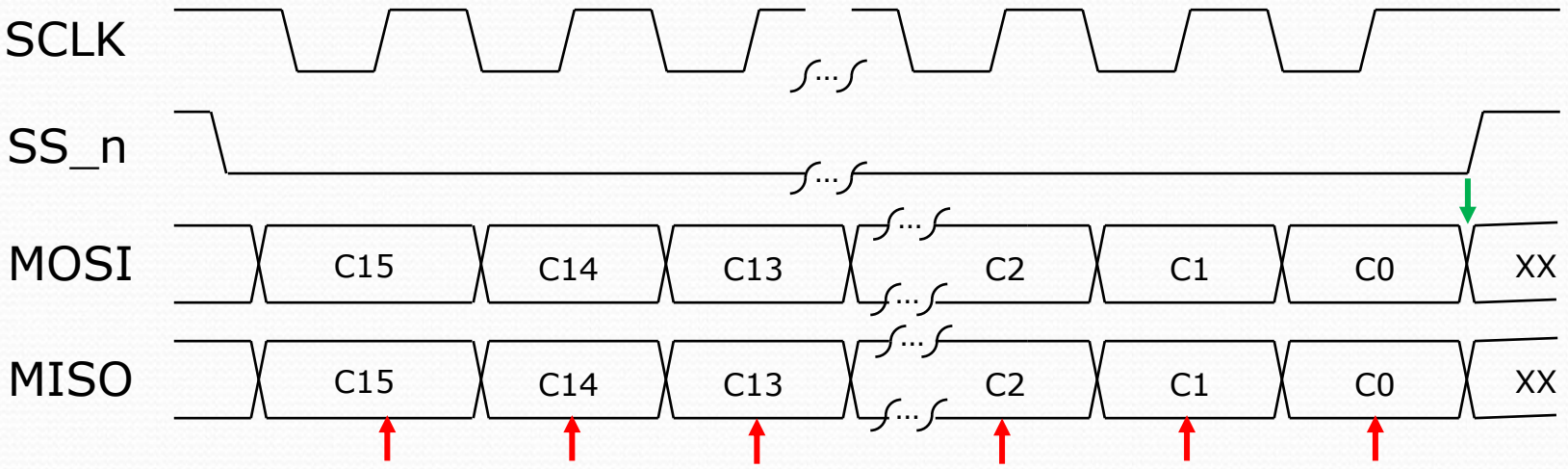
Exercise 17 SPI Tranceiver

- 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 on SCLK fall
    - ✓ MISO sampled on SCLK rise

Exercise 17: SPI (HW4 Problem5)

"A friend of mine took 551 last semester and did a SPI peripheral. I will just copy theirs". Nope…this one is specified quite different, but in subtle ways.



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

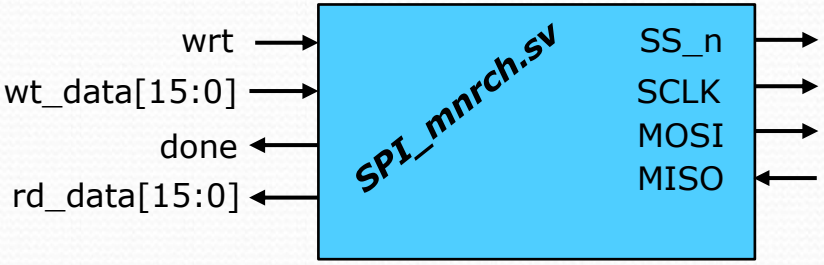
The sampled version of **MISO** in turn gets shifted into our 16-bit shift register. (on **SCLK** fall)

When **SS_n** first goes low there is a short 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 serf (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 (but prevent the fall of **SCLK**) (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 serf.

Exercise 17: SPI (HW4 Problem5)

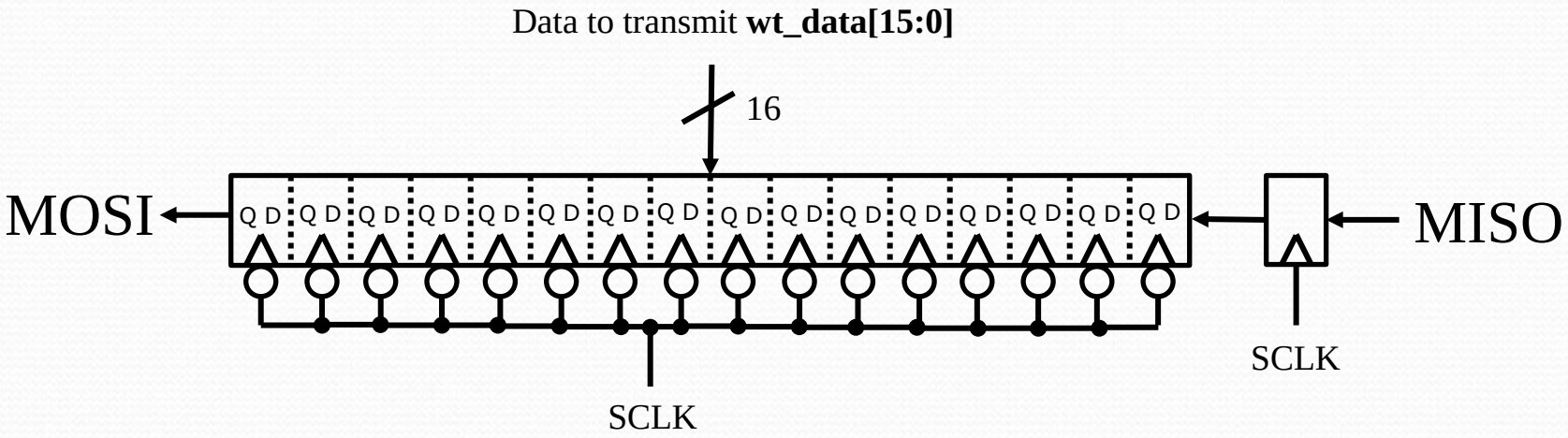
- You will implement **SPI_mnrch.sv** with the interface shown.
- SCLK frequency will be 1/32 of the 50MHz clock (i.e. it comes from the MSB of a 5-bit counter running off clk)
- I had better not see any ***always*** blocks triggered directly on **SCLK**. We only use **clk** when inferring flops.
- Remember you are producing **SCLK** from the MSB of a 5-bit counter. So for example, when that 6-bit counter equals 5'b01111 you know **SCLK** rise happens on the next clk *(sample MISO (see red arrow))*. Likewise when that 5-bit counter equals 5'b11111 you know SCLK fall happens next *(enable your 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
wt_data[15:0]	in	Data (command) being sent to inertial sensor.
done	out	Asserted when SPI transaction is complete. Should stay asserted till next **wrt**
rd_data[15:0]	out	Data from SPI serf. For inertial sensor we will only ever use [7:0]

Exercise 17: Cartoon Diagram!

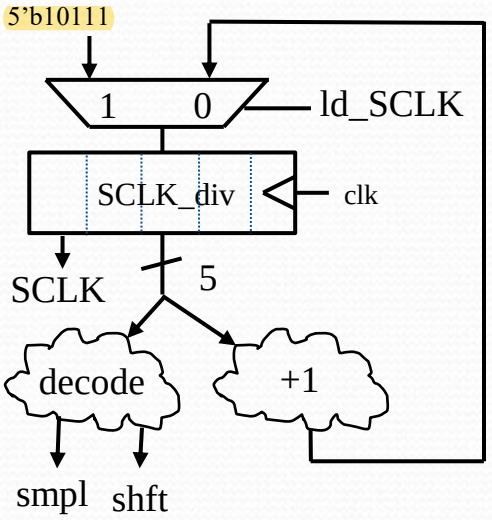
- Essentially, we need a 16-bit shift register that can parallel load data that we want to transmit, then shift it out (MSB first), at the same time it receives data from the serf in the LSB
- The bit coming from the serf (MISO) is sampled on the rise of SCLK, and then put into our shift register on the fall of SCLK.



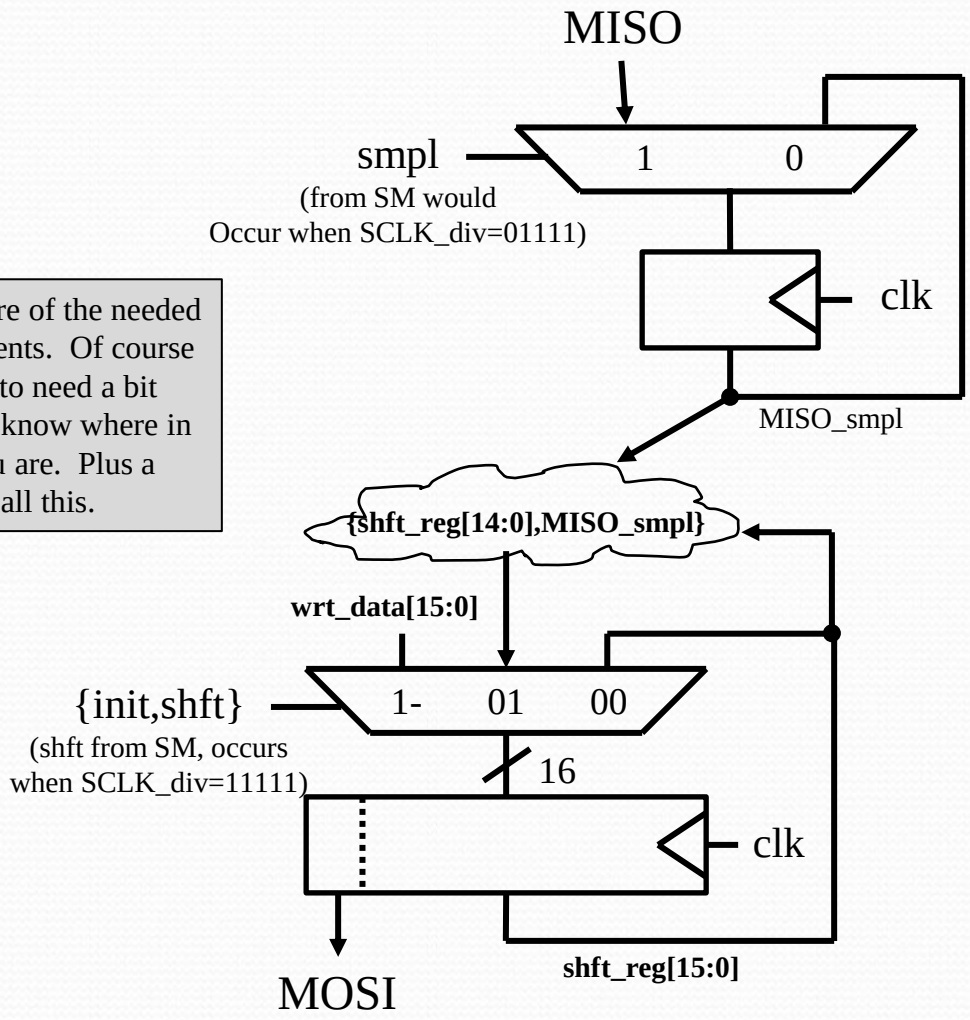
- Don’t make me regret drawing this diagram by actually implementing it this way. All our flops are always on **clk** rise, nothing else.

Exercise 17: SPI Tranceiver Hints:

"Heart of you SPI monarch is a 16-bit shift register. The MSB of this shift register forms MOSI. A version of MISO sampled at "SCLK rise" is shifted in as the LSB. The register is shifted during a SPI transaction at "SCLK fall".

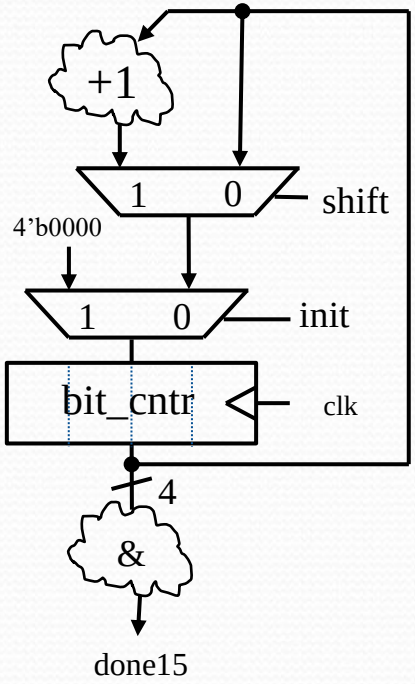


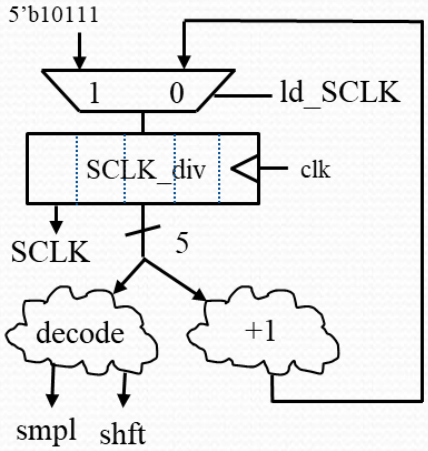
This is a picture of the needed datapath elements. Of course you are going to need a bit counter too to know where in the 16-bits you are. Plus a SM to control all this.

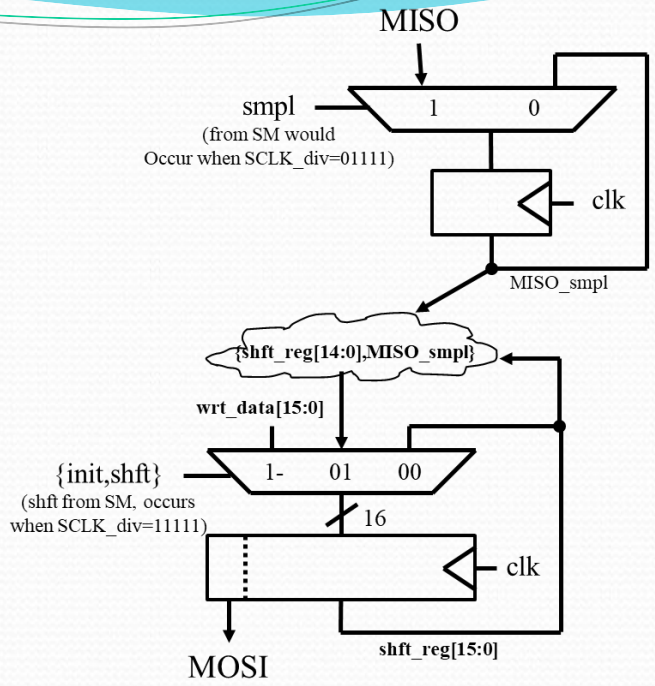


A synchronous reset of **SCLK_div** to a value like 10111 can help with the creation of a "front porch" (see previous slide). You are the monarch, you generate **SCLK**, so you know exactly when rise/falls are going to occur. If **sclk_div** = 01111 then a rise of **SCLK** is going to occur on the next **clk** edge.

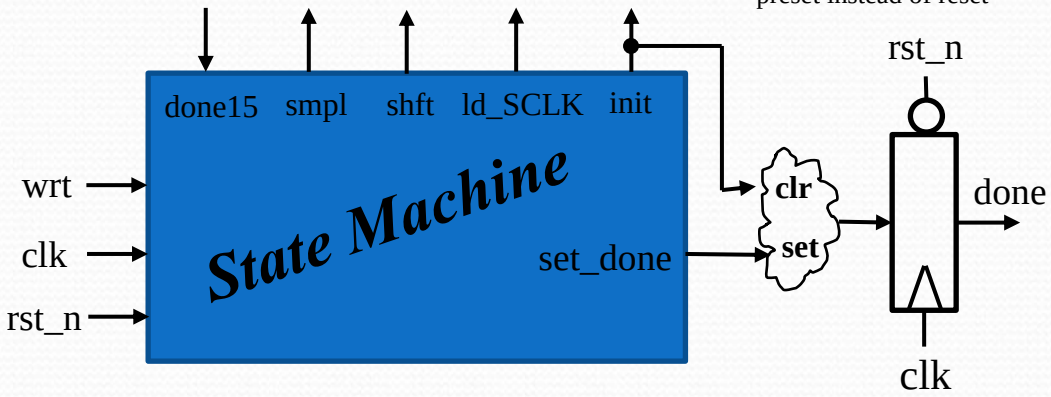
Exercise 17: SPI







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

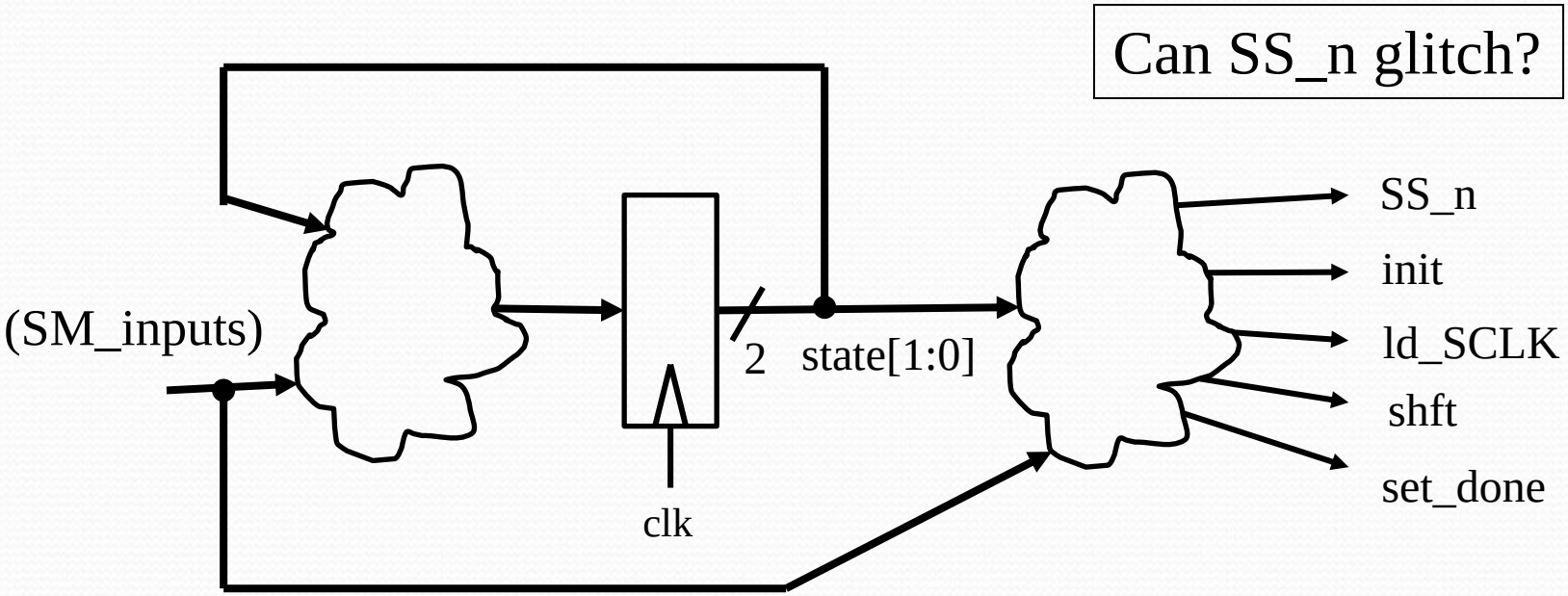
Exercise 17: SPI

- **SCLK** Requirements:
  - **SCLK** will be 1/32 of our system clock (50MHz/32 = 1.5625MHz)
  - **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 5-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.

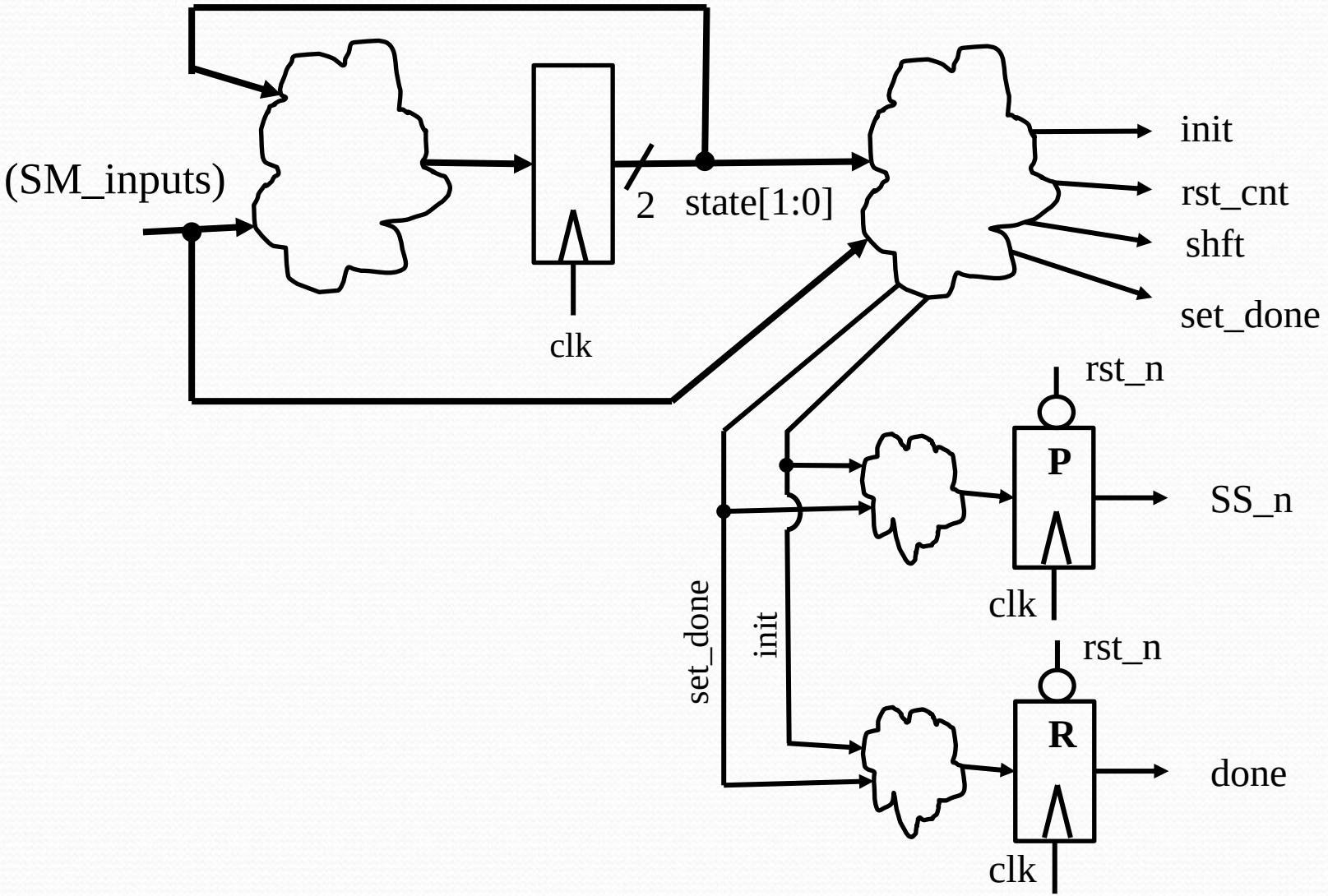
- Recommended **SCLK** implementation
  - **SCLK** comes from bit[4] of a 5-bit counter
  - This 5-bit counter is only counts during SPI transactions (otherwise loads 5'b10111)
  - The bits of this counter are not all preset or reset, but rather a combination such that **SCLK** is normally high and has 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!***

Possible SPI SM Implementation:



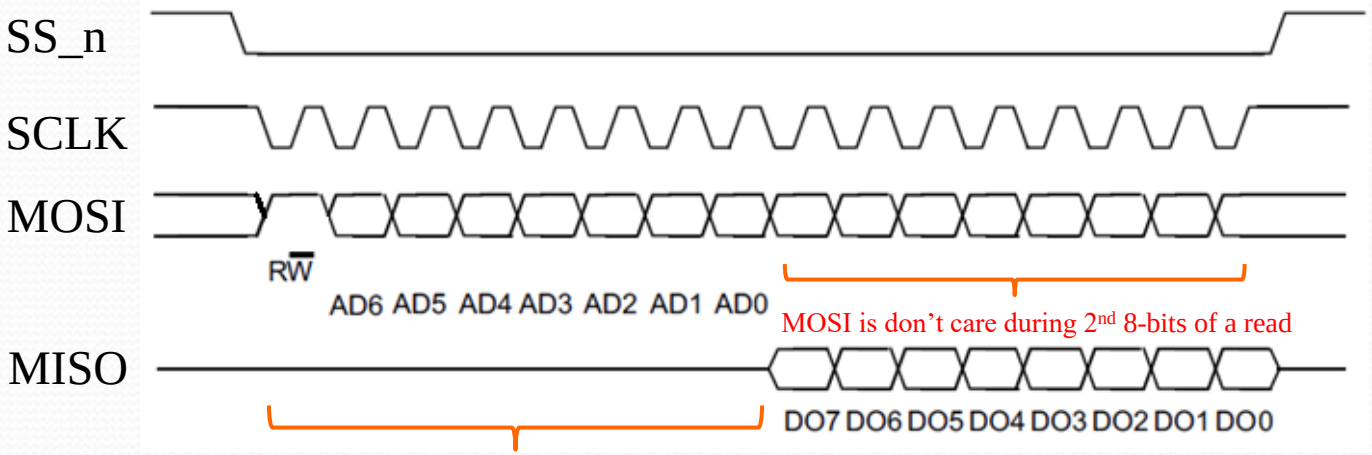
Correct SPI SM Implementation:



Exercise 17: Testing your SPI_mnrch.sv

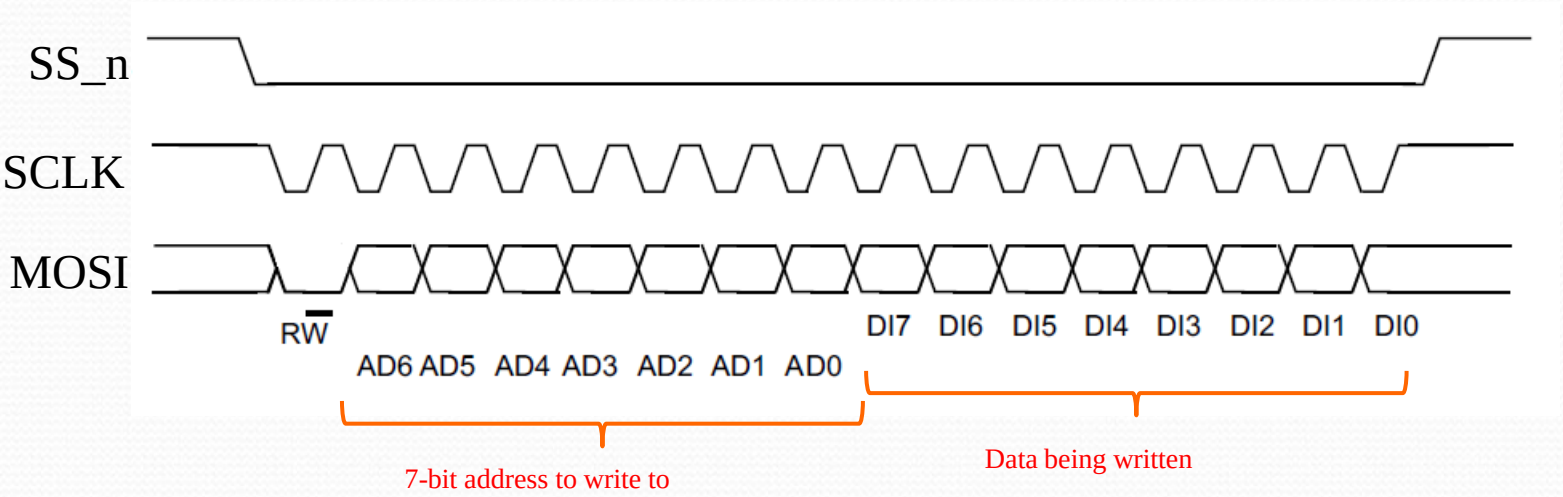
- The inertial Sensor (Nemo family from ST micro) is configured and read via a SPI interface
  - One can think of a single 16-bit SPI transaction to the Nemo as two 8-bit transactions.
  - For the first 8-bits of the SPI transaction, the sensor is looking at MOSI to see what register is being read/written. The MSB is a R/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)

如果是讀取，則請求暫存器中的資料將在 SPI 事務的第 2 個 8 位元期間傳回 MISO



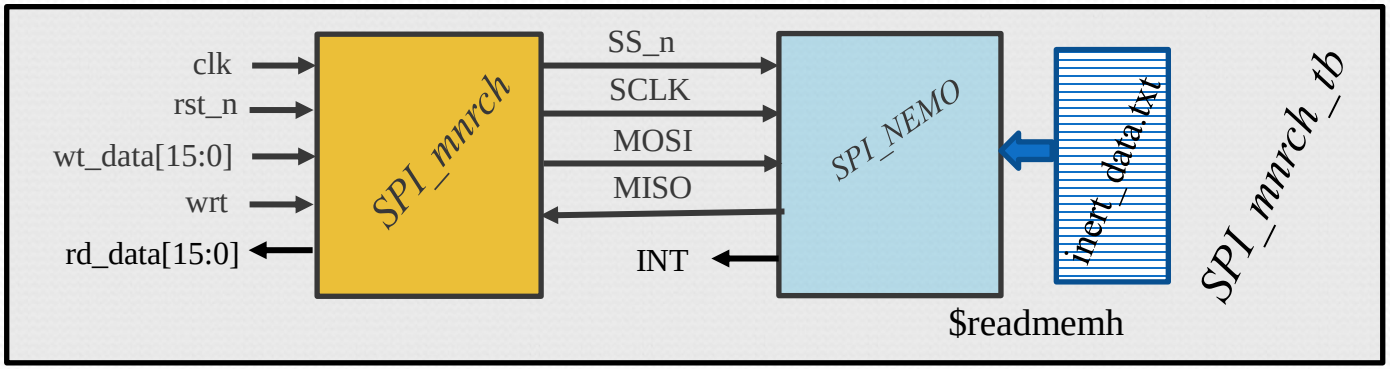
Exercise 17: Testing your SPI_mnrch.sv

- 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.

Exercise 17: Testing your SPI_mnrch.sv



- Create **SPI_mnrch.sv** block
- Download **SPI_NEMO.sv** (model of ST's Nemo inertial sensor) from Canvas page
- Create a testbench (***SPI_mnrch_tb***) in which the ***SPI_mnrch*** drives the ***SPI_NEMO***. Test and debug.
- The NEMO contains a "WHO_AM_I" read only register at address 0x0F. This register always returns the value 0x6A. So your first test should be a read from 0x0F. This would mean **wt_data** = 16'h8Fxx. When the transaction is finished you would expect **rd_data** to contain 16'hxx6A.
- Next you should write to the register to configure the **INT** output pin to assert when new data is ready. This is done by writing 0x02 to the register at address 0x0D. **wt_data** = 16'h0D02. You would not expect any particular response on **rd_data**, but a signal internal to SPI_NEMO called **nemo_setup** should go high.

Exercise 17: Testing your SPI_mnrch.sv

- If NEMO_setup has been asserted you should eventually see the **INT** pin assert. At this time you could read inertial data from either of the YAW registers.

Addr/Data: wt_data	Description:
16'h8Fxx	Who Am I register (returns 0x6A)
16'hA6xx	yawL ➔ yaw rate low from gyro
16'hA7xx	yawH ➔ yaw rate high from gyro

Reading registers from NEMO

Addr/Data: wt_data	Description:
16'h0D02	INT config register

Writing register to NEMO

- **INT** pin is cleared when ***yawL*** is read.
- Look at the contents of the file: ***inert_data.hex***. It has a header row telling you the data for AX,AY,…roll,yaw
- If the first register you read after **INT** was asserted was ***yawL*** then you would should get a result of 16'hxx8d and the **INT** pin should drop.
- Create a self-checking testbench (**SPI_mnrch_tb.sv**) using this information. You should probably at least wait for a 2nd assertertion of **INT** and read a few registers from the 2nd line of data in ***inert_data.hex***.
- Submit as much as you have done by the end of the 2nd day. This has to be finished for HW4.