Break your team into 2 teams of 2

- There are two exercises specified in Ex20
- One is sensorCondition (processes all the sensors). Creates **cadence_vec** from filtered cadence signal. Performs exponential averages on torque and current. Instantiates **desiredDrive** and determines loop error. Also instantiates **telemetry** because it is a convenient spot to do so.
- The other is the creation and test the inertial sensor interface (configures and reads inertial sensor to determine incline).
- One sub-team works on **sensorCondition.sv** and one works on **inert_intf.sv**

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***;

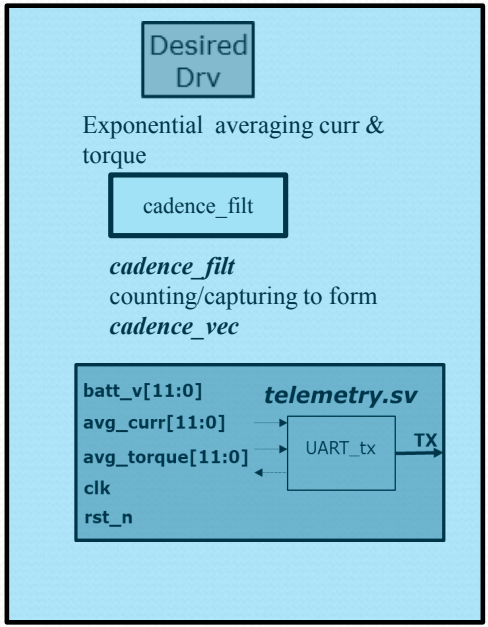
c. Perform an exponential average (of weight 32) on the raw 12-bit ***torque*** reading to produce ***avg_torque***. The accumulator of this exponential average should be seeded with 16X ***torque*** when pedaling resumes (***not_pedaling*** falls). This exponential average needs to be synchronized with 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. **not_pedaling** should also force **error** to zero.

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



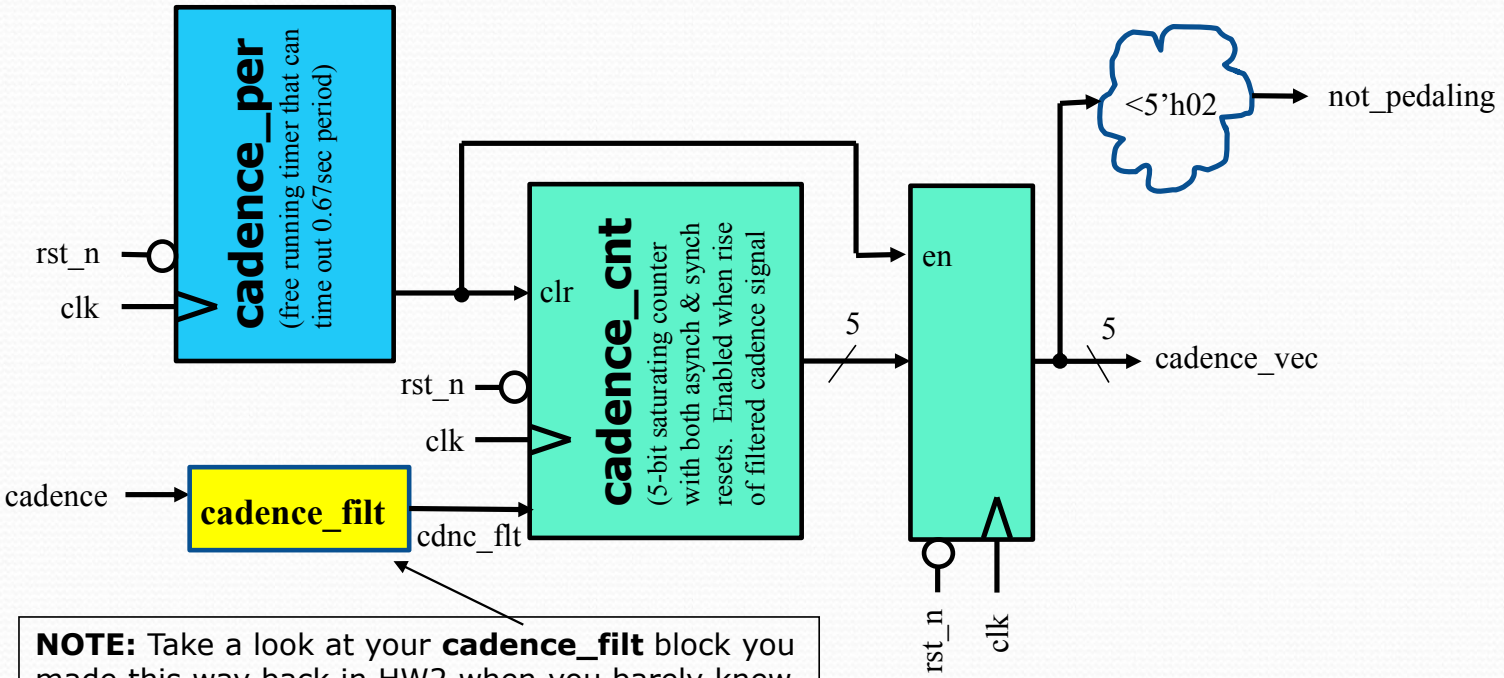
The rising edges of ***cadence_filt*** should be counted over a 0.67sec interval to form a 5-bit signal called ***cadence_vec.*** This 5-bit vector form of cadence feeds into **desiredDrv**.

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. Please call the counter you use for this **cadence_per**

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 exceed 5'h1F and roll over to report a smaller number.

sensorCondition (some hints on cadence_vec)



NOTE: Take a look at your **cadence_filt** block you made this way back in HW2 when you barely knew what you are doing. It probably needs work.

In addition to what you see here you will probably want to implement a falling edge detector on **not_pedaling**. When **not_pedaling** falls (i.e. when the rider resumes pedaling) we need to "seed" the torque exponential average with 16X the instantaneous **torque**. You might also want a rise edge detector on **cdnc_flt** because we only include new samples in the **torque_avg** on rising edges of **cdnc_flt**.

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.

Ours

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

4-bits wider W=16

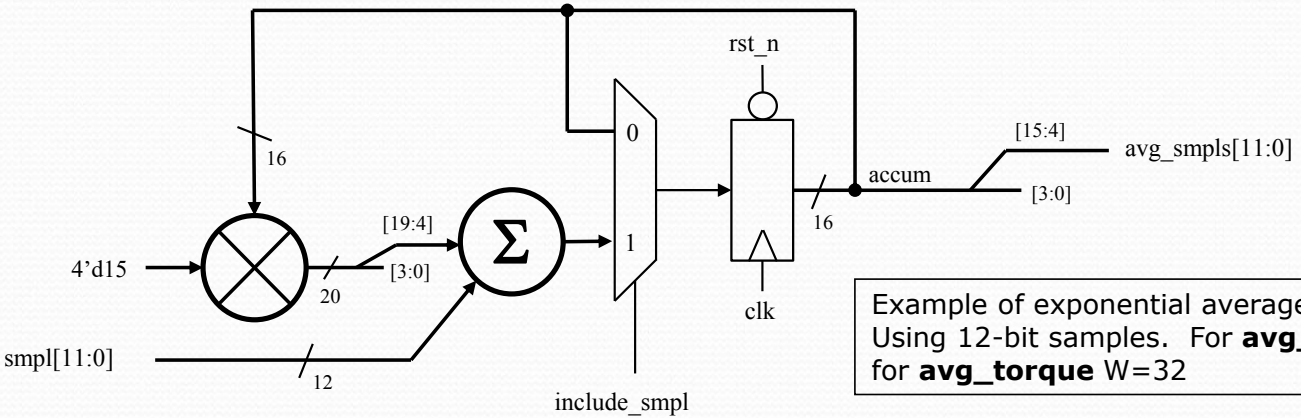
For every new sample (**smpl**) to be averaged the accumulator would be updated as:
accum = ((**accum*(W-1))/W) + smpl**.

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

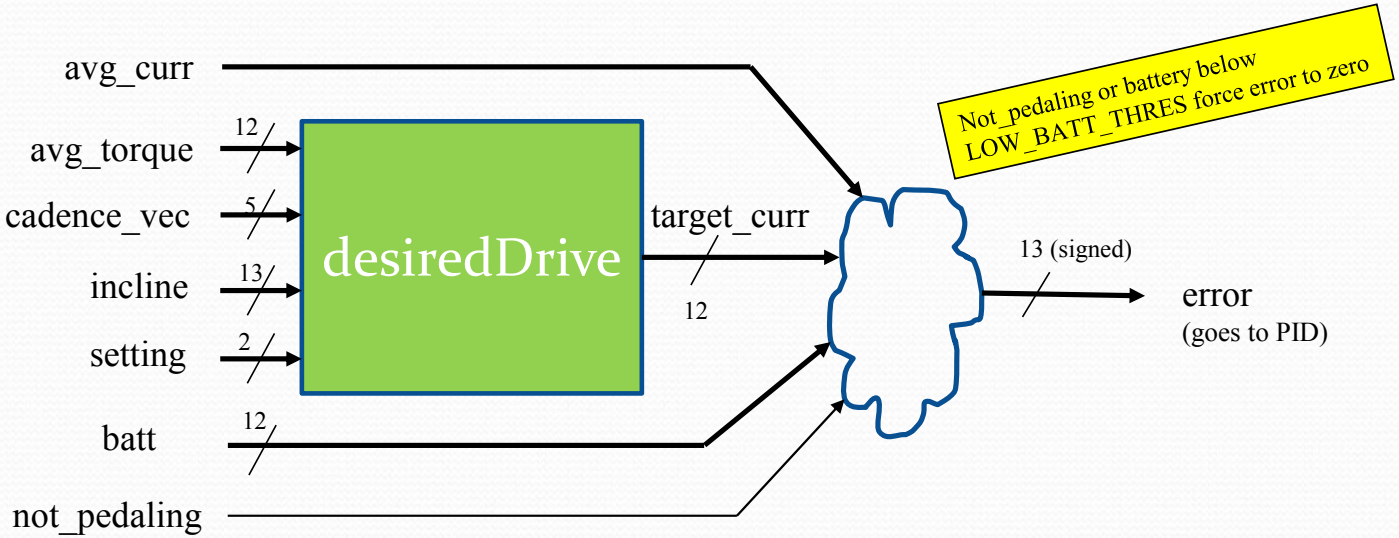
NOTE: new samples do not have to be included in the average every clock cycle (see ***include_smpl***)

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

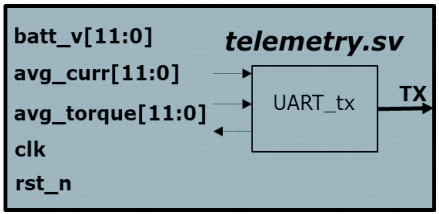
For **avg_torque** a new sample will be included once every rise edge of **cadence_filt.**

NOTE: the torque sensor has a large weight (W=32), so is slow to respond. When the rider 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 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)



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 (simulation & testing)

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.

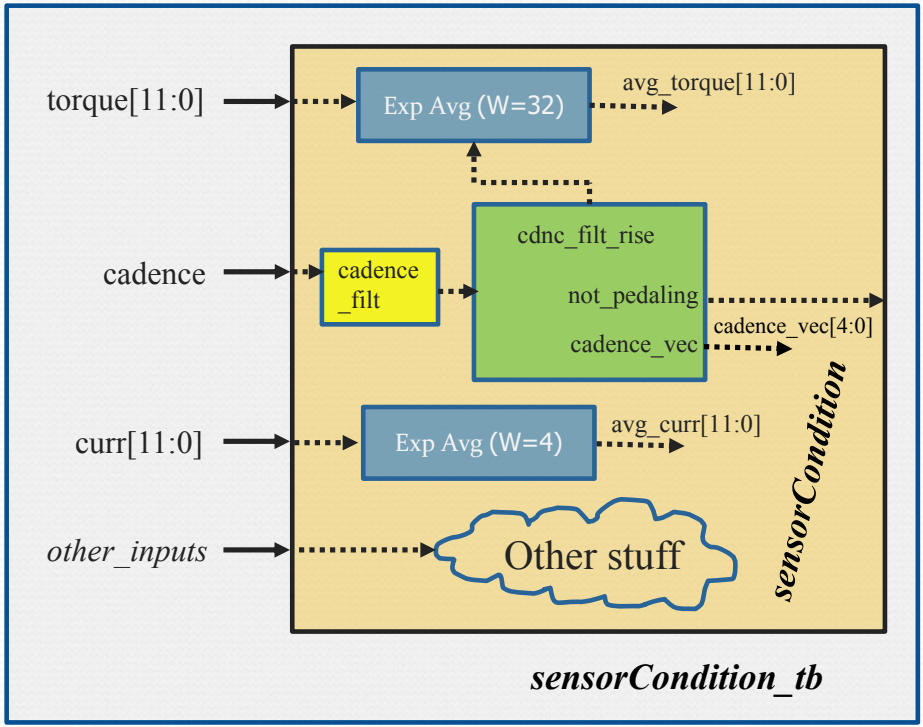
Again a **generate if** can work nicely for this.

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.

sensorCondition (simulation & testing)



Your goal is to just test your exponential average implementations and the **cadence_vec** circuitry. So **torque**, **cadence**, **curr** and of course **clk** & **rst_n** are the only inputs you need to be concerned about.

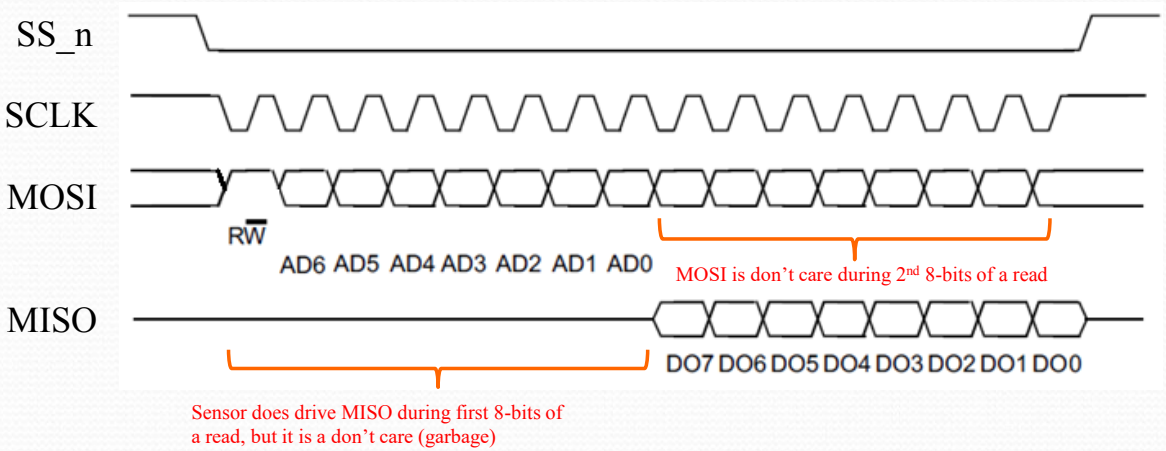
Of course your testbench is passing in **FAST_SIM** as 1 to speed up simulation. A cadence period of around 4096 clocks is the right ballpark

Apply static **torque** and **curr** values. Do your exponential average blocks seem to give a reasonable result after a number of updates?

With this testing you are not looking at primary outputs of **sensorCondition**. You instead are trying to check if some of the internal values (like **avg_torque**, **cadence_vec**, & **avg_curr**) of **sensorCondition** are good. A parent can observe the internal signals of its children using the . Operator. If the instance of **sensorCondition** in your testbench is called **iDUT**, then **iDUT.avg_torque** would be a signal the testbench can observe.

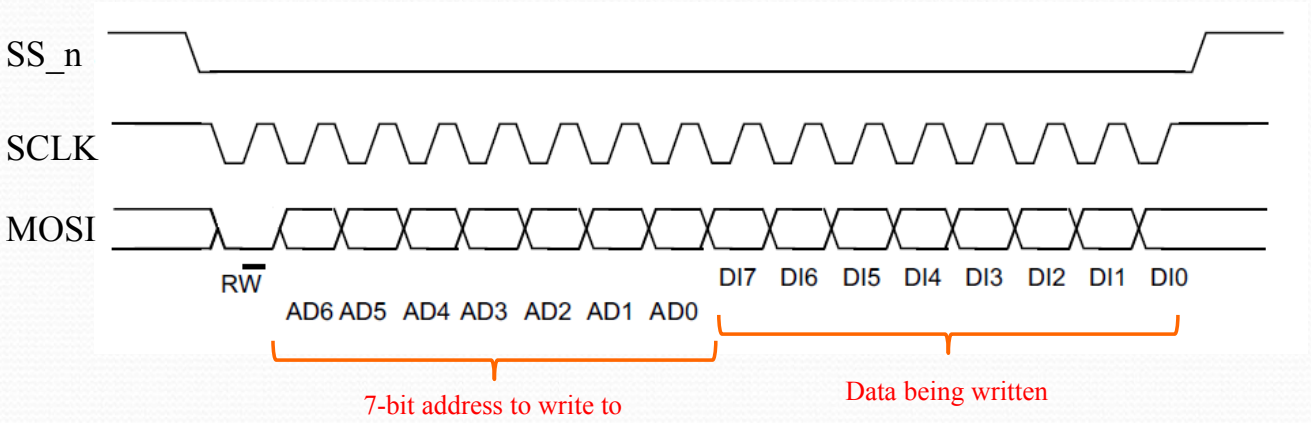
Inertial Sensor Interface

- 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)



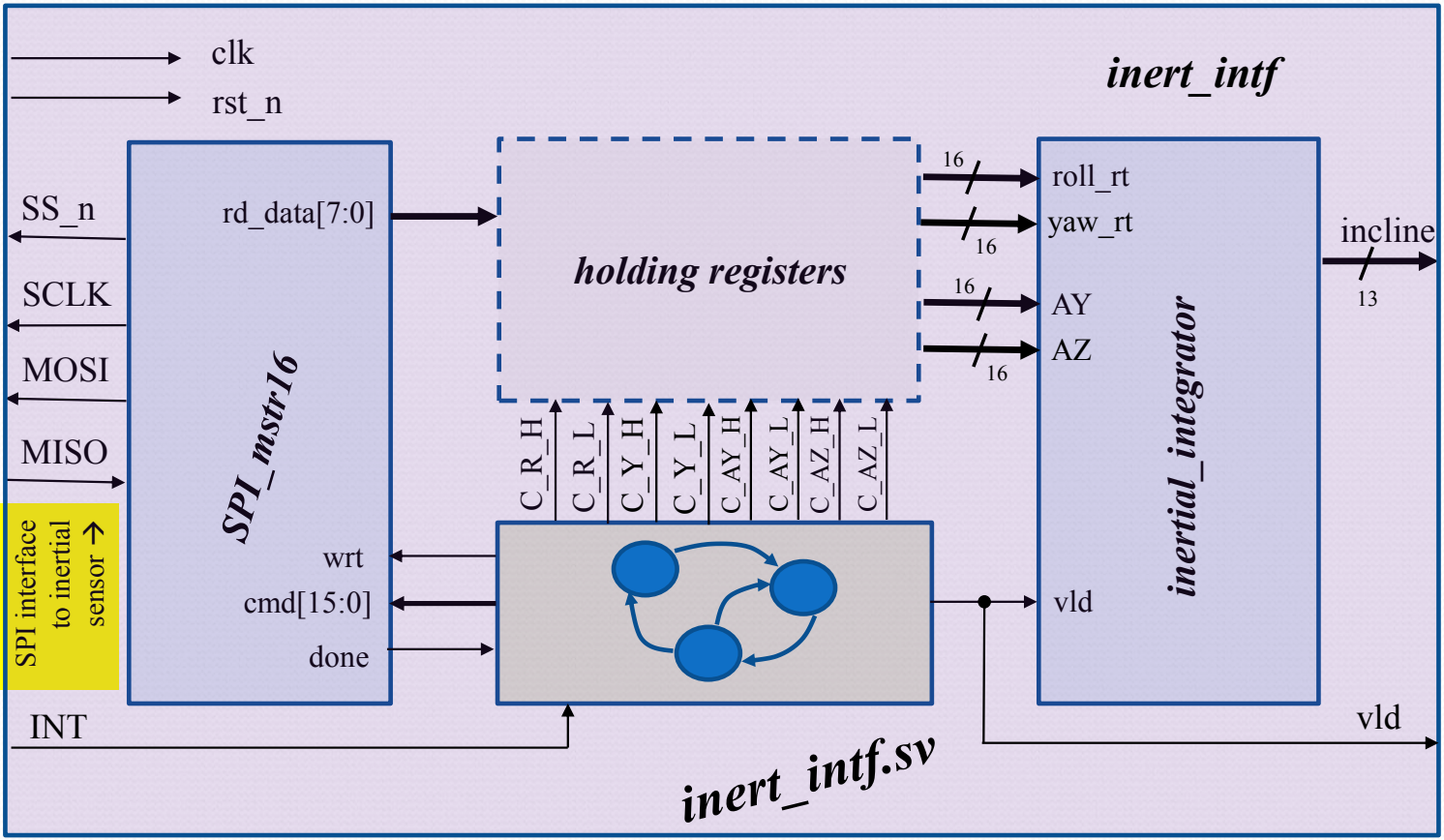
Inertial Sensor Interface (write)

- 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)



- 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

Does this description sound vaguely familiar? The design problem of the midterm was a baby version of this SM.

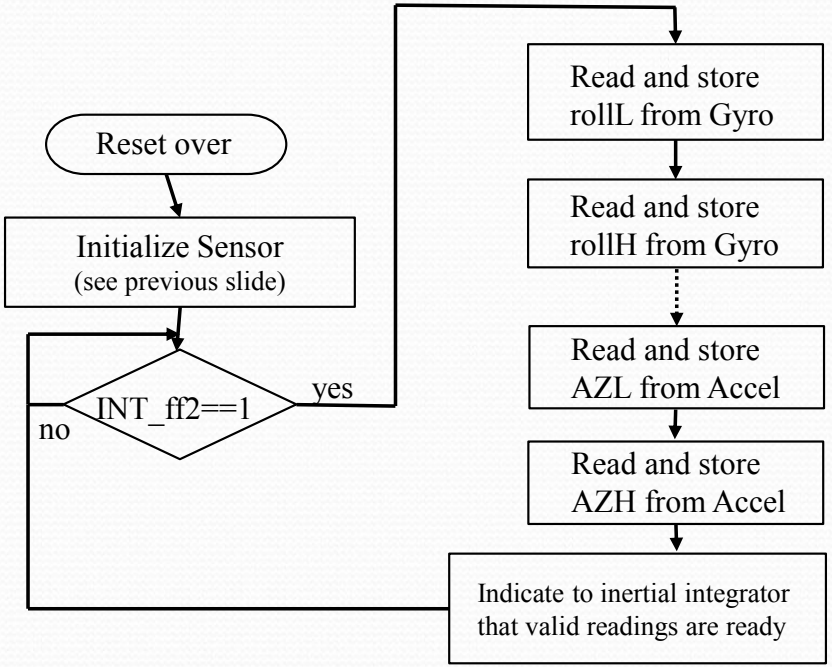
- After reset de-asserts we will **wait for a 16-bit timer to expire**. This ensures the inertial sensor chip is also done with its initialization and ready to accept commands via the SPI bus.
- The system must write to some registers to configure the inertial sensor to operate in the mode we wish. The 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.



- 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 you send is a don't care. The lower byte of the data you receive (via MISO) is the data the inertial sensor is returning.

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)

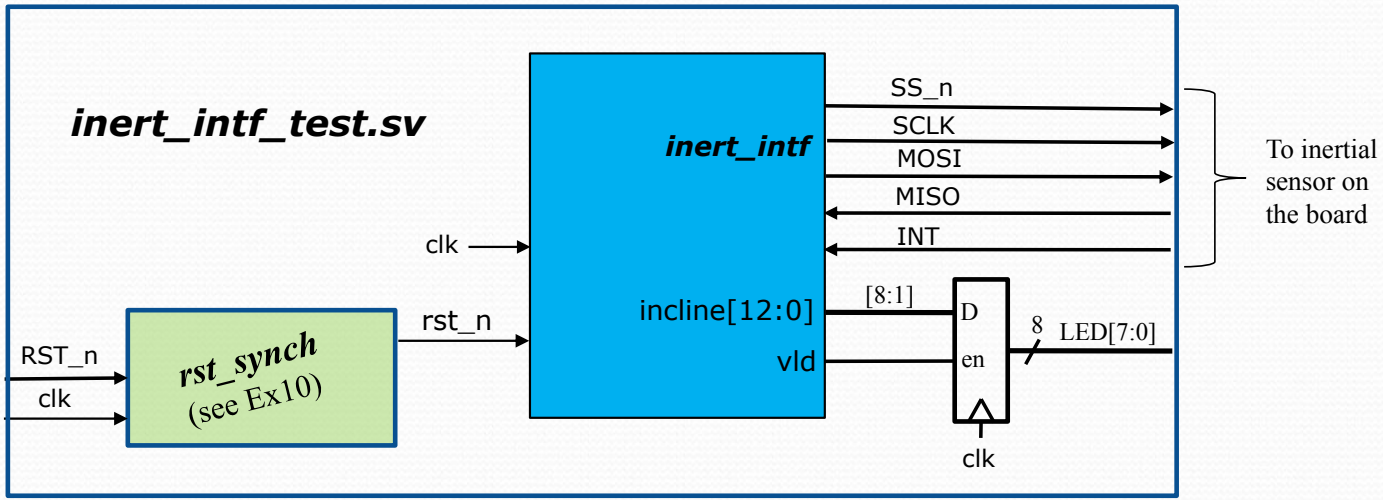
- The block **inertial_integrator.sv** takes the raw sensor readings and performs the sensor fusion. It computes both incline and roll internally, but only **incline** an output.

- 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.

inert_intf (testing it by mapping to test platform)



Create the block **inert_intf_test.sv** as you see here. It will instantiate your **inert_intf.sv** block and a copy of **rst_synch**. Infer an enable flop to grab bits [8:1] of ***incline***. These flops (***LED[7:0]***) will be driven onto the LEDs of the DE0-Nano board.

An **inert_intf_test.qsf** and **inert_intf_test.qpf** file have been provided. Compile your design in Quartus and map it to a board on one of the test stations. As you tilt the board counter clockwise (uphill) you should see the value on the LEDs display positive increasing numbers. As you tilt it clockwise (downhill) the numbers should become negative.