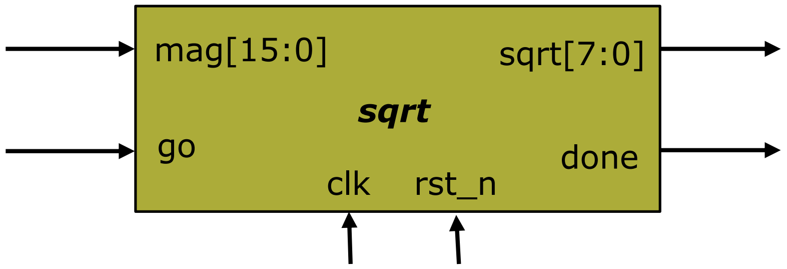
ECE 551
HW3

- Due Weds March 6th in class
- Work Individually
- Remember What You Learned From the Cummings SNUG paper
- Use descriptive signal names and comment your code

HW3 Problem 1 **(30pts)** SM Design

You do this one on your own

- Implement a square root function of an **unsigned** 16-bit number. (produces an 8-bit result)
- Consider an implementation that uses 8 successive multiplies.
  - Start with MSB of an 8-bit register set
  - Multiply this register by itself (0x80 x 0x80)
  - Is the product greater than the number we are taking the sqrt of (mag[15:0])?
    - If so clear the MSB of the register and set the next bit. (Now using 0x40)
    - If not keep the MSB set and set the next bit. (Now using 0xC0)
  - Repeat this process till you are testing if the LSB should be set or cleared



- Calculation starts when go is asserted.
- When calculation is complete done is asserted.

HW3 Problem 1 (continued) **(30pts)** SM Design

- You will need datapath (multiplier, and an 8-bit register for this)
- You will need a counter to know when you have tested all 8 bits. Consider using a 1-hot 8-bit register instead of a 3-bit encoded register. This register can now serve as your counter as well as a mask for what bit you are testing.
- You will need control signals for the above hardware, figure them out.
- You can code the datapath and SM in the same Verilog file if you wish
- Code the Verilog (**sqrt.sv**) well, using what you have learned (make Clifford proud)
- Code up a testbench (**sqrt_tb.v**) and test your unit

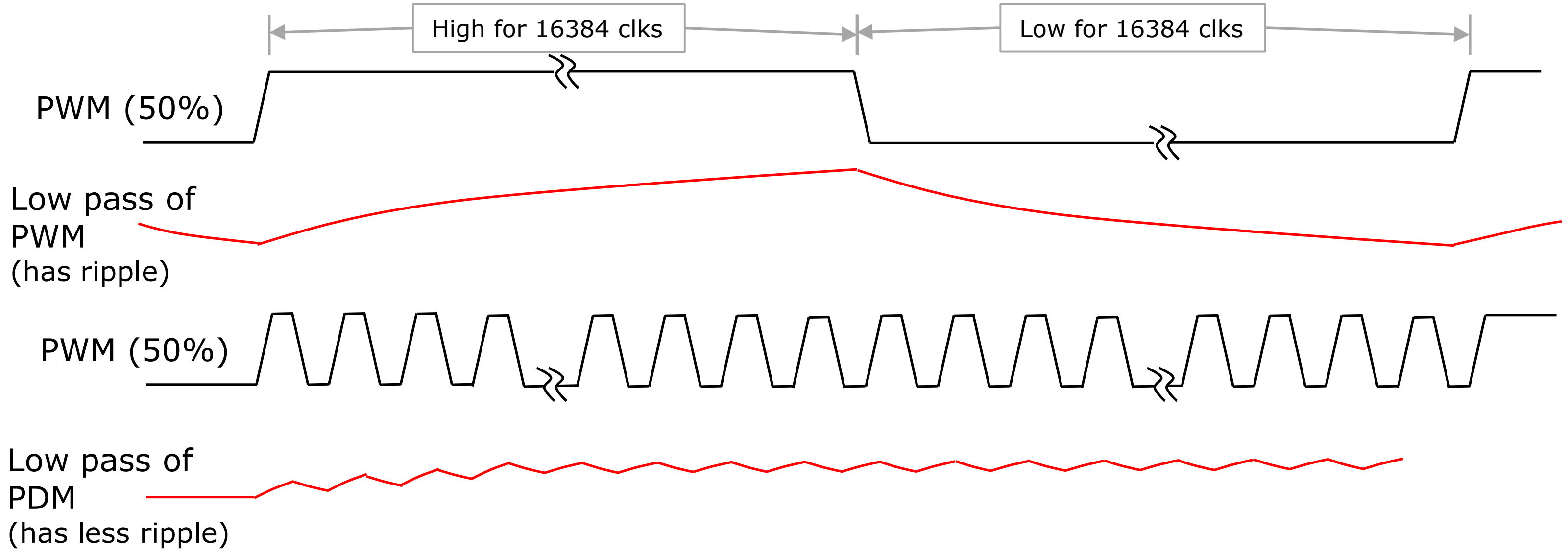
- Submit to the Dropbox for HW3
  - **sqrt.sv** (and sub files if any)
  - **sqrt_tb.v**
  - Proof that your unit ran correctly

  - **NOTE**: the human ear perceives the volume of sound proportional to the sqrt of the magnitude. This could be a potentially useful function for LED drive on the project.

HW3 Problem 2 (20pts) PDM

Done as in Class Exercise07 on Feb 22nd and Feb25th

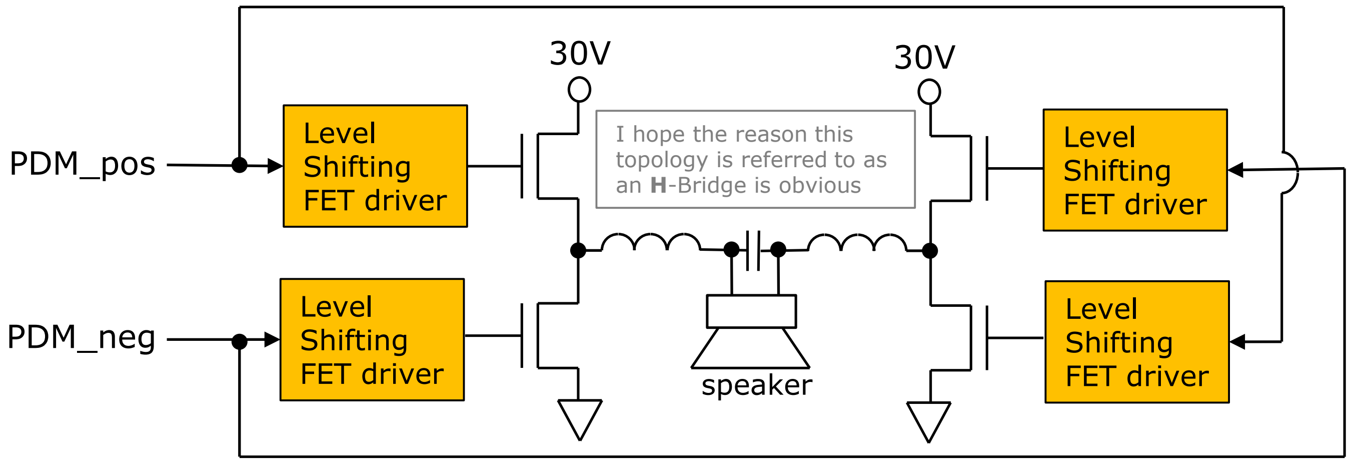
- To drive the speakers we eventually need an analog signal. So we need a form of Digital to Analog Conversion (DAC).
- There are many forms of DAC. One of the simplest is Pulse Width Modulation (PWM) followed by low pass filtering.
- A variation of PWM is PDM. PDM and PWM are conceptually similar, but PDM is easier to low pass filter than PWM.
- Consider a 15-bit PWM signal vs a 15-bit PDM signal. Both driving 50% duty cycle.


High for 16384 clks
Low for 16384 clks
PWM (50%)
Low pass of PWM (has ripple)
PWM (50%)
Low pass of PDM (has less ripple)

- As is shown above for a low pass filter with a given time constant, the PDM signal filters better and gives a lower ripple. This is an exaggerated view. The time constant of the LPF on the board will result in tiny ripple, and thus very little distortion in music quality.

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

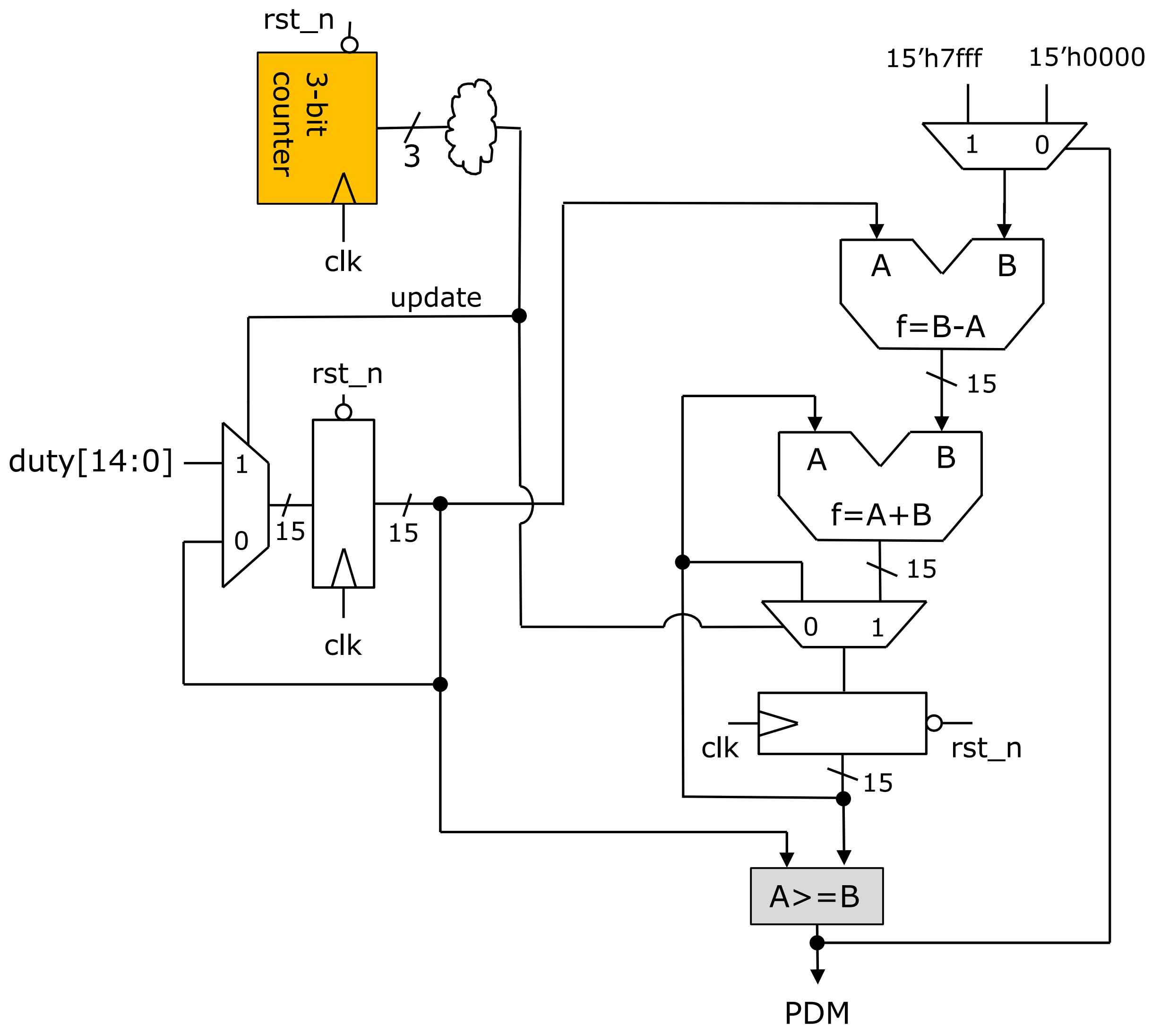
90Watts per channel…How do we do that?



30V
30V
PDM_pos
PDM_neg
Level Shifting FET driver
Level Shifting FET driver
Level Shifting FET driver
Level Shifting FET driver
I hope the reason this topology is referred to as an **H**-Bridge is obvious
speaker

- Being a child of the 80's I don't believe in headphones. If your neighbors aren't complaining you are not doing it right.
- The puny 0 to 3.3V PDM signals will be stepped up to 30V signals with 4A drive capability through the use of power MOSFETs in an H-Bridge topology. An LC circuit built across the H-Bridges forms the lowpass filter we need.
- The level shifting FET drivers are pretty neat chips, but they have limits. They cannot operate at the 50MHz of our main system clock. So our PDM signal cannot be operating at 50MHz. We will need to slow it down by a factor of 6 (6.25MHz).
- We will create a signal called **update** in our PDM block. This signal will be true when a 3-bit counter is full. Most all functions in our PDM block will only update when **update** is true. Thus we slow down the PDM to 6.25MHz without doing something stupid like running off a divided clock (see next page).

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



Signal:	Dir:	Description:
clk	in	50MHz system clk
rst_n	in	Asynch active low
duty[14:0]	in	Specifies duty cycle (unsigned 15-bit)
PDM	out	PDM signal out (glitch free)

Pretty much code what you see using mixture of dataflow and ***always*** blocks. Code it flat. No need to introduce hierarchy here.

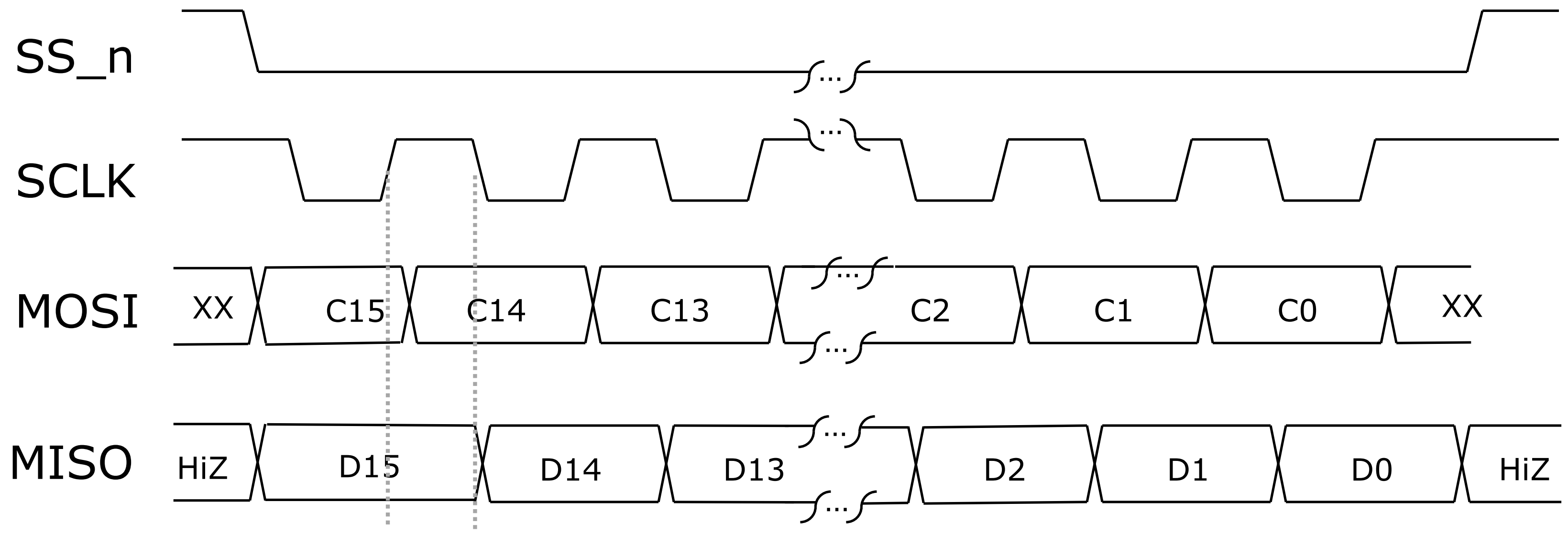
Create a testbench. Test it at multiple levels of **duty[14:0]** Theoretically you need to run for 32768*8 clocks at each setting. **Turn in: PDM.sv, PDM_tb.v** Proof that it ran and worked

HW3 Problem 3 (**50pts)** 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 Prob3 (50pts) SPI Packets



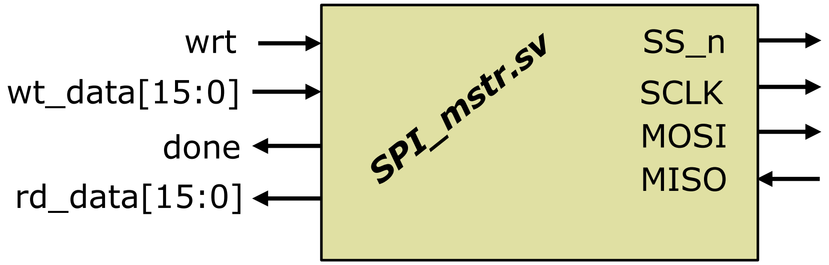
A SPI packet inherently involves a send and receive (full duplex). The full duplex packet is always initiated by the master. Master controls SCLK, SS_n, and MOSI. The slave drives MISO if it is selected. If the slave is not selected it should leave MISO high impedance. The A2D converter is the only SPI peripheral we have in the system.

The SPI master will have a 16-bit shift register. The MSB of this shift register is MOSI. MISO will feed into the LSB of this shift register. The shift register should shift **two system clocks after** the rise of SCLK, this eliminates any timing difficulties. The A2D on the DE0-Nano samples MOSI on the positive edge of SCLK, and changes MISO on the negative edge of SCLK. Of course all your flops are based purely on clk (system clock), not SCLK! SCLK is a signal output from your SPI master.

SCLK will be 1/16 of our system clock (50MHz/16 = 3.125MHz

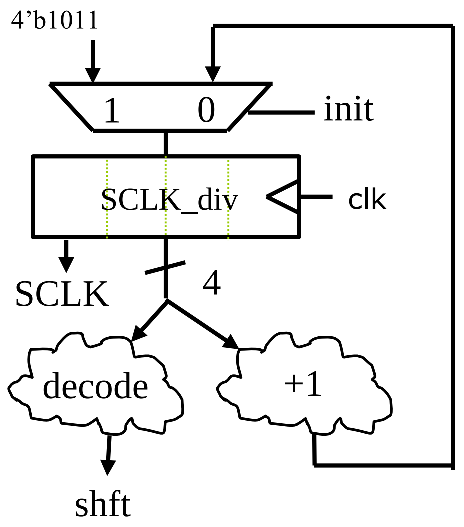
HW3 Prob3 (50pts) SPI Packets

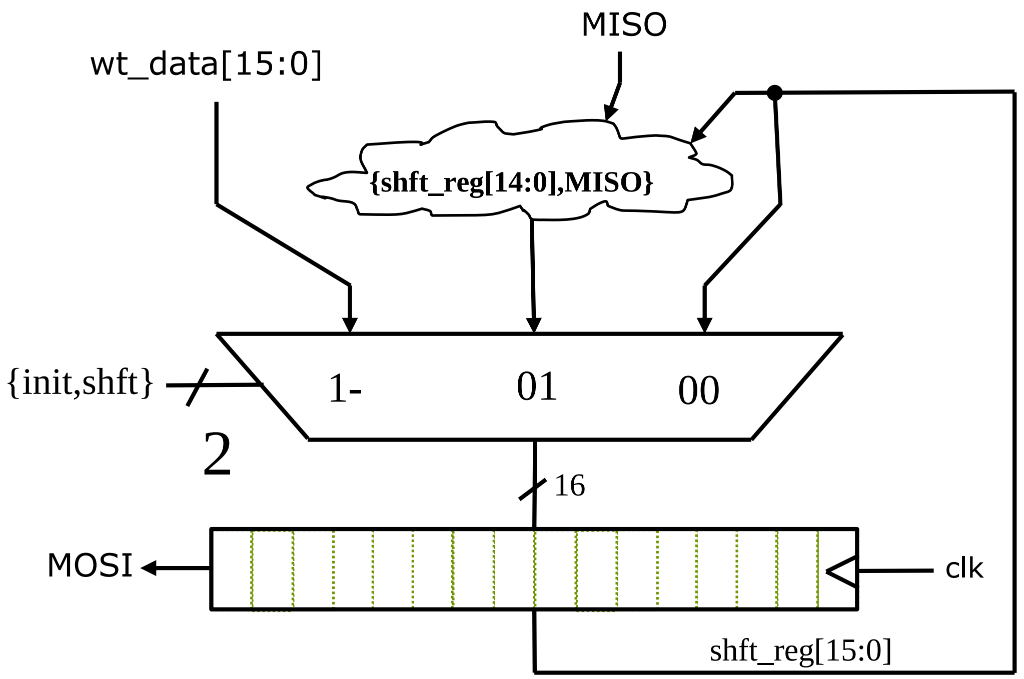
- You will implement **SPI_mstr.sv** with the interface shown.
- SCLK frequency will be 1/16 of the 50MHz clock (i.e. it comes from the MSB of a 4-bit counter running off clk)
- I had better not see any ***always*** blocks triggered directly on **SCLK**. We only use **clk** when inferring flops.
- Remember you are producing **SCLK** from the MSB of a 4-bit counter. So for example, when that 4-bit counter equals 4'b0111 you know **SCLK** rise happens on the next clk. Perhaps more pertinent…when that 4-bit counter equals 4'b1001 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
wrt	in	A high for 1 clock period would initiate a SPI transaction
wt_data[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 Prob3 (50pts) SPI Implementation:





The main datapath of the SPI master consists of a 16-bit shift register. The MSB of this shift register provides MOSI. The shift register can be parallel **load**ed with the data to send to the A2D, or it can left shifted one position taking MISO as the new LSB, or it can simply maintain.

Since the SPI master 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 (4'b1011). 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 4'b1011?

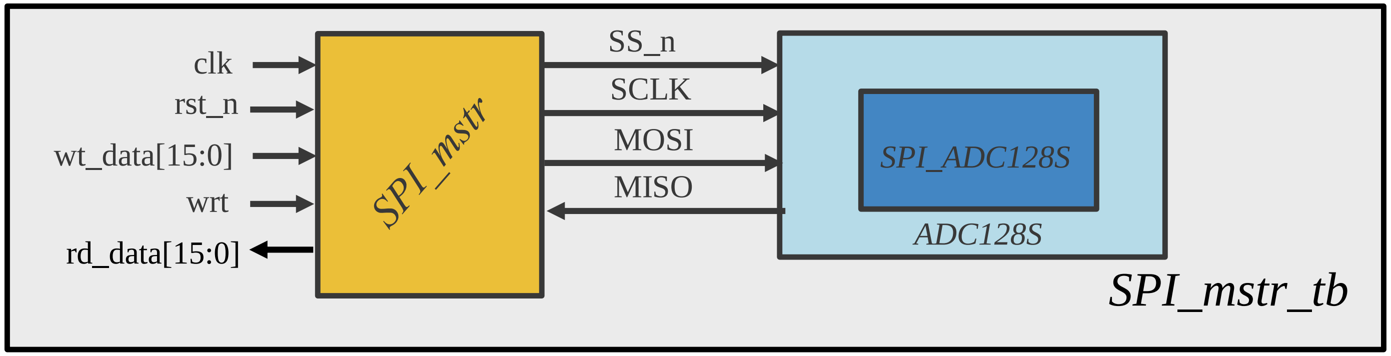
HW3 Prob3 (50pts) Architecture Hints

- SCLK Requirements:
  - SCLK will be 1/16 of our system clock (50MHz/16 = 3.125MHz)
  - 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[3] of a 4-bit counter
  - This 4-bit counter is only counts during SPI transactions (otherwise loads 4'b1011)
  - 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 Prob3 (50pts) 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 Prob3 (50pts) 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:
2	0xC00	You are requesting channel 2 for next time, but it returns channel 0 for first read.
2	0xC02	Has not decremented 0xC00 by 0x10 yet, but this is channel 2 from last request
3	0xBF2	Two reads have been performed so it decremented by 0x10, but this is still channel 2.
3	0xBF3	This is a channel 3 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