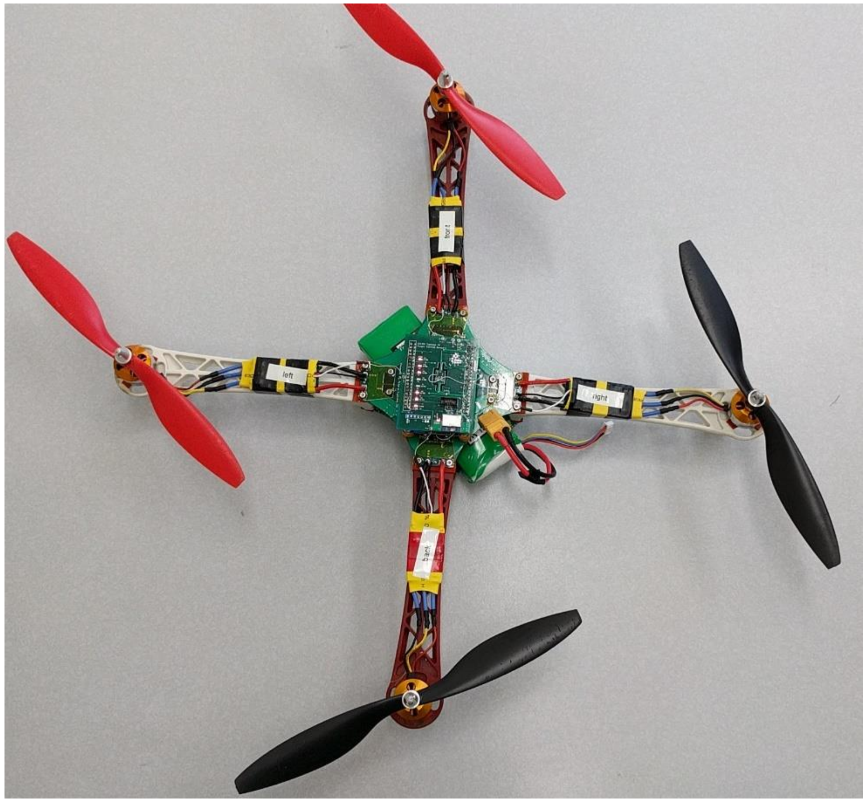
ECE 551
Project Spec

Spring '21

Cyclone IV

(get it?)

The "brains" are an Altera Cyclone IV and it has 4 powerful motors connected to 10" props. Gotta love the double entendre



Grading Criteria: (Project is 28% of final grade)

- Project Grading Criteria:
  - Quantitative Element 12.5%

  (yes this could result in extra credit)

$$Quantitative = \frac{Eric\_ProjectArea}{YourSynthesizedArea}$$

Note: The design has to be functionally correct for this to apply

  - Project Demo (87.5%)
    - ✓ Code Review (12.5%)
    - ✓ Testbench Method/Completeness (17.5%)
    - ✓ Synthesis Script review (7.5%)
    - ✓ Post-synthesis Test run results (10%)
    - ✓ Results when placed in EricJohn Testbench (25%) (this is quantitative too: (number of tests passed)/(number of tests)
    - ✓ Test of your code mapped to the actual quad copter and tested on test jigs. (15%)

Extra Credit Opportunity:

Appendix C of ModelSim tutorial instructs you how to run code coverage

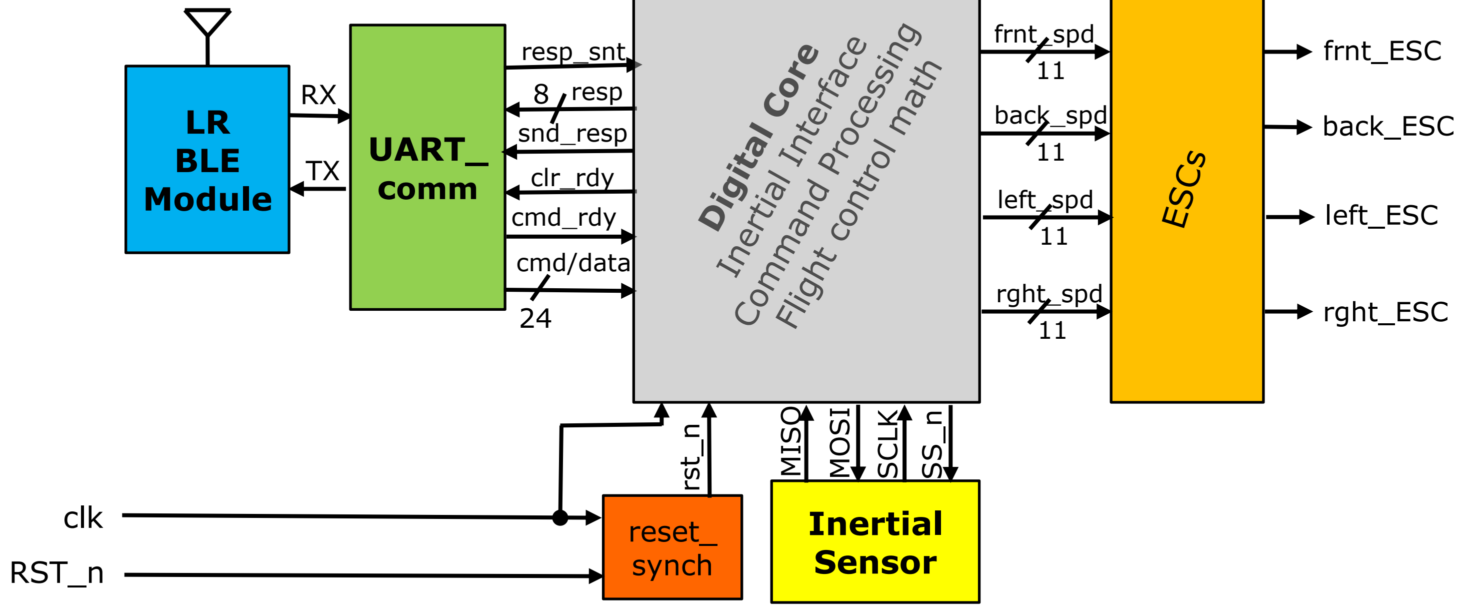
- Run code coverage on a single test and get 1% extra credit
- Run code coverage across your test suite and get a cumulative number and get 2% extra credit.
- Run code coverage across your test suite and give concrete example of how you used the results to improve your test suite and get 2.5% extra credit.

Project Due Date

- Project Dropbox Submission Due:
  - Friday Apr 30th evening.

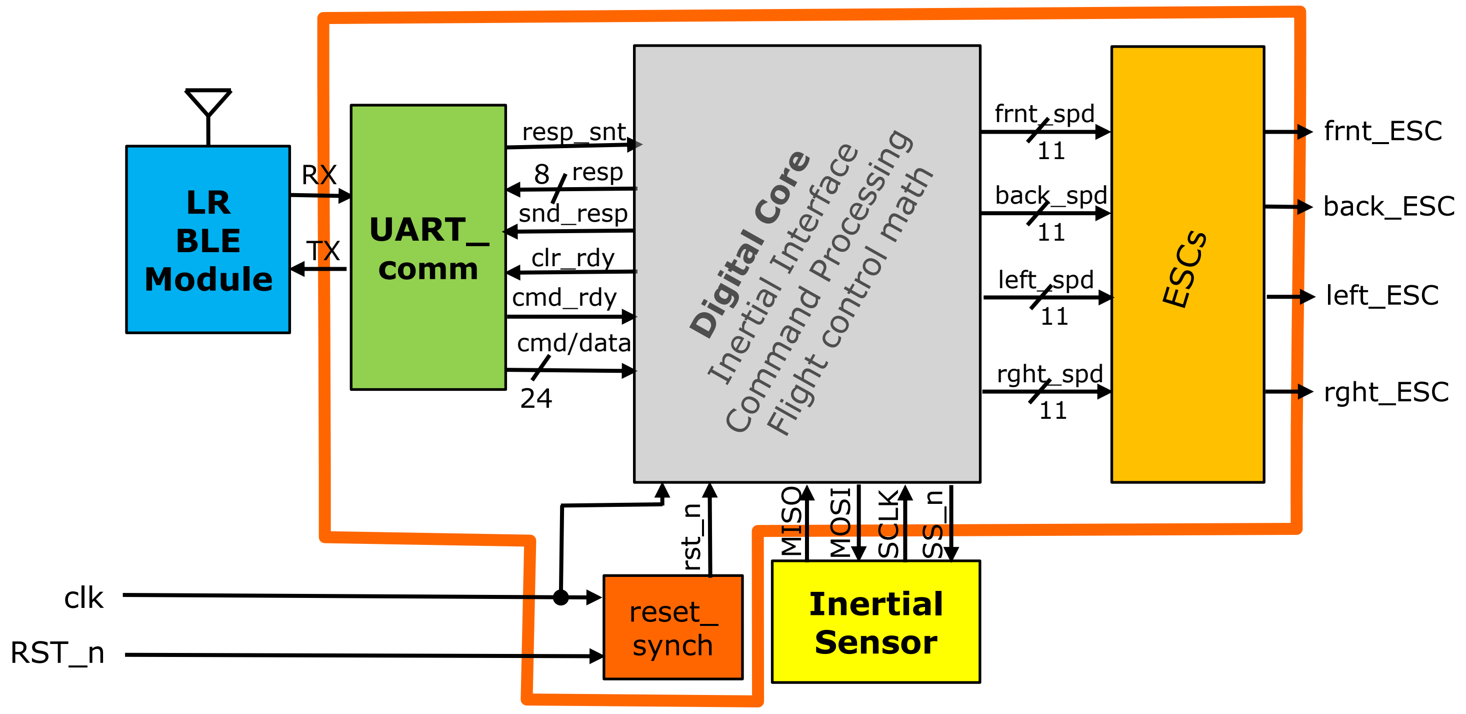
- Project Grade Involves:
  - ✓Code Review
  - ✓ Testbench Method/Completeness
  - ✓ Synthesis Script & Results review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in EricJohn testbench
  - ✓ Results when tested on demo platform (a constrained quadcopter) (sorry, can't let you fly it for real…too dangerous)

Block Diagram



The quadcopter will receive pitch, roll, yaw, and thrust commands wirelessly via a long range Bluetooth Low Energy module. The received pitch, roll, yaw represent the desired orientation of the quadcopter. We will also interface with a 6-Axis (3-axis accelerometer, 3-axis gyro) inertial sensor. The readings will be combined to obtain the actual pitch, roll, and yaw of the quadcopter. The difference between desired orientation and actual orientation will form the error term feeding into a PD control loop (implemented in digital domain of course). The result of these calculations will determine the drive speed of the 4 motors needed to keep the copter at the desired orientation.

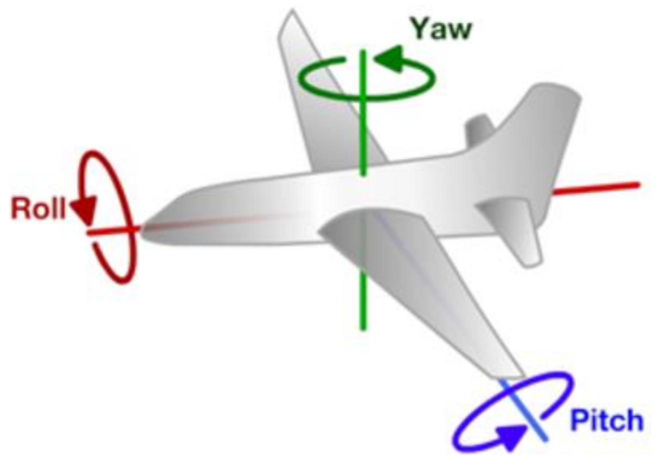
What is synthesized DUT vs modeled?



The blocks outlined in red above are pure digital blocks, and will be coded with the intent of being synthesized.

You Must have a block called **QuadCopter.sv** which is top level of what will be the synthesized. (what is outlined in red).

What is Pitch, Roll, & Yaw?

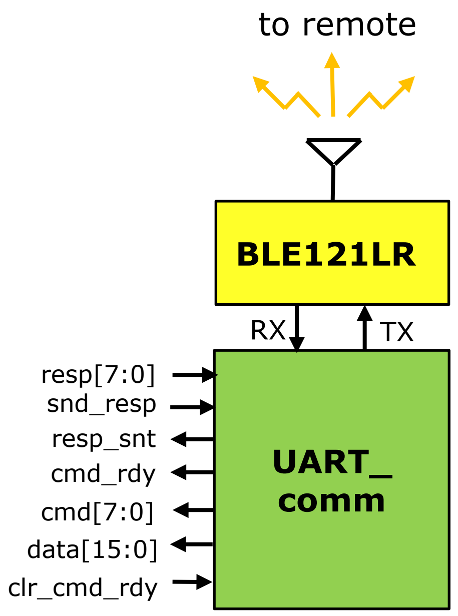


- This document references pitch, roll, and yaw frequently so you need to understand what they are.
- The diagram above shows an airplane, but the same concept applies to a quadcopter.
- In the end this entire project boils down to controlling the speed of 4 motors to control pitch, roll, and yaw.

UART Interface

- A UART serial interface will be used to convey commands via the blue tooth module (BLE121LR), and to send battery data back to the BLE121LR. The BLE121LR in turn wirelessly communicates to the remote control device.
- All packets from the UART Interface are 24-bits, and can be considered to be 8-bits of command and 16-bits of data. All commands received are responded to with a single byte of data. The response for all commands is a positive acknowledge (0xA5).

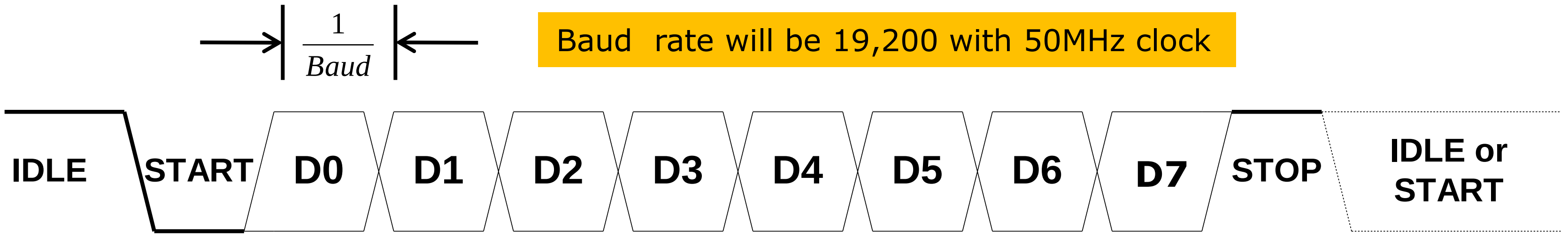
Signal:	Dir:	Description
clk,rst_n	in	Clock and reset. 50MHz system clock, and active low reset
resp[7:0]	in	Response to controller. Either 0xA5 or upper 8-bit of batt reading
snd_resp	in	This signal is pulsed high for at least one clock cycle to start transmission of response
resp_snt	out	Asserted when ***resp[7:0]*** has finished transmitting
cmd_rdy	out	Asserted when a new 24-bit command has been received from the remote
cmd[7:0]	out	Upper 8-bits of 24-bit command to copter, represent command byte
data[15:0]	out	Lower 2-bytes of 24-bit command. A desired ptch, roll, yaw or thrust
clr_cmd_rdy	in	Asserted by core when it has ‘digested” the command.
RX/ TX	in/ out	Serial data out (to BG121LR, then to remote) Serial data in (from BG121LR, from remote initially)



What is UART (RS-232)

- RS-232 signal phases
  - Idle
  - Start bit
  - Data (8-data for our project)
  - Parity (no parity for our project)
  - Stop bit – channel returns to idle condition
  - Idle or Start next frame

Baud rate will be 19,200 with 50MHz clock



- Receiver monitors for falling edge of Start bit. Counts off 1.5 bit times and starts shifting (right shifting since LSB is first) data into a register.

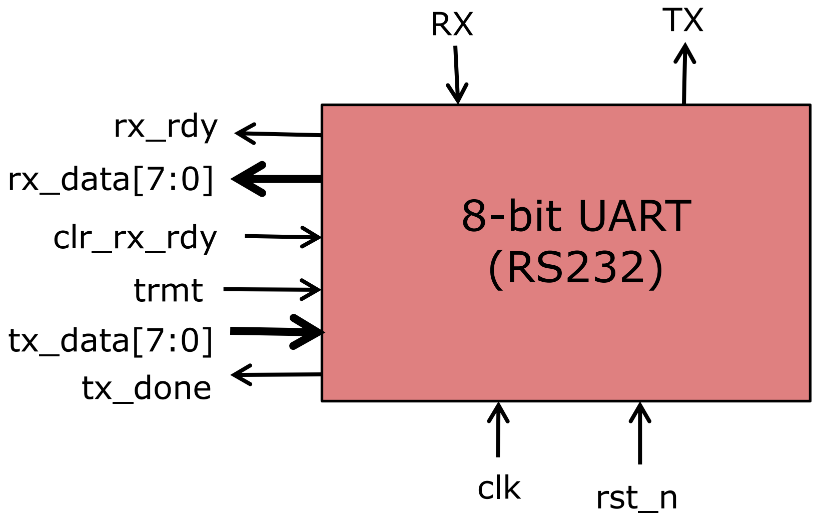
- Transmitter sits idle till told to transmit. Then will shift out a 9-bit (start bit appended) register at the baud rate interval.

What is RS232 (UART)

- Full RS-232 involves handshaking signals
  - RTS, CTS, DTR, …
  - We will only use RX & TX assuming sender and receiver can "digest" bytes fast enough. Common use.

- When 8-bit data received UART will raise ***rdy*** and data will be presented on ***rx_data***[7:0].

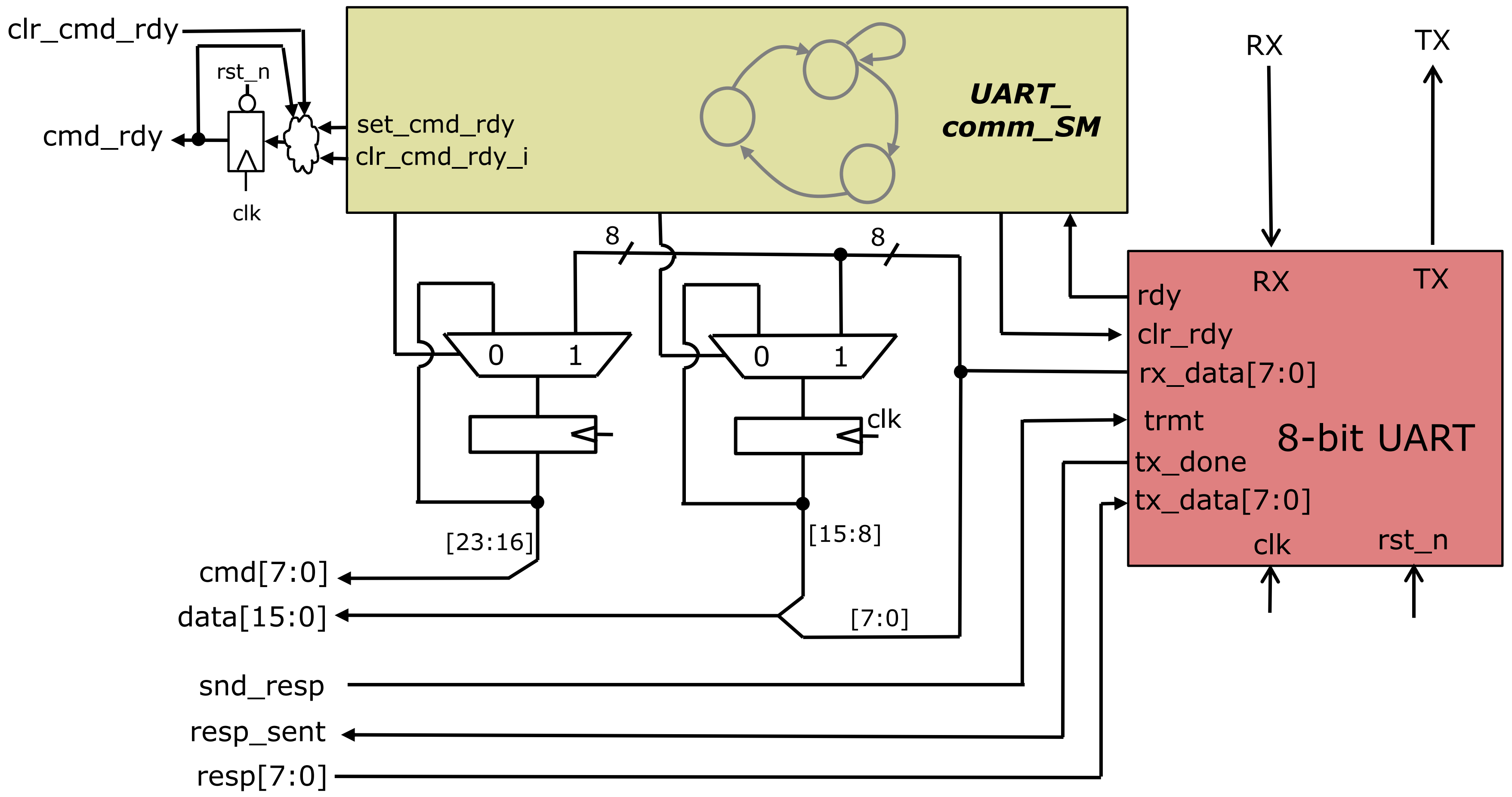
- When core has consumed data it will raise ***clr_rdy***, to indicate to the UART it should drop ***rdy***.



- To transmit the core will be present data on ***tx_data***[7:0] and then raise ***trmt***. When the UART is done transmitting it will raise ***tx_done***.

UART_comm

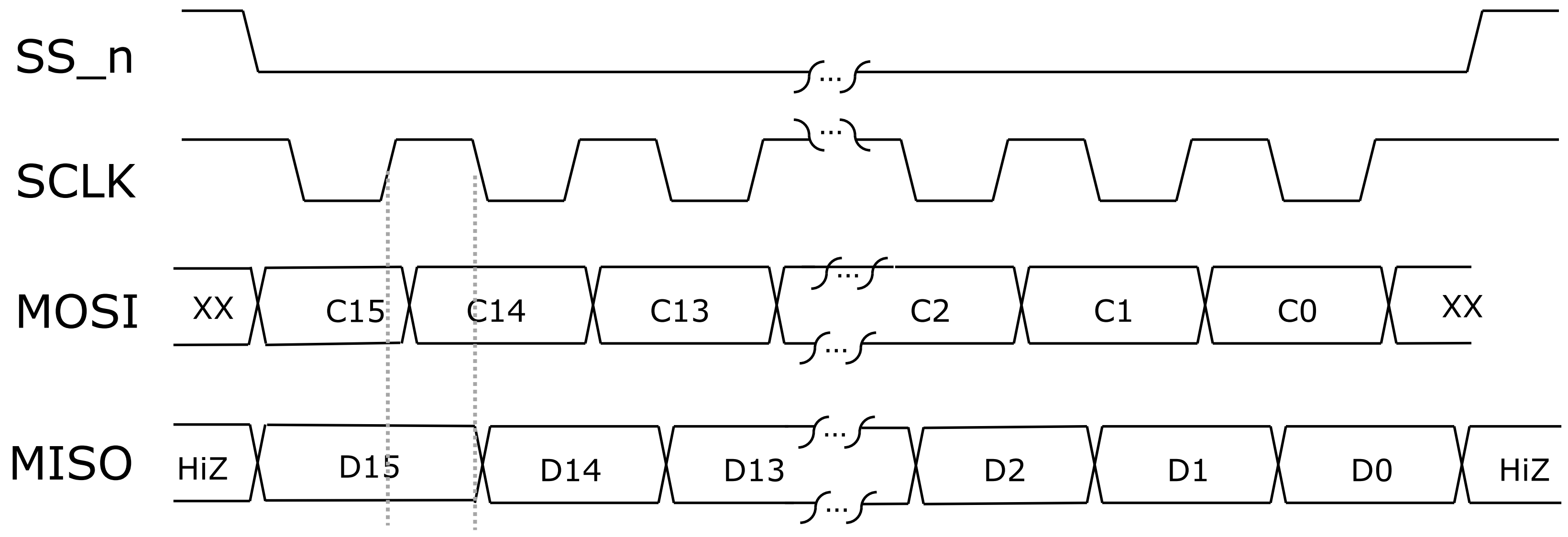
- Previous slides showed an 8-bit UART transceiver. We want to put a wrapper around it to make a unit that receives 24-bit packets and transmits 8-bit packets.



What is SPI?

- Simple Monarch/Serf serial interface (Motorola long long ago)
  - **S**erial **P**eripheral **I**nterconnect (very popular physical interface)
  - 4-wires for full duplex
    - ✓ MOSI (Monarch Out Serf In) (We drive this to 6-axis inertial)
    - ✓ MISO (Monarch In Serf Out) (Inertial sensor drives this back to us)
    - ✓ SCLK (Serial Clock)
    - ✓ SS_n (Active low Serf Select) (For us we only have one SS per SPI channel)
  - 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 slightly after SCLK rise (2 system clocks after)
    - ✓ MISO sampled at that same time.

SPI Packets



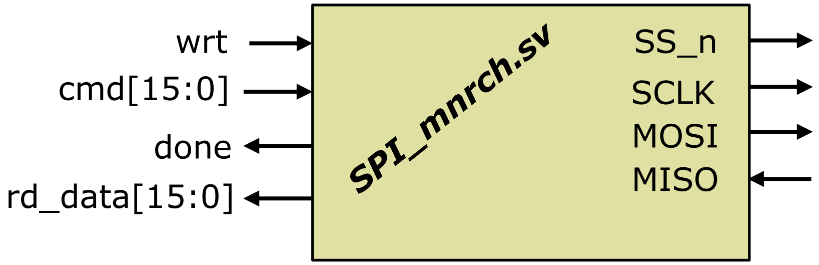
A SPI packet inherently involves a send and receive (full duplex). The full duplex packet is always initiated by the monarch. The monarch controls SCLK, SS_n, and MOSI. The serf drives MISO if it is selected. If the serf is not selected it should leave MISO high impedance. The inertial sensor is the only SPI peripheral we have in the system.

The SPI master will have a 16-bit shift register. The MSB of this shift register is MOSI. MISO will feed into the LSB of this shift register. The shift register should shift **two system clocks after** the rise of SCLK, this eliminates any timing difficulties. The inertial sensor samples MOSI on the positive edge of SCLK, and changes MISO on the negative edge of SCLK. Of course all your flops are based purely on clk (system clock), not SCLK! SCLK is a signal output from your SPI master.

SCLK will be 1/16 of our system clock (50MHz/16 = 3.125MHz

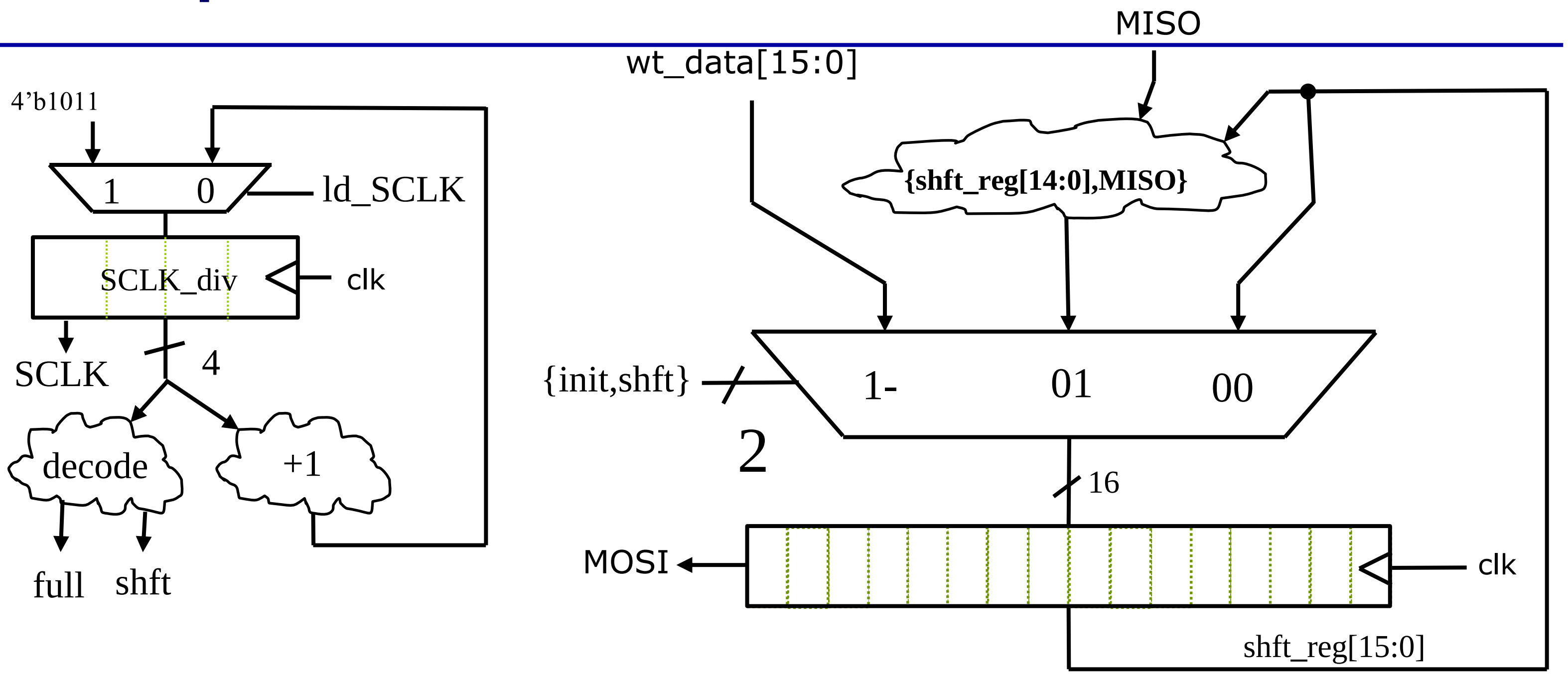
SPI Unit for Inertial Interface & A2D

- The 6-axis inertial sensor on the board can be read with a SPI monarch that implements the 16-bit SPI transaction mentioned above.
- You will implement **SPI_mstr16.sv** with the interface shown.
- SCLK frequency will be 1/16 of the 50MHz clock (i.e. it comes from the MSB of a 4-bit counter running off clk)
- Remember you are producing **SCLK** from the MSB of a 4-bit counter. So for example, when that 4-bit counter equals 4'b0111 you know **SCLK** rise happens on the next clk. Perhaps more pertinent…when that 4-bit counter equals 4'b1001 you should enable the shift register because you would then force a sample of **MISO** into the LSB of the shift register at two system clocks after **SCLK** rise.



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]

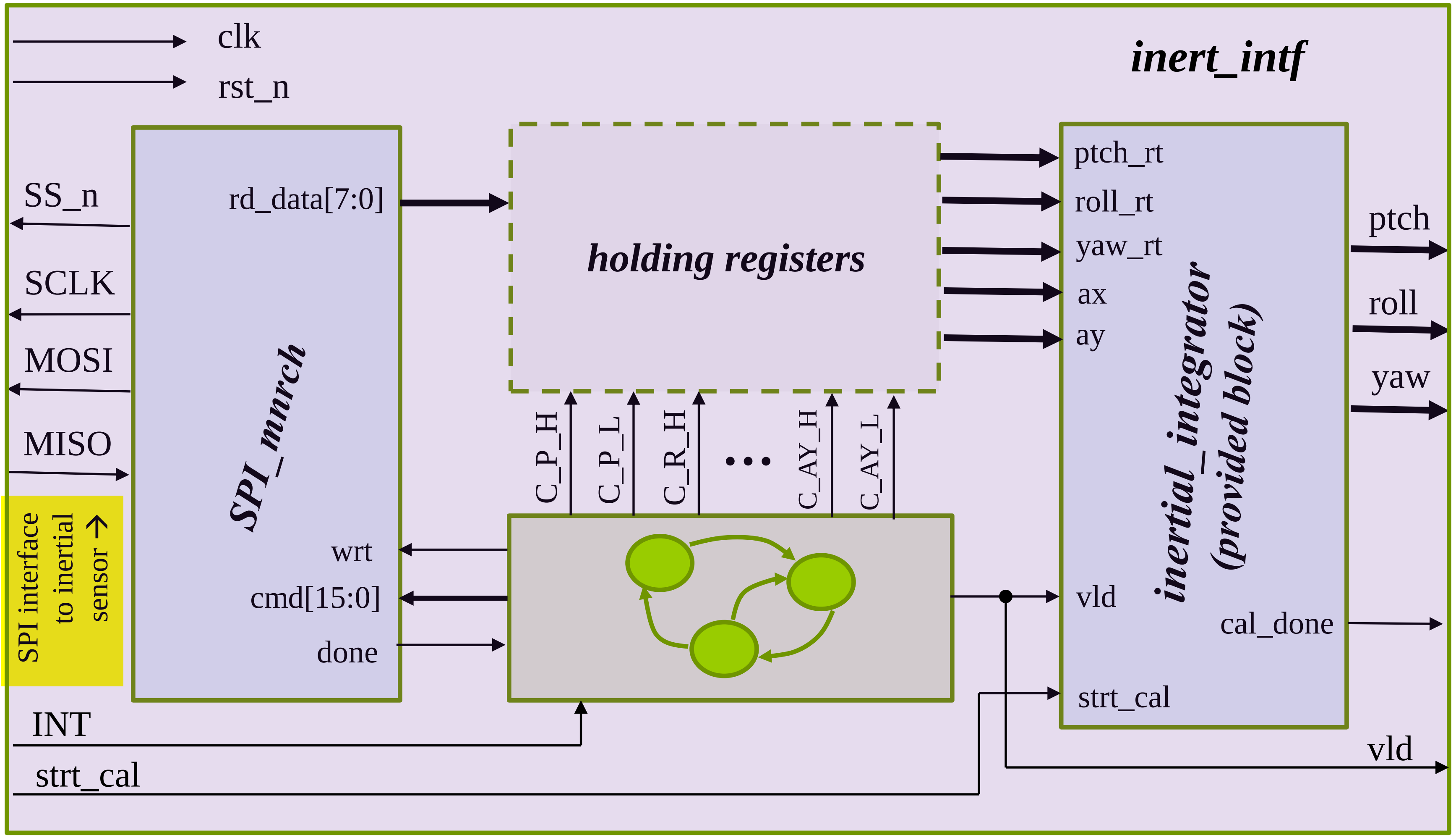
SPI Implementation



The main datapath of the SPI monarch consists of a 16-bit shift register. The MSB of this shift register provides **MOSI**. The shift register can be parallel loaded with the data to send, or it can left shift one position taking **MISO** as the new LSB, or it can simply maintain.

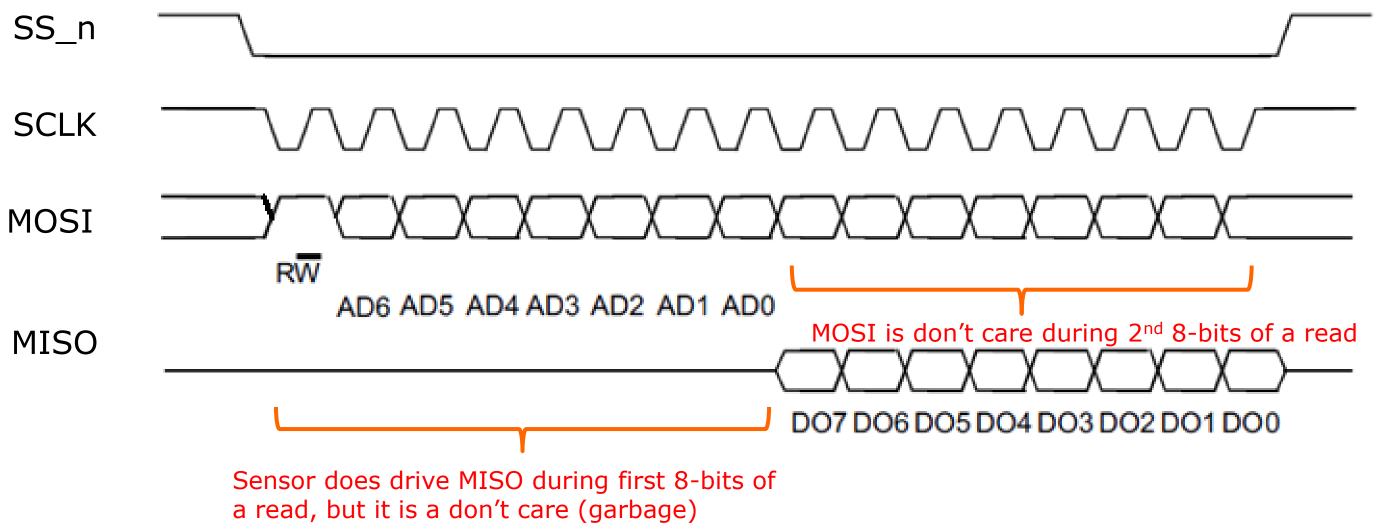
Since the SPI monarch is also generating **SCLK** it can choose to shift this register in any relationship to **SCLK** that it desires. To alleviate timing difficulties it is best that the shift register is shifted two system clocks after **SCLK** rise. Note the value SCLK_div is loaded with (4’b1011). Look back at the waveforms. There is a little time from when **SS_n** falls till the first fall of **SCLK**. Do you get the idea of loading with 4’b1011?

Inertial Sensor Interface



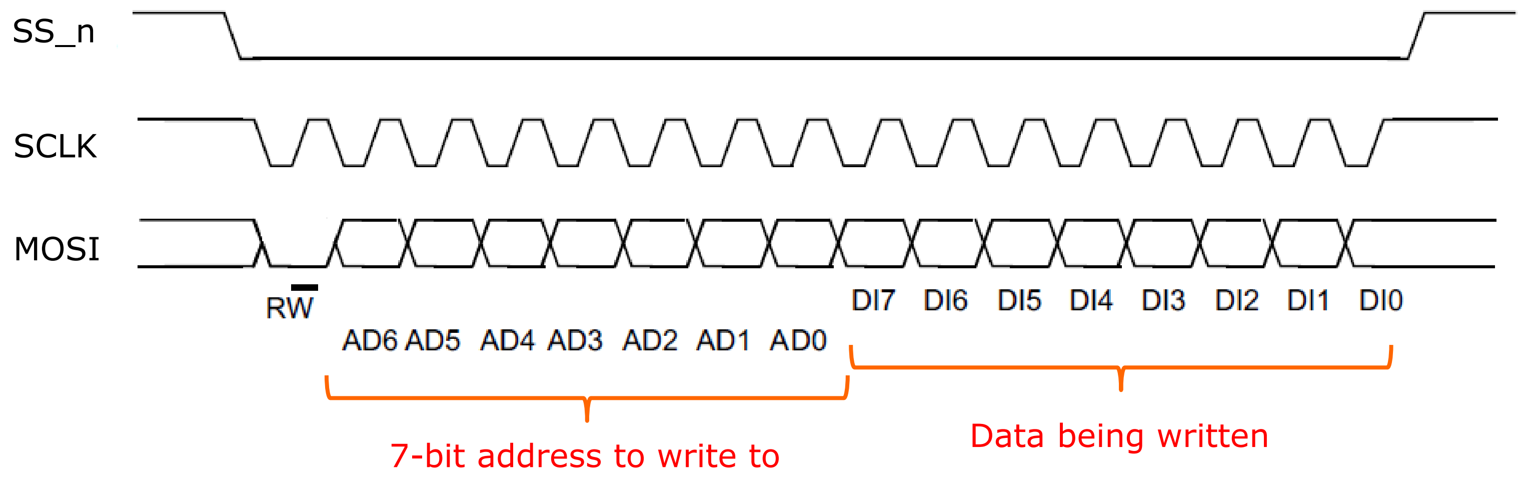
Inertial Sensor Interface

- The inertial Sensor is configured and read via a SPI interface
  - Unlike the A2D which requires two 16-bit transactions to complete a single conversion with the inertial sensor reads/writes are accomplished with single 16-bit transaction.
  - For the first 8-bits of the SPI transaction, the sensor is looking at MOSI to see what register is being read/written. The MSB is a $R/\overline{W}$ bit, and the next 7-bits comprise the address of the register being read or written.
  - If it is a read the data at the requested register will be returned on MISO during the 2nd 8-bits of the SPI transaction (see waveforms below for read)


SS_n
SCLK
MOSI
MISO
$R\overline{W}$
AD6 AD5 AD4 AD3 AD2 AD1 AD0
MOSI is don't care during 2nd 8-bits of a read
DO7 DO6 DO5 DO4 DO3 DO2 DO1 DO0
Sensor does drive MISO during first 8-bits of a read, but it is a don't care (garbage)

Inertial Sensor Interface (write)

- During a write to the inertial sensor the first 8-bits specify it is a write and the address of the register being written. The 2nd 8-bits specify the data being written. (see diagram below)


SS_n
SCLK
MOSI
$R\overline{W}$
AD6 AD5 AD4 AD3 AD2 AD1 AD0
DI7 DI6 DI5 DI4 DI3 DI2 DI1 DI0
7-bit address to write to
Data being written

- Of course the sensor is returning data on MISO during this transaction, but this data is garbage and can be ignored.

Initializing Inertial Sensor

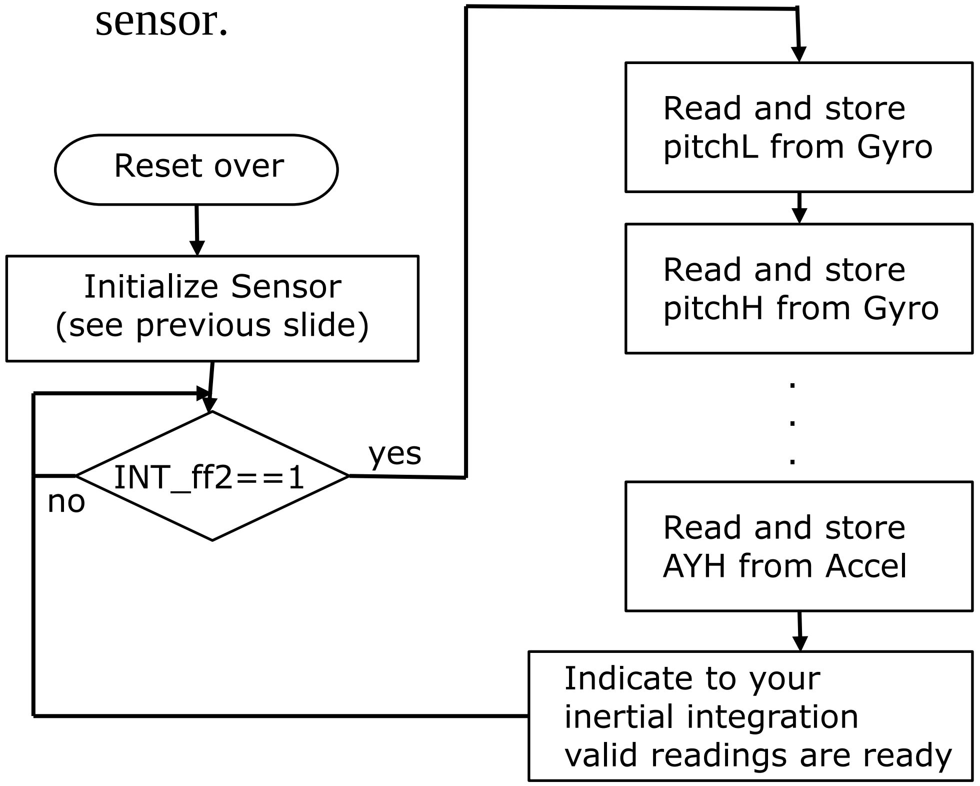
- After reset de-asserts the system must write to some registers to configure the inertial sensor to operate in the mode we wish. The table below specifies the writes to perform.

Addr/Data to write:	Description
0x0D02	Enable interrupt upon data ready
0x1062	Setup accel for 416Hz data rate, +/- 2g accel range, 100Hz LPF
0x1162	Setup gyro for 416Hz data rate, +/- 125°/sec range.
0x1460	Turn rounding on for both accel and gyro

- You will need a state-machine to control communications with the inertial sensor. Obviously we are also reading the inertial sensor constantly during normal operation. The initialization table above just specifies what some of your first states in that state-machine are doing.

Reading Inertial Sensor

- After initialization of the inertial sensor is complete the inertial interface state-machine should go into an infinite loop of reading gyro and accel data.
- The sensor provides an active high interrupt (INT) that tells when new data is ready. Double flop that signal (for meta-stability reasons) and use the double flopped version to initiate a sequence of reads (10 reads in all) from the inertial sensor.


Reset over
Initialize Sensor (see previous slide)
INT_ff2==1
yes
no
Read and store pitchL from Gyro
Read and store pitchH from Gyro
.
.
.
Read and store AYH from Accel
Indicate to your inertial integration valid readings are ready

- You will have 10 8-bit flops to store the 10 needed reading from the inertial sensor. These are: pitchL, pitchH, rollL, rollH, yawL, yawH, AXL, AXH, AYL, AYH. We don't need AZ for our purposes.

Reading Inertial Sensor (continued)

- The table below specifies the addresses you need to use to read inertial data. Recall for a read the lower byte of the 16-bit packet is a don’t care.

Addr/Data:	Description:
0xA2xx	pitchL ➔ pitch rate low from gyro
0xA3xx	pitchH → pitch rate high from gyro
0xA4xx	rollL ➔ roll rate low from gyro
0xA5xx	rollH ➔ roll rate high from gyro
0xA6xx	yawL ➔ yaw rate low from gyro
0xA7xx	yawH ➔ yaw rate high from gyro
0xA8xx	AXL ➔ Acceleration in X low byte
0xA9xx	AXH ➔ Acceleration in X high byte
0xAAxx	AYL ➔ Acceleration in Y low byte
0xABxx	AYH ➔ Acceleration in Y high byte

Gyro Reading Integration (and calibration)

- The Pitch, Roll, and Yaw readings we obtain from the gyro are not angles, but rather angular rates (we get °/sec). Therefore we have to integrate to get degrees of rotation about each axis.
- Don’t worry, integration simply means summing over time. So we simply need an accumulator register for each axis that sums in the signed angular rate readings we are getting.
- Lets imagine this for the yaw axis (rotation of the quadcopter in a flat plane).
- Imagine the quadcopter was just hovering level, and not yawing. The yaw angular rate readings would be small negative and small positive numbers, and would on average sum to zero since the copter is not rotating in a flat plane.
- As awesome as the inertial sensor is (and it is pretty freaking amazing) it is not that good. Just the act of soldering an inertial sensor to a board will affect the offsets of its gyro readings. So even if it was perfectly factory compensated, it will not be once it is in your application.
- There will have to be a calibration phase where we take a bunch (2048 in our case) of readings (when we know the quadcopter is not moving) and integrate them to discover an offset term that we will apply to all future gyro readings. Of course this is done for all 3-axis (pitch, roll, and yaw).

inertial_integrator.sv (at least that's what I called this unit)

Signal:	Dir:	Description:
clk, rst_n	In	clock & reset
strt_cal	In	From ***cmd_cfg*** unit. Will go high for 1 clock cycle indicating integration of gyro readings to determine offsets should commence.
cal_done	Out	To ***cmd_cfg*** unit. Let cmd config unit know that calibration is over.
vld	In	High for a single clock cycle when new inertial readings are valid.
ptch_rt[15:0]	In	16-bit signed raw pitch rate from inertial sensor
roll_rt[15:0]	In	16-bit signed raw roll rate from inertial sensor
yaw_rt[15:0]	In	16-bit signed raw yaw rate from inertial sensor
ax [15:0]	In	Will be used for sensor fusion (roll)
ay [15:0]	In	Will be used for sensor fusion (pitch)
ptch[15:0]	out	Fully compensated (both offset and fusion) 16-bit signed pitch.
roll[15:0]	out	Fully compensated (both offset and fusion) 16-bit signed roll
yaw[15:0]	out	Offset compensated 16-bit signed yaw. A really good system would use a magnetometer to perform fusion for long term yaw correction, but this project is complex enough

inertial_integrator.sv (needed internal registers & vectors)

Register:	**Purpose:**
state[1:0]	You will need 3 or 4 states in your SM to control this unit
ptch_int[26:0]	Pitch integrating accumulator. This is register ***ptch_rt*** is summed into
roll_int[26:0]	Roll integrating accumulator. This is the register ***roll_rt*** is summed into
yaw_int[26:0]	Yaw integrating accumulator. This is the register ***yaw_rt*** is summed into
ptch_off[15:0], roll_off[15:0], yaw_off[15:0]	Holds offset term calculated during calibration that is subtracted from all future ***ptch_rt/roll_rt/yaw_rt*** readings.
smpl_cntr[10:0]	Counter capable of counting 2048 samples
ax_accum[19:0], ay_accum[19:0]	Accumulators used for averaging 16 samples of **ax** and **ay** to form ax_avg and ay_avg used in fusion calculations

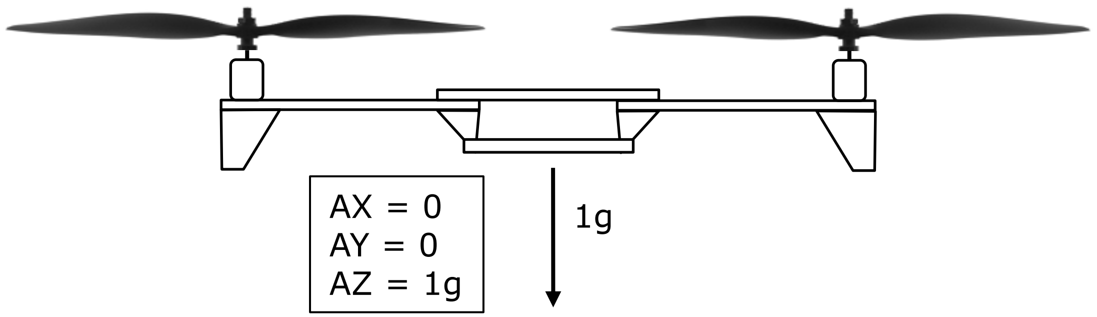
Vector: (wire)	**Purpose:**
ptch_comp[15:0], roll_comp[15:0], yaw_comp[15:0]	During calibration this is the raw rate signal. During normal operation this is the offset compensated rate signal (i.e. ptch_rt – ptch_off)
Others	There are other internal intermediate vectors needed in this unit, but they are related to sensor fusion and I have not covered that yet.

inertial_integrator.sv (overview of function)

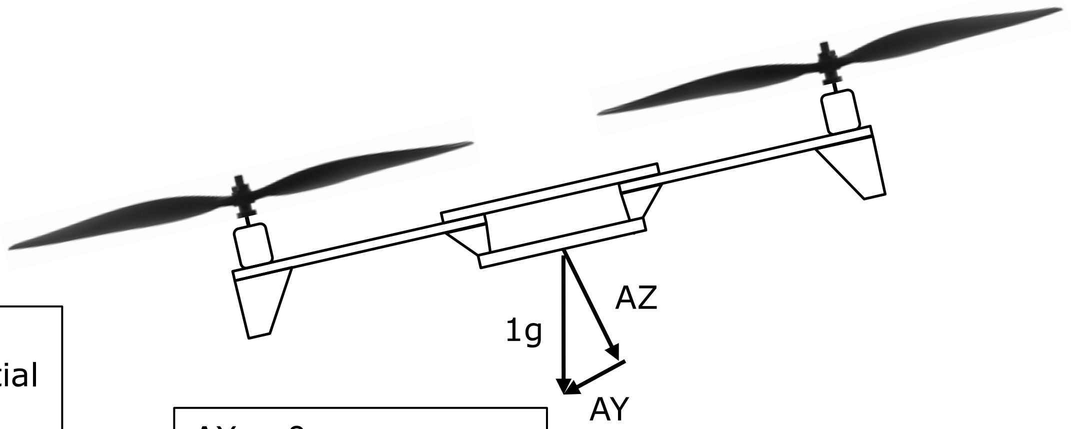
- Sits in IDLE doing nothing (keeps its inertial integrators zeroed) until it receives a ***strt_cal*** signal from ***cmd_cfg***.
- During CALIBRATION it accumulates the raw rate readings (i.e sign extended ***ptch_rt***) in its three integrating accumulators for the next 2048 samples.
- When calibration is complete it will do three things
  - Indicate ***cal_done*** to ***cmd_cfg***
  - Capture bits [26:11] of the integrating accumulators into the 16-bit offset registers. **NOTE:** We will later be making the number of samples a parameter. Bits [26:11] works for 2048 samples, What works for 8 samples?
  - Clear the integrating accumulators.
- During FLIGHT operation this unit will integrate the compensated rate signals, along with a correction factor that comes from sensor fusion (more on this next later)
- Sign extend bits [26:13] of each integrating accumulator to form the respective 16-bit ***ptch, roll, yaw*** signals output signals. Why bits [26:13]? Because I say so.

What is Sensor Fusion?

- One can get pitch and roll from an accelerometer. The earth's gravity always provides one **g** of acceleration straight down.



AX = 0
AY = 0
AZ = 1g



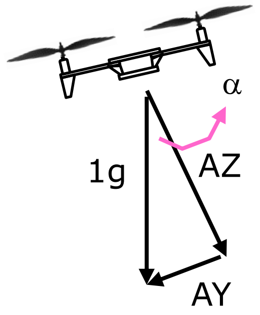
So why get pitch and roll
From the gyro? We have a 6-axis inertial sensor so we have AX, AY, and AZ too

AX = 0
AY = sin(pitch)*1g
AZ = cos(pitch)*1g
Pitch = $\tan^{-1}$(AY/AZ)

What is Sensor Fusion?

- Accelerometers are noisy, and Gyros are subject to long term drift.
    - Accelerometer readings are noisy (especially on a quadcopter with a bunch of not well balanced plastic props rotating at wicked speeds).
    - Gyro readings are amazingly quiet (even in presence of vibration) but since we are integrating any remaining offset error will result in a long term drift.
- Sensor fusion is an attempt to take the best from each to create a low noise, no drift version of pitch and yaw.
    - The fusion data is dominated by the integrated gyro readings, thus it stays low noise.
    - However, pitch and roll are also calculated via accel readings, and the long term average of the integrated gyro readings is “leaked” toward the accelerometer results. In electrical engineering terms the accelerometer gets to establish the DC operating point, but the gyro readings determine the transient response.
    - An accelerometer cannot determine yaw. In an ideal world we would also have a 2-axis magnetometer (electronic compass) to provide a fusion input to keep yaw DC established. As is, this copter suffers from long term yaw drift.

We are making a C-130, not a F-16



- Remember this one? *(you should have learned it in a math class somewhere along the line)*
  - For small angles (α) the $\tan^{-1}$(opposite/adjacent) ≈ opposite/adjacent
  - Taking it one step further for small angles: adjacent ≈ hypotenuse = unity
  - So… for us pitch is simply proportional to AY and roll is proportional to –AX.
  - This is only true for small angles, but we are making a cargo plane, not fighter plane so angles will always be small.

Perform this math inside ***inertial_integrator.sv***

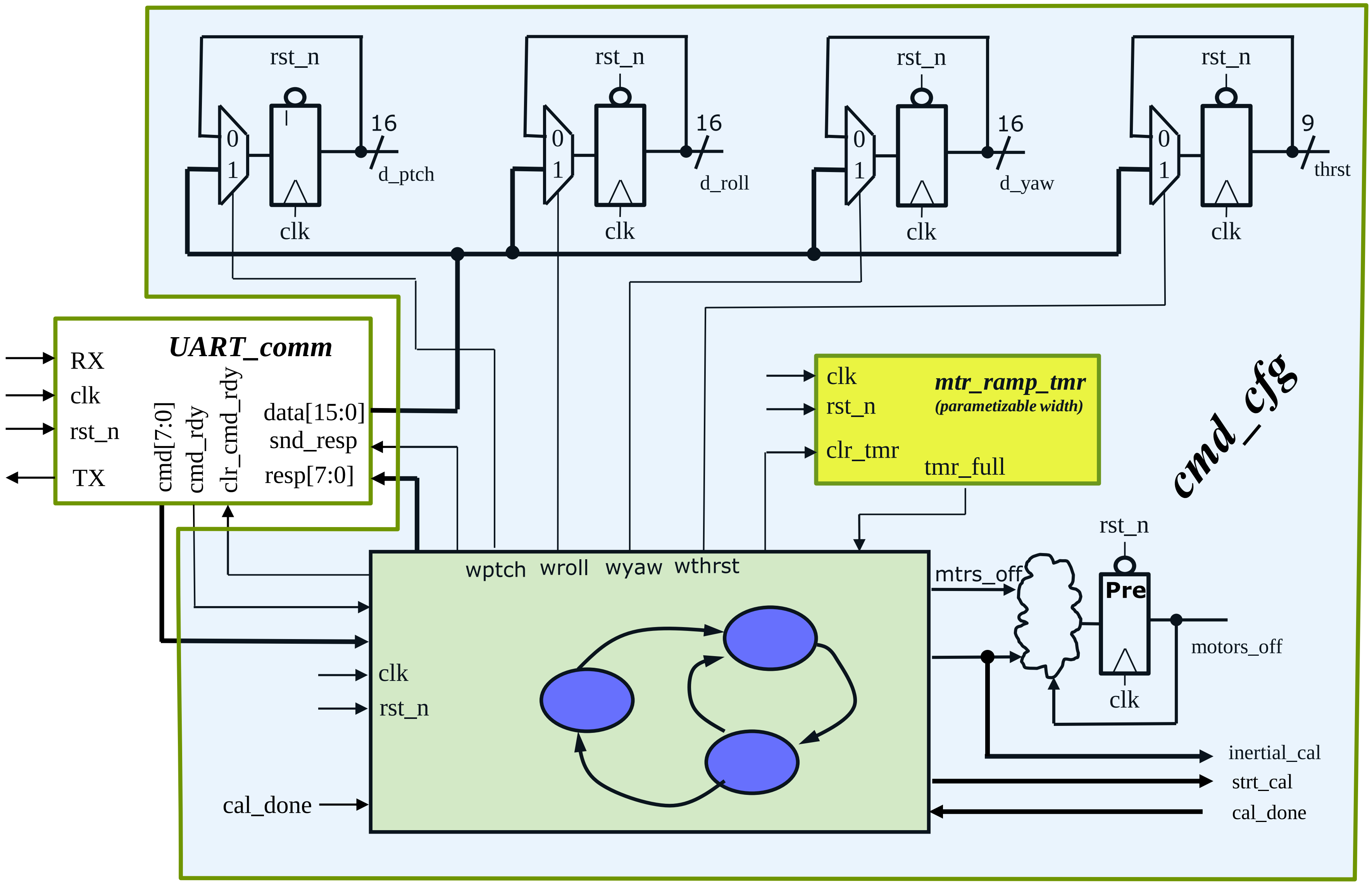
```
ptch_g_product = ay * $signed(327);                        // 327 is fudge factor
ptch_g = {{4{ptch_g_product[24]}},ptch_g_product[24:13]};  // pitch angle calculated from accel only

if (state==FLIGHT)          // if flying
  if (ptch_g>ptch)          // if pitch calculated from accel > pitch calculated from gyro
    fusion_ptch_offset = +4096;
  else
    fusion_ptch_offset = -4096;
else
  fusion_ptch_offset = 0;
```

fusion_ptch_offset is added into next integration of pitch rate accumulator. Of course similar is done for roll, except roll will use: roll_g_product = -ax * $signed(327). **NOTE:** the negative sign.

Fusion Calcs…litte more explicit

- While in FLIGHT or during CAL the pitch integrator and roll integrator are summing their respective rate signals. But we are also going to sum in this **fusion_XX_offset** term to "leak" the gyro angular measurement to agree with the accelerometer gyro measurement.
  - `ptch_int <= ptch_int + {{11{ptch_comp[15]}},ptch_comp} + fusion_ptch_offset;`
  - `roll_in <= roll_int + {{11{roll_comp[15]}},roll_comp} + fusion_roll_offset;`
- **fusion_XX_offset** will be zero during CAL.
- During FLGHT **fusion_ptch_offset** will be +2048 if **ptch_g > ptch** and will be **-**2048 if **ptch_g < ptch.** Similar for **fusion_roll_offset**

cmd_cfg Unit


rst_n
rst_n
rst_n
rst_n
0
1
16
d_ptch
clk
16
d_roll
clk
16
d_yaw
clk
9
thrst
clk
UART_comm
RX
clk
rst_n
TX
cmd[7:0]
cmd_rdy
clr_cmd_rdy
data[15:0]
snd_resp
resp[7:0]
clk
rst_n
clr_tmr
mtr_ramp_tmr
(parametizable width)
tmr_full
cmd_cfg
wptch
wroll
wyaw
wthrst
mtrs_off
rst_n
Pre
clk
motors_off
clk
rst_n
cal_done
inertial_cal
strt_cal
cal_done

cmd_cfg Unit

- The ***cmd_cfg*** unit interprets the 24-bit commands from UART_comm (sent via BLE from the remote control). The interface is given in the table below:

Signal:	**Dir:**	**Description:**
clk, rst_n	in	clock and reset
cmd_rdy	in	New command valid from ***UART_wrapper***
cmd[7:0]	in	Command opcode
data[15:0]	in	Data to accompany command
clr_cmd_rdy	out	Knocks down ***cmd_rdy*** after ***cmd_cfg*** has "digested" command
resp[7:0]	out	Response back to remote. Typically pos_ack (0xA5)
send_resp	out	Indicates ***UART_wrapper*** should send response
d_ptch, d_roll, d_yaw [15:0]	out	Desired pitch, roll, yaw as 16-bit signed numbers. ***cmd_cfg*** will have registers that store the desired pitch, roll, and yaw and output them to ***flght_cntrl*** unit.
thrst[8:0]	out	9-bit unsigned thrust level. Goes to ***flght_cntrl***.
Continued next slide		

cmd_cfg Unit (interface continued)

Signal:	Dir:	Description:
strt_cal	out	Indicates to ***inertial_integrator*** to start calibration procedure. Only 1 clock wide pulse at end of 1.34 sec motor spinup period
inertial_cal	out	To ***flght_cntrl*** units. Held high during duration of calibration (including motor spin up). Keeps motor speed at CAL_SPEED.
cal_done	in	From ***inertial_integrator***. Indicates calibration is complete.
motors_off	out	Goes to ***flght_cntrl***, shuts off motors. (user back off throttle and gestures down to shut off copter after landing, or more likely after crashing).

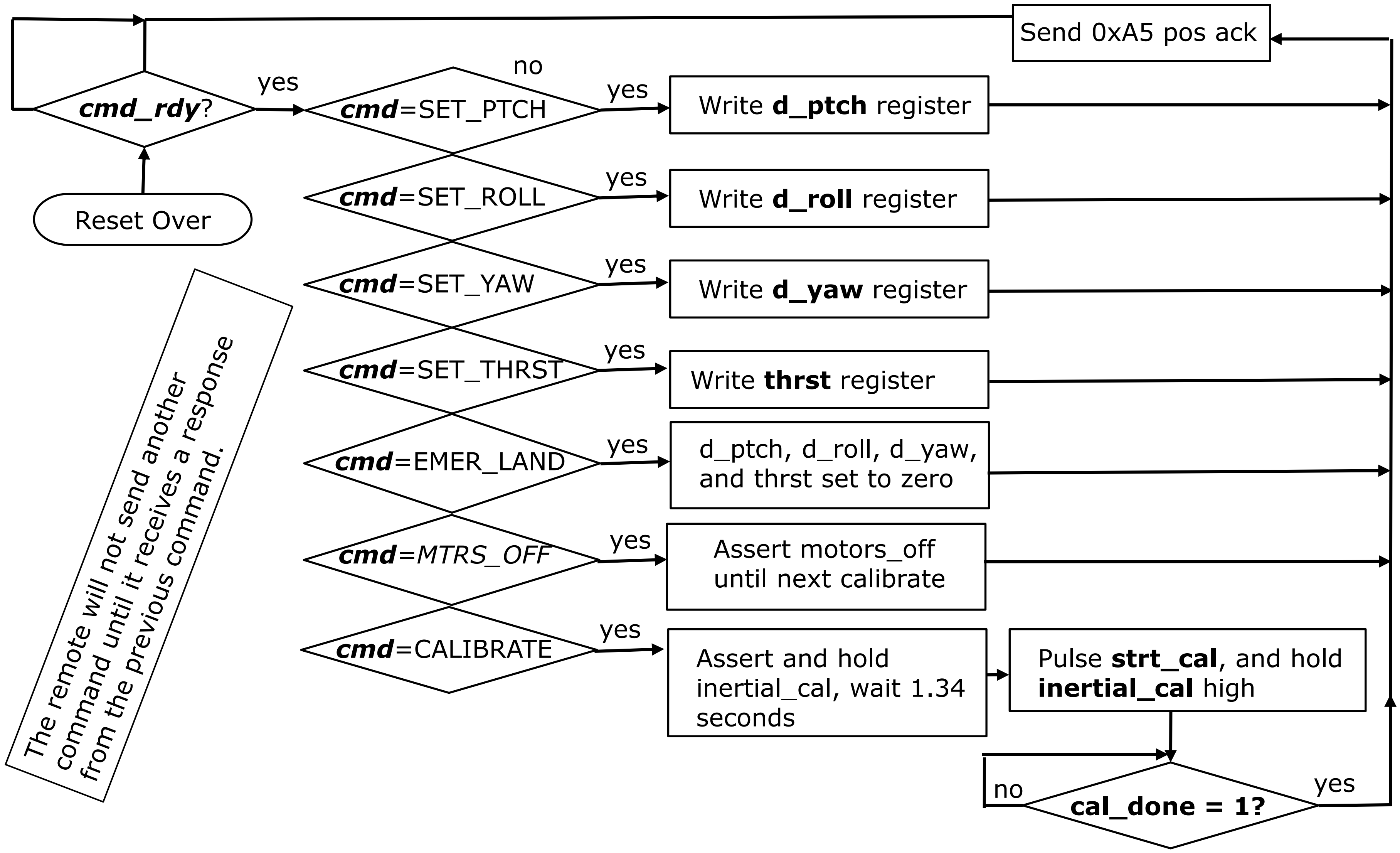
- In addition to **d_ptch**, **d_roll**, **d_yaw**, **thrst**, **motors_off**, other internal flops you are likely to need include:
  - 3-bits of statemachine flops
  - A (26-bit/9-bit) timer (bring motors up to speed for 1.34 sec prior to calibration start). This should employ a FAST_SIM parameter to expedite simulations. When FAST_SIM is 1 then it will be a 9-bit wide timer.

Command Set (commands received over UART interface)

Command Encoding: All commands are 24-bits. The upper byte [23:16] encodes the opcode.

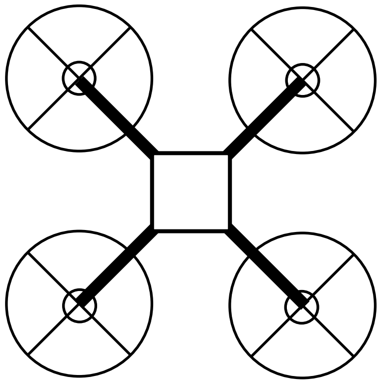
Bits[23:16]	Bits[15:0]	Description:
8'h02	16'hpppp	Set desired pitch. The lower 16-bits represent the desired pitch. It is a **signed** 16-bit number.
8'h03	16'hrrrr	Set desired roll. The lower 16-bits represent the desired roll. It is a **signed** 16-bit number.
8'h04	16'hyyyy	Set desired yaw. The lower 16-bits represent the desired yaw. It is a **signed** 16-bit number.
8'h05	16'h0ttt	Set the desired thrust. It is an **un**signed 9-bit number.
8'h06	16'hxxxx	Calibrate quadcopter. Motors are turned on so they run at minimum cal speed. After 1.34 seconds with motors on the gyro calibration starts (discovery of offset for gyro readings)
8'h07	16'h0000	Emergency land. Pitch, roll and yaw set to zero (level) thrust set to zero (minimum run speed). Copter will land as soon as possible.
8'h08	16'hxxxx	Motors off (assert motors_off signal to flight_cntr) and keep it asserted until we see a calibrate copter (0x06) command.

cmd_cfg Functionality (command processing)

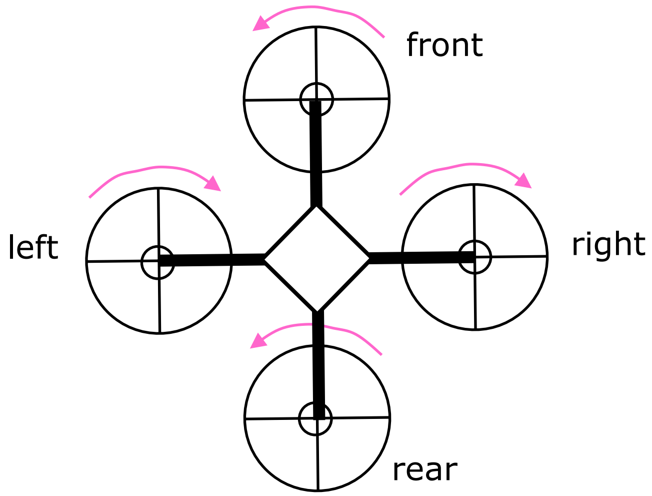


Send 0xA5 pos ack
cmd_rdy?
yes
no
Reset Over
cmd=SET_PTCH
yes
Write d_ptch register
cmd=SET_ROLL
yes
Write d_roll register
cmd=SET_YAW
yes
Write d_yaw register
cmd=SET_THRST
yes
Write thrst register
cmd=EMER_LAND
yes
d_ptch, d_roll, d_yaw, and thrst set to zero
cmd=MTRS_OFF
yes
Assert motors_off until next calibrate
cmd=CALIBRATE
yes
Assert and hold inertial_cal, wait 1.34 seconds
Pulse strt_cal, and hold inertial_cal high
no
cal_done = 1?
yes
The remote will not send another command until it receives a response from the previous command.

Flight Control


"x" formation
not us

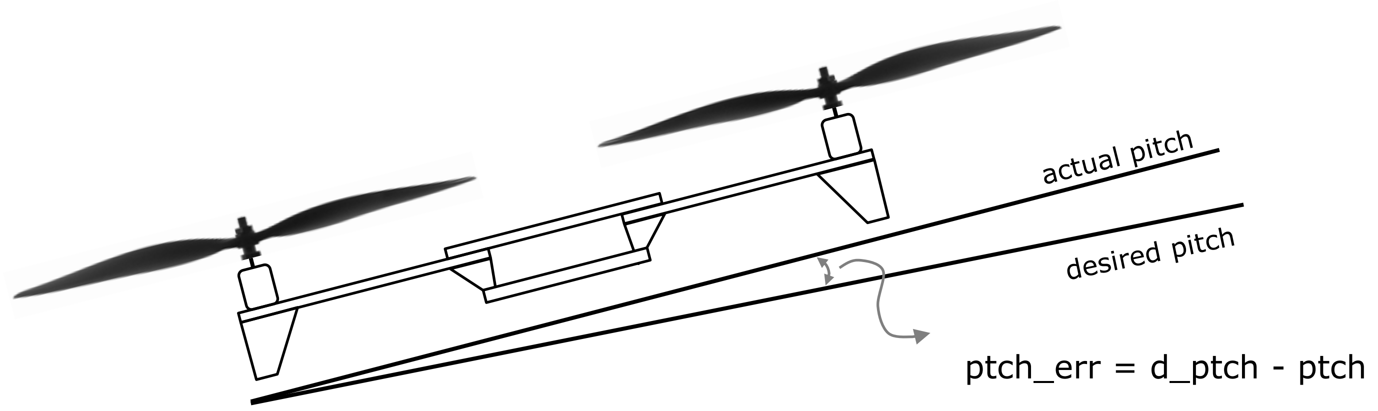
Quadcopters can fly in an "x" formation or a "+" formation. Choice is somewhat arbitrary. Due to orientation of inertial sensor mount ours will fly as a "+"



"+" formation is what we fly

- The left and right props spin clockwise (thus create a counter clockwise counter torque)
- The front and back props spin counter clockwise (thus create a clockwise counter torque)
- To pitch + (up) we would drive the front motor harder than the rear motor
- To roll + (to the right) we would drive the left motor harder than the right motor
- To yaw + (counter clockwise) we would drive the left & right motors harder than the front & back motors

Flight Control PID



- PID is a very common control scheme. It is based on the idea that your control input is based on an error term, where the error is the desired set point minus the actual measurement.
- Then the control input (front vs rear motor speed in this example) is based on a combination of terms. One term **P**roportional to the error. One term that is the **I**ntegral of the error over time, and one term that is proportional to the **D**erivative of the error term.

$$ptch\_control = P_k * ptch\_err + I_k * \int ptch\_err + D_k * \frac{dptch\_err}{dt}$$

Flight Control PID (continued)

- Do you think if we gave the exact same speed command to all 4 motors the quadcopter would fly level?
  - Do you think the cheap motors are all that well matched?
  - Do you think the ESC’s for the motors are all calibrated?
  - Do you think I built the chassis with perfect balance?

- So even at level flight (ptch = 0, roll = 0) there are some motors that are being driven harder than others.
- If there was just a **P** term (no **I** term or **D** term) then for one motor to be driven harder than another there would have to be a non-zero error term…right? (think about it)
  - This means to get the copter to fly level the pilot would have to use “stick” input that was not neutral.
  - This is why the **I** term exists. To drive the error to zero (level control input results in level flight) the error is integrated over time to form a portion of the control input.

- OK…so why do we need the **D** term.
  - Many control systems do not need a **D** term. If the thing you are trying to control has very little lag (responds quickly to the control input) or has very little inertia then the **D** input is not needed. Most power electronics are controlled with just **PI** control.
  - However, we have definite inertia in our system. Once you start pitching or rolling the quadcopter it has angular momentum. As you are approaching your desired angle you need to back off (or even reverse) your control input so the angular momentum does not shoot you way past the desired point.

Flight Control PID (continued)

- Our control input comes from a glove with an inertial sensor on it. However you hold your hand the quadcopter mimics it. Without an "I" term you have some small residual error. Meaning you have to hold your hand slightly non level to achieve perfectly level flight. Trust me, you don't even notice this. So to simplify things we are removing the I term. We are doing **PD** control.

- The derivative will simply be done by keeping track of the previous error terms in a queue and creating an error proportional to ***err – prev_err*** where ***prev_err*** is the oldest value in our queue**.** It is a bit cheesy but works fine.

- The amount of **P** vs **D** to mix together, and the depth of the prev_err queue used for **D** term were found through much trial and error. A large part of my engineering career has been built on guess and check. The math will be specified in excruciating detail as part of HW2 & HW4. Pay very close attention to it and implement it exactly as specified.

The speed of the motors are given below in general terms. If you think about it, it should make sense to you.

Front_speed = thrst + MIN_RUN_SPEED – ptch_Pterm – ptch_Dterm – yaw_Pterm – yaw_Dterm

Back_speed = thrst + MIN_RUN_SPEED + ptch_Pterm + ptch_Dterm – yaw_Pterm – yaw_Dterm

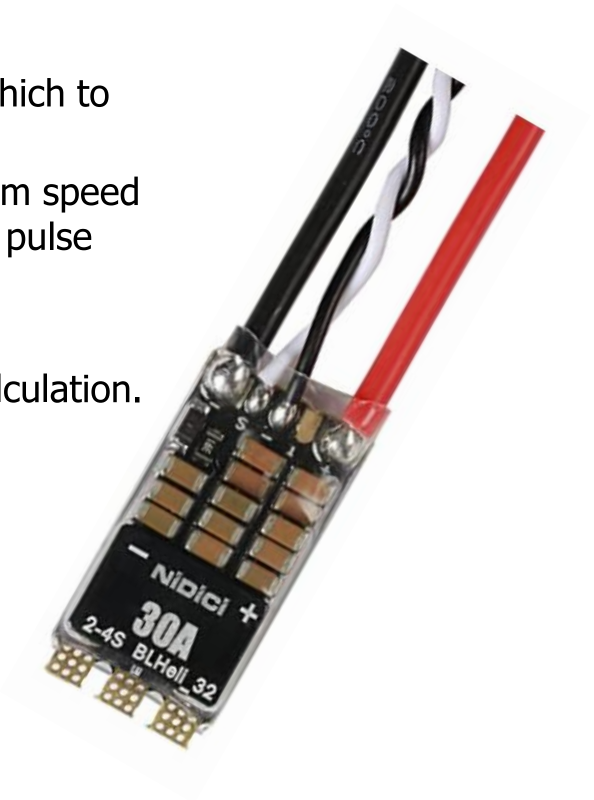
Left_speed = thrst + MIN_RUN_SPEED - roll_Pterm - roll_Dterm + yaw_Pterm + yaw_Dterm

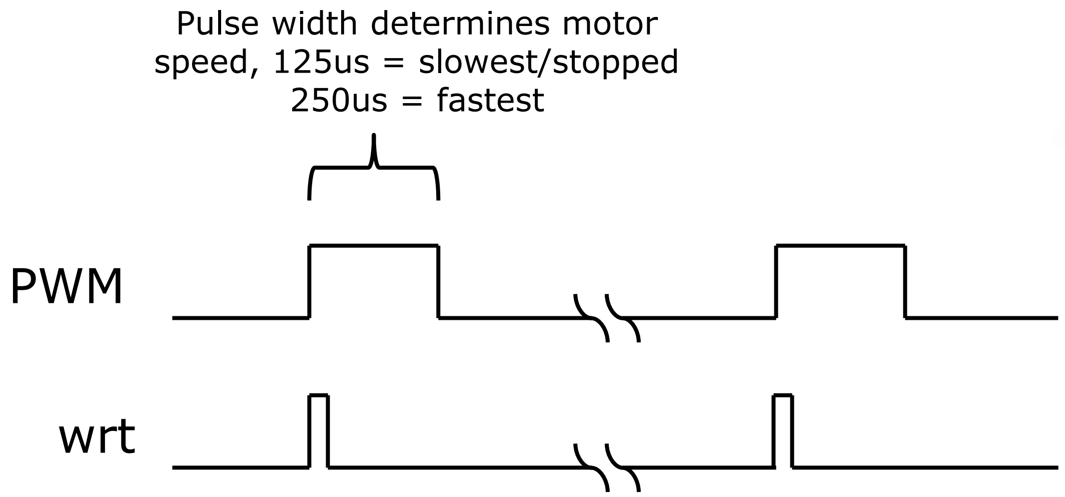
Right_speed = thrst + MIN_RUN_SPEED + roll_Pterm + roll_Dterm + yaw_Pterm + yaw_Dterm

(**Note:** thrst is the overall thrust command that comes from a slide potentiometer held in the pilots other hand)

ESC = Electronic Speed Control

- ESC's take in a single pulse signal to determine the speed at which to run the motor. The width of the pulse determines the speed.
- A well calibrated ESC would have the motor running at minimum speed with a 125us pulse, and at maximum speed with a 250us wide pulse
- We will do one shot pulses where the leading edge of a pulse corresponds to a write (**wrt**) signal.
- The **wrt** pulses will be provided once every PD control loop calculation.





Pulse width determines motor speed, 125us = slowest/stopped 250us = fastest
PWM
wrt

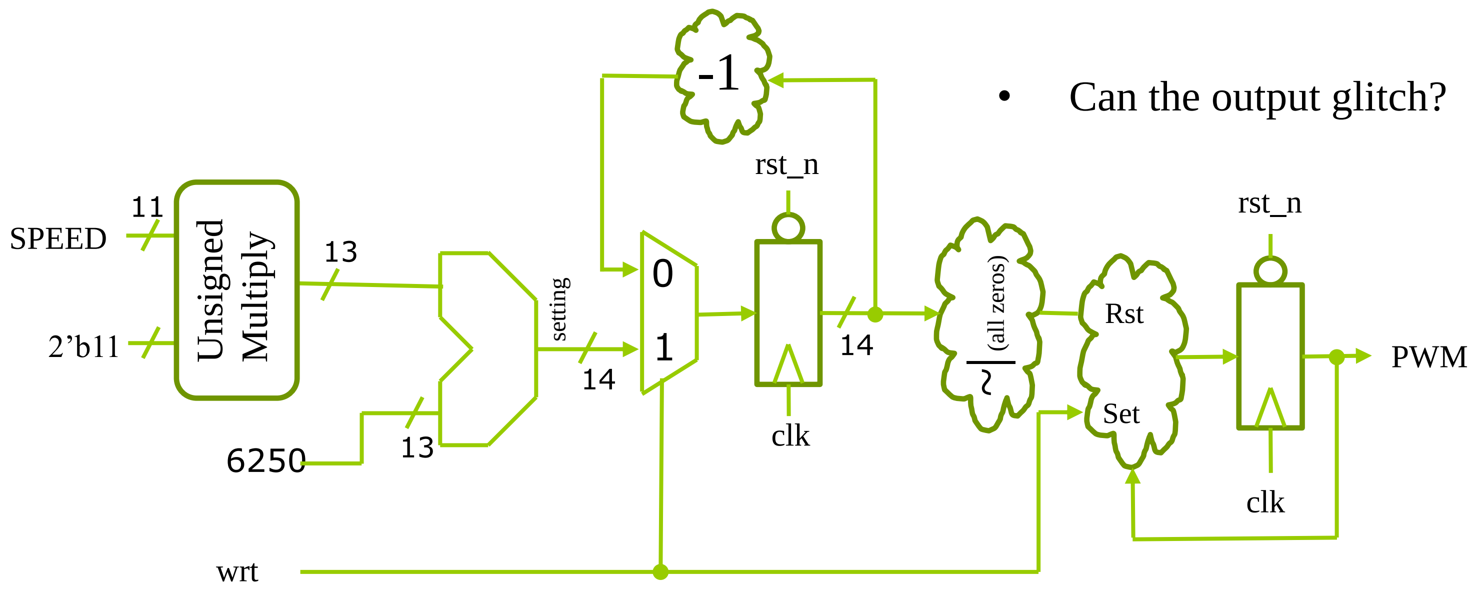
ESC_Interface

Signal:	Dir:	Description:
clk	in	50MHz clock
rst_n	in	Active low asynch reset
wrt	in	Initiates new pulse. Synched with PD control loop
SPEED[10:0]	in	Result from flight controller telling how fast to run each motor.
PWM	Out	Output to the ESC to control motor speed. It is effectively a PWM signal.

- If SPEED is zero the pulse should be 125us wide (6250 system clocks)
- If SPEED is 2047 the pulse should be an additional 125us wide. Meaning an additional 6250 system clocks wide (total width of 12500 system clocks)
- 2047*3 = 6141 clocks. Close enough to 6250 for me. That means we can get to 99.1% full throttle with a SPEED value of 2047 (full value).
- So for each additional count of SPEED we add 3 system clocks.

ESC_Interface Implementation

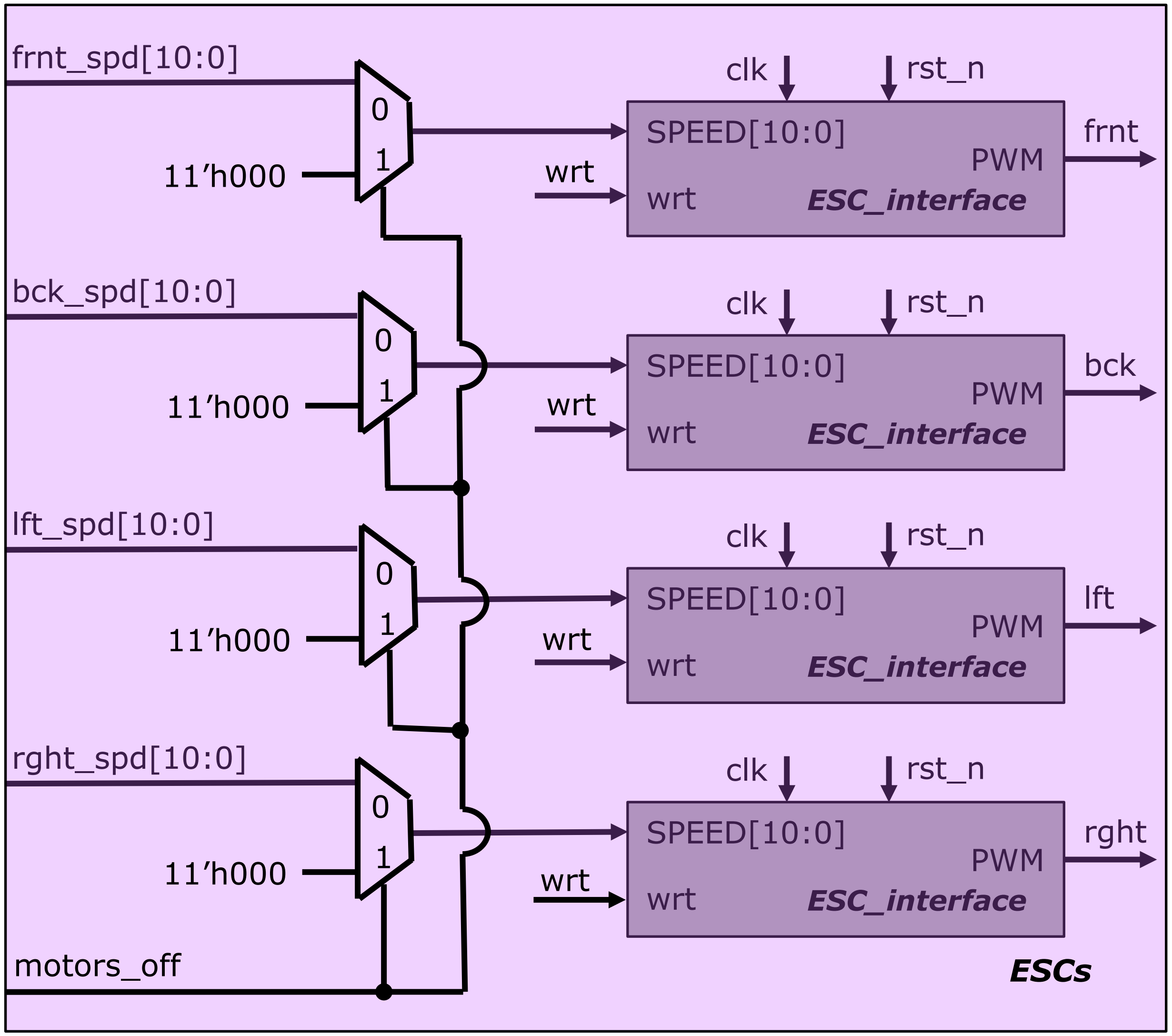
- Study this implementation…how does it work?
- What initiates a rising edge on the PWM output signal?
- Why addition of SPEED*3 with 6250?
- Why count down?



- Can the output glitch?

- This Set/Reset/Hold flop output is a very handy construct, and we will use this output style several places in future units… remember it.

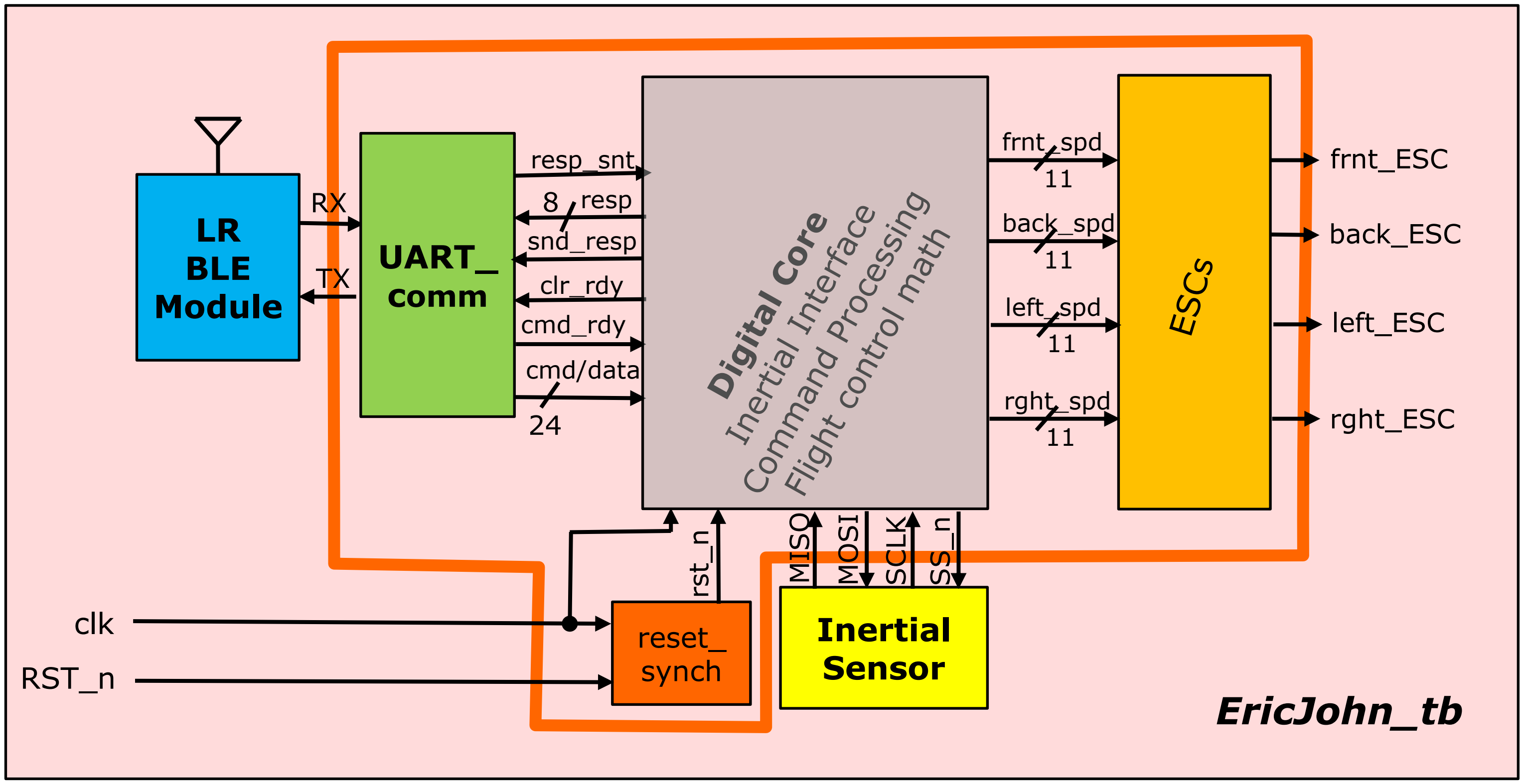
ESCs Block



The ESCs block is a simple block that instantiates 4 copies of your ESC_intertface (done in Exercise11)

It also implements the motors_off feature by forcing the speed to be zero if **motors_off** is asserted.

Required Hierarchy & Interface



Your design will be placed in an "EricJohn" testbench to validate its functionality. It must have a block called **QuadCopter.sv** which is top level of what will be the synthesized DUT.
The interface of **QuadCopter.sv must match exactly** to our specified **QuadCopter.sv** interface
Please download QuadCopter.sv (interface skeleton) from the class webpage.

The hierarchy/partitioning of your design below QuadCopter is up to your team.
The hierarchy of your testbench above QuadCopter is up to your team.

QuadCopter Interface

Signal Name:	Dir:	Description:
clk	in	50MHz clock
RST_n	in	Unsynchronized reset (from push button) goes through ***clk_rst_smpl*** block to get synchronized and form **rst_n** (global reset to all other blocks).
SS_n	out	Active low slave select to inertial sensor
SCLK	out	SCLK of SPI interface to inertial sensor
MOSI	out	Master Out Slave In to inertial sensor
MISO	in	Master In Slave Out from intertial sensor
INT	in	INT signal from intertial sensor (rising edge indicates new data ready)
RX	in	UART input from Bluetooth link
TX	out	UART output to Bluetooth link
FRNT,BCK,LFT, RGHT	out	ESC signals to motors

Provided Modules & Files: (available on website under: Project as Collateral.zip)

File Name:	**Description:**
Quadcopter_tb.sv	**Optional** testbench template file.
QuadCopter.sv	**Requried** interface skeleton verilog file. **Copy this** you can change internals if you wish, but interface signals must remain as is.
CycloneIV.sv	Model of Inertial sensor and overall physics of the quadcopter. You need this to make a "complete" testbench.
inertial_integrator .sv	Does the integration and fusion. Not provided because it was too hard for you to implement, but because it was too hard for me to specify well.
SPI_iNEMO3	Model of the inertial sensor (child of CycloneIV model of copter)

Synthesis:

- You have to be able to synthesize your design at the QuadCopter level of hierarchy.
- You should have a synthesis script. It will be reviewed.
- Your synthesis script should write out a gate level netlist of QuadCopter (**QuadCopter.vg**).
- You should be able to demonstrate at least one of your tests running on this post synthesis netlist successfully.
- Timing (400MHz for clk) is pretty easy to make. Your main objective is to minimize area.

Use of **compile_ultra** is not permitted
(has been seen to cause random functional bugs in the past)

Synthesis Constraints:

Contraint:	Value:
Clock frequency	266.6MHz for clk (3.75ns period)
Input delay	0.25ns after clk for all inputs
Output delay	0.5ns prior to next clk rise for all outputs
Drive strength of inputs	Equivalent to a NAND2X2_RVT gate from our library
Output load	0.1pF on all outputs
Wireload model	16000 area model
Max transition time	0.15ns
Clock uncertainty	0.20ns

NOTE: Area should be taken after all hierarchy in the design has been smashed.

Use of **compile_ultra** is not permitted
(has been seen to cause random functional bugs in the past)