Cómo hacer un antirrebotes con un biestable JK en una FPGA

Creador: David Rubio G.

Entrada: https://soceame.wordpress.com/2024/12/17/como-hacer-un-antirrebotes-con-un-biestable-jk-en-una-fpga/

Blog: https://soceame.wordpress.com/

GitHub: https://github.com/DRubioG

Fecha última modificación: 23/02/2025

Un antirrebotes es un sistema que se utiliza para evitar rebotes en la señal de entrada de una FPGA, estos rebotes son importantes de cara la utilización de máquinas de estados con señales provenientes del exterior de la FPGA.

Es importante entender que un rebote suele tener una duración de unos 10-20ns, eso es importante de cara al número de ciclos de reloj que se tienen que analizar en el sistema antirrebotes.

Muchas veces se ve un antirrebotes utilizando una máquina de estados, eso implica que para entrada que necesite un sistema antirrebotes se utilice una máquina de estados, lo que resulta al final un consumo mayor de recursos. Para ello existen otros métodos más basados en lógica combinacional (o sea, puertas lógicas).

Antirrebotes basado en un biestable JK

La ventaja de este antirrebotes es que para hacer el antirrebotes utiliza más puertas lógicas normales. Es por ello que un biestable JK se puede modelar mediante puertas lógicas. Además, solo se requiere de unos pocos biestables D, los suficientes para garantizar que se puede medir el pulso de entrada.

Biestable JK

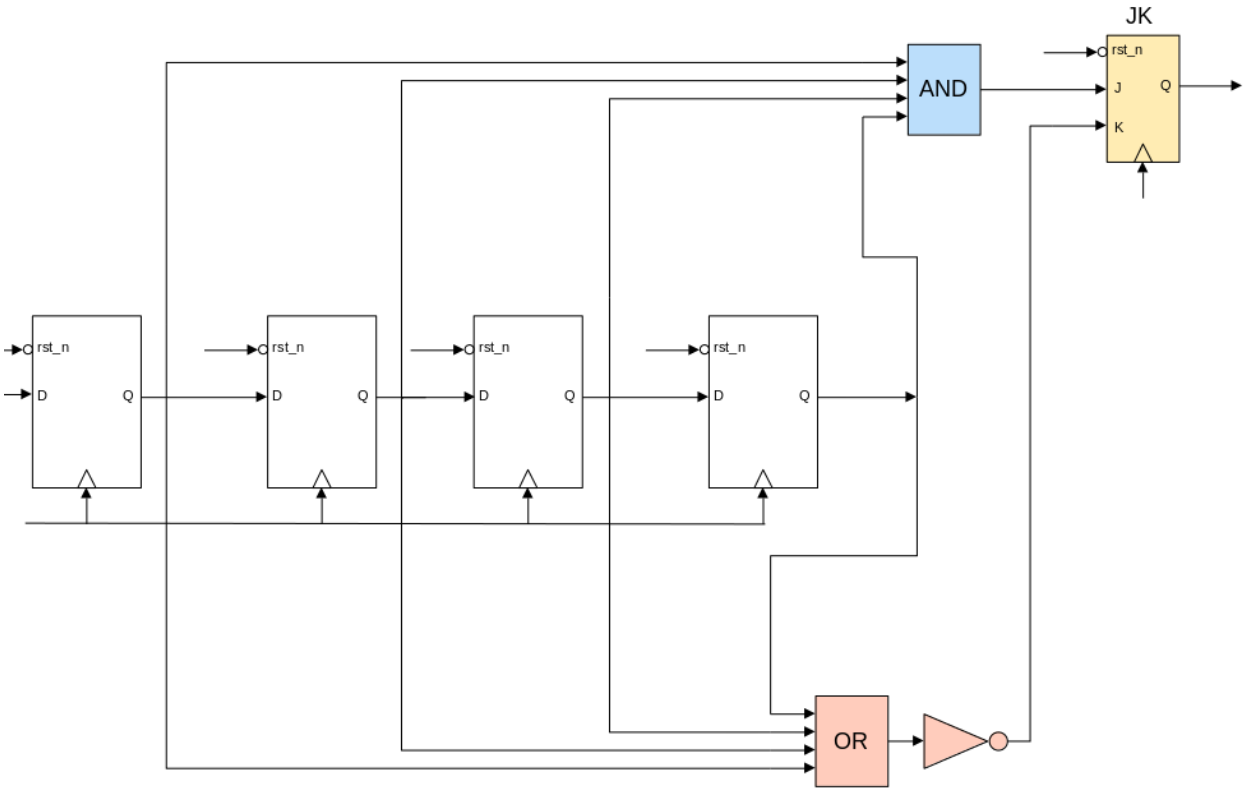
Un biestable JK tiene una tabla de la verdad siguiente:

Flanco	**J**	**K**	Q
(subida)	0	0	*(se mantiene la salida)*
(subida)	0	1	0
(subida)	1	0	1
(subida)	1	1	*(se alterna la salida)*

El uso de un biestable JK es para mantener el valor a la salida en caso de que J sea ‘0’ y que K sea ‘0’.

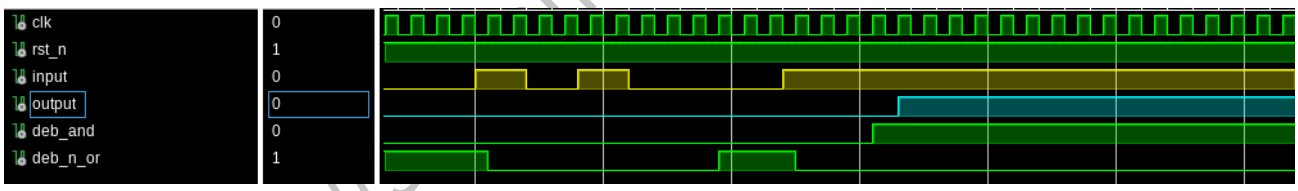
El modelo básico de un antirrebotes con un biestable JK es el siguiente.

NOTA: la OR negada se puede cambiar por una NOR, pero VHDL no permite hacer una NOR en cadena con otra NOR, solo permite dos entradas por puerta lógica, como sí que permite con las AND, entonces, queda reflejado quedaría implementado en VHDL *(si quieres utilizar una NOR puede utilizar una nor_reduce que si permite más de dos entradas)*.

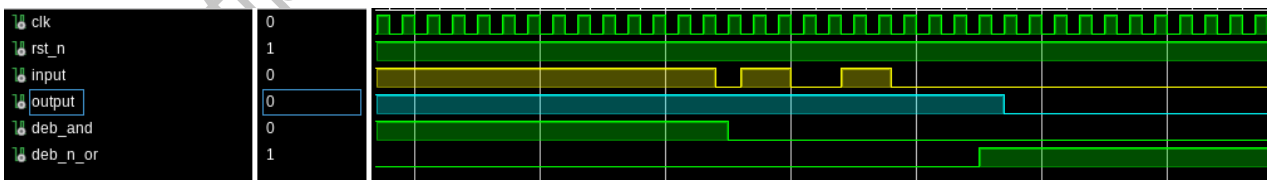


En este modelo se basa en comprobar el estado de los biestables D de entrada.

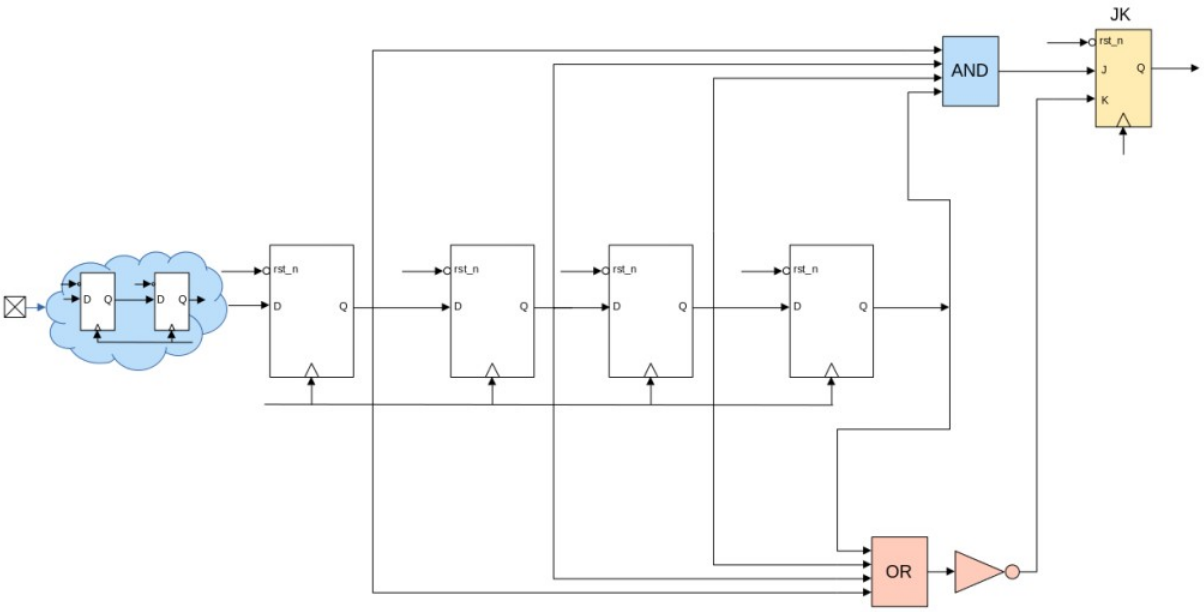
- **Para el cambio a nivel alto**, se tienen que dar las condiciones favorables de que la AND sea todo ‘1’s y que que la OR negada sea ‘0’, así el biestable JK pone la salida a ‘1’.



- **Para el cambio a nivel bajo**: se tienen que dar las condiciones favorables de que la OR negada sea ‘1’, y la AND sea ‘0’, así el biestable JK pone a la salida un ‘0’.



Por último es conveniente recordar que cualquier entrada aunque se le ponga un sistema antirrebotes necesita que esa entrada esté sincronizada con el reloj para evitar posibles metaestabilidades causadas en la entrada del antirrebotes.



Ejemplo

Aquí dejo un ejemplo de cómo se implementaría.

NOTA: esto es un ejemplo «educativo», para poder utilizarlo a alto nivel sería necesario hacer mejoras, como encapsular el sincronismo en un módulo y el biestable JK también en su propio módulo, también falta un reset síncrono, y por último, se podría utilizar una *and_reduce* y un *or_reduce*. También se puede cambiar la OR negada por una *nor_reduce*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity antidebounce is
    Port (
        clk : in std_logic;
        rst_n : in std_logic;
        input : in std_logic;
        output : out std_logic
    );
end antidebounce;

architecture Behavioral of antidebounce is

signal input_sync1, input_sync2 : std_logic;

signal d_ff1, d_ff2, d_ff3, d_ff4 : std_logic;
signal J, K : std_logic;
signal Q : std_logic;

begin

----------------SYNC---------------------------
SYNC : process(clk, rst_n)
    begin
        if rst_n = '0' then
```

```
            input_sync1 <= '0';
            input_sync2 <= '0';
        elsif rising_edge(clk) then
            input_sync1 <= input;
            input_sync2 <= input_sync1;
        end if;
    end process;
-----------------------------------------


D_FF: process(clk, rst_n)
    begin
        if rst_n = '0' then
            d_ff1 <= '0';
            d_ff2 <= '0';
            d_ff3 <= '0';
            d_ff4 <= '0';
        elsif rising_edge(clk) then
            d_ff1 <= input_sync2;
            d_ff2 <= d_ff1;
            d_ff3 <= d_ff2;
            d_ff4 <= d_ff3;
        end if;
    end process;


AND_L  : J <= d_ff1 and d_ff2 and d_ff3 and d_ff4;
N_OR_L : K <= not (d_ff1 or d_ff2 or d_ff3 or d_ff4);


JK_FF : process(clk, rst_n)
    begin
        if rst_n = '0' then
            Q <= '0';
        elsif rising_edge(clk) then
            if J = '0' and K = '0' then
                Q <= Q;
            elsif J = '0' and K = '1' then
                Q <= '0';
            elsif J = '1' and K = '0' then
                Q <= '1';
            elsif J = '1' and K = '1' then
                Q <= not Q;
            end if;
        end if;
    end process;

    output <= Q;
end Behavioral;
```