**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **04/06/2020** | **Name:** | **Navya R** |
| **Course:** | **HDL design** | **USN:** | **4AL16EC041** |
| **Topic:** | **Hardware modelling using Verilog**  **FPGA and ASIC Interview questions** | **Semester & Section:** | **8 A** |
| **Github Repository:** | **Navya-R** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **Report** **Clock Domain Crossing** The following section explains clock domain interfacing One of the biggest challenges of system-on-chip (SOC) designs is that different blocks operate on independent clocks. Integrating these blocks via the processor bus, memory ports, peripheral busses, and other interfaces can be troublesome because unpredictable behavior can result when the asynchronous interfaces are not properly synchronized. A very common and robust method for synchronizing multiple data signals is a handshake technique as shown in diagram below This is popular because the handshake technique can easily manage changes in clock frequencies, while minimizing latency at the crossing. However, handshake logic is significantly more complex than standard synchronization structures.    FSM1 (Transmitter) asserts the req (request) signal, asking the receiver to accept the data on the data bus. FSM2 (Receiver) generally a slow module asserts the ack (acknowledge) signal, signifying that it has accepted the data. It has loop holes: when system Receiver samples the systems Transmitter req line and Transmitter samples system Receiver ack line, they have done it with respect to their internal clock, so there will be setup and hold time violation. To avoid this we go for double or triple stage synchronizers, which increase the MTBF and thus are immune to metastability to a good extent. The figure below shows how this is done. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **04/6/2020** | **Name:** | **Navya R** | |
| **Course:** | **Python** | **USN:** | **4al16ec041** | |
| **Topic:** | **Application 8: Build a Web-based Financial Graph** | **Semester & Section:** | **8 A** | |
| **AFTERNOON SESSION DETAILS** | | | |
| **REPORT**  Candlestick data is a very essential way to show how data in the stock market moves. Some may use it to see how a stock price is doing. Some may also add color to it to visualize it better. Many also use it to map out trading patterns. They are also very helpful because instead of showing one stock price they have four different price points. These include the open price, close price, high price and low price. Outlining the Code Assuming you have prior Python knowledge, I will be creating this all in a [Jupyter Notebook](https://jupyter.org/" \t "_blank). I will be pulling the data from Yahoo using [pandas\_datareader](http://pandas_datareader/" \t "_blank). Then I will be using [plotly](https://plotly.com/" \t "_blank) to graph this information to visualize them to candlesticks. So a simple outline would look like this:   1. Import necessary libraries 2. Pull data from Yahoo using pandas\_datareader 3. Store data into a DataFrame 4. Match the DataFrame with plotly candlestick format 5. Use plotly to visualize data from the DataFrame  Creating the Code Starting off in Jupyter I do all the necessary imports.   |  | | --- | | import pandas as pd | |  | from pandas\_datareader import data as web | |  | import plotly.graph\_objects as go |   For this example, I will be using Microsoft as my stock. I set the ticker symbol to a variable and then I use pandas\_datareader to get information from Yahoo and store that into a variable. It should automatically save as a DataFrame object. For the date I just have it set to the beginning of last year.  Stock=’MSET   |  | | --- | |  | |  |  | |  | df = web.DataReader(stock, data\_source='yahoo', start='01-01-2019') |   In order for plotly to understand our data, we need to match it with the correct information. They have made it simple and use “traces”, think of traces as options for the graph. We define what we want to use from the DataFrame and then we set these in the options.  We can now set the chart layout in plotly.   |  | | --- | | trace1= { | |  | 'x': df.index, | |  | 'open': df.Open, | |  | 'close': df.Close, | |  | 'high': df.High, | |  | 'low': df.Low, | |  | 'type': 'candlestick', | |  | 'name': 'MSFT', | |  | 'showlegend': True | |  | } |   Then we can now show the visualization. It should look like something like this. Feel free to use the tools to change it around. | | | |
| Bonus [Moving averages](https://www.investopedia.com/terms/m/movingaverage.asp) can also be plotted! Most traders use moving averages to see what direction a stock will go to. Some also use varying days, within these days if a crossover (when varying moving averages intersect) happens they can use it as a signal to buy or sell. Adding it to our code is really simple. We just create separate traces for each moving average. I have created a trace for a 30 day moving average and another for 50 days.   |  | | --- | | # Calculate and define moving average of 30 periods | |  | avg\_30 = df.Close.rolling(window=30, min\_periods=1).mean() | |  |  | |  | # Calculate and define moving average of 50 periods | |  | avg\_50 = df.Close.rolling(window=50, min\_periods=1).mean() | |  |  | |  | trace2 = { | |  | 'x': df.index, | |  | 'y': avg\_30, | |  | 'type': 'scatter', | |  | 'mode': 'lines', | |  | 'line': { | |  | 'width': 1, | |  | 'color': 'blue' | |  | }, | |  | 'name': 'Moving Average of 30 periods' | |  | } | |  |  | |  | trace3 = { | |  | 'x': df.index, | |  | 'y': avg\_50, | |  | 'type': 'scatter', | |  | 'mode': 'lines', | |  | 'line': { | |  | 'width': 1, | |  | 'color': 'red' | |  | }, | |  | 'name': 'Moving Average of 50 periods' | |  | } | | | | |