ECE 551
HW3

- Due Weds Mar 9th in class
- Work Individually
- Remember What You Learned From the Cummings SNUG paper
- Use descriptive signal names and comment your code

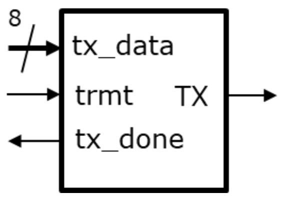
HW3 Problem 1 (**20pts)** Telemetry

You do this one on your own

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.

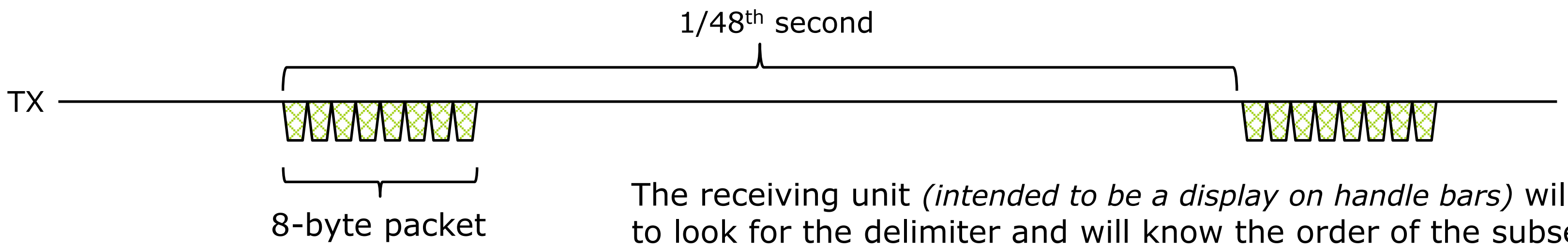
A UART transmitter sends a byte at a time serially over the TX line. Its operation is quite simple. You present a byte you wish it to transmit on ***tx_data[7:0]*** and then hold ***trmt*** high for one clock cycle. When it has completed transmitting that byte it will indicate it by raising ***tx_done***.

UART Transmitter



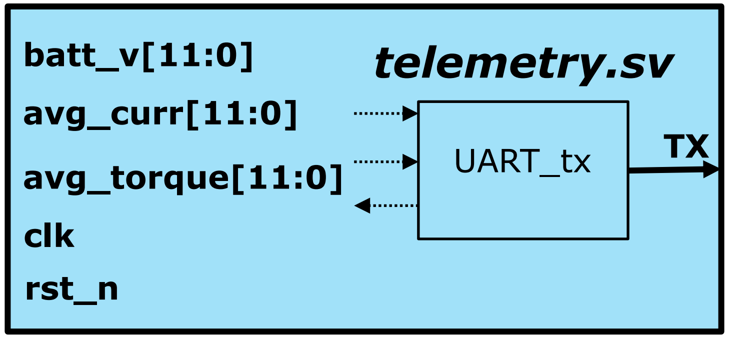
You will make a wrapper around **UART_tx.sv** that will periodically send out ***batt_v[11:0], avg_curr[11:0]*** and ***avg_torque[11:0]***. 47.68 times a second (hmm…interesting number…I wonder why that was chosen (recall 50MHz clk)) your SM will leave its IDLE state and start sending a sequence of 8 bytes (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 {4'h0,***batt_v[11:8]***}	low byte ***batt_v[7:0]***	high byte {4'h0,***avg_curr[11:8]***}	low byte ***avg_curr[7:0]***	high byte {4'h0,***avg_torque [11:8]***}	low byte ***avg_torque[7:0]***

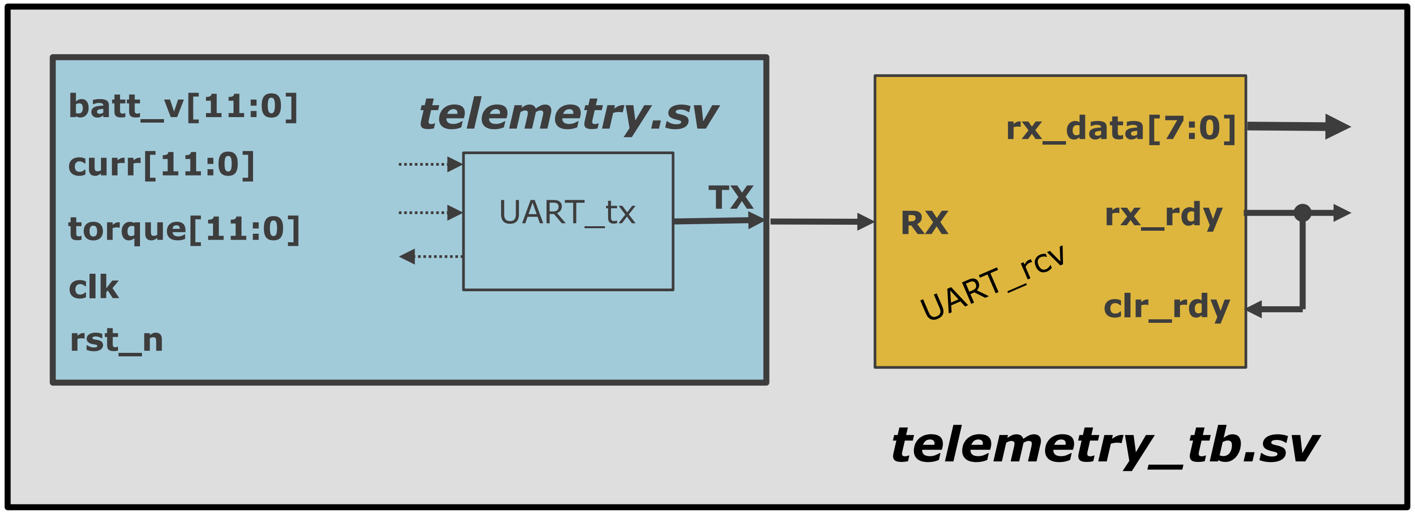


The receiving unit *(intended to be a display on handle bars)* will know to look for the delimiter and will know the order of the subsequent bytes, hence can decode and display the data. A UART receiver (**UART_rcv.sv**) is provided to aid in testing.

HW3 Problem 1 (**20pts)** Telemetry



You are to create **telemetry.sv** with the interface shown. Of course you also need to make a testbench to ensure its correct operation. **UART_rcv.sv** *(available on Canvas page)* can be useful in your testbench.

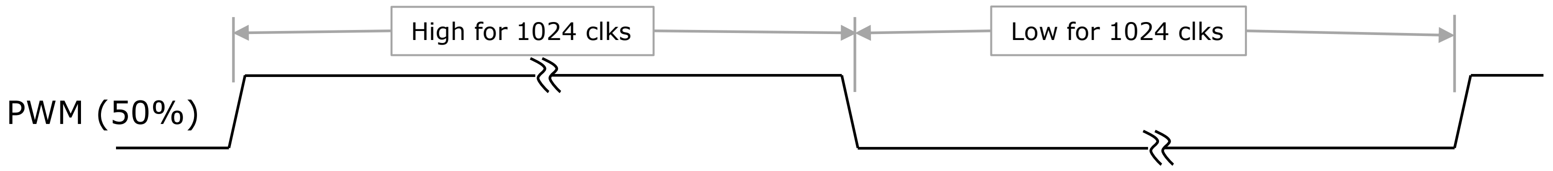


Submit: **telemetry.sv, telemetry_tb.sv** and proof you ran the testbench. (Use good SM coding style for **telemetry.sv**)

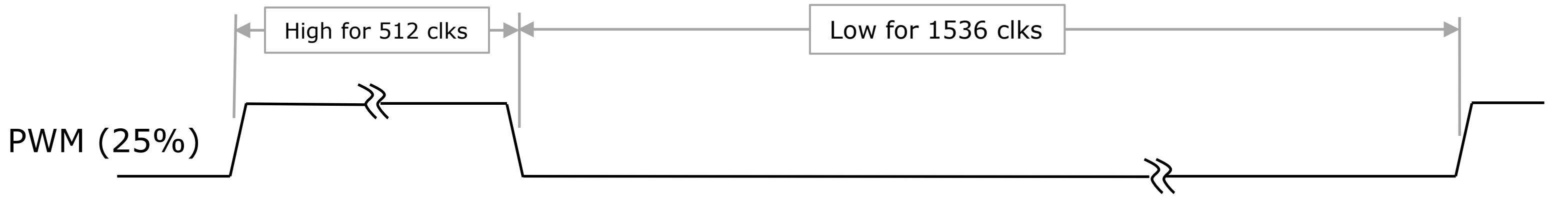
HW3 Problem 2 **(20pts)** PWM

Started as in Class Exercise09 on Feb 21st

- Obviously 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 (211). Since our system clock is 50MHz this is 40usec.



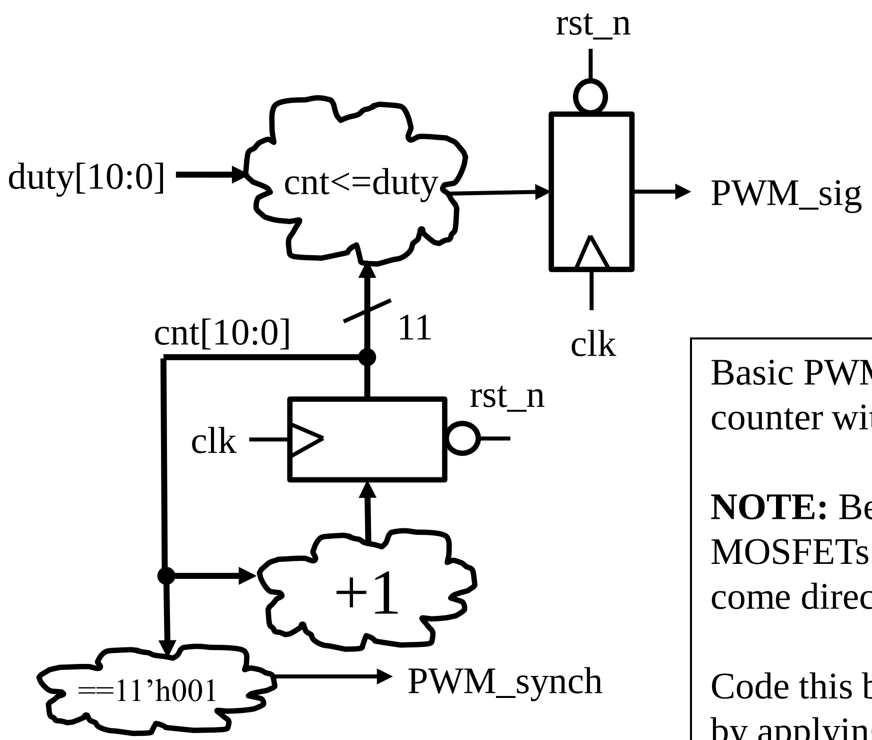
- Second example is 25% duty cycle



HW3 Problem 2 (20pts) PWM

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'h001 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.

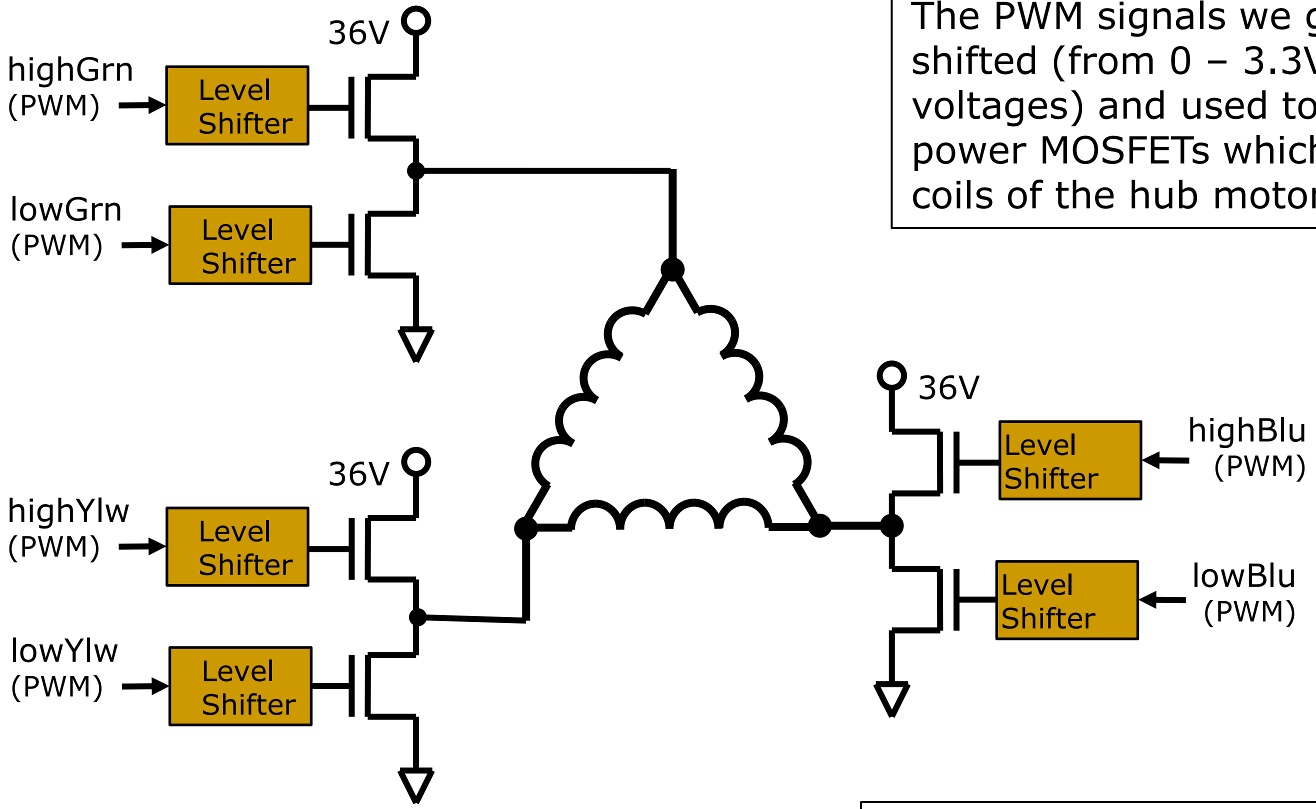
Code this basic PWM and create a testbench for it. Simulate by applying a few different duty[10:0] input values and observing the output.

Submit: PWM.sv, **PWM_tb.sv**, and waveform images.

HW3 Problem 3 (15pts) non-overl

Started as in Class Exercise10 on Feb 25th

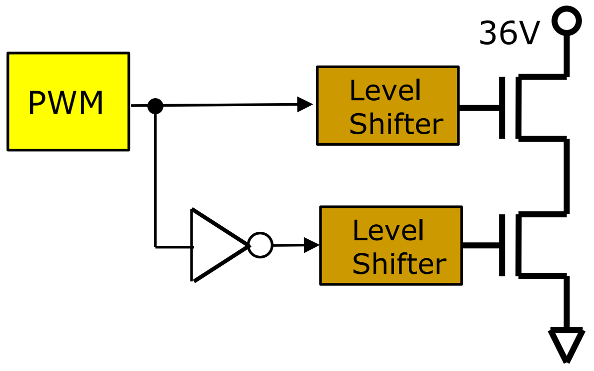
The PWM signals we generate are level shifted (from 0 – 3.3V signals to higher voltages) and used to drive the gates of power MOSFETs which in turn drive the coils of the hub motor.



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 vs the low driver.

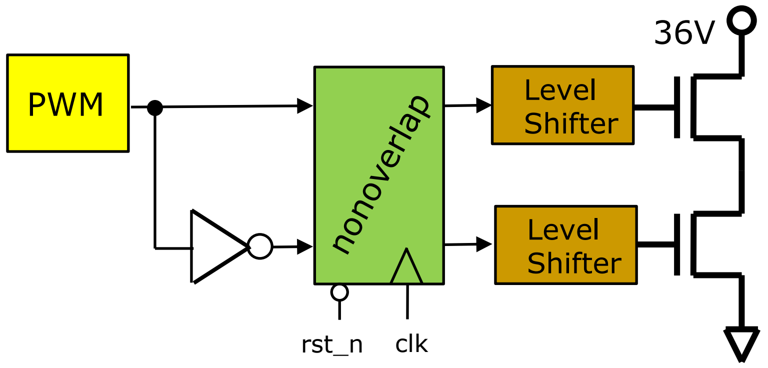
The power MOSFETs are very powerful (very low ohmic and capable of passing a lot of current…like 100A). Given the proper conditions they can self-destruct. Actually that would be the "improper conditions".

HW3 Problem 3 **(15pts)** non-overlap



Given the power of the 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 usec)* 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 *(after level shifting)* will never overlap. This non-overlap block will create a dead time (32 clocks) where both output signals are low for a while whenever an input changes.

HW3 Problem 3 (**15pts)** non-overlap

nonoverlap.sv specifications:

- Whenever ***highIn*** or ***lowIn*** change both ***highOut*** and ***lowOut*** should go low on the next clock cycle (Next rising clk edge).

- 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 after the 32 clk cycle deadtime.

See next slide for implementation hints

HW3 Problem 3 (**15pts)** non-overlap implementation

nonoverlap.sv implementation hints:

- You will need a 5-bit counter (dead time counter) that can be cleared via a signal (like when there is a change). It could possibly be free running, but you might want to have an enable signal.
- ***highOut*** and ***lowOut*** should come from flops that are asynch reset to zero, synchronously set to zero (under the condition of changed inputs) or a copy of ***highIn*** & ***lowIn*** if dead time has expired.
- You may want a simple state machine to control it, although it can be implemented without.

nonoverlap.sv testing:

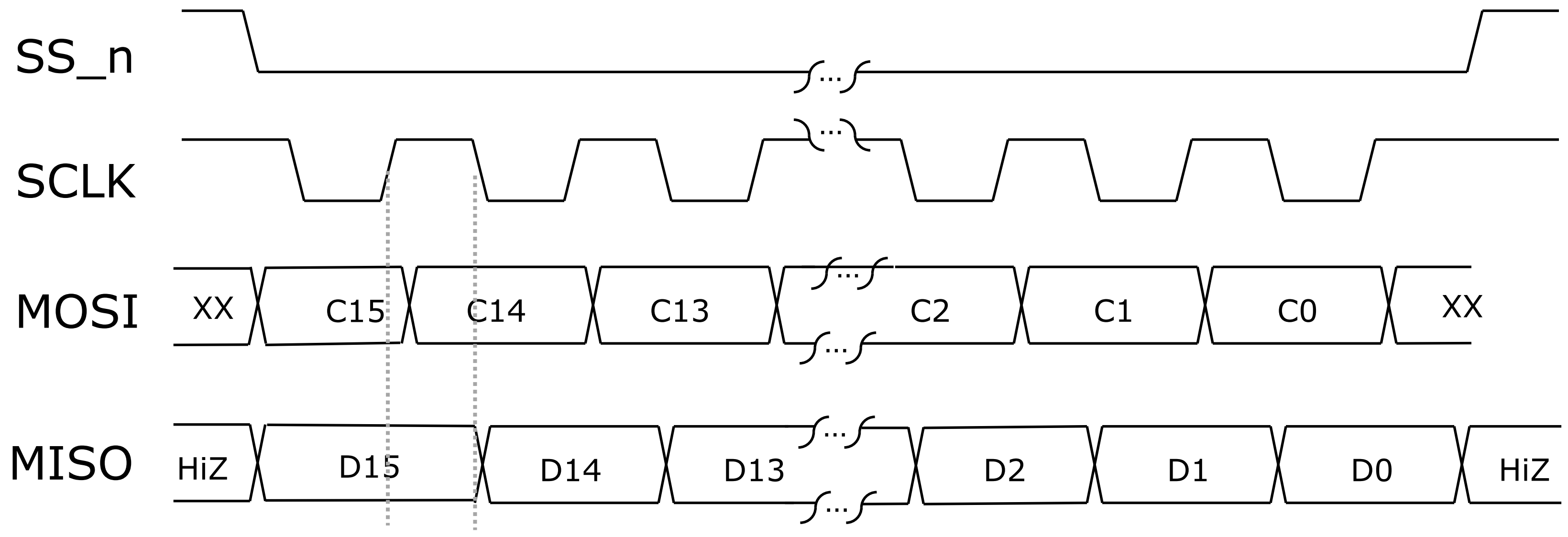
- The testbench can have a single stimulus signal and its inverse feeding highIn/lowIn. Upon a change in this signal both outputs should go low for 32-clocks, then one of them should go high.

Turn in **nonoverlap.sv** along with a self-checking testbench (**nonoverlap_tb.sv**) and proof the test bench was run/debugged.

HW3 Problem 4 (**45pts**) 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.

HW3 Problem4 (45pts) SPI Packets



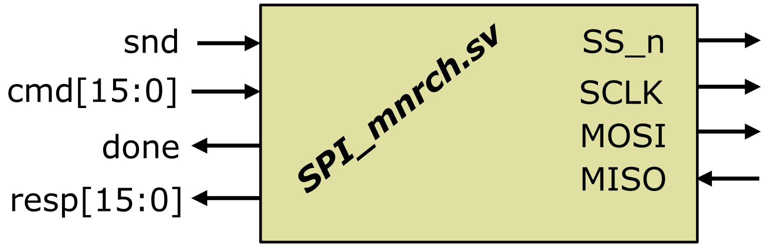
A SPI packet inherently involves a send and receive (full duplex). The full duplex packet is always initiated by the monarch. The monarch controls SCLK, SS_n, and MOSI. The serf drives MISO if it is selected. If the serf is not selected it should leave MISO high impedance. The inertial sensor and an A2D converter are the SPI peripherals we have in our system.

The SPI monarch 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 serfs sample MOSI on the positive edge of SCLK, and change 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 monarch.

SCLK will be 1/32 of our system clock (50MHz/32 = 1.5625MHz

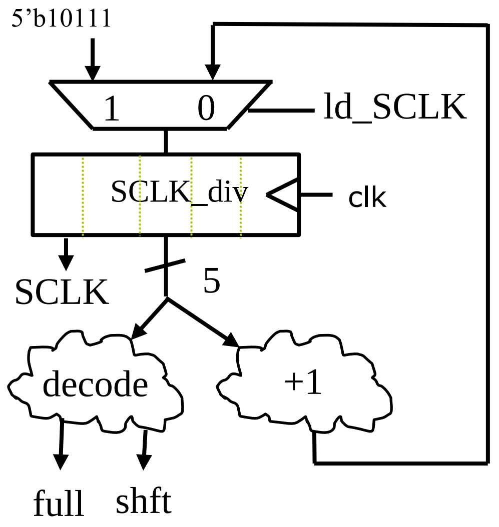
HW3 Prob4 (45pts) SPI Unit

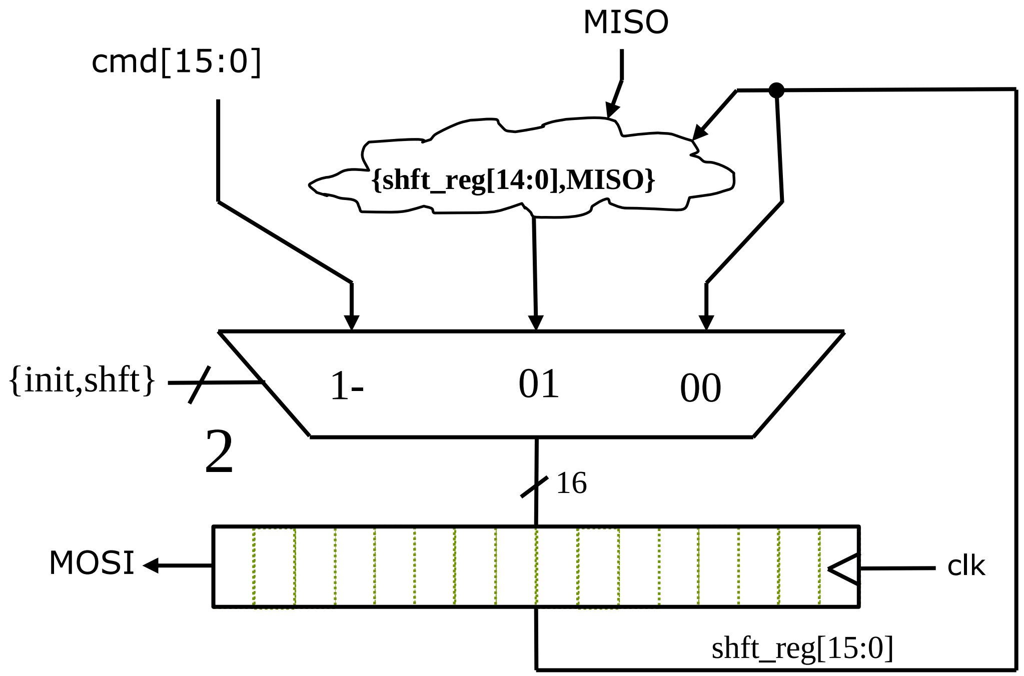
- You will implement **SPI_mnrch.sv** with the interface shown.
- SCLK frequency will be 1/32 of the 50MHz clock (i.e. it comes from the MSB of a 5-bit counter running off **clk**)
- I had better not see any ***always*** blocks triggered directly on **SCLK**. We only use **clk** when inferring flops.
- 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]

HW3 Prob4 (45pts) SPI Implementation:

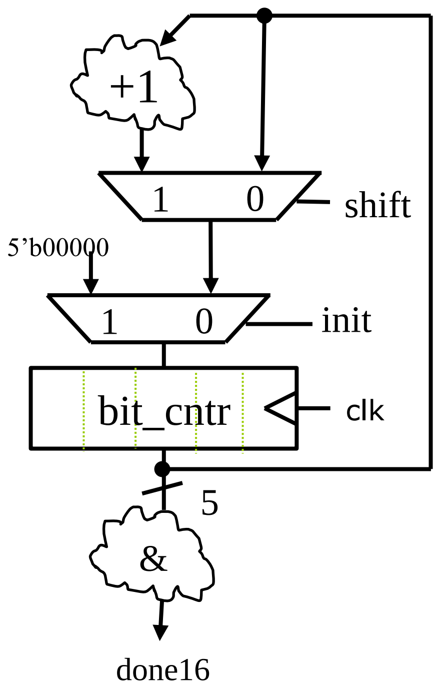


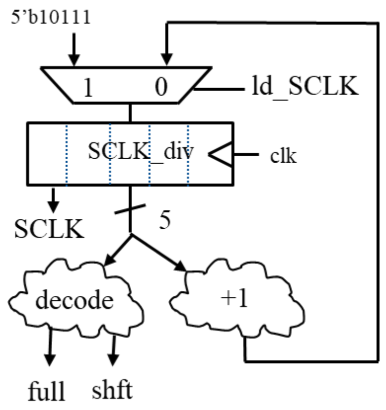


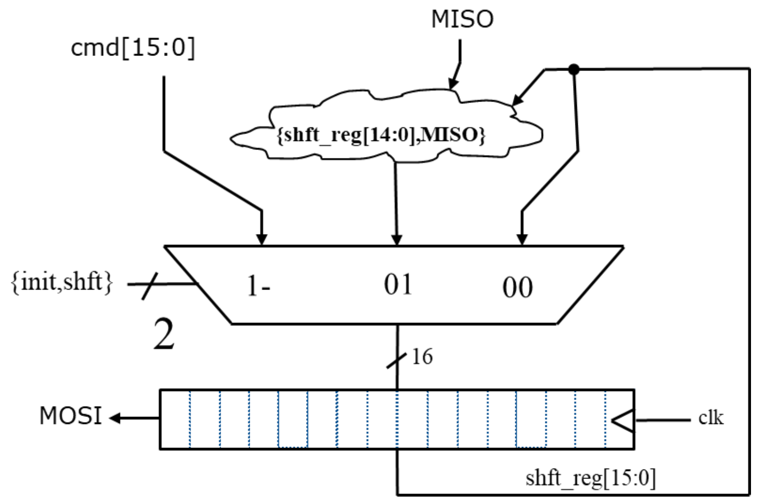
The main datapath of the SPI monarch consists of a 16-bit shift register. The MSB of this shift register provides **MOSI**. The shift register can be parallel loaded with the data to send, or it can left shift one position taking **MISO** as the new LSB, or it can simply maintain.

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?

HW3 Prob4 (45pts) SPI Implementation:

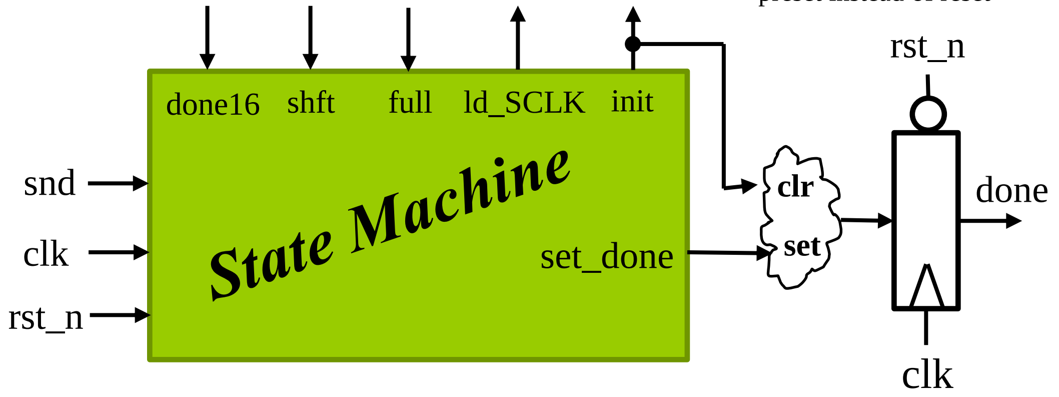




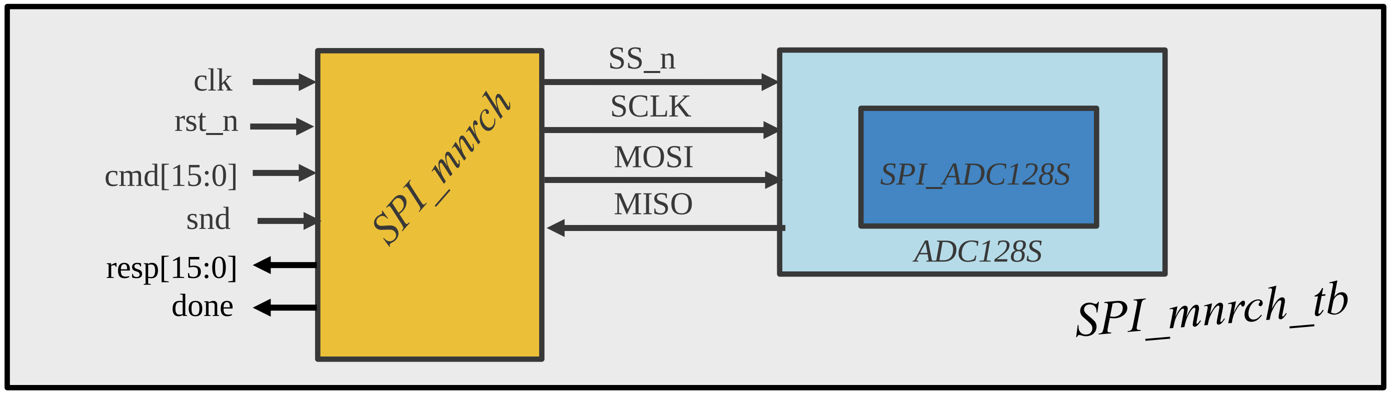


In addition to **SCLK_div** and main shift register you also need a **bit_cntr** to keep track of how many times the shift regiter has shifted. Of course you also need a state machine.

Very similar implementation to generate SS_n except preset instead of reset



HW3 Prob4 (45pts) Testing SPI_mnrch.sv



- Download **ADC128S.sv** (model of A2D converter on DE0-Nano, and a SPI serf)
- Also download **SPI_ADC128S.sv** (child of ADC128S.sv that you need)
- Create a testbench in which your **SPI_mnrch.sv** drives the **ADC128S**. Test and debug.
- To read a channel from the ADC128S you send: {2'b00,chnl[2:0],11'h000} (i.e. the channel is specified by bits [13:11] of the packet you send.
- During a read the ADC128S is returning the channel you requested in the last SPI packet *(since it obviously cannot respond with data for the current SPI packet since you are just now telling it what channel you want)*.

HW3 Prob4 (45pts) Testing continued:

- The response of ADC128S is: 0xC00 + chnnl for the first two reads. The 0xC00 part decrements by 0x10 for every 2 reads. For the first read it assumes you are reading channel 0 so it would return 0xC00.

- The table below outlines the behavior if you gave it 4 reads in a row:

NOTE: when performing consecutive reads to the **ADC128S.sv** model you have to give it a clock period to breath between transactions. So delay one system clock after ***done*** before sending another SPI transaction.

Channel Read	Expected Response	Description:
1	0xC00	You are requesting channel 2 for next time, but it returns channel 0 for first read.
1	0xC01	Has not decremented 0xC00 by 0x10 yet, but this is channel 1 from last request
4	0xBF1	Two reads have been performed so it decremented by 0x10, but this is still channel 1.
4	0xBF4	This is a channel 4 response from last request

- **Submit:**
  - **SPI_mnrch.sv**
  - Your testbench (**SPI_mnrch_tb.sv**) (should be self-checking, and I recommend what is shown in table above)
  - Output from your self checking test bench proving you ran it successfully