MEMORIAS

Prácticas de VHDL

María González Herrero

Santiago Molpeceres Díaz

UNIVERSIDAD NEBRIJA

Índice

Práctica 1: Activar un LED mediante un botón

Para esta primera práctica vamos a comprender en profundidad el correcto funcionamiento de un botón, para ello lo vamos a implementar para que se encienda un LED por sync y debouncer. El sistema antirrebotes (debouncer), se encargará de limpiar una señal exterior y comprobar ambas veces si ha ocurrido algún error. Después también tendremos un sincronizador, el cual hará que esa señal que ya tenemos de activación limpia proveniente de debouncer y la pondrá al mismo tiempo que el reloj del sistema para que éste la pueda leer. Por último, tendremos la parte 'top' en la cual se juntarán todos los componentes que hemos mencionado antes, para poder implementarlo en nuestra FPGA.

Debouncer

Nuestro sistema antirrebotes está formado por dos componentes principales, un contador y una máquina de estados. El contador se activará o reiniciará en algunos estados de la maquina hasta poder devolver la señal de contado completa que la FSM tendrá en cuenta estos valores.

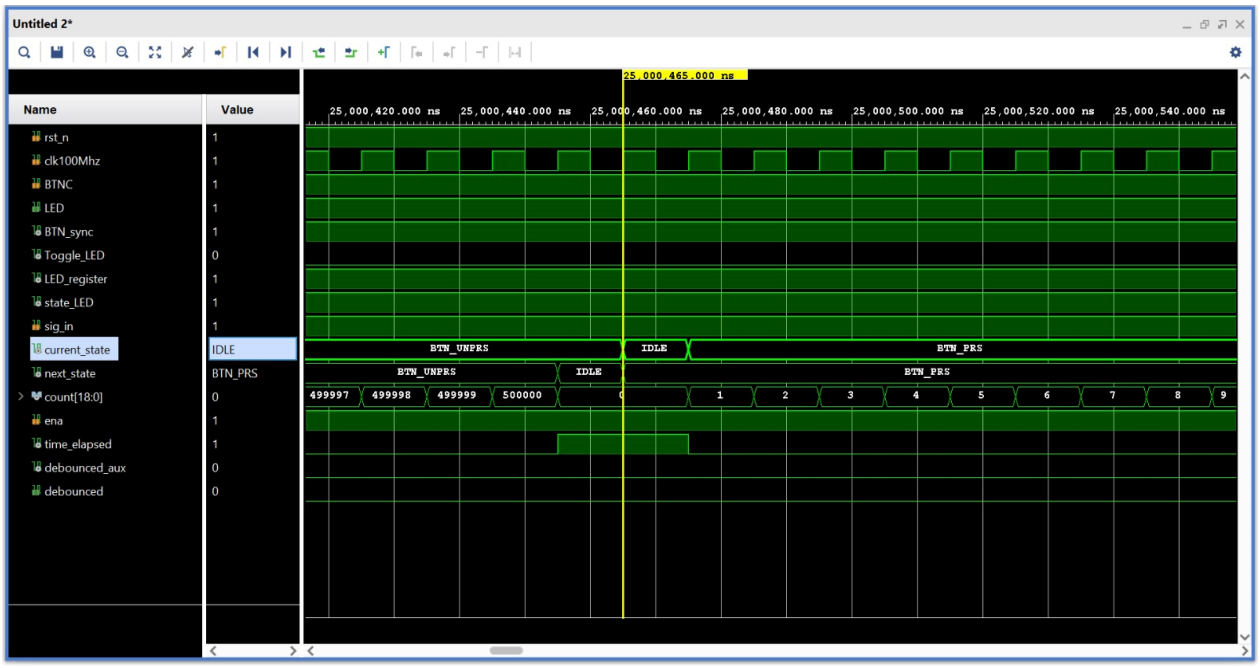
FSM

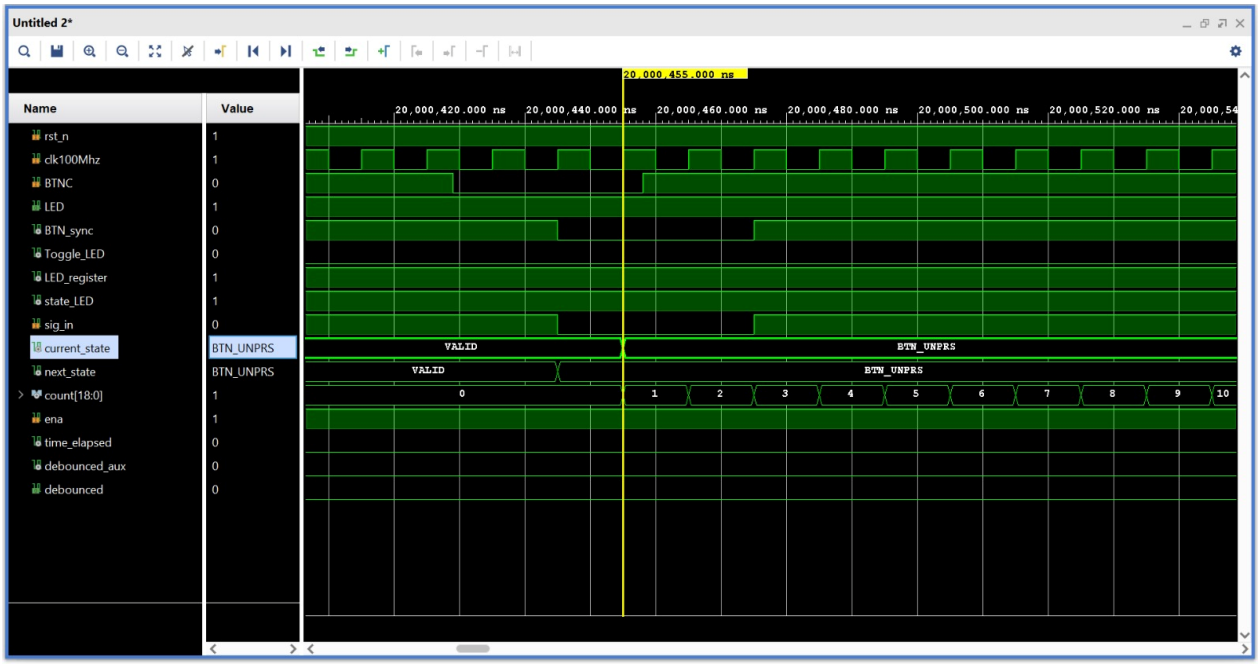
En el estado Idle se reiniciará el contador siempre a 0, ya que es desde este punto el que contamos como inicio del evento. Esta misma operación la haremos de la misma forma para Valid, ya que eso significara que ya hemos hecho la comprobación de haber presionado dos veces el botón una vez al iniciar el estado y otra al terminar el contador, por lo que es por eso que se necesita reiniciar el contador.

En el estado en el cual teníamos pulsado el botón, sumamos uno al contador en cada ciclo y si el siguiente estado no el VALID y el contador aun no ha llegado al límite predefinido, en caso contrario se reiniciará y se activará la bandera de time_elapsed. Finalmente, en el top llamaremos a todos los módulos.

Una vez hechos todos los registros los valores del LED para poder sacarlos como la señal exterior. En otro proceso cambiaremos el valor del LED real según el valor del toggle unido a la salida de conformación del debouncer.

Testbench


Untitled 2*
Name
Value
rst_n
clk100Mhz
BTNC
LED
BTN_sync
Toggle_LED
LED_register
state_LED
sig_in
current_state
next_state
count[18:0]
ena
time_elapsed
debounced_aux
debounced
IDLE
BTN_PRS
25,000,465.000 ns
25,000,420.000 ns
25,000,440.000 ns
25,000,460.000 ns
25,000,480.000 ns
25,000,500.000 ns
25,000,520.000 ns
25,000,540.000 ns
BTN_UNPRS
IDLE
BTN_PRS
499997
499998
499999
500000


Untitled 2*
Name
Value
rst_n
clk100Mhz
BTNC
LED
BTN_sync
Toggle_LED
LED_register
state_LED
sig_in
current_state
next_state
count[18:0]
ena
time_elapsed
debounced_aux
debounced
BTN_UNPRS
BTN_UNPRS
20,000,455.000 ns
20,000,420.000 ns
20,000,440.000 ns
20,000,460.000 ns
20,000,480.000 ns
20,000,500.000 ns
20,000,520.000 ns
VALID
BTN_UNPRS
VALID
BTN_UNPRS

Ahora mediante la simulación del testbech, podemos comprobar cómo se comporta nuestra máquina, empezando por poner el reset a nivel bajo y así se irán a iniciar todas las variables a su estado o valor correspondientes.

El primer estado es el de Idle, el cual es un estado de espera que su función es la de esperar que btn se pulse y espera a que se propague el btn_sycrn y eso significa que vamos a cambiar de estado.

Una vez que hayamos cambiado de estado, como podemos observar en las imágenes el contador empieza a subir y hasta que no llegue hasta 500.000 pulsos (tiempo máximo que hemos declarado, en el que va a cambiar de estado), no puede cambiar al siguiente estado que es el de VALID.

En el estado de ‘valid’ el debounce se mantiene a 1 un solo pulso y para poder logarlo usamos debounce_aux para poder propagar esa señal. Cuando dejamos de pulsar el botón, este se propaga y hasta que sigin no valga 0 en un ciclo de reloj a nivel alto, no nos vamos a actualizar a botón_presc y mantenemos como antes esos 500.000 ciclos, hasta volver al estado de Idle.

Práctica 2: Funcionamiento de la interfaz PDM

En esta segunda práctica tenemos el objetivo de crear una señal digital PDM de una señal de audio. Para ellos debemos de crear dos módulos, el primero será un FSCLK, es decir, un reloj síncrono de menor frecuencia al del sistema y un shift register, el cual pasará los datos de serie a paralelo para guardarlo en el registro.

Para ello primero analizaremos como se comporta este módulo (FSCLK), primero debemos de definir la frecuencia del sistema y la velocidad a la que queremos que vaya el reloj de salida del módulo.

En este caso predefinimos los genéricos con los valores que vamos a utilizar. Como utilizamos una frecuencia del sistema a 100MHz y el reloj objetivo es de 1,5MHz.

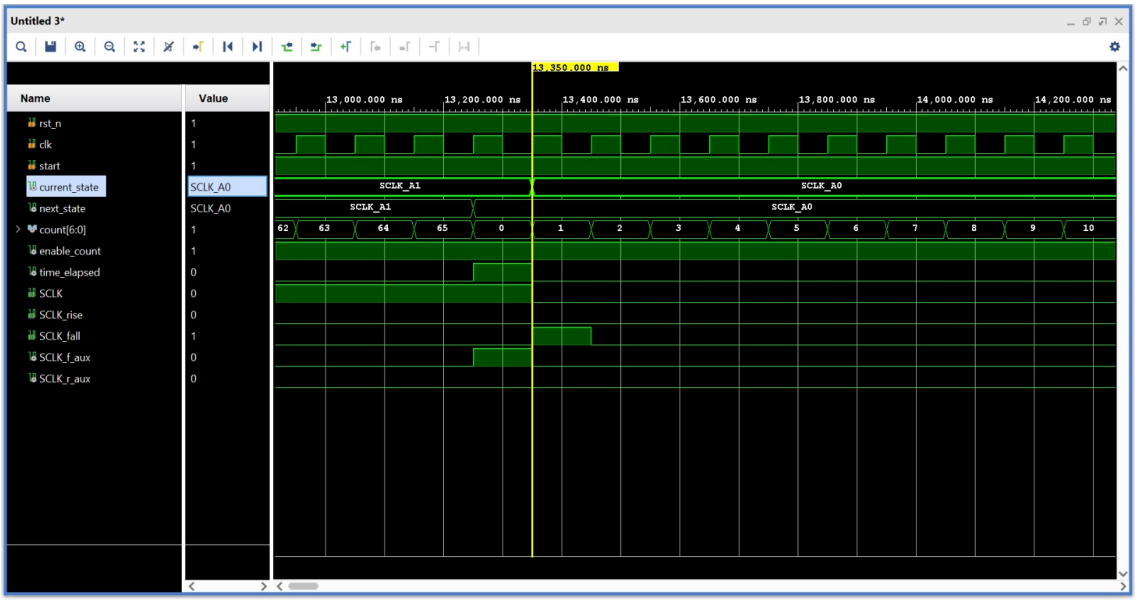
Tendremos que hacer un controlador con el reloj del sistema el cual con cada activación hará que el toggle se active con el reloj generado. Para poder saber la longitud de ciclos (del reloj original) del reloj a generar hay que dividir ambos relojes y el resultado obtenido será el tiempo de espera del contador.

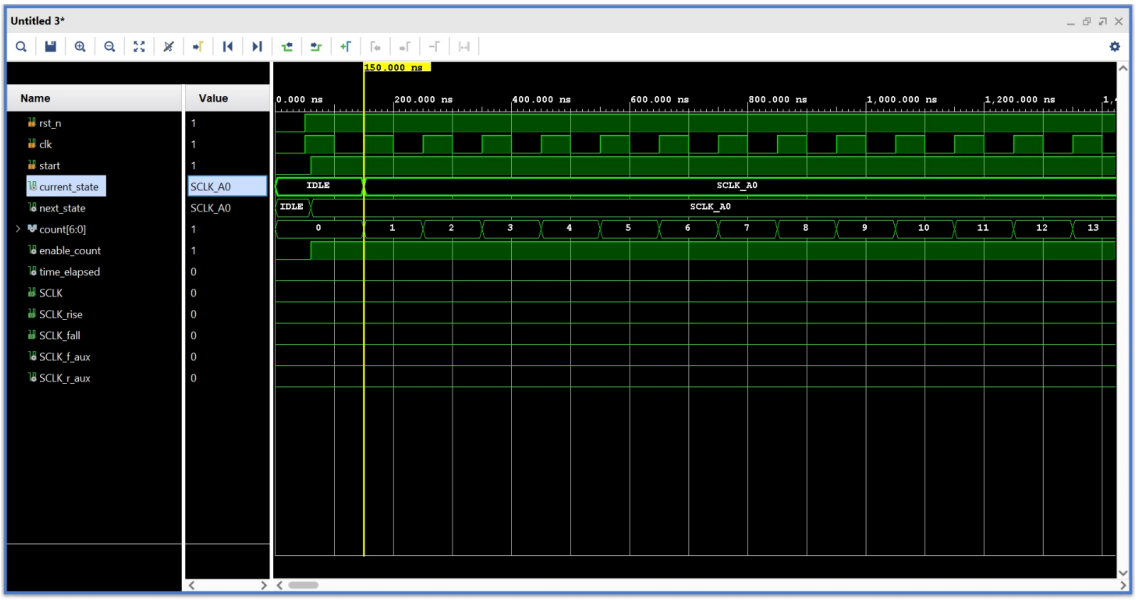
Con todo esto creado ahora ya podemos crear la máquina de estados con la que se podrá llevar la cuenta, siguiendo la marca del contador si debemos de subir o de bajar el pulso de reloj generado

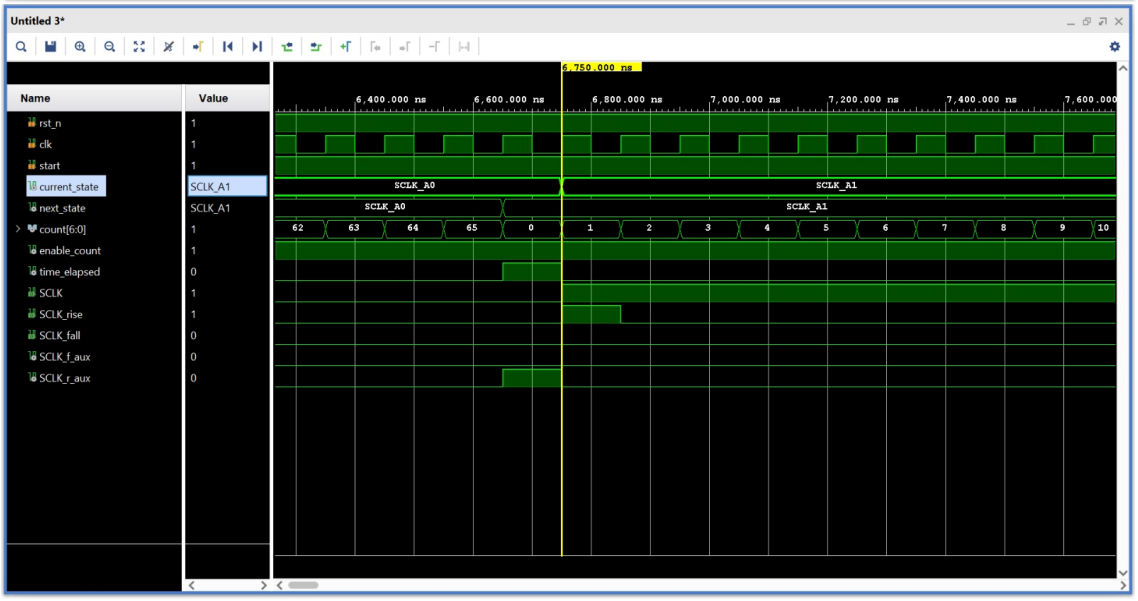
```
Desplazar: process (clk,rst_n,SEL) begin
if (rst_n ='0')then
    desplazador <= (others => '0');
else
    if(rising_edge (clk) )then
        case SEL is
        when "00"   =>
            desplazador <=desplazador;
        when "01"   =>
            desplazador<= din &desplazador(N-1 downto 1);
        when "10"   =>
            desplazador<= desplazador(N-2 downto 0) & din;
        when "11"   =>
            desplazador <= D;
        when others =>
            desplazador <= (others => '0');
        end case;
    end if;
end if;
end process;
```

En nuestra maquina de estados dependerá en todo momento de SEL, el cual actualizaremos o no de cierta manera según el valor que estemos pasando del sel.

Testbench


Untitled 3*
Name
Value
rst_n
clk
start
current_state
next_state
count[6:0]
enable_count
time_elapsed
SCLK
SCLK_rise
SCLK_fall
SCLK_f_aux
SCLK_r_aux
SCLK_A0
SCLK_A1
13,350.000 ns
13,000.000 ns
13,200.000 ns
13,400.000 ns
13,600.000 ns
13,800.000 ns
14,000.000 ns
14,200.000 ns


Untitled 3*
Name
Value
rst_n
clk
start
current_state
next_state
count[6:0]
enable_count
time_elapsed
SCLK
SCLK_rise
SCLK_fall
SCLK_f_aux
SCLK_r_aux
IDLE
SCLK_A0
150.000 ns
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
1,000.000 ns
1,200.000 ns


Untitled 3*
Name
Value
rst_n
clk
start
current_state
next_state
count[6:0]
enable_count
time_elapsed
SCLK
SCLK_rise
SCLK_fall
SCLK_f_aux
SCLK_r_aux
SCLK_A0
SCLK_A1
6,750.000 ns
6,400.000 ns
6,600.000 ns
6,800.000 ns
7,000.000 ns
7,200.000 ns
7,400.000 ns
7,600.000

Vamos a analizar estas capturas de pantalla, para empezar, tenemos un reset a nivel bajo, que hace que inicialice todas las variables y señales al valor que le hemos declarado que debe de tener. Como podemos ver en las imágenes el reset esta en 1, eso significa que es combinacional y que le decimos al current state que queremos cambiar de estado. Esto quiere decir que cuando tengamos la señal start a 1 Idle va a cambiar de estado.

En ese momento empezará un contador (número de ciclos que implica nuestro retraso) a contar, es decir, para poder lograr los Hz necesarios tenemos que mantener nuestro sistema 66 ciclos en pausa (el contador se mantiene hasta que no valga 66).

Una vez pasado el tiempo de espera, time elapsed pasará a nivel alto, es decir, a 1 junto a SCLK_r_aux para que así la señal se pueda mantener a 1 en el siguiente ciclo de reloj. Como resultado de estos cambios el reloj secundario se mantendrá a 1 en los siguientes 66 ciclos de reloj.

Práctica 3: Automatización del Testbench

En esta práctica vamos a automatizar el testbench del top de nuestra primera práctica, para ello lo primero que debemos hacer el leer un archivo .csv, (proporcionado por el profesor) en el cual vamos a obtener el valor de las señales que aplicaremos en nuestro testbench.

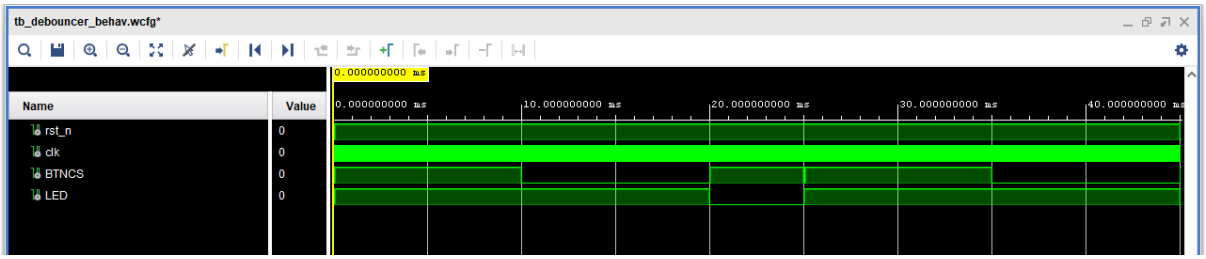
Los pasos que vamos a seguir para poder hacerlo se pueden asemejar a otros lenguajes de programación con los que ya estamos familiarizados como C, para poder leer y escribir un archivo.

Lo primero será iniciar la lectura de nuestro archivo con las variables a leer y creemos las variables para utilizarlas en nuestro programa.

Primero debemos de comprobar si existe el archivo al cual escribiremos el resultado final y si no es así, pues lo crearemos. También debemos de indicar que debe de saltar las líneas vacías y los comentarios. Y así ya podemos asignar cada variable en el mismo orden en el que están en el archivo .csv de input.

Dependiendo si el valor que obtuvimos del LED es correcto o no, debemos de escribir un mensaje que indique que está ocurriendo. Y así, podremos comprobar el valor obtenido en un diagrama con los outputs del .csv.

Al principio del diagrama vemos que hay al poco de iniciar un momento en el cual nuestra señal del botón se encontrará a 0, que cuadrará con los primeros 20ns, después de ese tiempo, se activará el botón y el LED se encenderá. De la misma manera cuando en los siguientes 0,01 segundos se deje de pulsar el botón, no hará ningún cambio visible, pero esto se debe a como esta configurado nuestro toggle del led en la practica 1.



Durante los próximos 0,5 segundos el botón se volverá a activar y en el imput del .csv se espera el valor de 1 en el led por lo que se devuelve un error. Después el botón se pulsa durante los siguientes 10ns haciendo que este pase a valer 0 y no hace ningún cambio como pasó antes.

Simulación de la practica 1

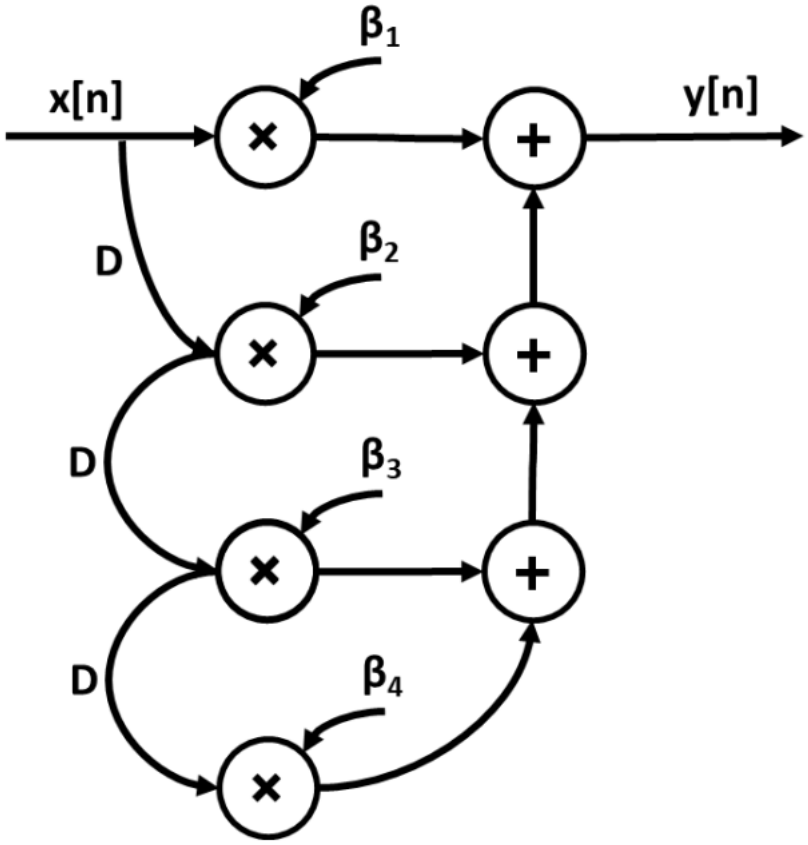
```
delay: 10 ns
rst_n: 0
BTNC: 0
Valor correcto de LED 0


delay: 10 ns
rst_n: 1
BTNC: 0
Valor correcto de LED 0


delay: 10000000 ns
rst_n: 1
BTNC: 1
Valor correcto de LED 1


delay: 10000000 ns
rst_n: 1
BTNC: 0
Valor correcto de LED 1


delay: 5000000 ns
rst_n: 1
BTNC: 1
Valor esperado de LED 1
Valor obtenido de LED 0


delay: 10 ns
rst_n: 1
BTNC: 0
Valor esperado de LED 1
Valor obtenido de LED 0

delay: 10000000 ns
rst_n: 1
BTNC: 1
Valor esperado de LED 0
Valor obtenido de LED 1


delay: 10000000 ns
rst_n: 1
BTNC: 0
Valor esperado de LED 0
Valor obtenido de LED 1


delay: 10 ns
rst_n: 0
BTNC: 1
Valor correcto de LED 0


delay: 10 ns
rst_n: 0
BTNC: 1
Valor correcto de LED 0
```

Practica 4: Conexión Software a FPGA

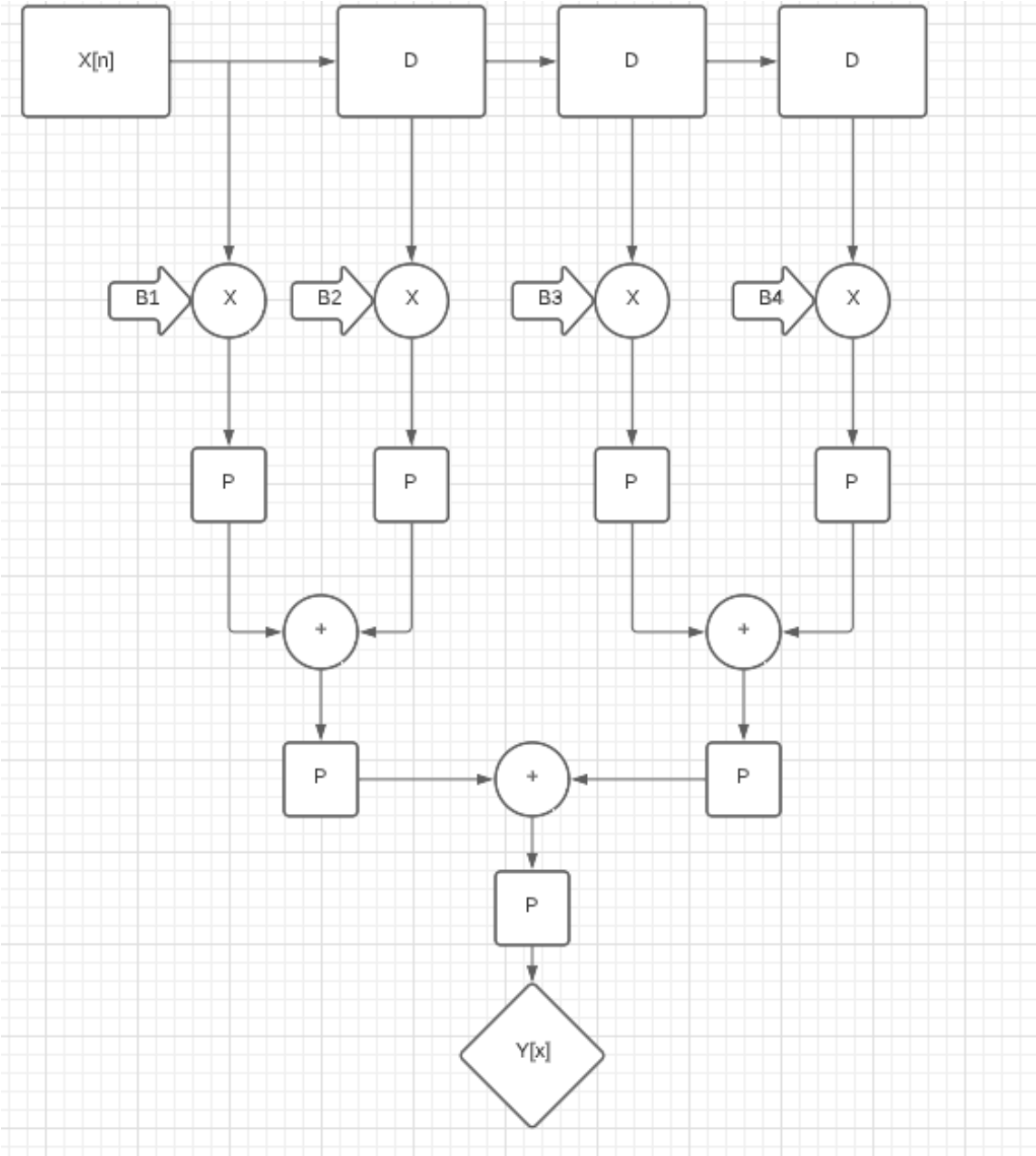
La demostración del funcionamiento se la mostré en el horario de clase al profesor.

Práctica 5: Implementación de un filtro digital

En esta práctica se va a implementar un filtro FIR (Finite Impulse Response). El filtro FIR es de los más sencillos y habituales en los diseños digitales consta de sumas y multiplicaciones.
El filtro proporcionado por el profesor es el siguiente.

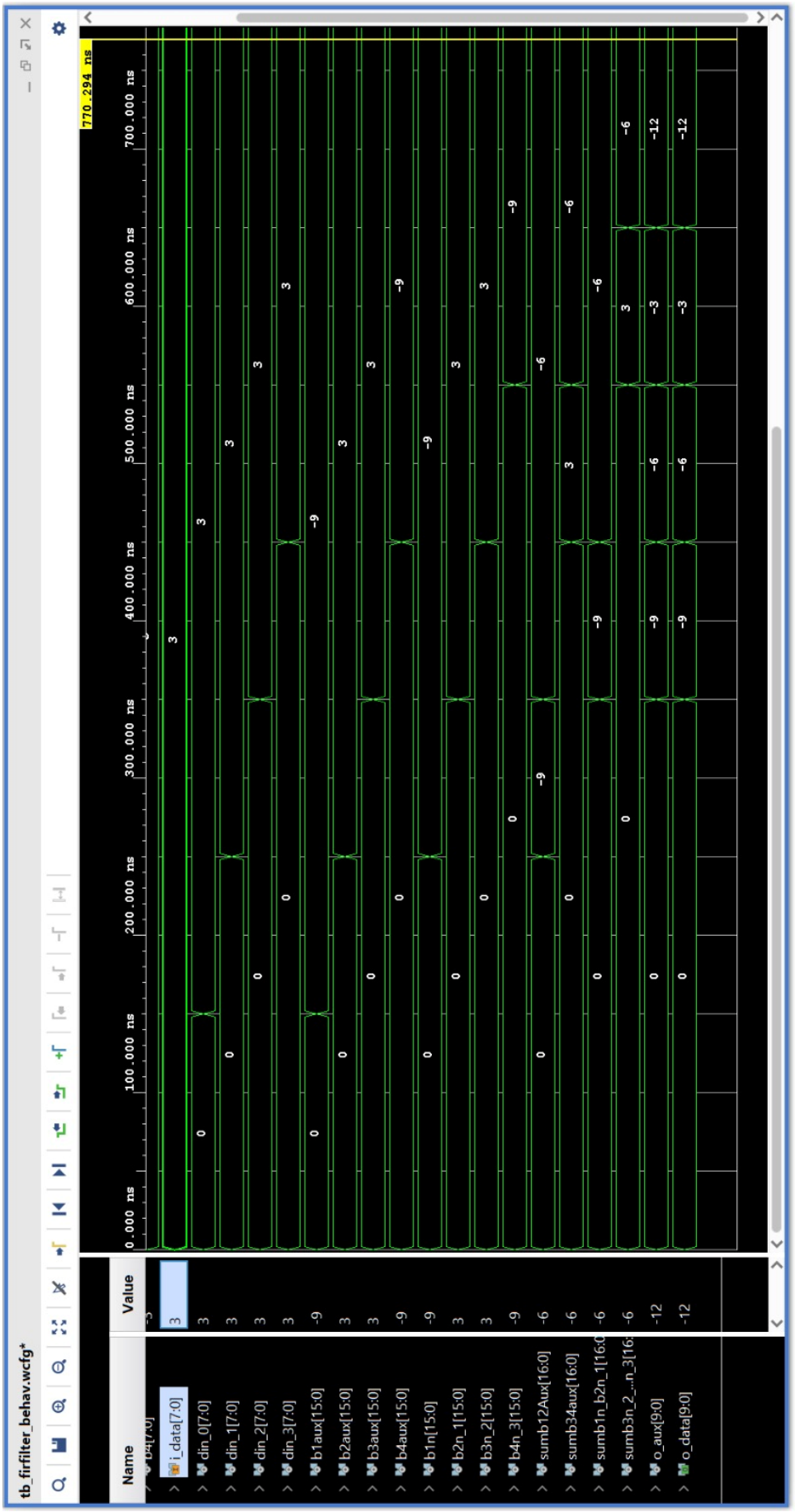


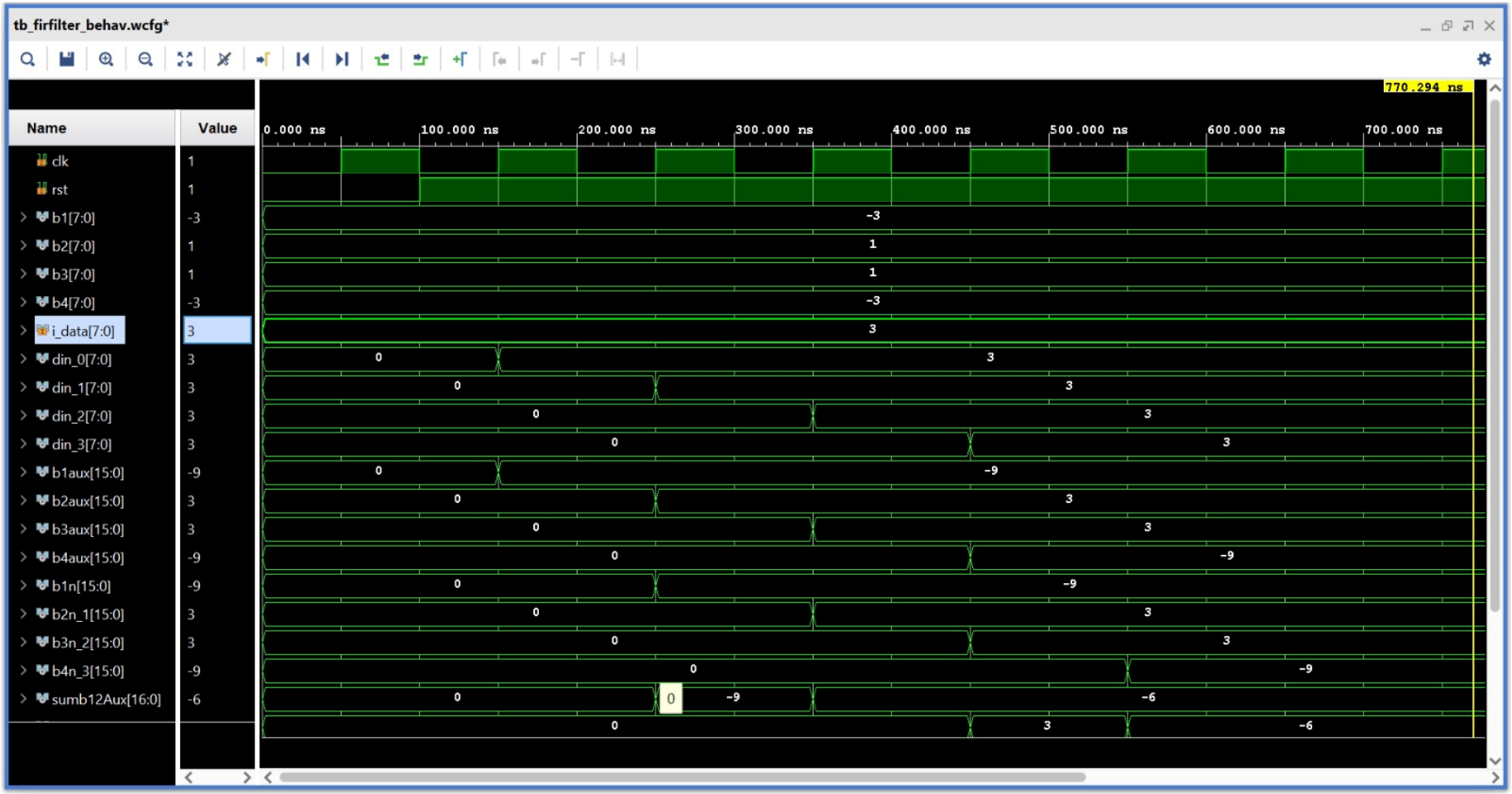
A este circuito se le puede implementar la técnica del Pipeline. El Pipeline es un método por el que se busca sacar más eficiencia al mismo, aumentando la latencia de salida de nuestro resultado, pero reduciendo accesos aumentando el área. Así somos capaces de controlar mejor la información. Como propuesta de modelo ofrecemos la siguiente.



Este modelo es más eficiente ya que registramos cada salida de cada operación, y ofrece una mayor paralelización para realizar la cuenta más rápidamente.

TestBench


tb_firfilter_behav.wcfg*
Name
Value
i_data[7:0]
din_0[7:0]
din_1[7:0]
din_2[7:0]
din_3[7:0]
b1aux[15:0]
b2aux[15:0]
b3aux[15:0]
b4aux[15:0]
b1n[15:0]
b2n_1[15:0]
b3n_2[15:0]
b4n_3[15:0]
sumb12Aux[16:0]
sumb34aux[16:0]
sumb1n_b2n_1[16:0]
o_aux[9:0]
o_data[9:0]
770.294 ns
0.000 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
500.000 ns
600.000 ns
700.000 ns


tb_firfilter_behav.wcfg*
770.294 ns
Name
Value
0.000 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
500.000 ns
600.000 ns
700.000 ns
clk 1
rst 1
b1[7:0] -3
b2[7:0] 1
b3[7:0] 1
b4[7:0] -3
i_data[7:0] 3
din_0[7:0] 3
din_1[7:0] 3
din_2[7:0] 3
din_3[7:0] 3
b1aux[15:0] -9
b2aux[15:0] 3
b3aux[15:0] 3
b4aux[15:0] -9
b1n[15:0] -9
b2n_1[15:0] 3
b3n_2[15:0] 3
b4n_3[15:0] -9
sumb12Aux[16:0] -6