Universidad Autónoma de Baja California

Facultad de Ciencias Químicas e Ingeniería



**CIRCUITOS DIGITALES AVANZADOS**

**Practica 7**

**Dispositivos Programables**

**Docente:** Lara Camacho Evangelina

**Alumnos:** Gómez Cárdenas Emmanuel Alberto **1261509**

León Romero Pablo Constantino **1253171**

Contenido

[OBJETIVO 3](#_Toc58018322)

[EQUIPO 3](#_Toc58018323)

[FUNDAMENTO TEORICO 3](#_Toc58018324)

[Flujo de datos 3](#_Toc58018325)

[Estructura WITH - SELECT - WHEN 4](#_Toc58018326)

[Estructura PROCESS 5](#_Toc58018327)

[Estructura IF - THEN - ELSE 7](#_Toc58018328)

[Estructura WHEN - CASE 8](#_Toc58018329)

[Estructura LOOP 8](#_Toc58018330)

[Desarrollo 10](#_Toc58018331)

[Parte 1. Contador 10](#_Toc58018332)

[Codigo ContadorModuloM 10](#_Toc58018333)

[Codigo ContadorModuloM\_tb 11](#_Toc58018334)

[Simulación ContadorModuloM 11](#_Toc58018335)

[Parte 2. Registro de desplazamiento 12](#_Toc58018336)

[Código RegistroDesplazamiento 12](#_Toc58018337)

[Código RegistroDesplazamiento\_tb 13](#_Toc58018338)

[Simulación Registro de desplazamiento 14](#_Toc58018339)

[Parte 3. Sumador en serie 15](#_Toc58018340)

[Codigo SumadorSerie 15](#_Toc58018341)

[Codigo SumadorSerie\_tb 16](#_Toc58018342)

[Registro de desplazamiento 18](#_Toc58018343)

[Flip flop tipo D 19](#_Toc58018344)

[Simulación SumadorSerie 3 bits 20](#_Toc58018345)

[Simulación SumadorSerie 4 bits 20](#_Toc58018346)

[Simulación SumadorSerie 5 bits 20](#_Toc58018347)

[CONCLUSIONES 21](#_Toc58018348)

[VIDEO DE PRACTICA 21](#_Toc58018349)

# OBJETIVO

Diseñar circuitos secuenciales en dispositivos programables FPGA.

# EQUIPO

Computadora con el IDE Xilinx Vivado u otro software para desarrollo de código para FPGAs.

# FUNDAMENTO TEORICO

## Flujo de datos

Cuando se desarrollan programas en VHDL u otro lenguaje de descripción de hardware, hay que recordar que la ejecución no va a ser igual a la de un programa de computadora. El código en VHDL **describe hardware**, no una serie de instrucciones de un procesador. Un circuito electrónico puede contener elementos que ejecutan acciones al mismo tiempo, en paralelo. Por ejemplo, en el medio sumador de la Fig. 1, las entradas A y B se aplican a dos compuertas lógicas, XOR y AND, y cada una obtiene una salida. Estas dos acciones son ejecutadas en paralelo y es a lo que se le llama **concurrencia**.

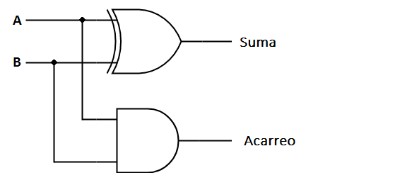


Figura 1. Medio sumador.

El lenguaje VHDL es concurrente, por lo tanto, no ejecuta las instrucciones en el orden en que están escritas. Habiendo dos o más instrucciones, no las ejecuta una tras la otra, sino que éstas pueden ejecutarse a la vez.

La instrucción básica en la ejecución concurrente es la asignación de señales por medio del símbolo <=, tal como se vio en la práctica anterior con el medio sumador:

suma <= a xor b; acarreo <= a and b;

Existen además otras estructuras de alto nivel que facilitan la asignación de señales con base al cumplimiento de ciertas condiciones, estas son **WHEN - ELSE** y **WITH - SELECT - WHEN**.

***Estructura WHEN - ELSE***

Sentencia de selección múltiple. Su sintaxis es:

<señal> <= <asignación\_1> WHEN <condición\_1> ELSE

<asignación\_2> WHEN <condición\_2> ELSE

...

<asignación\_n> WHEN <condición\_n> ELSE

<asignación\_m>;

Un ejemplo de uso:

z <= "00" WHEN a = b ELSE

"10" WHEN a > b ELSE

"11";

Siempre tiene que haber una asignación de valor, por lo tanto, cuando una condición no va a generar cambios en la señal, se usa la palabra reservada **UNAFFECTED**.

z <= a WHEN b = '1' ELSE UNAFFECTED;

### Estructura WITH - SELECT - WHEN

Es similar a la estructura switch de los lenguajes de programación para computadoras.

Realiza la asignación en base al estado de una señal. Su sintaxis es:

WITH <señal\_1> SELECT

<señal\_2> <= <asignación\_1> WHEN <estado\_señal1>,

<asignación\_2> WHEN <estado\_señal2>,

...

<asignación\_n> WHEN OTHERS;

Un ejemplo de uso:

WITH x SELECT

z <= a WHEN "00",

b WHEN "01",

c WHEN "10",

d WHEN OTHERS;

En la estructura **WITH - SELECT - WHEN**, todos los estados posibles de la señal de control

(señal\_1 en la descripción) deben ser incluidos en el conjunto de opciones y no deben repetirse. A diferencia, en la estructura **WHEN\_ELSE** no es forzoso incluir todos los posibles estados, incluso puede operar con expresiones con diferentes argumentos, como se ve en el ejemplo:

z <= '0' WHEN reset = '0' ELSE

d WHEN clk = '1';

### Estructura PROCESS

Esta estructura es concurrente respecto a que todos los PROCESS y otras sentencias concurrentes en el código se ejecutan a la vez. Sin embargo, las instrucciones que están adentro del PROCESS se ejecutan **secuencialmente**, una detrás de otra, siguiendo el orden en el que están escritas en el código. Esta estructura permite describir un circuito en base a su **comportamiento**. Su sintaxis es:

PROCESS [lista de sensibilidad]

[declaración de variables del proceso]

BEGIN

[sentencias secuenciales]

END PROCESS;

La **lista de sensibilidad** indica todas las señales que provocan que se ejecute el proceso, esto cuando por lo menos una de ellas cambia de valor. Por ejemplo, un proceso declarado como process(clk,rst)se ejecuta cada vez que las señales clk o rst cambian de estado. Todas las señales referenciadas en un proceso deberían estar en la lista de sensibilidad.

Ejemplo de uso donde se describe un flip flop D:

entity DFlipFlop is

port (d,clk: in std\_logic;

pre,clr: in std\_logic;

q: out std\_logic);

end DFlipFlop;

architecture behave of DFlipFlop is

begin

process(clk,pre,clr)

begin

if (clr='0') then -- clr asíncrono tiene mayor precedencia.

q <= '0';

elsif(pre='0')then -- pre asíncrono tiene la siguiente precedencia.

q <= '1';

elsif rising\_edge(clk) then -- operación síncrona, ocurre si clr=pre='1'

q <= d; -- y hay un flanco ascendente en el clk.

end if;

end process;

end;

En VHDL, variables y señales son elementos distintos. Las señales se tienen que declarar entre la sentencia ARCHITECTURE y su correspondiente BEGIN. Las variables se declaran entre PROCESS y su BEGIN. Otra diferencia es que las **señales actualizan su valor solo hasta que termina la presente ejecución del proceso**, mientras que las variables toman su nuevo valor inmediatamente. Dentro de un proceso se pueden usar tanto señales como variables.

Las variables se usan generalmente para retener los resultados inmediatos en la implementación de un algoritmo, de la siguiente manera:

1. Se asigna el estado de una señal a una variable, la señal es un dato de entrada al algoritmo. Se usa el operador **:=** en la asignación a la variable.
2. Se ejecuta el algoritmo (proceso).
3. El resultado que quedó en la variable se copia a una señal, esta señal es entonces un dato de salida del algoritmo.

No se puede acceder a los valores de las variables fuera de su proceso. Una señal si puede ser accedida en uno o más procesos, pero **no más de un proceso puede modificar su estado**. En la Fig. 2 se muestra el uso que generalmente se da a señales y variables dentro de los procesos.

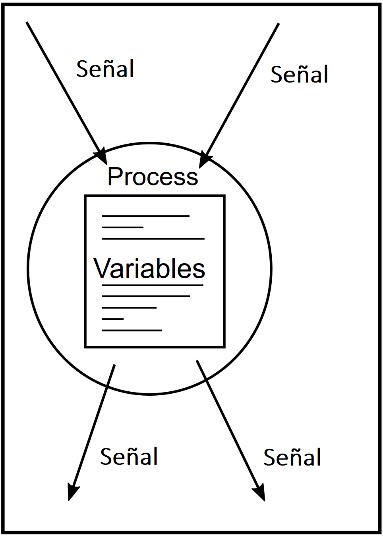


Figura 2. Uso de señales y variables en un proceso.

Ejemplo:

ENTITY SenalesYVariables IS

PORT (c: IN std\_logic);

END ENTITY;

ARCHITECTURE behave OF SenalesYVariables IS

SIGNAL a,b: std\_logic;

BEGIN

PROCESS(c)

VARIABLE z: std\_logic; BEGIN a <= c and b; -- "a" actualiza su valor solo hasta que termina -- la ejecución actual del proceso.

z := a or c; -- "z" se actualiza inmediatamente. Recuerde que "a" -- no ha cambiado en este momento.

END PROCESS;

END ARCHITECTURE;

***Sentencias de selección secuenciales***

### Estructura IF - THEN - ELSE

Permiten seleccionar el código a ejecutar en base a una o más condiciones. Es una estructura secuencial. Su sintaxis es:

IF <condición\_1> THEN

[sentencias]

ELSIF <condición\_2> THEN

[sentencias]

ELSE

[sentencias]

END IF;

Un ejemplo de uso:

if (s = "00") then

z <= a;

elsif (s = "11") then

z <= b;

else

z <= c;

end if;

Puede haber más de un elsif. Las cláusulas elsif y else son opcionales, pero si la estructura if-then-else no está completamente especificada (no tiene else), esto implica que se va a implementar con un elemento de memoria (no un multiplexor, por ejemplo) o incluso la implementación podría no ser la adecuada al diseño (por ejemplo, el compilador podría colocar una señal directamente a tierra/vcc).

### Estructura WHEN - CASE

Permite evaluar una expresión para seleccionar el código a ejecutarse. Se tienen que tomar en cuenta todos los casos, para esto se puede colocar como última opción la sentencia **WHEN OTHERS**. Su sintaxis es:

CASE <expresión> IS

WHEN <valor\_1> => [sentencias]

WHEN <valor\_2> => [sentencias]

WHEN <rango\_de\_valores> => [sentencias]

WHEN OTHERS => [sentencias]

END CASE;

Un ejemplo de uso:

Case s is

when “00” =>

z <= a;

when “11” =>

z <= b;

when others =>

z <= c;

end case;

### Estructura LOOP

Se usa para crear bucles. En VHDL, existe el bucle **FOR** y **WHILE**. Su sintaxis es:

[etiqueta:] [WHILE <condición> | FOR <condición>] LOOP

[sentencias]

[exit;]

[next;]

END LOOP [etiqueta];

Ejemplo de uso de **FOR LOOP**:

inicio: for k in N-1 downto 0 loop

Q(k) <= '0';

end loop inicio;

La variable k es implícita en el FOR LOOP, no necesita declararse.

Ejemplo de uso de **WHILE LOOP**:

inicio: while (k > 0) loop

Q(k) <= '0'

k := k – 1;

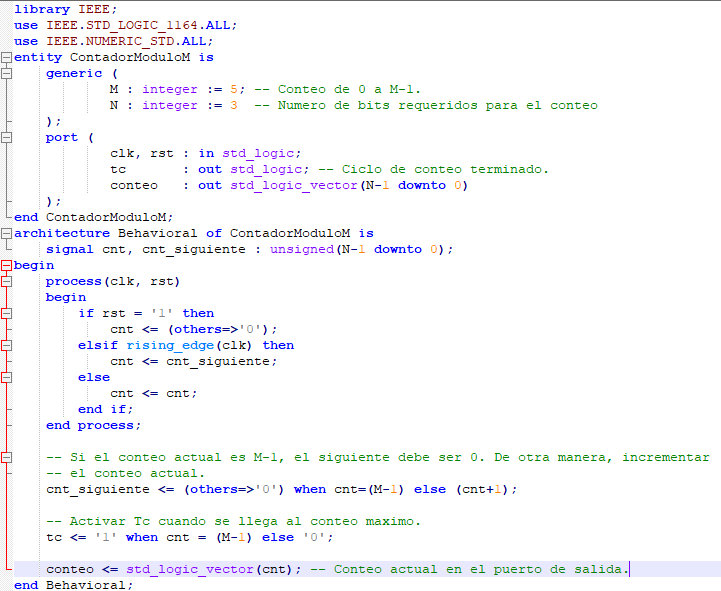
end loop inicio;

La variable k debe declararse como una variable de proceso, entre PROCESS y BEGIN. En este ejemplo, su formato podría ser variable k: integer := N-1;

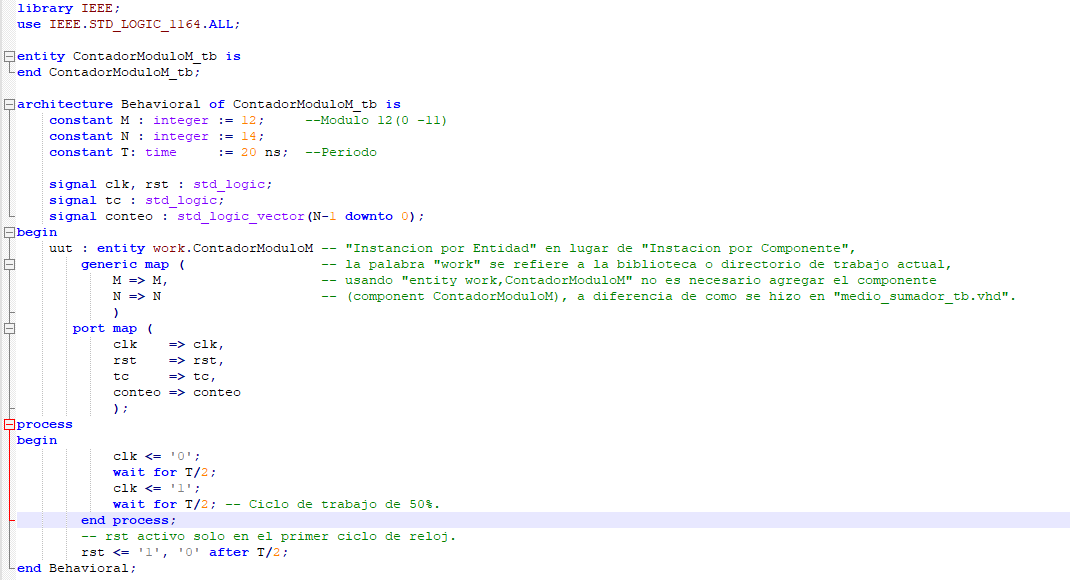
# Desarrollo

## Parte 1. Contador

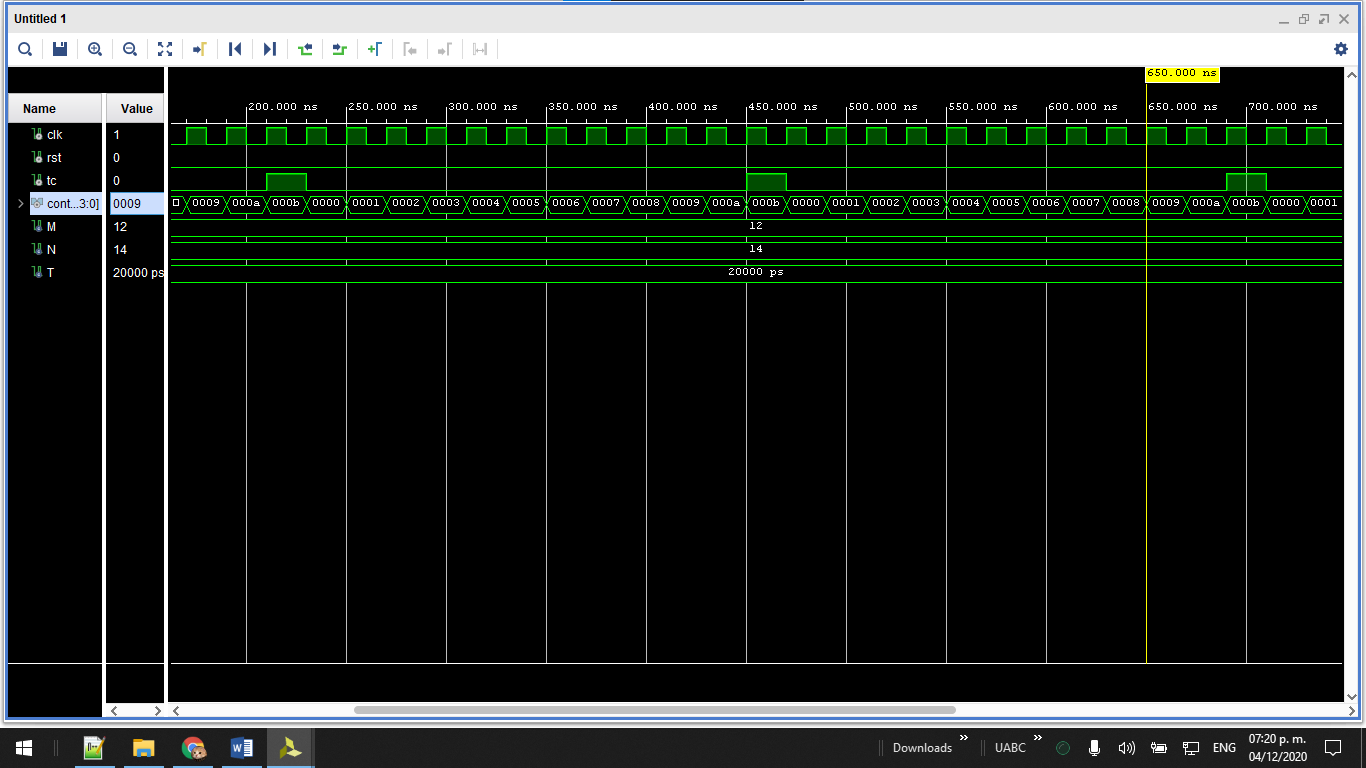
### Codigo ContadorModuloM



### Codigo ContadorModuloM\_tb

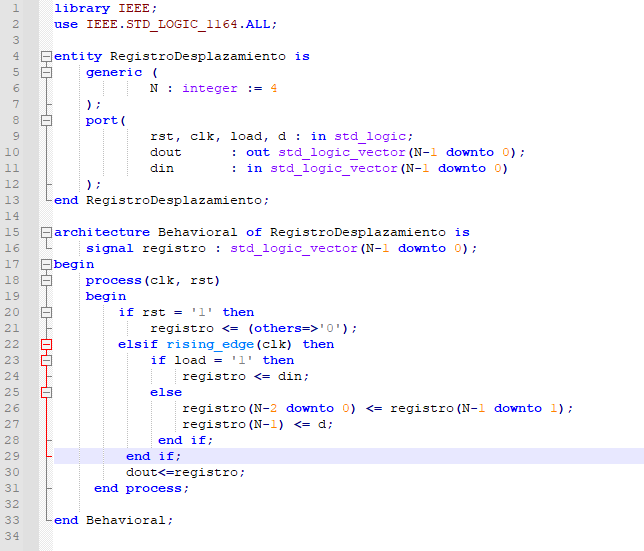


### Simulación ContadorModuloM

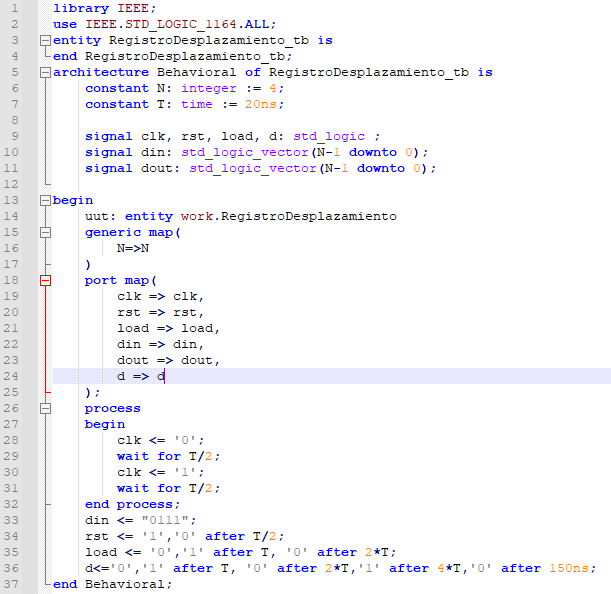


## Parte 2. Registro de desplazamiento

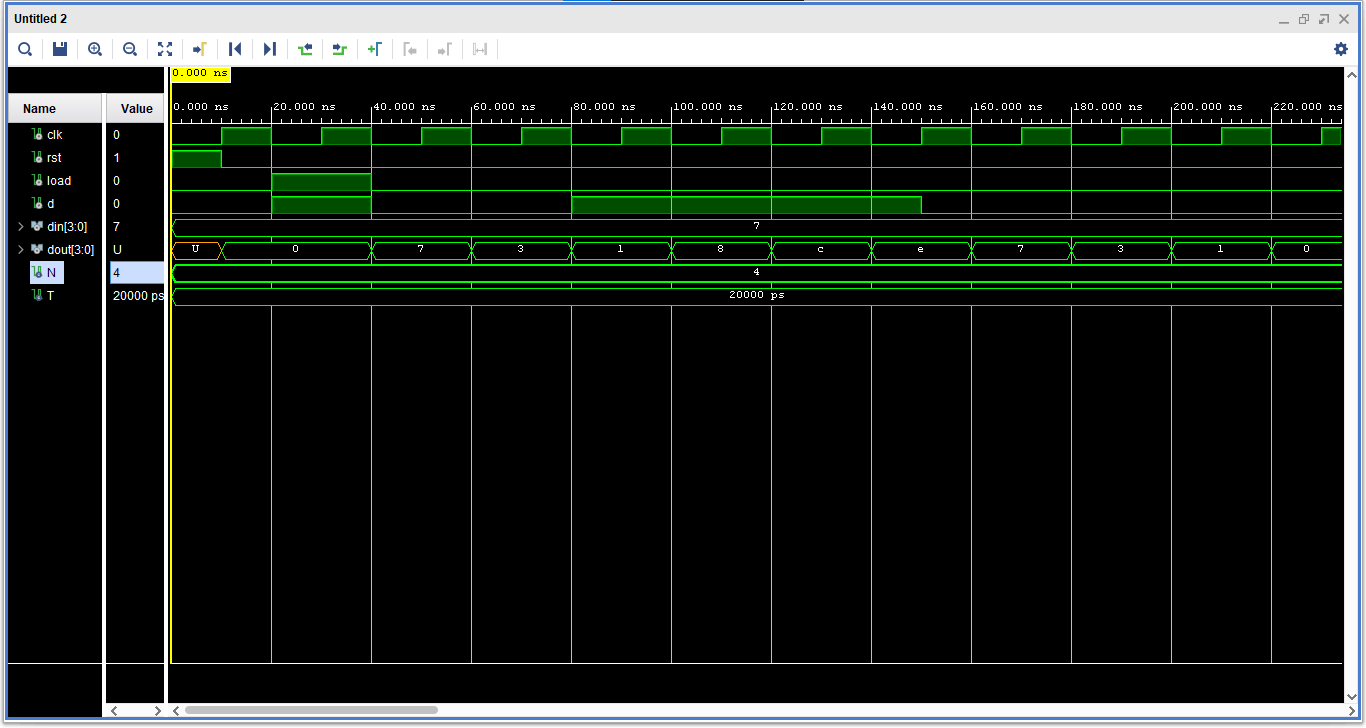
### Código RegistroDesplazamiento



### Código RegistroDesplazamiento\_tb

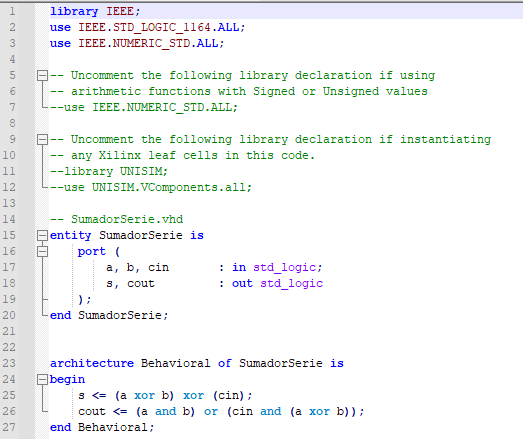


### Simulación Registro de desplazamiento

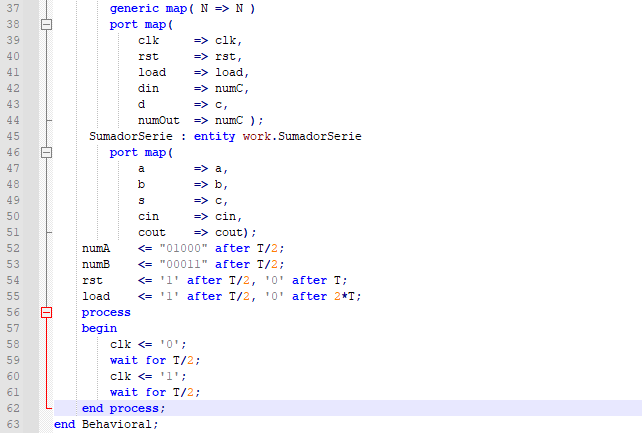
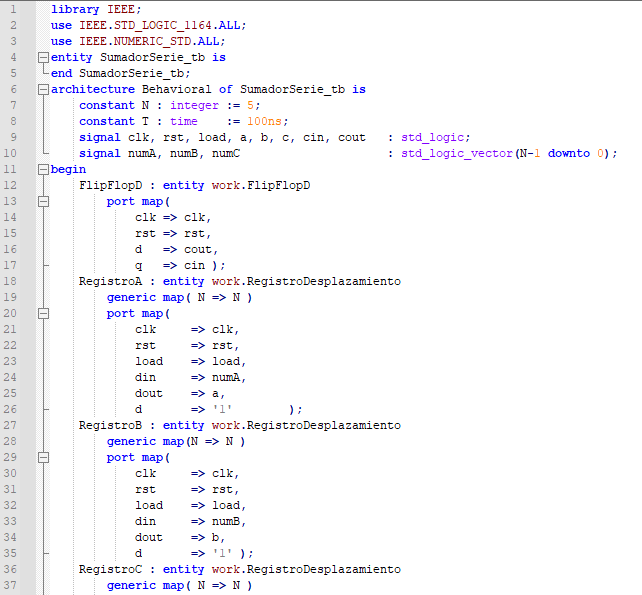


## Parte 3. Sumador en serie

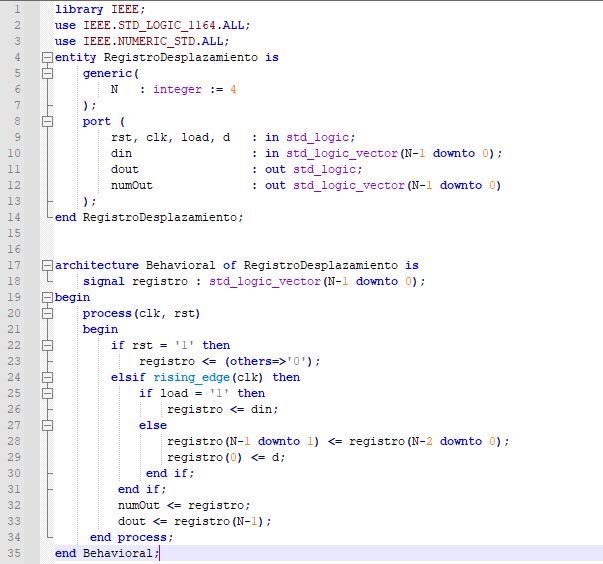
### Codigo SumadorSerie



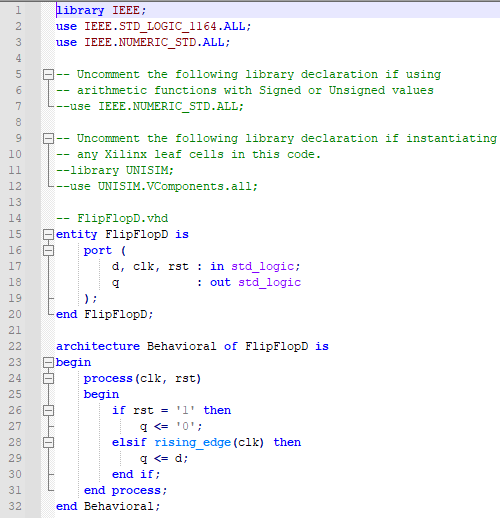
### Codigo SumadorSerie\_tb



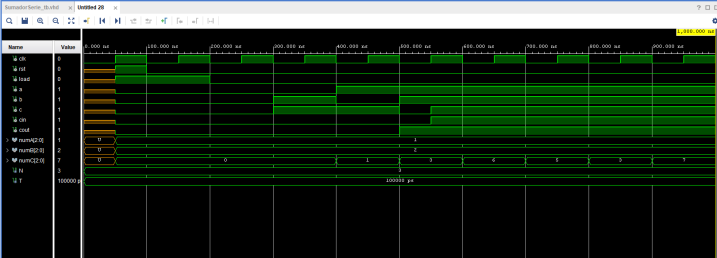
### Registro de desplazamiento



### Flip flop tipo D



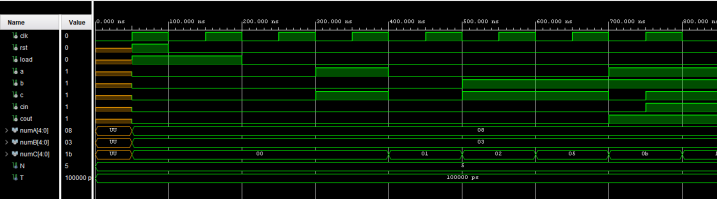
### Simulación SumadorSerie 3 bits



### Simulación SumadorSerie 4 bits



### Simulación SumadorSerie 5 bits



# CONCLUSIONES

**Gómez Cárdenas Emmanuel Alberto:**

En esta práctica nos pudimos familiarizar un poco más con el manejo del programa Vivado, ya que sigue siendo un poco complejo. Gracias al programa pudimos diseñar y simular circuitos con un nivel de complejidad aún más alto que los anteriores como lo es el registro de desplazamiento o el sumador.

**Pablo Constantino León romero:**

Esta práctica fue desafiante dado que no hemos tenido demasiada teoría en programación vhdl. Aun así, creo que se pudo lograr un buen resultado y se aprendieron bastantes cosas. Se entendí mejor el manejo de las diversas partes del código como, la entity, arquitectura, proceso, etc. Además, se vieron nuevas formas de implementar circuitos ya conocidos por de antes.

# VIDEO DE PRACTICA

<https://drive.google.com/drive/folders/1iFYMaCSydAgiQBsk1RVU_BWEk5s5KEQV?usp=sharing>