

**Department of Electrical Engineering**

**Lab Report 5: Asynchronous Up Counter**

Name: Socheath Sok, ID: 014470701

Partner: Ashley Tran

Group: 11

Professor: Dr. Ray Wang

Class: EE 301

Date: November 15, 2017

**Abstract**

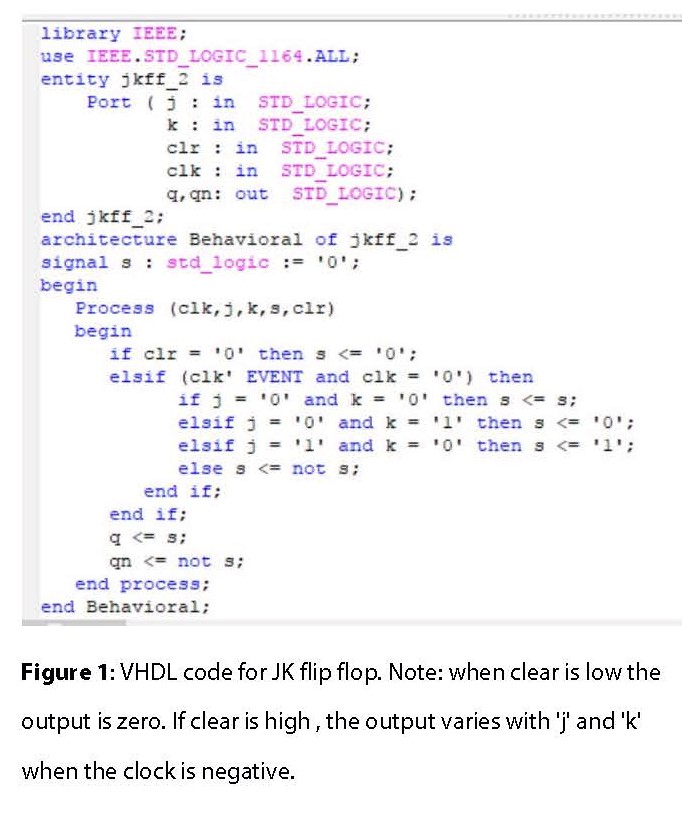
The goal of this lab is to design a mod-16 asynchronous ripple up counter by using the J-K flip-flops as the component. As part of the design, we had to decide whether the JK flip flop is positive or negative edge triggered. Doing so will determine how the four flipflops needed to make the mod-16 counter should be connected. Unlike previous labs, since the clock is, now, part of the design, a few new commands will be included as part of the codes and the constraint file so that the clock will produce the appropriate output and can be controlled manually on the board during the implementation. After we done with the up counter, some adjustments were made to the code to create an asynchronous down counter and the results will be shown in the extra credit section.

**Introduction**

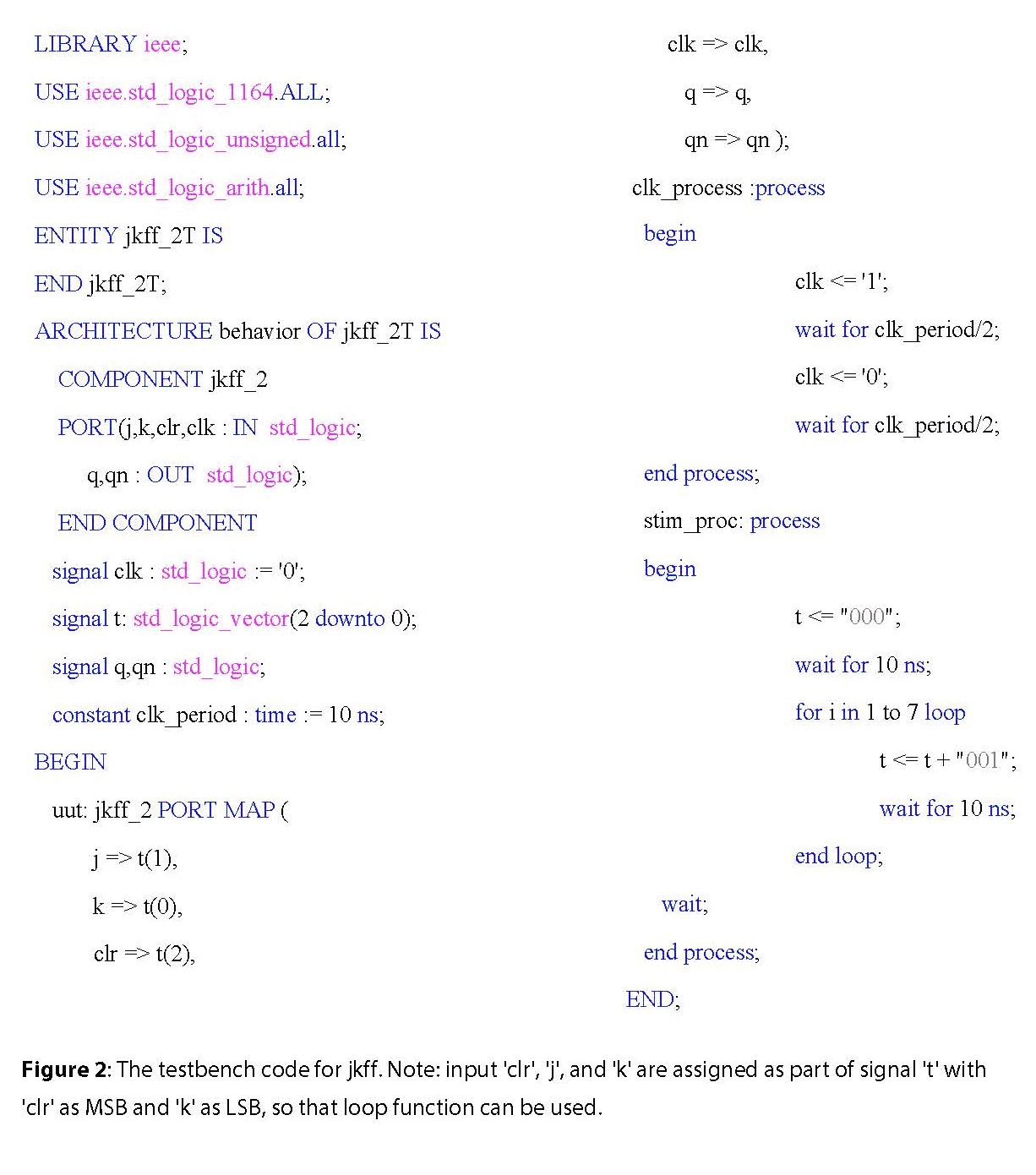
For the J-K flip flop, there are 4 inputs (j, k , clr, and clk) and 2 outputs (q and qn). Using the ‘if’ and ‘elseif’ statements, we defined how the outputs should be as clear, ‘j’, ‘k’ and the clock changes. We represented the output ‘q’ and ‘qn’ with the signal ‘s’ so that it can be used many times as both the input and the output. Furthermore, signal ‘s’ is defined as ‘0’ to determine a starting point for the output. Additionally, the command ‘EVENT’ allows us to defined the trigger point for our clock. When clear is low, the output is set to low. Otherwise, when the clock is negative, the output will change according to ‘j’ and ‘k’. For example, when the clear is high and ‘j’ and ‘k’ are both low, the output remained the same as the starting point of ‘0’. The rest of the outputs can be seen in the truth table below. The behavior of our J-K flip flop and how we included the command ‘EVENT’ as part of the code is shown in **Figure 1**.

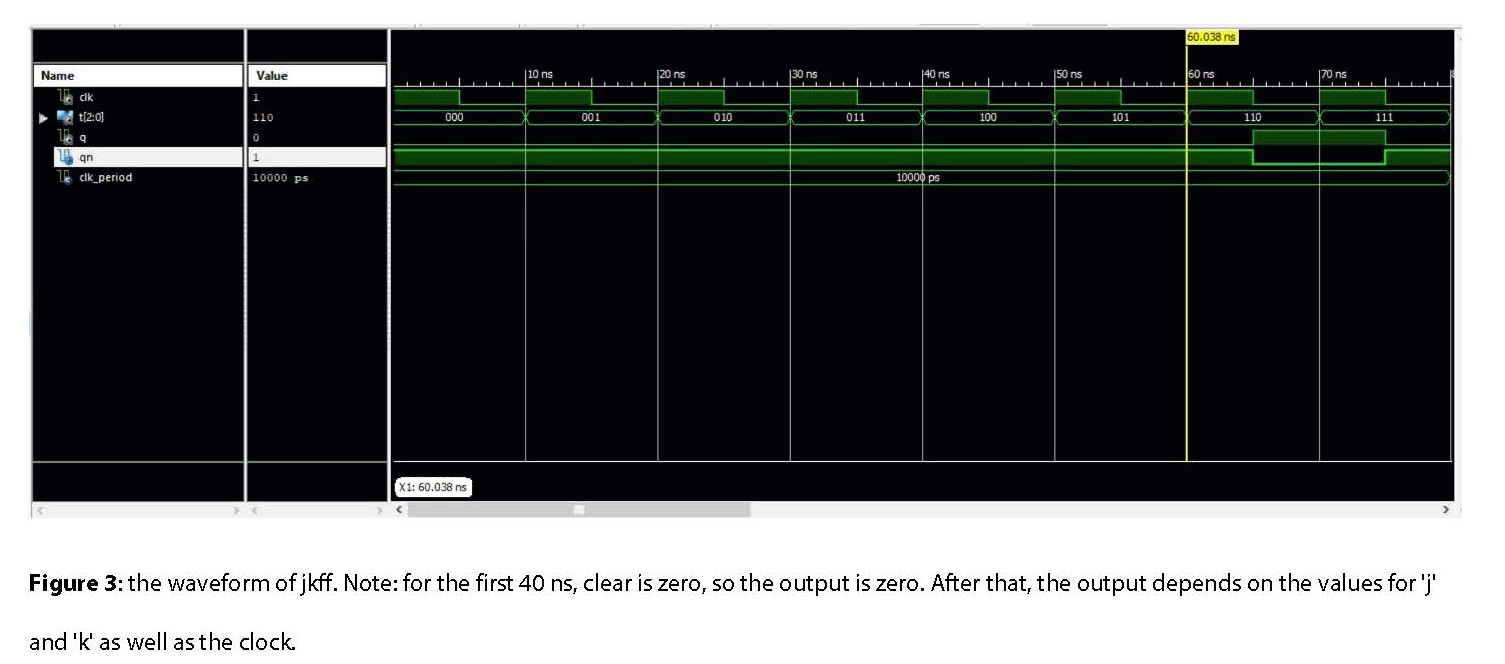
**Truth Table** (note : Q = 0 when Clear = 0)

| **Clock** | **Clear** | **J** | **K** | **Q** | **Qn** |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 0 | 0 | Q |  |
|  | 1 | 0 | 1 | 0 | 1 |
|  | 1 | 1 | 0 | 1 | 0 |
|  | 1 | 1 | 1 |  | Q |



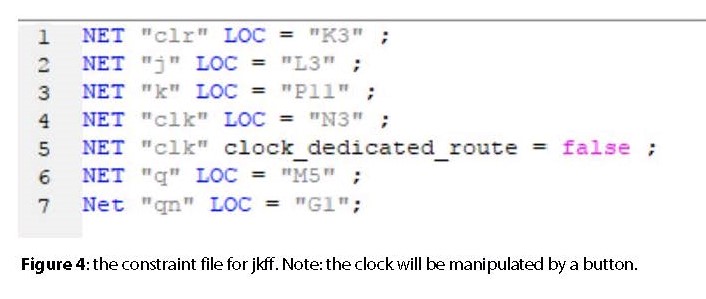
Moving on to the testbench code, the clock section was already generated by the program as part of the skeleton code. The clock period is defined as ‘10 ns’, but the clock will toggle every ‘5 ns’ because under ‘clk process’, the wait statements are set to ‘clock\_period /2’ . In order to use the loop function, we assigned input clr (MSB), ‘j’ and ‘k’ (LSB) to a signal ‘t’. Signal ‘t’ will start at “000” and increase “001” every 10 ns. The testbench code for the up counter can be seen in **Figure 2** while the behavior of our flip flop is shown in the waveform in **Figure 3**.



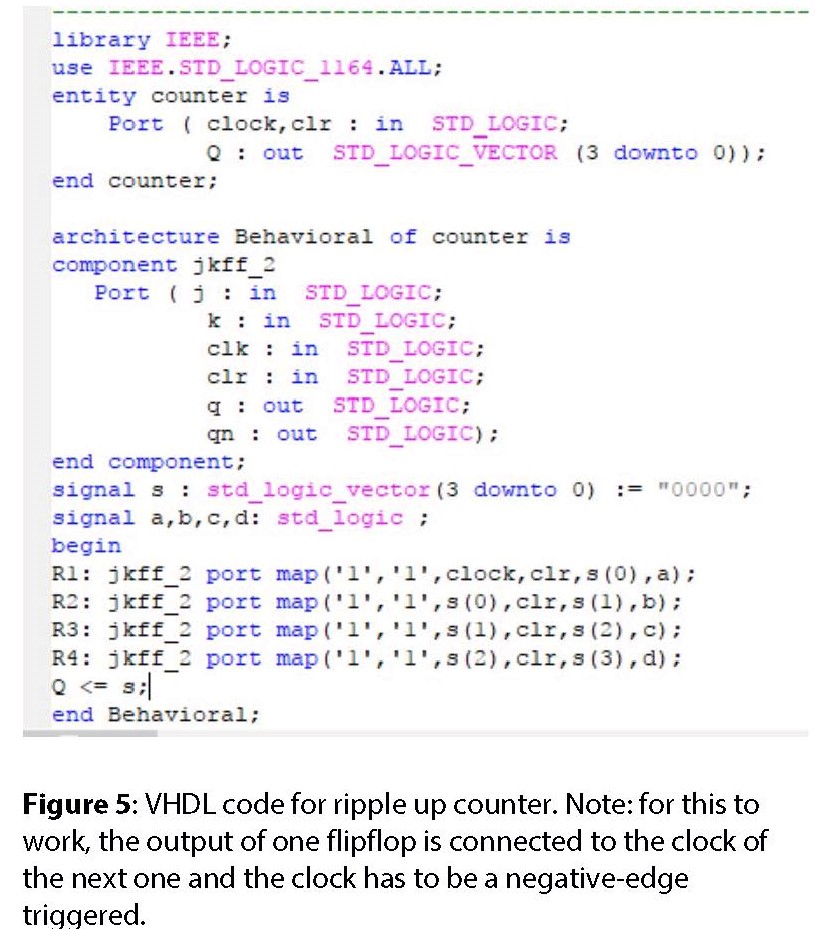


Since we wanted to control the clock manually, we had to assign the clock to one of the push buttons and let the program know by putting in a statement

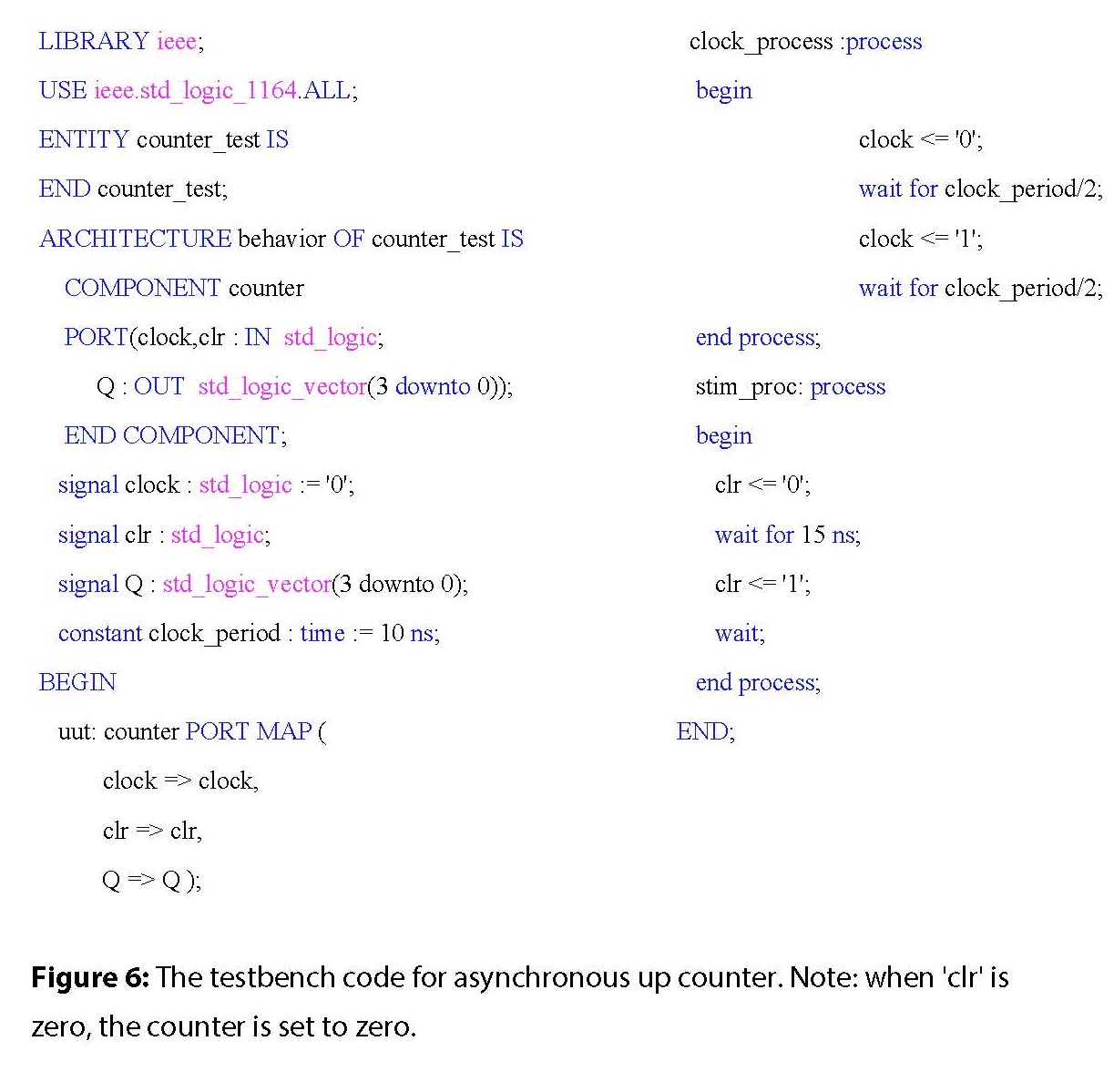
“NET “clk” clock\_dedicated\_route = false ;”. The new statement along with other key binding statements are shown in the J-K flip flop’s constraint file in Figure 4.

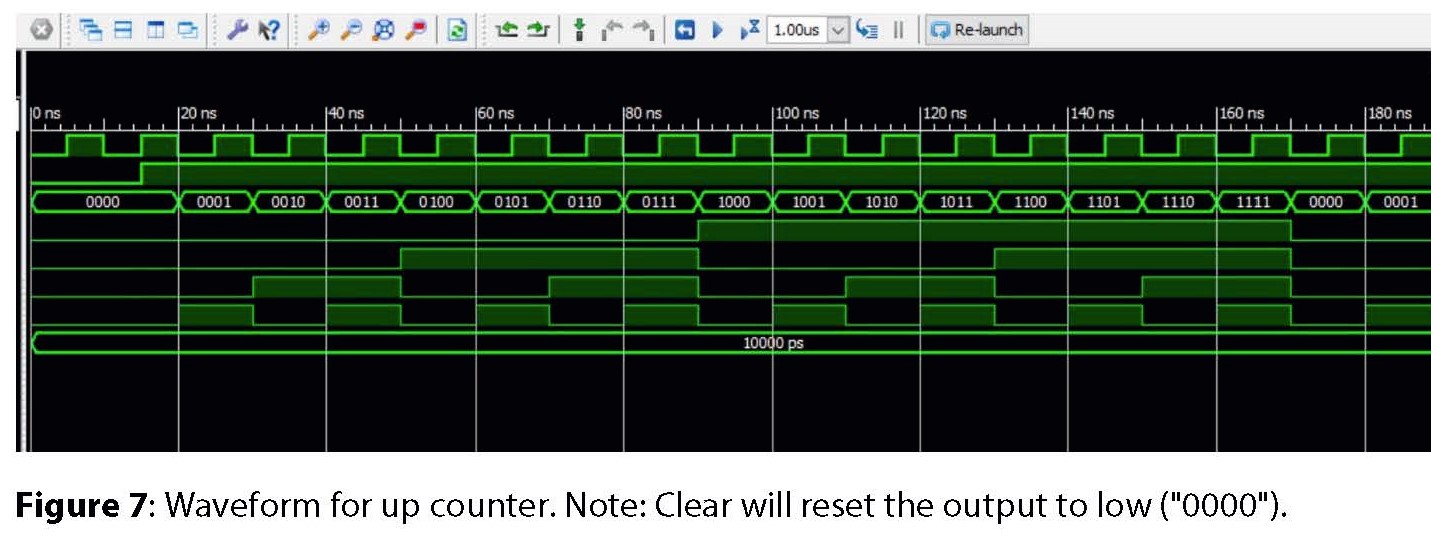


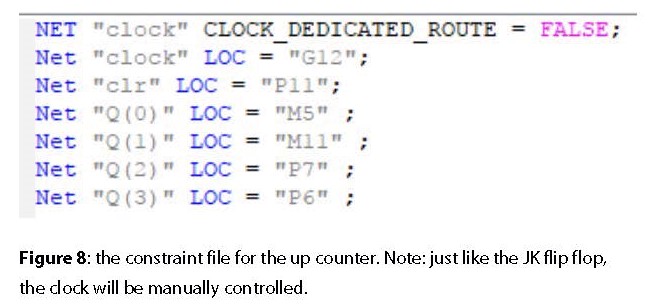
After we finished making and checking the JKFF, we created another project for the up counter. Then, we imported in the JKFF file to use as the component. To make the mod-16 counter, we need four JKFFs and those flip flops can be connected using the ‘port map’ function. Similar to the JKFF code, we used signal ‘s’ to represent the output ‘Q’ and put ‘clock and ‘clear’ as the inputs. Since this is an asynchronous counter, the output of one flip flop must be connected to the clock of the next one. Even though, ‘qn’ are not connected to anything, we still had to fill those spots in ‘port map’ sections, so we created signal ‘a’, ’b’, ‘c’ and ‘d’ and use it to accomplish that. With this set up, ‘qn’ will not affect the circuit, but will produce warnings during the synthesizing process because they are not connected to anything. The VHDL code for our ripple up counter is shown in **Figure 5**.



Next, we created the testbench code for the counter. Just like the JKFF, the clock process was already defined by the program. Under the ‘stimulus process’, however, we made the clear low for the first 15 ns to ensure that our counter started at “0000”. After that, clear will be high for the rest of the remaining duration and the output changes at the negative edge of the clock. After the output reaches “1111”, it will reset back to “0000” and repeat the counting process. The testbench code and the waveform are shown in **Figure 6** and **Figure 7**, respectively. Lastly, we included the statement “NET “clk” clock\_dedicated\_route = false ;” in the counter constraint file and the finished result is shown in **Figure 8**.

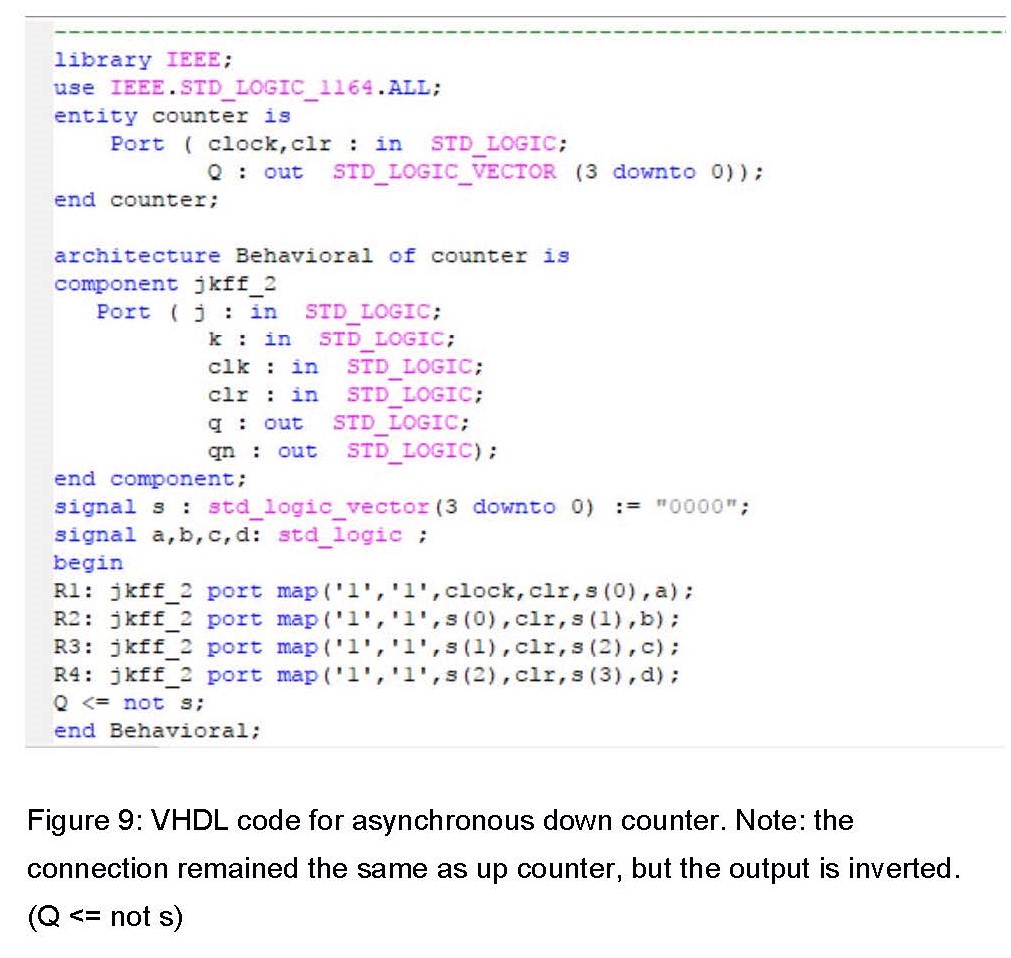


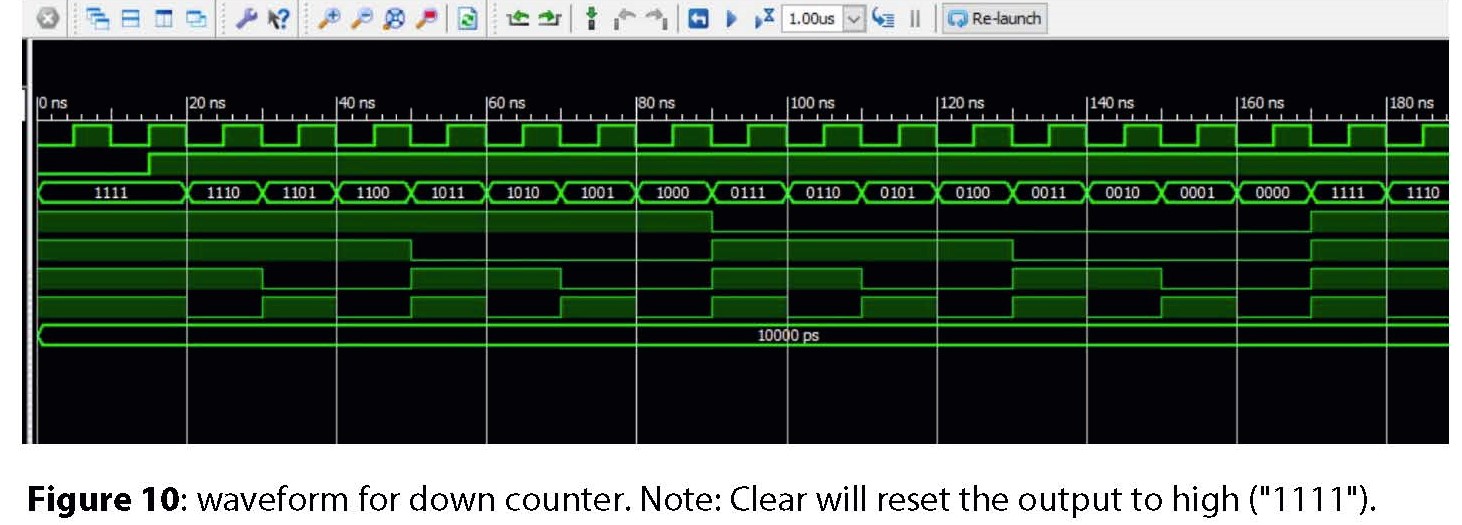




***Extra Credit***

To make the asynchronous down counter, all we had to do was inverted the signal ‘s’ before assigning it to the output. With this set up, however, the input clear will reset the output to high, which is the opposite of behavior shown for the up counter where clear will reset the output to low. The VHDL code for the down counter can be seen in **Figure 9** and the waveform is shown in **Figure 10**. As for the testbench code and the constraint file, they remained the same as the up counter.





**Discussion**

There are many ways to make a counter. Using the J-K flip-flop as a component is only one way to do it. Nevertheless, doing so teaches us how to use a clock as part of our code. With the help of the function ‘EVENT’, we can define where the clock will trigger. Additionally, the clock can be controlled manually by including a statement

“NET “clk” clock\_dedicated\_route = false ;” in the constraint file. Overall, we learned new concepts and methods which can be useful in future labs.