

INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

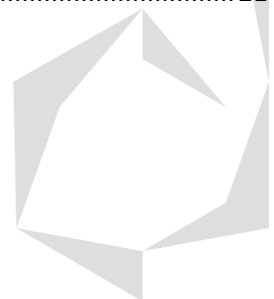
ELECTRÓNICA DIGITAL: CIRCUITOS LÓGICOS, LENGUAJE VHDL Y VERILOG

XILINX (64-BIT PROJECT NAVIGATOR) & ADEPT

Flip Flops, Registros,
Antirrebotes y Memorias RAM/ROM

Contenido

Flip Flops	2
Tipos de Flip Flops RS, D, JK y T.....	2
Código Verilog: Flip Flop Tipo RS (Set Reset)	4
Código Verilog: Flip Flop Tipo D	5
Código Verilog: Flip Flop Tipo JK	5
Código Verilog: Flip Flop Tipo T	5
Circuitos Combinacionales y Secuenciales.....	6
Registros y Shift Registers	6
Código VHDL: Registro	8
Código VHDL: Shift Register	9
Simulación: Shift Register.....	9
Retardo o Delay Antirrebotes (Delay, Millis y Shift Register).....	10
Código Arduino: millis o delay() - Antirrebotes:	11
Código Verilog: Shift Register - Antirrebotes:	11
Código VHDL: Shift Register - Antirrebotes:	12
Simulación: Shift Register - Antirrebotes: $TCLK = 4ms$; $\#delays = 3$; $tdelay = 10ms$	12
Memoria RAM y ROM	13
Capacidad de Memoria.....	14
Leer o Escribir Información en una Memoria.....	14
Tipos de Memorias.....	16
Memorias RAM y ROM: Array en Verilog y VHDL.....	17
Código Verilog - Memoria ROM/RAM:	18
Código VHDL - Memoria ROM/RAM:	19
Simulación: Memorias RAM/ROM.....	20
Memoria ROM (Read); $RW = 1$ y $CS = 1$	20
Memoria RAM (Read); $RW = 1$ y $CS = 0$	20
Memoria RAM (Write); $dataInRAM = 10101111$ y 11110101 ; $RW = 0$ y $CS = 0$	20
Referencias:	21



Flip Flops

Tipos de Flip Flops RS, D, JK y T

Los Flip Flops son la unidad de memoria más básica, ya que estos pueden mantenerse en un estado binario de forma indefinida, siempre y cuando se les continúe administrando tensión, esto significa que pueden almacenar el valor de 1 bit, o sea una variable que puede adoptar valores de 0 lógico (0 a 1.5V aproximadamente), 2 o Indefinido (de 1.5 a 2V aprox.) o 1 lógico (alrededor de 2 a 5V).

Existen diferentes tipos de Flip Flops hechos con distintas compuertas lógicas AND, OR, NOT, NAND, NOR, etc. Todos cuentan con un número de entradas donde se dicta la acción a realizar con el bit guardado dentro del flip flop y en las salidas se ve reflejada dicha acción.

- **Flip Flop Tipo RS (Set Reset):** Este tipo de Flip Flop también llamado como **Latch RS**, está hecho usualmente con 2 compuertas NOR y cuenta con:
 - **2 entradas:**
 - **S (Set):** Si tiene un 1 se ingresa el valor de 1 lógico al bit de la memoria y si tiene un 0 se deja tal cual como está el estado actual del flip flop, por lo que se le llama el **modo memoria**.
 - **R (Reset):** Si tiene un 1 se reinicia el valor del flip flop, por lo que se ingresa el valor 0 lógico al bit de la memoria y si tiene un 0 se deja tal cual como está el estado actual, por lo que se le llama el **modo memoria**.
 - **2 salidas:**
 - **Q:** Almacena el bit ingresado por las entradas Set o Reset.
 - **Q':** Es el inverso de Q.

El flip flop RS internamente puede estar conformado por una serie de compuertas NAND o NOR y lo que se puede hacer con él es simplemente **guardar el valor de 1 bit en la memoria**, ya sea de 1 lógico (**modo ajuste**), 0 lógico (**modo reset**) o mantener el valor que estaba previamente ingresado (**modo memoria**), el **modo indefinido** no tiene función, por lo que se debe evitar usarlo:

F-F RS NOR

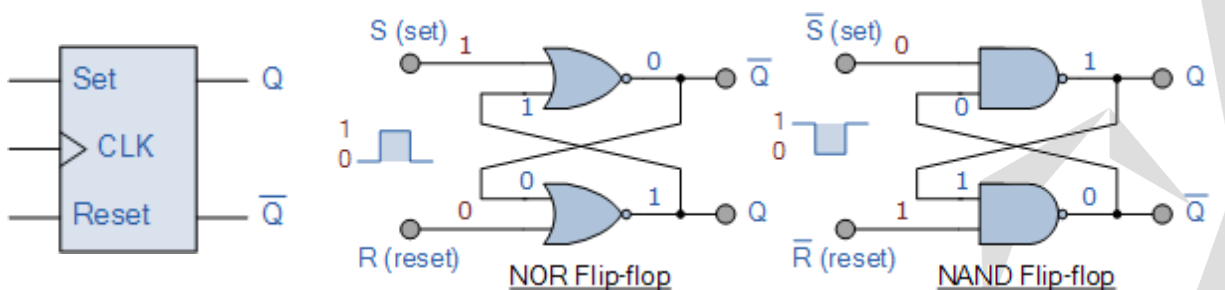
Modo Ajuste
Reset
Memoria
Indefinido

Entradas		Salidas	
R	S	Q	Q'
0	1	1	0
1	0	0	1
0	0	0	1
1	1	?	?

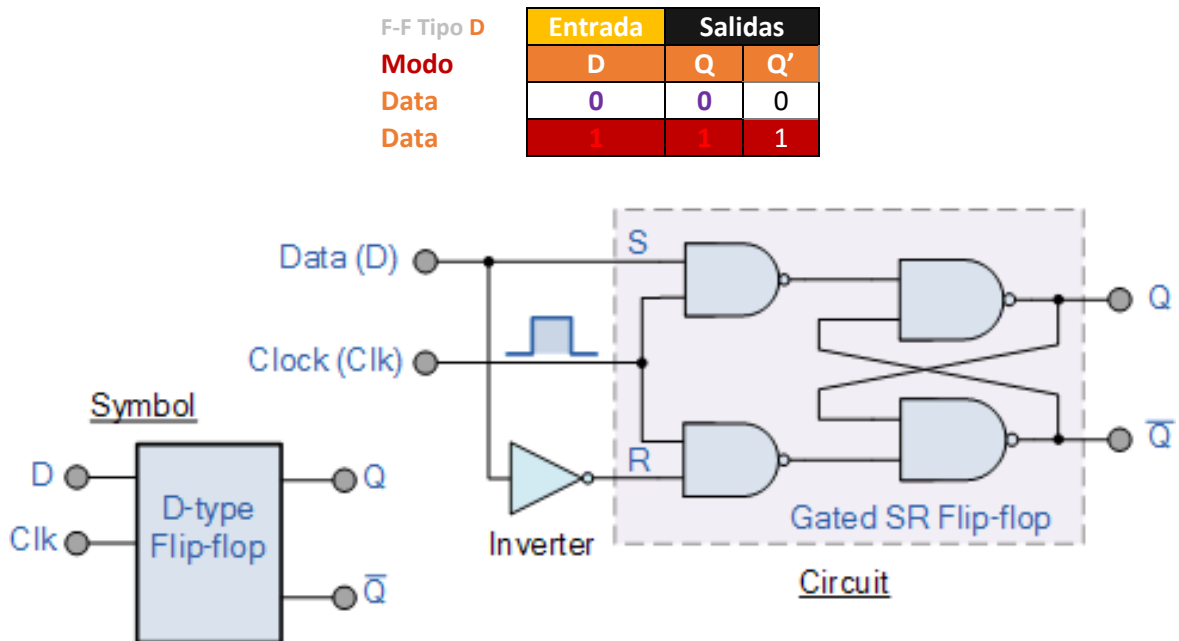
F-F RS NAND

Modo Ajuste
Reset
Memoria
Indefinido

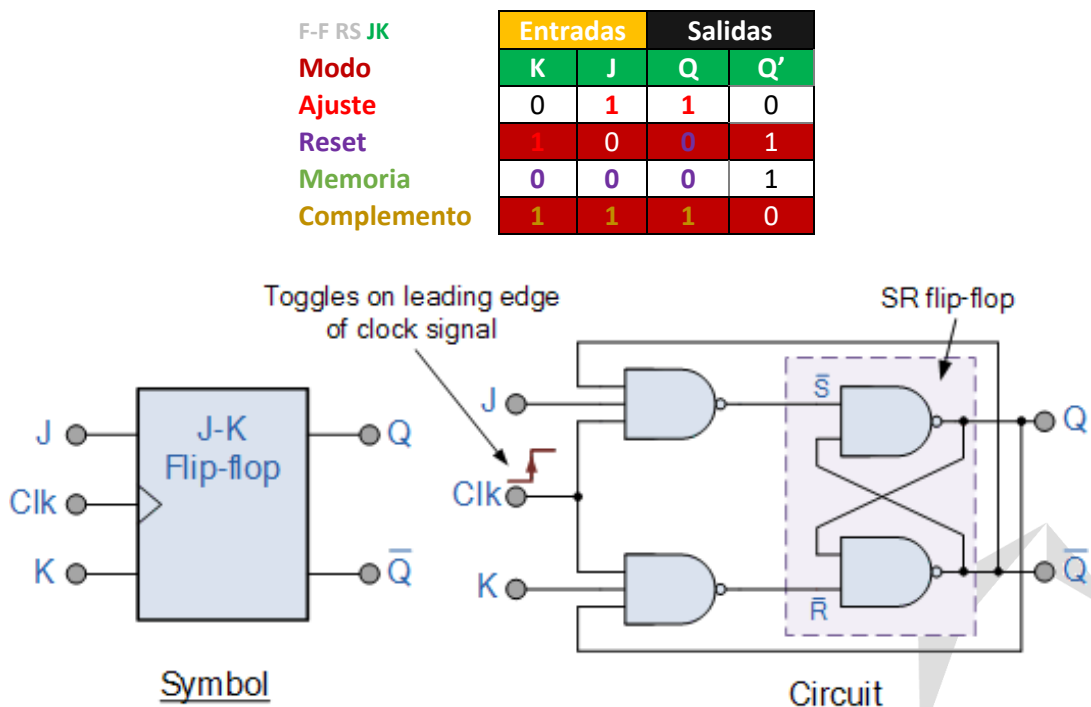
Entradas		Salidas	
R	S	Q	Q'
1	0	1	0
0	0	0	1
1	1	0	1
0	0	?	?



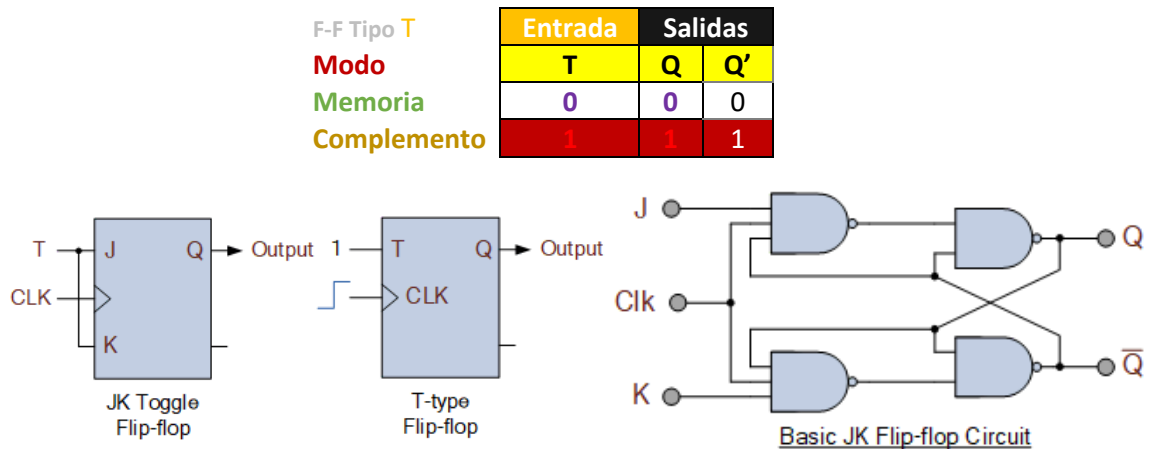
- **Flip Flop Tipo D:** De forma interna tiene incluida la estructura de un **flip flop tipo RS**, pero como el **flip flop tipo D** solo cuenta con una entrada (llamada **Data**), se puede ingresar un **0 lógico** o **1 lógico** de forma directa a la salida **Q**, donde **Q'** de nuevo representará su valor inverso.



- **Flip Flop Tipo JK:** Este flip flop funciona de forma muy parecida al **flip flop tipo RS**, donde ahora la entrada **J** es equivalente a la entrada **S (Set)** y la **K** es equivalente a la entrada **R (Reset)**, pero su gran diferencia es que aquí no existe el **modo indefinido** (que no tiene función), en su lugar, se tiene el **modo complemento**, que **invierte el estado anterior que se tenía en la salida Q** y la salida **Q'** otra vez representa su valor inverso.



- **Flip Flop Tipo T:** De forma interna tiene incluida la estructura de un **flip flop tipo JK**, pero como el **flip flop tipo T** solo cuenta con una entrada (llamada **Toggle**), se pueden ingresar solo dos valores que significarán lo siguiente:
 - **0 lógico:** Será equivalente al **modo memoria**, dejando sin modificar el valor que estaba previamente ingresado en la salida **Q** de la **memoria de 1 bit creada con el flip flop**.
 - **1 lógico:** Será equivalente al **modo complemento**, que **invierte el estado anterior que se tenía en la salida Q**.



A continuación, se mostrará el código en Verilog que representa cada uno de los tipos de Flip Flops, pero no se hará un programa de ellos porque no hay muchas aplicaciones que se pueda realizar al guardar 1 solo bit a la vez:

Código Verilog: Flip Flop Tipo RS (Set Reset)

```
//1.-Flip Flop Tipo SR: Su entrada S (Set) indica el bit a guardar y su entrada R (Reset) "borra" el
//contenido de la memoria de 1 bit poniendo un 0 lógico en la salida Q, donde siempre se guarda el
//contenido del flip flop, la salida Q' es su valor inverso, pero tiene el problema de que si se manda
//un 1 en R y un 1 en S, ese estado esta indefinido y el dispositivo no sabe qué hacer.

module FF_SR (
    input wire CLK_1hz, //Reloj a 1Hz.
    input wire S,        //Entrada S.
    input wire R,        //Entrada R.
    output reg QSR,      //Salida Q.
    output reg QNSR     //Salida Q'.
);

//REG: No es ni una entrada ni una salida porque no puede estar vinculada a ningún puerto de la NEXYS 2,
//solo sirve para almacenar y usar valores que sobrevivirán durante la ejecución del código y que, además
//se deben usar dentro de un condicional o bucle.
reg tmp;

//La instrucción always@() se usa para poder usar condicionales o bucles y tiene su propio begin y end.
//posedge() sirve para identificar un flanco de subida en la señal que tenga entre su paréntesis, si se
//quisiera identificar un flanco de bajada se usaría la instrucción negedge().
always@(posedge(CLK_1hz))
begin
    if (R == 1'b0 & S == 1'b1) begin //Modo Ajuste (Set): Coloca un 1 lógico en la memoria.
        tmp = 1'b1;
    end
    else if (R == 1'b1 & S == 1'b0) begin //Modo Reset: Coloca un 0 lógico en la memoria.
        tmp = 1'b0;
    end
    else if (R == 1'b0 & S == 1'b0) begin //Modo Memoria: Deja igual lo que se había guardado.
        tmp = tmp;
    end
    else if (R == 1'b1 & S == 1'b1) begin //Modo Indefinido.
        tmp = 1'bx;
    end
    QSR = tmp; //Asignación de la salida Q.
    QNSR = ~tmp; //Asignación de la salida Q'.
end
endmodule
```

Código Verilog: Flip Flop Tipo D

```
//2.-Flip Flop Tipo D: Su entrada D directamente se almacena en la salida Q, la salida Q' es su inversa.
module FF_D (
    input wire CLK_1Hz, //Reloj a 1Hz.
    input wire D,        //Entrada D.
    output reg QD,        //Salida Q.
    output reg QND        //Salida Q'.
);

//La instrucción always@() se usa para poder usar condicionales o bucles y tiene su propio begin y end.
//posedge() sirve para identificar un flanco de subida en la señal que tenga entre su paréntesis, si se
//quisiera identificar un flanco de bajada se usaría la instrucción negedge().
always@(posedge(CLK_1Hz))
    begin
        QD = D;           //Modo Data: Almacena lo que ingrese en la entrada D a la salida Q.
        QND = ~ QD;       //Asignación de la salida Q'.
    end
endmodule
```

Código Verilog: Flip Flop Tipo JK

```
//3.-Flip Flop Tipo JK: Su entrada J (Set) indica el bit a guardar y su entrada K (Reset) "borra" el
//contenido de la memoria de 1 bit poniendo un 0 lógico en la salida Q, donde siempre se guarda el
//contenido del flip flop, la salida Q' es su valor inverso. Podemos ver que es idéntico al flip flop RS
//pero su gran diferencia es que no tiene un estado indefinido, en su lugar tiene el modo complemento,
//que invierte el valor previamente guardado en la salida Q.
module FF_JK (
    input wire CLK_1Hz, //Reloj a 1Hz.
    input wire J,        //Entrada J.
    input wire K,        //Entrada K.
    output reg QJK,       //Salida Q.
    output reg QNJK       //Salida Q'.
);

//La instrucción always@() se usa para poder usar condicionales o bucles y tiene su propio begin y end.
//posedge() sirve para identificar un flanco de subida en la señal que tenga entre su paréntesis, si se
//quisiera identificar un flanco de bajada se usaría la instrucción negedge().
always@(posedge(CLK_1Hz))
    begin
        if (K == 1'b0 & J == 1'b1) begin //Modo Ajuste (Set): Coloca un 1 lógico en la memoria.
            tmp = 1'b1;
        end
        else if (K == 1'b1 & J == 1'b0) begin //Modo Reset: Coloca un 0 lógico en la memoria.
            tmp = 1'b0;
        end
        else if (K == 1'b0 & J == 1'b0) begin //Modo Memoria: Deja igual lo que se había guardado.
            tmp = tmp;
        end
        else if (K == 1'b1 & J == 1'b1) begin //Modo Complemento: Invierte lo que había en la salida Q.
            tmp = ~tmp;
        end
        QSR = tmp; //Asignación de la salida Q.
        QNSR = ~tmp; //Asignación de la salida Q'.
    end
endmodule
```

Código Verilog: Flip Flop Tipo T

```
//4.-Flip Flop T: Cuando en su entrada T se ingresa un 0 lógico, se activa el modo memoria, que deja el
//bit guardado en la salida Q tal cual como estaba y cuando se ingresa un 1 lógico, se activa el modo
//complemento, que invierte el valor previamente guardado en la salida Q, la salida Q' es su inverso.
module FF_T (
    input wire CLK_1Hz, //Reloj a 1Hz.
    input wire T,        //Entrada T.
    output reg QT,        //Salida Q.
    output reg QNT        //Salida Q'.
);

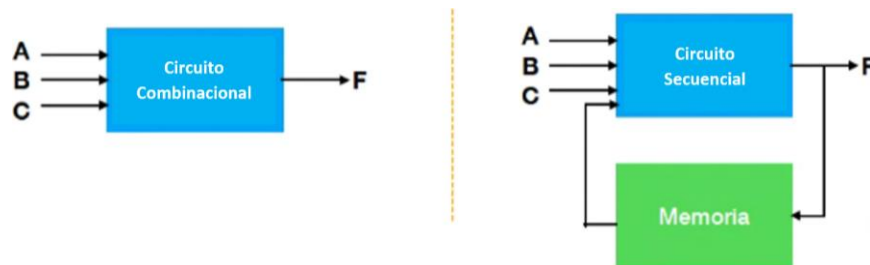
//La instrucción always@() se usa para poder usar condicionales o bucles y tiene su propio begin y end.
//posedge() sirve para identificar un flanco de subida en la señal que tenga entre su paréntesis, si se
//quisiera identificar un flanco de bajada se usaría la instrucción negedge().
always@(posedge(CLK_1Hz))
    begin
        if (T == 1'b1) begin //Modo Complemento: Invierte lo que estaba almacenado en la salida Q.
            QT = ~QT;
            QNT = ~QNT;
        end
        else begin //Modo Memoria: Deja igual lo que se había guardado anteriormente.
            QT = QT;
            QNT = ~QNT;
        end
    end
endmodule
```



Circuitos Combinacionales y Secuenciales

A grandes rasgos existen dos tipos de circuitos lógicos:

- **Circuitos combinacionales:** Todas **sus salidas son en función de sus entradas** en forma directa, **no almacenan información ni tienen entradas de retroalimentación**.
- **Circuitos secuenciales:** Sus **salidas dependen de sus entradas** como se realiza en los circuitos combinacionales, pero en este caso **si puede tener entradas de retroalimentación, además se tiene almacenamiento de información** (que se activa a través de la **señal de reloj**) y **sus salidas de igual forma dependen de su estado actual**.
 - Esta es la base de los sistemas de inteligencia artificial digitales, ya que deben estar conectados siempre a una memoria para aprender de casos pasados.

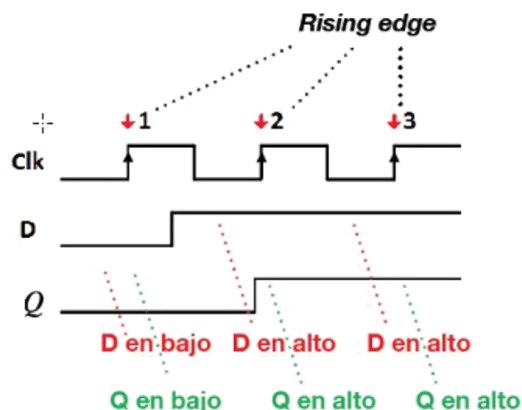


Los flip flops pertenecen a los circuitos secuenciales, porque dependen de la **señal de reloj** y de algunos valores guardados en memoria.

El flip flop más utilizado en los circuitos secuenciales de los 4 existentes es el **tipo D**, debido a su **simplicidad de uso**, pero lo que no se había mencionado antes es que **todos los tipos de memorias de 1 bit leen el estado de sus entradas cuando reciben el flanco de subida de una señal de reloj**, lo que vuelve al mismo **flip flop** en un circuito secuencial.

Registros y Shift Registers

Lo que sucede en el almacenamiento de datos es que, dentro del **flip flop tipo D** (que es la unidad de almacenamiento más pequeña) estará guardado **un 1 o 0 lógico**, **el cual previamente fue ingresado en su pin D y realmente se encontrará almacenado en la salida Q**, pero cuando en la entrada CLK del flip flop **se perciba el flanco de subida proveniente de una señal de reloj**, se analizará de nuevo la entrada **D**, para ver si el valor almacenado en la salida **Q** debe ser cambiado por otro.



D primero esta en bajo.

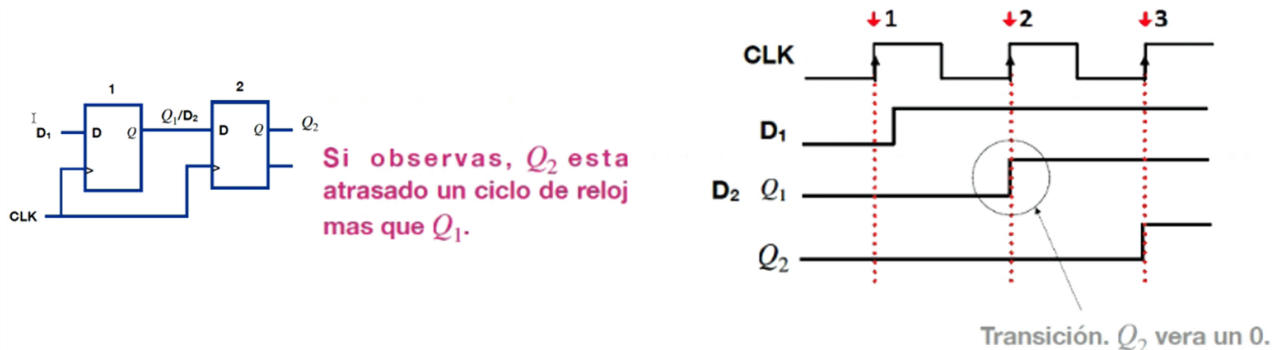
Entre el 1er y 2do ciclo reloj cambia el estado de D, de bajo a alto.

El FF estará atento a los cambios en D.

¿Cuándo registrará este cambio? En el siguiente **rising edge** del reloj.

Esto implica que **existe un cierto tiempo de retraso** cuando se reemplaza un dato por otro dentro del **flip flop**, ya que el nuevo dato se almacenará en **la salida Q solo cuando se perciba un nuevo flanco de subida en la señal de reloj**, por esta situación, se podrían conectar en serie **las salidas Q** con las **entradas D** de varios flip flops, logrando así que ese dato almacenado se traslade de un flip flop a otro antes de ser sustituido, a esto se le llama **registrar la información**, lo cual **permite guardar varios bits** que hayan sido introducidos a la **entrada D**, en vez de solo uno.

Cuando **más de un flip flop** esté conectado en serie, todos serán controlados por **1 misma señal de reloj**, dejando de ser una memoria de 1 bit y convirtiéndose en una de múltiples bits llamada **registro**.



En los **registros** donde se ingresen los datos por una vía (también llamados **shift register** o **registros de corrimiento**) el dato que ingrese a la **entrada D de un flip flop**, no se pasará al otro instantáneamente, se debe dejar pasar un número de **flancos de subida proporcional en la señal de reloj** para que esto ocurra.

Por ejemplo, para que un dato se guarde en la **salida Q2 de un segundo FF**, **deben haber pasado 2 flancos de subida en la señal de reloj**, como se describe a continuación:

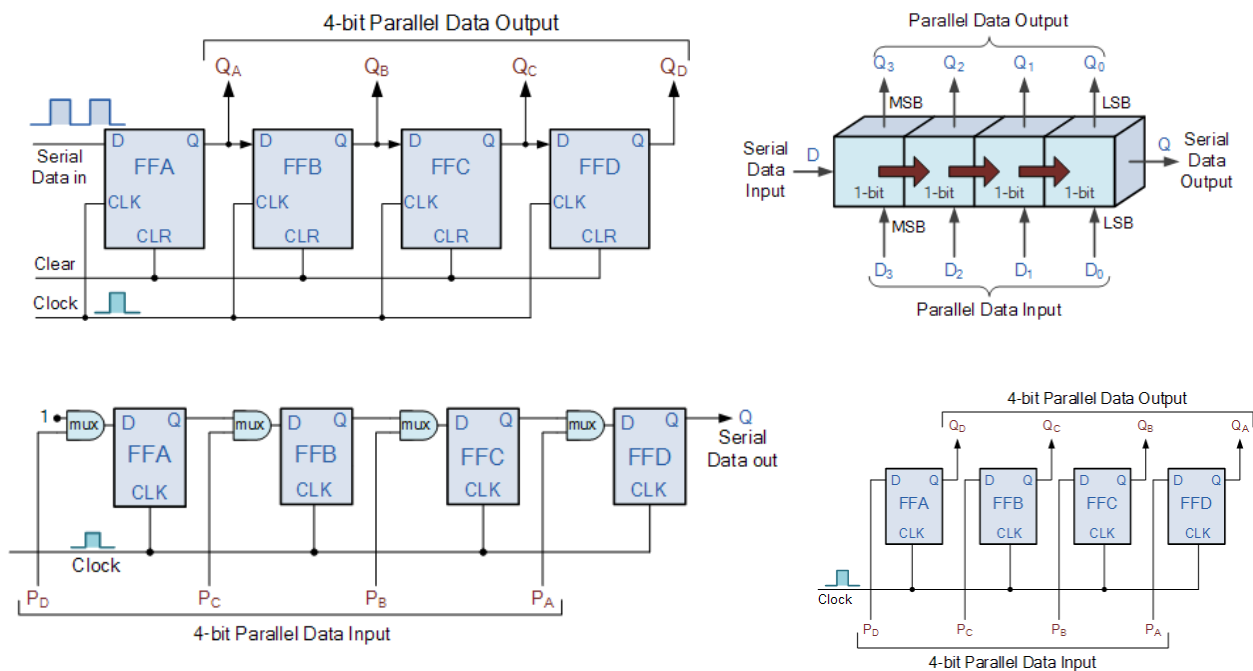
- **En el primer flanco:** La **entrada ingresa en D1** y pasa a su salida **Q1** cuando ocurra el **flanco clk**.
- **En el segundo flanco:** El dato pasará de la **salida Q1** a la **entrada D2** e inmediatamente se asignará a la **salida Q2**, por lo que pasaran dos ciclos de reloj para que el dato llegue desde **D1** hasta **Q2**.

Un registro implica una mayor capacidad de almacenamiento, en el ejemplo descrito anteriormente se podrá guardar un número binario de 2 bits. Donde, cada que llegue un bit nuevo, se recorrerá el que había en el **1er flip flop** (FF) hacia el **2do FF** y el dato que estaba almacenado en el **2do FF** se perderá.

Todos los FF de los **registros** siguen el **paso de los flancos de subida de una sola señal de reloj** al realizar el almacenamiento de datos, pero hay diferencia en la forma que se ingresan y se extraen los datos que se quieren almacenar, a continuación, se describen todas esas maneras:

- **SIPO (Serial Input Parallel Output):** El registro tiene una sola entrada y varias salidas, por lo que:
 - **Almacenamiento de datos:** Los datos son cargados en el registro a través de un solo cable, moviendo un bit a la vez, pasando el dato de **Flip Flop a Flip Flop** a través de su conexión en serie.
 - **Extracción de datos:** Los datos almacenados son extraídos a través de un **bus** de forma paralela, donde **un bus representa el conjunto de varios cables**, del cual se extraen los bits de información de forma individual en cada cable.
- **SISO (Serial Input Serial Output):** El registro tiene una sola entrada y salida, por lo que:

- **Almacenamiento y Extracción de datos:** Los datos son ingresados y extraídos del registro desplazándose un bit a la vez y a través de un solo cable (en serie).
- **PISO (Parallel Input Serial Output):** El registro tiene varias entradas y una sola salida, por lo que:
 - **Almacenamiento de datos:** Los datos son cargados al registro de forma paralela (a través de un bus).
 - **Extracción de datos:** Los datos almacenados son extraídos del registro desplazándose un bit a la vez y a través de un solo cable (en serie).
- **PIPO (Parallel Input Parallel Output):** El registro tiene varias entradas y salidas, por lo que:
 - **Almacenamiento de datos:** Los datos son cargados en el registro a través de un bus de forma paralela.
 - **Extracción de datos:** Los datos almacenados son extraídos a través de un bus de forma paralela.



A los **registros** que tienen una **entrada o salida serial** se les llama **registros de corrimiento** o **shift register**, porque mueven los bits de sus entradas o salidas a través de todos los flip flops que los conforman, siguiendo el paso que les dicta la frecuencia de la **señal de reloj**.

A continuación, se mostrará el código en VHDL que representa al **registro normal** y **shift register**, pero no se crearán estos programas en realidad, son solamente para la demostración de su funcionamiento:

Código VHDL: Registro

```
--1.-Registro: El registro es resultado de una conexión en serie de varios flip flops
--(usualmente de Tipo D) para crear así una memoria que no sea de 1 bit sino de varios,
--pero en el código de VHDL esto solo representa al codificar el mismo comportamiento de
--un flip flop de tipo D, donde la entrada simplemente se asigna a la salida.
--Cabe mencionar que todos los flip flops conectados en serie que conforman al registro
--siguen el paso marcado de una misma señal de reloj, además los registros que reciben
--sus datos a través de un bus (conjunto de varios cables), son nombrados como:
--PISO (Parallel Input Serial Output): Cuando su salida se recibe a través de un solo cable.
--PIPO (Parallel Input Parallel Output): Cuando su salida se recibe a través de un bus.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--Librerías para poder usar el lenguaje VHDL.
```

```

entity Registro is
    Port ( clk : in  STD_LOGIC; --Señal de reloj para el almacenamiento de datos del registro.
          --El código de un registro simple se codifica como un flip flop tipo D, pero manejando
          --vectores en vez de entradas y salidas de 1 bit.
          entradaRegistro : in  STD_LOGIC_VECTOR (7 downto 0);
          salidaRegistro : out  STD_LOGIC_VECTOR (7 downto 0)
        );
end Registro;

architecture vectorFlipFlop of Registro is
begin
    --La instrucción process se utiliza para poder ejecutar condicionales y bucles, en su paréntesis
    --recibe las entradas que vayan a estar involucradas.
    process(clk, entradaRegistro) begin
        --La instrucción rising_edge indica que cada que ocurra un flanco de subida en el reloj, se
        --ejecute una acción en específico, en este caso esa acción es que se trasladen los datos
        --de la entrada a la salida del registro.
        --SI se quisiera ejecutar esta acción con el flanco de bajada de la señal de reloj se debe
        --usar la instrucción fallin_edge.
        if(rising_edge(clk)) then
            salidaRegistro <= entradaRegistro;
        end if;
    end process;
end vectorFlipFlop;

```

Código VHDL: Shift Register

--Shift Register: El registro de corrimiento es resultado de una conexión en serie de
--varios flip flops (usualmente de Tipo D) para crear así una memoria que no sea de 1 bit
--sino de varios, pero en el código de VHDL esto solo representa al codificar el mismo
--comportamiento de un flip flop tipo D, donde la entrada simplemente se asigna a la salida.
--Cabe mencionar que todos los flip flops conectados en serie que conforman al registro
--siguen el paso marcado de una misma señal de reloj, además los registros que reciben
--sus datos a través de un bus (conjunto de varios cables), son nombrados como:
--SISO (Serial Input Serial Output): Su entrada y salida se reciben a través de un solo cable.
--SIPO (Serial Input Parallel Output): Su entrada se recibe con un cable y su salida a través
--de un bus.

```

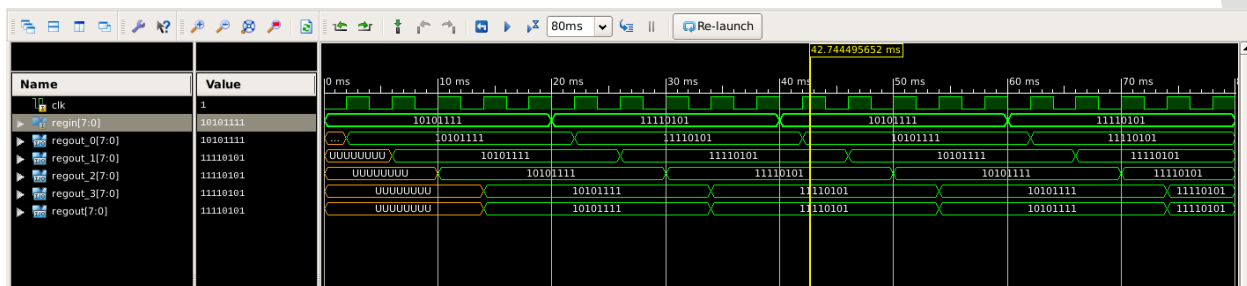
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--Librerías para poder usar el lenguaje VHDL.

entity ShiftRegister is
    Port ( clk : in  STD_LOGIC; --Señal de reloj para el almacenamiento de datos del registro.
          --El código de un registro simple se codifica como un flip flop tipo D, pero manejando
          --vectores en vez de entradas y salidas de 1 bit.
          RegIn : in  STD_LOGIC_VECTOR (7 downto 0);
          RegOut_0 : inout STD_LOGIC_VECTOR (7 downto 0);
          RegOut_1 : inout STD_LOGIC_VECTOR (7 downto 0);
          RegOut_2 : inout STD_LOGIC_VECTOR (7 downto 0);
          RegOut_3 : inout STD_LOGIC_VECTOR (7 downto 0);
          RegOut : inout STD_LOGIC_VECTOR (7 downto 0)
        );
end ShiftRegister;

architecture vectorFlipFlop of ShiftRegister is
begin
    --La instrucción process se utiliza para poder ejecutar condicionales y bucles, en su paréntesis
    --recibe las entradas que vayan a estar involucradas.
    process(clk, RegIn, RegOut_0, RegOut_1, RegOut_2, RegOut_3) begin
        --La instrucción rising_edge indica que cada que ocurra un flanco de subida en el reloj, se
        --ejecute una acción en específico, en este caso esa acción es que se trasladen los datos
        --de la entrada a la salida del registro.
        --SI se quisiera ejecutar esta acción con el flanco de bajada de la señal de reloj se debe
        --usar la instrucción fallin_edge.
        if(rising_edge(clk)) then
            RegOut_0 <= RegIn;
            RegOut_1 <= RegOut_0;
            RegOut_2 <= RegOut_1;
            RegOut_3 <= RegOut_2;
        end if;
        RegOut <= RegOut_3;
    end process;
end vectorFlipFlop;

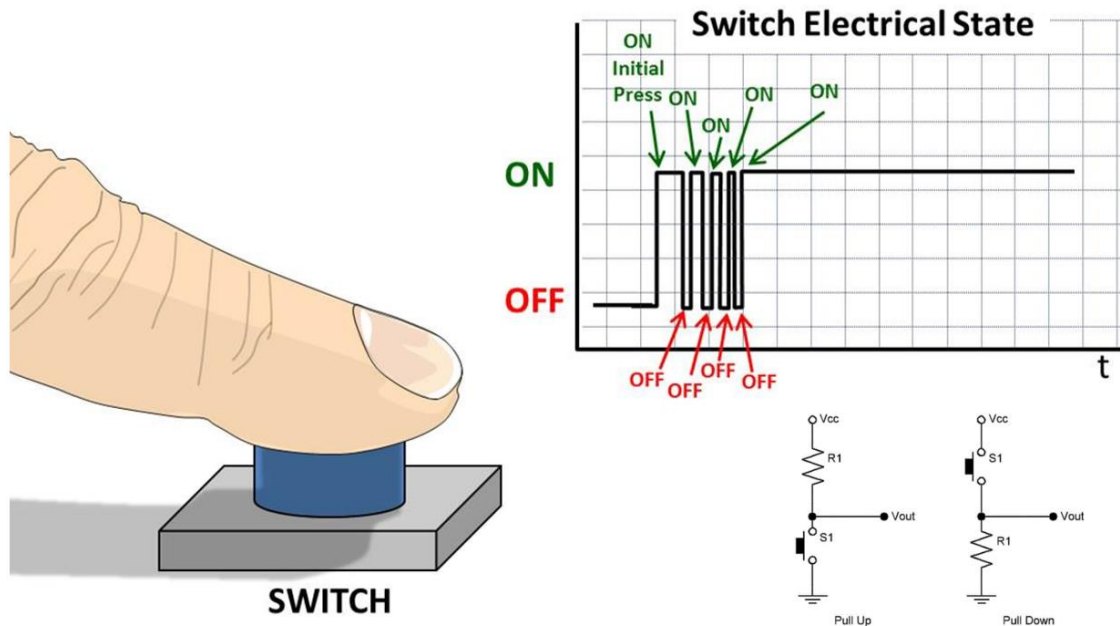
```

Simulación: Shift Register

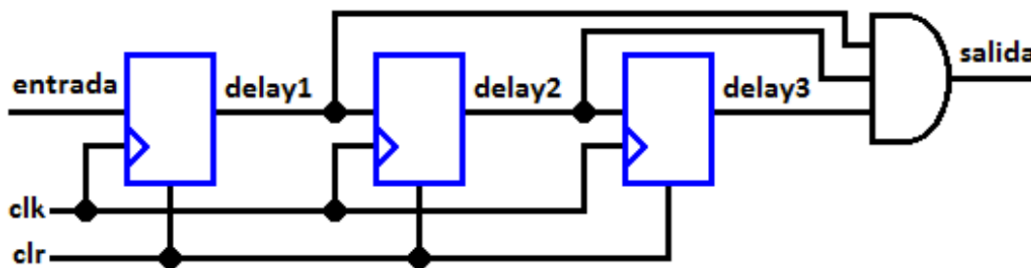


Retardo o Delay Antirrebotes (Delay, Millis y Shift Register)

Los rebotes son falsas pulsaciones que se producen al presionar un botón o interruptor. El código de antirrebotes se utiliza para evitar que sean detectadas las señales no deseadas causadas por los rebotes mecánicos al accionar un switch, donde al accionarse tienen un efecto similar al de cuando se suelta una pelota y esta va rebotando contra el piso hasta que se detiene, el rebote siempre se dará no importando si se coloca una resistencia pull up o pull down para detectar el cambio de tensión.



En un código de Arduino esto se haría a través de un temporizador que detenga la ejecución del programa por unos milisegundos, utilizando el método `millis()` o `delay()`, para así permitir que pasen las falsas pulsaciones, pero en el código de VHDL y Verilog esto se debe realizar a través de un **shift register** que retrase la señal, cabe mencionar que en este código se agrega una entrada llamada **clear** que será **asíncrona**, esto se refiere a que **no se acciona siguiendo el paso de los flancos de subida de la señal de reloj**, sino que se ejecuta en cualquier momento.



El tiempo de retraso en el antirrebotes ocasionado por el **shift register** se calcula de la siguiente manera:

$$t_{delay} = T_{CLK} \left(\#delays - \frac{1}{2} \right) = \frac{1}{f_{CLK}} \left(\#delays - \frac{1}{2} \right)$$

$$f_{CLK} = \frac{1}{T_{CLK}} = \frac{1}{t_{delay}} \left(\#delays - \frac{1}{2} \right)$$

A continuación, se presentará el código de antirrebotes en Arduino, VHDL y Verilog para implementarse con N bits.

Código Arduino: millis o delay() - Antirrebotes:

```
void ANTIRREBOTES(){
  //ANTIRREBOTES CON DELAY DE 5 MILISEGUNDOS: Para ello se debe medir el tiempo.
  /*unsigned long: Variable sin signo de 32 bits, puede abarcar números muy grandes de 0 a 4,294,967,295.*/
  unsigned long ultimaInterrupcion = 0;
  /*millis(): Método que devuelve el tiempo transcurrido en milisegundos, empezando a contar desde que se
  enciende la placa Arduino y no deteniéndose hasta que esta se apague o llegue a su límite, que es el
  mismo dado por el tipo de dato unsigned long: De 0 a 4,294,967,295 milisegundos = 1,193.0464 horas
  49.7102 días = 49 días y 17 horas.
  La función se puede utilizar para detener el programa, sustituyendo al método delay(), la razón de ello
  es que el método delay() detiene completamente la ejecución del programa, mientras que el método millis()
  permite que otros procesos se ejecuten mientras solo una parte del código se pausa.*/
  unsigned long tiempoInterrupcion = millis();

  if(tiempoInterrupcion - ultimaInterrupcion > 5){

    //CÓDIGO A EJECUTARSE DESPUÉS DE HABER DEJADO EL TIEMPO DE ESPERA DEL ANTIRREBOTES

    /*Actualización del tiempo guardado en la variable ultimaInterrupcion para así ejecutar de nuevo el
    código antirrebotes la próxima vez que sea activada la interrupción.*/
    ultimaInterrupcion = tiempoInterrupcion;
  }
}
```

Código Verilog: Shift Register - Antirrebotes:

```
//Antirrebotes: Una de las aplicaciones del registro de corrimiento, que es resultado de una
//conexión en serie de varios flip flops (usualmente de Tipo D), donde simplemente se conecta
//la entrada de un FF tipo D a la salida de otro siguiendo el paso marcado de una misma señal
//de reloj, es la del código antirrebotes, que sirve para crear un pequeño retraso de tiempo en
//el programa, que deje pasar las falsas pulsaciones que se producen al presionar un botón.
module shiftRegisterAntirrebotes(
    input clk,           //Señal de reloj.
    input clr,           //Botón de clr para reiniciar el conteo del debounce.
    //El código de un shift register se codifica como un flip flop tipo D, asignando de
    //forma temporal valores de forma directa a signals.
    input entradaBoton,  //Botón que genera los rebotes.
    output salidaBoton   //Salida sin rebotes.
);

//Antes del always pero dentro del módulo es donde se declaran las variables de tipo reg e
//integer que no son ni entradas ni salidas, solo existen durante la ejecución del programa
//e interactúan con el condicional o bucle de la instrucción process.
reg delay_1;
reg delay_2;
reg delay_3;

//always@ se usa para poder usar condicionales o bucles y tiene su propio begin y end.
//POSEDGE: La instrucción posedge() solo puede tener una entrada o reg dentro de su paréntesis
//y a fuerza se debe declarar en el paréntesis del always@(), además hace que los condicionales
//o bucles que estén dentro del always@() se ejecuten por si solos cuando ocurra un flanco de
//subida en la entrada que tiene posedge() dentro de su paréntesis, el flanco de subida ocurre
//cuando la entrada pasa de valer 0 lógico a valer 1 lógico y el hecho de que la instrucción
//posedge() haga que el código se ejecute por si solo significa que yo directamente no debo
//indicarlo con una operación lógica en el paréntesis de los condicionales o bucles, si lo hago
//el programa me arrojará error, aunque si quiero que se ejecute una acción en especifico cuando
//se dé el flanco de subida en solo una de las entradas que usan posedge(), debo meter el nombre
//de esa entrada en el paréntesis del condicional o bucle, también si uso un posedge, todas las
//entradas deben ser activadas igual por un posedge.
//Si se quisiera ejecutar esta acción con el flanco de bajada de la señal de reloj se debe
//usar la instrucción negedge().
always@(posedge(clk), posedge(clr))
begin
    if(clr) begin           //Si el botón clr es presionado el conteo del debounce se reinicia.
        delay_1 = 1'b0;
        delay_2 = 1'b0;
        delay_3 = 1'b0;
    end

    //Cada que ocurra un flanco de subida en el reloj, se trasladaran los datos de entrada a la
    //salida del registro de corrimiento, pero como pasa a través de varias signal, esto es
    //lo que ocasionará el retraso en tiempo, que genera el delay, este tiempo estará sujeto a
    //la frecuencia de la señal de reloj, ya que cada que se perciba un flanco de subida, con el
    //diferente número de signals, se estará esperando el tiempo del periodo de la señal entre 2.
    //t_delay = (periodoReloj/2) + ((#signals delay-1) * periodoReloj)
    //t_delay = (1/frecuenciaReloj)/2 + ((#signals delay-1) * (1/frecuenciaReloj))
    //Si se quisiera ejecutar esta acción con el flanco de bajada de la señal de reloj se debe
    //usar la instrucción negedge() dentro del always@() de arriba.
    else begin
        //No debo poner el caso cuando if(clkNexys2) porque eso ya lo está haciendo la instrucción
        //always@(posedge()).
        delay_1 = entradaBoton;
        delay_2 = delay_1;
        delay_3 = delay_2;
    end
end
```

```

end
//Con el simbolo & se concatenan los valores de cada delay y de esta forma se estará
//dejando que pasen #signals_delay-1 = 2 ciclos de reloj para no captar los pulsos falsos al
//presionar el botón.
//Para asignar el valor de un reg a una salida se usa la palabra reservada assign.
assign salidaBoton = delay_1 & delay_2 & delay_3;
endmodule

```

Código VHDL: Shift Register - Antirrebotes:

```

--Antirrebotes: Una de las aplicaciones del registro de corrimiento, que es resultado de una
--conexión en serie de varios flip flops (usualmente de Tipo D), donde simplemente se conecta
--la entrada de un FF tipo D a la salida de otro siguiendo el paso marcado de una misma señal
--de reloj, es la del código antirrebotes, que sirve para crear un pequeño retraso de tiempo en
--el programa, que deje pasar las falsas pulsaciones que se producen al presionar un botón.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--Librerías para poder usar el lenguaje VHDL.

entity shiftRegisterAntirrebotes is
    Port ( clk : in  STD_LOGIC;          --Señal de reloj.
          clr : in  STD_LOGIC;          --Botón de clr para reiniciar el conteo del debounce.
          -- --El código de un shift register se codifica como un flip flop tipo D, asignando de
          -- forma temporal valores de forma directa a signals.
          entradaBoton : in  STD_LOGIC;  --Botón que genera los rebotes.
          salidaBoton : out STD_LOGIC    --Salida sin rebotes.
    );
end shiftRegisterAntirrebotes;

architecture debounce of shiftRegisterAntirrebotes is
    --Dentro del process pero antes de su begin es donde se declaran las variables de tipo signal
    --que no son ni entradas ni salidas, solo existen durante la ejecución del programa e interactúan
    --con el condicional o bucle de la instrucción process.
    signal delay_1 : STD_LOGIC;
    signal delay_2 : STD_LOGIC;
    signal delay_3 : STD_LOGIC;

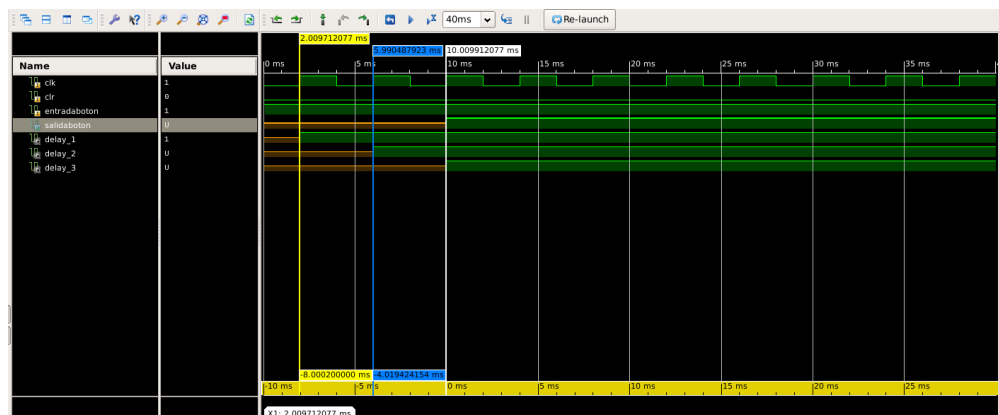
begin
    --La instrucción process se utiliza para poder ejecutar condicionales y bucles, en su
    --paréntesis se indican las entradas que vayan a estar involucradas.
    process(clk, clr, entradaBoton) begin
        if(clr = '1') then --Si el botón clr es presionado el conteo del debounce se reinicia.
            delay_1 <= '0';
            delay_2 <= '0';
            delay_3 <= '0';

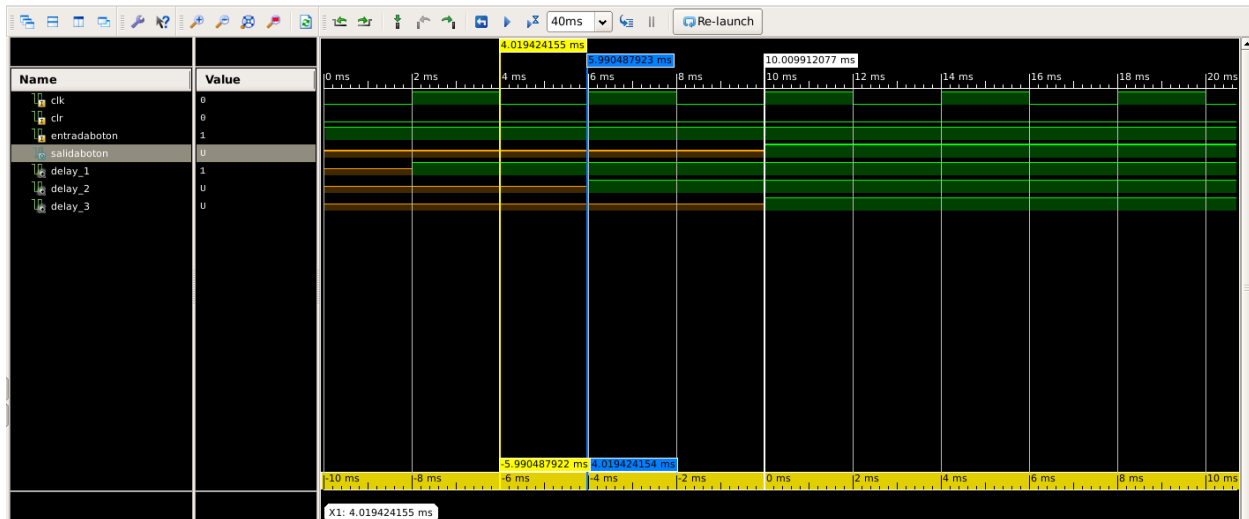
            --La instrucción rising_edge indica que cada que ocurra un flanco de subida en el reloj, se
            --ejecute una acción en específico, en este caso esa acción es que se trasladen los datos
            --de la entrada a la salida del registro, pero como pasa a través de varias signal, esto es
            --lo que ocasionará el retraso en tiempo, que genera el delay, este tiempo estará sujeto a
            --la frecuencia de la señal de reloj, ya que cada que se perciba un flanco de subida, con el
            --diferente número de signals, se estará esperando el tiempo del periodo de la señal entre 2.
            --t_delay = (periodoRelej/2) + ((#signals delay-1) * periodoRelej)
            --t_delay = (1/frecuenciaRelej)/2 + ((#signals delay-1) * (1/frecuenciaRelej))
            --Si se quisiera ejecutar esta acción con el flanco de bajada de la señal de reloj se debe
            --usar la instrucción fallin_edge.
            elsif(rising_edge(clk)) then --El tiempo de retardo se ejecuta en función de la señal clk.
                delay_1 <= entradaBoton;
                delay_2 <= delay_1;
                delay_3 <= delay_2;
            end if;
        end process;

        --Con la instrucción and se concatenan los valores de cada delay y de esta forma se estará
        --dejando que pasen #signals_delay-1 = 2 ciclos de reloj para no captar los pulsos falsos al
        --presionar el botón.
        salidaBoton <= delay_1 and delay_2 and delay_3;
    end debounce;
end

```

Simulación: Shift Register - Antirrebotes: $T_{CLK} = 4ms$; $\#delays = 3$; $t_{delay} = 10ms$



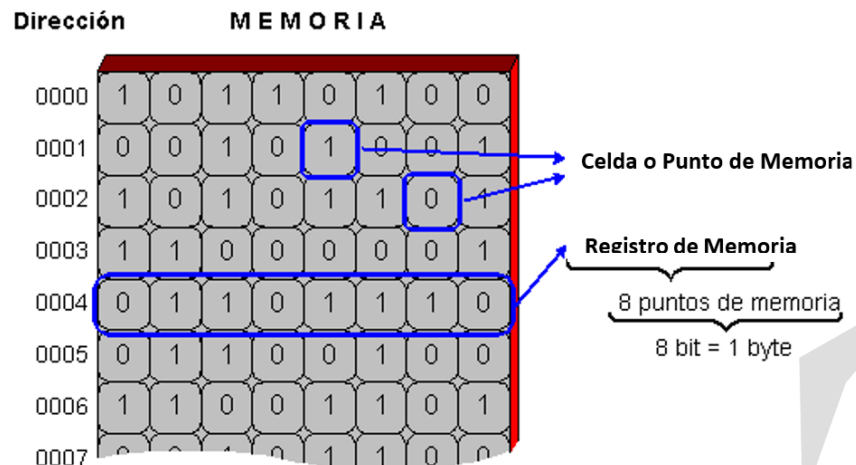


Memoria RAM y ROM

Una **memoria** no es más que un dispositivo en forma de **matriz de bits** donde se almacenan valores binarios de **0** y **1**, esto se puede hacer de diferentes maneras, ya sea con flip flops, condensadores, discos magnéticos, semiconductores, etc. A continuación, se describen sus partes importantes:

- **Celda o Punto:** Representa una parte de la memoria donde se puede almacenar solo un bit, ya sea con valor **0** o **1**.
- **Registro o Posición:** Un registro está conformado por una **fila de varias celdas conectadas en serie**, donde ya no solo se puede guardar un bit, sino un número binario de varios bits.
- **Memoria:** Dispositivo que puede guardar varios números binarios y está conformada por **varias filas de registros**. Su **capacidad** es descrita por el número de registros y celdas que la conforman:

$$\#Registros \times \#Celdas = \#BitsQuePuedeAlmacenar = \frac{\#Bits}{8} = \#BytesQuePuedeAlmacenar$$
 - La **capacidad de una memoria** indica cuantos bits puede albergar.
 - Al describir la capacidad de una memoria es usual utilizar la unidad de **byte**, que corresponde a **8 bits**.



Capacidad de Memoria

La unidad de medida de las memorias no es la misma a la medida decimal, aunque se diga igual, el término y valor de la capacidad de las memorias se mide en **bibyte**, que corresponde no a un exponencial con base 10, sino a un exponencial con base 2. Se mide de esta manera porque $2^{10} = 1024$ y ese valor es el más cercano al 1000, que corresponde al kilo de la base decimal, de ahí nos basamos para lo demás.

Decimal System (SI)			Binary System		
Name	Symbol	Decimal Unit	Name	Symbol	Decimal Unit
Kilobyte	KB	10^3	Kibibyte	KiB	2^{10}
Megabyte	MB	10^6	Mebibyte	MiB	2^{20}
Gigabyte	GB	10^9	Gigibyte	GiB	2^{30}
Terabyte	TB	10^{12}	Tebibyte	TiB	2^{40}
Petabyte	PB	10^{15}	Pebibyte	PiB	2^{50}
Exabyte	EB	10^{18}	Exbibyte	EiB	2^{60}
Zettabyte	ZB	10^{24}	Zebibyte	ZiB	2^{70}
Yottabyte	YB	10^{21}	Yobibyte	YiB	2^{80}

atatus

Por ejemplo, si se tiene una memoria con **4G de registros** y **32 celdas**, entonces se realiza el siguiente procedimiento para saber su **capacidad**:

$$\text{Capacidad} = \# \text{Registros} \times \# \text{Celdas} = \# \text{Bits Que Puede Almacenar} = \frac{\# \text{Bits}}{8} = \# \text{Bytes Que Puede Almacenar}$$

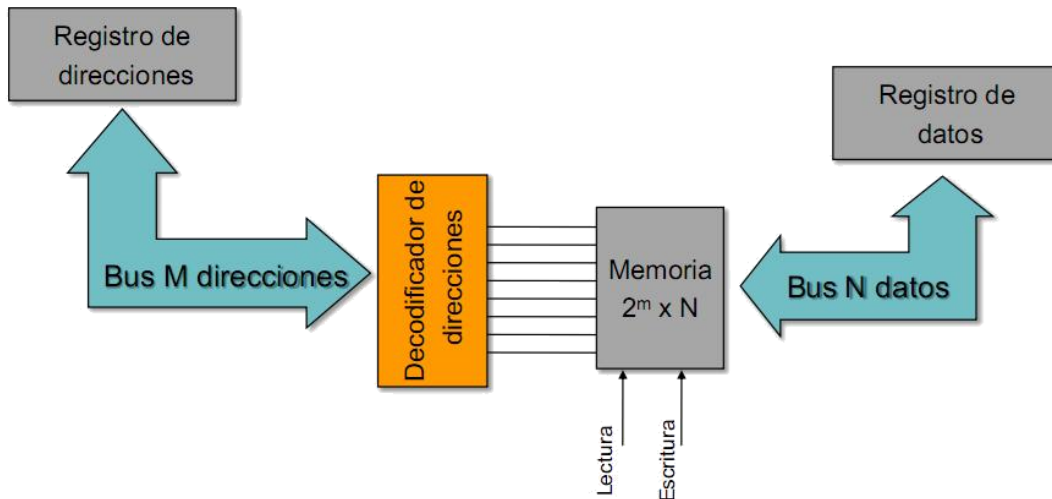
$$\text{Capacidad} = 4\text{G} \times 32 = 4 * 2^{30} * 32 = 128 \text{ Gigibits} = \frac{128 \text{ Gigibits}}{8} = 16 \text{ Gigibyte} = 16 [\text{Gigabytes}]$$

Leer o Escribir Información en una Memoria

- **Dirección de memoria:** Para acceder al contenido de una memoria se debe indicar la **posición, registro o fila de su matriz**, a esto se le llama **dirección de memoria** y sus posiciones se cuentan desde cero de forma binaria. **Como esto se realiza de forma binaria, se debe hacer a través de varios cables** (llamados **Bus de direcciones o ADDRESS BUS**).
 - **Bus de datos o DATA BUS:** La información que haya solicitado a través de una **dirección de memoria** se me retorna en un bus, llamado **bus de datos** y es simplemente una serie de cables, donde se retorna en cada uno 1 bit de información, ya sea con valor 0 o 1.
- **Memoria ROM (Read Only Memory):** A este tipo de memoria no se le pueden ingresar datos, solo se le pueden extraer. Sus datos almacenados no se pierden al retirar su alimentación y el propósito de estos es, no ser alterados mientras el sistema está en funcionamiento.
- **Memoria RAM (Random Access Memory):** A este tipo de memoria se le pueden ingresar o extraer datos durante el funcionamiento del sistema, pero solamente estarán almacenados en un corto tiempo, al remover la alimentación, sus datos serán eliminados.

- **R/W**: Para decidir si la memoria **almacena (Write)** o **devuelve (Read)** datos, existe un cable de **lectura y escritura (R/W)**, donde dependiendo del estado en el que se encuentre este cable, el **Bus de datos escribirá en la memoria** o **me proporcionará los datos almacenados en ella**. Por lo cual podemos decir que **ninguna memoria se borra, todas se sobrescriben con 0**.

Esquema de funcionamiento de una memoria: lectura/escritura



Si sabemos la **capacidad** de una memoria, podemos hacer el proceso inverso y **a través del número de cables del bus de datos**, calcular el número de **registros** que la conforman y cuantas **celdas** tienen. A continuación, se resolverá este ejercicio con una memoria que tiene una capacidad de **4GB** y un **bus de datos** con **5 cables**:

$$\text{Capacidad} = \# \text{Registros} \times \# \text{Celdas} = \# \text{Direcciones} \times \# \text{Palabra} = \# \text{Direcciones} \times \# \text{Datos}$$

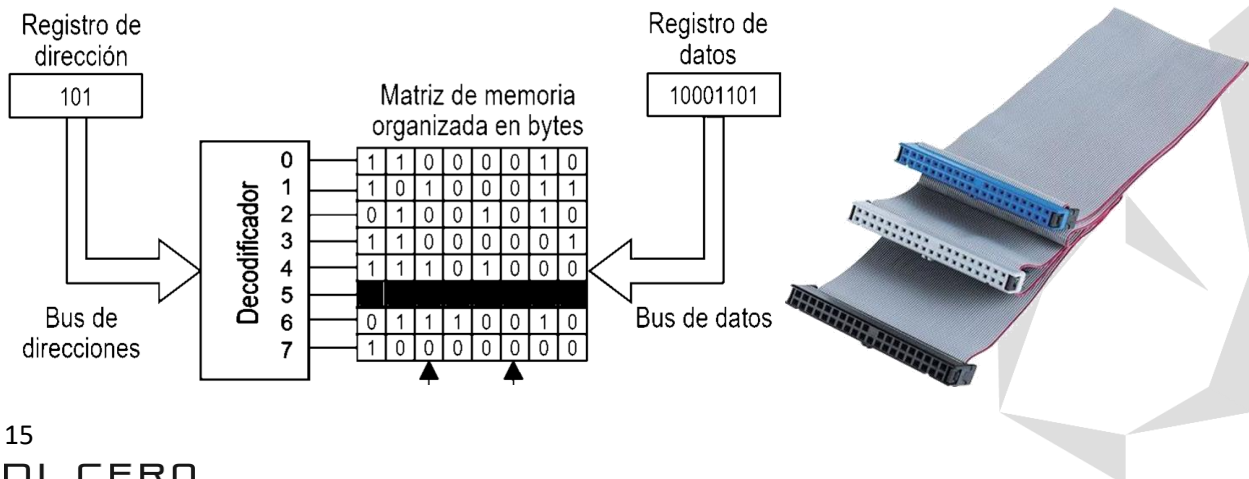
$$\# \text{Datos} = 2^{\# \text{Cables}}$$

$$\text{Capacidad} = 4 [\text{Gigabytes}] = 4 \text{ Gigibyte} = 8 * 4 \text{ Gigibyte} = 32 \text{ Gigibits} = \# \text{Direcciones} \times \# \text{Datos}$$

$$\# \text{Datos} = 2^5 = 32$$

$$\# \text{Direcciones} = \frac{32 \text{ Gigibits}}{32} = 1 \text{ Gibibit} \therefore \text{El bus de direcciones tendrá 30 cables} = 1 \text{ Gibibit} = 2^{30}$$

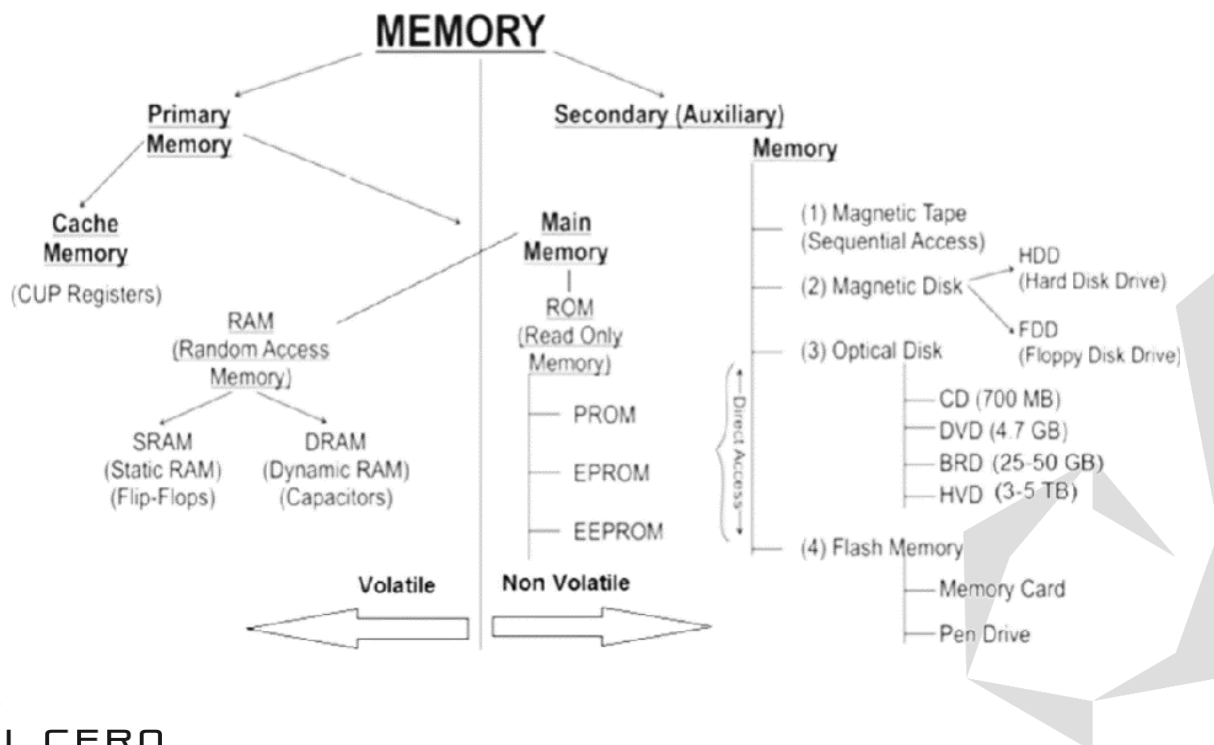
El resultado se obtuvo al ver la tabla de equivalencias entre el sistema decimal y binario.



Tipos de Memorias

Existen diferentes formas de clasificar a las memorias, ya sea por su tipo de almacenamiento de forma **volátil** o **no volátil** (dependiendo de si pueden retener su información o no cuando se les retira su fuente eléctrica), su tecnología de fabricación, capacidad de almacenamiento, etc. Siendo la más común la que se muestra a continuación:

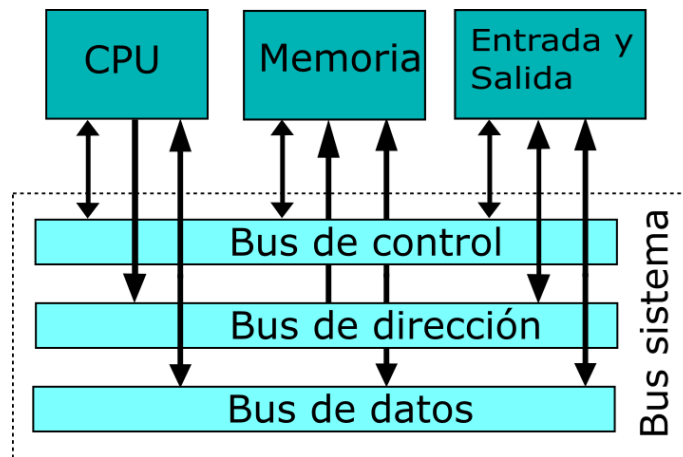
- **Memoria Volátil:** No conserva su información al ser removida su fuente de alimentación:
 - **RAM (Random Access Memory):** Memoria de lectura y escritura estática o dinámica.
 - **Memoria Caché:** Memoria intermedia que se encuentra ubicada entre el CPU y la memoria **RAM** de una computadora. Su principal objetivo es mejorar el rendimiento y la eficiencia del acceso a los datos que la CPU necesita con mayor frecuencia.
- **Memoria NO Volátil:** Conserva su información después de ser removida su fuente de alimentación:
 - **ROM (Read Only Memory):** Memoria de solo lectura.
 - **PROM (Programmable Read Only Memory):** Memoria que se programa una sola vez. Una vez programada, no puede ser modificada.
 - **EPROM (Erasable Programmable Read Only Memory):** Memoria similar a la **PROM**, pero con la ventaja de que puede ser borrada y actualizada múltiples veces a través de ser expuesta a una luz ultravioleta durante un tiempo específico.
 - **EEPROM (Electrically Erasable Programmable Read Only Memory):** Memoria de lectura y escritura que puede ser programada y actualizada eléctricamente sin necesidad de exposición a luz ultravioleta.
 - **FLASH (EEPROM):** Memoria parecida a la **EEPROM**, pero con una capacidad de almacenamiento más grande y una velocidad de escritura y lectura mayor. Se utiliza en dispositivos de almacenamiento portátil como unidades USB, tarjetas de memoria, SSD (Solid State Drives), etc.



Además, cabe mencionar que la **manufactura de las memorias permanentes (no volátiles)** ya en productos reales **puede pertenecer a las siguientes dos grandes clasificaciones**:

- **Disco duro, óptico o memoria HDD (Hard Drive Disk)**: En este tipo de memoria se usan láseres para quemar hoyos en sustratos reflectantes, donde la información binaria es representada por la luz ya sea reflejada, o no reflejada.
- **Unidad de estado sólido o memoria SDD (Solid State Drive)**: La memoria de semiconductores no tiene partes móviles, por lo que se le llama memoria de estado sólido y puede contener más información por unidad área que la memoria del disco, además de que accede a ella de forma más rápida y no se deteriora tan rápido como la HDD.

Finalmente, vale la pena mencionar que las memorias no actúan solas y no siempre están conectadas por medio de cables, a veces internamente se encuentran en comunicación con otras partes dentro de un mismo sistema, como lo puede ser un CPU, microcontrolador, etc.



Memorias RAM y ROM: Array en Verilog y VHDL

En conclusión, a la información guardada en una memoria se le llama **datos (data)**. Cuando escribimos en una memoria, la acción es de **escritura (write)** y cuando obtenemos información de una memoria lo que hacemos es **leerla (read)**. Para acceder a la información requerimos de una **dirección (address)**.

Memory Map Model



Cuando se crea un código en Arduino, la memoria se puede ejemplificar simplemente al asignar el valor de una variable o array, debido a que el proceso de indicar la dirección de memoria y demás pasos los hace el mismo lenguaje internamente; pero en lenguajes HDL se puede diseñar este sistema de forma más visible, por lo que a continuación, se mostrará el código en Verilog y VHDL que representa la **memoria ROM (Read Only Memory)**, **RAM (Random Access Memory)** y una combinación de ambas, más que nada para ejemplificar la creación de arrays en ambos lenguajes, se utiliza un switch RW para seleccionar si la

memoria RAM se lee o escribe y un switch CS para elegir cuál de los dos contenidos de las memorias RAM o ROM se verá reflejado en la salida.

Código Verilog - Memoria ROM/RAM:

```
//1.-Memorias RAM y ROM: A la información guardada en una memoria se le llama datos (data). Cuando escribimos
//en una memoria, la acción es de escritura (write) y cuando obtenemos información de una memoria lo que
//hacemos es leerla (read). Para acceder a la información requerimos de una dirección (address).
//A las memorias ROM (Read Only Memory) solo se les puede leer sus datos, mientras que a las memorias RAM
//(Random Access Memory) se les puede leer o escribir datos cuando sea.
//En este caso se creará una memoria de 8 registros (filas) con 8 celdas (columnas) cada uno, por lo que tendrá
//una capacidad de 8x8 = 64 bits = 8 bytes, por lo que necesitara un memory adress de 3 bits para que pueda
//contar de 0 a 8.

module memoriaRAM_ROM(
    input clk,                //Reloj de 50MHz proporcionado por la NEXYS 2 en el puerto B8.
    //Las acciones de la memoria pueden ser sincronicas: Siguen el paso de la señal de reloj.
    //O pueden ser asincronicas: Se ejecutan en cualquier momento no importando la señal de reloj.
    input [2:0] adressBus,    //Dirección de memoria.
    input RW,                //Switch RW (Read/Write) para escribir o leer la memoria RAM.
    //Con RW = 1 se puede leer el contenido de ambas memorias, con RW = 0 se escribe en la RAM.
    input [7:0] dataInRAM,    //Bus de datos ingresados a la memoria RAM.
    input CS,                //CS (Chip Selector) para saber si la salida es de la RAM o ROM.
    //Con CS = 1 se elige leer el contenido de la ROM, con CS = 0 se lee el contenido de la RAM.
    output reg [7:0] dataBus   //Bus de datos extraídos de la memoria RAM.
    //En Verilog todas las salidas que se vaya a usar en los condicionales debo declararlas como reg, osea
    //Register, los tipos de salidas reg lo que van a hacer es almacenar valores temporalmente.
);

//Antes del always pero dentro del módulo es donde se declaran las variables de tipo
//reg e integer que interactúan con los condicionales o bucles.
//Con una única línea de código se puede declarar un array tipo matriz de 8 elementos, cada uno con
//longitud de 8 bits, hay que tomar en cuenta que los bits se cuentan de 7 a 0, pero las posiciones del
//array se cuentan de 1 a 8, esto se realiza con la siguiente nomenclatura:
//tipoDeDato [tamanoElementos] nombreTipoDeDato [numeroDeElementos];
reg [7:0] ROM [1:8];
initial begin
    ROM[1] <= 8'b00000000;
    ROM[2] <= 8'b00000001;
    ROM[3] <= 8'b00000010;
    ROM[4] <= 8'b00000011;
    ROM[5] <= 8'b00000100;
    ROM[6] <= 8'b00000101;
    ROM[7] <= 8'b00000110;
    ROM[8] <= 8'b00000111;
end
//REG: No es ni una entrada ni una salida porque no puede estar vinculada a ningún puerto de la NEXYS 2,
//solo existe durante la ejecución del código y sirve para poder almacenar algún valor de forma temporal.
//En las memorias RAM, como se les puede escribir o leer datos, se declara como signal.
//No hay una forma de declarar un array constante para la ROM, por eso igual se declara como reg.
reg [7:0] RAM [1:8];
initial begin
    RAM[1] <= 8'b00000000;
    RAM[2] <= 8'b10000001;
    RAM[3] <= 8'b01000010;
    RAM[4] <= 8'b11000011;
    RAM[5] <= 8'b00100100;
    RAM[6] <= 8'b10100101;
    RAM[7] <= 8'b01100110;
    RAM[8] <= 8'b11100111;
end
//Se usan reg adicionales para guardar temporalmente el contenido de las memorias.
reg [7:0] dataOutROM;        //Bus de datos extraídos de la memoria ROM.
reg [7:0] dataOutRAM;        //Bus de datos extraídos de la memoria RAM.

//La obtención de datos de la memoria ROM es sincrónica porque depende de la señal de reloj, no es así en
//todas las memorias RAM o ROM, puede ser como sea, pero aquí se hace así para mostrar sus diferencias.
//always@ se usa para poder usar condicionales o bucles y tiene su propio begin y end.
//POSEDGE: La instrucción posedge() solo puede tener una entrada o reg dentro de su paréntesis
//y a fuerza se debe declarar en el paréntesis del always@(), además hace que los condicionales
//o bucles que estén dentro del always@() se ejecuten por si solos cuando ocurra un flanco de
//subida en la entrada que tiene posedge() dentro de su paréntesis, el flanco de subida ocurre
//cuando la entrada pasa de valer 0 lógico a valer 1 lógico y el hecho de que la instrucción
//posedge() haga que el código se ejecute por si solo significa que yo directamente no debo
//indicarlo con una operación lógica en el paréntesis de los condicionales o bucles, si lo hago
//el programa me arrojará error, aunque si quiero que se ejecute una acción en específico cuando
//se dé el flanco de subida en solo una de las entradas que usan posedge(), debo meter el nombre
//de esa entrada en el paréntesis del condicional o bucle, también si uso un posedge, todas las
//entradas deben ser activadas igual por un posedge.
//Si se quisiera ejecutar esta acción con el flanco de bajada de la señal de reloj se debe
//usar la instrucción negedge().
always@(posedge(clk)) begin
    //La instrucción posedge() hace que este condicional solo se ejecute cuando ocurra un flanco de
    //subida en la señal de reloj clk proveniente de la NEXYS 2.
    //$unsigned(): Método que convierte un numero binario en uno entero. Además, cabe mencionar que
    //a esta conversión se le suma un 1 porque para acceder a las posiciones de un array se cuenta
    //desde 1, no desde 0.
    dataOutROM = ROM[$unsigned(adressBus)+1];                //Lectura de la memoria ROM.
end

//La lectura y escritura de datos en la memoria RAM es asíncrona porque NO depende de la señal de reloj.
```

```

always@(RW or dataInRAM or adressBus) begin
    //Si el switch RW está en 1 lógico, la memoria es de lectura, sino es de escritura.
    if (RW == 1'b1) begin
        dataOutRAM = RAM[$unsigned(adressBus)+1]; //Lectura de la memoria RAM.
    end else begin
        RAM[$unsigned(adressBus)+1] = dataInRAM; //Escritura de la memoria RAM.
        dataOutRAM = RAM[$unsigned(adressBus)+1]; //Lectura de la memoria RAM.
    end
end

always@(CS or dataOutROM or dataOutRAM) begin
    //Si el switch RW está en 1 lógico, el data bus muestra la salida de la memoria ROM, sino muestra la
    //salida de la memoria RAM.
    if (CS == 1'b1)
        //Para asignar el valor de un reg a una salida se usa la palabra reservada assign.
        dataBus = dataOutROM;
    else
        dataBus = dataOutRAM;
end
endmodule

```

Código VHDL - Memoria ROM/RAM:

```

--1.-Memorias RAM y ROM: A la información guardada en una memoria se le llama datos (data). Cuando escribimos
--en una memoria, la acción es de escritura (write) y cuando obtenemos información de una memoria lo que
--hacemos es leerla (read). Para acceder a la información requerimos de una dirección (address).
--A las memorias ROM (Read Only Memory) solo se les puede leer sus datos, mientras que a las memorias RAM
--(Random Access Memory) se les puede leer o escribir datos cuando sea.
--En este caso se creará una memoria de 8 registros (filas) con 8 celdas cada uno, por lo que tendrá una
--capacidad de 8x8 = 64 bits = 8 bytes, por lo que necesitara un memory adress de 3 bits para contar
--de 0 a 8.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--Librerías para poder usar el lenguaje VHDL
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--Librería declarada para poder hacer operaciones matemáticas sin considerar el signo.

entity memoriaRAM_ROM is
    Port ( clk : in STD_LOGIC;          --Reloj de 50MHz proporcionado por la NEXYS 2 en el puerto B8.
          --Las acciones de la memoria pueden ser sincronicas: Siguen el paso de la señal de reloj.
          --O pueden ser asincronicas: Se ejecutan en cualquier momento no importando la señal de reloj.
          adressBus : in STD_LOGIC_VECTOR(2 downto 0); --Dirección de memoria.
          RW : in STD_LOGIC;          --Switch RW (Read/Write) para escribir o leer la memoria RAM.
          --Con RW = 1 se puede leer el contenido de ambas memorias, con RW = 0 se escribe en la RAM.
          dataInRAM : in STD_LOGIC_VECTOR(7 downto 0); --Bus de datos ingresados a la memoria RAM.
          CS : in STD_LOGIC;          --CS (Chip Selector) para saber si la salida es de la RAM o ROM.
          --Con CS = 1 se elige leer el contenido de la ROM, con CS = 0 se lee el contenido de la RAM.
          dataBus : out STD_LOGIC_VECTOR(7 downto 0)); --Bus de datos extraídos de la memoria RAM.
end memoriaRAM_ROM;

architecture Behavioral of memoriaRAM_ROM is
    --LOS ARRAYS EN VHDL SE DECLARAN DENTRO DE LA ARQUITECTURA, PERO ANTES DE SU BEGIN Y SE CONFORMAN DE:
    --TYPE: Con esta instrucción se define un tipo de dato que tiene un nombre en específico y define un
    --número de elementos que tienen un tamaño en específico a través de la siguiente nomenclatura:
    --type nombreTipoDeDato is array (numeroDeElementos) of tamañoDeSusElementos;
    type arreglo is array (1 to 8) of STD_LOGIC_VECTOR(7 downto 0); --Array tipo matriz de 8x8.
    --CONSTANT: Instrucción usada para crear constantes, tiene un nombre propio y además si es un array, debe
    --tener el tipo de dato previamente declarado con la instrucción type, se le asignan valores con el símbolo :=
    --En las memorias ROM, como no se les debe cambiar el contenido de sus datos, se declara como constant.
    constant ROM : arreglo := ( "00000000", --memoryAddress = 000 = 1
                                "00000001", --memoryAddress = 001 = 2
                                "00000010", --memoryAddress = 010 = 3
                                "00000011", --memoryAddress = 011 = 4
                                "00000100", --memoryAddress = 100 = 5
                                "00000101", --memoryAddress = 101 = 6
                                "00000110", --memoryAddress = 110 = 7
                                "00000111"); --memoryAddress = 111 = 8

    --SIGNAL: No es ni una entrada ni una salida porque no puede estar vinculada a ningún puerto de la NEXYS 2,
    --solo existe durante la ejecución del código y sirve para poder almacenar algún valor de forma temporal,
    --se debe declarar dentro de la arquitectura y antes de su begin, cuando se usa para crear un array en VHDL
    --no se les asigna un tipo de forma directa, sino que, se declara un nombre y se le asigna el tipo de dato
    --previamente declarado con la instrucción type. En Verilog la declaración de arrays si se puede hacer directa
    --pero en VHDL se debe hacer en partes, primero declarando el tipo de dato y luego asignándolo.
    --En las memorias RAM, como se les puede escribir o leer datos, se declara como signal.
    signal RAM: arreglo := ( "00000000", --memoryAddress = 000 = 1
                              "10000001", --memoryAddress = 001 = 2
                              "01000010", --memoryAddress = 010 = 3
                              "11000011", --memoryAddress = 011 = 4
                              "00100100", --memoryAddress = 100 = 5
                              "10100101", --memoryAddress = 101 = 6
                              "01100110", --memoryAddress = 110 = 7
                              "11100111"); --memoryAddress = 111 = 8

    --signal que representa un array que podrá adoptar varias matrices de valores distintos.
    signal dataOutROM : STD_LOGIC_VECTOR(7 downto 0); --Bus de datos extraídos de la memoria ROM.
    signal dataOutRAM : STD_LOGIC_VECTOR(7 downto 0); --Bus de datos extraídos de la memoria RAM.

begin
    --La obtención de datos de la memoria ROM es sincrona porque depende de la señal de reloj, no es así en
    --todas las memorias RAM o ROM, puede ser como sea, pero aquí se hace así para mostrar sus diferencias.
    process(clk) begin
        --La instrucción rising_edge() hace que este condicional solo se ejecute cuando ocurra un flanco de
        --subida en la señal de reloj clk proveniente de la NEXYS 2.
        if(rising_edge(clk)) then

```

```

--conv_integer(): Método que convierte un numero binario en uno entero. Además, cabe mencionar que
--a esta conversión se le suma un 1 porque para acceder a las posiciones de un array se cuenta
--desde 1, no desde 0.
dataOutROM <= ROM(conv_integer(addressBus)+1);

end if;
end process;

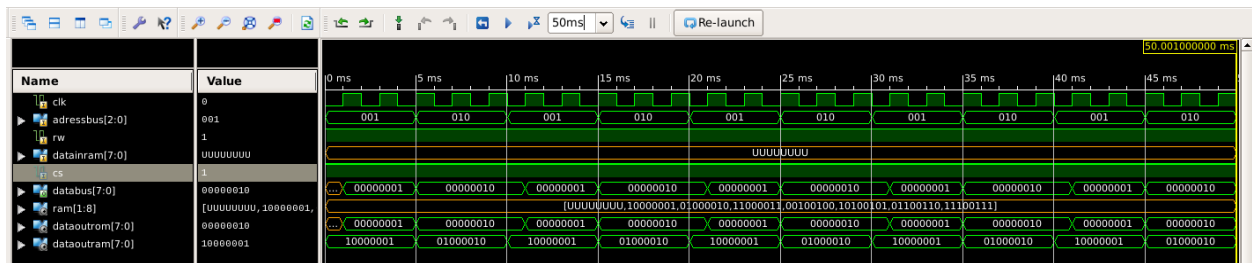
--La lectura y escritura de datos en la memoria RAM es asincrónica porque NO depende de la señal de reloj.
process(RW, dataInRAM) begin
--Si el switch RW está en 1 lógico, la memoria es de lectura, sino es de escritura.
if (RW = '1') then
dataOutRAM <= RAM(conv_integer(addressBus)+1); --Lectura de la memoria RAM.
else
RAM(conv_integer(addressBus)+1) <= dataInRAM; --Escritura de la memoria RAM.
dataOutRAM <= RAM(conv_integer(addressBus)+1); --Lectura de la memoria RAM.
end if;
end process;

process(CS, dataOutROM, dataOutRAM) begin
--Si el switch RW está en 1 lógico, el data bus muestra la salida de la memoria ROM, sino muestra la
--salida de la memoria RAM.
if (CS = '1') then
dataBus <= dataOutROM;
else
dataBus <= dataOutRAM;
end if;
end process;
end Behavioral;

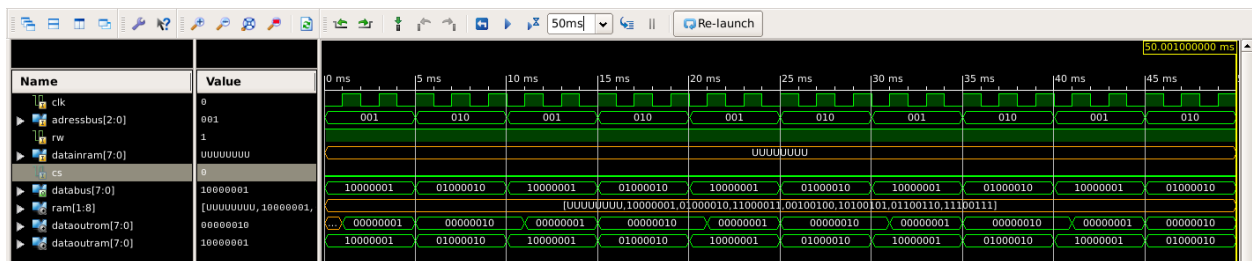
```

Simulación: Memorias RAM/ROM

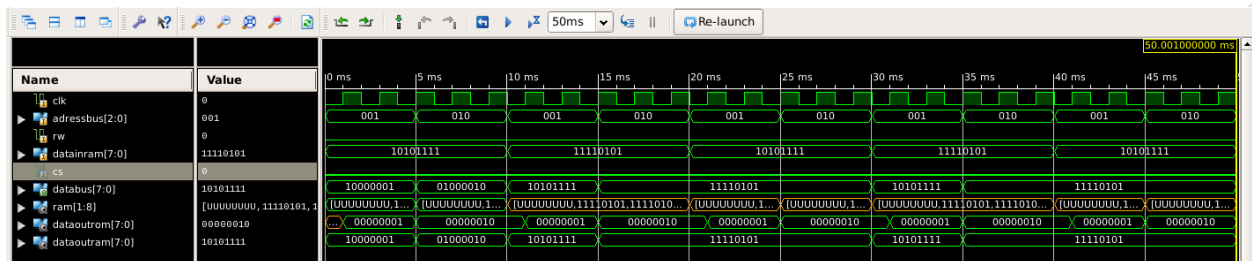
Memoria ROM (Read); RW = 1 y CS = 1




Memoria RAM (Read); RW = 1 y CS = 0



Memoria RAM (Write); dataInRAM = 10101111 y 11110101; RW = 0 y CS = 0



Referencias:

Les Ingenieurs, “ FLIP FLOP RS, D, JK y T | TODOS los FLIP FLOPS en 6 MINUTOS | ELECTRÓNICA DIGITAL”, 2016 [Online], Available: <https://www.youtube.com/watch?v=3xy9WDgfgJg>

Electrónica FP, Fernando Manso, “Registros.”, 2020 [Online], Available: <https://www.youtube.com/watch?v=Kaq59Z9Z7Us&list=PLuzS0jdNRVvpQmCxFV4S2eqfji90BnDub&index=78>

Electrónica FP, Fernando Manso, “Introducción a las memorias. Parte 1.”, 2020 [Online], Available: <https://www.youtube.com/watch?v=sARr8wGOhg&list=PLuzS0jdNRVvpQmCxFV4S2eqfji90BnDub&index=73>

Electrónica FP, Fernando Manso, “Introducción a las memorias. Parte 2.”, 2020 [Online], Available: <https://www.youtube.com/watch?v=uJHV-J4Au-l&list=PLuzS0jdNRVvpQmCxFV4S2eqfji90BnDub&index=74>

Electrónica FP, Fernando Manso, “Introducción a las memorias. Parte 3.”, 2020 [Online], Available: <https://www.youtube.com/watch?v=sARr8wGOhg&list=PLuzS0jdNRVvpQmCxFV4S2eqfji90BnDub&index=73>

