ECE 551 Project Spec

Fall ’19 **eBike Controller**

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

- Project Grading Criteria:
  - Quantitative Synthesis Element 17.5%

    (yes this could result in extra credit)

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

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

Note: Minimum synthesized frequency to the Synopsys 32nm library is 300MHz. No benefit to faster than 300MHz.

  - Project Demo (82.5%)
    - ✓ Code Review (10%)
    - ✓ Testbench Method/Completeness (15%)
    - ✓ Synthesis Script review (7.5%)
    - ✓ Post-synthesis Test run results (10%)
    - ✓ APR of unit in IC_Compiler (7.5%)
    - ✓ Results when placed in EricWangnan Testbench (22.5%)
    - ✓ Test of code on eBike platform & eBike (10%)

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.

- Synthesize to a 300MHz clock (instead of 250MHz) and get 1.5% extra (have to use the area from 300MHz run though).

Project Due Date

- Project Demos will be held in B555:
  - Monday (12/9/19) from 1:00PM till evening.
    - ✓ 1.5% Extra Credit for demoing on Monday
  - Tuesday (12/10/19) from 1:00PM till evening.
    - ✓ 0.75% Extra Credit for demoing on Tuesday
  - Wednesday (12/11/19) from 1:00PM till evening.

- Project Demo Involves:
  - ✓ Code Review
  - ✓ Testbench Method/Completeness
  - ✓ Synthesis Script & APR review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in EricWangnan testbench
  - ✓ Demo of your code on and eBike.

Test Platform

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

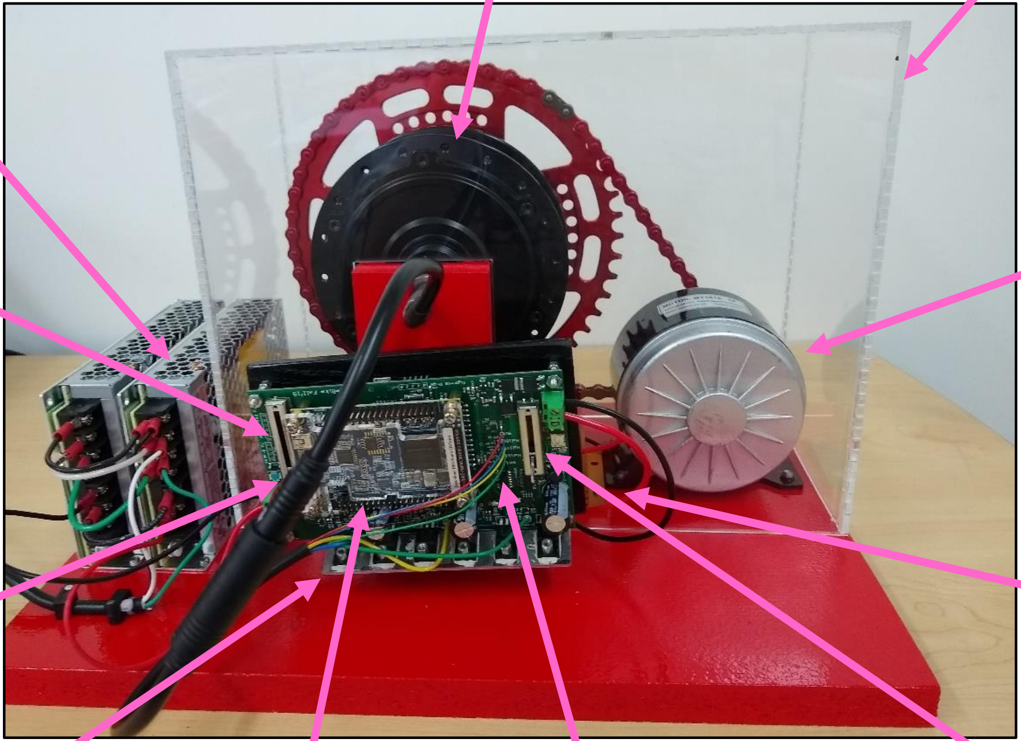
E-Bike hub motor (250W brushless DC)

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

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



Push button that mimics rider squeezing the brake handle

Load resistor to dissipate the power the generator generates.

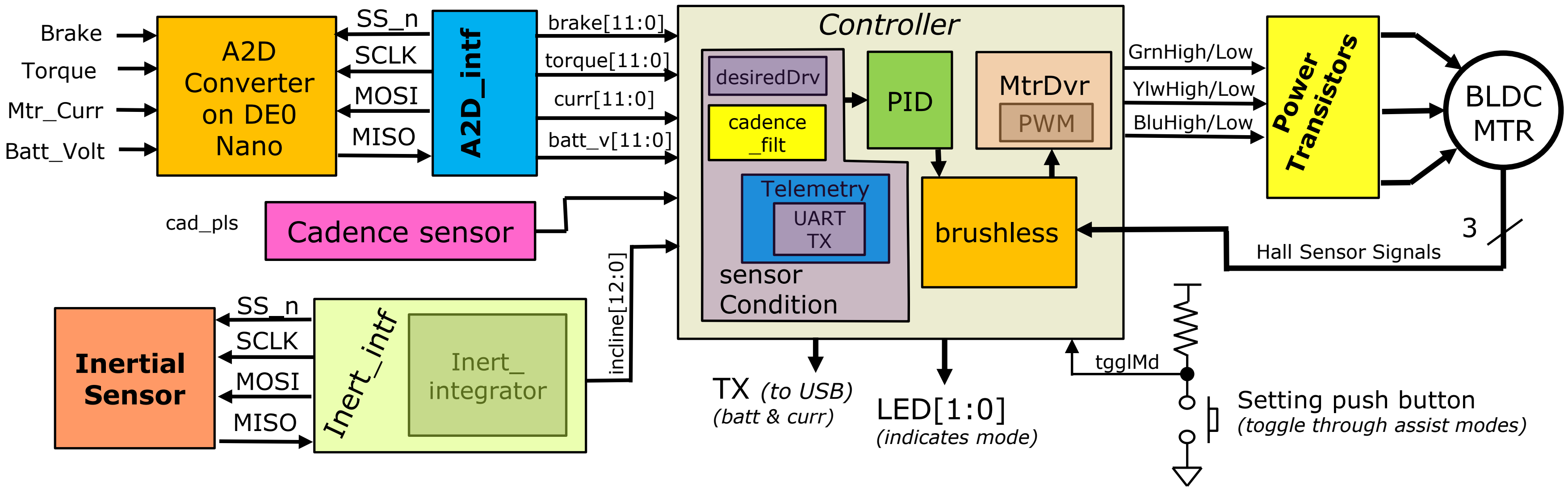
You can see the 6 Power FETs that drive the motor coils are mounted to an aluminum heat sink

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.

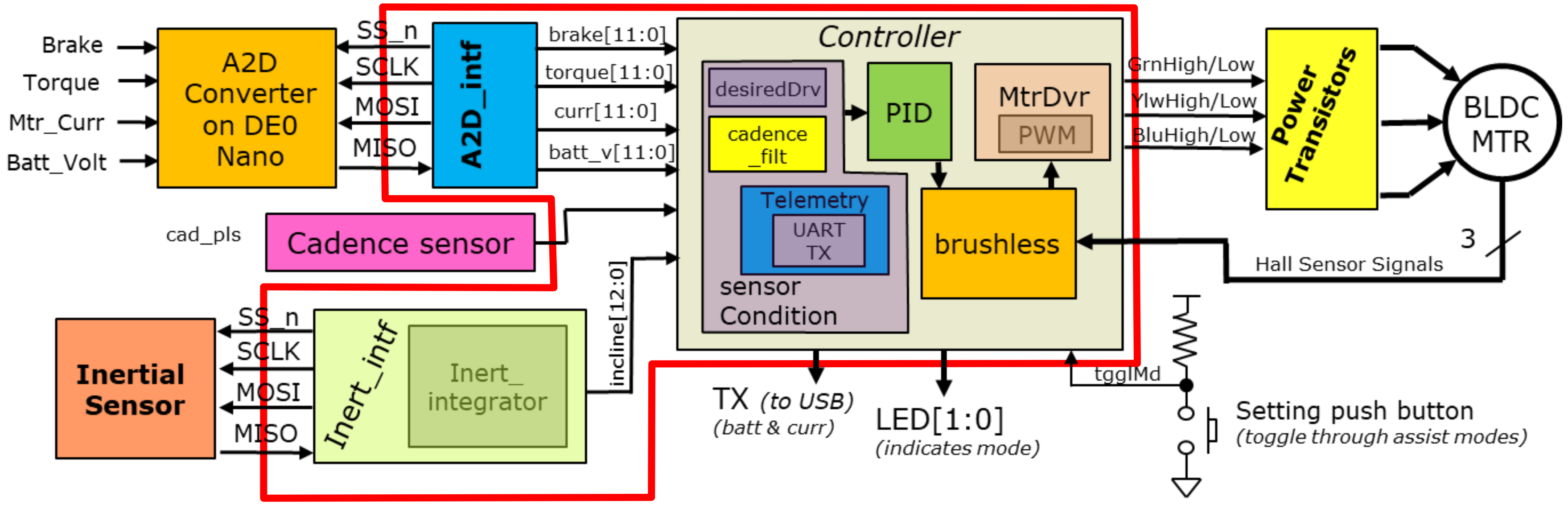
This slide potentiometer mimics the cadence sensor

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 and laid out as an APR block via IC_Compiler. For practical purposes we will also map that logic to FPGA so we can run the demos.

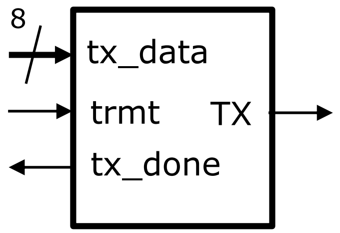
You Must have a block called **eBike.v** which is top level of what will be the synthesized DUT.

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

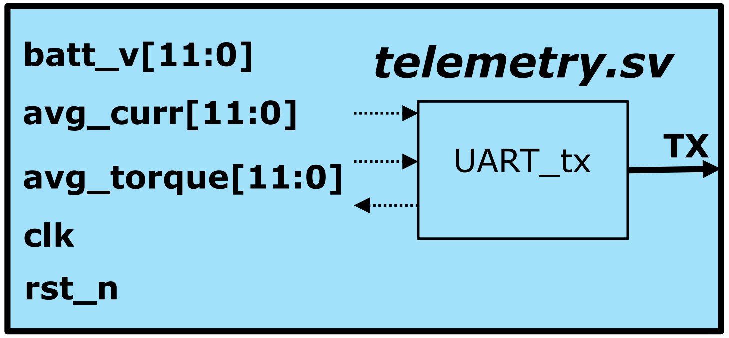


A wrapper around **UART_tx.sv** is needed that will periodically send out ***batt_v[11:0], avg_curr[11:0]*** and ***avg_torque[11:0]***. The packet (sent roughly 48 times a second) will be 8 bytes in length. The packet consists of a 2-byte delimiter of 0xAA 0x55 followed by 6-bytes of payload.

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

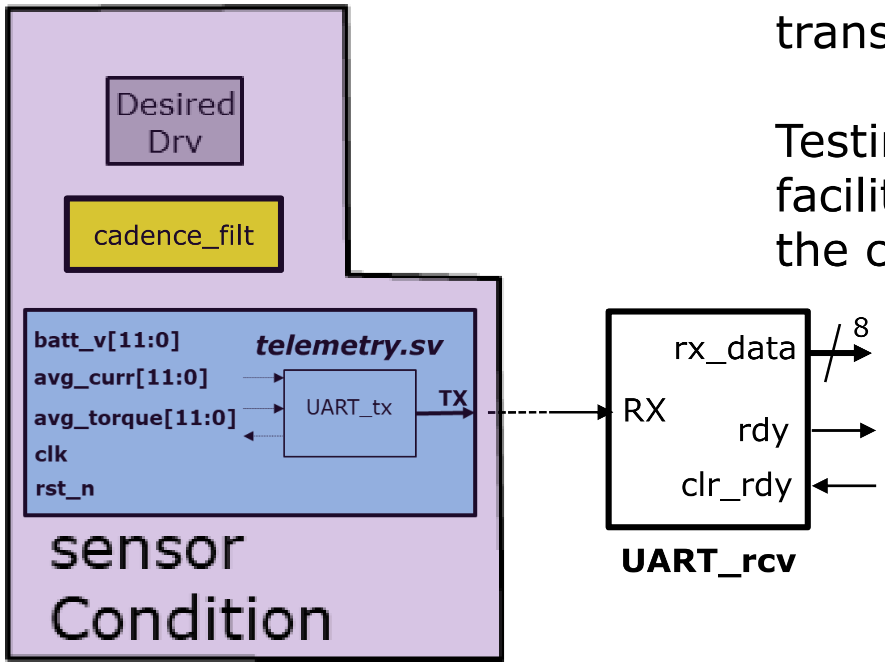


telemetry.sv should have the interface shown.

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_vec[4:0]	in	Unsigned number representing the speed of the rider's pedaling.
incline[12:0]	in	Signed number (you just delt with this in Part I)
setting[1:0]	in	unsigned. Represents level of assist motor provides 11 => a lot of assist, 10 => medium, 01 => little
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)

- The 13-bit signed **incline** signal should be saturated to a 10-bit signed value (**incline_sat**)

- **incline_factor** is simply **incline_sat** + 256 and is an 11-bit signed signal.

- Limit **incline_factor** and create a new 9-bit saturated signal (**incline_lim**) that is clipped with respect to negative values. If **incline_factor** is negative **incline_lim** 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 6-bit unsigned signal called **cadence_factor** should be created from **cadence_vec**. If **cadence_vec** is < 2 the rider is not pedaling and **cadence_factor** should be set to zero, otherwise **cadence_factor** should be **cadence_vec** + 32.

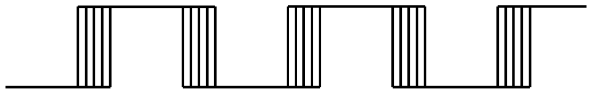
- A 13-bit signed signal **torque_off** is **avg_torque** minus TORQUE_MIN. TORQUE_MIN should be a localparam and set to 12'h380.

- **torque_pos** is a negative clipped version of **torque_off**. If **torque_off** is negative **torque_pos** will be set to zero.

- The assist level of the motor is a product of **torque_pos**, **incline_lim**, **cadence_factor**, and **setting**. It is a 29-bit unsigned number.

- **target_curr** should be assist_prod divided by 2^{14}. If this would result in a value greater than 0xFFF it should be saturated to 0xFFF.

cadence_filt

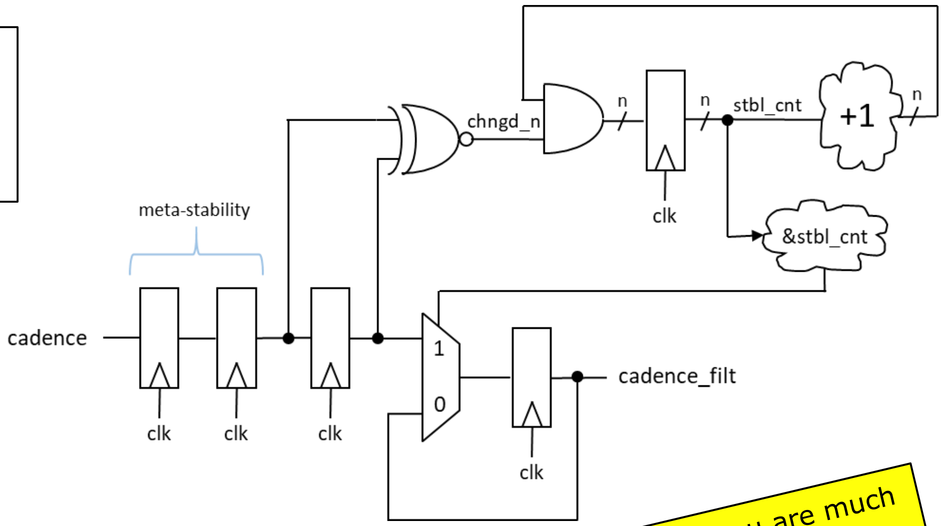


cadence signal

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:

a. Convert the filtered cadence signal (***cadence_filt***) into a 5-bit vector (***cadence_vec***) that represents pedaling speed. Also produces a signal (**not_pedaling**) that is used to inhibit drive when the rider's pedaling rate falls below a threshold.

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

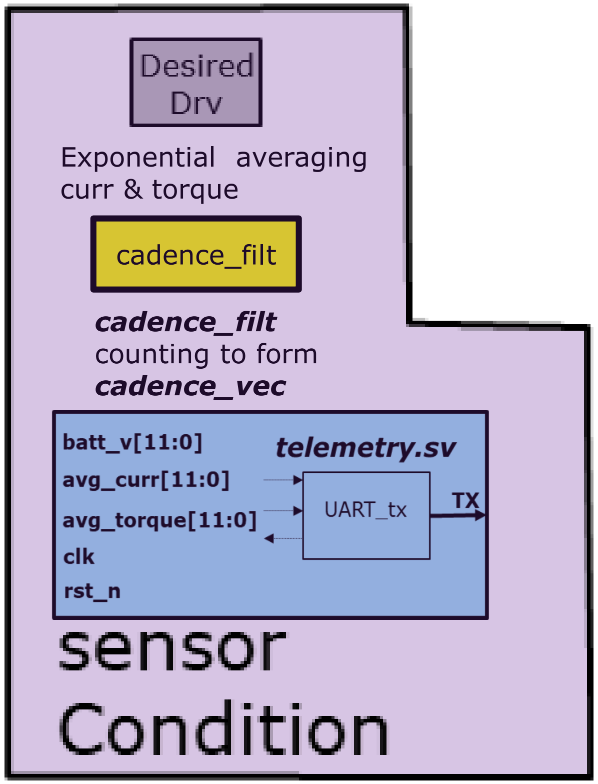
c. Perform an exponential average (of weight 32) on the raw 12-bit ***torque*** reading to produce ***avg_torque***. The accumulator of this exponential average should be seeded with 16X ***torque*** when pedaling resumes (***not_pedaling*** falls). This exponential average needs to be synchronized with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

d. 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.

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

sensorCondition (interface & cadence details)

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	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
setting[1:0]	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



The rising edges of ***cadence_filt*** should be counted over a 0.67sec interval to form a 5-bit signal called ***cadence_vec.*** This 5-bit vector form of cadence feeds into **desiredDrv**.

Why 0.67sec? It is a power of 2 of our clock, and it is long enough to give a decent average, and short enough to be responsive when the user stops pedaling.

not_pedaling is asserted if ***cadence_vec*** is less than 5'h02

Take care that ***cadence_vec*** saturates. Meaning if the rider was pedaling very fast we would not roll over 5'h1F and report a smaller number.

sensorCondition (what is exponential average)

Many sensor readings require averaging. Running averages are great (responsive yet reliable) 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 **$\log_2$(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.

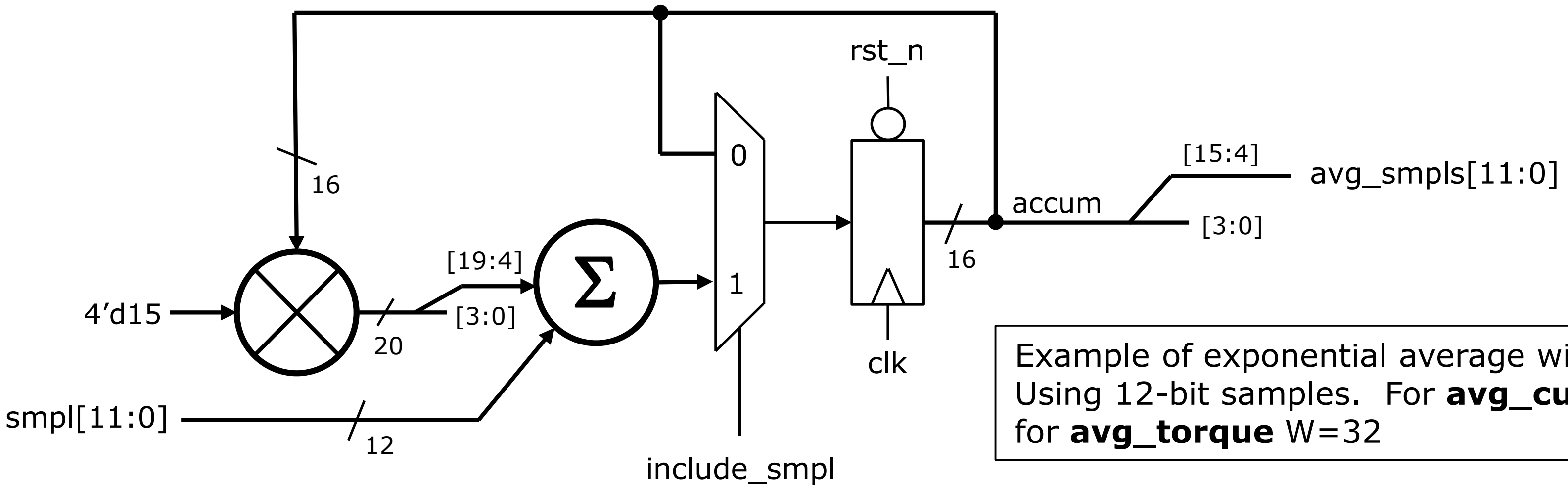
For every new sample (**smpl**) to be averaged the accumulator would be updated as:
   accum = ((**accum*(W-1))/W) + smpl**.

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**=4, 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

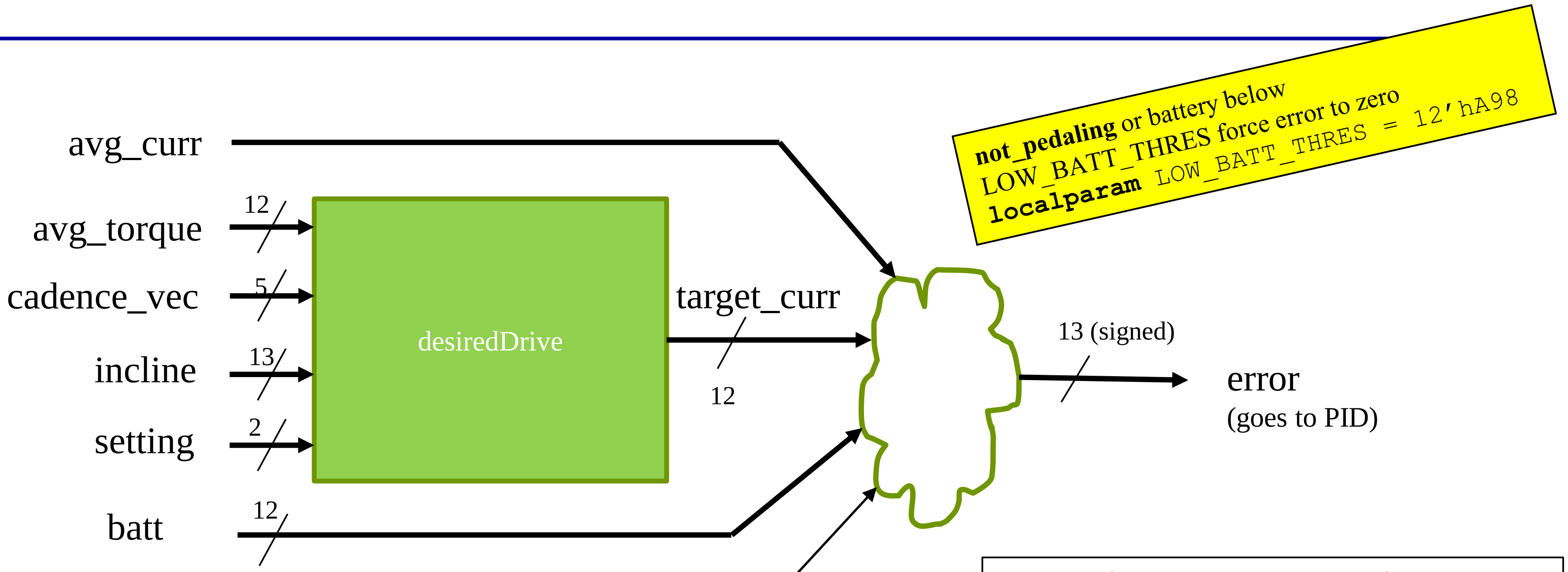
NOTE: new samples do not have to be included in the average every clock cycle (see ***include_smpl***)

For **avg_curr** a new sample will be included once every 2^{22} clocks (84ms).

For **avg_torque** a new sample will be included once every rise edge of **cadence_filt.**

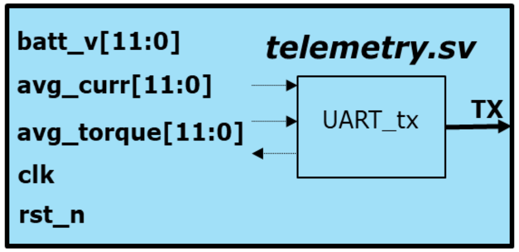
NOTE: the torque sensor has a large weight (W=32), so is slow to respond. When the rider starts pedaling again (fall of **not_pedaling**) we want to seed the accumulator proportional to the first torque reading (seed it with {1'b0,torque,4'b0000}). This makes the exponential average quicker to ramp up and provide to assist to the rider sooner. 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**.

sensorCondition (other stuff)



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.



sensorCondition (fast simulation)

Both **cadence_per** at 0.67sec and the update rate of the exponential average for **avg_curr** are problematic for simulation times. Similar to what we did in PID we want to introduce a parameter called **FAST_SIM**. **FAST_SIM** should be used to reduce the check of **cadence_per** to 16-bits wide, and the current averaging update to 16-bits wide.

Again a **generate if** can work nicely for this.

Go ahead now and modify your code to add a parameter and timing differences based on a **FAST_SIM** parameter.

NOTE: you will also have to add a **FAST_SIM** parameter to **cadence_filt** and pass **FAST_SIM** down to it. The "stable" counter inside **cadence_filt** was 16-bits wide. When FAST_SIM is asserted it should only be checked to a 9-bit width (otherwise it will filter the smaller pulse width we use for the **cadence** signal during fast simulations).

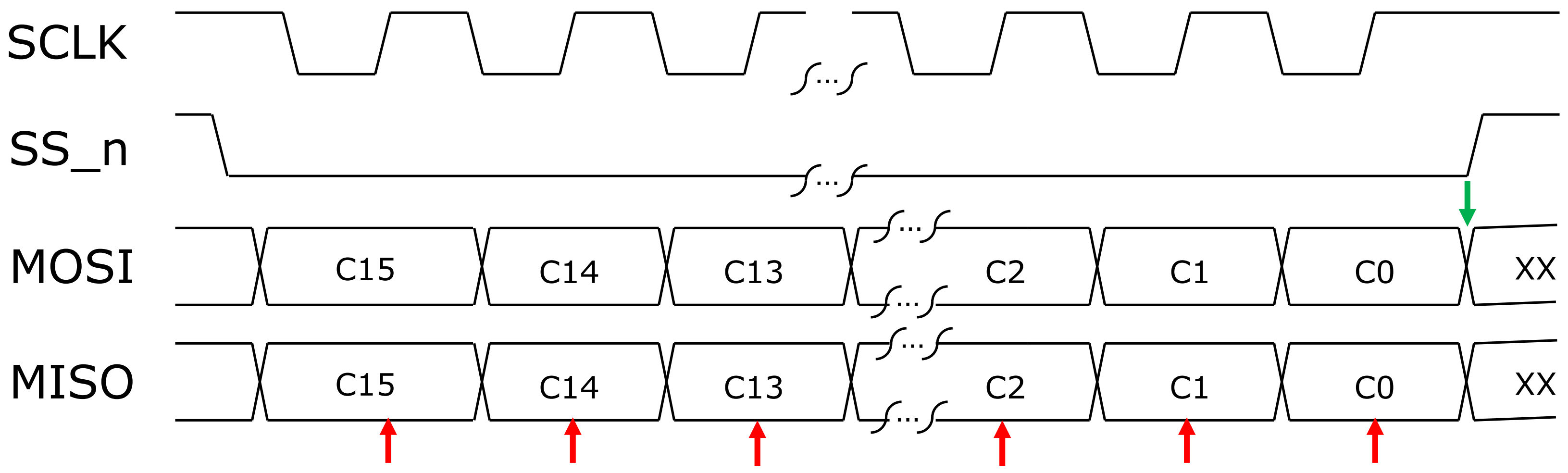
Now you are ready to simulate. This block is very central to the whole system and is difficult to test individually. It will be well wrung out by full chip testing, At this level, however, you will **at least** want to ensure your **cadence_vec** circuitry and your exponential average circuitry are working. See the next page for recommendations.

What is SPI?

- Simple bi-directional serial interface (Motorola long long ago)
  - **S**erial **P**eripheral **I**nterconnect (very popular physical interface)
  - 4-wires for full duplex
    - ✓ MOSI (Master Out Slave In) (we drive this to A2D to inform what channel to read)
    - ✓ MISO (Master In Slave Out) (A2D sends data back over this line)
    - ✓ SCLK (Serial Clock)
    - ✓ SS_n (Active low Slave Select) (Our system only has a single slave, but in a system with multiple slaves this acts as a one hot selector of the active slave)
  - There are many different variants
    - ✓ MOSI Sampled on clock low vs clock high
    - ✓ 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 use the variant used by the A2D on the DE0_nano board.
    - ✓ MOSI changes after SCLK rise, and MISO is sampled at that time as well.

      (your SPI master should shift its shift register 2 system clocks after SCLK rise)
    - ✓ SCLK normally high
    - ✓ 16-bit packets

SPI Packets



Shown above is a 16-bit SPI packet. The master is changing (shifting) **MOSI** on the falling edge of **SCLK**. The slave device (6-axis inertial sensor or A2D converter) changes **MISO** on the falling edge too. We sample **MISO** on the rising edge (see red arrows).

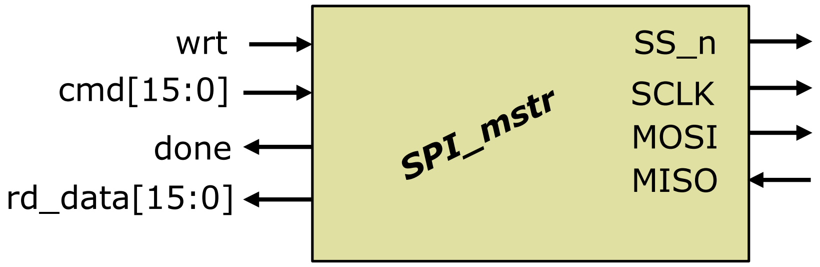
The sampled version of **MISO** in turn gets shifted into our 16-bit shift register. (on **SCLK** fall)

When **SS_n** first goes low there is a bit of a period before **SCLK** goes low. Our 16-bit shift register does not shift on the first fall of **SCLK**. This is called the "front porch".

At the end of the transaction C0 from the slave (on **MISO**) is sampled on the last rise of **SCLK**. Then there is a bit of a "back porch" before **SS_n** returns high. When **SS_n** returns high we shift our 16-bit shift register one last time (see green arrow) so "C0" captured on **SCLK** rise (last red arrow) is shifted into our shift register and we have received 16-bits from the slave.

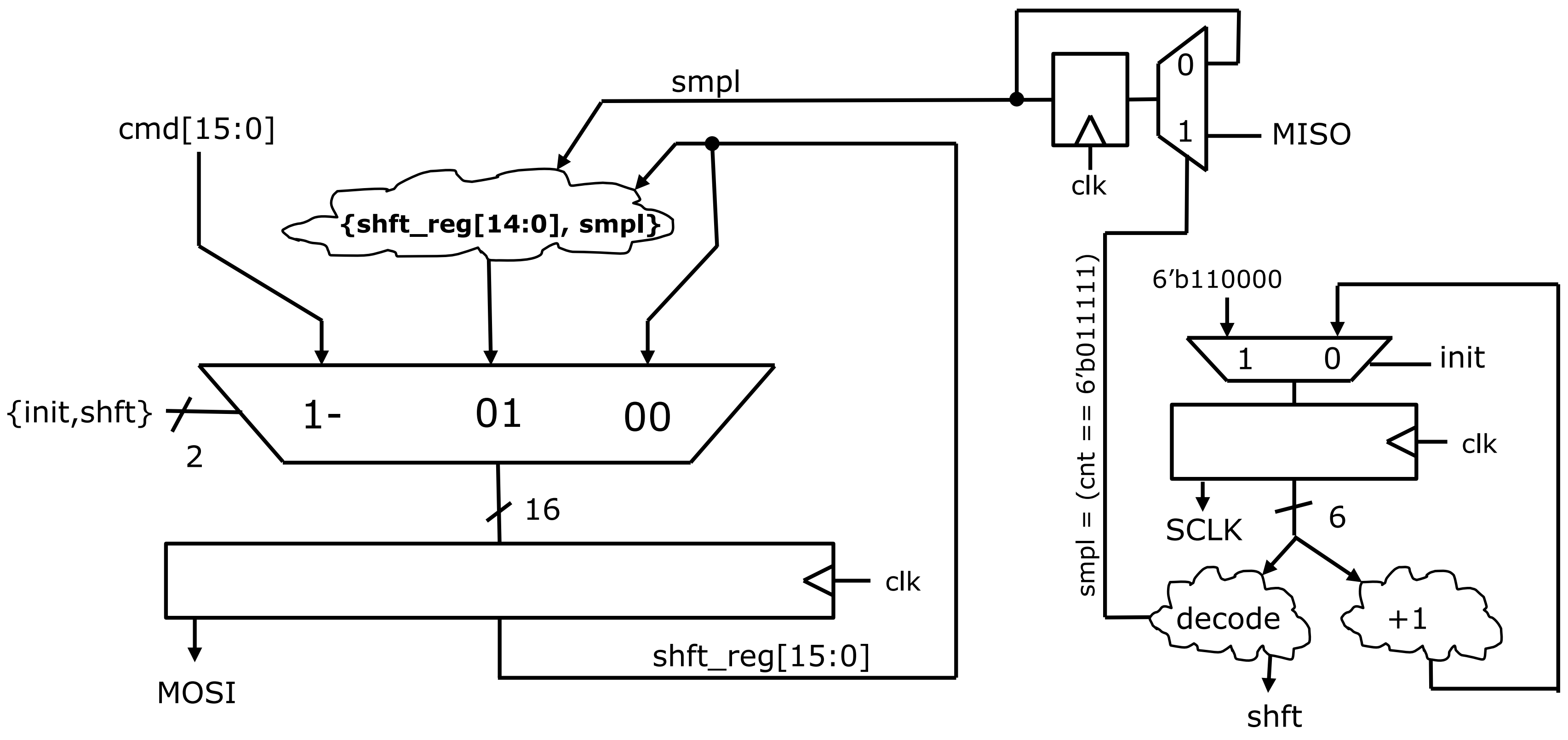
SPI Unit for Inertial Interface & A2D

- Both the 6-axis inertial sensor, and the A2D on the DE-0 Nano board can be read with a SPI master that implements the 16-bit SPI transaction mentioned above.
- You will implement **SPI_mstr.sv** with the interface shown.
- SCLK frequency will be 1/64 of the 50MHz clock (i.e. it comes from the MSB of a 6-bit counter running off clk)
- Although the description says things like: "the shift register is shifted on **SCLK** fall" and "**MISO** is sampled on **SCLK** rise". I had better not see any ***always*** blocks triggered directly on **SCLK**. We only use **clk** when inferring flops.
- Remember you are producing **SCLK** from the MSB of a 6-bit counter. So for example, when that 6-bit counter equals 6'b011111 you know **SCLK** rise happens on the next clk, so you can enable a sample of **MISO** then. Similar logic is used for when to shift the main 16-bit shift register.



Signal:	Dir:	Description:
clk, rst_n	in	50MHz system clock and reset
SS_n, SCLK, MOSI	in	SPI protocol signals outlined above
wrt	in	A high for 1 clock period would initiate a SPI transaction
cmd[15:0]	in	Data (command) being sent to inertial sensor or A2D converter.
done	out	Asserted when SPI transaction is complete. Should stay asserted till next **wrt**
rd_data[15:0]	out	Data from SPI slave. For inertial sensor we will only ever use [7:0] for A2D converter we will use bits [11:0]

SPI Implementation

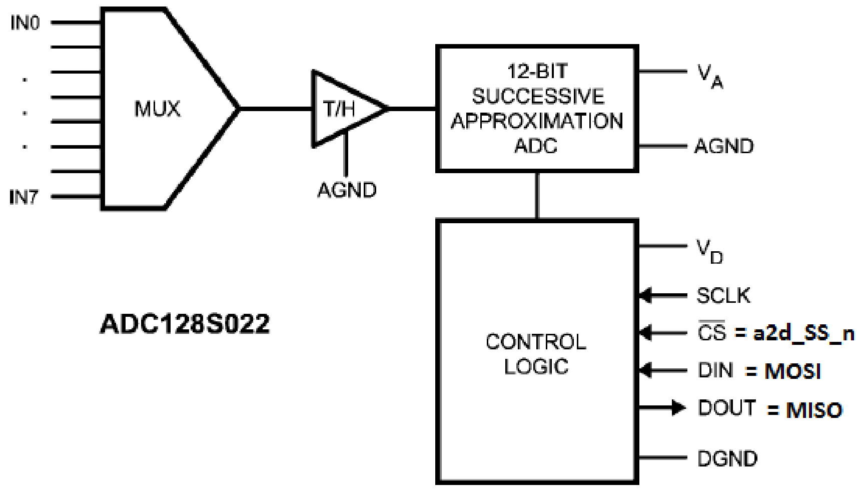


Heart of a SPI unit is a shift register and a counter. The MSB of the counter forms SCLK. The MSB of the shift register forms MOSI. A sampled version of MISO feeds into the LSB position of the shift register. Sampling and shifting are based on the value of the count. EVERYTHING works off clock, not SCLK. Of course a small state machine is needed to coordinate control of this SPI datapath to make it chooch.

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



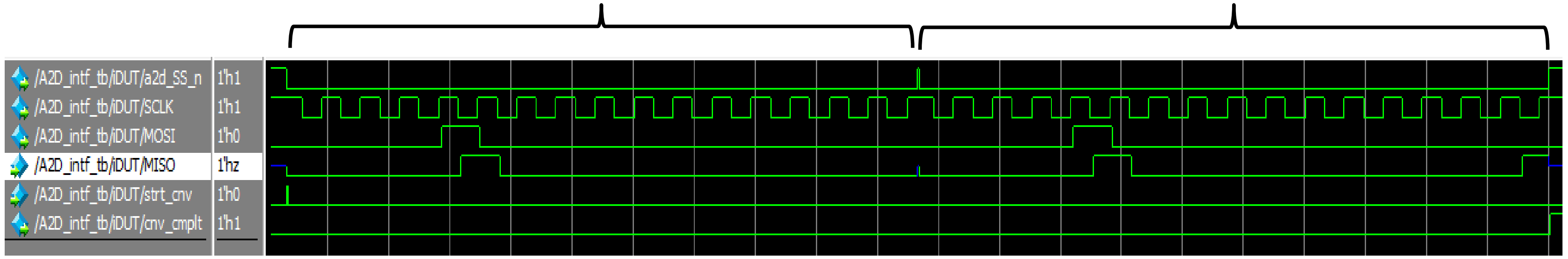
To read the A2D converter one sends the 16-bit packet {2'b00,addr,11'b000} twice via the SPI. There needs to be a 1 system clock cycle pause between the first SPI transaction completing, and the second one starting

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

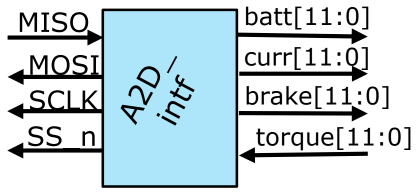
Our use of the A2D converter will involve two 16-bit SPI transactions nearly back to back (separated by 1 system clock cycle).

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.

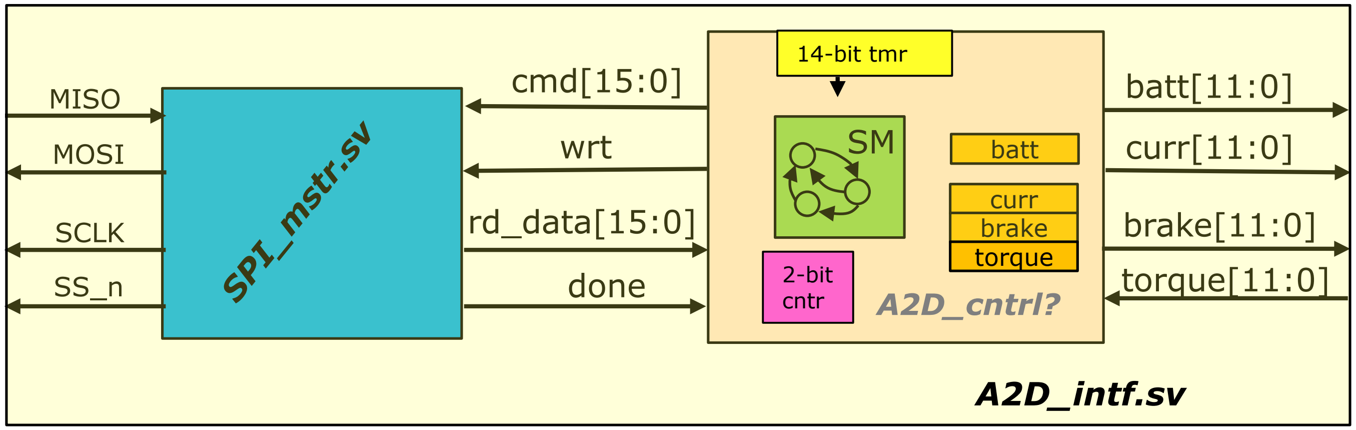
For the next 16-bit transaction the data sent over MOSI to the A2D does not matter that much. We are really just trying to get the data back from the A2D over the MISO line.

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

A2D Interface



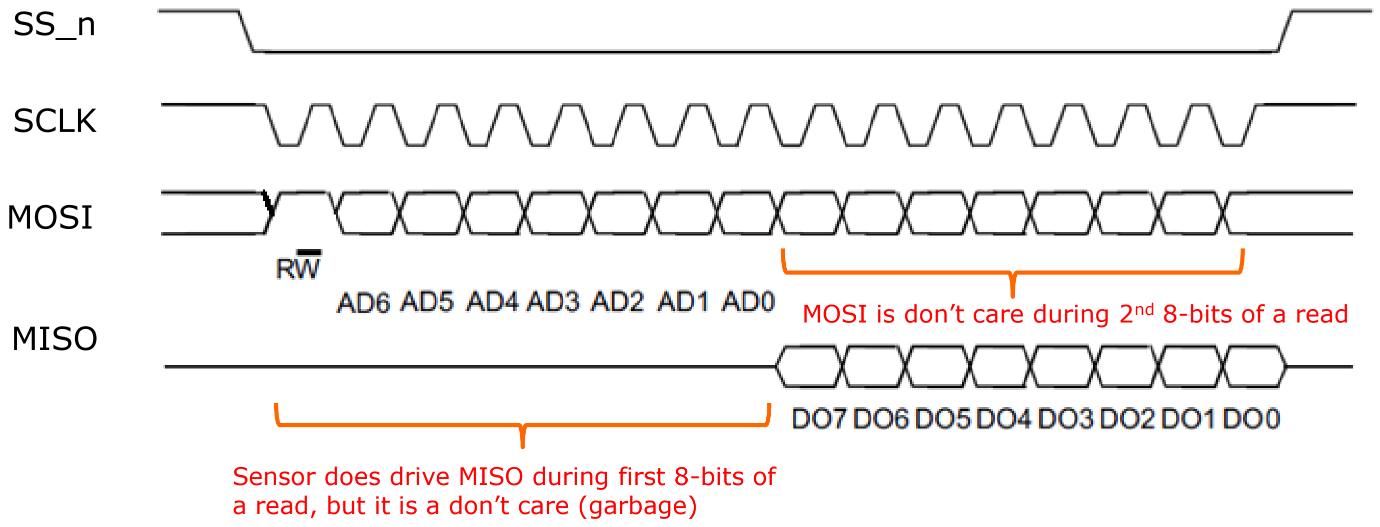
- You will build a block called **A2D_intf.sv**
- It will perform round robin conversions on channels (0,1,3,4) (battery voltage, motor current, brake lever, pedal torque)
- It maintains an internal 2-bit counter to know which measurand is next to be converted.
- A new channel is to be converted every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



- You will make use of **SPI_mstr**, 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.

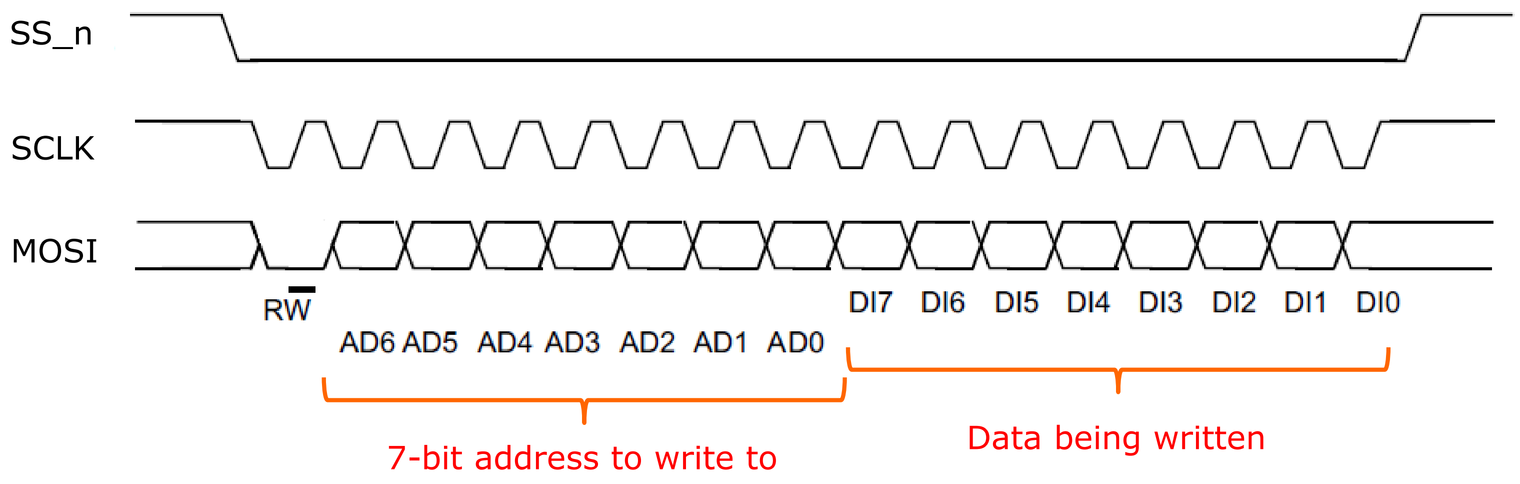
Inertial Sensor Interface

- 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/$\overline{\text{W}}$ 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
R$\overline{\text{W}}$
AD6 AD5 AD4 AD3 AD2 AD1 AD0
MOSI is don't care during 2nd 8-bits of a read
MISO
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)

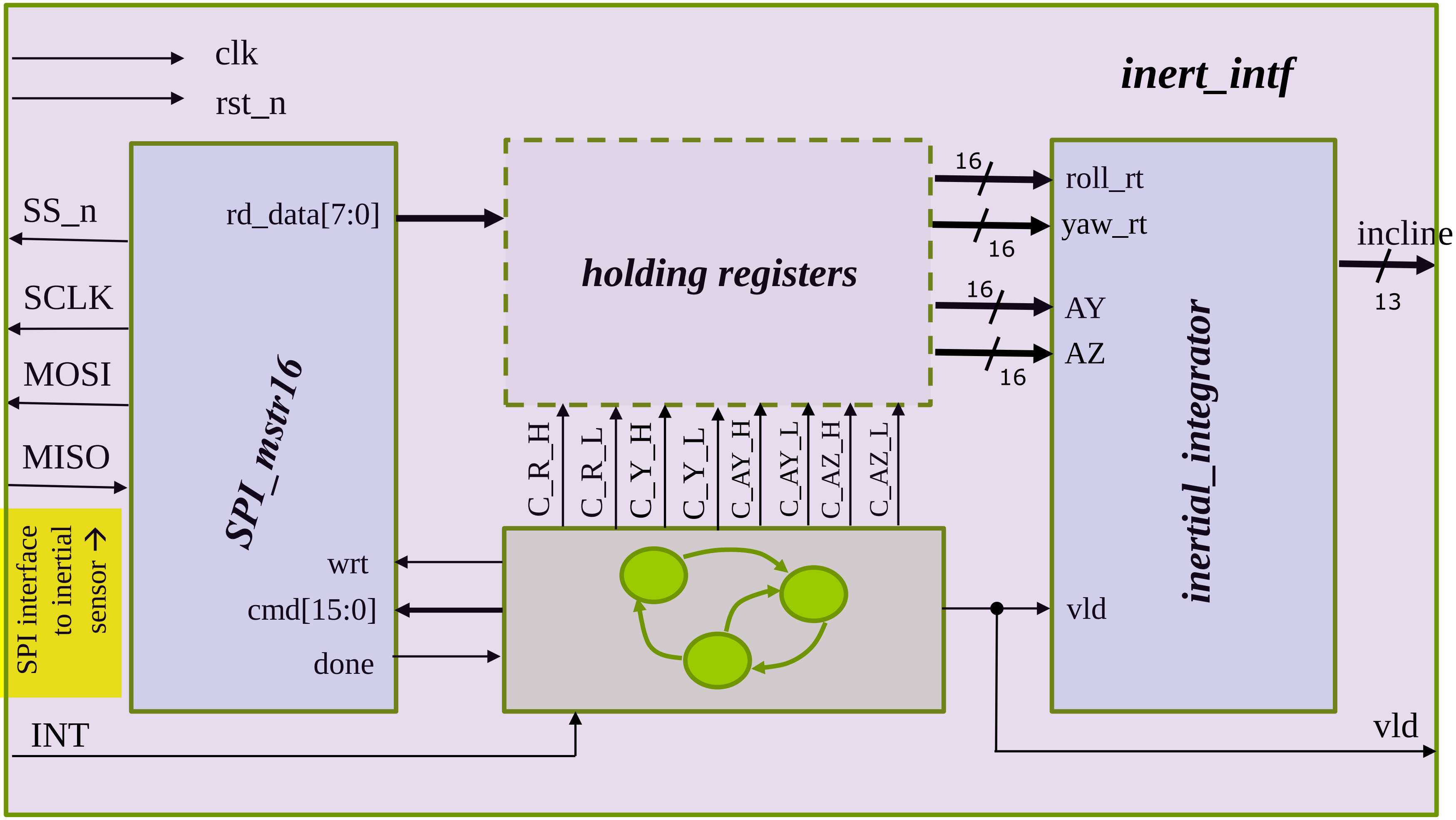
Inertial Sensor Interface (write)

- 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

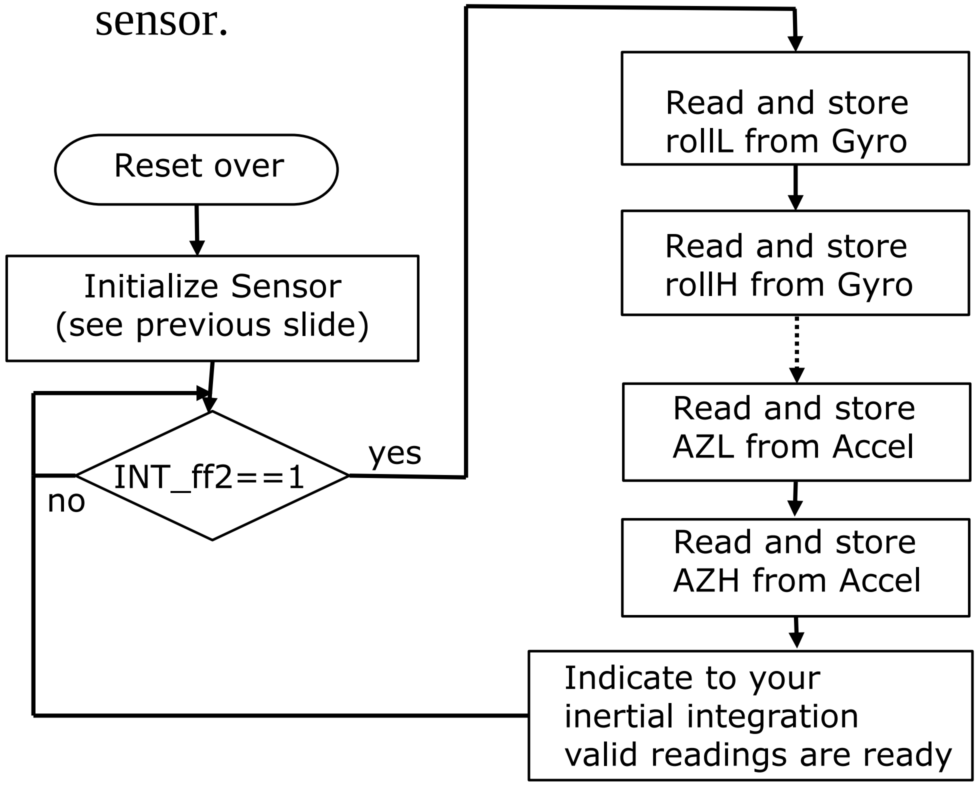
- After reset de-asserts the system must write to some registers to configure the inertial sensor to operate in the mode we wish. The table below specifies the writes to perform.

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

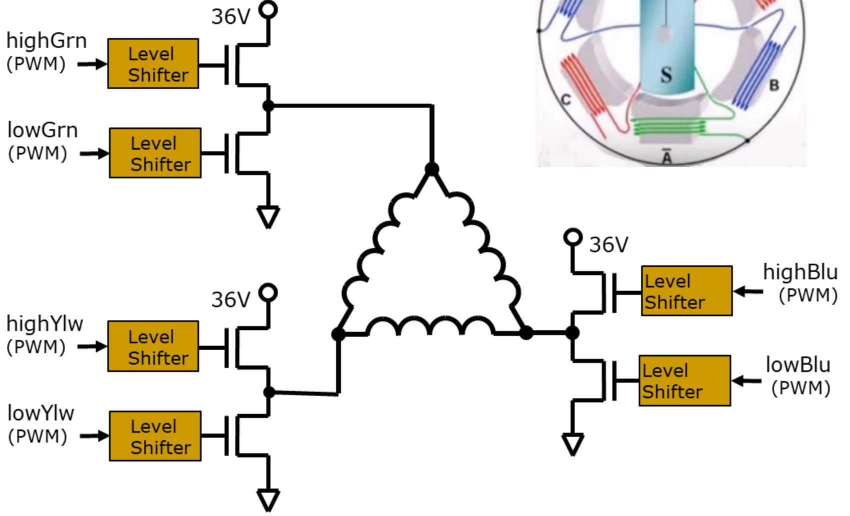
Inertial Integration (sensor fusion)

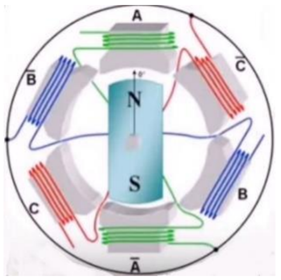
- The block **inertial_integrator.sv** takes the raw sensor readings and performs the sensor fusion. It computes both incline and roll internally, but only **incline** an output.
- If the magnitude of roll exceeds a threshold then the **incline** output is zeroed. When the rider is taking a hard corner (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.



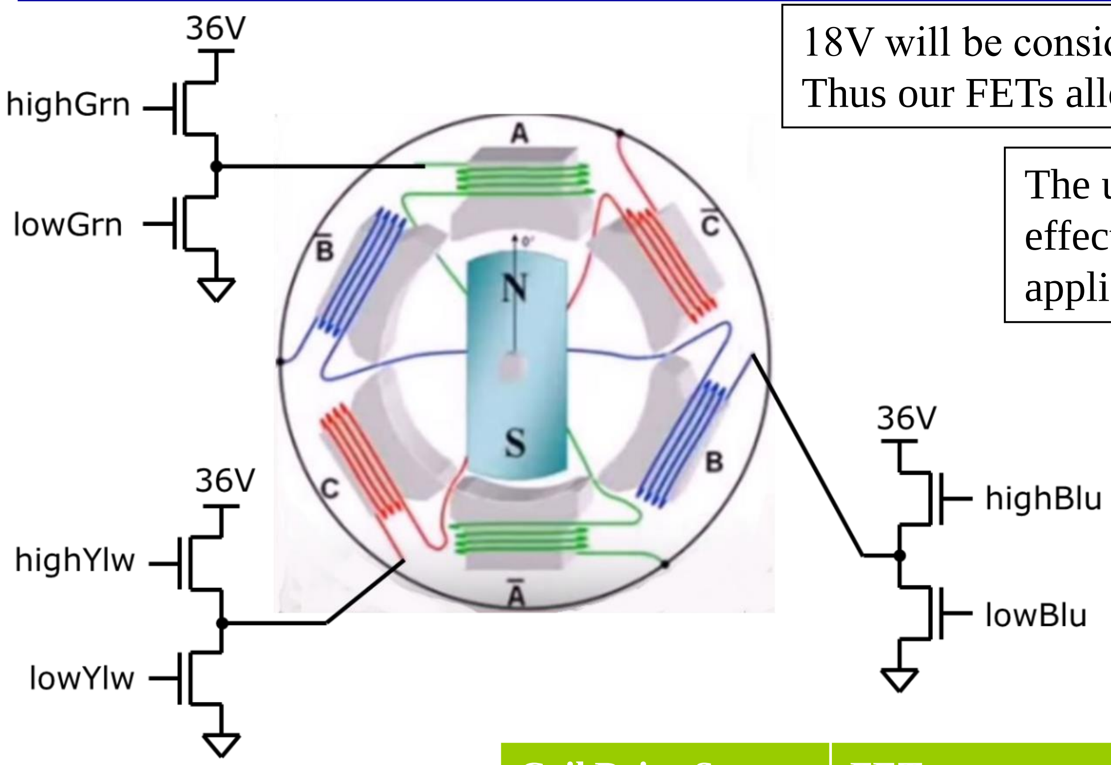


- **brushless.sv** will be the block that inspects the hall sensor signals and determines how to drive each coil. Each coil can be driven 1 of 4 ways: not driven (both high/low FETs off); driven "forward"; driven "reverse"; driven for dynamic braking (high FET off, low FET PWMed)

- 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 (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: **rotation_state = {hallGrn,hallYlw,hallBlu};**
- The following table outlines how we drive our coils

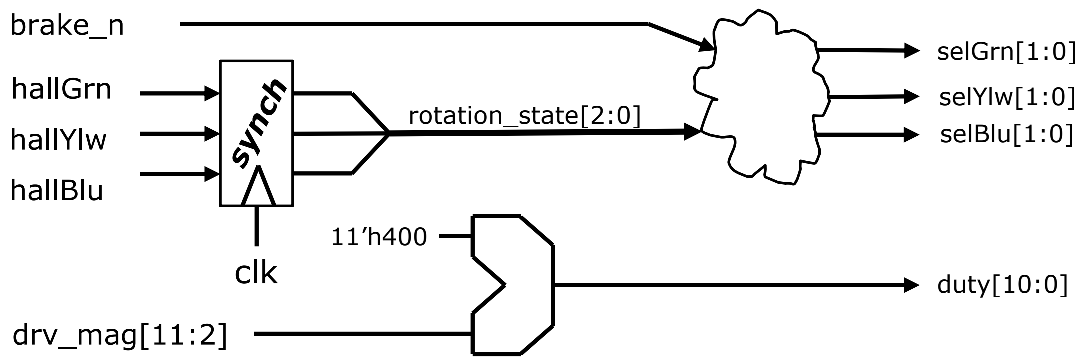
rotation_state	3’b101	3’b100	3’b110	3’b010	3’b011	3’b001
coilGrn	for_curr	for_curr	High Z	rev_curr	rev_curr	High Z
coilYlw	rev_curr	High Z	for_curr	for_curr	High Z	rev_cur
coilBlu	High Z	rev_curr	rev_curr	High Z	for_curr	for_curr

- 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.

- Since 18V is our virtual ground and given by a PWM duty cycle of 0x400 the out going **duty[10:0]** should be 11’h400 + drv_mag[11:2].

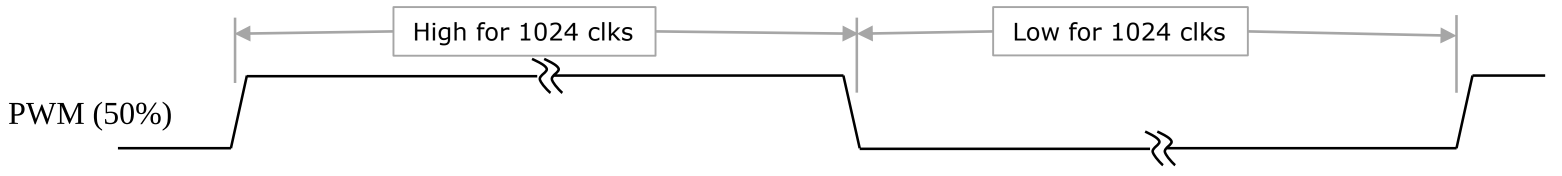
Brushless (how to drive the coils)

Signal:	Dir:	Description:
clk	in	50MHz clock (reset not needed)
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
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

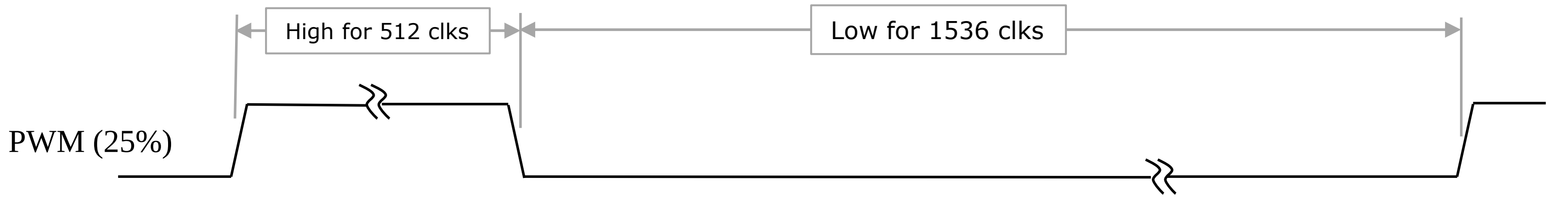


PWM (vary strength of motor drive)

- 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.



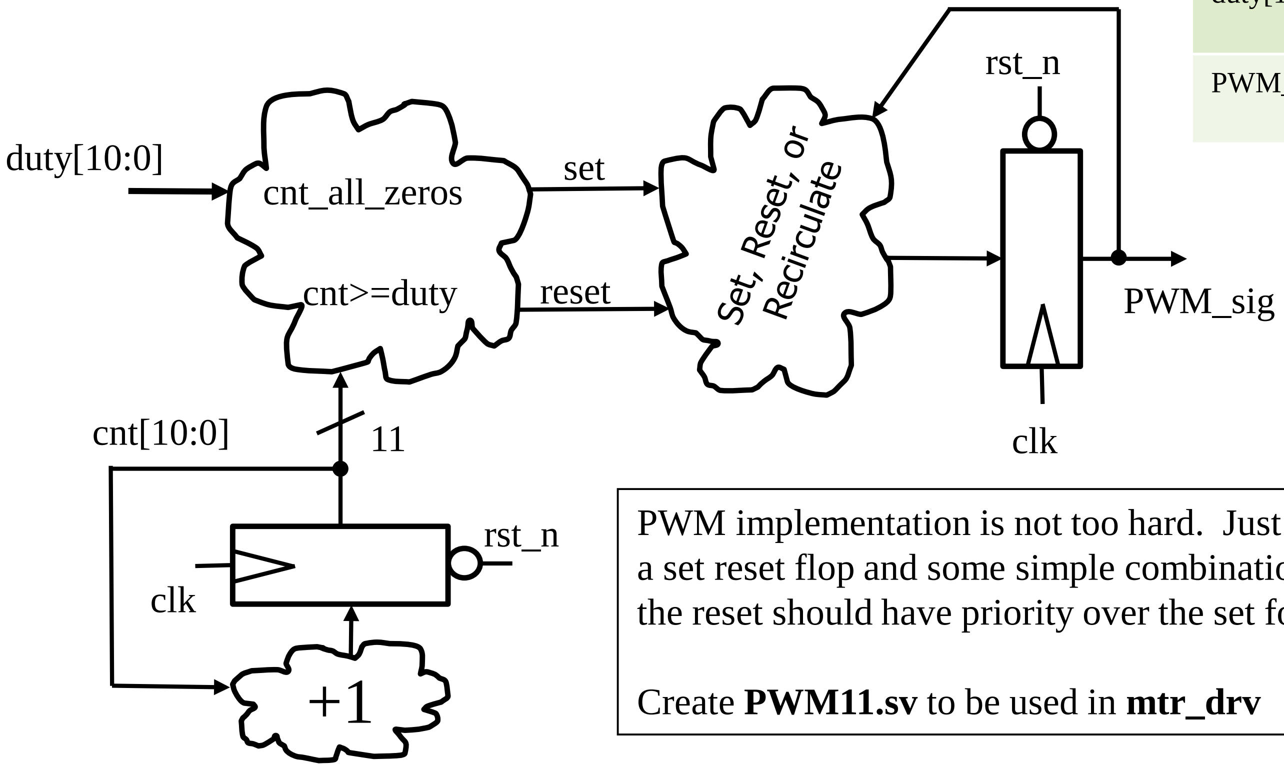
- Second example is 25% duty cycle



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.

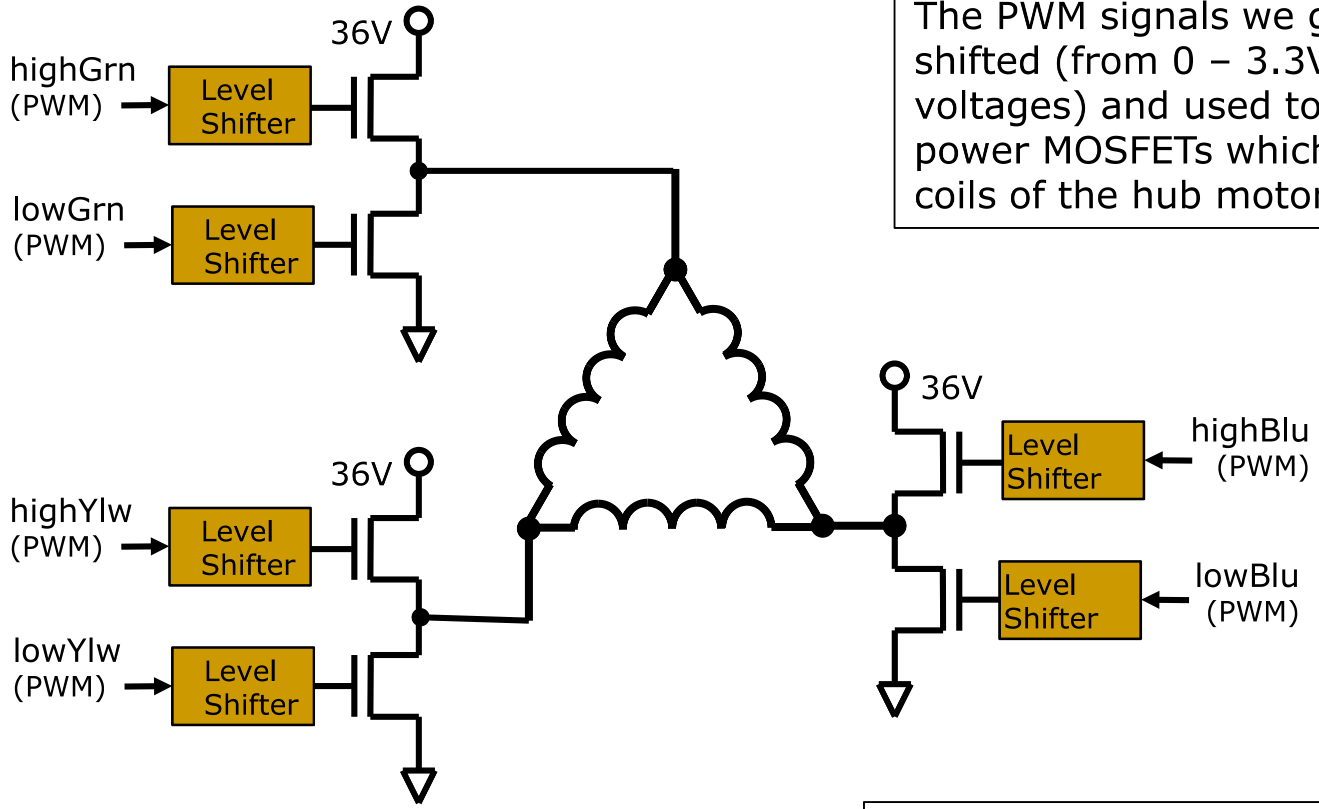
Signal:	Dir:	Description:
clk	in	50MHz system clk
rst_n	in	Asynch active low
duty[10:0]	in	Specifies duty cycle (unsigned 11-bit)
PWM_sig	out	PWM signal out (glitch free)



PWM implementation is not too hard. Just an 11-bit counter with a set reset flop and some simple combinational logic. Note that the reset should have priority over the set for the flop.

Create **PWM11.sv** to be used in **mtr_drv**

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

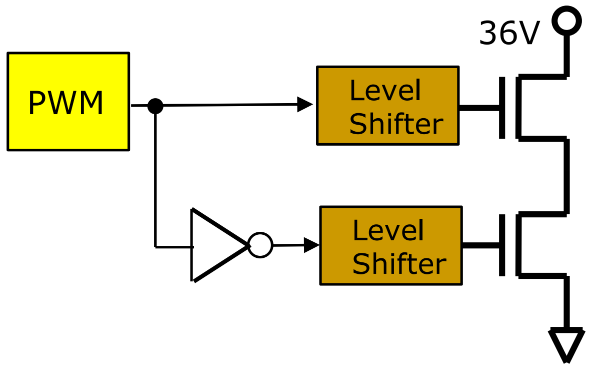


The PWM signals we generate are level shifted (from 0 – 3.3V signals to higher voltages) and used to drive the gates of power MOSFETs which in turn drive the coils of the hub motor.

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.

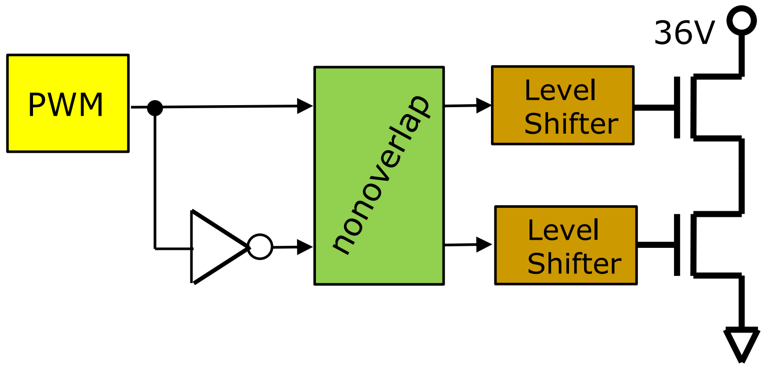
The power MOSFETs are very powerful (very low ohmic and capable of passing a lot of current…like 100A). Given the proper conditions they can self-destruct. Actually that would be the "improper conditions".

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:

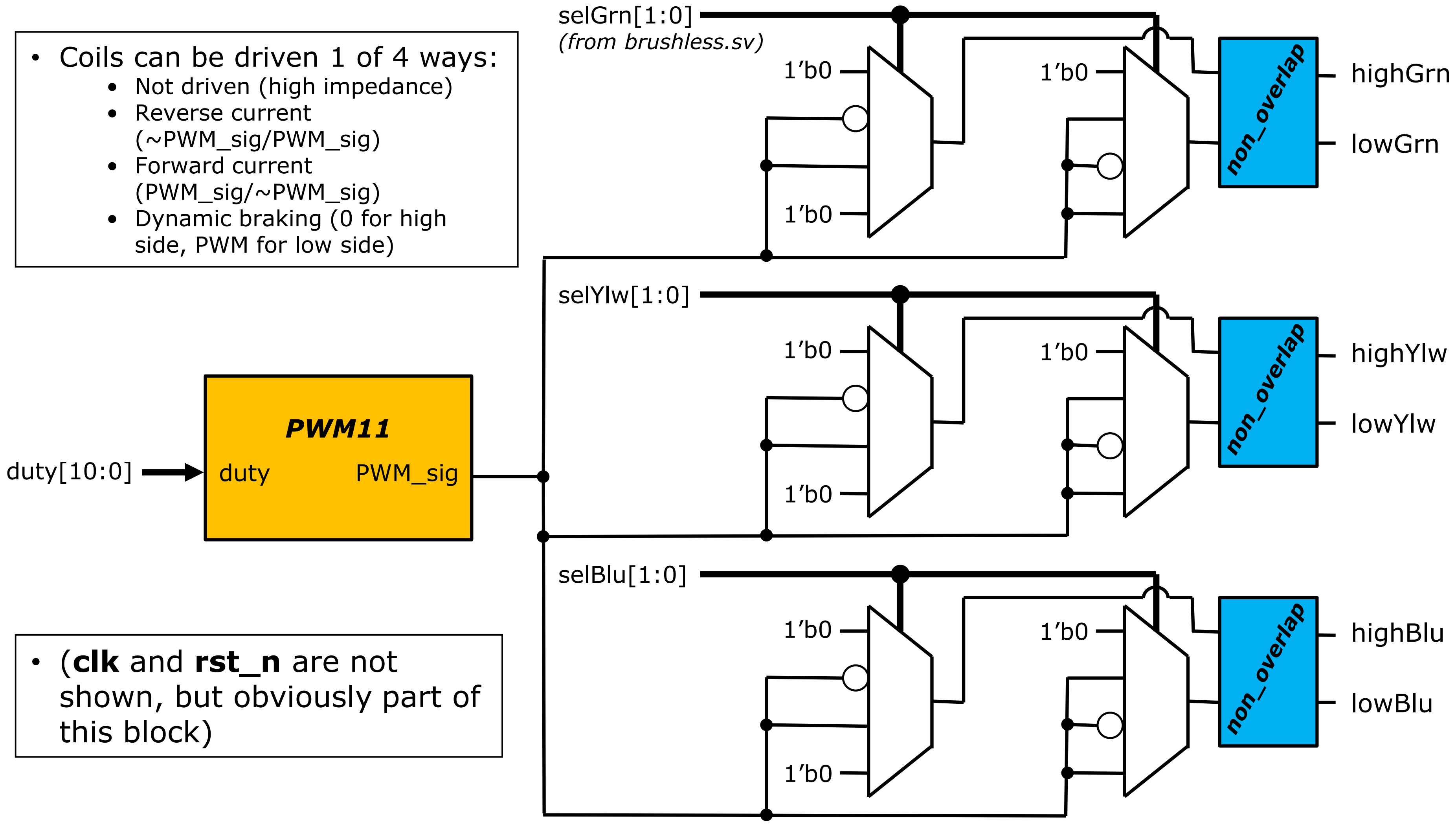
- Whenever ***highIn*** or ***lowIn*** change both ***highOut*** and ***lowOut*** should go low on the next clock cycle.
- Once ***highOut*** and ***lowOut*** are forced low (from a change in either) they should remain forced low for **32** system clocks.
- Both ***highOut*** and ***lowOut*** should 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.**

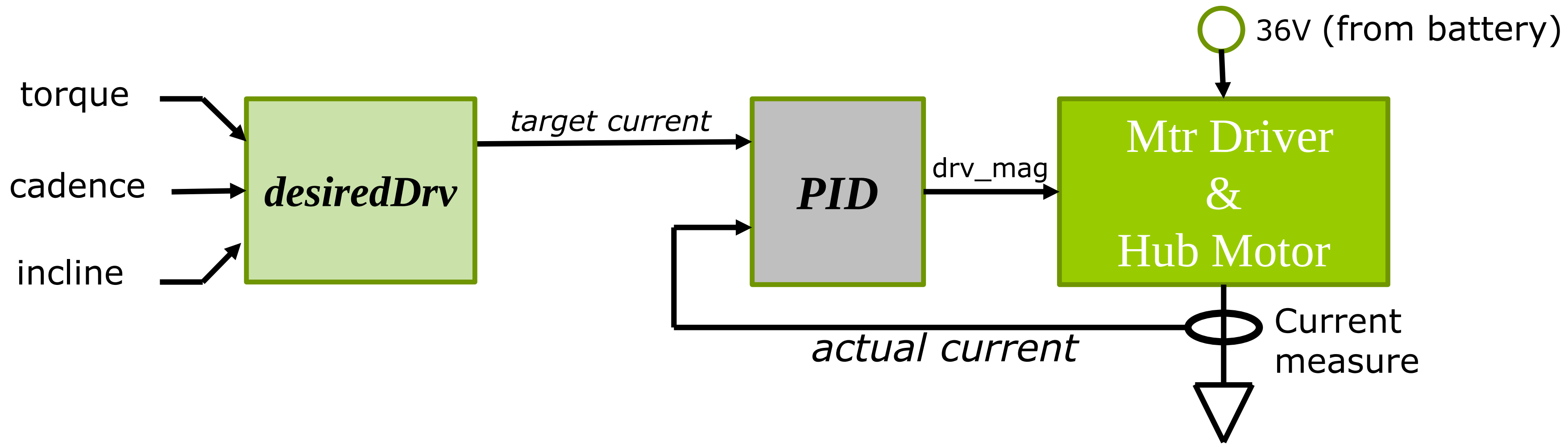
mtr_drv

- 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)



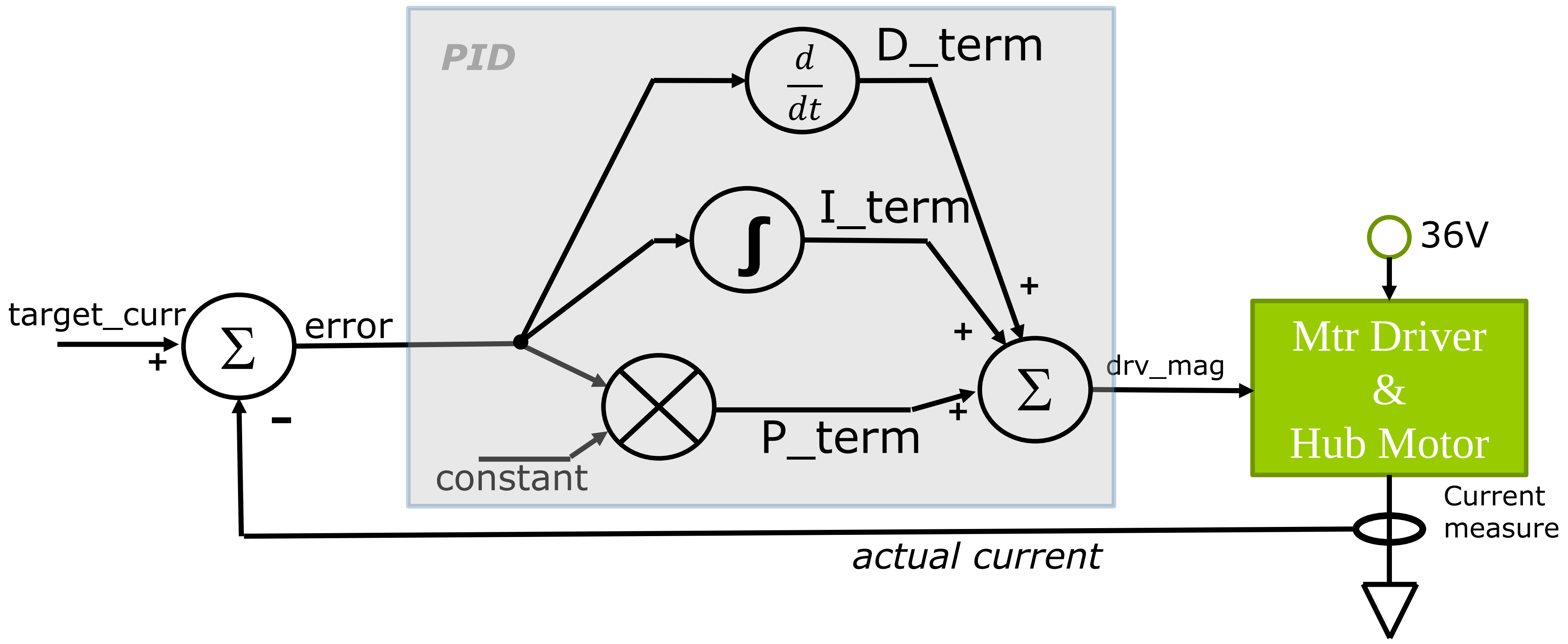
PID Controller



- 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.

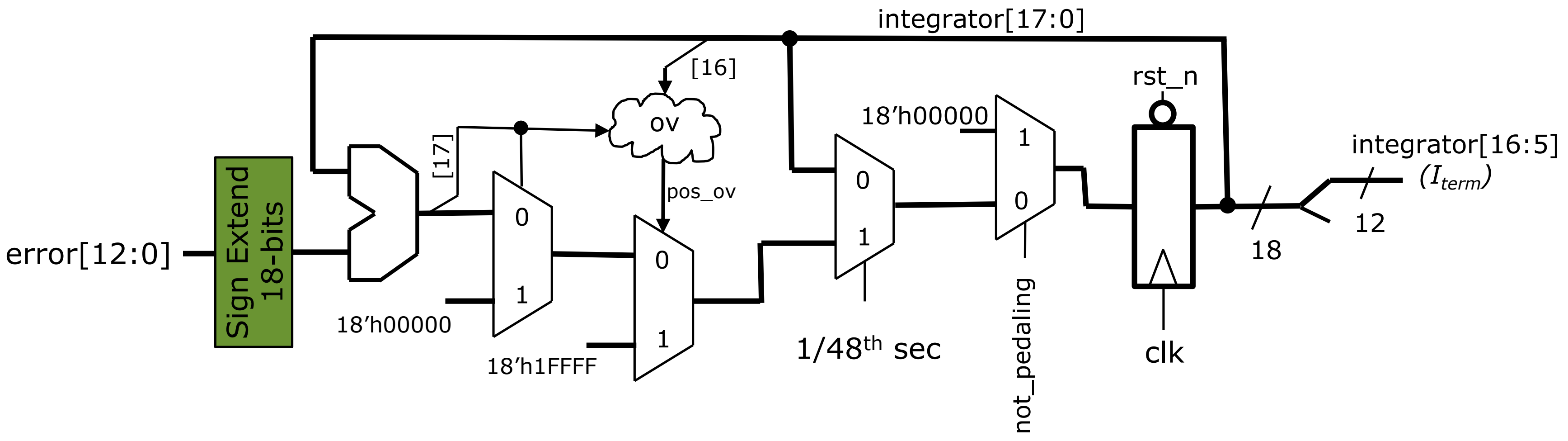
PID Controller



- 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**.

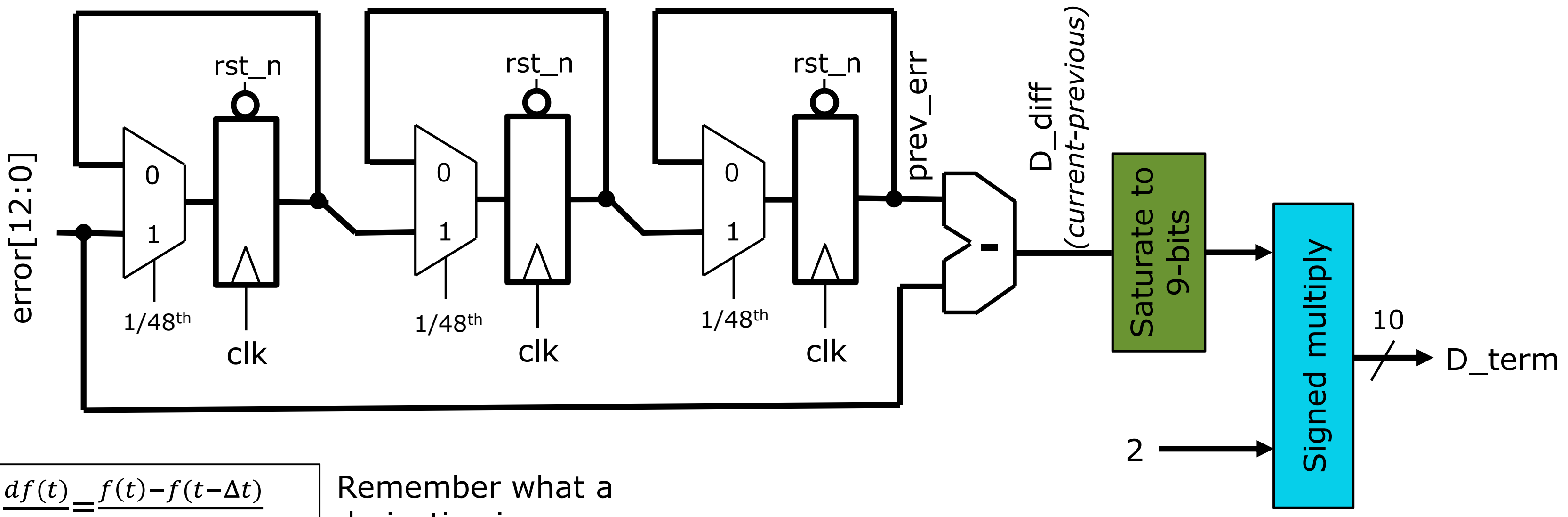
PID Controller (P_term & I_term)

- 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)

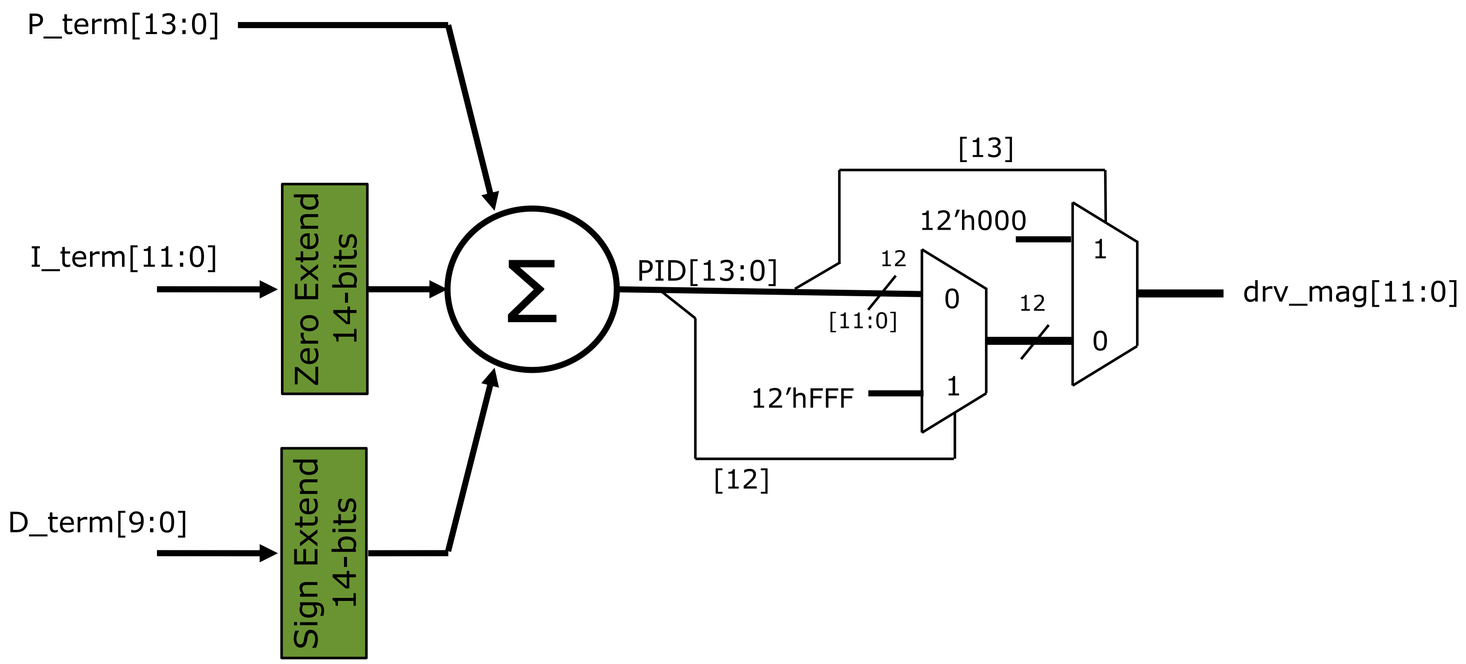


$$\frac{df(t)}{dt} = \frac{f(t) - f(t - \Delta t)}{\Delta t}$$

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)

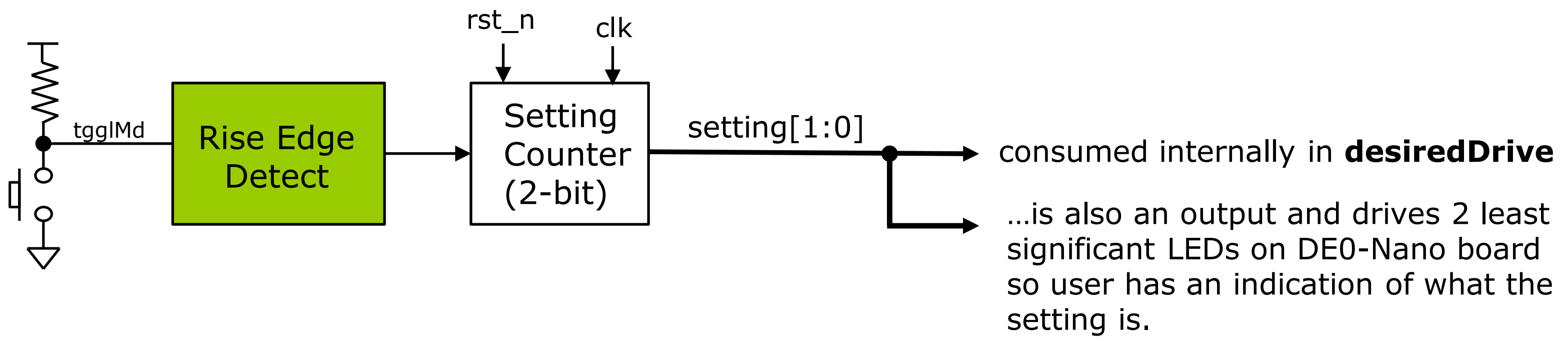


We sum the 3 terms together to form a 14-bit signal ***PID***. ***drv_mag*** is an unsigned numbers (e-bikes don't assist you to go in reverse). Is it possible for PID to be negative? Possibly, ***P_term*** and ***D_term*** could be negative while ***I_term*** was small. If ***PID*** is negative we clip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

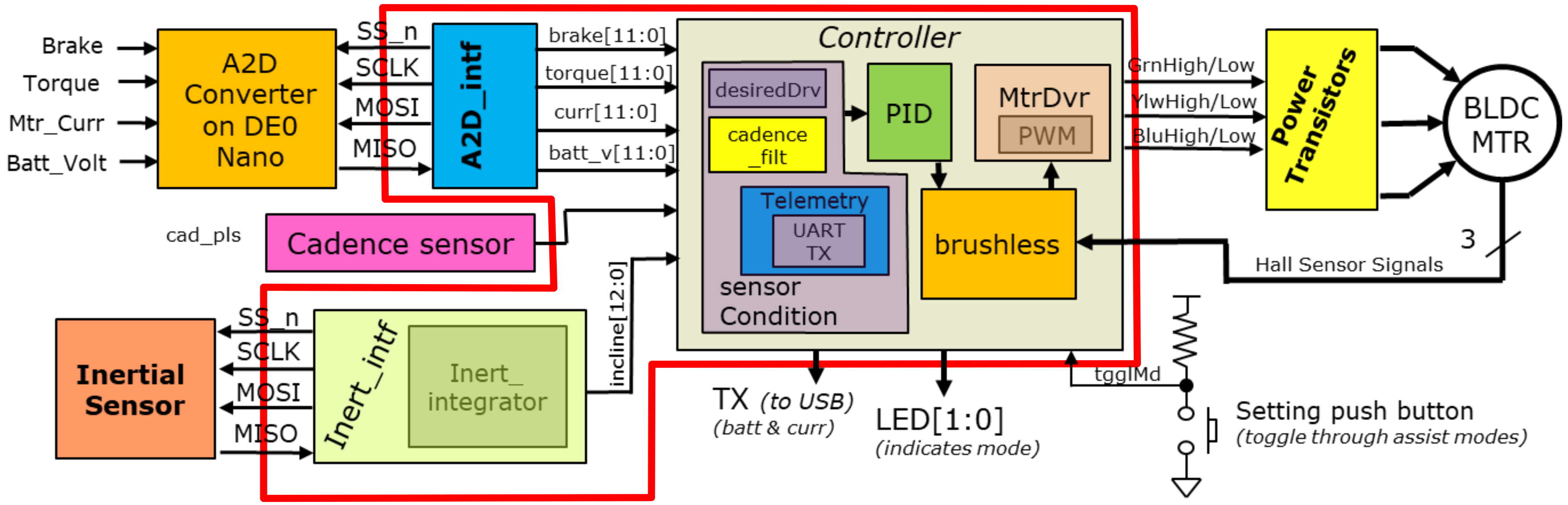

rst_n
clk
tgglMd
Rise Edge Detect
Setting Counter (2-bit)
setting[1:0]
consumed internally in **desiredDrive**
…is also an output and drives 2 least significant LEDs on DE0-Nano board so user has an indication of what the setting is.

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).

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.

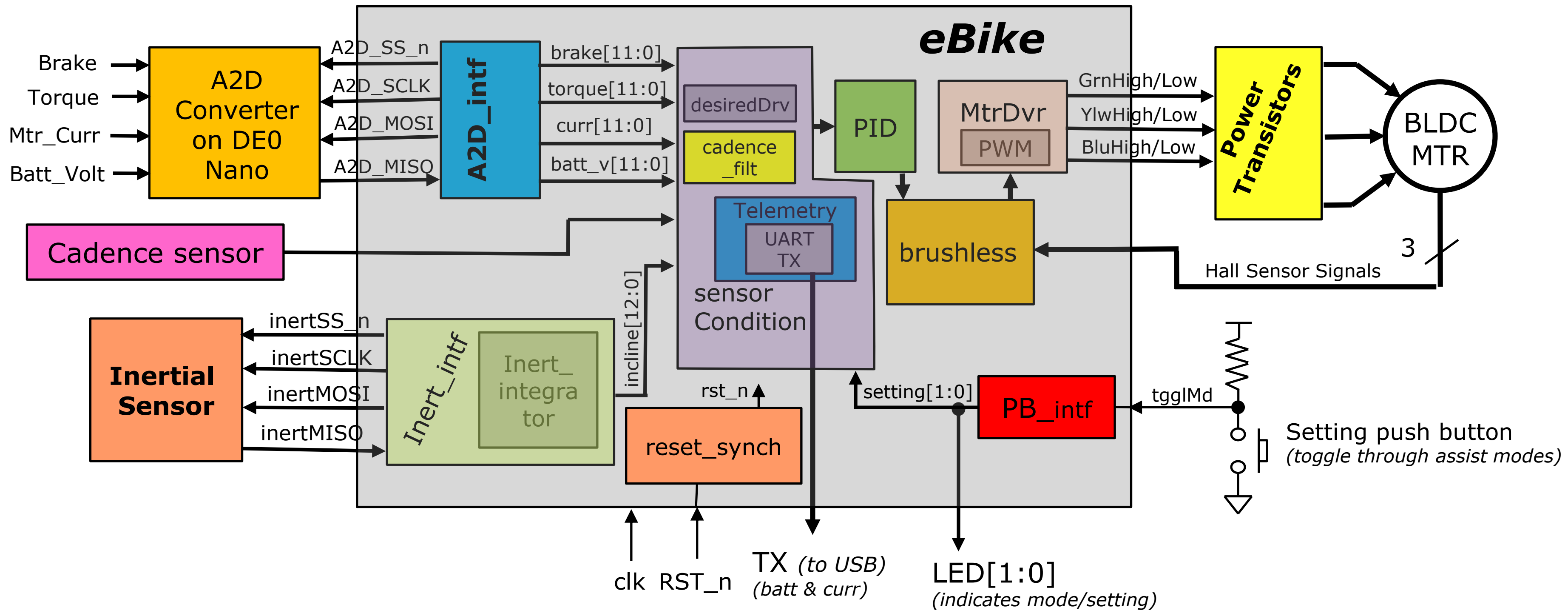
There is a level of hierarchy shown here called **Controller** that excludes **A2D_intf** and **Inert_intf**. This level of hierarchy is **not** required.

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 an EricWangnan 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

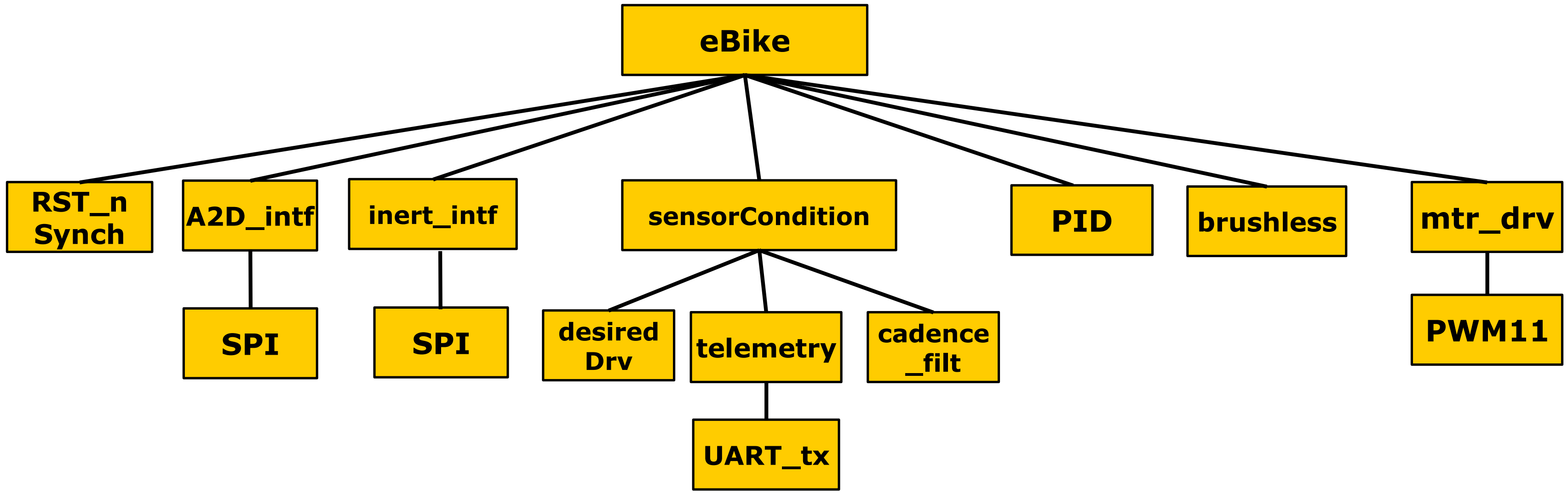
Signal Name:	**Dir:**	**Description:**
clk	in	Clock input (50MHz)
RST_n	in	Active low input from push button. Should be synchronized inside equalizer
tgglMd	in	From push button. Will cycle through the **setting**s.
setting[1:0]	out	Assist level setting. These bits drive LEDs on DE0-Nano so rider can 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 (250MHz) is mildly challenging. There is 1.5% extra credit if you achieve 300MHz (but your design will be larger too so watch that).

Synthesis Constraints:

Contraint:	Value:
Clock frequency	250MHz (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 250MHz minimum. There is 1.5% extra credit if you meet 300MHz.
Input delay	0.5ns 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 NAND2X1_RVT gate from our library
Output load	0.15pF on all outputs
Wireload model	16000 square micron size
Max transition time	0.15ns
Clock uncertainty	0.12ns

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

EricsSynthesizedArea = 14423

APR

- You must also use IC_Compiler to produce and APR block