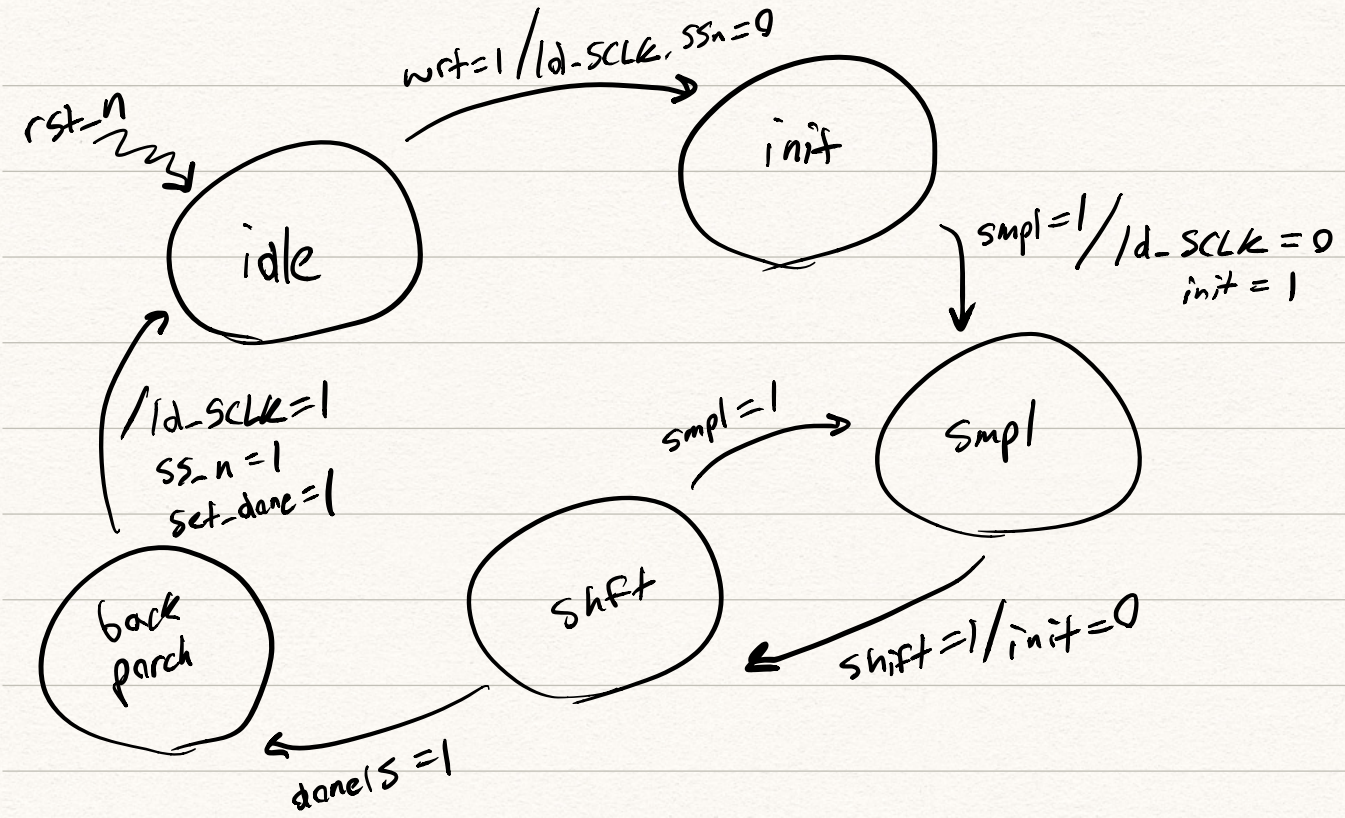
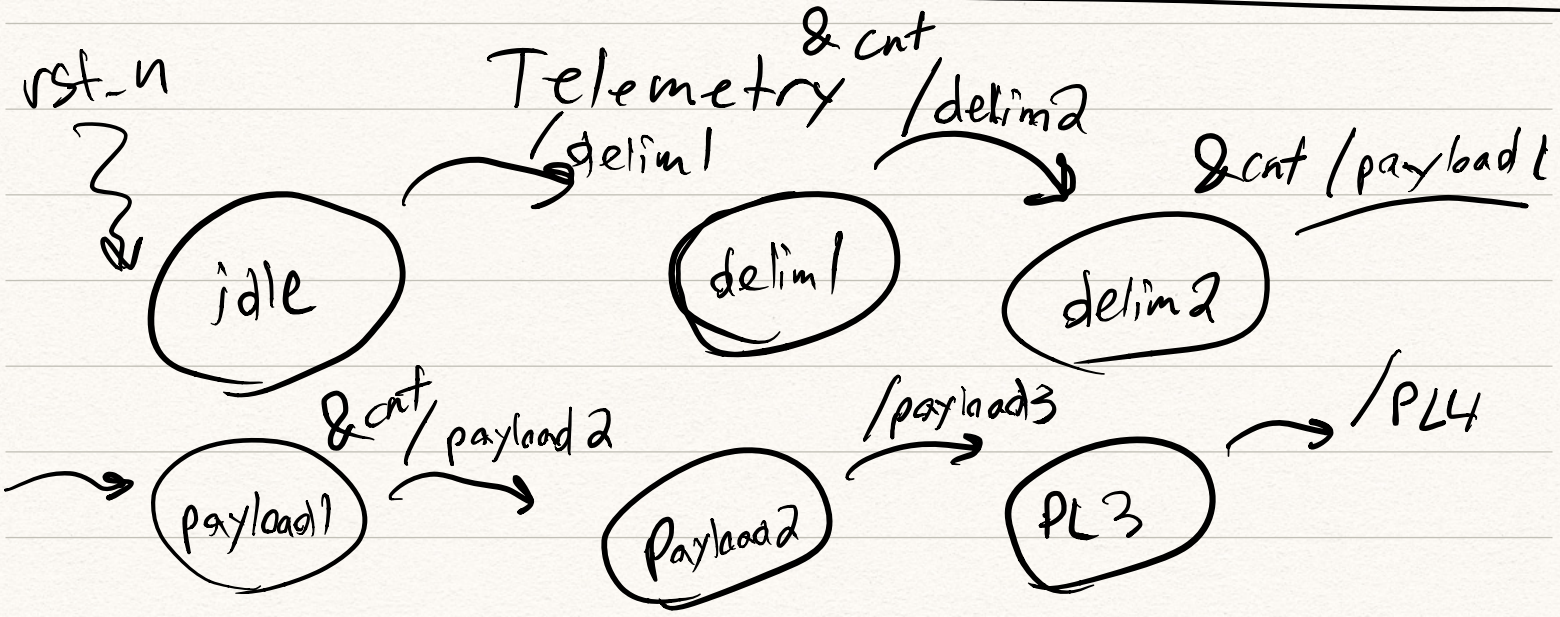




Test : batt_v : 011100

ECE 551
HW3

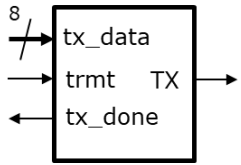
- Due Weds Oct 16th in class
- Work Individually
- Remember What You Learned From the Cummings SNUG paper
- Use descriptive signal names and comment your code

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.

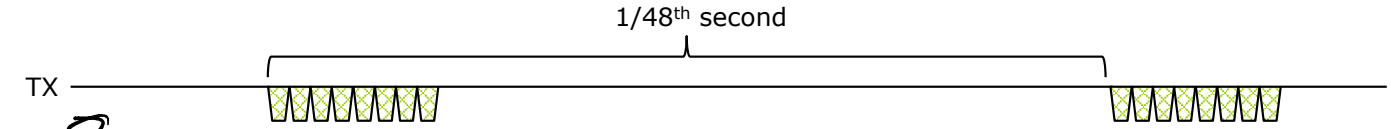
UART Transmitter



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***.

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 Hoffman chose that) your SM will leave it 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]***

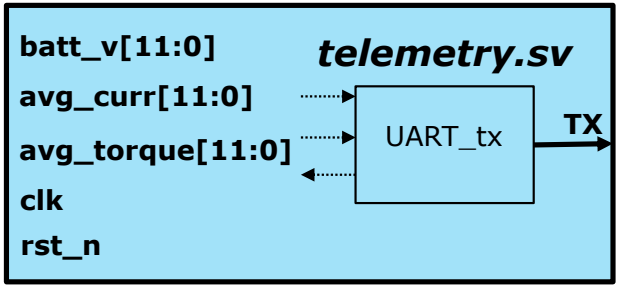


$\frac{8}{5\times10^6}$

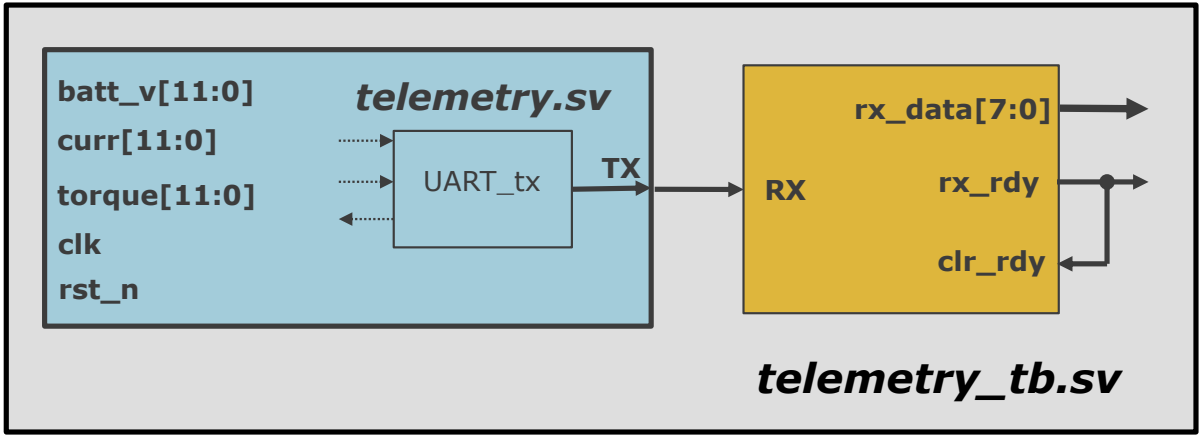
20 bit cnt

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.

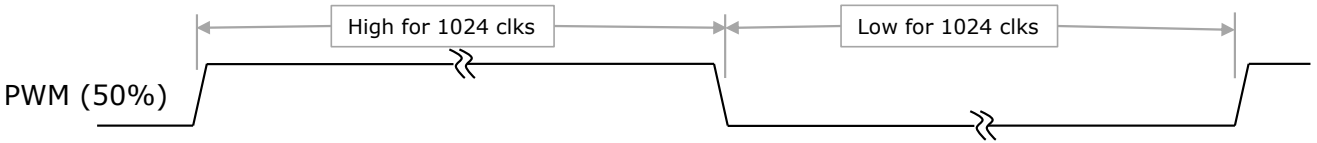


Submit: **telemetry.sv, telemetry_tb.sv** and proof you ran the testbench. (Use good SM coding style for **telemetry.sv**)

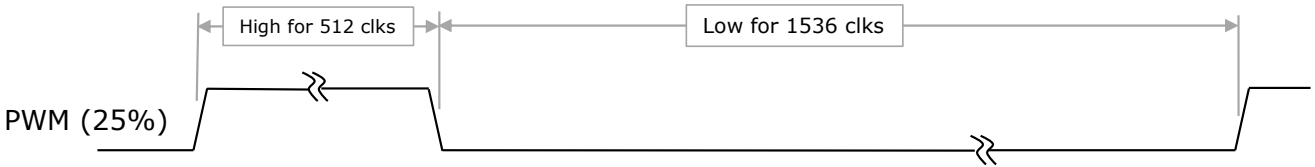
HW3 Problem 2 (**20pts**) PWM

Started as in Class Exercise07 on Oct 4th

- 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.



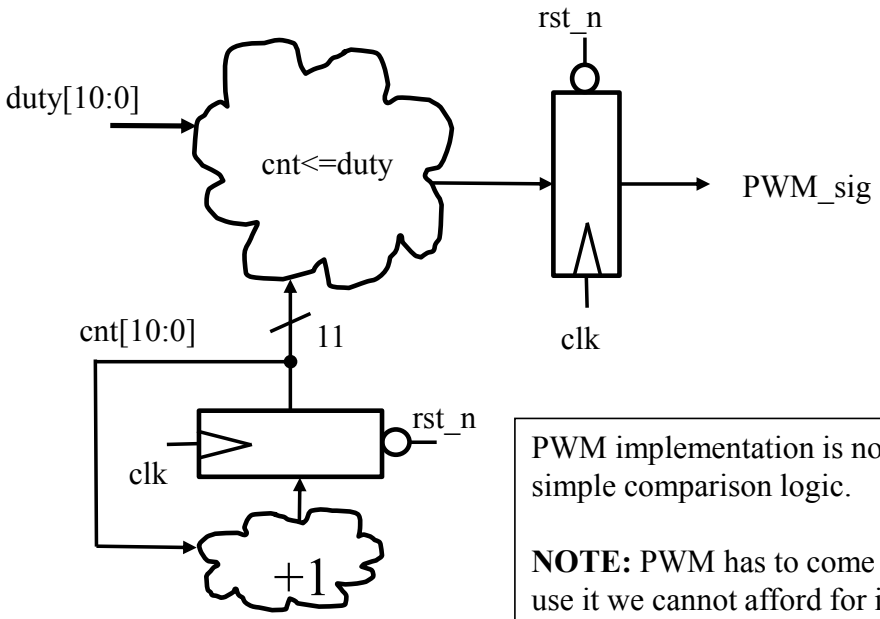
- Second example is 25% duty cycle



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)



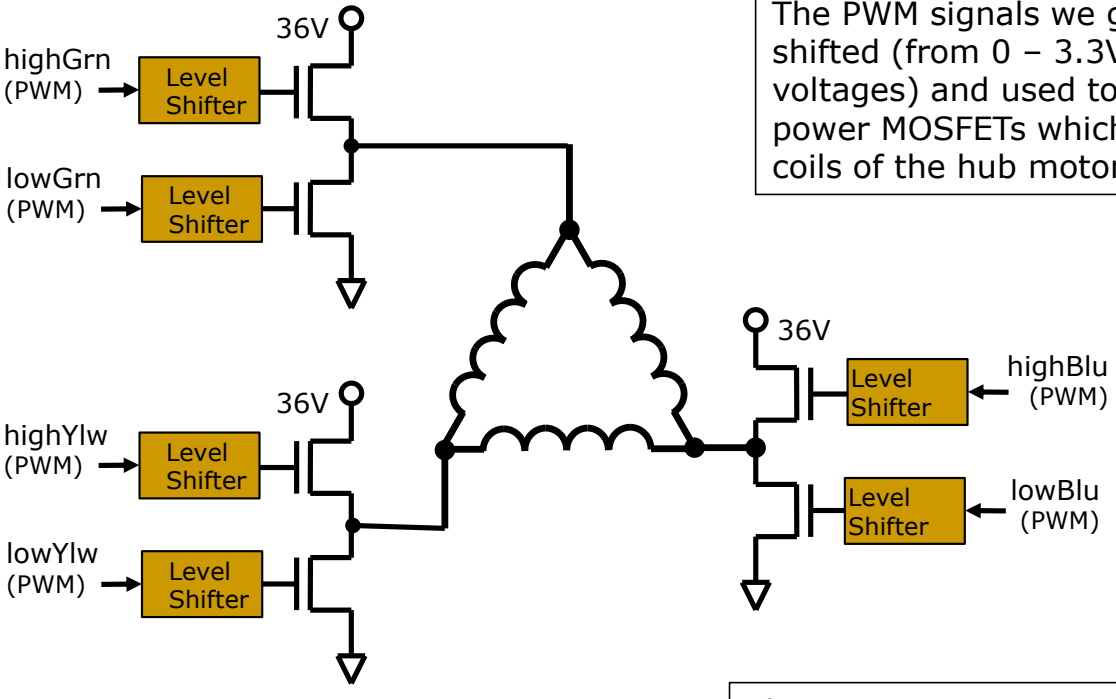
PWM implementation is not too hard. Just an 11-bit counter with simple comparison logic.

NOTE: PWM has to come directly out of a flop, because of how we use it we cannot afford for it to glitch.

Submit **PWM11.sv**, **PWM11_tb.sv**, and proof you ran your testbench

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

Started as in Class Exercise08 on Oct 7th

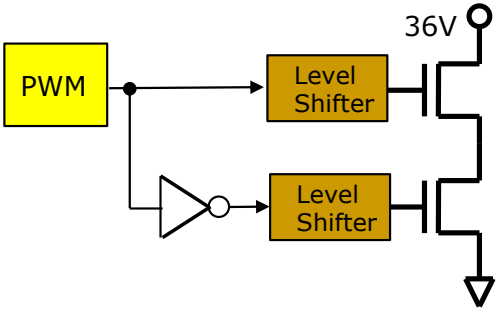


The PWM signals we generate are level shifted (from 0 – 3.3V signals to higher voltages) and used to drive the gates of power MOSFETs which in turn drive the coils of the hub motor.

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.

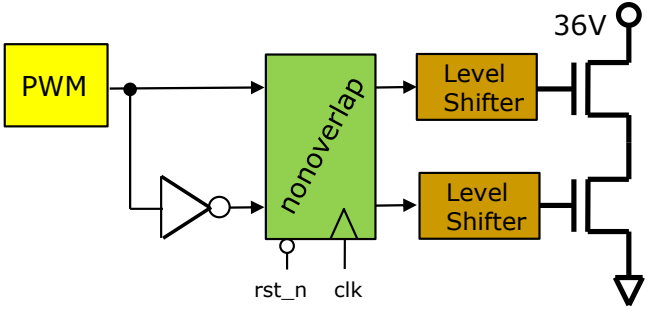
The power MOSFETs are very powerful (very low ohmic and capable of passing a lot of current…like 100A). Given the proper conditions they can self-destruct. Actually that would be the "improper conditions".

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.

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.

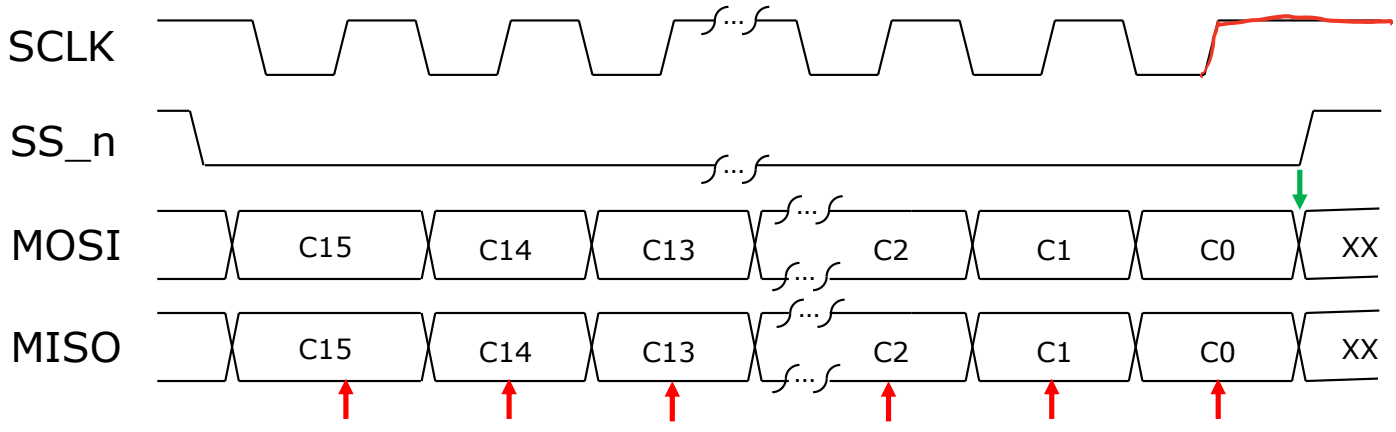
Turn in **nonoverlap.sv** along with a self-checking testbench (**nonoverlap_tb.sv**) and proof the test bench was run/debugged.

HW3 Problem 4 (**45pts**) What is SPI

- Simple 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

HW3 Problem4 (45pts) SPI Packets



A SPI packet involves a send and receive (full duplex) initiated by the master. Master controls SCLK, SS_n, and MOSI. The slave drives MISO if it is selected

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) changes **MISO** on the falling edge too. We sample **MISO** on the rising edge (see red arrows).

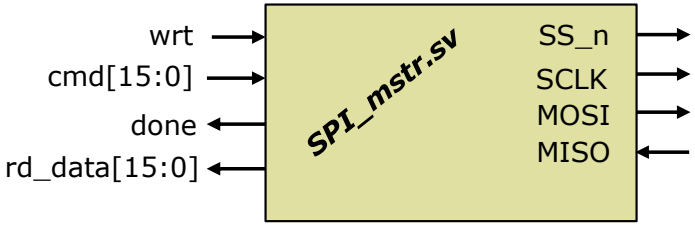
The sampled version of **MISO** in turn gets shifted into our 16-bit shift register. (on **SCLK** fall)

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.

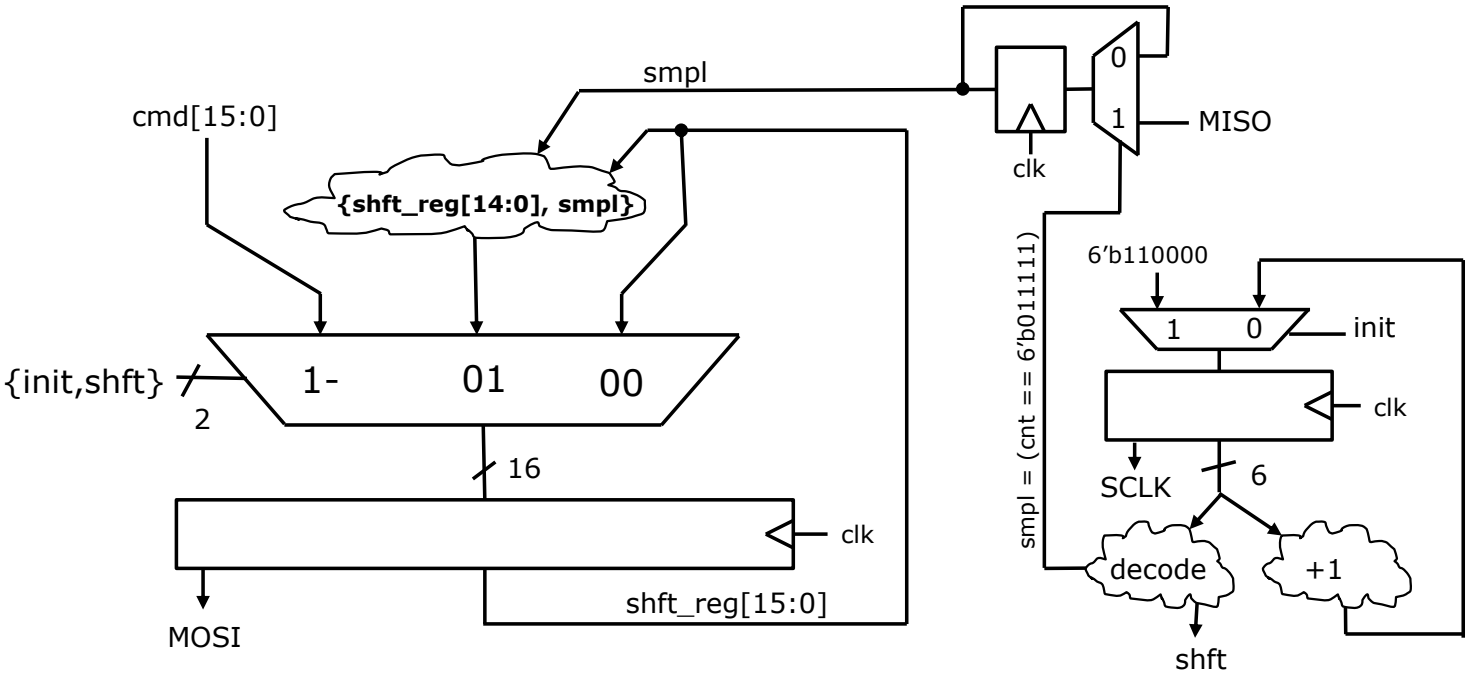
HW3 Prob4 (45pts) SPI Packets

- 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 thing 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, MISO	3-out 1-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]

HW3 Prob4 (45pts) 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.

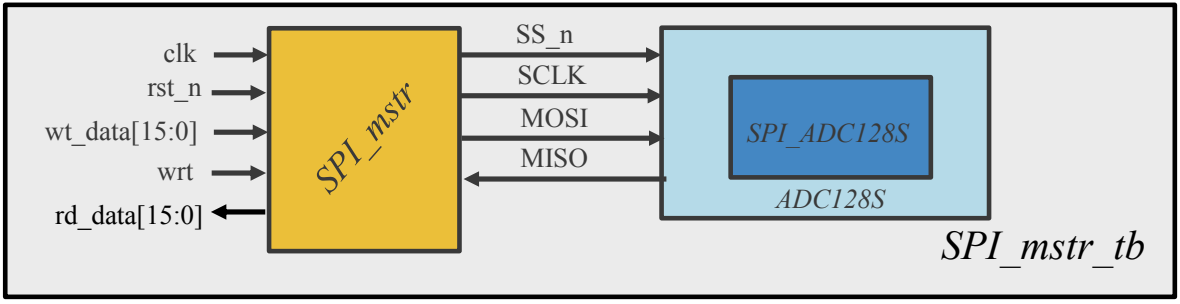
HW3 Prob4 (45pts) Architecture Hints

- SCLK Requirements:
  - SCLK will be 1/64 of our system clock (50MHz/64 = 781kHz)
  - SCLK is normally high and toggles during SPI transactions
  - Want a delay from start of transaction (SS_n fall) till first fall of SCLK
  - Look back 2-slides at the waveworms. We want a bit of a "back porch" on SCLK. A time in which it is high prior to SS_n returning high.

- Recommended SCLK implementation
  - SCLK comes from bit[5] of a 6-bit counter
  - This 6-bit counter is only counts during SPI transactions (otherwise loads 6'b110000)
  - The bits of this counter are not all preset or reset, but rather a combination such that SCLK is normally high and has its first negative edge a few system clocks after the transaction starts.
  - Perhaps will need to dedicate a state to creating the "back porch".

- Remember…for DUT Verilog (Verilog you intend to synthesize). If I see: ***always @(posedge*** … ← This next signal better be ***clk*** or I am going to blow a gasket.

HW3 Prob4 (45pts) Testing SPI_mstr.sv



- Create **SPI_mstr.sv** block
- Download **ADC128S.sv** (model of A2D converter on DE0-Nano, and a SPI slave)
- Also download **SPI_ADC128S.sv** (child of ADC128S.sv that you need)
- Create a testbench in which the **SPI_mstr.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.

- If you gave it 4 reads in a row:

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_mstr.sv (this is individual exercise, everyone submits their own)**
  - Your testbench (**SPI_mstr_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