

**Multiple of Three Sequence Detector Design:**

1. Knowing that “A number is entered serially [...] starting from the most-significant bit” and “the number is effectively multiplied by two and incremented by zero or one, depending on the input received”, this means that we are working with positive integers only, and that bits are added from the right extremity of the bit sequence. Hence, the following figure represents the state diagram of the multiple-of-three sequence detector.

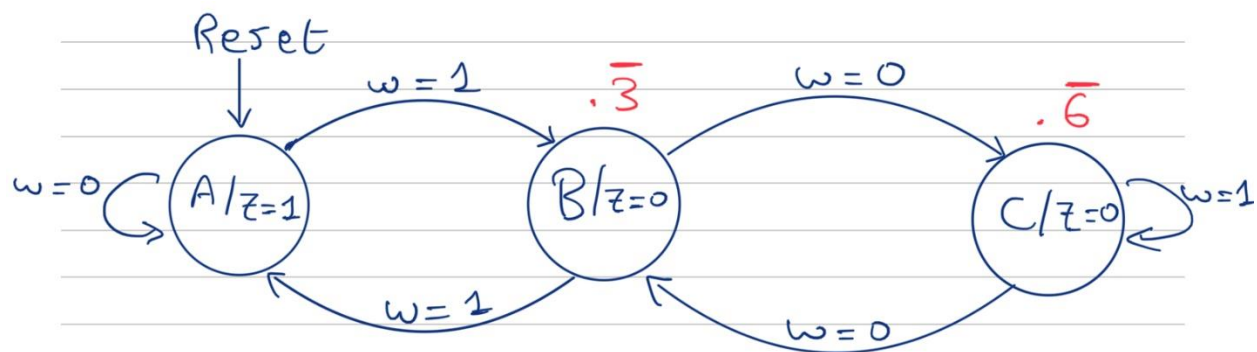


Figure 1: State diagram of the multiple-of-three sequence detector. The red notations represent the decimal part of an arbitrary number when divided by three ( $.3$  means that  $n \bmod 3 = 1$  (1-modulus number) and  $.6$  means that  $n \bmod 3 = 2$  (2-modulus number)), which is a pivotal information in this assignment, as explained below. Note that when  $w = 0$ ,  $n$  becomes  $n' = 2*n$ , and when  $w = 1$ ,  $n$  becomes  $n' = 2*n + 1$ . Additionally,  $z=0$  means that the current number is not divisible by 3, and  $z=1$  means that  $n \bmod 3 = 0$ . By simple observation of this diagram, it is easy to discern that this state model is a Moore machine, as the FSM output depends on current state and the outputs  $z_i$  appear inside the nodes.

The multiple-of-three sequence detector's algorithm is deduced from experimenting with many numbers and noticing a specific pattern regarding the decimal part of the divide-by-three result of the number: assuming some positive integer  $n$  divisible by three (state A), then  $2n$  would also be divisible by 3, hence the self-loop when  $w = 0$ , but  $2*n + 1$  would always result a 1-modulus number when divided by 3, hence the transition from state A to B when  $w = 1$  (e.g. for  $n = 15$ ,  $\frac{(15 \times 2 + 1)}{3} = \frac{31}{3} = 10.\bar{3}$ ). Similarly, at state B, a 1-modulus number  $k$  is divisible by 3 when  $k' = 2*k + 1$  (e.g. for  $k = 31$ ,  $\frac{31 \times 2 + 1}{3} = \frac{62 + 1}{3} = \frac{63}{3} = 21$ ), hence the transition back to state A when  $w = 1$ , and becomes a 2-modulus number when doubled with no incrementation (e.g. for  $k = 31$ ,  $\frac{31 \times 2}{3} = \frac{62}{3} = 20.\bar{6}$ ), which explains the B-to-C state transition when  $w = 0$ . Finally, at state C, a 2-modulus number becomes a 1-modulus number if doubled (e.g. for  $m = 62$ ,  $\frac{2 \times 62}{3} = \frac{124}{3} = 41.\bar{3}$ ), hence the backward transition from state C to B, and stays a 2-modulus number when  $m' = 2*m + 1$  (e.g. for  $m = 62$ ,  $\frac{2 \times 62 + 1}{3} = \frac{125}{3} = 41.\bar{6}$ ), hence the self-loop when  $w = 1$ .

2. Before proceeding with the state-assigned table, it is a great practice to derive the design's state table first.

Present state	Next state		Output $z$
	$w=0$	$w=1$	
A	A	B	1
B	C	A	0
C	B	C	0

Figure 2: State table of the multiple-of-three sequence detector.

Thus, by assigning "00" to state A, "01" to state B and "11" to state C, these two-bit sequences representations would allow us to directly derive the state-assigned table, which is basically the state table but bit-wise documented. Note that a two-bit sequence has a maximum capacity of four state options, so the fourth state "11" can be used as a don't-care state that would benefit us in Boolean simplification, state minimization and design optimization in later steps.

	Present state	Next state		Output $z$
		$w=0$	$w=1$	
	$y_2 y_1$	$y_2 y_1$	$y_2 y_1$	
A	00	00	01	1
B	01	10	00	0
C	10	01	10	0
	11	dd	dd	d

Figure 3: State-assigned table of the multiple-of-three sequence detector.

3. Consider the state table of Figure 2. We apply the state minimization procedure and begin by assigning all states to the same block:  $P_1 = (ABC)$ . This initial block can be divided in two. All states that produce  $z = 1$  (state A) are assigned to one block, and all states that produce  $z = 0$  (state B & C) are assigned to another block:  $P_2 = (A)(BC)$ . The 0-sucessors of (BC) are (CC) and the 1-sucessors are (AB). While the 0-sucessors belong to the same block, the 1-sucessors do not, as A and B belong to different blocks. More precisely, noticing that state B's 1-sucessor, A, belongs to a different block than the 1-sucessor of C, we can deduce that B cannot be equivalent to C and thus should be assigned to a different block, which lead to a new partition:  $P_3 = (A)(B)(C)$ .

Finally, we know that single elements blocks are trivially belonging to the same block, as there's no multiple states together that might interfere with each other and create inconveniences. Hence, we can directly assert that  $P_4 = P_3 = (A)(B)(C)$  is the final partition, and thus state A, B and C are far from being equivalent to each other, which means that the original state diagram and state-assigned table of Figure 1 and 3 respectively cannot be updated, as they hold no redundant states and are provided and derived in their optimal configuration from the very beginning. This concludes the state minimization procedure step and thus our design optimization is verified.

4. To implement the controller using D-FFs, we must first acknowledge that n-bit input sequences require n D flip-flops. Hence, the two-bits input sequence representing the different states of the multiple-of-three sequence detector requires two D flip-flops to be implemented. Using the state-assigned table of Figure 3, we can use Karnaugh maps to derive the logic expressions of the FSM's next-state variables,  $Y_1 = D_1$  and  $Y_2 = D_2$  and its output z. This is called the logic expression derivation step, where all evaluations are presented in the figure below.

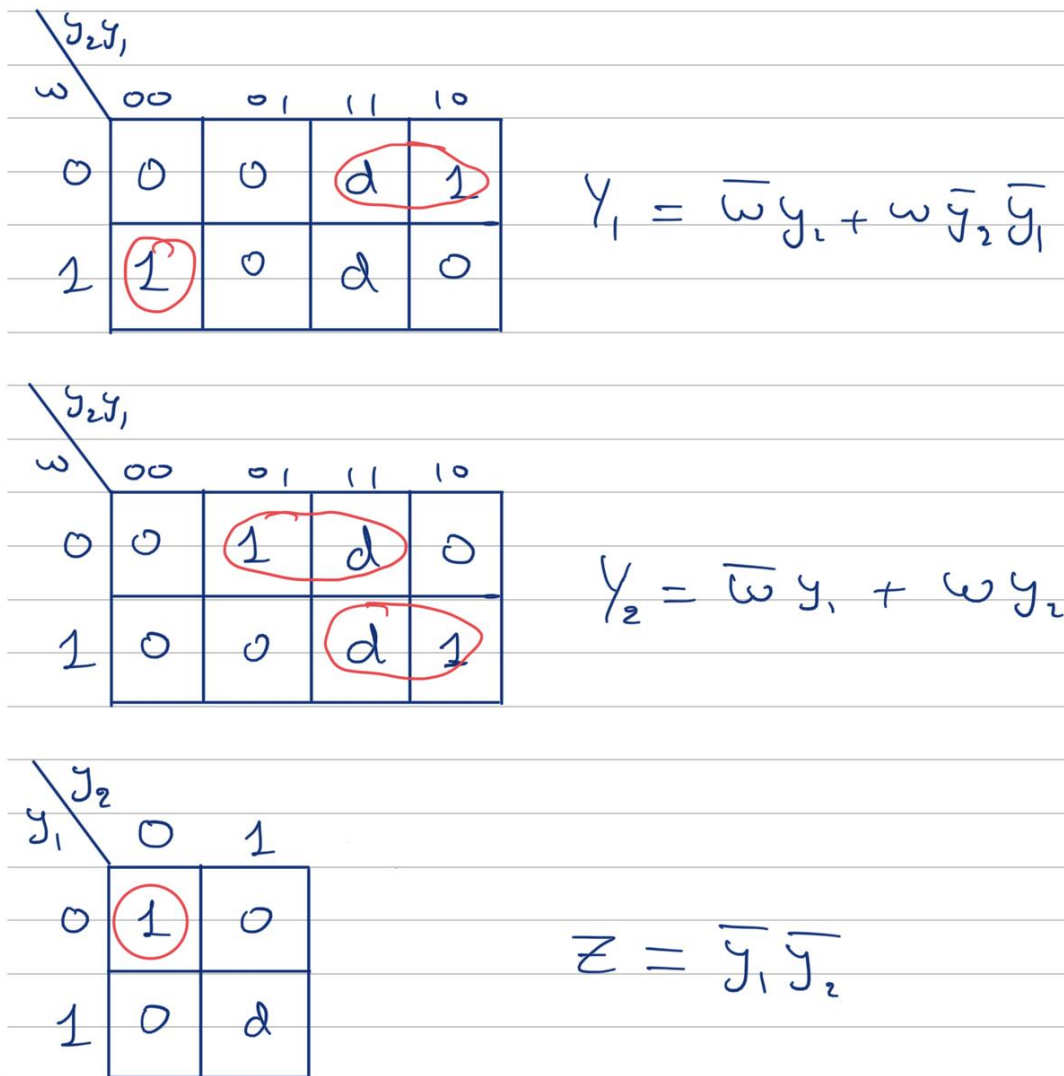


Figure 4: the logic expression derivation for the multiple-of-three sequence detector, where K-maps are used to derive the logic equations for each D flip-flop's inputs  $Y_1 = D_1$  and  $Y_2 = D_2$ , as well as for the controller output z.

5. Based on the simplified logic equations derived in part 4, we can draw the below diagram representing the controller circuit built using D flip-flops with asynchronous reset and serial input.

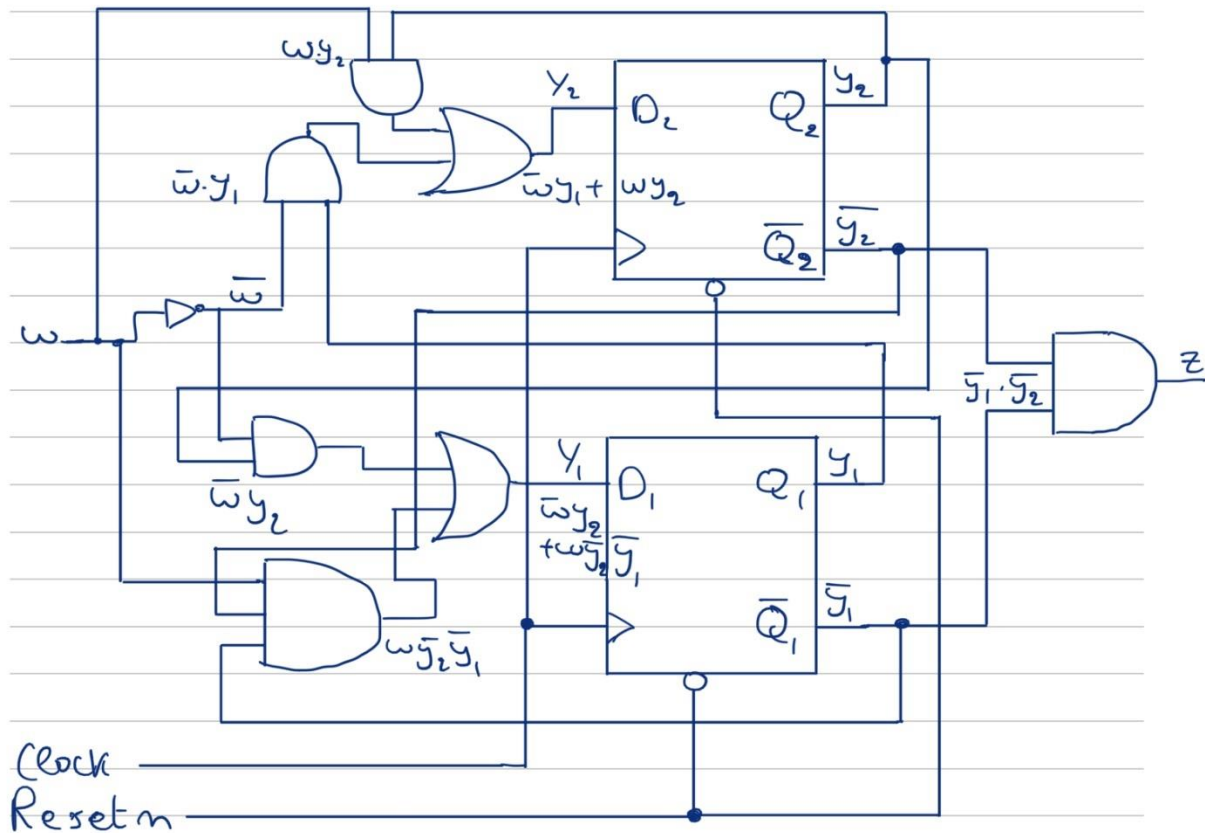
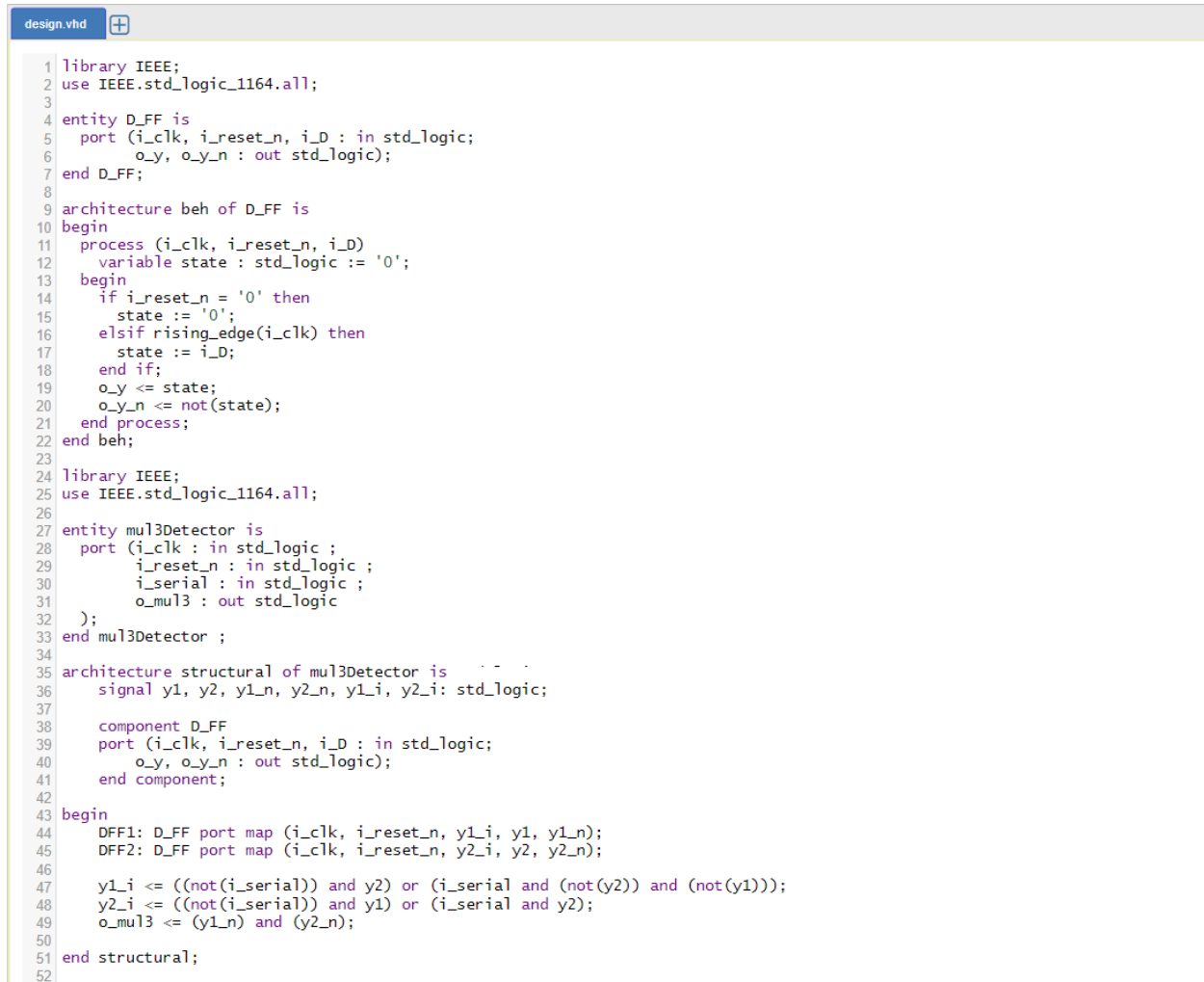


Figure 5: Schematic diagram of the multiple-of-three sequence detector design.

**Caveat:** for the sake of code neatness, I will be presenting the VHDL codes as two separated design.vhd codes for the structural and behavioral implementation of the multiple-of-three sequence detector FSM. They are named "design\_structural.vhd" and "design\_behavioral.vhd" respectively in the accompanying .zip file, where both share one same testbench.vhd file in order to prove the two design files' output superposition, which will validate that both implementation approaches are equivalent to each other.

6. The following figure is a screen snipping of the design\_structural.vhd code implemented on EDA Playground.



```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity D_FF is
5   port (i_clk, i_reset_n, i_D : in std_logic;
6         o_y, o_y_n : out std_logic);
7 end D_FF;
8
9 architecture beh of D_FF is
10 begin
11   process (i_clk, i_reset_n, i_D)
12     variable state : std_logic := '0';
13   begin
14     if i_reset_n = '0' then
15       state := '0';
16     elsif rising_edge(i_clk) then
17       state := i_D;
18     end if;
19     o_y <= state;
20     o_y_n <= not(state);
21   end process;
22 end beh;
23
24 library IEEE;
25 use IEEE.std_logic_1164.all;
26
27 entity mul3Detector is
28   port (i_clk : in std_logic ;
29         i_reset_n : in std_logic ;
30         i_serial : in std_logic ;
31         o_mul3 : out std_logic
32   );
33 end mul3Detector ;
34
35 architecture structural of mul3Detector is
36   signal y1, y2, y1_n, y2_n, y1_i, y2_i: std_logic;
37
38   component D_FF
39     port (i_clk, i_reset_n, i_D : in std_logic;
40           o_y, o_y_n : out std_logic);
41   end component;
42
43 begin
44   DFF1: D_FF port map (i_clk, i_reset_n, y1_i, y1, y1_n);
45   DFF2: D_FF port map (i_clk, i_reset_n, y2_i, y2, y2_n);
46
47   y1_i <= ((not(i_serial)) and y2) or (i_serial and (not(y2)) and (not(y1)));
48   y2_i <= ((not(i_serial)) and y1) or (i_serial and y2);
49   o_mul3 <= (y1_n) and (y2_n);
50
51 end structural;
```

Figure 6: The design.vhd code of the structural implementation of the multiple-of-three sequence detector.

An explanation for this implementation is the structural definition that is characterized by the modulation of complex circuits using instances of simpler circuits and a description of how they should be connected. Hence, if we observe the following simplified diagram representation of the FSM design (Figure 7), we can notice that the structural implementation describes exactly how our circuit is physically connected, but expressed in coding language using signals and components.

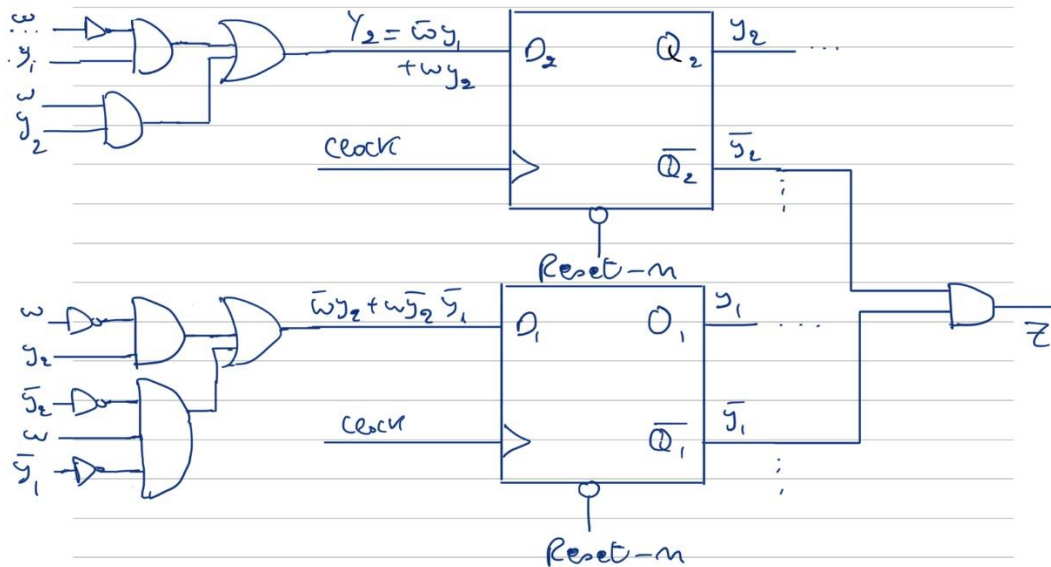


Figure 7: Simplified representation of the multiple-of-three sequence detector design.

- The following figure is a screen snipping of the testbench.vhd code implemented on EDA Playground with the top entity chosen to be mul3DetectorTB. This same testbench is applied on both the codes of Figure 6 (structural implementation) and Figure 10 (behavioral implementation).

```

testbench.vhd
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity mul3DetectorTB is
5 end mul3DetectorTB;
6
7 architecture driver of mul3DetectorTB is
8   component mul3Detector
9     port (
10       i_clk : in std_logic ;
11       i_reset_n : in std_logic ;
12       i_serial : in std_logic ;
13       o_mul3 : out std_logic
14     );
15   end component;
16
17   signal tb_clk: std_logic := '0';
18   signal tb_reset_n: std_logic := '0';
19   signal tb_serial : std_logic := '0';
20   signal tb_o_mul3 : std_logic;
21
22 begin
23   UUT : mul3Detector port map (i_clk=>tb_clk, i_reset_n=>tb_reset_n, i_serial=>tb_serial, o_mul3=>tb_o_mul3);
24
25   process
26   begin
27     for i in 1 to 20 loop
28       tb_clk <= '0';
29       wait for 10 ns ;
30       tb_clk <= '1';
31       wait for 10 ns ;
32     end loop ;
33     wait ;
34   end process ;
35
36   tb_reset_n <= '1' after 1 ns, '0' after 100 ns, '1' after 101 ns;
37   tb_serial <= '1' after 10 ns, '0' after 20 ns, '1' after 30 ns, '1' after 40 ns, '0' after 50 ns, '1' after
60 ns, '0' after 70 ns, '1' after 80 ns, '1' after 90 ns, '0' after 100 ns, '1' after 110 ns, '0' after 120
ns, '1' after 130 ns, '1' after 140 ns, '0' after 150 ns, '1' after 160 ns, '0' after 170 ns, '1' after 180
ns, '1' after 190 ns, '0' after 200 ns, '0' after 210 ns, '1' after 220 ns, '1' after 230 ns, '0' after 240
ns, '0' after 250 ns, '1' after 260 ns, '1' after 270 ns, '0' after 280 ns, '0' after 290 ns, '1' after 300
ns, '1' after 310 ns, '0' after 320 ns, '0' after 330 ns, '1' after 340 ns, '1' after 350 ns, '0' after 360
ns, '1' after 370 ns, '0' after 380 ns, '1' after 390 ns, '1' after 400 ns;
38
39 end driver;

```

Figure 8: The corresponding testbench.vhd code of the multiple-of-three sequence detector, applied on both the structural and behavioral implementations.

As a visualization, below lies the EPWave graph representation of the structural implementation of the multiple-of-three sequence detector, where the codes of Figures 6 and 8 are getting executed.

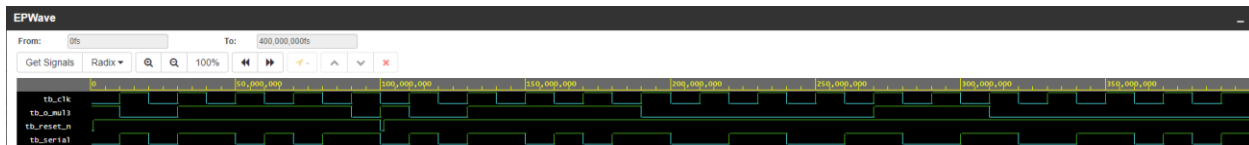


Figure 9: the EPWave graphical representation of the testbench of Figure 8 applied on the structural design code of Figure 6.

8. The following figure is a screen snipping of the design\_behavioral.vhd code implemented on EDA Playground.

```

design.vhd
VHDL Design

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity mul3Detector is
5     port (i_clk : in std_logic ;
6           i_reset_n : in std_logic ;
7           i_serial : in std_logic ;
8           o_mu13 : out std_logic
9     );
10 end mul3Detector ;
11
12 architecture behavioral of mul3Detector is
13     type myState is (A, B, C);
14     signal y: myState;
15 begin
16     process (i_reset_n, i_clk)
17     begin
18         if i_reset_n = '0' then
19             y <= A;
20         elsif rising_edge(i_clk) then
21             case y is
22                 when A =>
23                     if i_serial = '0' then y <= A;
24                     else y <= B;
25                     end if;
26                 when B =>
27                     if i_serial = '0' then y <= C;
28                     else y <= A;
29                     end if;
30                 when C =>
31                     if i_serial = '0' then y <= B;
32                     else y <= C;
33                     end if;
34             end case;
35         end if;
36     end process;
37     o_mu13 <= '1' when y = A else '0';
38 end behavioral;

```

Figure 10: The design.vhd code of the behavioral implementation of the multiple-of-three sequence detector.

An explanation for this implementation is the behavioral definition that focuses on how the design is behaving in terms of results rather than how physically connected is the design in terms of concrete configuration. Hence, we can simply describe the behavioral of this detector design as an exact replication of its FSM implementation that is heavily inspired by the design's state diagram and state table of Figures 1 and 2. As an illustration, we can similarly provide the EPWave graph representation of this behavioral implementation, where the codes of Figures 10 and 8 are executed as design and testbench codes respectively.

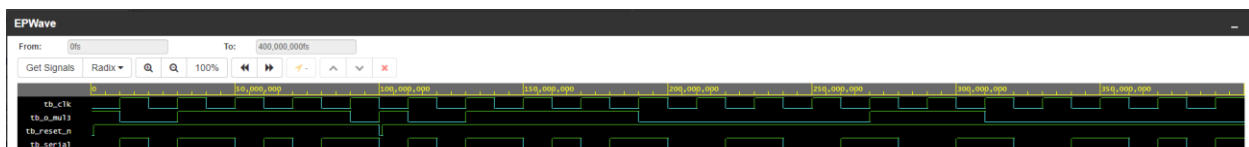


Figure 11: the EPWave graphical representation of the testbench of Figure 8 applied on the behavioral design code of Figure 10.

The graphical superposition between Figure 9 and 11 for the same testbench of Figure 8 is trivially observable, which proves the equivalence of both of these implementations, the correctness of our physical design derived in parts 2-5, and the validation of our applied testbench.

The graphs, however, require some scrutiny to discern whether the outputs received are correctly indicating whether the input number is a multiple of three or not. Hence, let us consider two regions of the EPWave. A good remark is to remember that D flip-flops only reflect the input value at the rising edge of the clock waveform, and thus not all values provided in the testbench of Figure 8 are taken into effect.

For example, between 0 and 90 ns, we have the following sequence of input values: 1, 0, 1, 1, 0, 1, 0, 1, 1, 0. However, only the odd-ordered inputs are considered, because they are getting introduced along the rising edge of the clock signal (which, by the way, happens for odd-numbered tens of nanoseconds : 10 ns, 30 ns, 50 ns, 70 ns, 90 ns, etc...). Thus, the above sequence is reduced to 1, 1, 0, 0, 1. Using the state diagram of Figure 1, this gives the following state sequence, starting from A: B, A, A, A, B, and thus `tb_o_mul3` will give the following sequence, starting with `z = 1` at 0 ns and with a step of 20 ns after `t = 10 ns`: 0, 1, 1, 1, 0. This can be clearly interpreted from the identical graphs represented above.

As a transitional idea, notice that at 100 ns, when `reset_n` was set to 0 for some infinitesimal time (1 ns), this is reflected by a reset of the state diagram of Figure 1 back to state A and `z = 1`, which is clearly observed in the graph as well.

Similarly, between 210 and 290 ns, `o_mul3` is at state B (between 190 and just before 210 ns) and we have the following sequence of input values: 0, 1, 1, 0, 0, 1, 1, 0. This is reduced to 0, 1, 0, 1, and thus `o_mul3` should have the following states: C, C, B, A, or 0, 0, 0, 1 in terms of bit-wise output values. This can be clearly deduced from the EPWave graphs presented above, hence verifying the superposition between the design's derivations and VHDL code files.

As a conclusion, we can now assert that our multiple-of-three sequence detector design is now complete, and proves to be correctly implemented thanks to a uniform testbench applied on two implementations of the same objective, leading to identical results and thus a finalized digital prototype.