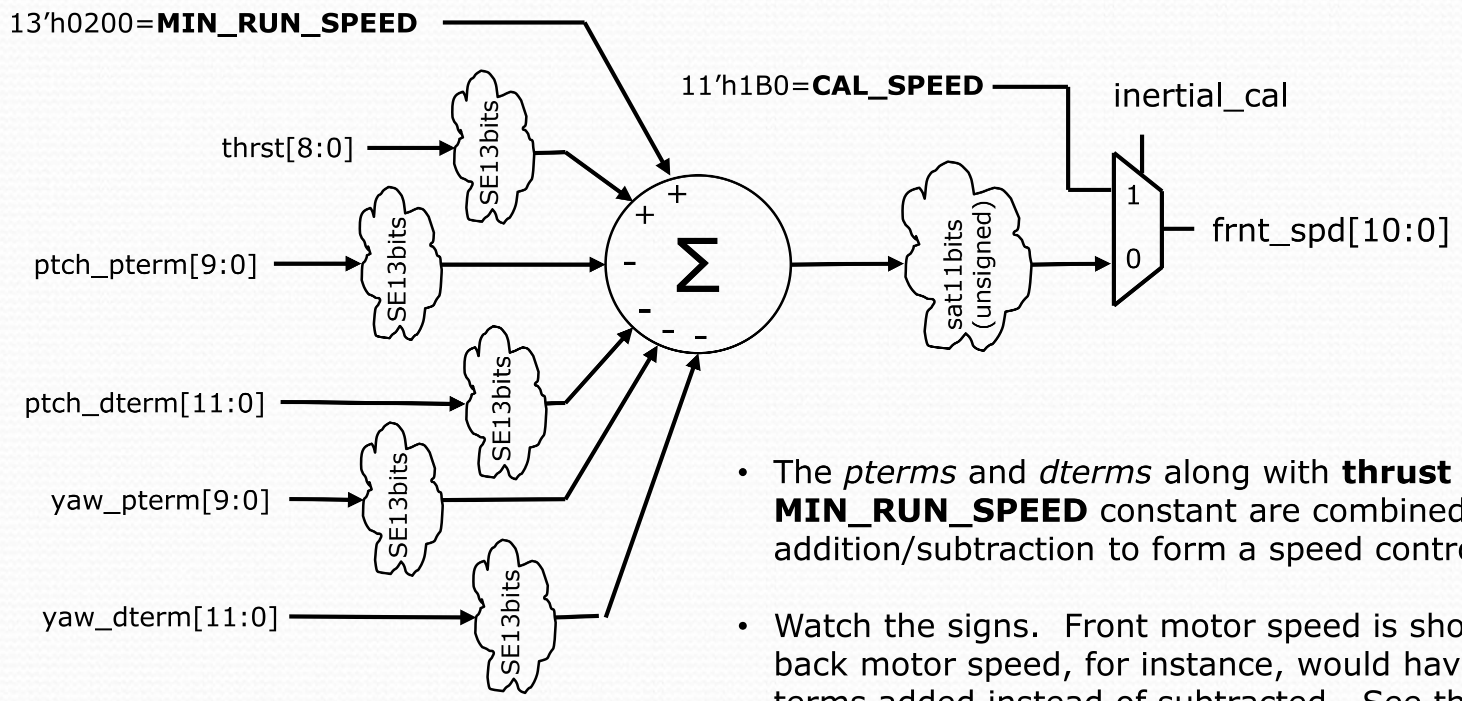
The following list is a verbal step by step of what is presented in the previous two slides. Compare it directly to the dataflow diagrams of the previous two slides and see if it "jives" in your mind.

1. Form error term of actual angle minus desired angle (i.e. **ptch – d_ptch**), make it 17-bits wide so it can't overflow.
2. Saturate the error term to 10-bits, call it: ***_err_sat** (i.e. **ptch_err_sat**)
3. Create ***_pterm** which is 5/8 of the saturated error term.
4. Create a queue of parametized depth and width of 10-bits. *(HINT: use a for loop)* This queue will store previous error terms. The queue should be reset to zero on rst_n and should shift on **vld**. The oldest queue entry will be subtracted from the current error to form ***_D_diff**.
5. Form ***_D_diff** which is the current saturated error minus the oldest queue entry. Could this overflow a 10-bit value? Not with any reasonable values we could possibly encounter during flight, so just keep it a 10-bit value.
6. Saturate ***_D_diff** to 6-bits in width (max pos value of 0x1F, max neg of 0x20)
8. Multiply the saturated *_D_diff 6-bit value by a 6-bit constant defined in a localparam. This should be a signed multiplier and will give a 12-bit product.

9. The above steps should be performed for ptch,roll, and yaw. Next will be the combining to form the motor speeds
10. We are adding several terms together. In order to not risk overflow we will bring the math up to 13-bits wide.
11. Sign extend and add/subtract the various terms to form the various calculated motor speeds. I called these intermediate terms **frnt_calc, bck_calc, lft_calc, rght_calc**.
12. Saturate these 13-bit values to 11-bits. This is an unsigned saturation. Motors on a quadcopter can't be driven backwards. Best you can do is stop the motor, so if the value is less than zero then clamp it at zero. If it is greater than 0x7FF then clamp it at 0x7FF.
13. Finally infer the mux to force the motor speed to CAL_SPEED during calibration. CAL_SPEED will be an 11-bit constant defined via a localparam.

(See the next slide for further clarification of steps 11-13)

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



- The *pterms* and *dterms* along with **thrust** and the **MIN_RUN_SPEED** constant are combined through addition/subtraction to form a speed control.
- Watch the signs. Front motor speed is shown here, but back motor speed, for instance, would have the **ptch** terms added instead of subtracted. See the project spec (around pg 39) for details on signs of the terms.
- The final saturation is an **unsigned** saturation. Motors cannot be driven backwards, so if the math results in a negative number we saturate to zero.
- During calibration of the inertial sensor motors all get set to a constant speed defined in localparam CAL_SPEED

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

(A skeleton **flght_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
inertial_cal	1	in	Indicates quadcopter is performing inertial cal, and motor speed should be set to CAL_SPEED
d_ptch, d_roll, d_yaw	[15:0]	in	Desired pitch, roll, and yaw. These are signed numbers
ptch, roll, yaw	[15:0]	in	Actual pitch, roll, and yaw from inertial interface unit
thrst	[8:0]	in	Overall thrust level from pilot input on slide potentiometer. This is unsigned number.
frnt_spd, bck_spd, lft_spd, rght_spd	[10:0]	out	Motor speeds expressed as 11-bit unsigned numbers. These goes to ESCs to form eventual motor control signal.

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

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

- We will perform two tests on flght_cntrl math.
  - The first test (**flght_cntrl_dbg_tb.v**) is a functional test that is easier to debug, but not complete (only really tests ptch related math in any detail).
  - 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 (**flght_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.
- **flght_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. You could still have copy/paste errors in your roll/yaw code, or sign problems in your construction of lft_spd/rght_spd

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

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

- For **flght_cntrl_stim.hex** the vector is 108 bits wide and is assigned as follows:

Stimulus Bit Range:	Signal Assignment:
stim[107]	rst_n
stim[106]	vld
stim[105]	inertial_cal
stim[104:89]	d_ptch[15:0]
stim[88:73]	d_roll[15:0]
stim[72:57]	d_yaw[15:0]
stim[56:41]	ptch[15:0]
stim[40:25]	roll[15:0]
stim[24:9]	yaw[15:0]
stim[8:0]	thrst[8:0]

- For **flght_cntrl_resp.hex** the vector is 44 bits wide and is assigned as follows:

Stimulus Bit Range:	Signal Assignment:
resp[43:33]	frnt_spd
resp[32:22]	bck_spd
resp[21:11]	lft_spd
resp[10:0]	rght_spd

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

- 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?

flght_cntrl_chk_tb.v

Submit **flght_cntrl.sv**, **flght_cntrl_chk_tb.v** to the dropbox

Exercise 14 flght_cntrl_chk_tb.v Hints:

- Instantiate DUT (flght_cntrl.sv) connecting all the inputs to the respective bits of a 108-bit wide vector of type reg.
- Declare a “memory” of type reg that is 108 bits wide and has 1000 entries. This is your stimulus memory
- Declare a “memory” of type reg that is 44 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 #6 so you are #1 after **clk** rise. Now check, do DUT outputs match the respective bits of the response vector?