ECS527U: Digital System Design
Lab 2: VHDL Modelling of Sequential Blocks

Bit Slice for Shift Register:

The shift register bit slice contains a multiplexer and a d flip-flop, where the multiplexer picks different inputs depending on the value of the selector input. The flip-flop then copies D to Q at every positive clock edge from the clock input.

The shift register bit slice has 6 inputs and 1 outputs. This meant declaring 7 ports in the entity, it also meant declaring the number bits expected through these ports. In this specific bit slice, the port **S** is a selector which selects the correct output from the multiplexer. For this reason the **S** port must be declared as standard logic vector **(1 downto 0)** to let the synthesiser know that it should expect 2 bits from this input port.

This bit slice only has two intermediate signals, one signal is the output from the multiplexer to the flip-flop and the other is the output from Q to the first input in the multiplexer. For this reason we declared two signal variables in the architecture. The **Mux** signal refers to the output from multiplexer to flip-flop and the **Qout** signals refers to the output from the flip-flop back to the multiplexer.

In order to describe the functionality of the flip-flop, a process statement was used, with the clock signal in the sensitivity list, this because clock signal triggers the output Q. Due to the statements inside the process being executed sequentially, the first statement in the process highlights that they will only be an output from the flip-flop when there is a positive clock edge, this is done using an **if** statement that has **CLK'event and CLK = '1'** inside is parentheses which indicates when there is a change in clock and that change is a rising edge then **Qout <= Mux** (Output from Multiplexer will equal the output from the flip-flop).

This bit slice also has a reset input that is synchronous and set the output Q to 0 at the next clock edge, for this reason we have made a nested **if** statement that goes inside the previous if statement that has **RST = '1'** inside its parentheses if this statement evaluates to true then **Qout <= '0'.**

After ending both if statement and the process using the keywords **end if** and **end process.** We used select statements to describe the functionality of the multiplexer, by defining what selector input corresponds to the multiplexer output, **Mux**.

Lastly we map the intermediate signal Qout to the output signal Q.

Bit Slice for Synchronous Counter:

The synchronous counter bit slice has 4 inputs and 2 outputs. Inside this bit slice a Full Adder is used along with a D Flip Flop, to produce the outputs Q via the D Flip Flop, and Cout via the Full Adder. The Q output is mapped to the intermediate signal B as an input to the Full Adder, whilst the intermediate output S is mapped to the intermediate signal D as an input to the D flip flop.

For this bit slice to work we would need to **port map** the present Full Adder component values to the intermediate signals of the Full Adder. This is done for efficiency so that changes made to the input values directly affect the intermediate signals, as well as the outputs from the Full Adder directly affecting the bit slice outputs. The objective of the Full Adder is to add the three one bit binary inputs: **Cin, A and B** to produce an output.

The functionality of the Bit Slice is described via the use of a process statement. This process statement includes the variables **CLRn and CLK** as part of its sensitivity statement. This suggests that the process is activated by changes to the clock and CLRn port. The statement **if(CLK'event and CLK = '1')**, is used to simulate the D Flip Flop mapping of D to Q at every positive edge clock pulse. This is because the D Flip Flop is synchronous with the clock.

The process also uses a nested if statement to simulate if the CLRn port is active low, however this is also synchronous so it only alters the value of Q at the next clock cycle. So to show this the if statement **if(CLRn = '0')** is nested, and the value of the intermediate output signal of the D Flip Flop **Qout** is changed to **'0'** if the CLRn port is active low, this covers all the cases of the process so we end the process using **end process;**.

Since we only port mapped the Full Adder components to the intermediate signals, we will need to map the inputs and outputs of the D Flip Flop. As seen in the code below we map the intermediate signals of the D Flip Flop and Full Adder together, in addition to mapping the output Q to the intermediate output of the D flip Flop.

```
Bout <= Qout;
D <= S;
Q <= Qout;
```

4-bit Universal Shift Register:

The 4-bit universal shift register uses the shift register bit slice described earlier and connects them together to make a 4-bit universal shift register. The shift register then allows the user to either shift the bits towards the most significant bit or the least significant bit, keep the output unchanged or load value in the D flip-flop.

Like previously in the bit slice this shift register has 6 input ports and 1 output ports declared in its entity. However this time it contains 2 standard vector ports **Din and Q** each 4 bits. **Din** is 4 bits and equals the output **Q** when **LD** is '1', **Q** is the sum of outputs of each bit slice, because they are 4 bit slices they are 4 bits.

Being that bit slice shift register already exists in our project we can use this description as a **component,** by copying the ports in the bit slice and mapping them to the signals created. This all done in the architecture of the code. The signals created are all intermediate signals of the component, being that inputs of the bit slice become intermediate signals in the 4-bit universal shift register. In order to create a 4 bit slice I used **generate** to loop the mapping for components to the signal. For this reason the intermediate signal Qint, Pint, Nint, D are all standard vector logics of 4 bits. They are then port mapped using the keyword **port map** and specifying the name of the component you are trying to port map to. For example: **reg: ShiftRegBit port map (P => Pint(i) , D =>Din(i), N => Nint(i), S => S, RST => RST, CLK => CLK).**

Pint and Nint, intermediate signals correspond to previous and next signals. Therefore to make output to be the previous signal the first input bit **Sin** needs to be the least significant bit and the first bits needs to be from Q0 to Q2 where **Sin** replaces Q3. This is done by concatenating these variables and making them equal to Pint. **Pint <= Qint(2 downto 0) & Sin**. To make the output the next bit then Sin needs to be the most significant bit and the next bits need to go from Q3 to Q1 where **Sin** replaces Q0. This is done by concatenating these variables and making them equal to Nint . Nint <= Sin & Qint(3 downto 1).

Now to describe the functionality of shift register, a process is used with concurrent statements. The clock signal is included in the sensitivity list to indicate that the process is triggered by the clock signal when its positive edge triggered.This means a nested if statement is used where the upper if statement checks if its positive edge triggered first and then checks for other input. The following inner if statement corresponds to the Shift register function and specification given in the lab.

4-bit Synchronous Up/Down Counter:

The 4 bit synchronous Up/Down Counter cascades 4 synchronous counter bit slices together in order to count up and down in value, this is dependent on the inputs CLRn, U_Dn and En. the counter is synchronous so it is directly affected by the clock similar to the bit slice, it is only affected by positive edge triggers.

The counter is different in the fact that q is only changed when the enable is active high, if not the Q output will remain unchanged at the next clock cycle. In addition since this is a 4 bit counter the output Q will be 4 bits in length so we use a logic vector **Q: out std_logic_vector(3 downto 0).**

In order to cascade the 4 slices together we must use the synchronous counter bit slice as a component and port map each of the components to intermediate signals and entity variables. The counter works by using the inputs U_Dn and En to produce an intermediate signal. **Xi<= Not U_Dn** with this every X component value is port mapped to Xi to determine whether to count up or down. The value for Xi is added with Ci which is mapped to the value C which is derived by **C <= En XNOR U_Dn.** however Ci is only mapped to C initially since after the first counter in order to cascade the bit slices we need to map the current Co straight to the Ci of the next bit slice.

In logic you generally cannot subtract values, so we cannot negate ‘1’ from the previous Q value, instead we use 2’s complement by adding X input of ‘1111’ and a Cin of 0 in addition to the Q current value. This in a Full Adder produces a carry output Co. we use a Not gate to simulate the input X since when the U_Dn is low we want to subtract ‘1’ so we Not the value of U_Dn to provide ‘0’, which we can map to each X port inthe 4 bit slices to give us ‘1111’. To give us a Cin value of ‘0’ to count down we must use the XNOR of En and U_Dn, this works since when enable is ‘1’ and U_Dn ‘0’ we receive the required value ‘0’ as our Cin.

In the design to add ‘1’ to the previous Q value. We do not require 2’s complement instead we simply want to add 1. In order to accomplish this the Not gate provides the 4 bit X value of ‘0000’ so we can simply write a ‘1’ to Cin to simulate the addition of ‘1’. This is done when U_Dn is equal to ‘1’ whilst the enable is high since, **Xi<= Not U_Dn** produces ‘0’ and **C <= En XNOR U_Dn** produces the desired Cin of ‘1’.

Finally the CLRn component port is port mapped to the entity port, which when active low causes Cin to remain unchanged, so producers do not care, cases for U_Dn.