ECE 551Project Spec

Spring ’22 **eBike Controller**

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

- Project Grading Criteria:
  - Quantitative Synthesis Element 15%

  (yes this could result in extra credit)

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

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

Note: Minimum synthesized frequency to the Synopsys 32nm library is 400MHz. No benefit to faster than 400MHz.

- Project Review (85%)
  - ✓ Code Review (12.5%)
  - ✓ Testbench Method/Completeness (15%)
  - ✓ Synthesis Script review (7.5%)
  - ✓ Post-synthesis Test run results (10%)
  - ✓ Results when placed in ourTestbench (22.5%)
  - ✓ Test of code on eBike platform & eBike (10%)
  - ✓ Teammates judgement of your contribution (7.5%)

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 coverage number and get another 1% 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 another 1% extra credit.

- Also some extra credit for submitting early

Project Due Date

- Project Dropbox Due Times:
  - Weds (5/4/22) submit project for 1.5% extra credit
  - Thurs (5/5/22) submit project for 0.75% extra credit
  - Fri (5/6/22) on-time submission of project
  - Sat (5/7/22) submit project late for 0.75% penalty

- Project Review Involves:
  - ✓Code Review
  - ✓ Testbench Method/Completeness
  - ✓Synthesis script review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in our testbench
  - ✓ Run of your code on eBike platform

Submitting the Project (what we need from you)

- Project Grading Criteria:
  - Quantitative Synthesis Element 15%

  (yes this could result in extra credit)

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

We need your **area_report.txt** from synthesis *(we will double check this by running synth so don't fake it)*

  - Project Review (85%)
    - ✓ Code Review (12.5%)
    - ✓ Testbench Method/Completeness (15%)

(you need to provide a 3-4 min video giving me a tour of your test-bench. What it covers and methodology)

    - ✓ Synthesis Script review (7.5%)
    - ✓ Post-synthesis Test run results (10%)

(you need to provide a 2min showing you running a test post synthesis)

    - ✓ Results when placed in our testbench (22.5%)
    - ✓ Test of code on eBike platform & eBike (10%)
    - ✓ Teammates judgement of your contribution (7.5%)

Just zip up all your code, all your **.sv** and **.v**. No harm in giving us more **source code** than needed. But please avoid including binary ***work*** directories. We don't need bloat eating up Canvas dropbox disk space.

Need a **.mp4** file. Keep it short and try to keep resolution low so file size is reasonable. If you exceed 4 min there will be a penalty.

We need your **.dc** file (top level synthesis script)

Need a **.mp4** file. Keep it short and try to keep resolution low so file size is reasonable. If you exceed 2 min there will be a penalty.

You need to complete the Google Form survey sent

If you did the extra credit (code coverage) you need to provide a short video showing the results and explaining how you obtained them. Keep it short!

Submit to Project Dropbox by Friday May 7th 11:59PM

- Create a new directory (folder)
- Copy all your **.sv** and **.v** files to it.
- Copy your **area_report.txt** from synthesis
- Copy your **.dc** script from synthesis
- Put your **.mp4** video tour video on there.
- Put your **.mp4** video of post synth results there.
- If you did code coverage put additional **.mp4** in there.
- **.zip** it. **.rar** will receive no credit
- Submit it to the Project Dropbox

ONLY one person per project team submit!

If you use your phone to make the video then lower your cameras resolution. There is a CameraMX app on android that takes low res video.

Complete the Google form rating your teammates.

Test Platform

Whole controller board mounted on a pivot to mimic going up/down hill

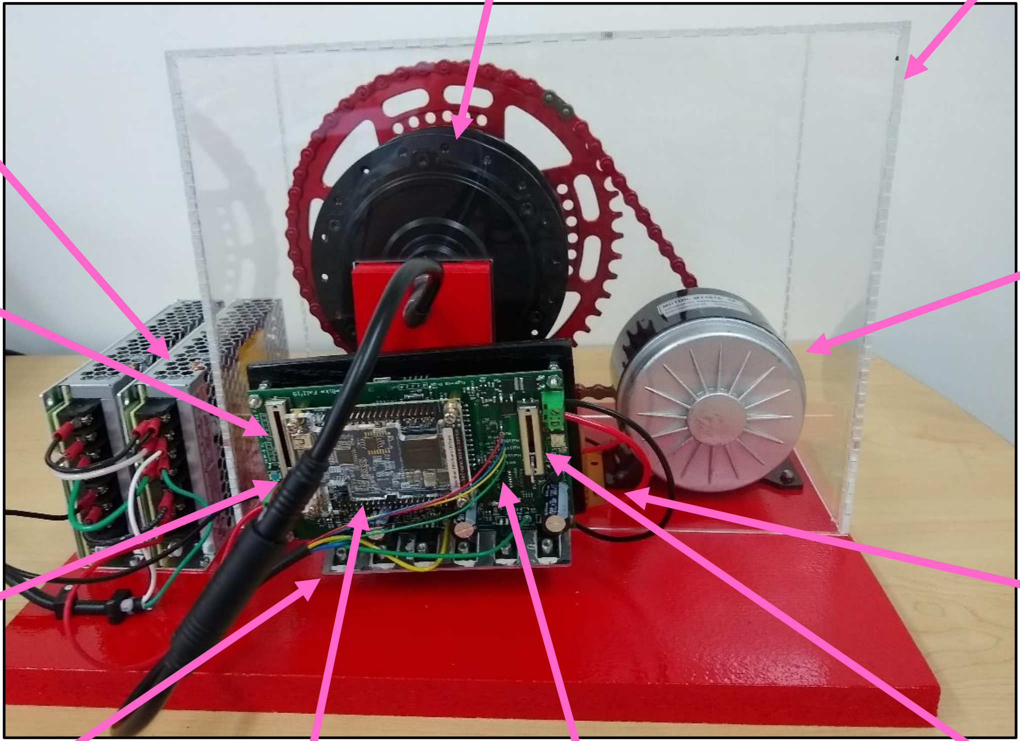
E-Bike hub motor (250W brushless DC)

Wouldn't want you getting your hair caught in the chain.

2 series 18V supplies to provide 36V DC

This slide potentiometer on the board mimics the pedaling torque sensor

Push button that mimics rider squeezing the brake handle



E-Bike motor is coupled (via chain) to generator. The generator serves as a mechanical load on the motor.

Load resistor to dissipate the power the generator generates.

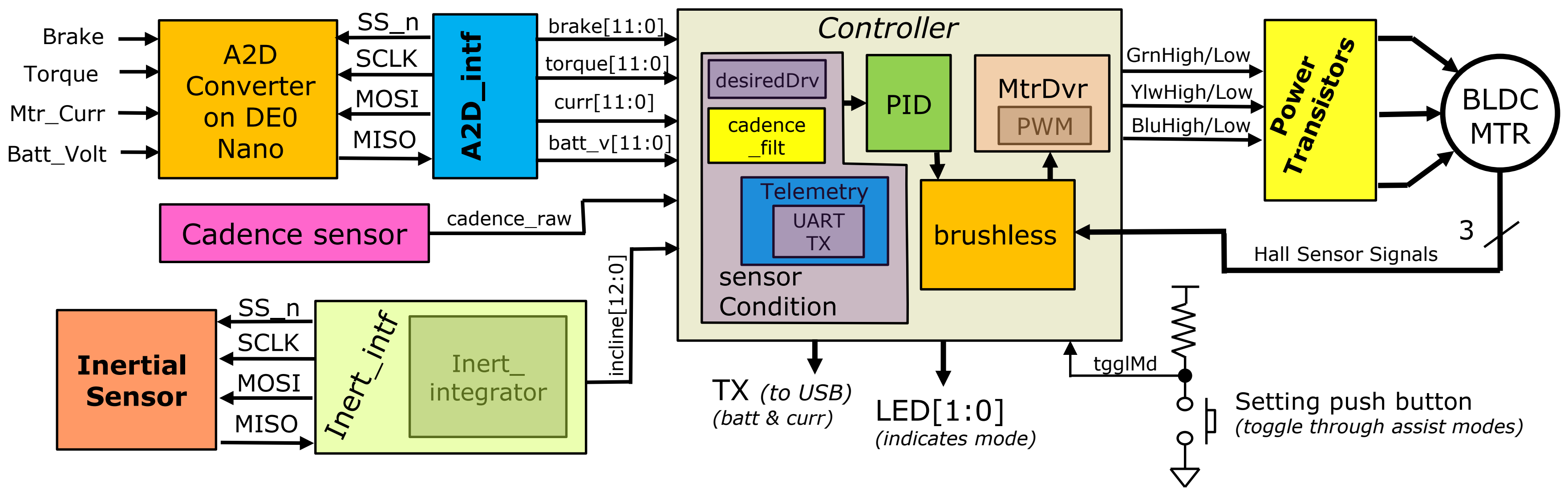
You can see the 6 Power FETs that drive the motor coils are mounted to an aluminum heat sink

The DE-0 Nano board contains the FPGA that is the "brains" of the operation.

Here you can see the Hall effect sensor wires coming in from the motor.

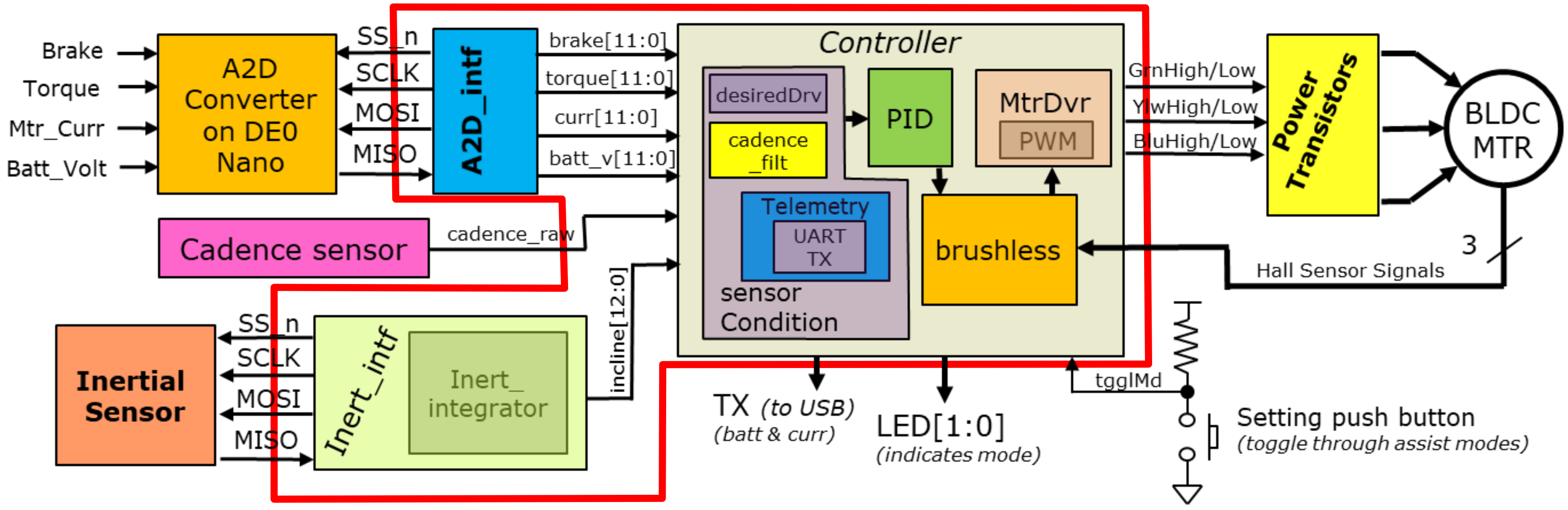
This slide potentiometer mimics the cadence sensor

Block Diagram of Digital Portion



The eBike rider is assisted by a brushless DC motor (BLDC Mtr). The digital portion of the design will include a controller that directly drives the gates of the power transistors that drive the motor coils. The desired level of assist is computed as a function of the rider's effort (torque & cadence) plus an extra assist for when they are going up hill. An inertial sensor (accelerometer/gyro combo) is used to determine the incline the rider is ascending. The desired assist is converted to actual motor controls via a PID loop and a 6-state brushless DC motor algorithm. There is a push button on the DE0-Nano that can be used to toggle the level of assist (off,hard,medium,easy) that level is displayed on LED[1:0]. The battery voltage is monitored and motor drive is shut down when the batteries are discharged too far. The brake lever is also monitored via the A2D converter. Regenerative braking is invoked when the Brake signal falls below a threshold. The battery, motor current, and torque are transmitted via a UART for an optional display.

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 via Synopsys to our standard cell library For practical purposes we will also map that logic to FPGA so we can run the demos.

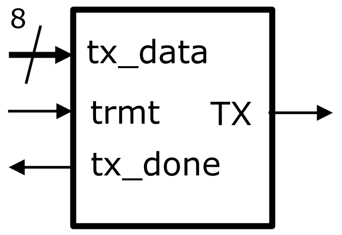
You Must have a block called **eBike.sv** which is top level of what will be the synthesized DUT.

Telemetry Module

The eBike controller will be acquiring battery voltage, motor current, and rider's input torque via an A2D converter. These 12-bit values will be periodically transmitted (via a UART) to an optional handlebar mounted display (to a USB port on our test station). The UART transmitter (**UART_tx.sv**) is provided and can be downloaded from the Canvas page.

A UART transmitter sends a byte at a time serially over the **TX** line. Its operation is quite simple. You present a byte you wish it to transmit on ***tx_data[7:0]*** and then hold ***trmt*** high for one clock cycle. When it has completed transmitting that byte it will indicate it by raising ***tx_done***.

UART Transmitter

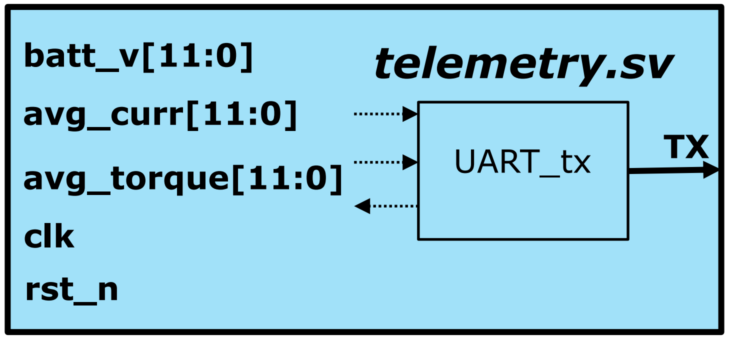


A wrapper around **UART_tx.sv** is needed that will periodically send out ***batt_v[11:0], avg_curr[11:0]*** and ***avg_torque[11:0]***. The packet (sent roughly 48 times a second) will be 8 bytes in length. The packet consists of a 2-byte delimiter of 0xAA 0x55 followed by 6-bytes of payload.

delim1	delim2	payload1	payload2	payload3	payload4	payload5	payload6
0xAA	0x55	high byte ***batt_v***	low byte ***batt_v***	high byte ***curr***	low byte ***curr***	high byte ***torque***	low byte ***torque***

The receiving unit (intended to be a display on handle bars) will know to look for the delimiter and will know the order of the subsequent bytes, hence can decode and display the data. A UART receiver (**UART_rcv.sv**) is provided to aid in testing.

Telemetry Module

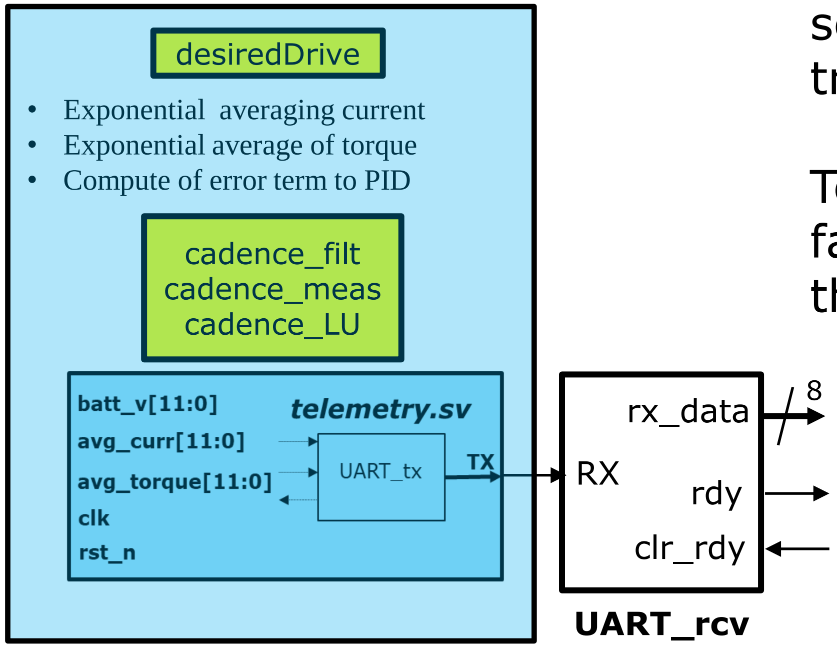


telemetry.sv should have the interface shown.

telemetry.sv will itself be a child of a larger unit called **sensorCondition.sv**

sensorCondition.sv has all conditioned sensor readings readily available for transmission by telemetry.

Testing of **telemetry.sv** can be facilitated by downloading **UART_rcv.sv** the companion to **UART_tx.sv**



NOTE: The current transmitted via telemetry should be the calculated average current. The torque transmitted should also be the calculated average torque. So…in short, transmit the average values not the raw for current & torque.

Desired Drive (establish target current)

Signal:	Dir:	Description:
avg_torque[11:0]	in	Unsigned number representing the toque the rider is putting on the cranks (force of their pedaling)
cadence[4:0]	in	Unsigned number representing the speed of the rider's pedaling.
not_pedaling	in	Asserts if cadence is so slow it has been determined the rider is not pedaling
incline[12:0]	in	Signed number (you just delt with this in Part I)
scale[2:0]	in	unsigned. Represents level of assist motor provides 111 => a lot of assist, 101 => medium, 011 => little, 000 => no assist
target_curr[11:0]	out	Unsigned output setting the target current the motor should be running at. This will go to the PID controller to eventually form the duty cycle the motor driver is run at.

The interface of **desiredDrive** is shown above. The following slide gives a verbal description of the functionality of this purely combinational block.

Desired Drive (description of calculations)

- First step is what you already did. The 13-bit signed **incline** signal should be saturated to a 10-bit signed value (**incline_sat**)

- Next create a signal called **incline_factor** that is a sign extended (to 11-bits) **incline_sat** + 256. Is it capable of being negative?

- Next you will limit **incline_factor** and create a new 9-bit saturated signal that is clipped with respect to negative values. Meaning if **incline_factor** was negative it will be clipped to zero. If **incline_factor** was > 511 it will be saturated to 511. This new signal should be called **incline_lim** (limited).

- A 5-bit input called **cadence** provides a signal proportional to the rider's cadence *(rate of pedaling)*. It is also accompanied by a signal called **not_pedaling** that is asserted if the **cadence** is too slow.

Desired Drive (description of calculations)

- **avg_torque[11:0]** is an unsigned number representing the force of the pedaling. The toque sensor, however, has a significant offset that needs to be subtracted out. Create a signal (called **torque_off)** that is **avg_torque** minus TORQUE_MIN. TORQUE_MIN should be a **localparam** and set to 12'h380. How wide does **torque_off** need to be?

- Can the above **torque_off** reading be negative? There will be variation in offsets between different torque sensors. So perhaps a sensor could have a lower offset then TORQUE_MIN, in which case the above subtraction of TORQUE_MIN could produce a slightly negative value. **torque_off** should be 13-bits wide and formed from the subtraction of two zero extended 12-bit quantities (**avg_torque** & TORQUE_MIN).

- We treat torque as an unsigned number *(you only pedal forward)* , however, a slightly negative value *(resulting from an overflow of avg_torque – TORQUE_MIN)* would look like a huge positive torque. We cannot have this.

Desired Drive (description of calculations)

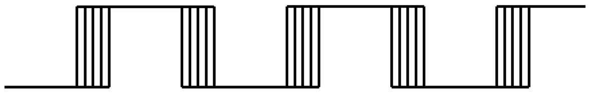
- Create a new signal (called **torque_pos**) that is just a zero clipped version of **torque_off**. Meaning, if **torque_off** is negative it assigns it to zero, otherwise it is simply a copy of the lower N-1 bits of **torque_off.**

- **scale** is a 3-bit input that can be thought of as an (off, low, medium, high) setting for the controller. 000=>off (no assist), 011=>low (little assist), 101=>medium, 111=>high (most assist to rider from motor).

- Create a new signal (called **assist_prod**) which will represent how much we wish the motor to assist the rider. When pedaling this will simply be a product of: **torque_pos**, **incline_lim**, **cadence**, and **scale**. When **not_pedaling** it is zero. How wide does this signal need to be? (OK…I'll tell you…30-bits, but make sure that jives with your thinking)

- Finally implement **target_curr** (the output that will eventually go to the PID controller). If any of bits [29:27] of **assist_prod** are set then we set this signal to 12'hFFF otherwise we set it to **assist_prod[26:15]**.

cadence_filt

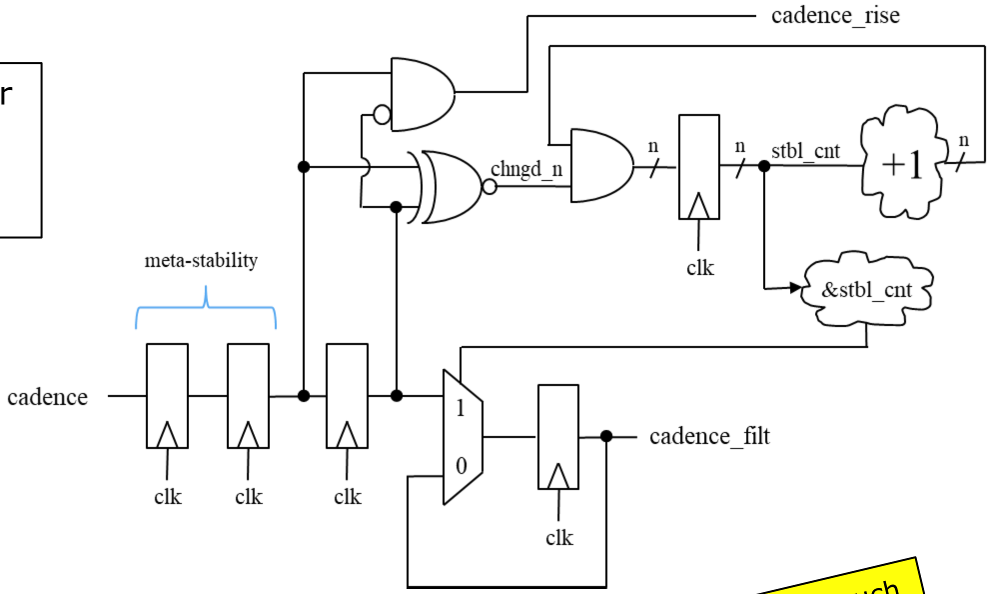


cadence signal

The cadence signal gives 32 pulses per rotation of the cranks. The signal, however, is very noisy and has many false edges at the transition points *(similar to bounce in a push button switch).* This needs to be filtered.

The "debounce" filter should look for greater than 1ms of stability in the signal before allowing the filtered version to be updated with the current signal. See the diagram

Whenever a change in the incoming ***cadence*** signal is detected a stability counter is knocked down. If the signal has been stable for longer than 1ms then the outgoing ***cadence_filt*** signal is updated with the incoming version. The resulting latency is not an issue in this application.



You did this block as part of HW2. However, you are much smarter now to take a few minutes to fix up that code.

sensorCondition (high level overview)

The **sensorCondition** performs the following functions:

1) Instantiates
   i. ***cadence_filt*** (see Ex08)
   ii. ***cacdence_meas***
   iii. ***cadence_LU*** → outputs 5-bit vector (**cadence**[4:0]) representing pedaling speed

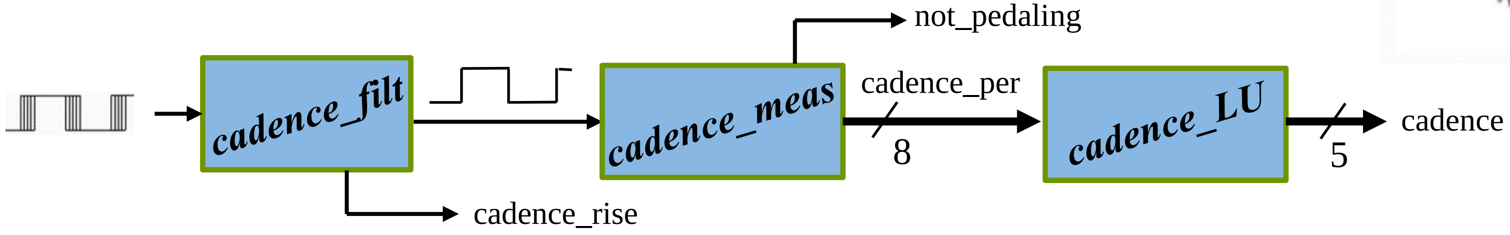
2) Perform an exponential average (of weight 4) on the raw 12-bit current reading (***curr***) to produce ***avg_curr***;

3) Perform an exponential average (of weight 32) on the raw 12-bit ***torque*** reading to produce ***avg_torque***. The accumulator of this exponential average should be seeded with 16X ***torque*** when pedaling resumes (***not_pedaling*** falls). This exponential average needs to be synchronized with the ***cadence_rise*** signal. Meaning the accumulator would update only on rising edges of ***cadence_filt***.

4) Since the **sensorCondition** block has many of the inputs to **desiredDrv** handy. It will instantiate **desiredDrv** to acquire ***target_curr***. Then it will form the ***error*** term (**target_curr – avg_curr**) that goes to the PID block. If the battery level (***batt***) falls below LOW_BATT_THRES (a localparam set to 12'hA98) ***error*** will be set to zero as a way of inhibiting drive when the battery level is too low. **not_pedaling** should also force **error** to zero.

5) Since most of the signals we transmit via the **telemetry** module are available in **sensorCondition** this is a convenient place to instantiate **telemetry**.

Cadence Blocks

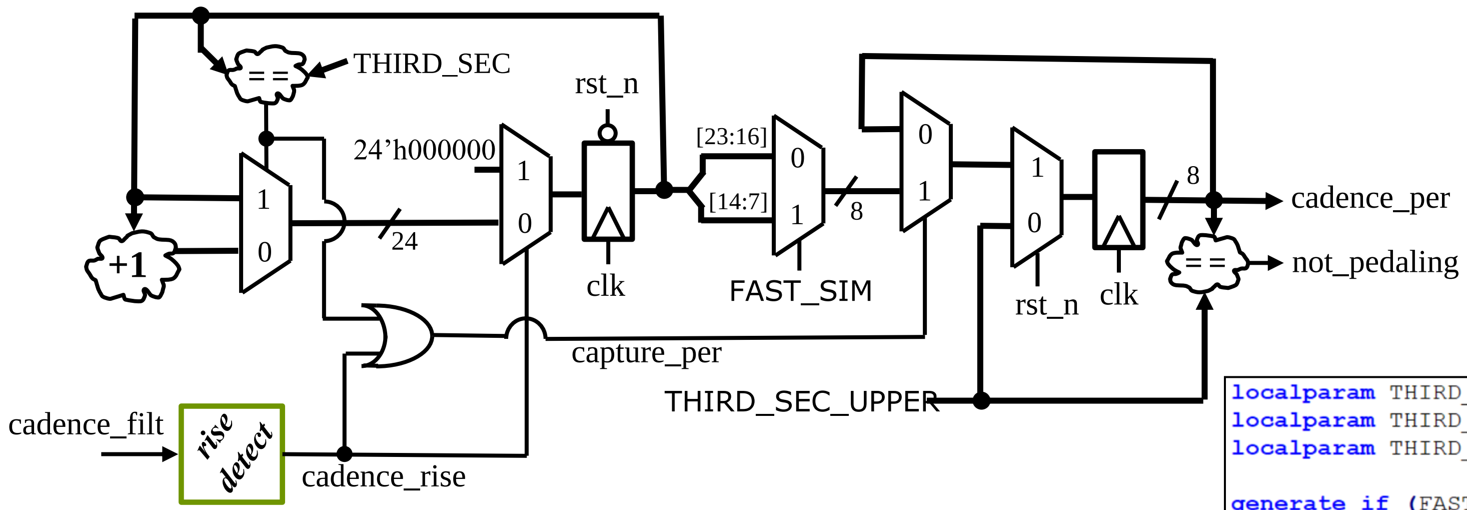


- ***cadence_filt*** takes the bouncy noisy cadence signal and filters it. You finished this in Ex08 but need to modify/update it a bit.

- ***cadence_meas*** is a new block you will create that will measure the period (number of clock cyles between cadence pulses) of the cadence signal. Only the upper 8-bits of a 24-bit period counter will be used. It also outputs a signal to inform if the rider is not pedaling.

- The cadence signal we really want is proportional to the square root of the inverse of the cadence period. We could make a block that both divides and calculates a square root. However, sometimes it is easier *(and lower area)* to simply employ a **L**ook **U**p table. ***cadence_LU*** is a provided block and was partially generated using Excel.

cadence_meas (make this)



```
localparam THIRD_SEC_REAL = 24'hE4E1C0;
localparam THIRD_SEC_FAST = 24'h007271;
localparam THIRD_SEC_UPPER = 8'hE4;

generate if (FAST_SIM)
  assign THIRD_SEC = THIRD_SEC_FAST;
else
  assign THIRD_SEC = THIRD_SEC_REAL;
endgenerate
```

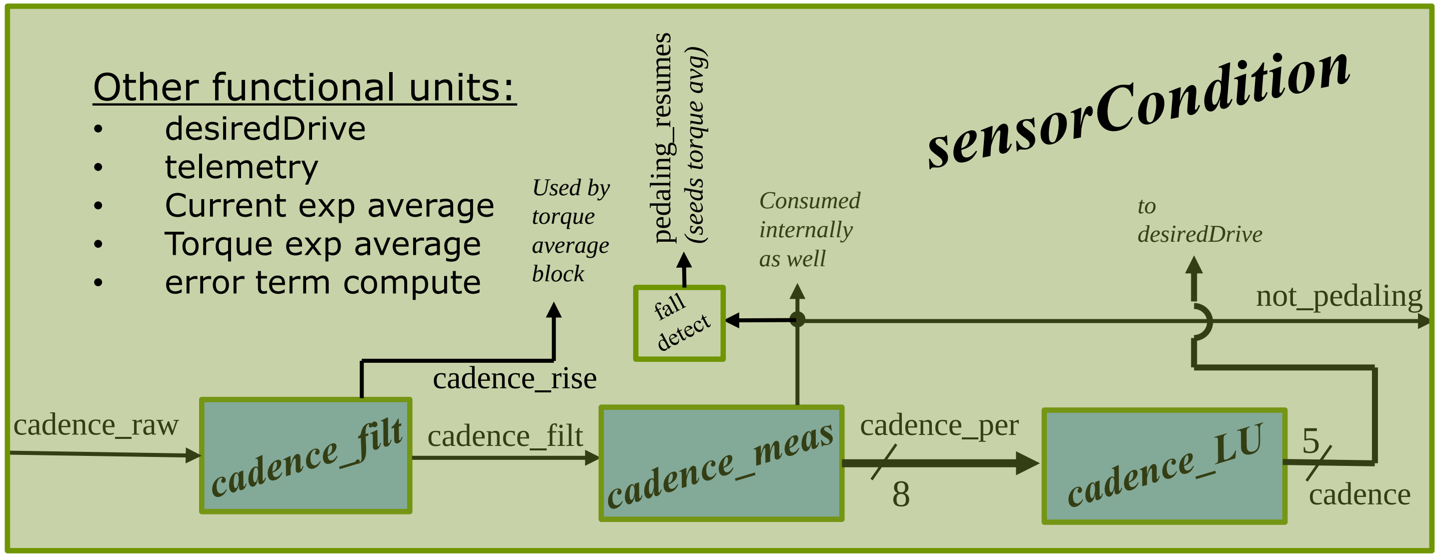
Study this implementation:

- On a rising edge of **cadence_filt** a 24-bit timer is cleared, but simultaneously its upper 8-bits are captured in an 8-bit register that forms **cadence_per**.
- If the 24-bit timer gets to **THIRD_SEC** then the timer is frozen at that value and the value is captured in the 8-bit **cadence_per** register. If this value equals **THIRD_SEC_UPPER** then the **not_pedaling** signal is asserted *(if no pulses were recorded in 1/3 of second then it is assumed the rider is not pedaling).*
- A parameter **FAST_SIM** is used for accelerated simulations and significantly shortens both the 1/3 second period (**THIRD_SEC**) and also grabs lower bits (14:7]) of the 24-bit counter to serve as **cadence_per**.

Cadence Blocks (putting it altogether inside sensorCondition)

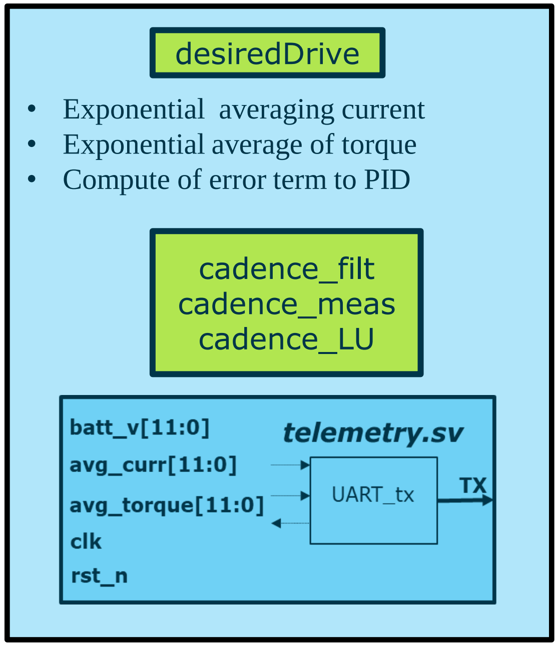
- ***cadence_LU*** is a provided block, and maps the 8-bit cadence period to a 5-bit cadence vector that is used by ***desiredDrive***. The relationship between cadence period and how much we want to assist the rider based on pedaling speed is both inverse and non-linear. Sometimes a Look Up table is the best way to handle this.

- There are other blocks in ***sensorCondition***, but you need to instantiate the 3 cadence related blocks in ***sensorCondition*** as shown.



sensorCondition (interface)

Signal:	Dir:	Description:
clk, rst_n	in	50MHz clock and asynch active low reset
torque[11:0]	in	Raw torque signal from A2D_intf
cadence_raw	in	Raw (unfiltered) cadence signal
curr[11:0]	in	Raw measurement of current through motor
incline[12:0]	in	Positive for uphill, negative for downhill
scale[2:0]	in	Assist level setting (off,low,med,high)
batt[11:0]	in	Raw battery voltage
error[12:0]	out	Signed error signal to PID
not_pedaling	out	Asserted when cadence_vec<2
TX	out	Output from telemetry module



sensorCondition (what is exponential average)

Many sensor readings require averaging. Running averages are great (responsive yet smoothing) but they are expensive. To perform a running average of 32 samples you have to keep the last 32 samples in a queue.

Exponential averages are a "poor man's running average". They are a way to get very close to the behavior of a running average without the expense of the queue.

An exponential average has an accumulator that is $\mathbf{log_2(W)}$ bits wider than the quantity of interest. So for example if we are to perform an exponential average of weight 16 (**W**=16) then we would need an accumulator 4-bits wider than the value we are averaging. If we are averaging a quantity coming from a 12-bit A2D converter we would need a 16-bit accumulator.

The accumulator (**accum**) would start at zero.

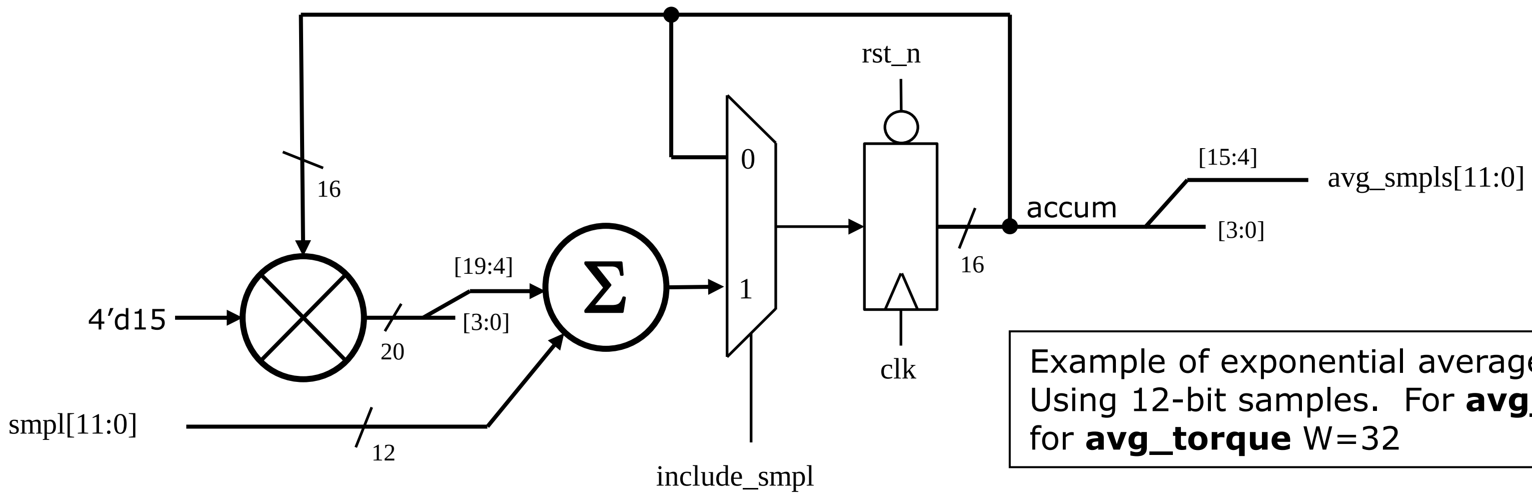
For every new sample (**smpl**) to be averaged the accumulator would be updated as:
  accum = ((**accum*(W-1))/W) + smpl**.

The average of the samples (**avg_smpls**) is then:
  avg_smpls = accum/W

Of course we are not idiots, so we would always choose **W** to be a power of two so there is no real division occurring...just shifting…or really just dropping of lower order bits.

The next slide gives a more visual representation of this for the example of **W**=16, with a 12-bit quantity.

sensorCondition (what is exponential average)



Example of exponential average with W=16 Using 12-bit samples. For **avg_curr** W=4, for **avg_torque** W=32

NOTE: new samples do not have to be included in the average every clock cycle (see ***include_smpl***)

For **avg_curr** a new sample will be included once every 2^{22} clocks (84ms) nominal, and once every 216 clocks (1.3ms) when **FAST_SIM** = 1.

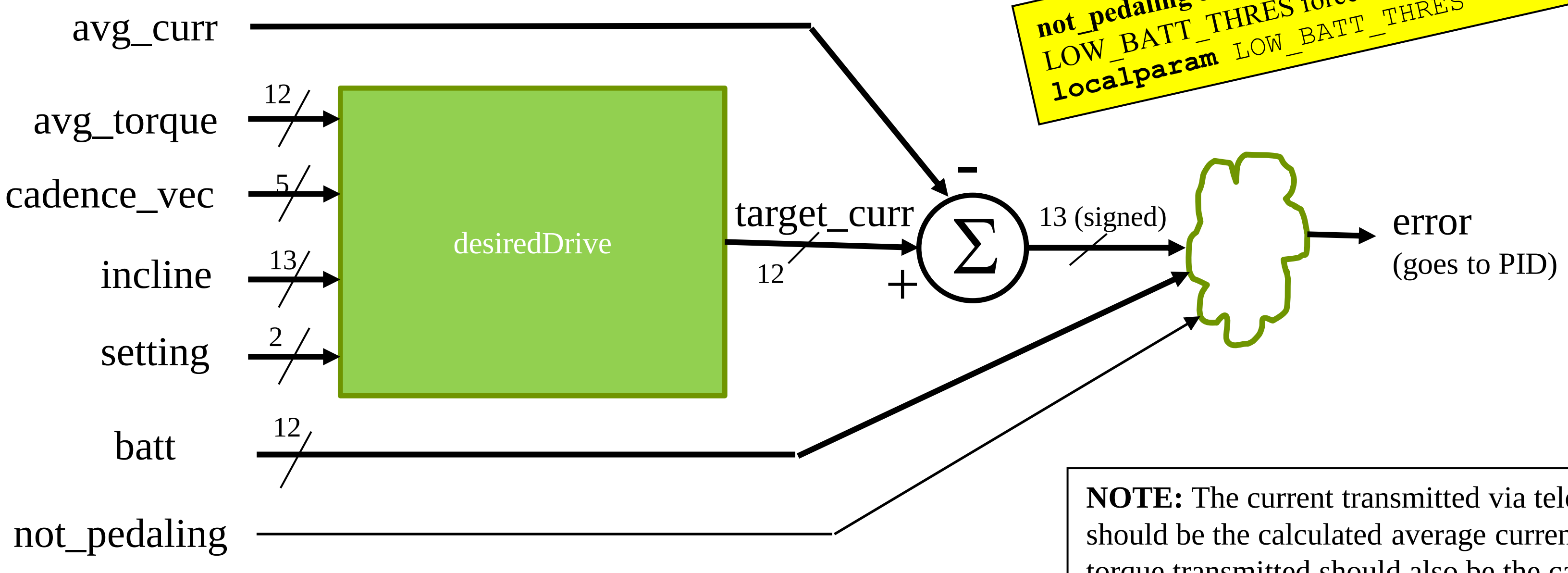
For **avg_torque** a new sample will be included using signal **cadence_rise** from ***cadence_filt***.

NOTE: the torque sensor has a large weight (W=32), so is slow to respond. When the rider resumes pedaling (fall of **not_pedaling**) we want to seed the accumulator proportional to the first torque reading (seed it with {1'b0,torque,4'b0000}). This makes the exponential average quicker to ramp up and provide to assist to the rider. This means the accumulator logic for torque will be a little more complex than what is shown here with a path for the seeded value to come in upon a fall of **not_pedaling (**i.e the signal **pedaling_resumes)**

sensorCondition (exponential avg coding hints)

- Code the exponential average functions for current (W=4) and torque (W=32) *(both 12-bit quantities being averaged)* flat inside ***sensorCondition***.
- You will need a 22-bit timer for determining when to include a new torque sample.
  - Use a generate-if statement based on FAST_SIM to make the
- Torque accumulator is a bit more complex than shown on previous slide in that it has a path to be seeded with {1'b0,**torque**,4'h0} if **pedaling_resumes**.
- Recall you need to compute the widths of the accumulators and what bits are discarded for divides based on W=4 or W=32. Can't just replicate what is shown on previous page.

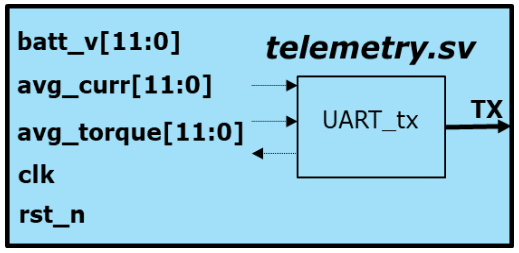
sensorCondition (other stuff)



not_pedaling or battery below LOW_BATT_THRES force error to zero
localparam `LOW_BATT_THRES = 12'hA98`

NOTE: The current transmitted via telemetry should be the calculated average current. The torque transmitted should also be the calculated average torque. So…in short, transmit the average values not the raw for current & torque.

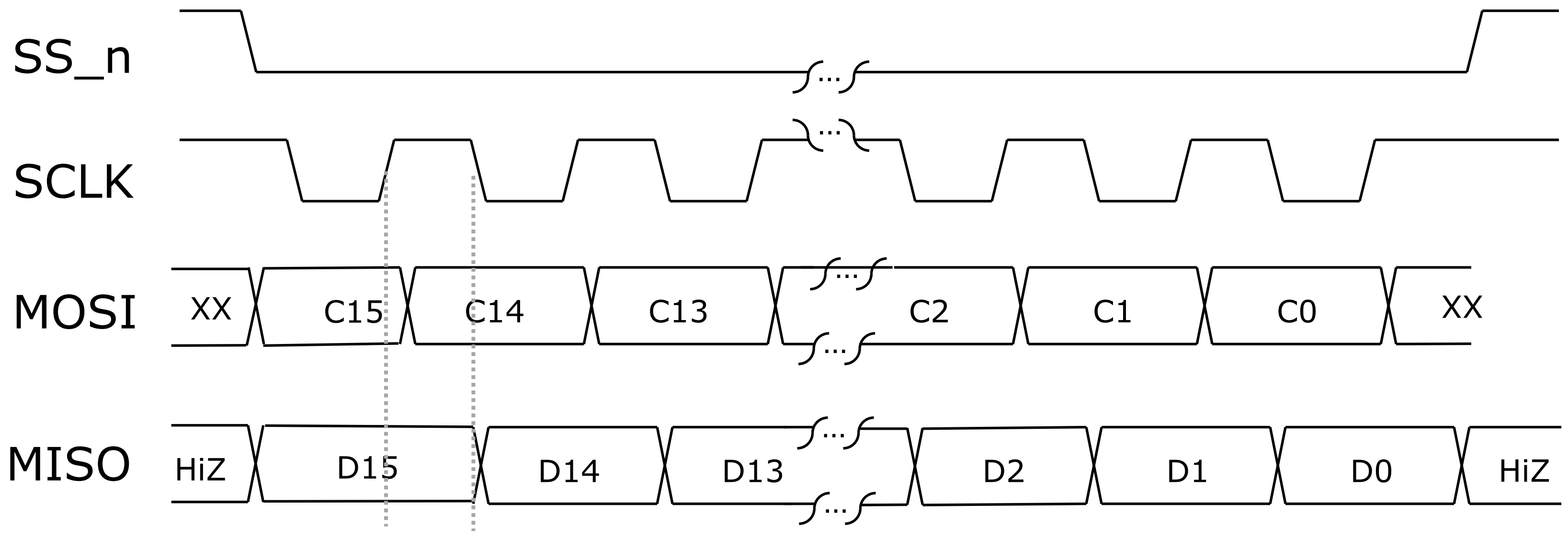
Since **sensorCondition** has so many of the signals vital to the operation of the eBike it is a good place to instantiate **desiredDrive** to compute loop **error** and **telemetry** to transmit results to an optional handlebar display unit.



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



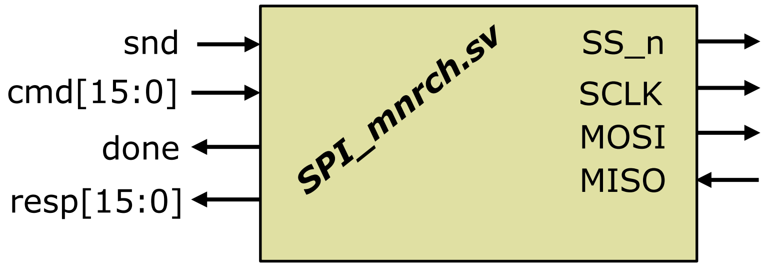
A SPI packet inherently involves a send and receive (full duplex). The full duplex packet is always initiated by the monarch. The monarch controls SCLK, SS_n, and MOSI. The serf drives MISO if it is selected. If the serf is not selected it should leave MISO high impedance. The inertial sensor is the 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 1/32 of our system clock (50MHz/32 = 1.5625MHz

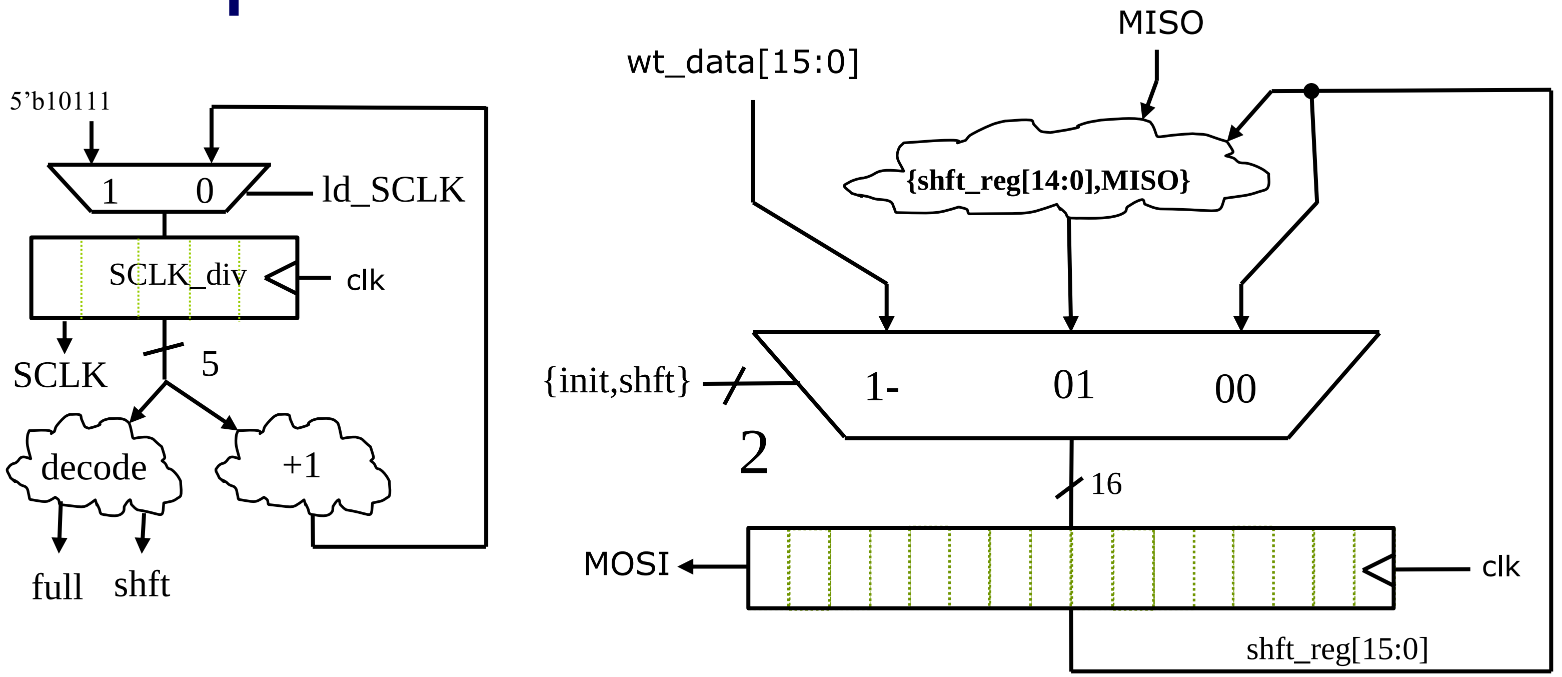
SPI Unit for Inertial Interface & A2D

- You will implement **SPI_mnrch.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**)
- 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. Perhaps more pertinent…when that 4-bit counter equals 5'b10001 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,MISO	3-out 1-in	SPI protocol signals outlined above
snd	in	A high for 1 clock period would initiate a SPI transaction
cmd[15:0]	in	Data (command) being sent to inertial sensor.
done	out	Asserted when SPI transaction is complete. Should stay asserted till next **wrt**
resp[15:0]	out	Data from SPI serf. For inertial sensor we will only ever use bits [7:0]

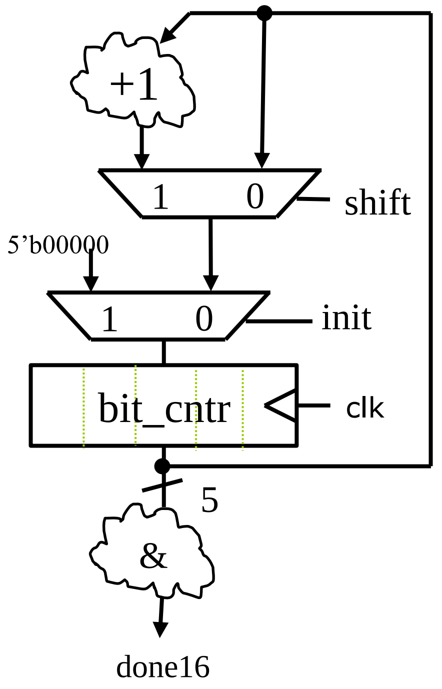
SPI Implementation

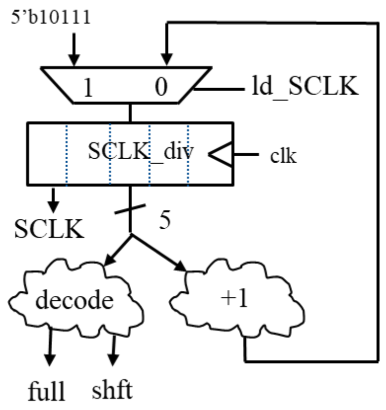


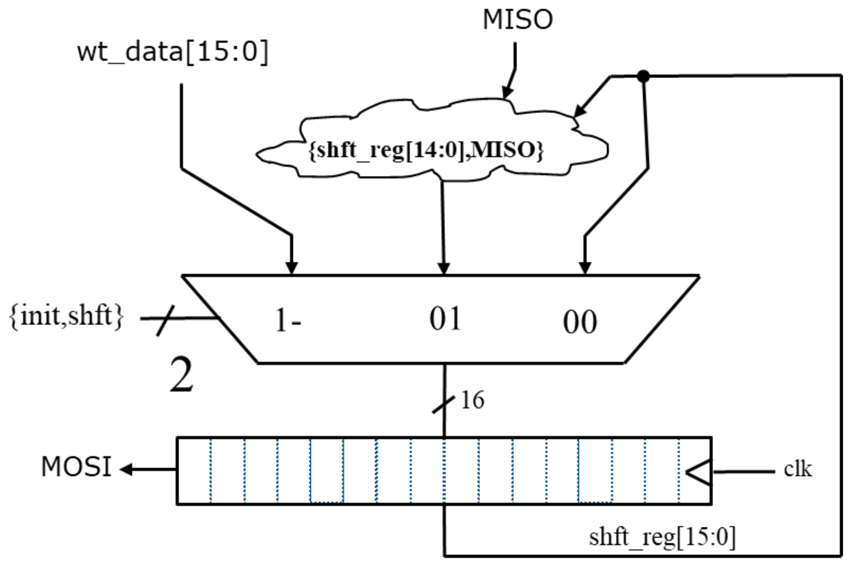
The main datapath of the SPI monarch consists of a 16-bit shift register. The MSB of this shift register provides **MOSI**. The shift register can be parallel loaded with the data to send, or it can left shift one position taking **MISO** as the new LSB, or it can simply maintain.

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 (5'b10111). 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 5'b10111?

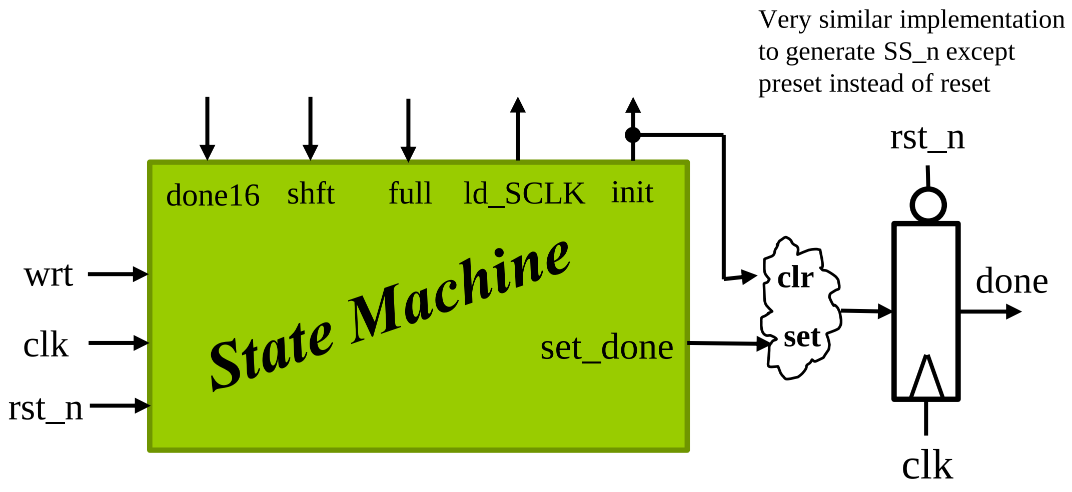
SPI Implementation







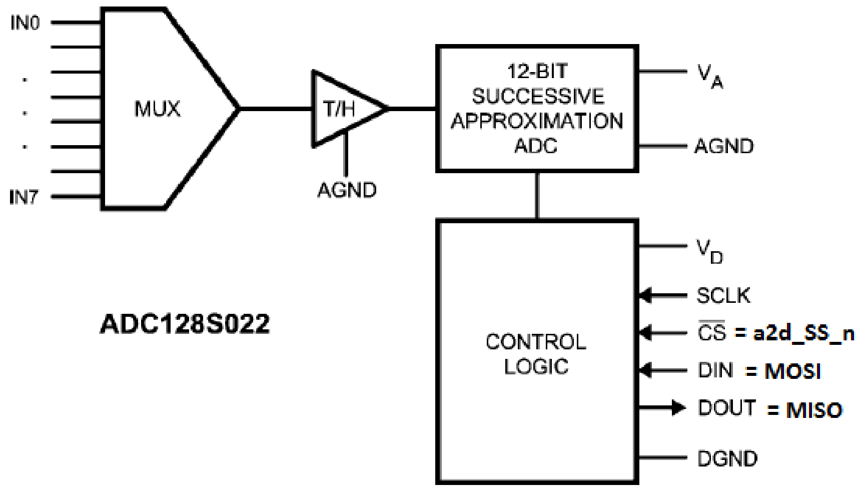
In addition to **SCLK_div** and main shift register you also need a **bit_cntr** to keep track of how many times the shift regiter has shifted. Of course you also need a state machine.


Very similar implementation to generate SS_n except preset instead of reset
done16
shft
full
ld_SCLK
init
State Machine
set_done
wrt
clk
rst_n
clr
set
rst_n
done
clk

A2D Converter (National Semi ADC128S022)

The ADC128S is a 12-bit eight channel A2D converter. Only one channel can be converted at a time. The A2D is read by via the SPI bus, and is used to convert the values of the six slide potentiometers.

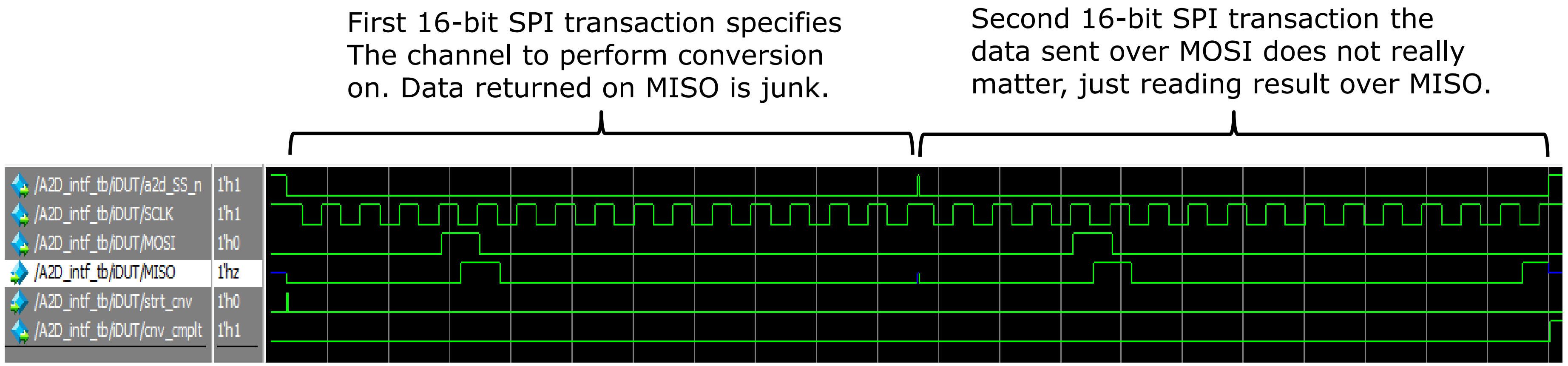
ADC Channel:	IR Sensor:
000 = addr	Battery voltage
001 = addr	Motor current
011 = addr	Brake lever
100 = addr	Pedal spindle torque



To read the A2D converter one sends the 16-bit packet {2’b00,addr,11’b000} twice via the SPI. There needs to be a 1 system clock cycle pause between the first SPI transaction completing, and the second one starting

During the first SPI transaction the value returned over MISO will be ignored. The first 16-bits are really setting up the channel we wish the A2D to convert. During the 2nd SPI transaction the data returned on MISO will be the result of the conversion requested in the previous transaction. Only the lower 12-bits are meaningful since it is a 12-bit A2D. The four channels of interest should be read in a round robin fashion with a 328usec delay between channels (full 14-bit count).

A2D Converter (Example SPI Read)


First 16-bit SPI transaction specifies The channel to perform conversion on. Data returned on MISO is junk.
Second 16-bit SPI transaction the data sent over MOSI does not really matter, just reading result over MISO.
/A2D_intf_tb/iDUT/a2d_SS_n 1'h1
/A2D_intf_tb/iDUT/SCLK 1'h1
/A2D_intf_tb/iDUT/MOSI 1'h0
/A2D_intf_tb/iDUT/MISO 1'hz
/A2D_intf_tb/iDUT/strt_cnv 1'h0
/A2D_intf_tb/iDUT/cnv_cmplt 1'h1

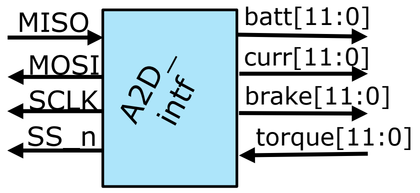
Our use of the A2D converter will involve two 16-bit SPI transactions nearly back to back (separated by 1 system clock cycle).

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 1:8 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 conversion.

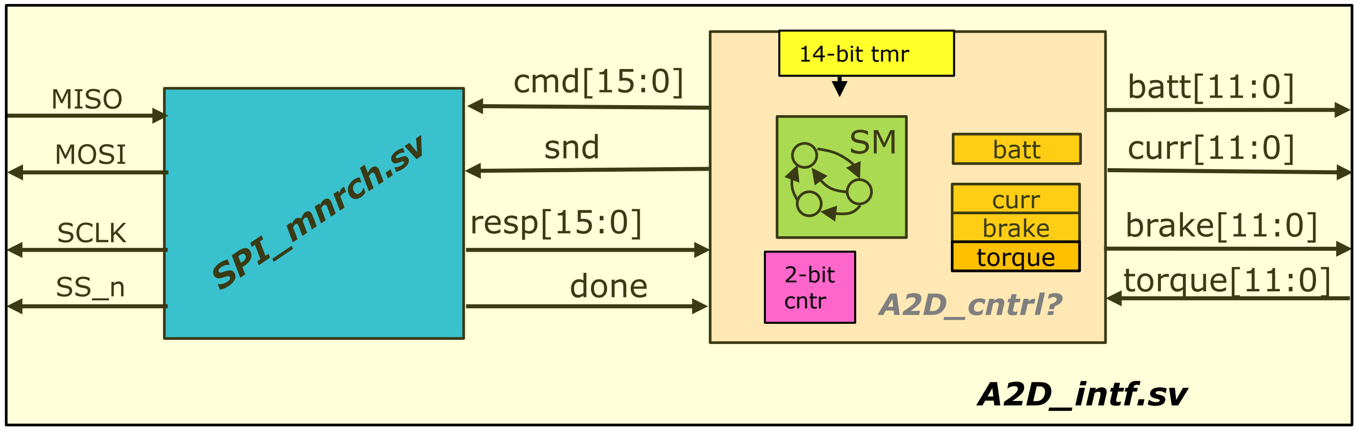
For the next 16-bit transaction the data sent over MOSI to the A2D does not matter that much. We are really just trying to get the data back from the A2D over the MISO line.

Note the timing of data vs SCLK edges. Note the behavior of SS_n. Note SCLK is normally high.

A2D Interface



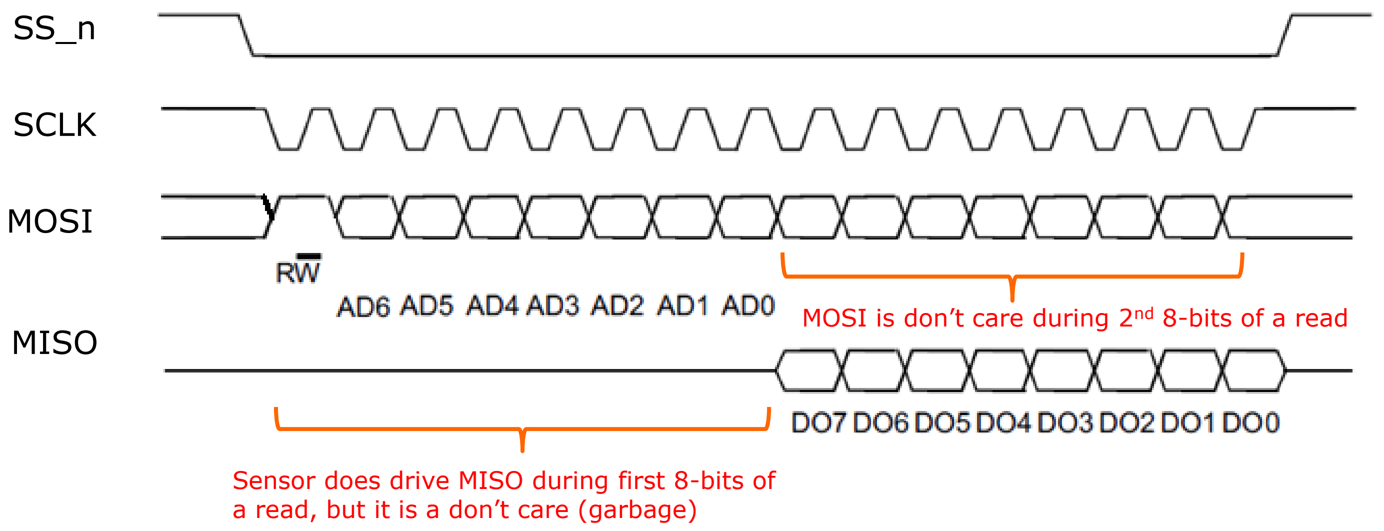
- You will build a block called **A2D_intf.sv**
- It will perform round robin conversions on channels (0,1,3,4) (battery voltage, motor current, brake lever, pedal torque)
- It maintains an internal 2-bit counter to know which measurand is next to be converted.
- A new channel is to be converted every 328usec (full 14-bit timer)



- You will make use of **SPI_mnrch**, and add control logic (State machine, some holding registers, a counter, and a timer) to produce **A2D_intf.sv**. Whether you wrap this control logic in a level of hierarchy (**A2D_cntrl**) or code it flat at the **A2D_intf** level is up to you.

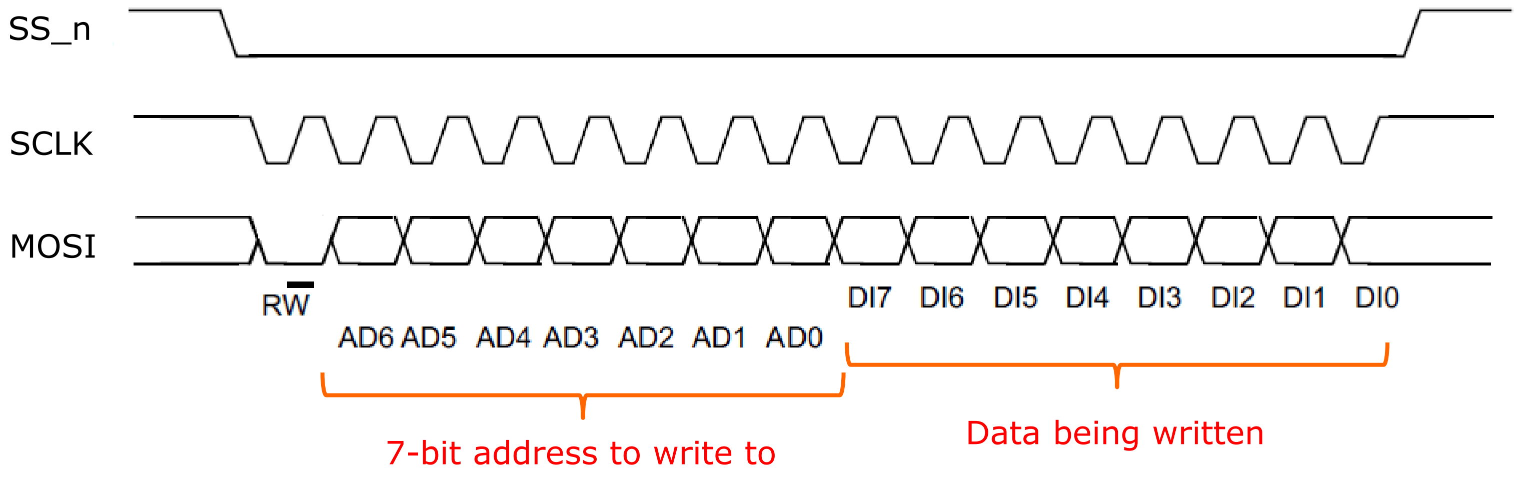
Inertial Sensor Interface

- 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)

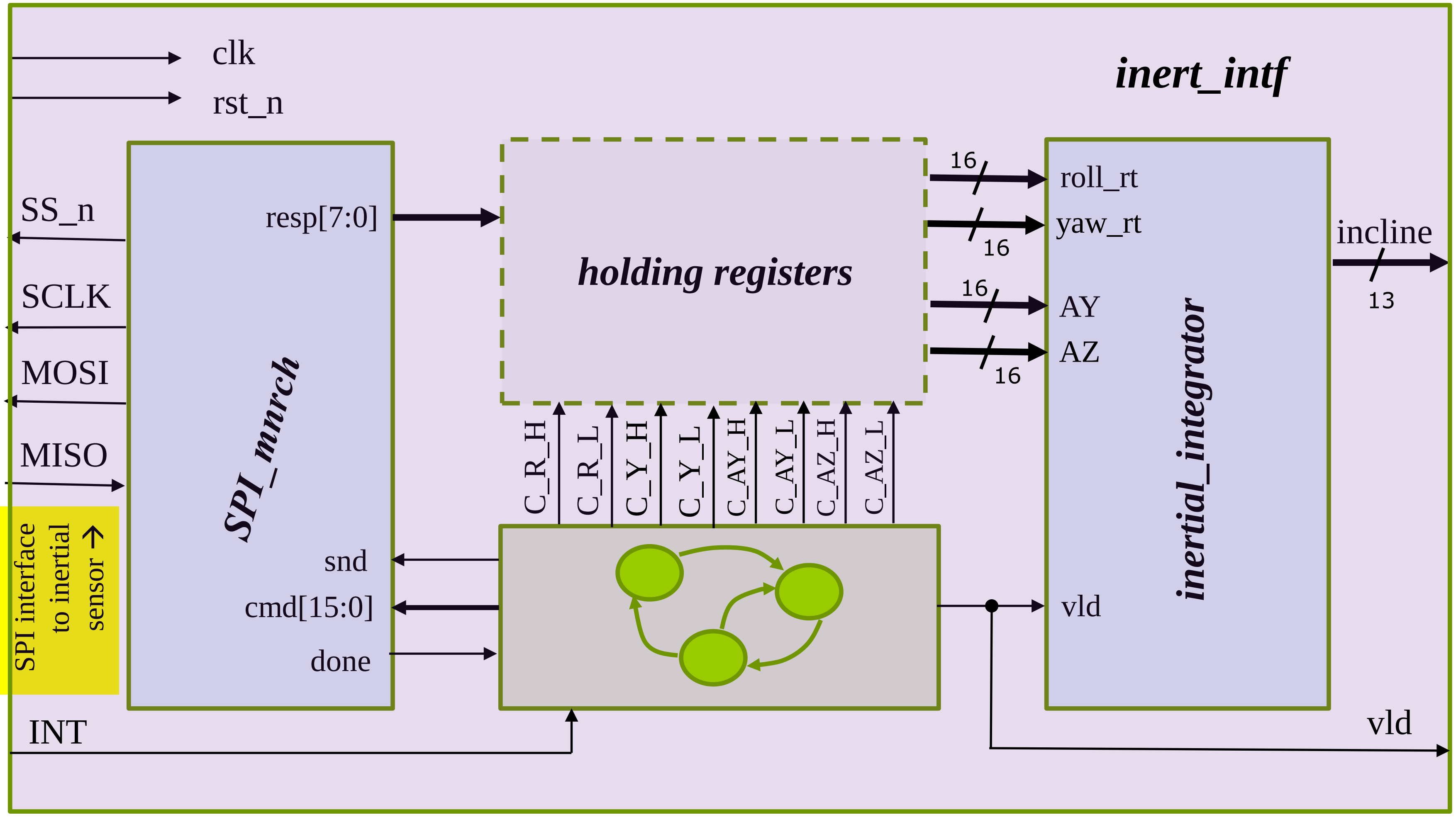
- During a write to the inertial sensor the first 8-bits specify it is a write and the address of the register being written. The 2nd 8-bits specify the data being written. (see diagram below)



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.

inert_intf (Inertial Interface Block Diagram)



Initializing Inertial Sensor

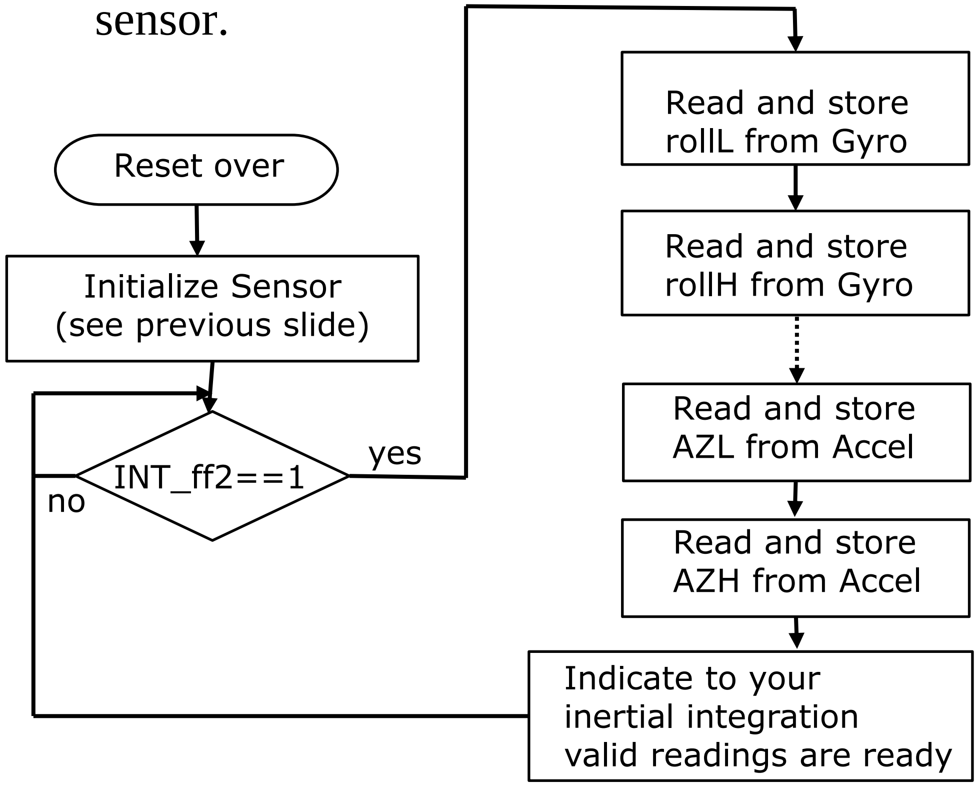
- After reset de-asserts the system must write to some registers to configure the inertial sensor to operate in the mode we wish. The table below specifies the writes to perform.

Addr/Data to write:	Description
0x0D02	Enable interrupt upon data ready
0x1053	Setup accel for 208Hz data rate, +/- 2g accel range, 50Hz LPF
0x1150	Setup gyro for 208Hz data rate, +/- 245°/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 (8 reads in all) from the inertial sensor.



Reset over
Initialize Sensor (see previous slide)
INT_ff2==1
yes
no
Read and store rollL from Gyro
Read and store rollH from Gyro
Read and store AZL from Accel
Read and store AZH from Accel
Indicate to your inertial integration valid readings are ready

- You will have eight 8-bit flops to store the 8 needed reading from the inertial sensor. These are: rollL, rollH, yawl, yawH, AYL, AYH, AZL and AZH. Even though only yaw and AY are needed to get incline we need roll and AZ to get the lean of the bike. Incline will be zeroed out if there is too much lean (incline becomes inaccurate during heavy turns).

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.

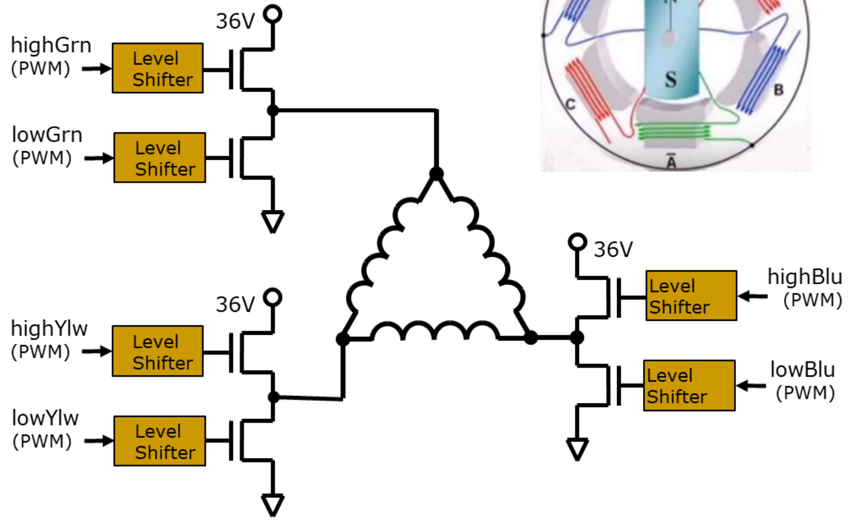
Addr/Data:	**Description:**
0xA4xx	rollL → roll rate low from gyro
0xA5xx	rollH → roll rate high from gyro
0xA6xx	yawL → yaw rate low from gyro
0xA7xx	yawH → yaw rate high from gyro
0xAAxx	AYL → Acceleration in Y low byte
0xABxx	AYH → Acceleration in Y high byte
0xACxx	AZL → Acceleration in Z low byte
0xADxx	AZH → Acceleration in Z high byte

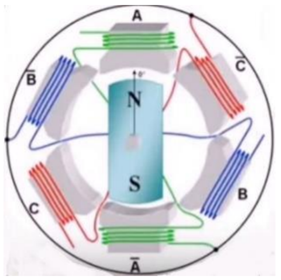
Inertial Integration (sensor fusion)

- The block **inertial_integrator.sv** takes the raw sensor readings and performs the sensor fusion. It computes both incline and roll internally, but only **incline** an output.
- If the magnitude of roll exceeds a threshold then the **incline** output is zeroed. When the rider is taking a hard corner (i.e. the bike is leaning (roll) due to the turn) then the incline measurement becomes inaccurate and is zeroed.
- The Verilog for **inertial_integrator.sv** is provided. It is not really that complex of a block and I am sure you could have handled it. It is, however, difficult to specify, plus you folks already have enough to do.
- I do encourage you to take a look at the code for **inertial_integrator.sv** just to satisfy your curiosity.

Brushless (how to drive the coils)

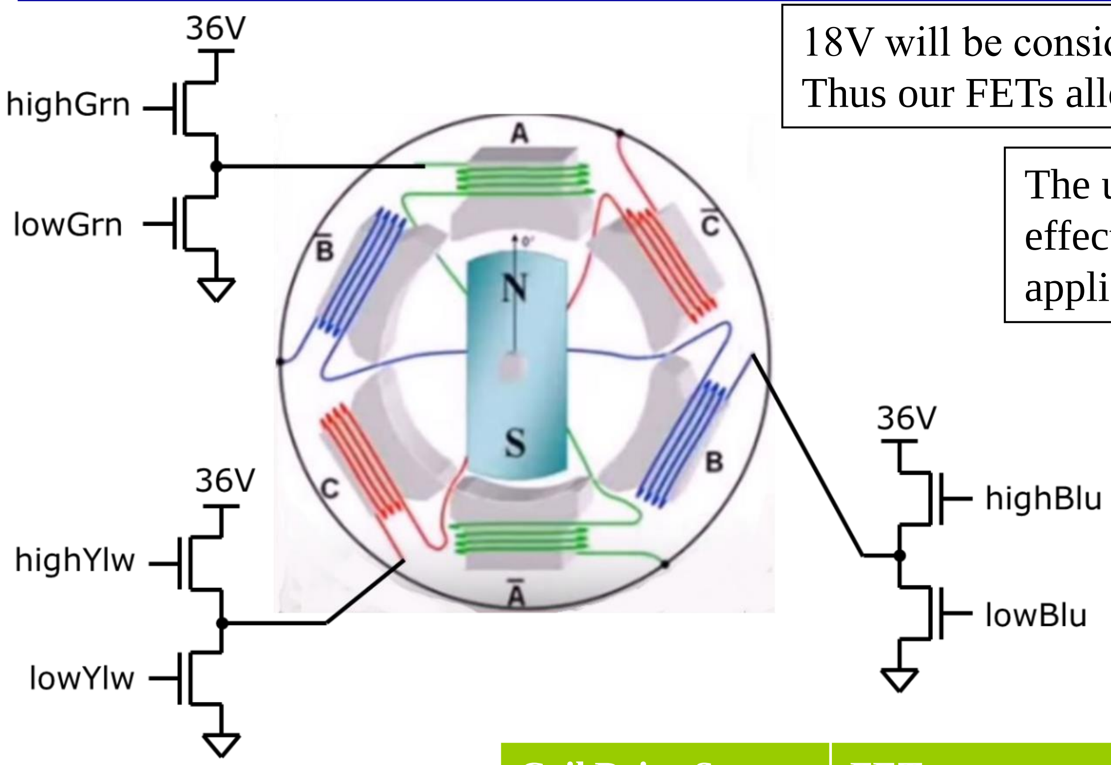
- We have 3 coil connections that we have to drive. We call them green, yellow, and blue. For any given coil we drive current in one direction, the opposite direction, or not at all. Plus a 4th condition for dynamic braking.
- How do we know the proper coil drive at any given time? That is determined by the position of the rotor. We know the rotor position from the hall sensor inputs.





- **brushless.sv** will be the block that inspects the hall sensor signals and determines how to drive each coil. Each coil can be driven 1 of 4 ways: not driven (both high/low FETs off); driven "forward"; driven "reverse"; driven for dynamic braking (high FET off, low FET PWMed)
- Are the hall effect sensors synchronous to our clock domain? What does that mean?
- If you are curious why PWMing the lower FETs creates dynamic braking ask me in person, but only if you know some basics about power conversion or power electronics.

Brushless (how to drive the coils)



18V will be considered "virtual ground" for the coils. Thus our FETs allow us to drive positive or negative.

The use of PWM allows us to control the effective magnitude of the voltage applied across the coil. (hence speed/torque)

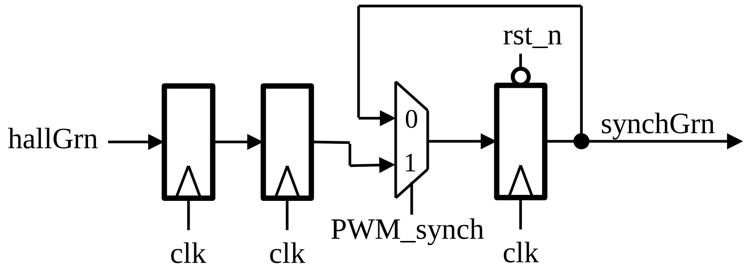
A PWM setting of 0x400 would be 50% and would represent driving a coil with 18V (virtual ground).

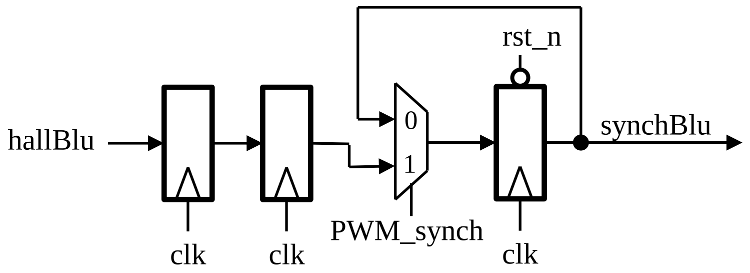
There are 4 possible states a coil can be driven in. Regenerative braking is a special case indicated by **brake_n** signal being low.

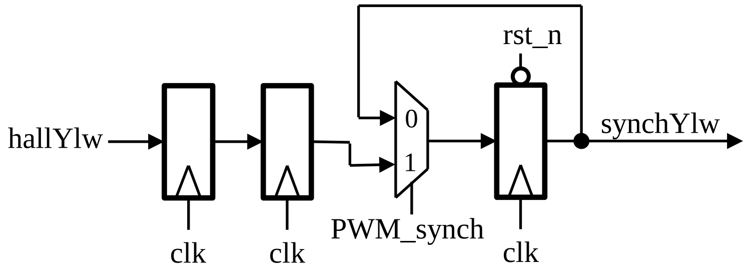
Coil Drive State:	FET gate controls:
Not driven (High Z)	Both high and low side low (both off)
Forward Current	High side driven with PWM, low side driven with ~PWM
Reverse Current	High side driven with ~PWM, low side driven with PWM
Regen Braking	High side low (off), low side driven with PWM

Brushless (synchronizing Hall Effect sensors)

- There are two forms of synchronizing we must concern ourselves with:
  - Synchronizing Hall sensors to our clock domain….ie. Double flopping
  - Synchronizing commutation to our PWM cycle
    - Coil drive and commutation are determined by hall sensor readings
    - Hall sensor readings are not correlated to our PWM cycle *(change in hall sensor reading could happen anywhere in duty cycle of our PWM).*
    - We ideally would like to switch commutation when all power FETs are off *(i.e. when the PWM is in its deadtime)*
    - So we want to create a set of hall sensor signals that are correlated to our PWM. We will use the **PWM_synch** signal to synchronize hall readings to the PWM cycle *(and thus ensure commutation occurs during the deadband nonoverlap block enforces)*







The hall effect sensor wires are also green, yellow, & blue.

Infer the shown synchronizer circuits to form double synchronized signals for all 3 hall effect signals.

Brushless (how to drive the coils)

Determining next drive conditions from hall effect sensor readings:

- The hall effect sensors tell us the current position of the rotor
- The hall effect sensor wires are also green, yellow, and blue.
- Form a 3-bit vector:

  assign rotation_state = {synchGrn,synchYlw,synchBlu};
- The following table outlines how we drive the coils relative to **rotation_state**

rotation_state	**3'b101**	**3'b100**	**3'b110**	**3'b010**	**3'b011**	**3'b001**
coilGrn	for_curr	for_curr	High Z	rev_curr	rev_curr	High Z
coilYlw	rev_curr	High Z	for_curr	for_curr	High Z	rev_cur
coilBlu	High Z	rev_curr	rev_curr	High Z	for_curr	for_curr

- In the case of **brake_n == 1'b0** (braking) all coils are driven in the regenerative braking state with the high side FET off and the low side FET PWMing.

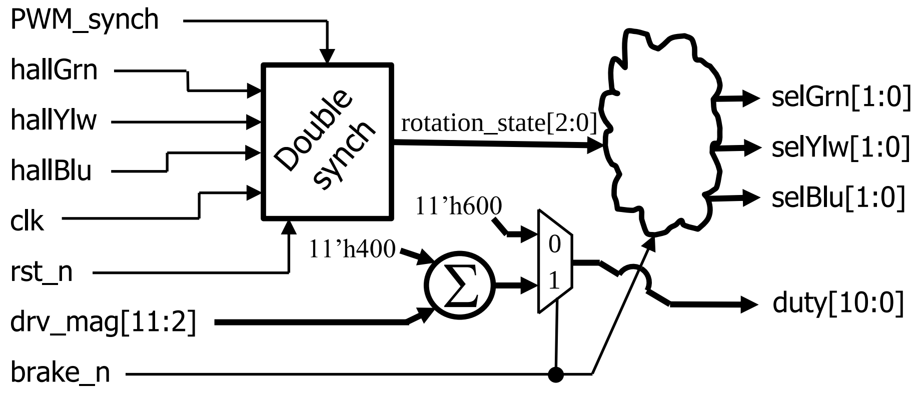
Brushless (how to drive the coils)

Signal:	Dir:	Description:
clk, rst_n	in	50MHz clock & asynch active low reset
drv_mag[11:0]	in	From PID control. How much motor assists (unsigned)
hallGrn,hallYlw, hallBlu	in	Raw hall effect sensors (asynch)
brake_n	in	If low activate regenerative braking at 75% duty cycle
PWM_synch	in	Used to synchronize hall reading with PWM cycle
duty[10:0]	out	Duty cycle to be used for PWM inside ***mtr_drv***. Should be 0x400+drv_mag[11:2] in normal operation and 0x600 if braking.
selGrn[1:0], selYlw[1:0], selBlu[1:0]	out	2-bit vectors directing how ***mtr_drv*** should drive the FETs. 00=>HIGH_Z, 01=>rev_curr, 10=>frwd_curr, 11=>regen braking

Code and test ***brushless.sv*** with the interface specified in this table.

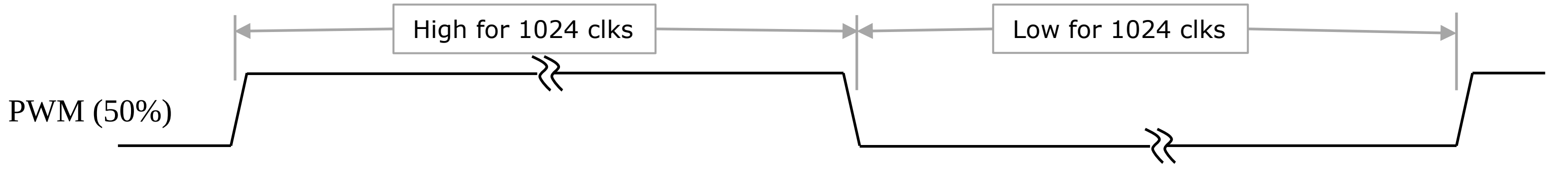
Submit: **brushless.sv**.

In next exercise we will work on testing it.

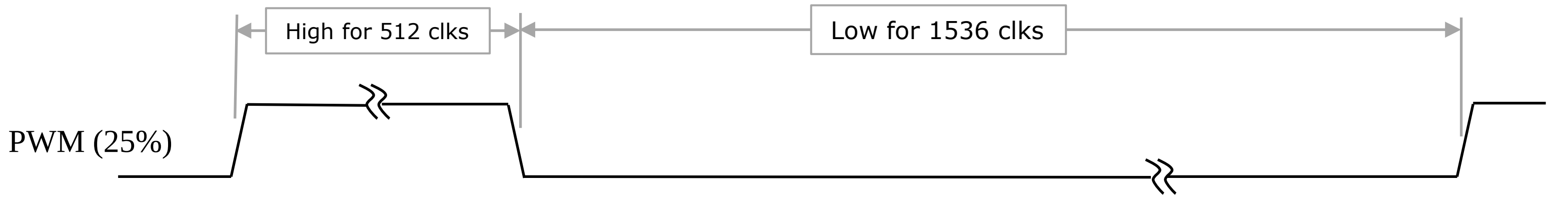


PWM (vary strength of motor drive)

- We have to have a way of varying the strength of the drive to the eBike motor. This will be done through **P**ulse **W**idth **M**odulation (PWM).
- PWM is commonly used as a simple way of varying intensity. It can be used on an LED. Turn the LED on at full brightness for 100usec then off for 100usec. The human eye will average the light intensity (your retina integrates), so the light will look like the LED is driven at ½ intensity.
- The same works with motor control. Drive the motor coil at full voltage for 50usec and off for 150usec. The inductance in the coil will "average" the current and it will look like the motor is driven at 25%.
- Consider an 11-bit PWM signal being driven at 50% duty cycle. The period of the PWM waveform is 2048 clocks (2^{11}). Since our system clock is 50MHz this is 40usec.



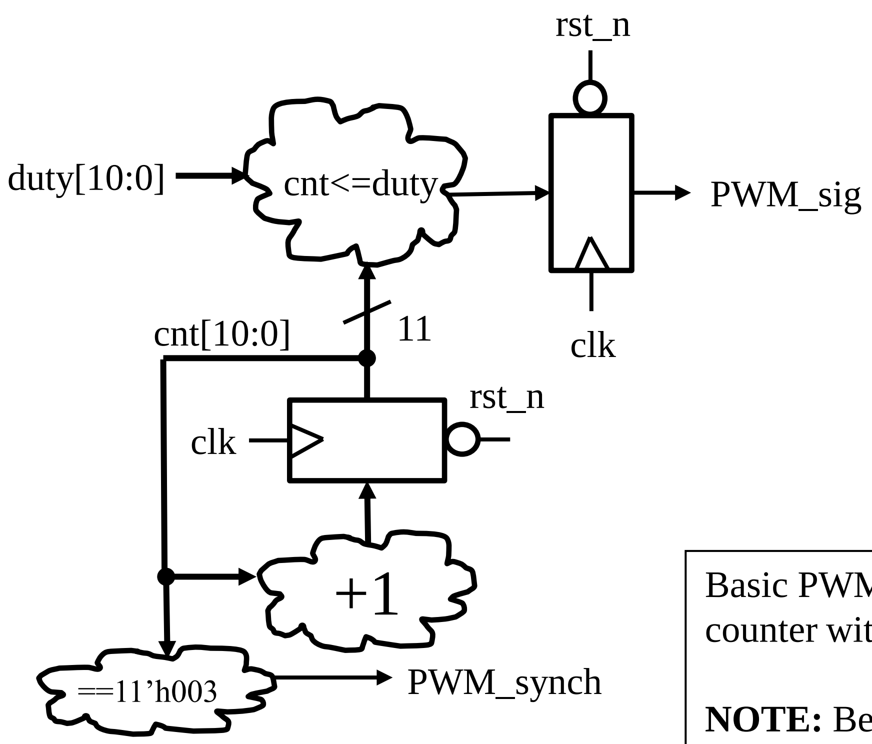
- Second example is 25% duty cycle



PWM (implementation)

A PWM module cannot achieve both zero duty cycle and 100% duty cycle. Think about it. If zero means zero duty then what does 0x3FF mean? It means we are on for 2047 out of 2048 clocks, so not quite 100%. We are fine with this.

Signal:	Dir:	Description:
clk	in	50MHz system clk
rst_n	in	Asynch active low
duty[10:0]	in	Specifies duty cycle (unsigned 11-bit)
PWM_sig	out	PWM signal out (glitch free)
PWM_synch	out	When cnt is 11'h003 output a signal to allow commutator to synch to PWM

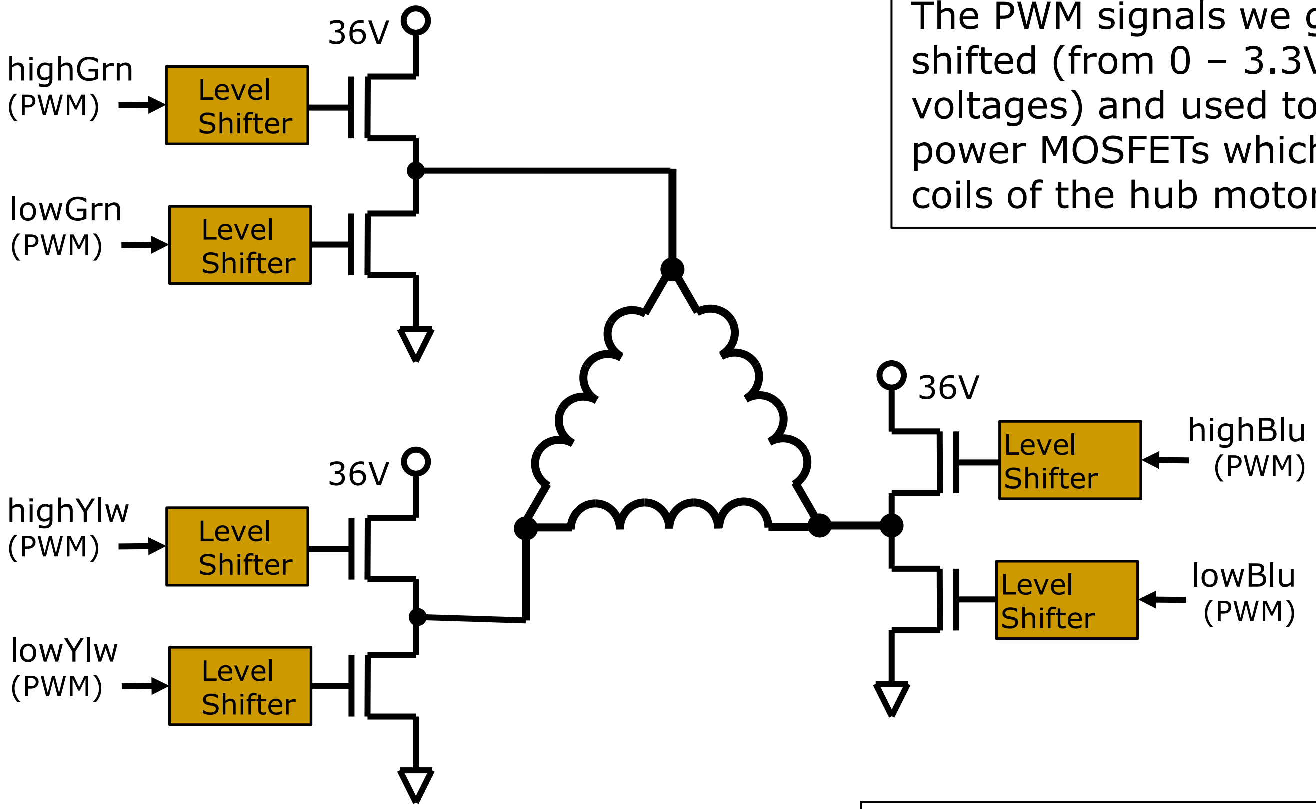


When we implement our brushless DC motor driver we will want to synch its commutation to the PWM period.

Basic PWM implementation is not too hard. Just an 11-bit counter with simple comparison logic.

NOTE: Because we use our PWM signal to switch power MOSFETs we cannot afford for it to glitch, therefore, it must come directly out of a flop.

Non-Overlap (Power MOSFETs will self destruct if allowed)

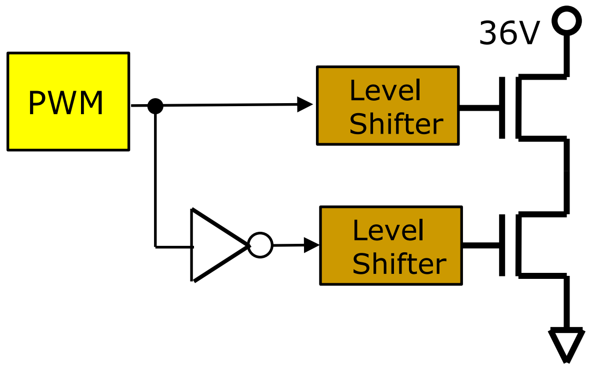


The PWM signals we generate are level shifted (from 0 – 3.3V signals to higher voltages) and used to drive the gates of power MOSFETs which in turn drive the coils of the hub motor.

The level shifters have some delay in their rise/fall times (in the 1 to 2usec vicinity). There is also some variation in the delay of the high driver from the low driver.

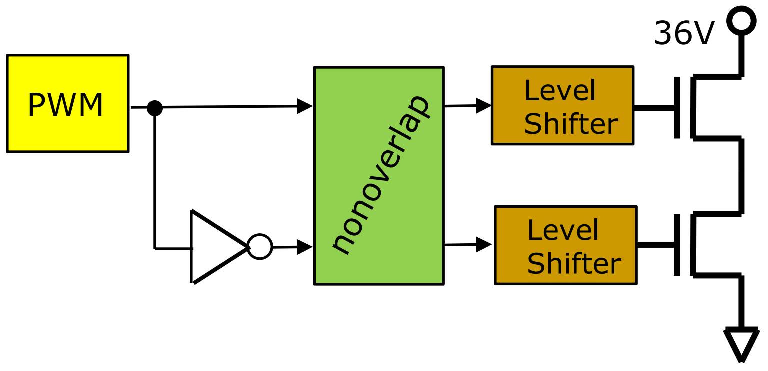
The power MOSFETs are very powerful (very low ohmic and capable of passing a lot of current…like 100A). Given the proper conditions they can self-destruct. Actually that would be the "improper conditions".

Non-Overlap (Power MOSFETs will self destruct if allowed)



Given the very low ohmic nature of power MOSFETs and the slow slope and variation in the level shifting gate drivers…do you see a problem with this configuration?

If both the high and low FETs are on at the same time (even for a fraction of a μsec) then hundreds of amps could flow from 36V to GND.



Signal:	Dir:	Description:
clk, rst_n	In	50MHz clock, and reset
highIn	In	Control for high side FET
lowIn	In	Control for low side FET
highOut	Out	Control for high side FET with ensured non-overlap
lowOut	Out	Control for low side FET with ensured non-overlap

We need a non-overlap block that ensures the high gate drive and low gate drive will never overlap. This non-overlap block will create a dead time where both output signals are low for a while whenever an input changes.

Non-Overlap (Power MOSFETs will self destruct if allowed)

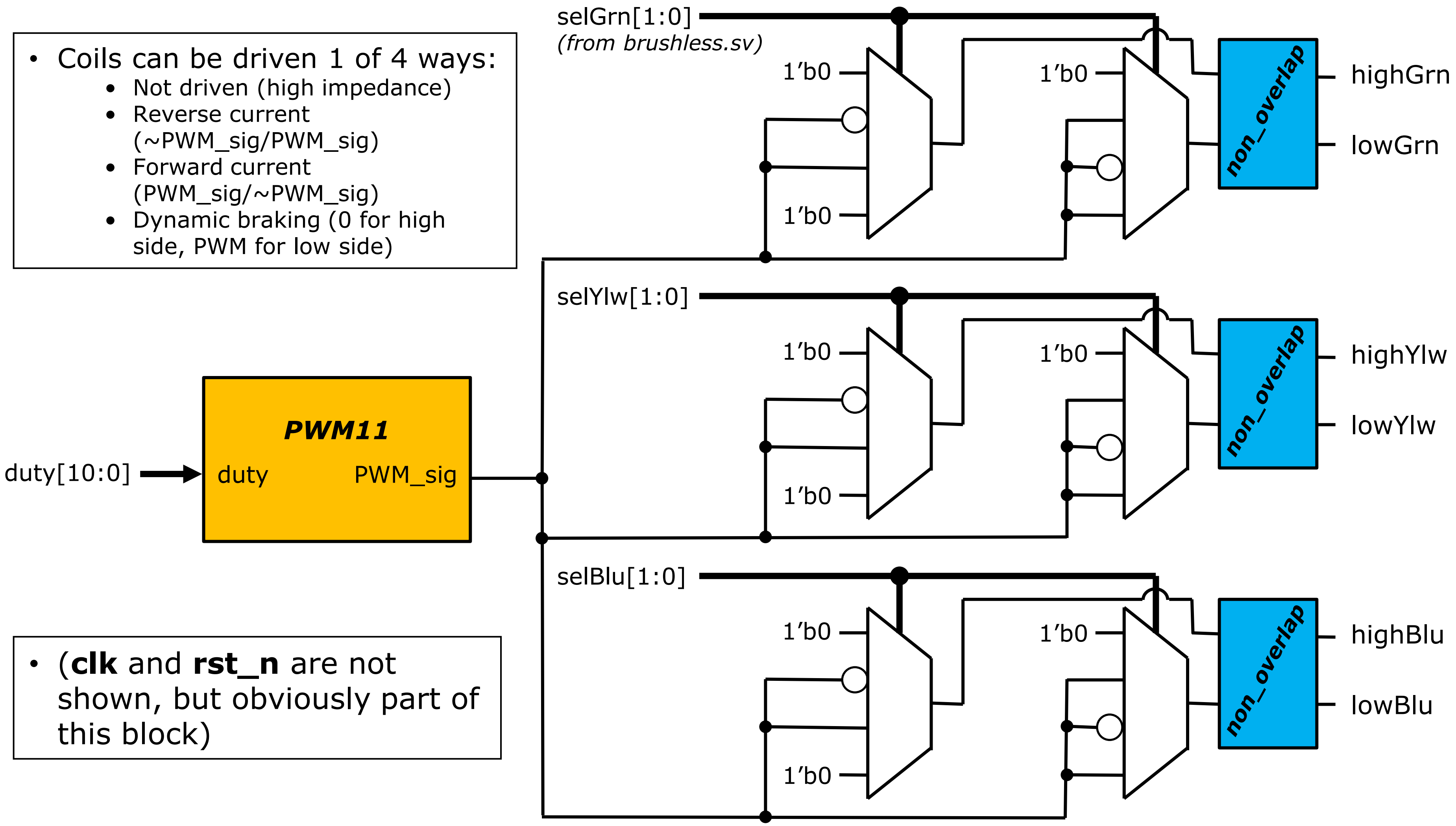
nonoverlap.sv specifications:

- Whenever ***highIn*** or ***lowIn*** change both ***highOut*** and ***lowOut*** should go low on the next clock cycle.
- Once ***highOut*** and ***lowOut*** are forced low (from a change in either) they should remain forced low for **32** system clocks.
- Both ***highOut*** and ***lowOut*** should come directly from flops so they cannot glitch (it is always possible for the output of combination logic to glitch).
- If ***highOut*** and ***lowOut*** are not being forced low (from a change) they should simply take their value from ***highIn*** and ***lowIn*** respectively.

This block along with **PWM11** will be used inside **mtr_drv.**

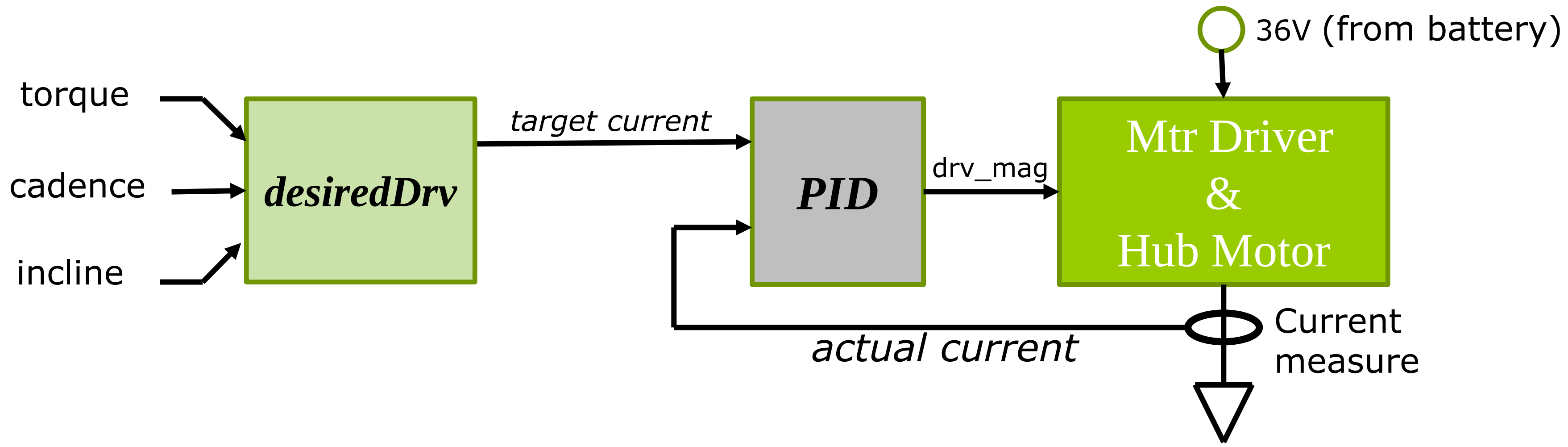
mtr_drv

- Coils can be driven 1 of 4 ways:
  - Not driven (high impedance)
  - Reverse current (~PWM_sig/PWM_sig)
  - Forward current (PWM_sig/~PWM_sig)
  - Dynamic braking (0 for high side, PWM for low side)



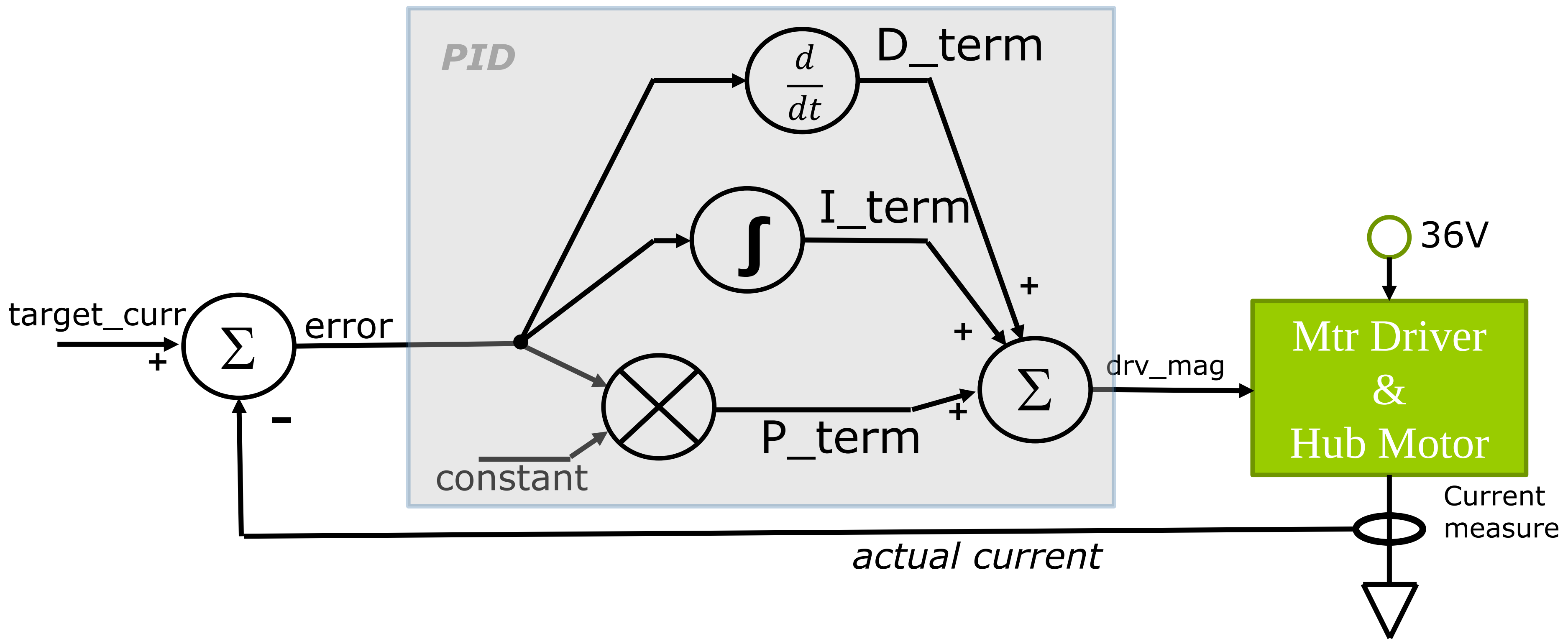
- (**clk** and **rst_n** are not shown, but obviously part of this block)

PID Controller



- There is no direct calculation that can be done to convert **target_curr** into **drv_mag**. This relationship has to be determined by a PID controller that will vary **drv_mag** until the actual current matches the target current.
- In our implementation the subtraction: **error = target_curr – avg_curr** occurs in the ***sensorCondition*** block, and **error** feeds directly into the PID block.

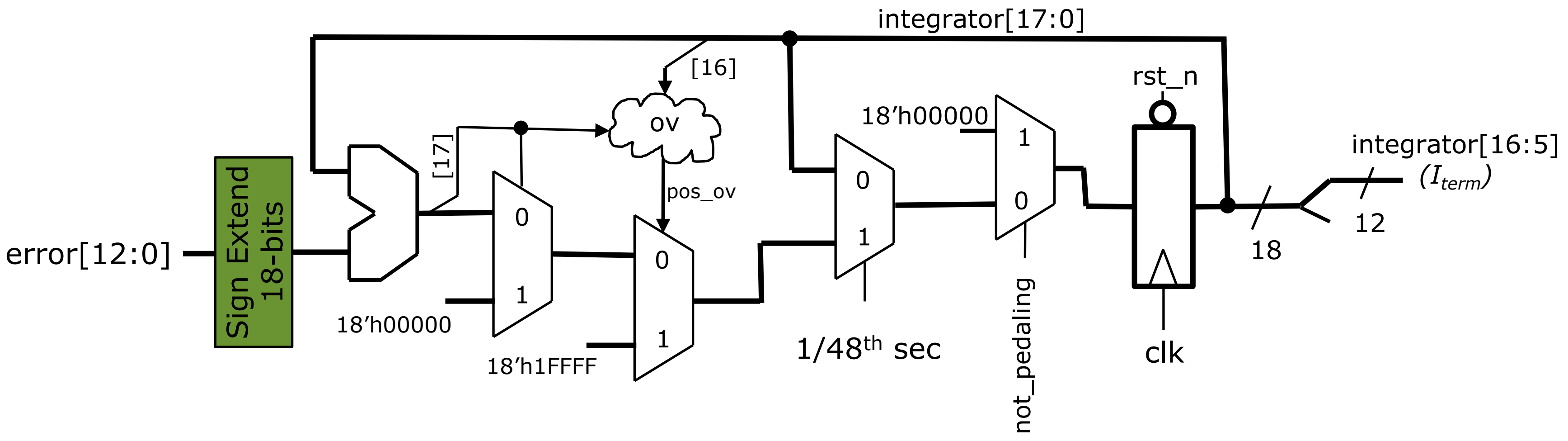
PID Controller



- 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**.

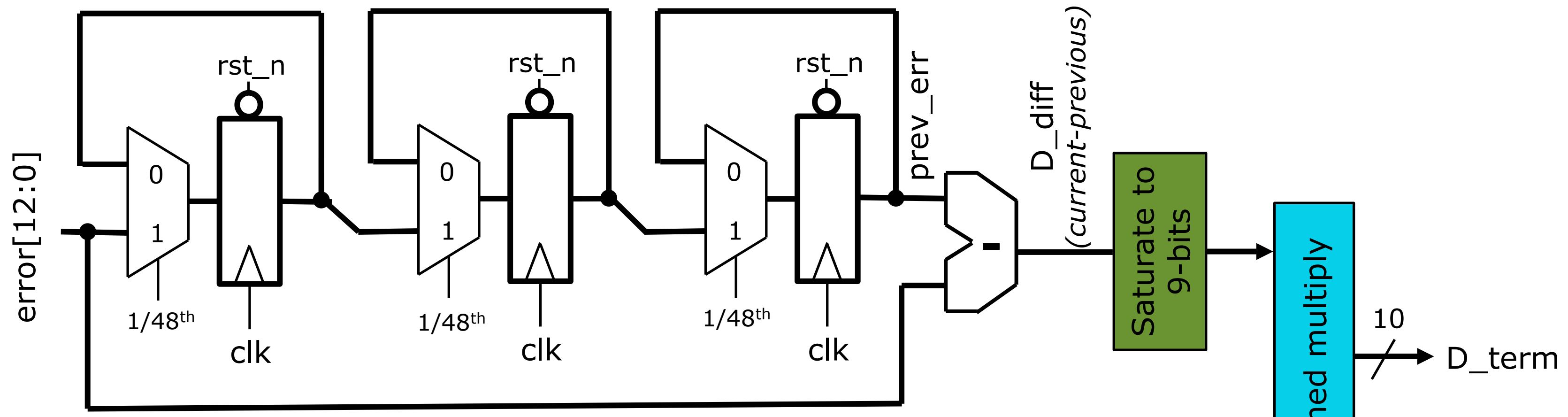
PID Controller (P_term & I_term)

- 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.

PID Controller (D_term)

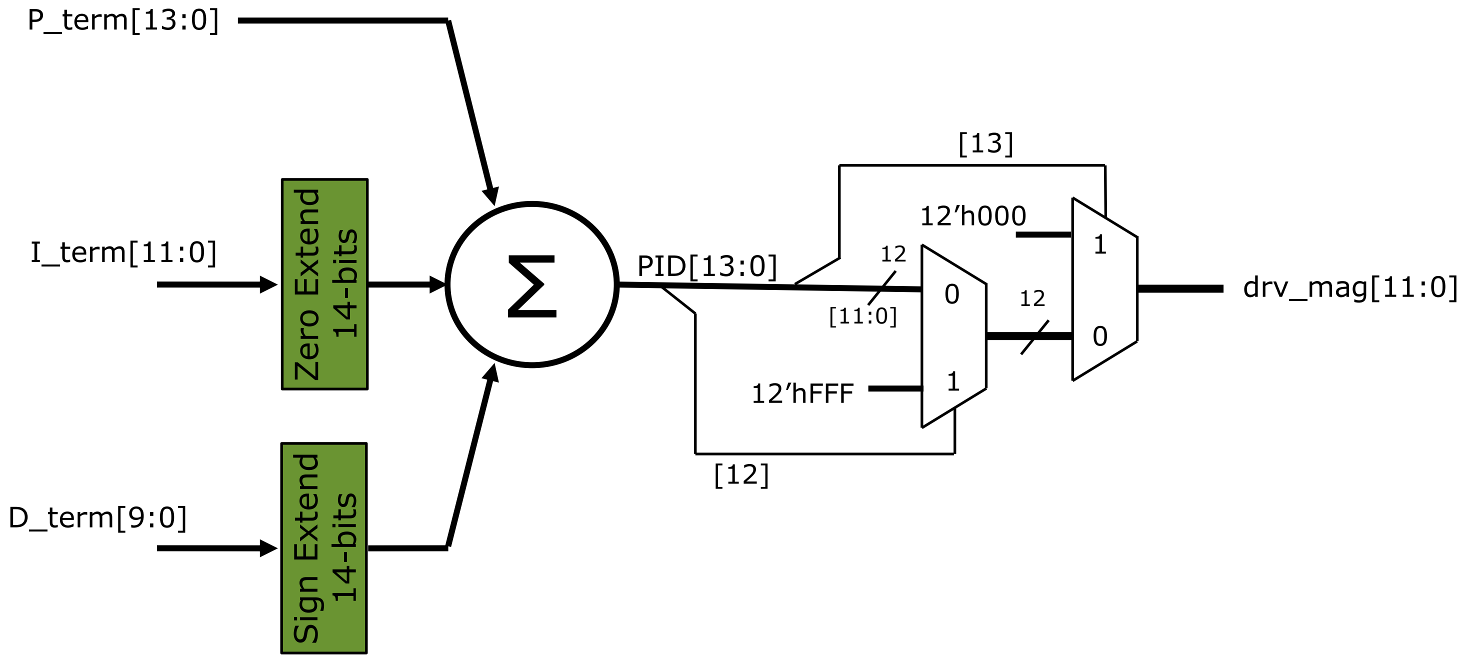


$$\frac{df(t)}{dt}=\frac{f(t)-f(t-\Delta t)}{\Delta t}$$

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 scaled by a factor of 2. So the "multiply" can really just be a signed extended left shift of 1-bit.

PID Math (Putting **PID** together)

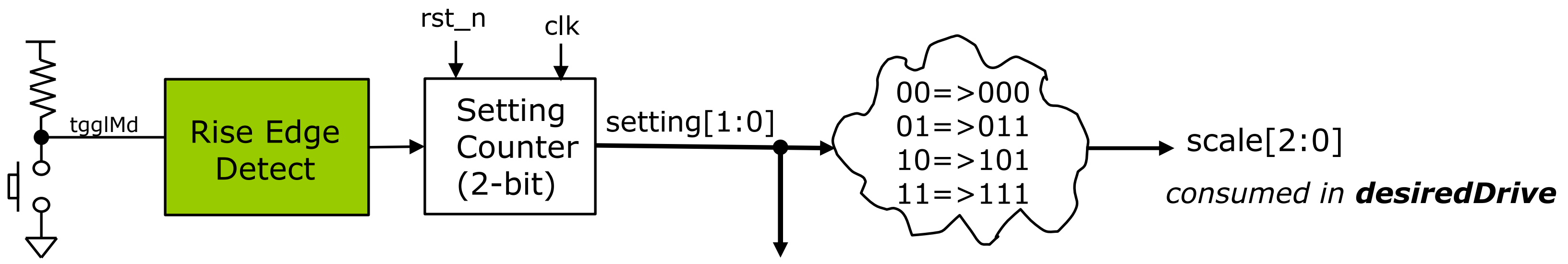


We sum the 3 terms together to form a 14-bit signal ***PID***. ***drv_mag*** is an unsigned numbers (e-bikes don't assist you to go in reverse). Is it possible for PID to be negative? Possibly, ***P_term*** and ***D_term*** could be negative while ***I_term*** was small. If ***PID*** is negative we clip it to zero. If ***PID*** exceeds 0xFFF we saturate it to 0xFFF.

drv_mag is the signal that goes to **mtr_drv** and is eventually converted to a PWM signal to control motor drive magnitude.

In addition to what was shown in these slides the PID block needs a timer to determine 1/48th sec intervals.

Setting Push Button Interface.



Also an output that drives 2 least significant LEDs on DE0-Nano board so user has an indication of what the setting is.

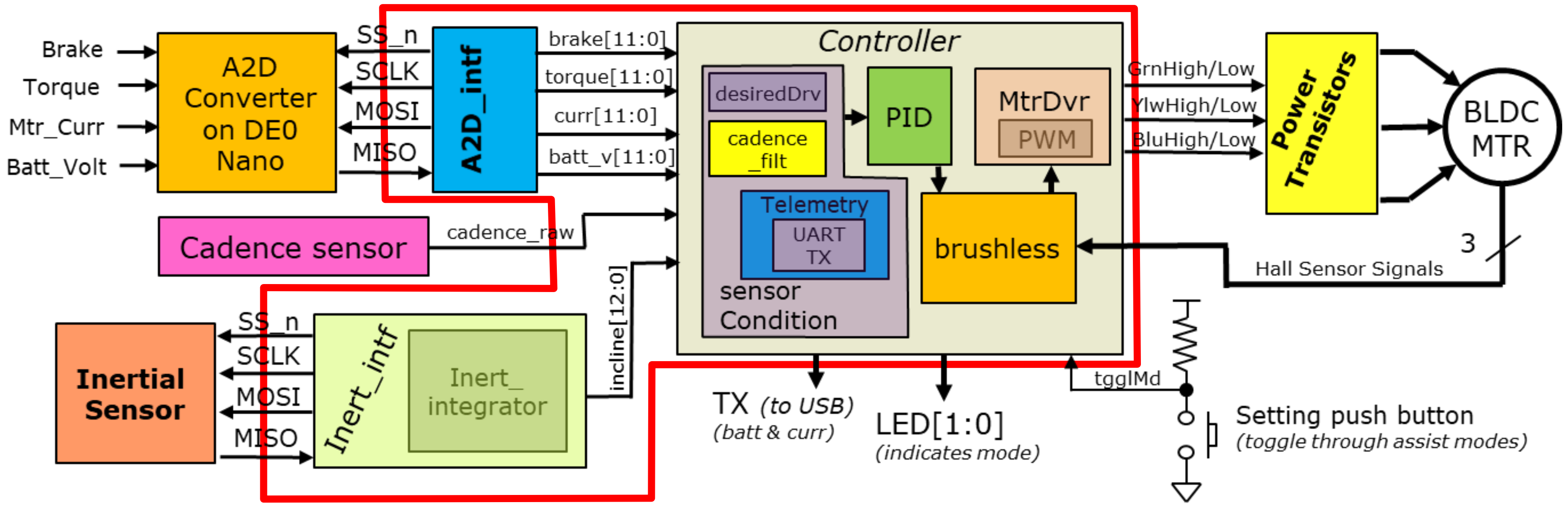
The last thing you have to make is rather trivial. There is a push button hooked to a signal called **tgglMd**. Every time it is released the 2-bit setting (00=>off, 01=>low assist, 10=>medium assist, 11=> max assist) should be incremented.

NOTE: setting should **default to 2'b10** upon reset (medium assist).

Using dataflow infer a combinational block to map the **setting** to a **scale**[2:0] factor.

Recall a pushbutton is asynch to our clock domain so the rise edge detector should have 3 flops in total.

Required Hierarchy & Interface



You **must** have a block called **eBike.sv** which is the top level of what will be synthesized. Shown in red here.

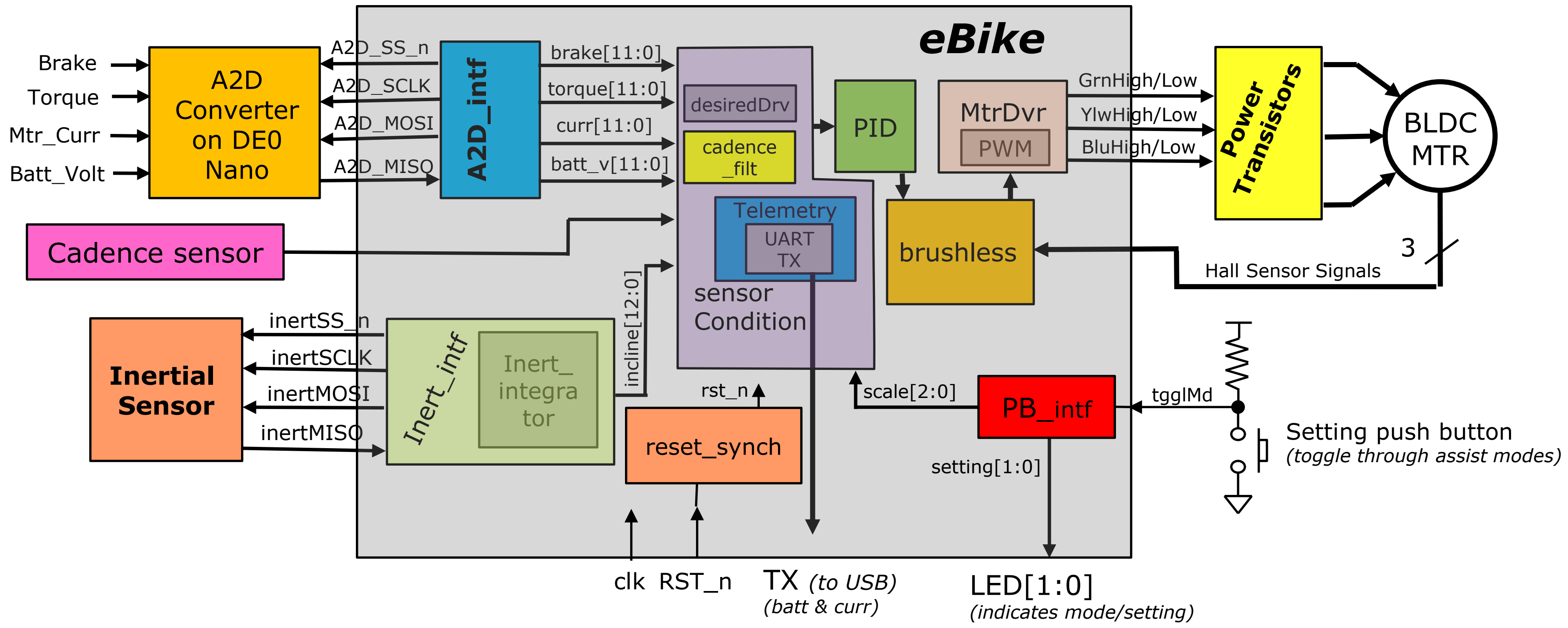
There is a level of hierarchy shown here called **Controller** that excludes **A2D_intf** and **Inert_intf**. This level of hierarchy is **not** required.

The interface of **eBike.sv** must match exactly to the specified interface. Please download the provided files from the project .zip and use the **eBike.sv** provided as your skeleton.

The hierarchy/partitioning of your design below the **eBike.sv** level is up to your team. The next slide shows my hierarchy. The hierarchy of your testbench is up to your team.

Your design will also be placed into out test bench, which is why it is critical it match at the **eBike.sv** level. Again…use the provided **eBike.sv** skeleton.

My Hierarchy (hierarchy below **eBike** up to you…you can match what I did if you like)



eBike Interface

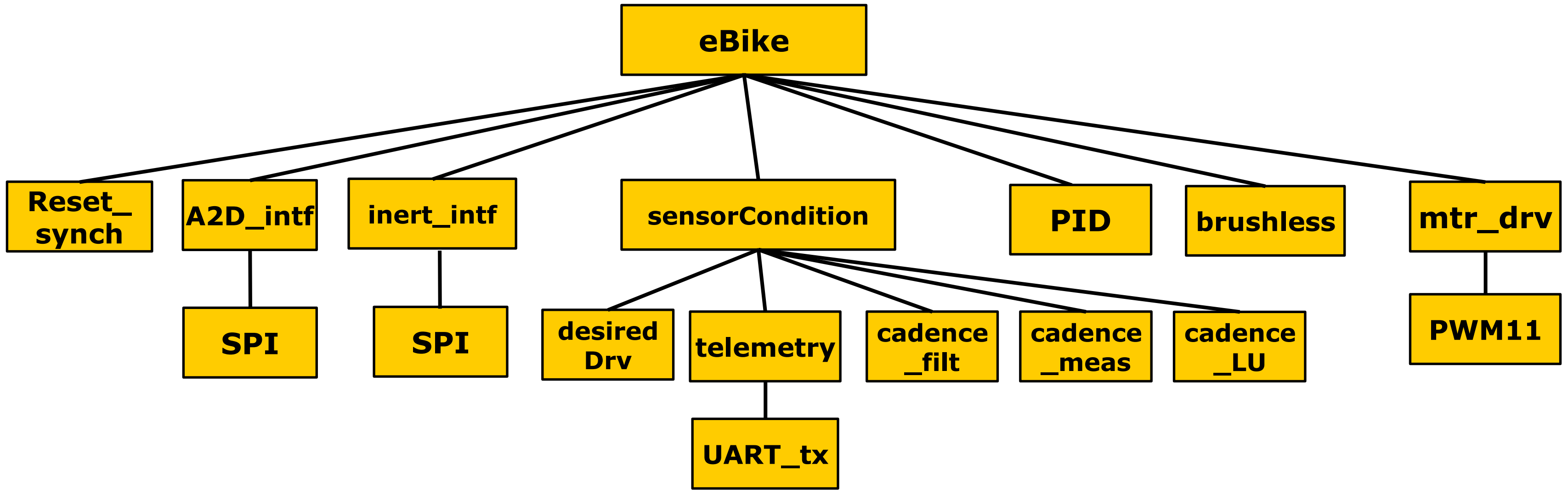
Signal Name:	Dir:	Description:
clk	in	Clock input (50MHz)
RST_n	in	Active low input from push button. Should be synchronized inside equalizer
tgglMd	in	From push button. Will cycle through the **setting**s.
setting[1:0]	out	Assist level setting. These bits drive LEDs on DE0-Nano so rider can see chosen setting
A2D_SS_n	out	Active low slave select to A2D SPI interface
A2D_SCLK	out	SPI bus clock (to A2D)
A2D_MOSI	out	Serial output data to SPI bus of A2D converter (**M**aster **O**ut **S**lave **I**n)
A2D_MISO	in	Serial input data from SPI bus of A2D converter (**M**aster **I**n **S**lave **O**ut)
hallGrn, hallYlw, hallBlu	in	Hall effect input from BLDC motor
highGrn, highYlw, highBlu, lowGrn, lowYlw, lowBlu	out	Gate controls for power MOSFETs driving motor coils. "high" signals drive the upper FET to source current into coil, "low" signals drive the lower FET to sink current from coil.
inertSS_n	out	Active low slave select to inertial sensor SPI interface
inertSCLK	out	SPI bus clock (to inertial sensor)
inertMOSI	out	Serial output data to inertial sensor
inertMISO	in	Serial input data from inertial sensor
cadence	in	Raw unfiltered cadence signal
TX	out	From telemetry module. Outputs info for optional display

Provided Modules & Files: (available on website under: Project)

File Name:	Description:
eBike.sv	**Requried** interface skeleton verilog file. **Copy this** and flesh it out with your design
UART_tx.sv	UART transmitter inside telemetry
UART_rcv.sv	UART receiver. Might be useful in testing
eBike_tb.sv	Up to you. This is a skeleton test bench you can start with.
eBikePhysics.sv	To be instantiated within your top level testbench. This combined model of the hub motor, and the inertial sensor can be very handy for fullchip simulations.
AnalogModel.sv	Can serve as a model of the A2D converter on the board. You provide 12-bit quantities for BATT, CURR, BRAKE, TORQUE and it provides a SPI slave to hook to your A2D_intf.
SPI_ADC128S.sv	SPI slave model for ADC on board. Child of AnalogModel.sv
eBike.qsf & eBike.qpf	.qsf and .qpf files are provided so you can map the design to the test platform and test there as well. **REMEMBER** to set your FAST_SIM parameter to false when mapping to the "real thing".

Synthesis:

- You have to be able to synthesize your design at the **eBike** level of hierarchy.



You are **NOT** allowed to use **compile_ultra**

- Your synthesis script should write out a gate level netlist of follower (eBike.vg).
- You should be able to demonstrate at least one of your tests running on this post synthesis netlist successfully.
- Timing (400MHz) is mildly challenging.

Synthesis Constraints:

Contraint:	Value:
Clock frequency	400MHz (yes, I know the project spec speaks of 50MHz, but that is for the FPGA mapped version. The standard cell mapped version needs to hit 400MHz minimum.
Input delay	0.3ns after clock rise for all inputs
Output delay	0.5ns prior to next clock rise for all outputs
Drive strength of inputs	Equivalent to a NAND2X2_LVT gate from our library
Output load	50fF on all outputs
Wireload model	16000 square micron size
Max transition time	0.20ns
Clock uncertainty	0.15ns

NOTE: Area should be taken after all hierarchy in the design has been smashed.
Area number to use is total area including interconnect estimate.

OurSynthesizedArea = 14740