ECE 551

HW4 *(100 pts)*

- Due Weds Nov 8th @ 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 UART

- In HW3 you produced a UART transmitter (UART_tx.sv) and a UART receiver (UART_rx.sv). Combine the two module (by simply instantiating each) into a UART transceiver (UART.sv).

- In this HW you will synthesize both modules using Synopsys on the CAE linux machines.

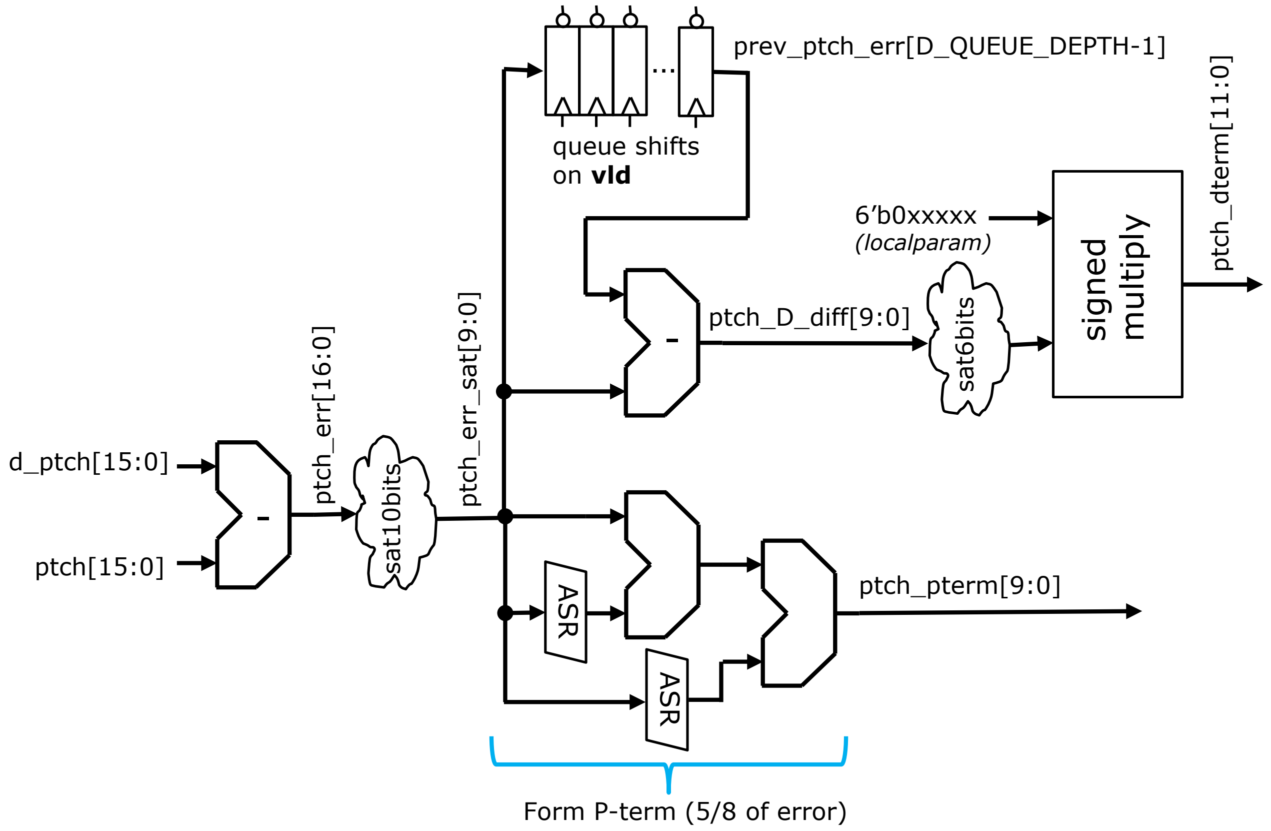
- (**NOTE:** if your UART modules are a big ball of stink, and one of your project partners has a better versions you can use their code. You still have to do the synthesis individually)

- Write a synthesis script to synthesize your UART.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 0.5 ns on all inputs other than clock
    - Defines a drive strength equivalent to a 2-input nand of size 2 from the TSMC library (ND2D2BWP) for all inputs except clock and rst_n
    - Defines an output delay of 0.75ns on all outputs.
    - Defines a 0.15pf load on all outputs.
    - Sets a max transition time of 0.15ns 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 (UART.dc)
    - The output reports for area (UART_area.txt)
    - The gate level verilog netlist (UART.vg)

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

- 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**



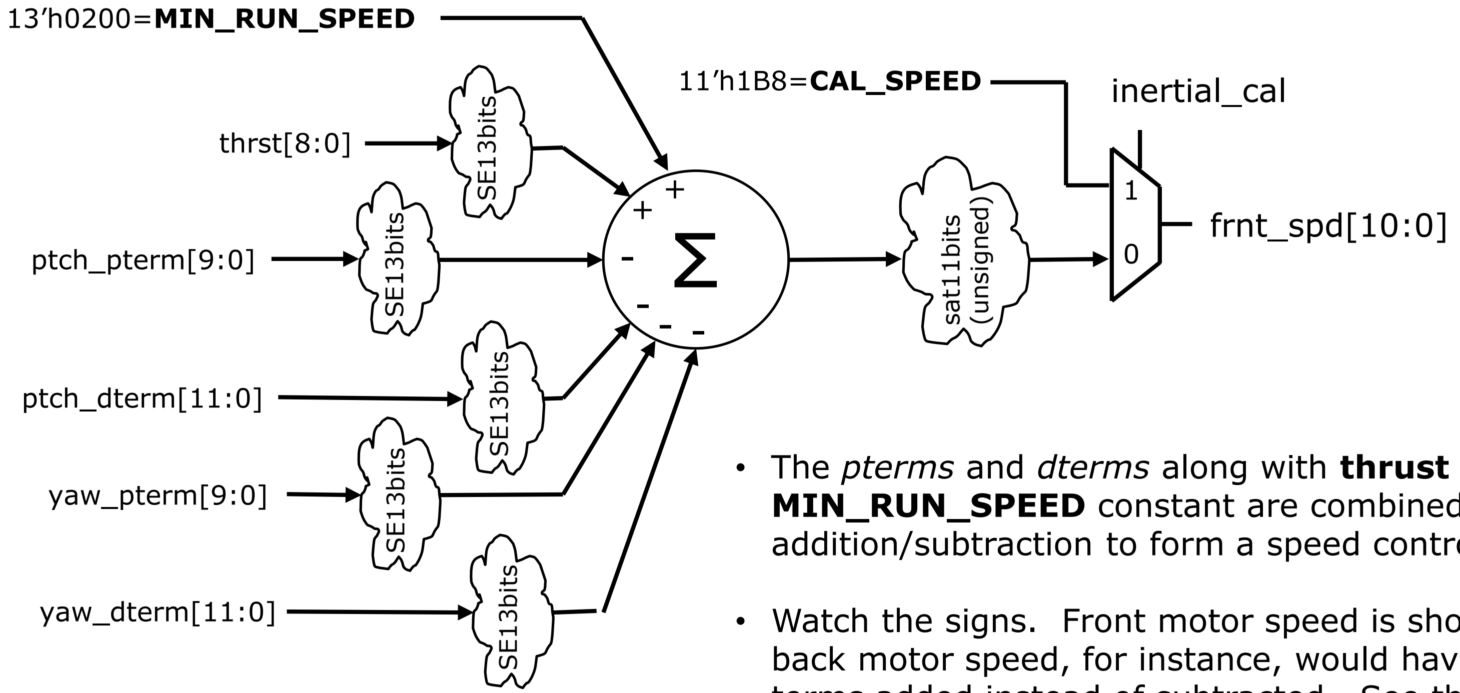
- Shown is a datapath for calculating the **P** and **D** terms for pitch. Roll and Yaw have identical math

- 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.

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



- 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 37) 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

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

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.

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

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.

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

- 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

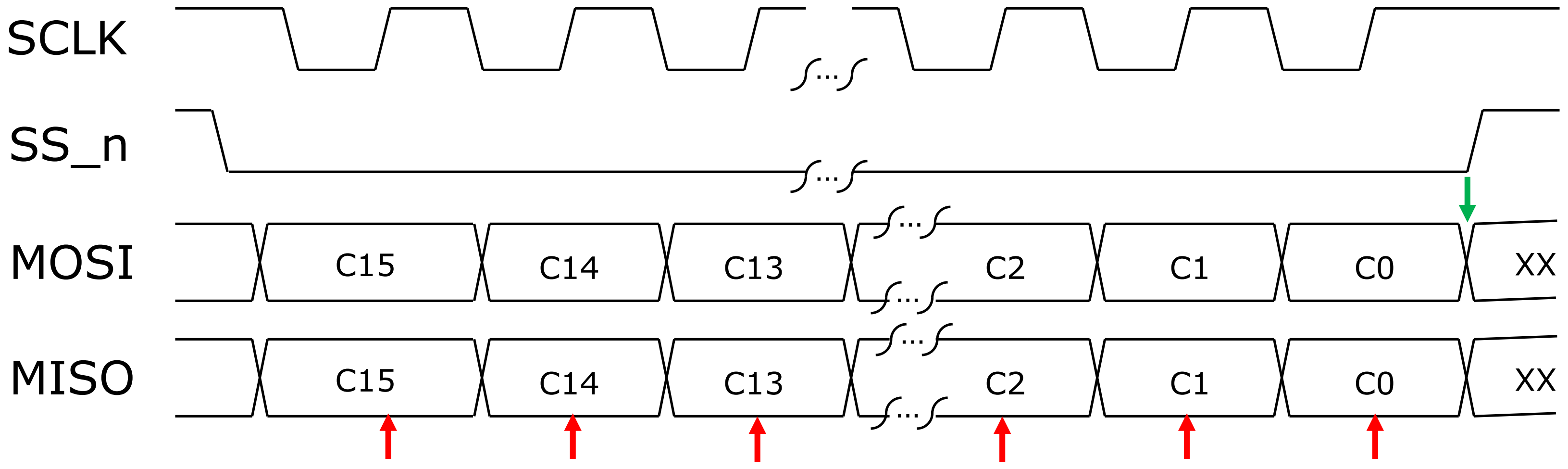
Submit **flght_cntrl.sv**, **flght_cntrl_chk_tb.v** to the dropbox

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).

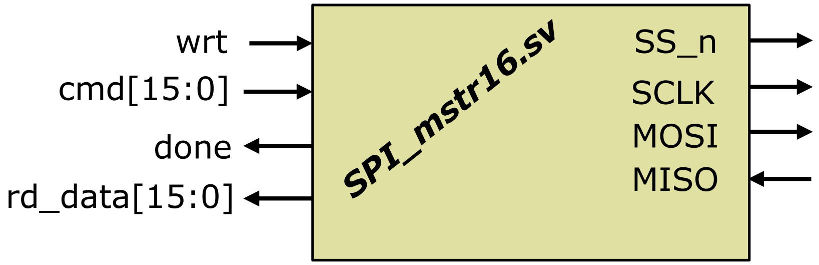
The sampled version of **MISO** in turn gets shifted into our 16-bit shift register. (on **SCLK** fall)

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**)