APPLIED HOMEWORK #3

Sequential Circuits

1 Description and Homework Objectives

In this Applied Homework, you will create many components that you will later use in AHW4 to implement a Manchester serial receiver. The common theme in the components made in this AHW is flops (registers). You will create a lot of components in this AHW and it will get a little tedious, but most components are simple variations of each other. At your demo, you will use a provided top-level file to allow you to test all of these circuits together.

After this homework, you should be able to:
1. Design sequential circuits for implementation in an FPGA
2. Use simulation to verify sequential circuit functionality
3. Use an FPGA prototyping system to verify sequential circuit funtionality

2 Applied Homework Tasks

You may work with a partner in your lab section on this assignment, or alone, as arranged during your first Applied Homework demo. If you work with a partner, we expect you to work together on the complete homework rather than sub-divide the tasks. Both partners are expected to thoroughly understand the work that is submitted. Specifically, each person should be able to, during the demo, demonstrate the circuits, explain how they work, and answer questions about them.

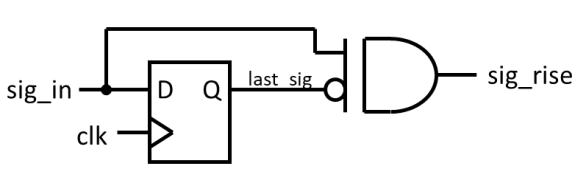
Before starting the work, download and unzip the AHW3 files from the class website—you will need these. Remember to save all your files to your CAE network storage so that you can access them later (including at the demo!).

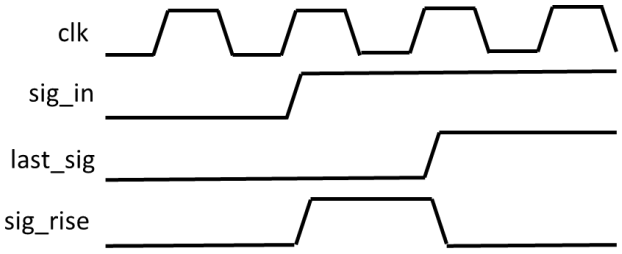
2.1 Copy Files from AHW2

In AHW2 you completed **segB.bdf**, **segE.bdf**, **bcd7seg.bdf, hex2seg,** and **bin2bcd** designs. You will need these again for AHW3 demo so copy them forward into the AHW3 demo directory.

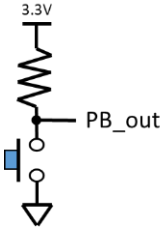
2.2 Edge Detection.

If we wanted to make a circuit that could detect a positive edge of an input signal, and produce and output that was high for one clock cycle how could we do that? The simplest way is to flop the input signal. The output of this flop represents the "last" value (older value) of the input. If the signal is currently high, but the "last" value was low then the signal must be having a rising edge. Study the diagram below for further understanding.





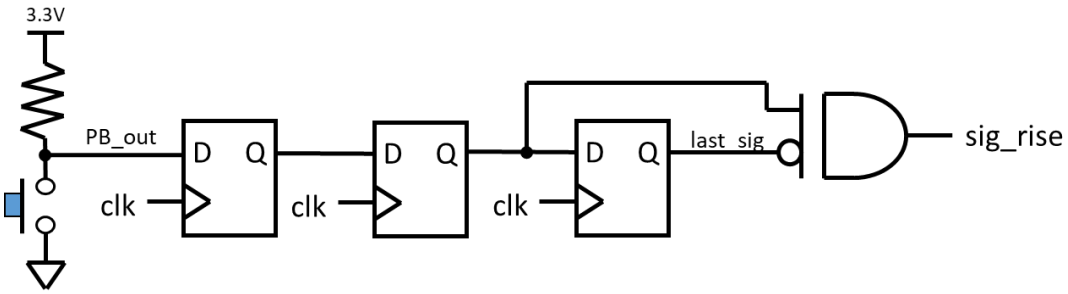
We wish to have a rising edge detector connected to a push button on the board. A typical push button circuit is formed by a switch to ground with a pull-up resistor.



When you push the button **PB_out** gets shorted to ground. There may be some "bounce" in the signal due to the mechanical diaphragm in the switch bouncing for a few ms after contact. When you release the button **PB_out** gets pulled up to 3.3V. How does the fall or rise of **PB_out** relate to your clock (**clk**)?

That's right…it is completely unrelated to your clock.

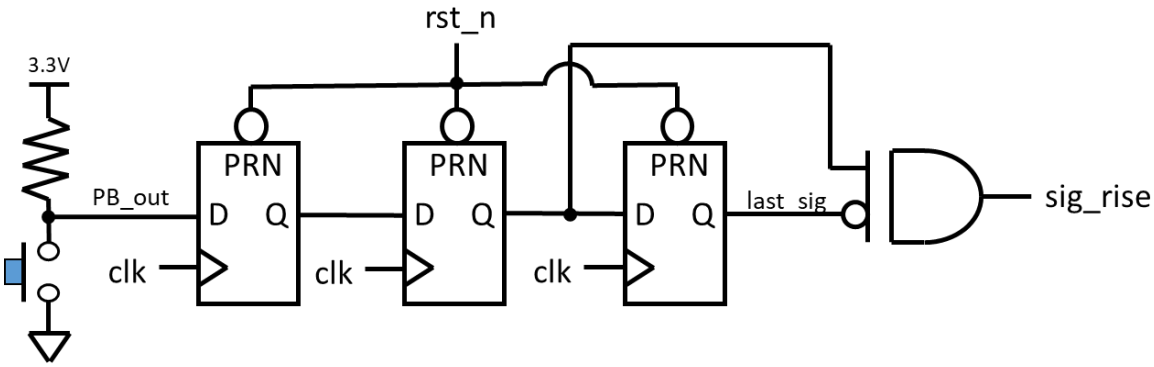
What do we know about using a signal completely unrelated to our clock? We have to put it through a synchronizer (two back to back flops) first to get rid of any possible meta-stability and to synchronize it to our clock domain. Given that our full positive edge detector circuit needs to look like:



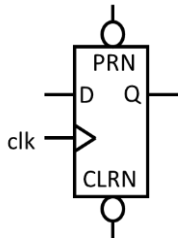
However, the above circuit is not quite right yet. What is the state of the 3 flops right after power up, but before any positive clock edges? We know the flops will power up to either a 0 or a 1, but we don't know which if we don't have an asynch reset/set signal.

Simulate it in your head if the 3 flops had power up to all hold 0's. The PB_out signal is pulled high. So a 1 starts "marching" down the flops. When the 2nd flop flops in a 1 it will look like a rising edge has occurred, but the user never pressed the button yet.

If you have a push button circuit that is normally pulled high you need to preset your flops, not reset them. So the edge detector circuit needs to look like:



In the **AHW3.zip** file you will find a basic flop cell called **d_ff.sv**. All flops and registers you make in this AHW will be from an instantiation(s) of this basic flop. This flop is a positive edge triggered flop with an asynch active low clear (**CLRN**), and an asynch active low preset (**PRN**).



This is our basic building block flop. If the reset (CLRN) or preset (PRN) functionality is not needed for a specific application then you simply tie them off inactive. What does it mean to "tie it off inactive"….well they are active low inputs, so you would tie them to a 1'b1. You will use 3 instantiations of this flop to build an edge detector.

You will find a file in the **AHW3.zip** file called **rise_edge_detect.sv**. Flesh out the missing Verilog to implement a rising edge detector. There is a self checking test bench (**rise_edge_detect_tb.sv**). Launch ModelSim, compile, and debug till you pass the test bench. Capture all the waves at the DUT level for the length of the simulation (**rise_edge_detect_sim.jpg**). Also capture the "YAHOO! test passed" message from the transcript window (**rise_edge_detect_yahoo.jpg**) Submit **rise_edge_detect.sv**, **rise_edge_detect_sim.jpg**, and **rise_edge_detect_yahoo.jpg** to the dropbox for AHW3.

2.3 D Flip-Flop with Enable and Only asynch Reset (no preset)

There are a number of applications where we do not want to allow a set of flops to update on every clock, but rather just when a signal from a state machine is high. Take for example a case in which a state machine is counting how many times something has occurred. The state machine would generate an enable signal to the counter, so it would only increment when the even occurred, not every clock cycle.

We will use our base flop (**d_ff.sv**) to create a flop with an enable signal (**d_en_ff.sv**).

With the creation of the edge detector for the push button you just saw an example of using an asynch **pre**set. However, that is quite rare, and the vast majority of the time your flop will just need a reset. So when we create **d_en_ff.sv** we will also tie the **PRN** signal of **d_ff** inactive and not have it as a signal at the **d_en_ff** level of hierarchy.

How do we create an enable flop? A basic flop (like **d_ff**) clocks in a new **D** input every positive edge of clock. We do not try to stop it from doing this. We just make it flop in the same value. So an enable flop is made by placing a 2:1 mux at the **D** input of a basic flop. If the **EN** signal is high a new value from the "outside world" is present at the **D** input of the basic flop. If **EN** is low then the current value of the flop is fed back to become the **D** input (it recirculates its current value). Sketch out this verbal description on a piece of paper…make sense?

You will find a file called **d_en_ff.sv** in the .zip. Flesh out the missing Verilog to implement a flop with enable. There is a self checking test bench (**d_en_ff_tb.sv**) available. You know the routine…get it working. Capture all the waves at the DUT level for the length of the simulation (**d_en_ff_sim.jpg**). Also capture the "YAHOO! test passed" message from the transcript window (**d_en_ff_yahoo.jpg**) Submit **d_en_ff.sv**, **d_en_ff_sim.jpg** and **d_en_ff_yahoo.jpg** to the dropbox for AHW3.

2.4 per_capture (9-bit register with enable)

The next several registers you implement are all needed for the Manchester serial receiver you will build in AHW4. One such register captures the period of the incoming serial signal. It is a simple 9-bit register that needs no reset. It simply captures an incoming **period** when instructed to do so by the signal **capture.**

Refer back to the video on structural/dataflow Verilog. Prof. Lipasti showed how to instantiate several instances of a cell in a single line (vectored instantiation). You also can look back at your AHW1 solution because you did this as well there for **RCA4_smarter.sv**. In all the design that follow in this AHW we expect you to use vectored instantiation. If we see 9 individual instantiations for this solution there will be a significant penalty.

You will find a file called **per_capture.sv** in the .zip. Flesh out the missing Verilog to implement the design. There is a self checking test bench (**per_capture_tb.sv**) available. You know the routine…get it working. Capture all the waves at the DUT level for the length of the simulation (**per_capture_sim.jpg**). Also capture the "YAHOO! test passed" message from the transcript window (**per_capture_yahoo.jpg**) Submit **per_capture.sv**, **per_capture_sim.jpg** and **per_capture_yahoo.jpg** to the dropbox for AHW3.

2.5 per_cnt (9-bit counter with synch reset)

This is the counter that will be counting the period of the serial bits. It requires a synchronous clr (setting of register to all zeros through the D input). **Asynch**ronous resets are typically controlled by a global resest signal the resets the state of the whole chip. When a state machine needs to reset (clear) a register via a control signal it is producing, that happens through a **synch**ronous reset (or clear).

In this class we focus quite a bit on a ripple carry adder using a chain of full adder cells. That is one of those things instructors do so students know what "goes on under the hood". In practice most of us digital designers realize the synthesis CAD tools are way better at designing arithmetic blocks than we are. So we just use a dataflow statement with a + sign when we need to infer an adder or an incrementor. You are allowed to do this for this implementation.

You will find a file called **per_cnt.sv** in the .zip. Flesh out the missing Verilog to implement the design. There is a self checking test bench (**per_cnt_tb.sv**) available. You know the routine…get it working. Capture all the waves at the DUT level for the length of the simulation (**per_cnt_sim.jpg**). Also capture the "YAHOO! test passed" message from the transcript window (**per_cnt_yahoo.jpg**) Submit **per_cnt.sv**,**per_cnt_sim.jpg** and **per_cnt_yahoo.jpg** to the dropbox for AHW3.

2.6 8-bit Shift Register (shft_reg)

There are countless different serial protocols (UART, SPI, I2C, I2S, Manchester, …). But pretty much all serial blocks no matter the protocol will employ a shift register to serialize the bits (if it is a transmitter) or coalesce the bits into a word (if it is a receiver). We are making a receiver of a protocol that transfers 8-bit words with the most significant bit going out first. Think about it…you have a stream of 8 bits coming in with the most significant bit first. You would want to shift the bits into the LSB of an 8-bit shift register that is shifting left. After 8 shifts the bit that came in first (the most significant bit) would be in the MSB position of your 8-bit shift register.

We will only want our shift register to left shift when the state machine tells it to via a signal called shft. We will use enable flops (d_en_ff) to achieve this.

You can use dataflow with vector concatenation to form the input 8-bit vector to your 8-bit enabled register. To left shift you throw away **shft_reg[7]**, promote **shft_reg[6:0]** to the 7:1 positions, and tack on **shft_in** into the **shft_reg[0]** spot.

Does the shift register need a reset? The contents of **shft_reg** form the output of the Manchester receiver. We know the contents of a flops are X (unknown) at power up. So until we have shifted in a full 8-bits the output of our Manchester receiver contains unknown bits. Is this OK?

The answer is yes, it is OK to have unknown output. The reason is the state machine will have a valid signal that tells whoever is looking at the output of the Manchester receiver that the output is valid. As long as this valid signal is held low after reset the actual output of the Manchester receiver can be unknown (no one should be looking at it).

You will find a file called **shft_reg.sv** in the .zip. Flesh out the missing Verilog to implement the design. There is a self checking test bench (**shft_reg_tb.sv**) available. You know the routine…get it working. Capture all the waves at the DUT level for the length of the simulation (**shft_reg_sim.jpg**). Also capture the "YAHOO! test passed" message from the transcript window (**shft_reg_yahoo.jpg**) Submit **shft_reg.sv**, **shft_reg_sim.jpg** and **shft_reg_yahoo.jpg** to the dropbox for AHW3.

2.7 bit_cnt (3-bit counter with synch clear & enable)

As mentioned the Manchester serial protocol we are dealing with transfers/receives 8-bit words. So it should be quite obvious that the state machine is going to need a 3-bit counter (a counter that can count 8 things) to keep track of which bit in the transfer it is currently working on.

The state machine will have the ability to synchronously clear this count via a signal called **clr**.

Of course this counter cannot be counting every clock because the protocol is not sending one bit per clock. So the counter has to have an enable so it only increments when instructed to by the signal **inc**.

Are you sick of this yet?….yaa I bet, but you are getting close. This register design and one more and you're done.

You will find a file called **bit_cnt.sv** in the .zip. Flesh out the missing Verilog to implement the design. There is a self checking test bench (**bit_cnt_tb.sv**) available. You know the routine…get it working. Capture all the waves at the DUT level for the length of the simulation (**bit_cnt_sim.jpg**). Also capture the "YAHOO! test passed" message from the transcript window (**bit_cnt_yahoo.jpg**) Submit **bit_cnt.sv**, **bit_cnt_sim.jpg** and **bit_cnt_yahoo.jpg** to the dropbox for AHW3.

2.8 state5_reg (5-bit state flops for a 1 hot SM with 5 states)

State machine flops always need an asynch reset. We have seen that a lot of the registers used in the datapath of the Manchester receiver do not need a reset. This is often (but not always) true for datapath registers. But control registers (things involved in the "brains" of the machine like state machines) always need to be reset by the global reset (which is best done as an asynch reset).

For ease of implementation when doing state machine designs by hand we use one-hot state machines. We will do that here as well. As we will see in AHW4 we need 5 states for our Manchester receiver SM so we will need a 5-bit state register.

Recall the reset then will have to reset the upper 4-bits, but preset the lowest bit.

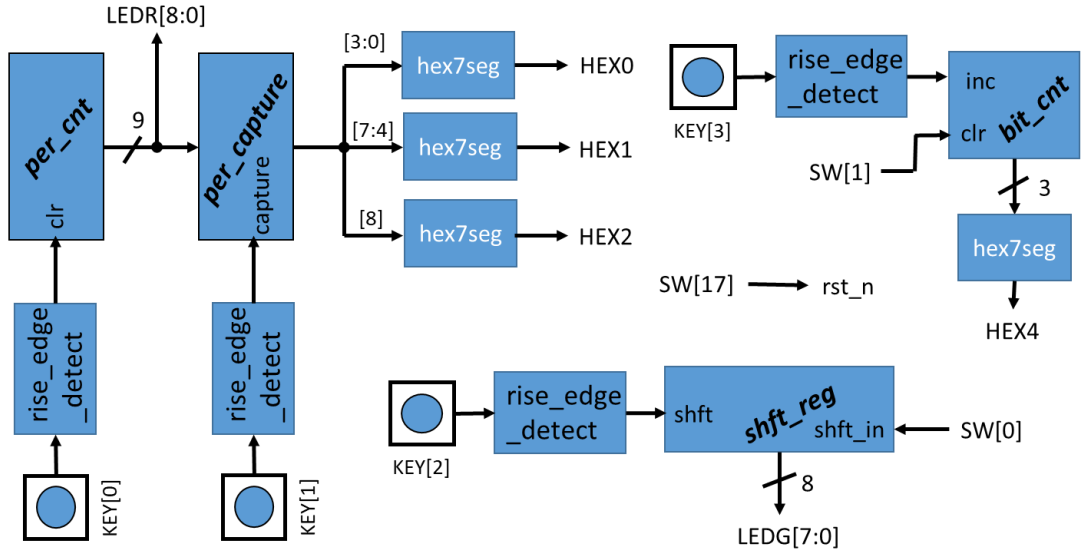
We still want you to implement this as a vectored instantiation of 5 **d_ff** cells (one line instantiation). To get the reset and preset signals correct you will have to make a couple of 5-bit reset/preset vectors.

You will find a file called **state5_reg.sv** in the .zip. Flesh out the missing Verilog to implement the design. There is a self checking test bench (**state5_reg_tb.sv**) available. You know the routine…get it working. Capture all the waves at the DUT level for the length of the simulation (**state5_reg_sim.jpg**). Also capture the "YAHOO! test passed" message from the transcript window (**state5_reg_yahoo.jpg**) Submit **state5_reg.sv**, **state5_reg_sim.jpg** and **state5_reg_yahoo.jpg** to the dropbox for AHW3.

3 Demo Tasks

DO NOT wait until your demo to read this!

During your Applied Homework Demo, you will compile AHW3_top.sv in Quartus and map it to the DE2 FPGA board. The mapping of our various register designs is as follows:



NOTE: The state5_reg block is not tested as part of the demo. It will be tested in AHW4 when we implement the state machine and assemble the complete Manchester receiver.

4 Applied Homework Submission

If working with a partner, your group should submit only <u>one</u> report, and both group members will receive the same credit for the report component of the Applied Homework. The grade for the demo component may differ based on demonstrated knowledge and understanding of the submitted work.

You will upload a number of files to the AHW3 dropbox, be sure that your files are named correctly.

- **rise_edge_detect.sv** – Verilog file of rising edge detector
- **rise_edge_detect_sim.jpg** – image of waveforms for simulation of rise_edge_detect.
- **rise_edge_detect_yahoo.jpg** – capture of transcript window
- **d_en_ff.sv** – Verilog file of d FF with enable

- **d_en_ff_sim.jpg** – Image of waveforms for simulation of d_en_ff
- **d_en_ff_yahoo.jpg** – capture of transcript window
- **per_capture.sv** – Verilog file of period capture register
- **per_capture_sim.jpg** – Image of waveforms for simulation of period capture
- **per_capture_yahoo.jpg** – capture of transcript window
- **per_cnt.sv** – Verilog file of period counter
- **per_cnt_sim.jpg** – Image of waveforms for simulation of period counter
- **per_cnt_yahoo.jpg** – capture of transcript window
- **shft_reg.sv** – Verilog file of shift register
- **shft_reg_sim.jpg** – Image of waveforms for simulation of shift register
- **shift_reg_yahoo.jpg** – capture of transcript window
- **bit_cnt.sv** – Verilog file of period bit counter
- **bit_cnt_sim.jpg** – Image of waveforms for simulation of bit counter
- **bit_cnt_yahoo**.jpg – capture of transcript window
- **state5_reg.sv** – Verilog file of state register
- **state5_reg_sim.jpg** – Image of waveforms for simulation of state register
- **state5_reg_yahoo.jpg** – capture of transcript window

Be prepared to answer questions about the design during the demo. Some sample questions follow:

1. What was the purpose of the back to back flops in the edge detector? Why are the flops preset instead of reset?
2. Why do some registers need a reset and some don't?
3. What is the purpose of the shift register in a serial receiver?
4. Why do all of LEDR[8:0] look like they are on?

The deadline for your report appears online on the course webpage.