UF Herbert Wertheim College of Engineering
Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

Dr. Christophe Bobda
Omar Elbasri, Alexander Aziz
EEL3701C - Fall 2025

Lab 3: Sequential Logic and Counters

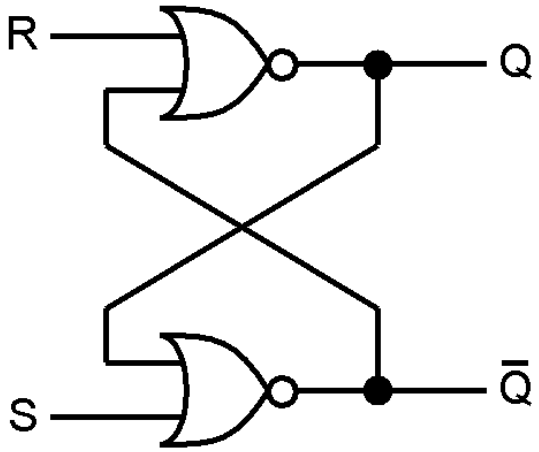
Overview

Previously, we only considered the application of combinational circuits, where the output is solely a function of the present input. Sequential circuits introduce the concept of memory, where not only do functions rely on current inputs, they rely on past information as well. Sequential Logic is used to construct state machines, more formally known as finite state machines (FSM).
In this lab, you will implement a counter, which is a type of finite state machine. A counter counts through a sequence of numbers in some order, potentially having a reset, or inputs to go forwards and backwards. The main difference between a counter and a finite state machine in this course is that the counters will generally output their state, while a finite state machine will have more logic associated with it. Finite state machines will be explored more in Lab 5.

Please watch all the videos in the appendix related to Quartus before asking questions about it.

Pre-Lab Procedure: Part 1 - The SR Latch

The SR (Set-Reset) Latch is one of the most basic memory elements in digital circuits. By using a feedback loop, we can connect gates together to create a type of memory storage. The Set (S) signal will set the output to true (1). Even if S turns on and off, the output will remain true. The Reset (R) signal will reset the output to false (0). Similarly, even if R turns on and off, the output will remain false.



UF Herbert Wertheim College of Engineering
Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

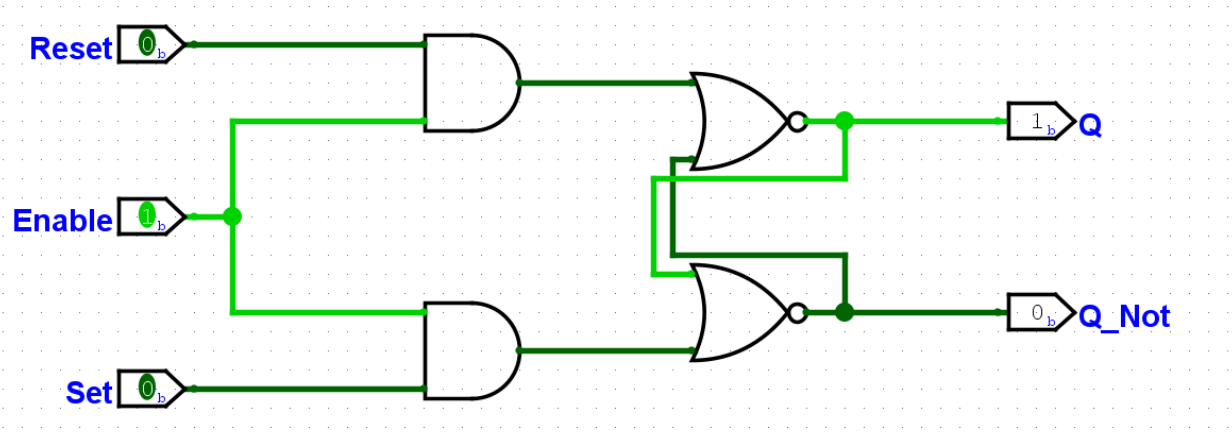
Dr. Christophe Bobda
Omar Elbasri, Alexander Aziz
EEL3701C - Fall 2025

Figure of a NOR SR Latch. It can also be made of NAND gates, which inverts the inputs.

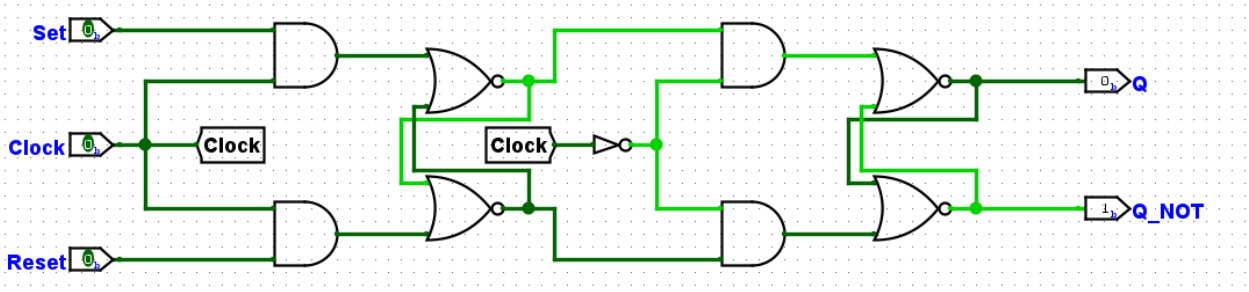
Create an SR Latch in **Quartus**, rather than Logisim, and include screenshots in your lab document. Simulate the behavior of the SR latch and use it to answer the pre-lab questions. Screenshots of this simulation are **not** necessary. Look in the appendix for a short tutorial on Quartus and waveform simulations.

Latches are great, but they have a weakness. Latches are **asynchronous** components, meaning that they update immediately when inputs change. When wanting to store information, like counters and FSMs do, you need to be confident that the data won't suddenly change. A component becomes **synchronous** by adding a **clock** signal, ensuring that data changes at predictable intervals.

By adding an **enable** that allows set and reset to change only when it is true, we're one step closer to making our component synchronous. This enable signal allows us to control when the inputs of the SR-latch will change. By combining two of these gated SR-latches together, we can make a **flip-flop** that only responds on the falling edge of a clock signal. This flip-flop is now synchronous, as it only functions when the clock pulses.



A gated SR-latch consists of a NOR SR-Latch and AND gates connected to a clock signal.



UF Herbert Wertheim College of Engineering
Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

Dr. Christophe Bobda
Omar Elbasri, Alexander Aziz
EEL3701C - Fall 2025

An SR flip-flop is a synchronous component that only changes on the rising edge of a clock.

From the design shown above, create an SR flip-flop in Quartus and submit a screenshot of your design. Simulate the behavior of the SR flip-flop and use it to answer the pre-lab questions. Screenshots of this simulation are **not** necessary.

Part 1 Summary:

1. Submit a screenshot of the SR Latch in Quartus. Do **not** submit screenshots of this simulation.
2. Submit a screenshot of the SR flip-flop in Quartus. Do **not** submit screenshots of this simulation.
3. Answer the following pre-lab questions based on your simulations of the latch and flip-flop.

Pre-Lab Questions: Part 1

1. When implementing an SR latch, what happens when both S=1 and R=1?
2. What is the purpose of the Clk signal?
3. What is the difference between a SR flip-flop and a SR latch?
4. Put a NOT gate in front of the S input of the SR flip-flop, then connect the input of that gate to the R input of the flip-flop so that S and R are always at opposite logic levels. What is the relationship between this single input and the output of the flip-flop?
5. Now do the same thing as in the previous question, except put a NOT gate in front of the R input. What is the relationship between the single input and the output of the flip-flop?

Pre-Lab Procedure: Part 2 - Building a State Machine

In the previous section we discussed how we can use a flip-flop to store the value of a circuit over a fixed time interval. **Sequential logic**, which involves circuits with flip-flops, differs from combinational logic because the outputs are not only reliant on the inputs passed into the circuit, but information contained within the flip-flops. This information inside the flip-flops of a sequential circuit is known as **state**. We can store our current state, and control how to transition to the next state with combinational logic. State bits are typically denoted by Q, and each state bit requires its own flip-flop.

In this section you are going to create a two-bit counter that counts in the following sequence:

1 -> 3 -> 2 -> 1 -> 3 -> 2 -> 1 -> …

The counter will also have an input GO(H) that when true, the counter should count. If GO(H) is false, the counter should hold its current value and not proceed.

When designing a state machine, we use a next-state truth table (NSTT) to show what state our circuit will move to based on the current state and other inputs. The next-state truth table takes in the current state as an input, and uses that in addition to any other inputs that might be present to determine the next state. Design a NSTT for the above-mentioned counter in counting-order, Use don't-cares to eliminate any unused signals, and for any undefined states. The NSTT should be comprehensive, containing columns for all of your inputs, as well as columns representing each state bit in both their current state and next-state. Note that RESET(L) is not included in the truth table because it is an asynchronous input, and not related to the logic.

Your NSTT should have the following column format:

Q1	Q0	GO(H)	Q1+	Q0+

For readability, you **must** color the NSTT gray every other row. **If you do not do this, points will be deducted.** These NSTTs should **not** be on paper.

Part 2 Counter Specifications:

- The count order should be 1 -> 3 -> 2 -> 1 -> … , where the 3 dots indicate repetition. Extending the count, it should go 1 -> 3 -> 2 -> 1 -> 3 -> 2 -> 1 …
- The counter increments only while GO(H) is true; otherwise it holds its current value.
- An additional asynchronous reset signal RESET(L) resets the counter to the number 1.
    - Note the (L) next to the name of the RESET signal. This indicates that the input is **active-low**, so the counter must reset when the input is **low**, rather than high.
    - Hint: The CLRN flip-flop input is an active low reset that makes the flip-flop go to 0, and the PRN flip-flop input is an active low reset that makes the flip-flop go to 1. Unused CLRN and PRN inputs should be tied to VCC.
- The most significant state bit must use a JK flip-flop.
- All other state bits must use D flip-flops.

The next step is to create combinational logic that represents our NSTT to be used in a circuit. While our NSTT demonstrates our desired behavior, we need to create equations that will represent this table. Furthermore, since we will be implementing our equations with logic gates, we will need to simplify our equations. *Generate MSOP or MPOS equations using K-maps for each flip-flop input.* Based on the presence of a D-FF and a JK-FF, your flip-flop inputs should be D0, J1, and K1. Use the excitation tables in the appendix of the D-FF and JK-FF to determine the necessary inputs to transform Q1 and Q0 into Q1+ and Q0+.

Now that we have equations we can begin designing our circuit. This counter will only be simulated, so do not worry about pin assignments for the DE10. To ensure the functionality of all features, simulate the counter in Quartus then use an image editing software to annotate the simulation. This simulation should show all features implemented in the counter, including resetting, going forward, and not going forward. It should be annotated such that it is clear what is happening on each clock edge and why it is happening.

Part 2 Summary:

1. Submit a comprehensive Next-State Truth Table (NSTT) for the counter.
2. Submit the MSOP or MPOS equations using K-maps for each flip-flop input. You **must** use don't cares in your k-maps to simplify equations. Both the MSOP and MPOS are acceptable for each input.
3. Submit a screenshot of the complete circuit in a Quartus .bdf file
4. Submit an annotated screenshot of the simulation of the circuit. The annotations should list the state of the counter after every clock edge, the relevant inputs for the next state, and any other information deemed necessary. The simulation should show **every feature** implemented for this counter.

Pre-Lab Questions: Part 2

1. What are don't-care entries in an NSTT and why are they useful when simplifying logic?
2. Why do we need flip-flops to build a counter? In other words, why can't we build a counter with only combinational logic?
3. Explain the difference between D flip-flops, SR flip-flops, JK flip-flops, and T flip-flops by describing, for each, the inputs it uses and how its output (Q) changes on a clock edge?

Pre-Lab Procedure: Part 3 - 3-Bit Counter

The previous sections had you implement a counter where the output would be based on the current state bits (Q). There are some scenarios where the desired output is different from the state representation. We can incorporate additional logic into our circuit to transform the state bits into the desired output format.

Consider the scenario where we only need 3 state bits for a counter but want to output a 4 bit number. By adding output logic, we can provide the necessary interface. The counter will also be able to count both forward and backwards depending on the input F(H).

Create a 3 bit counter that counts in the following sequence when F(H) is true:
4 -> 5 -> A -> B -> 2 -> 4 -> 5 -> …

The counter will count this sequence when F(H) is false:
4 -> 2 -> B -> A -> 5 -> 4 -> 2 -> …

These should be output to your DE10 on the 7-segment displays. Use either a button or switch (if one doesn't work, try the other) on the DE10 as your CLK signal.

Part 3 Counter Specifications:

- The count order should be 4 -> 5 -> A -> B -> 2 -> 4 -> … when going forward, and 4 -> 2 -> B -> A -> 5 -> 4 -> … when going backwards.
- An additional asynchronous reset signal RESET(L) resets the counter to the number 4.
    - The advice from the last section about the RESET(L) applies here as well.
- The most significant state bit (Q2) should use a JK flip-flop, the least significant (Q0) should use a D flip-flop, and the middle state bit (Q1) should use a T flip-flop. Derive the inputs necessary for each flip-flop by making k-maps based on Qn, Qn+, and F(H).
- Add an input F(H) that when false reverses the counter order to be 4 -> 2 -> B -> A -> 5 -> (repeat from the left)
- The outputs are named Y3, Y2, Y1, and Y0, and they should be determined based on the state bits. There is **no requirement** for what sequence of bits should correspond to what state, so feel free to do whatever is most convenient.
    - However, a table or other note stating which state equals which output is required.

Your NSTT should have the following column format:

Q2	Q1	Q0	F(H)	Q2+	Q1+	Q0+	Y3	Y2	Y1	Y0

For readability, you **must** color the NSTT gray every other row. **If you do not do this, points will be deducted.** These NSTTs should **not** be on paper.

Your DE10 should display Y3–Y0 on one of the seven-segment displays, using one of the DE10 buttons as the clock signal, and two switches for F(H) and RESET(L). A hex to seven-segment decoder component is provided to you in the appendix, as well as instructions on how to use it.

You will demo this circuit in lab, so you are not required to simulate it in Quartus. It is important to have the demo working before coming to your lab section.

Part 3 Summary:

1. Submit a comprehensive Next-State Truth Table (NSTT) for the counter (see the above required column format of the table). Label which state corresponds to which output, either on the side of the table, or as an additional note.

2. Submit the equations for the flip-flop inputs and the Y outputs. You **must** use don't cares in your K-maps to simplify equations.
3. A screenshot of the complete circuit in a Quartus .bdf file, including the component for the hex to seven-segment display.

In-Lab Procedure

1. You will demo your circuit working from **part 3** to your PI. **(30)**
2. Complete the quiz in-lab as specified by your PI. **(20)**

Submission Files

- lab3.qar (Quartus project archive file)
- lab3.pdf

Appendix Videos

Quartus project creation and simulations: https://www.youtube.com/watch?v=zs6NURBmJz0
DE10-Lite programming with Quartus: https://www.youtube.com/watch?v=o-EbrZn47zs
Component importing in Quartus: https://youtu.be/0JIe6lBd__I
Quartus Tutorial playlist:
https://www.youtube.com/playlist?list=PLHa2Bqz_II4sp_dW3sm-3SwNqeZqvGmGu
Ssgdecoder.vhd file: https://eel3701.ece.ufl.edu/assets/misc/ssgdecoder.vhd

Appendix A

Flip-flop Excitation Tables

JK Flip Flop			
J	**K**	**Q**	**Q+**
0	X	0	0
1	X	0	1
X	0	1	1
X	1	1	0

T Flip Flop		
T	**Q**	**Q+**
0	0	0
0	1	1
1	0	1
1	1	0

D Flip Flop		
D	**Q**	**Q+**
0	0	0
0	1	0
1	0	1
1	1	1

X = Don't care, regardless of value output will be what is in the table.
These tables represent the behavior of flip-flops on a clock edge. J, K, T, and D represent the inputs to a JK-FF, T-FF, and a D-FF respectively. Q represents the output of the FF before the clock edge, and Q+ represents the output of the FF after the clock edge.

Appendix B

Adding the provided 7 segment decoder Component to a Quartus .bdf
In Lab 2 you built a 7-segment decoder in Logisim to drive the DE10 hex display. In this lab we are moving to Quartus and providing a ready-made VHDL hex decoder for you to use in this and future labs. VHDL is a Hardware Description Language (HDL) that we will cover a bit later in the semester. For now all you need to know is that HDLs are used to describe circuits using text, rather than visually. Follow these steps to add the VHDL file (.vhd) as a Block Symbol (.bsf stands for block symbol file) in a Quartus Block Diagram (.bdf).

1. Download the provided ssgdecoder.vhd file and place it in your Quartus project folder.
2. After setting up a Quartus project click on "Project" and select "Add/Remove Files in Project"
3. Click on the three dots to open up file explorer and navigate to where you downloaded ssgdecoder.vhd and select it.
4. Once you see that the file is set up to be added to the project select "Ok" and the window should close.
5. Now in Project Navigator select Files and double click ssgdecoder.vhd so that it opens up in the Quartus text editor.
6. To make the block symbol file (.bsf) component to add to a .bdf file click on File>Create / Update>Create Symbol Files for Current File. This will create a .bsf file in your project directory.
7. Now open the .bdf in Quartus and add a Symbol like any of the other basic gate types.
8. Since the .bsf we created should be in the Project directory (if you followed the previous instructions) you can expand the project folder and find the ssgdecoder symbol to add to your .bdf.

Video Tutorial: https://youtu.be/0JIe6lBd__I

Appendix C

How to Simulate Designs in Quartus
Before downloading your design to the DE10 board, you want to be sure it behaves correctly. Simulation lets you test your circuit with different inputs and view its outputs over time without using real hardware. It's faster, safer, and makes debugging easier. As your projects grow larger, Quartus takes longer to synthesize, fit, and route the design to the FPGA. Doing a quick

functional simulation first means you can catch mistakes early without waiting through long compilation and routing steps every time you make a change.

1. Run Analysis and Synthesis to make sure your design doesn't have any syntactic errors.
2. Click File>New.
3. Select “University Program VWF” and click “OK” which should now show the Simulation Waveform Editor window
4. Click Edit>Insert>Insert Node or Bus.
5. Click Node Finder in the dialog box.
6. Set the filter to “Pins: All” or “Signals: All”.
7. Click List to display all available signals (they are called Nodes here).
8. Select the input and output signals you want to simulate.
    a. Click the > arrow to add them to the waveform.
    b. Click the >> arrow if you would like to add all listed signals to the waveform
9. Click OK to close Node Finder.
10. Click OK again to insert the signals into the waveform window.
11. Click and drag to highlight signals on the input rows, then set logic values to ‘1’ or ‘0’ by clicking the 1 or 0 buttons at the top, or by right-clicking the selected interval and forcing it high or low.
12. Click Simulation > Run Functional Simulation (or press the play button).
13. You will be prompted to save the waveform file in your project folder (e.g., Waveform.vwf). Save it.
14. View the output waveforms below the input signals.
15. Zoom in/out to examine timing and verify correctness.