Clock Dividers Made Easy

Mohit Arora

Design Flow and Reuse (CR&D)
ST Microelectronics Ltd
Plot No. 2 & 3, Sector 16A
Noida-201301, India
(www.st.com)

ABSTRACT

Dividing a clock by an even number always generates 50% duty cycle output. Sometimes it is necessary to generate a 50% duty cycle frequency even when the input clock is divided by an odd or non-integer number. This paper talks about implementation of unusual clock dividers. The paper starts up with simple dividers where the clock is divided by an odd number (Divide by 3, 5 etc) and then later expands it into non-integer dividers (Divide by 1.5, 2.5 etc). The circuits are simple, efficient and are cheaper and faster than any external PLL alternatives. This paper also covers Verilog code implementation for a non-integer divider.

INDEX

List of Figures

List of Tables

1. Introduction

In some designs, you need to provide a number of phase-related clocks to various components. In most cases, you generate the needed clocks by dividing a master clock by a power of two (synchronous division). However, sometimes, it is desirable to divide a frequency by an odd or even fractional divisor. In these cases, no synchronous method exists without generating a higher frequency master clock.

2. Simple clock divider where the input clock is divided by an odd integer

A synchronous divide by integer can be easily specified using a Moore machine. For example, Divide by 7.

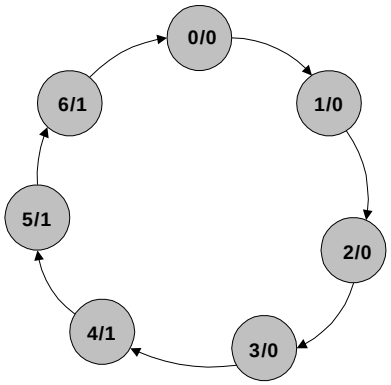

Figure 1: Divide by 7 using a Moore Machine

The above does not generate a 50% duty cycle.

3. Odd integer division with 50% duty cycle

Conceptually, the easiest way to create an odd divider with a 50% duty cycle is to generate two clocks at half the desired output frequency with a quadrature-phase relationship (constant 90° phase difference between the two clocks).
You can then generate the output frequency by exclusive-ORing the two waveforms together. Because of the constant 90° phase offset, only one transition occurs at a time on the input of the exclusive-OR gate, effectively eliminating any glitches on the output waveform.
Let's see how it works by taking an example where the reference clock is divided by 3.
Below are the sequential steps listed for division by an odd integer:

STEP I: Create a counter that counts from 0 to (N –1) and always clocks on the rising edge of the input clock where N is the natural number by which the input reference clock is supposed to be divided (N != Even)

For Divide by 3 : i.e. counts from 0 to 2 …N = 3
For Divide by 5 : i.e. counts from 0 to 4 …N = 5
For Divide by 7 : i.e. counts from 0 to 6 …N = 7

Note: *The counter is incremented on every rising edge of the input clock (ref_clk) and the counter is reset to ZERO when the terminal count of counter reaches to (N-1).*

STEP II: Take two toggle flip-flops and generate their enables as follows:

tff1_en : T FF1 enabled when the counter value = 0
tff2_en : T FF2 enabled when the counter value = (2 for Divide by 3 counter , 3 for Divide by 5 counter, 4 for Divide by 7 counter and likewise) as shown in Figure 2

STEP III : div1 : output of T FF1 → triggered on rising edge of input clock (ref_clk)
div2 : output of T FF2 → triggered on falling edge of input clock (ref_clk)

Note: *The output div1 and div2 of two T Flip flops generate the divide-by-2N waveforms as shown in Figure 2.*

STEP IV :
Final output clock: clkout (Divide by N) is generated by XORing the div1 and div2 waveforms.

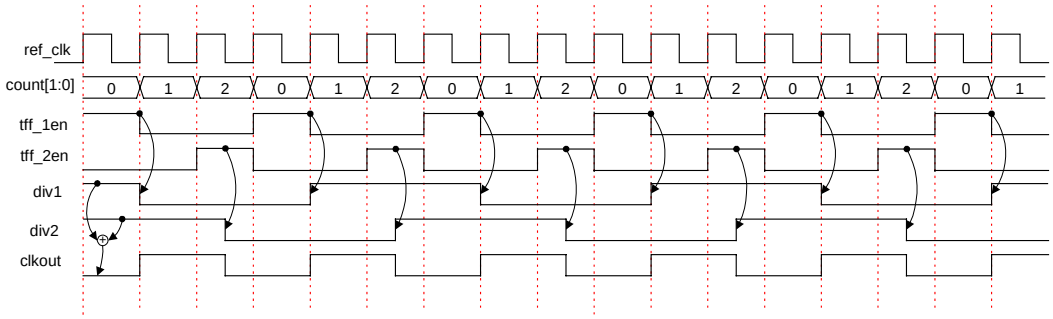

Figure 2: Timing diagram for Divide by 3 (N=2) with 50% duty cycle output

Complete circuit for Divide by 3 is shown on the next page.

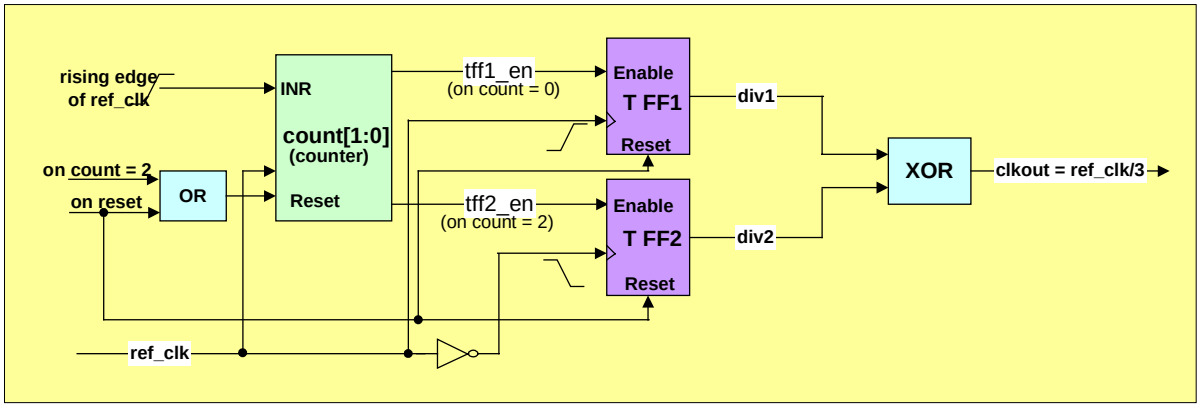

Figure 3: Divide by 3 using T flip-flop with 50 % duty cycle output

4. Non-integer division (duty cycle not 50%)

It's common practice to use a divide-by-N circuit to create a free-running clock based on another clock source. Designing such a circuit where N is a non-integer is not as difficult as it seems. First, consider what it means to Divide by 1.5. It simply means that, for every three clocks, you need to generate two symmetrical pulses

Lets start with a simple example where the clock is divided by 1.5 and the duty cycle is not 50 %.

4.1 Divide by 1.5 with duty cycle not exactly 50%

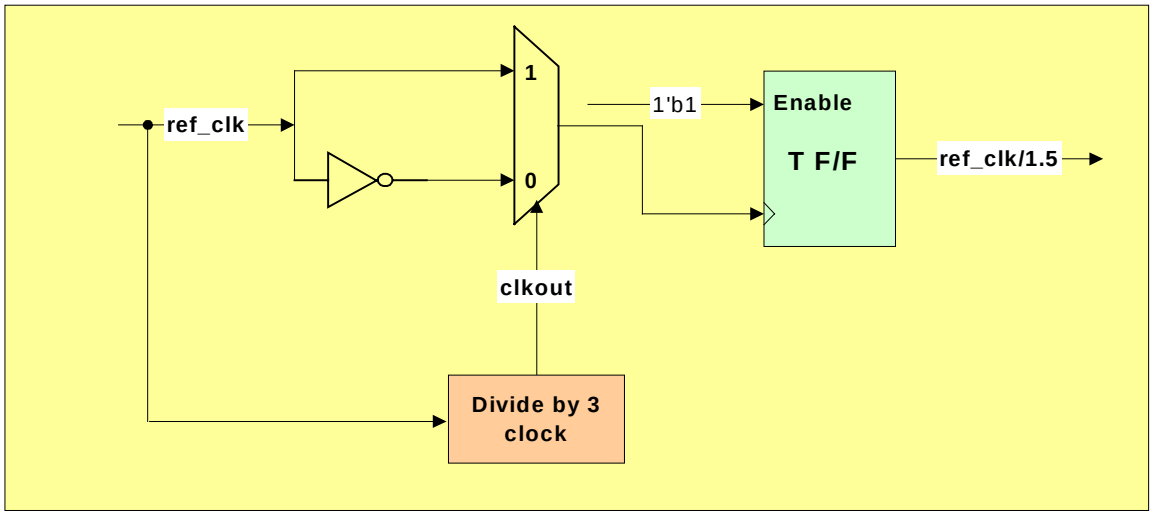

Figure 4: Divide by 1.5 using T flip-flop(Duty cycle not 50%)

The mux of Figure 4 selects the input clock when clkout is HIGH, otherwise it selects the inverted version of the input clock (ref_clk).

Figure 5 shows the timing for the Divide by 1.5 circuit shown in Figure 4

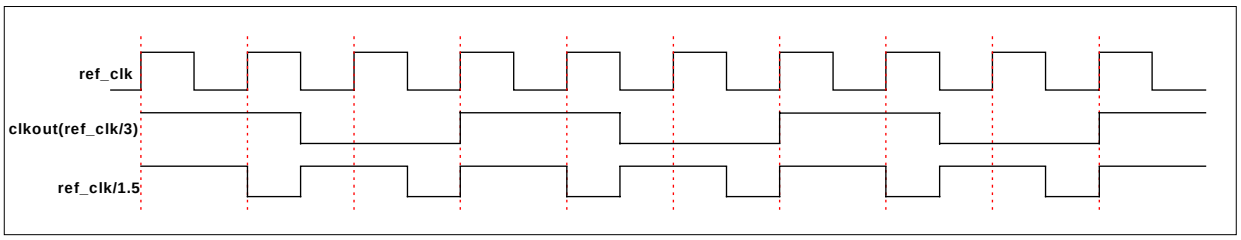

Figure 5: Timing diagram for Divide by 1.5 using T flip-flop (Duty Cycle not 50%)

Note: *The above circuit will work perfectly in simulation but might fail in synthesis due to the mux incorporated, which might introduce unequal delays when the select line of the mux toggles. The mux might not be able to change the output instantly and may produce gliches in the generated output clock.*

4.2 Divide by 4.5 with duty cycle not exactly 50% (counter implementation)

Let's go for an alternate approach where the circuit for divide by a non-integer is more optimized and the generated output clock is perfectly glitch-free.
Let's take an example with a Divide by 4.5. It means for every nine clocks, you need to generate two symmetrical pulses.
Below are the sequential steps listed for division by a odd integer:

STEP I: Take a 9-bit shift register and initialize it to 000000001 upon reset where the shift register is left-rotated on every rising clock edge.

STEP II: To generate the first pulse, you must shift the first bit by half a clock period and then OR it with the first and second bit.

STEP III: To generate the second pulse, you need to shift the fifth and sixth bits by half a clock period and then OR these bits with the original sixth bit.

Note: *All this shifting is necessary to ensure a glitch-free output waveform.*

Duty cycle for the above generated clock is 40% and it ensures glitch-free output.

Figure 6 shows the timing diagram for Divide by 4.5

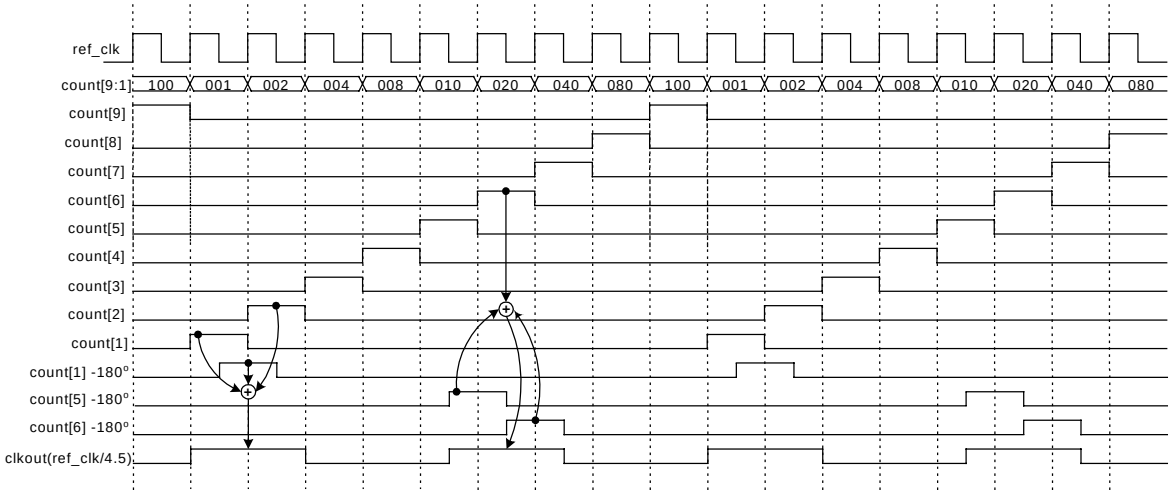

Figure 6:Timing diagram for counter implementation of Divide by 4.5 (duty cycle not 50%)

4.2.1 Verilog code for Divide by 4.5

Below is the Verilog code for the above generated Divide by 4.5 logic:

```
   /* Counter implementation
         reset value : 9'b000000001 */
always @( posedge ref_clk or negedge p_n_reset)
   if (!p_n_reset)
       count[9:1] <= 9'b000000001;
   else
       count[9:1] <= count[9:1] << 1;

/*count bit 1st, 5th and 6th phase shifter by 180o */
always @(negedge ref_clk or negedge p_n_reset)
       if (!p_n_reset)
          begin
            ps_count1 <= 1'b0;
            ps_count5 <= 1'b0;
            ps_count6 <= 1'b0;
          end
       else
          begin
            ps_count1 <= count[1];
            ps_count5 <= count[5];
            ps_count6 <= count[6];
           end

// generation of final output clock = (ref_clk /4.5)
assign clkout = (ps_count5 | ps_count6 | count[6]) |
                 (count[0] | count[1] | ps_count1);
```

---

5 Alternative approach for divide by-N

Each circuit assumes a 50% duty cycle of the incoming clock otherwise the fractional divider output will jitter and the integer divider will have unequal duty cycle.

All the circuits use combinatorial feedback around a LUT (look up table) that works perfectly and the output clock generated is glitch free.

Let's start up with a simple Divide by 1.5 circuit

5.1 Divide by 1.5 (LUT implementation)

Here again Divide by 1.5 is generated by first generating a Divide by 3 circuit. The two flip flops shown in Figure 7 form a Divide by 3 circuit (duty cycle not 50%).

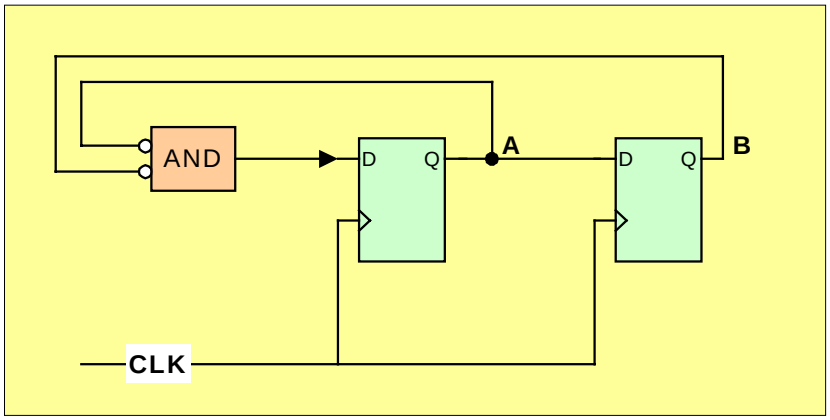

Figure 7: Divide by 3 (duty cycle not 50%)

Figure 8 shows the timing for the above circuit

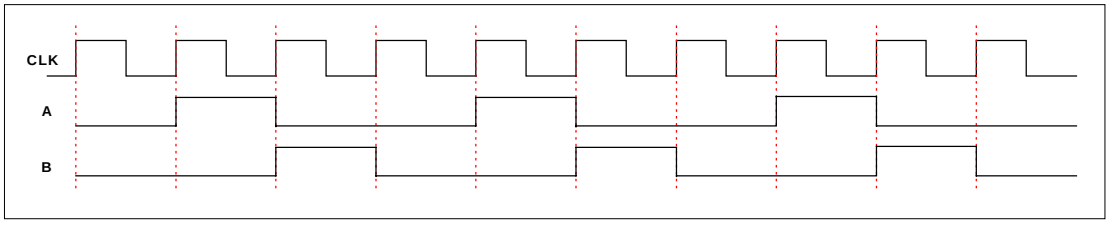

Figure 8:Timing diagram for Divide by 3 (duty cycle not 50%)

Now lets work out how the above circuit can be used to generate a Divide by 1.5 circuit.

Lets analyze the circuit given in Figure 9 :

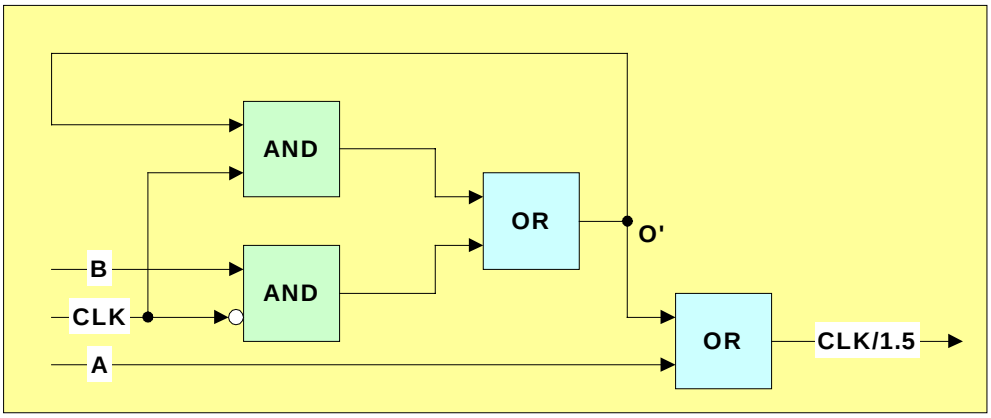

Figure 9: LUT Implementation for Divide by 1.5 (duty cycle output not 50%)

Here A and B inputs to the AND gate are from the outputs of the Divide by 3 circuit shown in Figure 7.
Figure 10 shows the overall timing for the above Divide by 1.5 circuit.

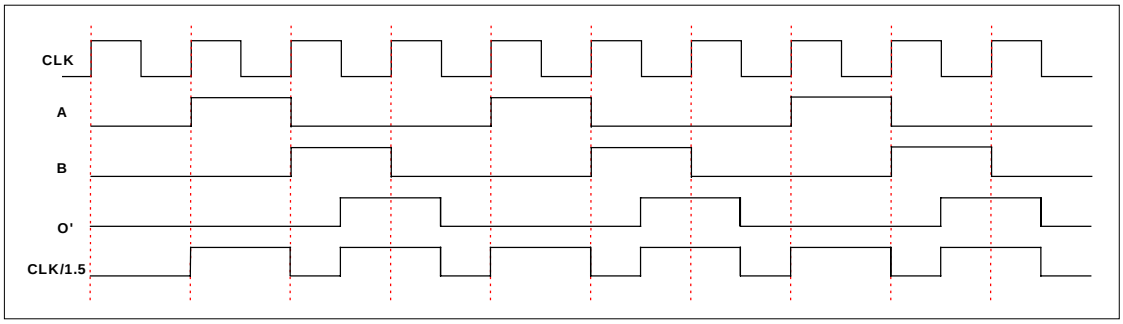

Figure 10: Timing diagram for Divide by 1.5 (LUT implementation)

Note: *The above circuit does not generate the output clock with 50% duty cycle. You cannot get anything better than 40%-60% with a digital circuit.*

A LUT implements only feedforward combinational logic and cannot hold any state information (feedback). In these implementations, is assumed that the LUT output is at O' and the feedback path forms one of the inputs of the LUT.

Proper functioning of such asynchronous sequential circuits depends on the fact that the transition of the CLK input to the LUT will always occur before the other inputs. Also differences in order of the transitions in the input do not disrupt the flow of states at the output and the correct waveform is always generated in relation to the input waveform A. This is

illustrated in Figure 11 where B, input to the LUT is slightly delayed with respect to the input clock.

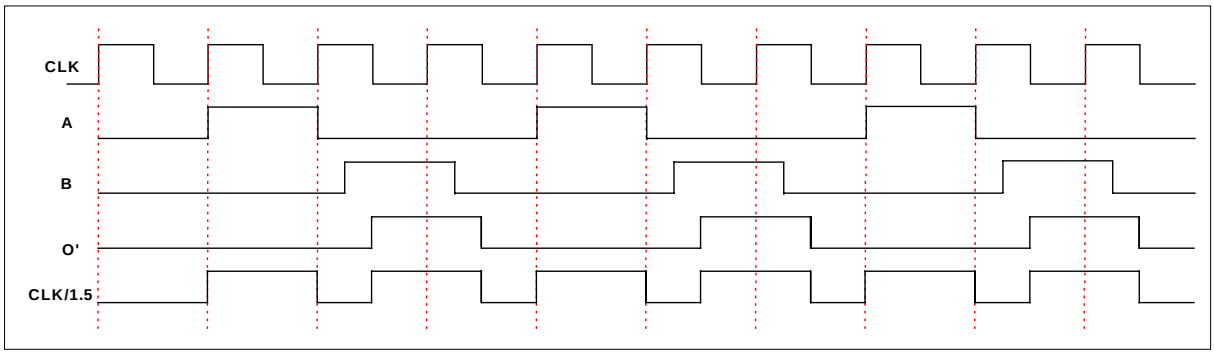

Figure 11: Timing diagram for Divide by 1.5 where input B is delayed with respect to CLK

Table 1 shows the LUT table and the output table for the Divide by 1.5 above.

CLK	B	O'	O'(LUT output)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

LUT Table

O'	A	CLK/1.5
0	0	0
0	1	1
1	0	1
1	1	1

Output Table

Table 1: LUT and the Output table for Divide by 1.5 circuit

5.2 Divide by 2.5 (LUT Implementation)

Here Divide by 2.5 is generated by first generating a Divide by 5 circuit.
The Divide by 5 circuit can be formed by three flip-flops as shown in Figure 12
Again here the duty cycle is not 50%.

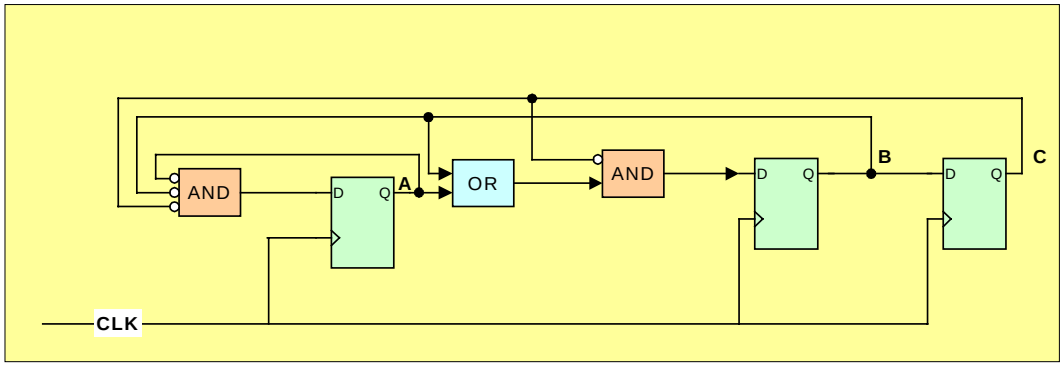

Figure 12: Divide by 5 (Duty cycle not 50%)

Figure 13 shows the timing for the above Divide by 5 circuit.

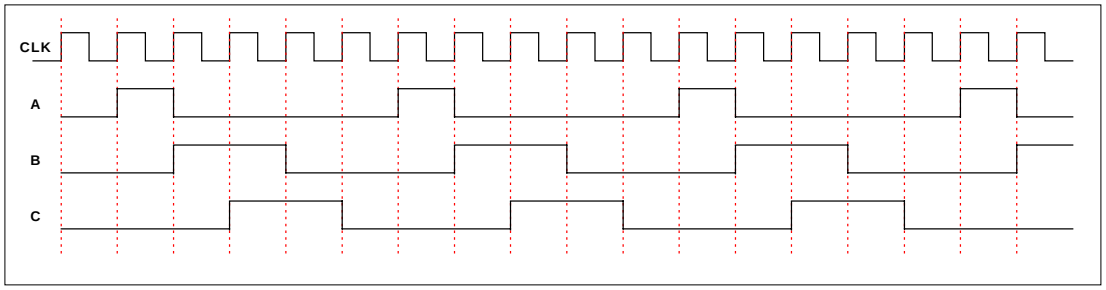

Figure 13: Timing diagram for Divide by 5 (duty cycle not 50%)

As seen from the timing diagram in Figure 13 that the flip-flop outputs B and C are both Divide by 5 where C is a one clock delayed version of B.

Divide by 2.5 can be generated by providing the outputs A, B and C (of the above Divide by 5 circuit) in the following way to the circuit shown below (Figure 14).

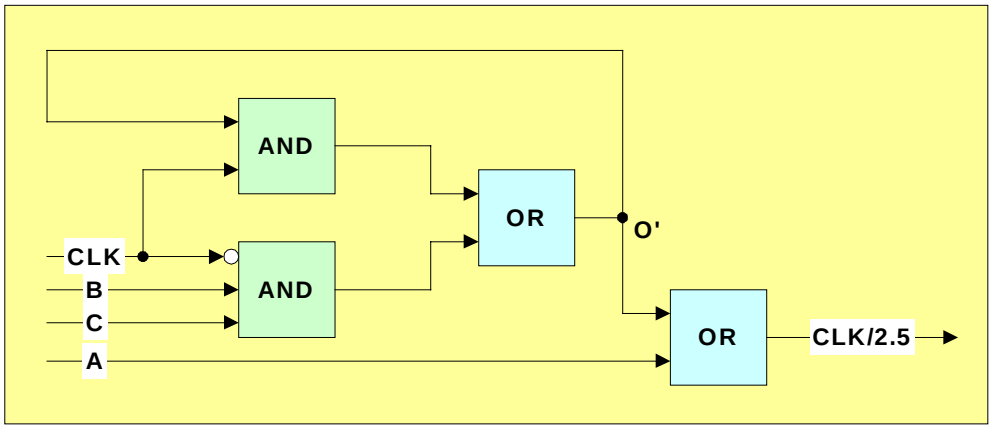

Figure 14: LUT Implementation for Divide by 2.5 (duty cycle output not 50%)

Figure 15 shows the overall timing for the above Divide by 2.5 circuit.

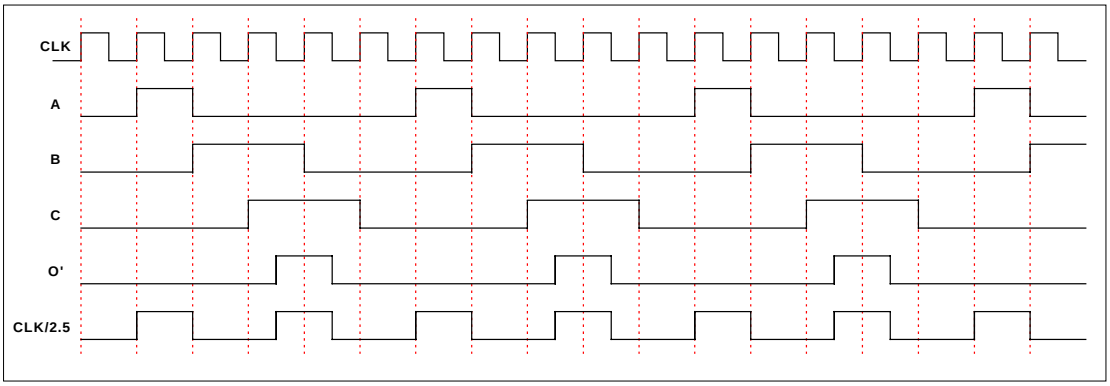

Figure 15: Timing diagram for Divide by 2.5 (LUT implementation)

In this design, the first output pulse is driven by A flip-flop, second output pulse is derived from the B AND C signal but is delayed by half the input clock (CLK).

Note: *Here in Figure 15 the output is sensitive to clock duration rather than clock transition This Latch based circuit might cause static timing analysis problem due to the combinational feedback loop.*

Table 2 shows the LUT table and the output table for the Divide by 2.5 above.

CLK	B	C	O'	O'(LUT Output)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

LUT Table

O'	A	CLK/2.5
0	0	0
0	1	1
1	0	1
1	1	1

Output Table

Table 2: LUT and the Output table for Divide by 2.5 circuit

5.3 Divide by 3 with 50% duty cycle output

The same circuit used in 5.1 (Divide by 1.5) can be used with a little modification as shown in Figure 16

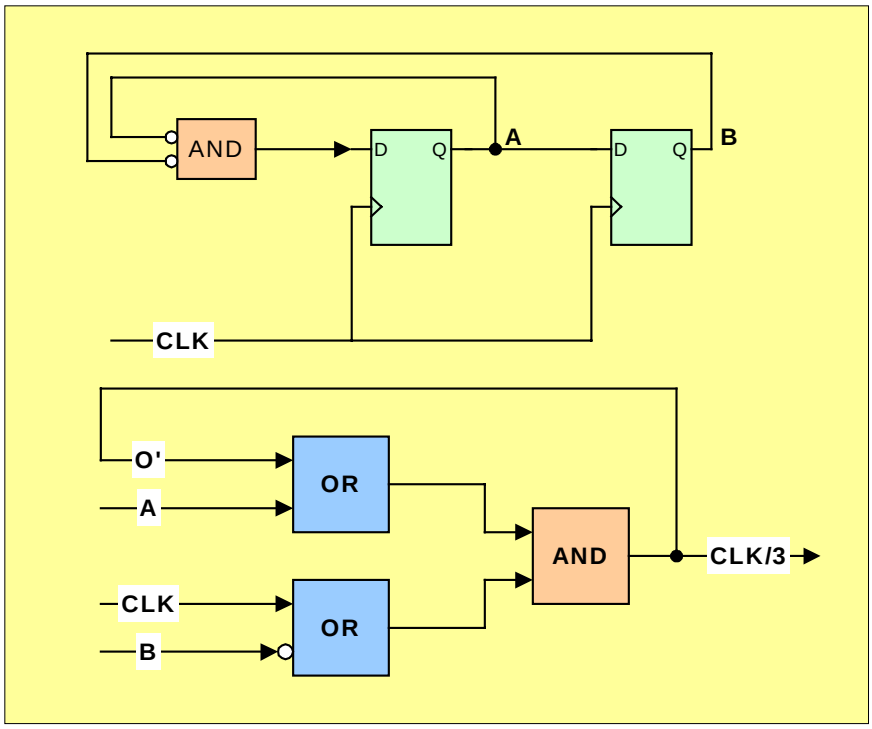

Figure 16: LUT Implementation for Divide by 3 (50% duty cycle output)

Figure 17 shows the timing diagram for the above Divide by 3 circuit.

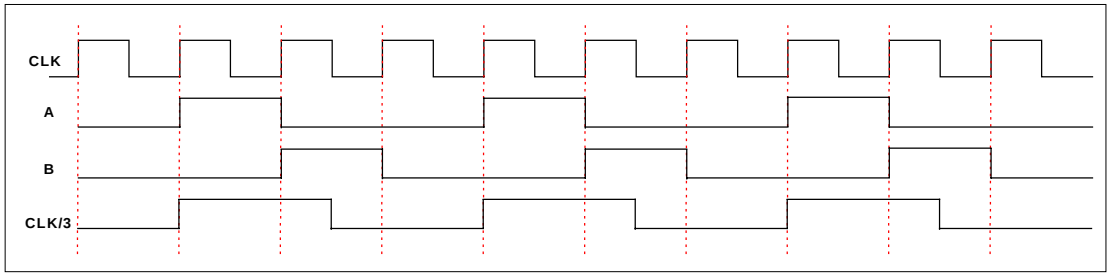

Figure 17: Timing diagram for Divide by 3 (LUT implementation)

The first output pulse is started by the A flip-flop and terminated by the B flip-flop when the clock (CLK) is low.

Table 3 shows the LUT table output for the Divide by 3 above.

CLK	B	A	O'	CLK/3
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 3: LUT table output for Divide by 3 circuit

5.4 Divide by 5 with 50% duty cycle output

The same circuit used in 5.2 (Divide by 2.5) can be used with a little modification as shown in Figure 18

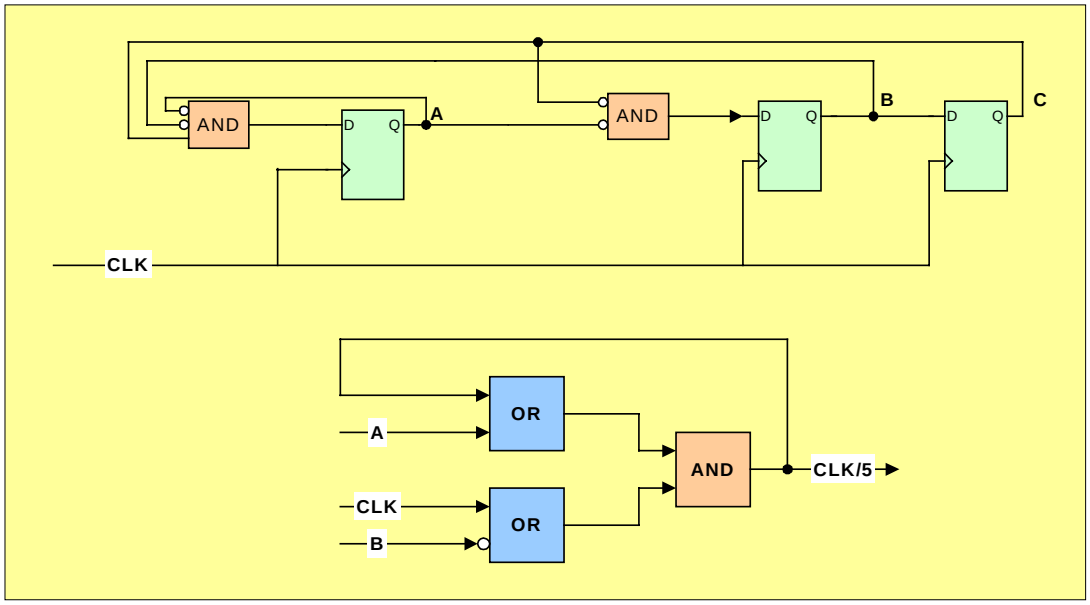

Figure 18: LUT Implementation for Divide by 5 (50% duty cycle output)

Figure 19 shows the timing diagram for the above Divide by 5 circuit.

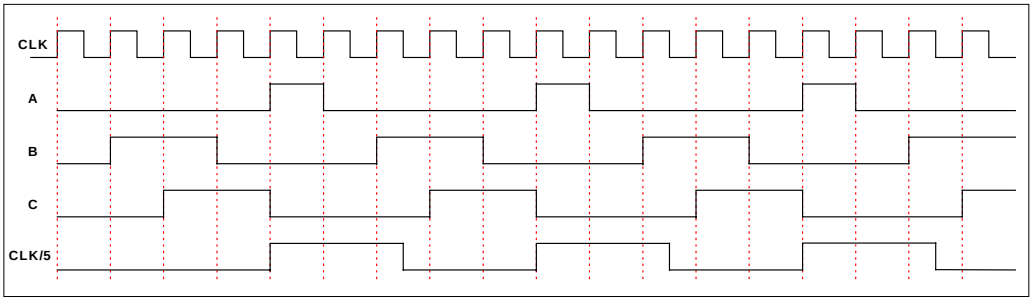

Figure 19: Timing diagram for Divide by 5 (LUT implementation)

Here above the first output pulse is started by the A flip-flop and terminated by the B flip-flop when the clock is low.

Table 4 shows the possible valid combinations for the Divide by 5 above.

CLK	A	B	C	CLK/5
1	0	0	0	1
0	0	0	0	1
1	0	1	0	1
0	0	1	0	0
1	0	1	1	0
0	0	1	1	0
1	0	0	1	0
0	0	0	1	0
1	1	0	0	1
0	1	0	0	1

Table 4: LUT output table for Divide by 5 circuit

6. Conclusions

In the end, it can be summarized that following the techniques given in this paper a design engineer can implement clock dividers in various ways but to choose the one which is optimized and produces glitch free output clock. LUT implementation discussed above might create some simulation problems due to its combinational feedback path. One should also avoid any sort of multiplexers in selection between the clock edges to generate the output clock since it might produce glitches at the output making the circuit fail at the gate level.
In the case of non-integer clock dividers (Divide by 1.5, Divide by 2.5 etc) one cannot get anything better than 40%-60% duty cycle with a digital circuit.

7. Acknowledgements

I would like to thank one and all who helped in the successful writing of this paper. I am indeed grateful to Mr. Clifford E. Cummings for helping me in reviewing the document and providing valuable feedback and comments in a short time.

8. References

[1].”Unusual Clock Dividers” by Peter Alfke,Xilinx Applications
[2]. EDN ACCESS “VHDL code implements 50% duty cycle divider” by Brain Boorman,Harris RF Communications,Rochester,NY
[2]. Samir Palnitkar, Verilog HDL, A Guide to Digital Design and Synthesis, Sunsoft Press A Prentice Hall.

[3]. Digital Design by Morris Mano

9. Author & Contact information

For any further information please contact:

Mohit Arora (mohit.arora@st.com)
Design Engineer