CS 254 - Assignment 7

## Group 14

The Encoder is in the file RLEncoder. It uses two sub-components:

1. FourbitUpCount: This is a simple counter which counts from 0 to 15. This is used to maintain the number of times the current character has occurred
2. DFlipFlop: This is the flip flop that was provided in the previous lab. It is used to store the current character

**Input and Output generation:** The basic idea is that I store the current character in a flip flop (the corresponding signal is named current\_char) and then when a new input comes, I compare the two and take action accordingly. The count of the current character is stored in a signal named cnt. It is incremented at each step until I set reset to 1 which denotes that the count must be set back to 1. (Note that cnt actually store value 1 less than the actual count. So, I have to update everything accordingly.)

The conditions to be checked are quite straight forward. First, we check if we actually need to generate an output or not. This happens when new character arrives or the count reaches 15. This is done using the condition: if (current\_char = input and unsigned(cnt) < 14).

If new output is to be generated, we check whether the current character is Esc or any other character. If it is Esc, the new output is “Esc n Esc” which is then pushed to buffer. (Usage of buffer is explained later). If it is not, we have to check whether the count is less than 3. If it is, then we just push the character cnt+1 number of times. Otherwise we push “Esc n c” to the buffer. This completes the process of how output is generated. Once the output is generated on the rising edge of the clock, current\_char is updated to input and the cnt is changed on the falling edge of clock. (The clock is represented by the signal clk.)

**Buffer:**  The buffer is maintained using the variable buff.It is of type memory which is defined in the code as array (97 downto 0) of std\_logic\_vector(7 downto 0). So it is basically a 2D array of std\_logic.

The buffer can be thought of as 8 shift registers of maximum length 97. This is denoted by buff (97 down to 1). At each step, the shift happens from i+1 to i in the loop:

for i in 1 to 97 loop -- shift the buffer.

buff(i-1) := buff(i);

end loop;

But this is different from the normal shift registers. In normal shift registers, usually a constant length of input is given. (Note that the input to buffer is the output that is discussed above). Here, both is not true. The input to buffer is of variable length as described above. Because of this, it is not possible to use a normal shift register. So, I have to use a variable length. The length is stored in index. (It is actually 1 more than the length and denotes the first empty cell). Whenever there is a new input to buffer, I append it to the current buffer and increase the length. The append is done by storing the input in buff(index), buff(index+1) and so on. The increase in length is done during the falling edge of the clock. How much it is to be incremented is stored in the variable increment. Note that due to shifting, the length is decreased by 1 each time. So, increment is 1 less than the number of inputs.

**Output:** Note that in the above model, there are no “gaps” between 2 outputs. So, we can just use the value that is just popped from the shift register as the output. This value is stored in buff (0). So, we just assign output to be this.

The only case when there is no output is when the buffer is empty. It can easily be checked if the buffer is empty or not by maintaining a new signal. This is named empty in the code. So, whenever empty = 1 i.e. the buffer is empty, we set the data valid signal(denoted by valid) to 0 otherwise 1.

**Start and End:** The signals 00000000 and 00000001 have special meaning for the encoder. 00000000 is the initial state i.e. the state before the first input(This is required as before the encoder start, we have to reset all the Flip Flops). 00000001 denotes the end of input. So, after all the inputs are processed, we send the signal 00000001 so that the last output is generated. Since these 2 signals are to be handled specially, there are some extra conditions so that they are not outputted as well.

**Testbench:** The testbench is present in the file ASCII\_Read\_test.vhd. The process in the testbench is again straight forward.

We first reset all the flip flops and variables by setting rst to 1 for half clock cycle(all the resets are done on the falling edge of clock, so we can just give that). This is done in the code:

rst <= '1';

clk <= '1';

wait for 20 ns;

clk <= '0';

wait for 10 ns;

rst <= '0';

Next, we start reading the file and sending the input to the encoder. The output from encoder is written the file if the valid signal is 1. We use a for loop outside this because there may be cases when the output file has more characters then input file. Also, the size of the input file or the output file can’t exceed 96 bytes. So, we loop 96 times. (The looping condition can be updated to loop when the end of input file is not reached or the output buffer is not empty. To do this, we will also have to send the empty signal from the entity to the testbench).