



Laboratorio 3:

# Máquinas de estados finitos

acondicionamiento de las señales de reloj y reset

Diseño automático de sistemas

**José Manuel Mendías Cuadros**

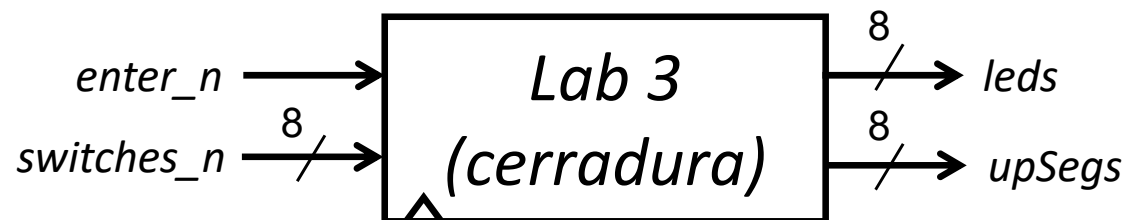
*Dpto. Arquitectura de Computadores y Automática  
Universidad Complutense de Madrid*



# Presentación



- Diseñar una **cerradura electrónica** con el siguiente comportamiento:
  - Inicialmente la cerradura estará **abierta**.
  - A cada **pulsación** de **enter\_n**, el sistema **leerá una clave de 8 bits**.
    - La **primera pulsación** almacenará la clave y cerrará la cerradura.
    - Las **sucesivas pulsaciones** abrirán la cerradura siempre y cuando la clave introducida coincida con la clave almacenada.
  - Se tendrán un **máximo de 3 intentos** para acertar la clave almacenada.
    - Tras fallar los 3 intentos la cerradura quedará indefinidamente cerrada.
- Tomará las señales y visualizará la cuenta del siguiente modo:
  - La **clave** la leerá de los **switches** de la placa XST a cada pulsación del pulsador PB-1.
  - El **estado** del cerrojo será visible en **todos los leds** de la placa XST.
  - El **número de intentos restantes** se visualizará en el **display 7-seg** de la placa XSA-3S.
    - Así como una "A" cuando esté abierto y una "C" cuando esté cerrado
  - Podrá **resetearse asíncronamente** pulsando el pulsador PB1 de la placa XSA-3S.

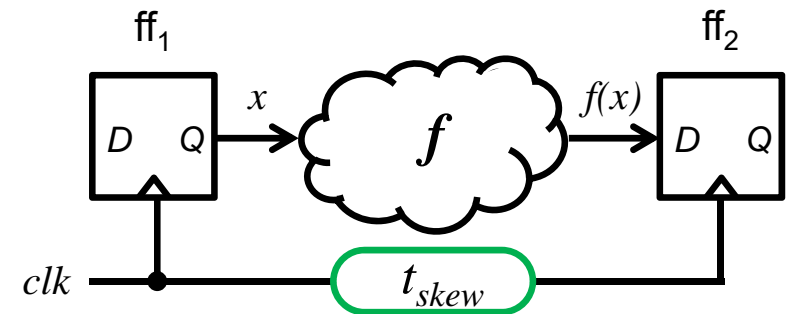
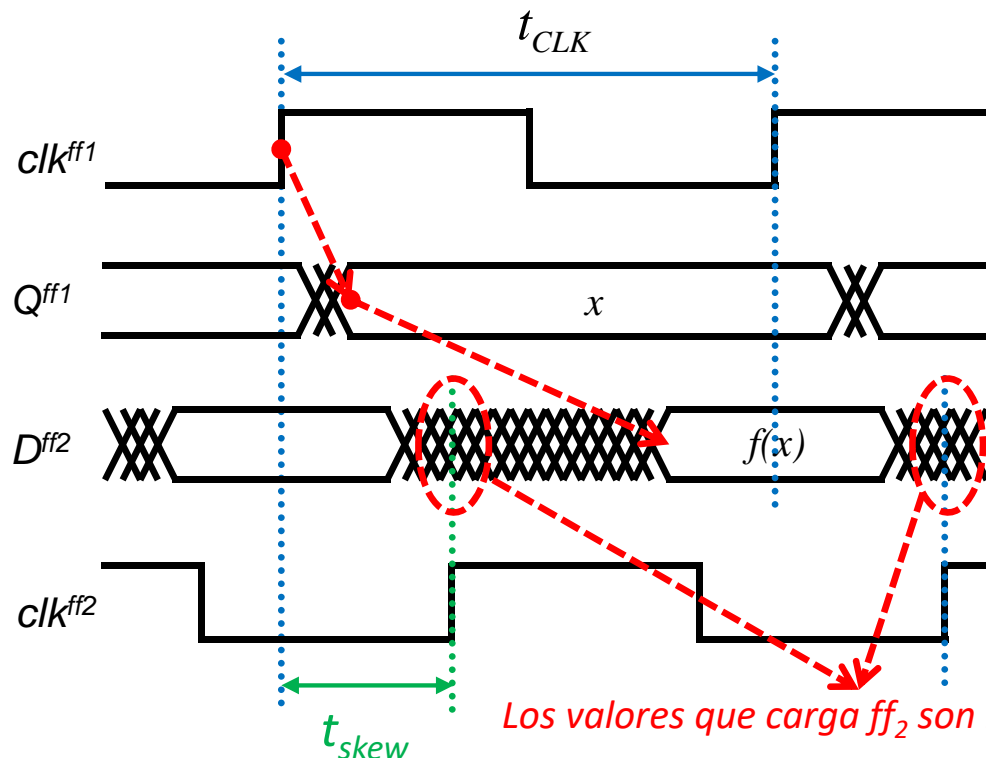


# Señal de reloj

## problema (i)



- El modelo de **temporización síncrona**, asume que los eventos del **reloj** llegan simultáneamente a todos los biestables del sistema.
  - Si la señal de reloj llega con cierto retraso (**skew**) a algunos flip-flops, el sistema se desincroniza o entra en metaestabilidad.
  - Ídem si la frecuencia del reloj no es perfectamente regular (**jitter**).



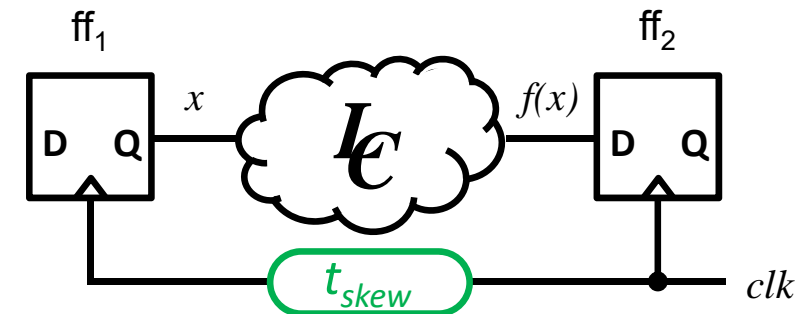
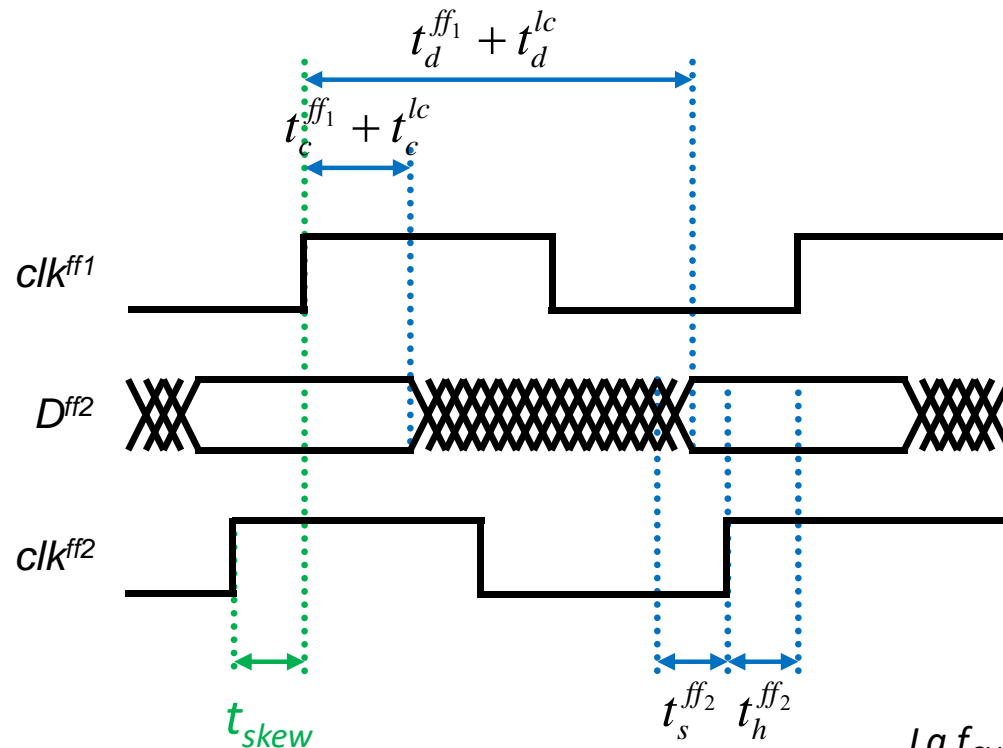
- Distinta longitud de cable
- Ruido (interferencias)
- Diferente carga local
- Variaciones locales de temperatura
- etc...

# Señal de reloj

## problema (ii)



- Las **herramientas EDA** pueden solventar este problema:
  - Reformulando las **ligaduras de retardo máximo y mínimo** que satisfacen la lógica combinacional sintetizada de manera que tenga en cuenta el skew y jitter.
- En el caso de **skew negativo** (datos y reloj en sentido contrario)
  - A costa de degradar el rendimiento del circuito.



**ligadura de retardo máximo:**

$$t_{CLK} \geq (t_d^{ff1} + t_d^{lc} + t_s^{ff2} + t_{skew})$$

**ligadura de retardo mínimo:**

$$(t_c^{ff1} + t_c^{lc}) \geq t_h^{ff2} - t_{skew}$$

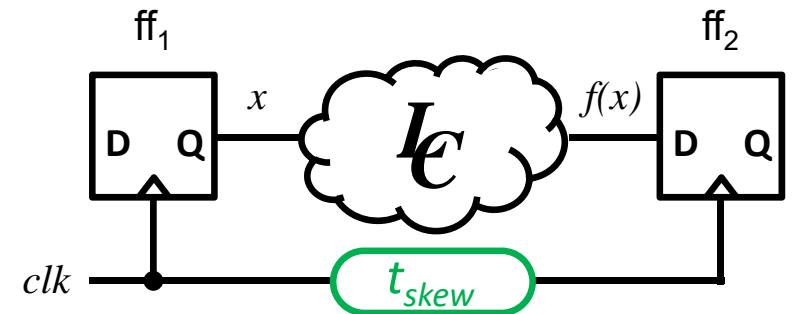
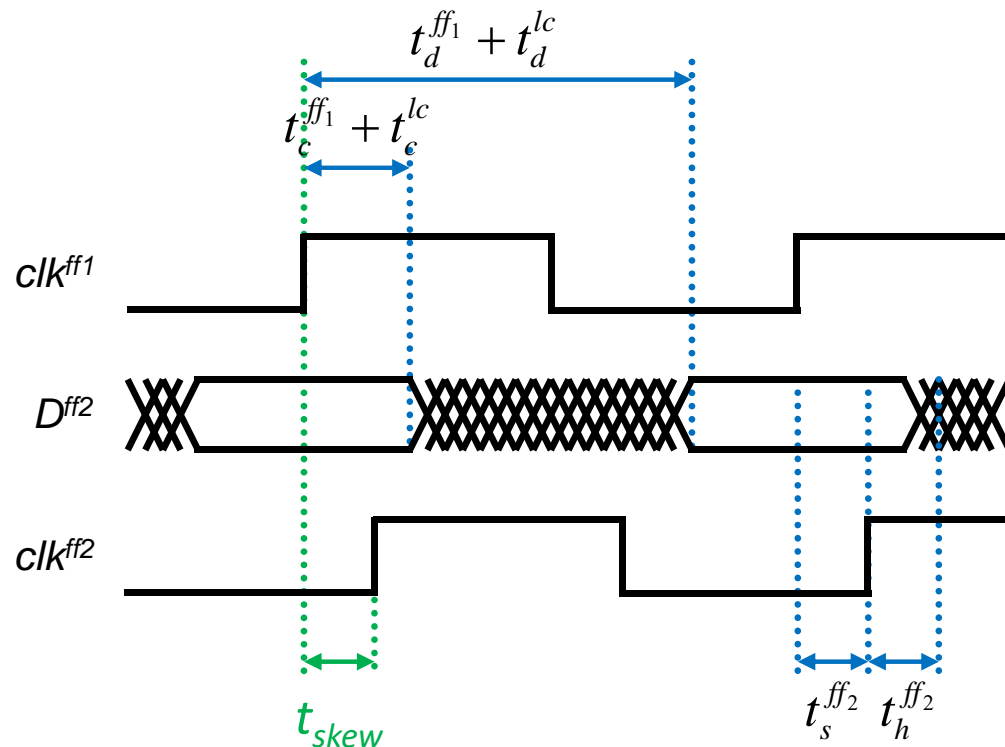
La  $f_{CLK}$  máxima alcanzable es menor que sin skew

# Señal de reloj

## problema (iii)



- En el caso de **skew positivo** (datos y reloj el mismo sentido)
  - A costa de añadir lógica extra (que aumenta el retardo) para evitar la llegada prematura de datos



**ligadura de retardo máximo:**

$$t_{CLK} \geq (t_d^{ff1} + t_d^{lc} + t_s^{ff2} - t_{skew})$$

**ligadura de retardo mínimo:**

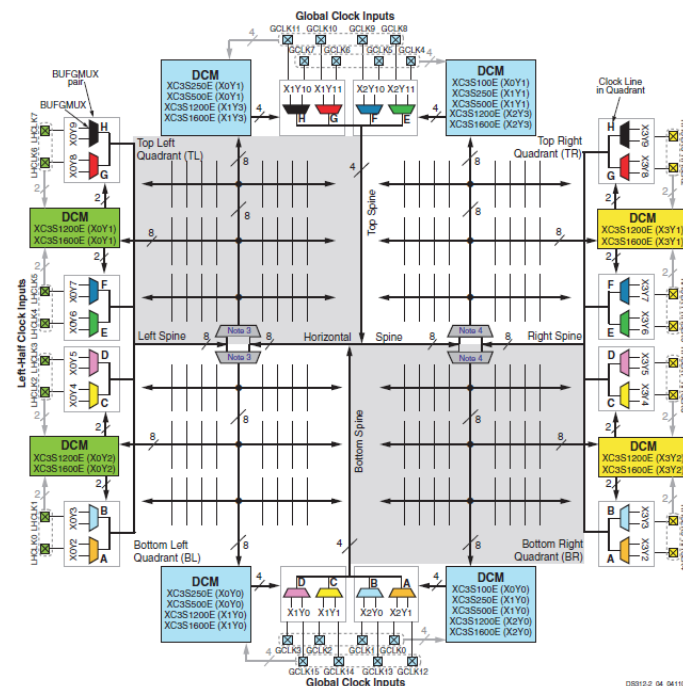
$$(t_d^{ff1} + t_c^{lc}) \geq t_h^{ff2} + t_{skew}$$

# Señal de reloj

## problema (iv)



- Por su lado las **FPGA**, para solventar el mismo problema, disponen de una **red global dedicada de distribución de reloj** de bajo skew y alto fanout.
  - Esta red solo puede conectarse a ciertos pines de entrada
    - Por ello la señal de reloj solo puede entrar por alguno de ellos
    - La conexión del pin a la red se realiza implícitamente cuando la herramienta identifica en el código fuente una señal de tipo reloj.
  - La red, en la **familia Spartan-3**, tiene **topología 'fish-bone'**

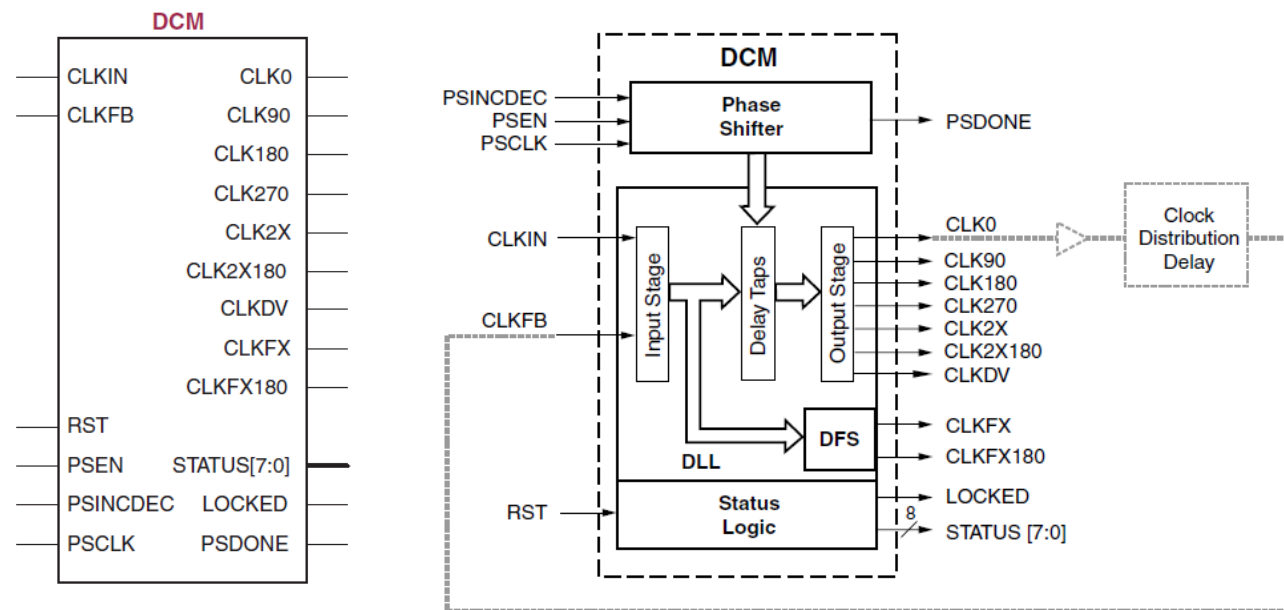


# Señal de reloj

## digital clock manager (i)



- Además, disponen de **módulos prefabricados para la gestión de reloj**
  - Que pueden conectarse entre el pin de entrada del reloj y la red de distribución.
  - La **familia Spartan-3** disponen de módulos **DCM (Digital Clock Managers)**
  - Dada una señal de reloj de entrada, un DCM puede:
    - **Multiplicar y/o dividir** su frecuencia.
    - Acondicionarla para que tenga un **factor de trabajo del 50%** y unas transiciones limpias.
    - **Desfasarla** una fracción de su periodo.
    - **Eliminar el skew** debido a retardos de su red de distribución interna/externa.
    - Transformarla una señal de reloj entre **diferentes estándares E/S**.



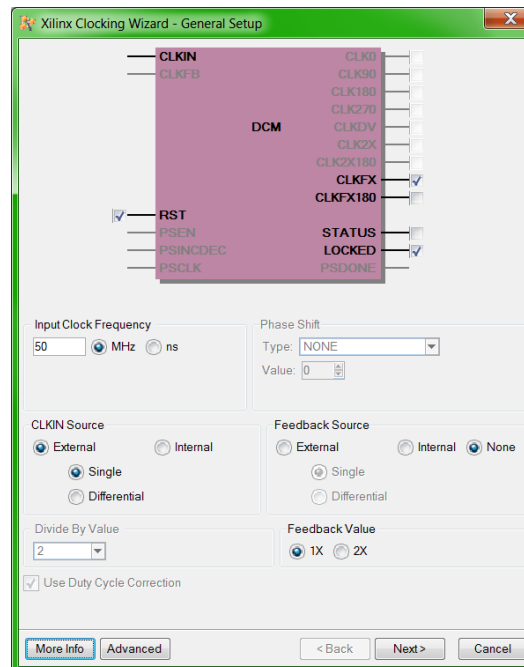


# Señal de reloj

## digital clock manager (ii)



- Para usar un DCM debe **instanciarse como componente**
  - Su comportamiento se configura a través de los valores que toman los genéricos durante instanciación.
- Para facilitar su instanciación y configuración puede usarse un wizard del Core Generator
  - **Project > New Source**
  - **IP (CORE Generator & Architecture Wizard) > Single DCM**







# Señal de reloj

## sintetizador de frecuencias

- Uno de los usos del DCM es como sintetizador de frecuencias
  - La frecuencia del reloj de entrada debe estar entre 1 MHz y 280 MHz
  - La frecuencia del reloj de salida podrá estar, según el modo del DCM entre:
    - 18 MHz y 210 MHz (en modo "LOW")
    - 210 MHz y 326 MHz (en modo "HIGH")
  - La frecuencia de salida se obtiene a partir de la frecuencia de entrada
    - Multiplicándola por un factor  $M=\{2..32\}$
    - Dividiéndola por un divisor  $D=\{1..32\}$

$$f_{clkOut} = \frac{M \cdot f_{clkIn}}{D}$$

- Por ejemplo, tomando un reloj de entrada de frecuencia 50 MHz, es posible obtener un amplio rango de frecuencias para el reloj de salida:

M	D	$f_{clkOut}$
2	1	100 MHz
3	2	75 Mhz
3	5	30 MHz
14	25	28 MHz

M	D	$f_{clkOut}$
4	5	40 MHz
21	25	42 Mhz
22	25	44 MHz
23	25	46 MHz

M	D	$f_{clkOut}$
28	5	280 MHz
21	25	210 Mhz
9	25	18 MHz
2	5	20 MHz



# Señal de reloj

## frequencySynthesizer.vhd



```
library ieee;
use ieee.std_logic_1164.all;

entity frequencySynthesizer is
  generic (
    FREQ      : natural;           -- frecuencia del reloj de entrada en KHz
    MODE      : string;           -- modo del sintetizador de frecuencia "LOW" o "HIGH"
    MULTIPLY  : natural range 2 to 32; -- factor por el que multiplicar la frecuencia de entrada
    DIVIDE    : natural range 1 to 32 -- divisor por el que dividir la frecuencia de entrada
  );
  port (
    clkIn  : in  std_logic; -- reloj de entrada
    ready  : out std_logic; -- indica si el reloj de salida es válido
    clkOut : out std_logic  -- reloj de salida
  );
end frequencySynthesizer;

library unisim;
use unisim.vcomponents.all;

architecture syn of frequencySynthesizer is
begin
  ...
end syn;
```

# Señal de reloj

## frequencySynthesizer.vhd



```
clockManager : DCM
  generic map
  (
    CLKDV_DIVIDE          => 2.0,
    CLKFX_DIVIDE          => DIVIDE,
    CLKFX_MULTIPLY        => MULTIPLY,
    CLKIN_DIVIDE_BY_2     => FALSE,
    CLKIN_PERIOD          => 1_000_000.0/real(FREQ),  -- periodo de la entrada (en ns)
    CLKOUT_PHASE_SHIFT   => "NONE",
    CLK_FEEDBACK          => "NONE",
    DESKEW_ADJUST         => "SYSTEM_SYNCHRONOUS",
    DFS_FREQUENCY_MODE    => MODE,
    DUTY_CYCLE_CORRECTION => FALSE,
    PHASE_SHIFT           => 0,
    STARTUP_WAIT          => FALSE
  )

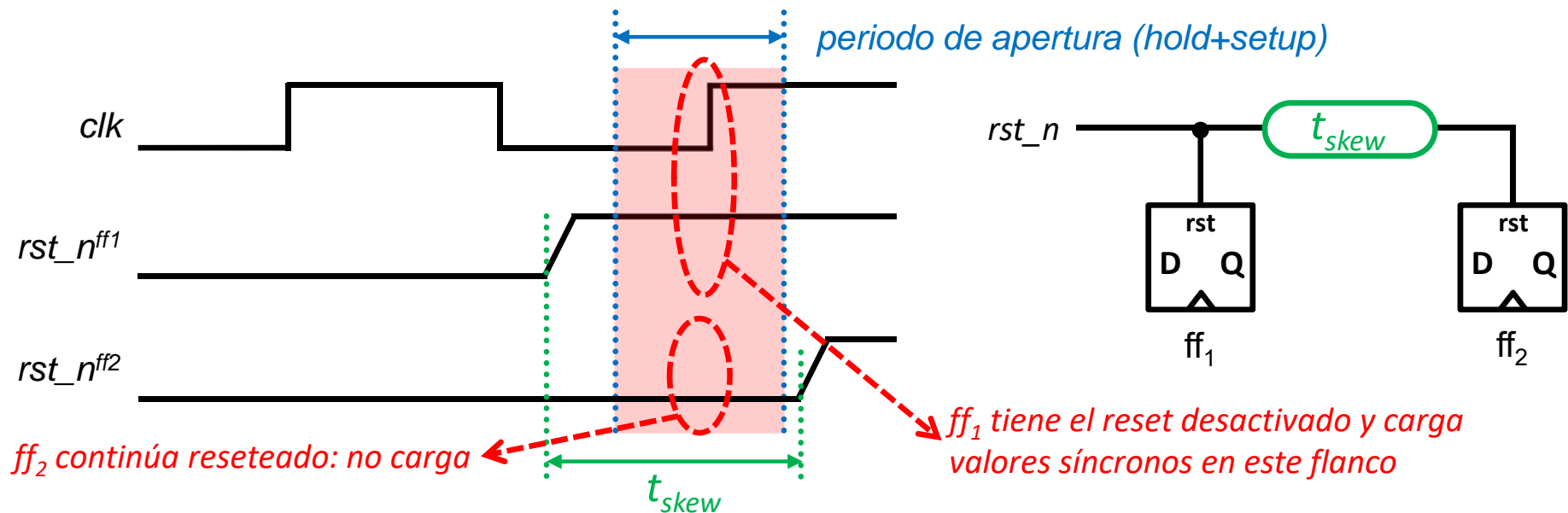
  port map
  (
    CLK0      => open,
    CLK90     => open,
    CLK180    => open,
    CLK270    => open,
    CLK2X     => open,
    CLK2X180  => open,
    CLKDV     => open,
    CLKFX     => clkOut,
    CLKFX180  => open,
    LOCKED    => ready,
    PSDONE    => open,
    STATUS    => open,
    CLKFB     => '0',
    CLKIN     => clkIn,
    PSCLK     => '0',
    PSEN      => '0',
    PSINCDEC  => '0',
    RST       => '0'
  );
```

# Señal asíncrona de reset

## problema



- El **reset** es una señal de **alta conectividad y naturaleza asíncrona**
  - El **retardo de su red de distribución** hace que sus **eventos no lleguen al mismo tiempo** a todos los biestables.
  - Si el **reset se activa** en cada biestable en distintos instantes no hay problema.
  - Si el **reset se desactiva en las proximidades del flanco de reloj**, puede que:
    - Parte de los **biestables comiencen su funcionamiento normal al final del ciclo** y otros **no lo hagan hasta el ciclo siguiente**.
    - Parte de los **biestables entren en metaestabilidad** porque comiencen a cargar los nuevos valores síncronos durante su periodo de apertura.

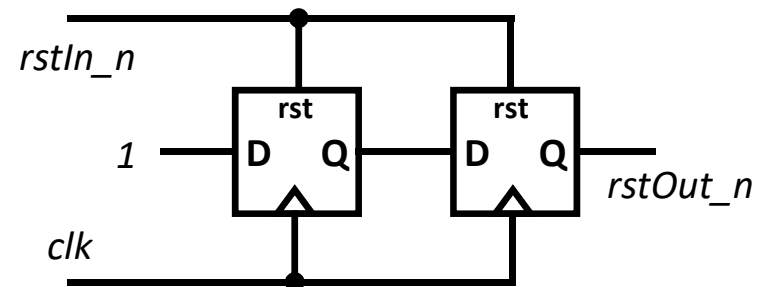
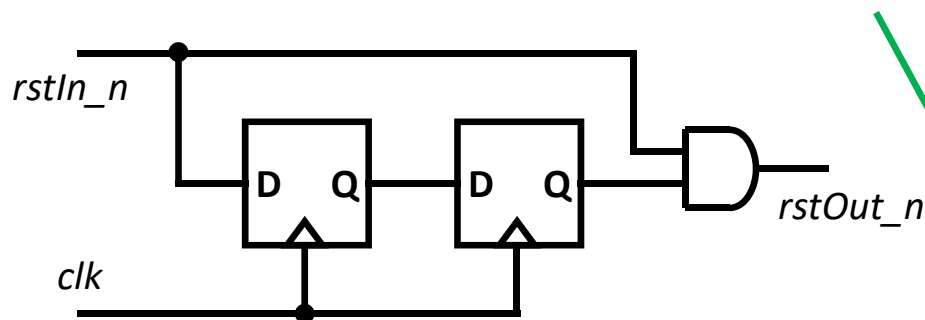




# Señal asíncrona de reset

## sincronizador de reset de 2 etapas (i)

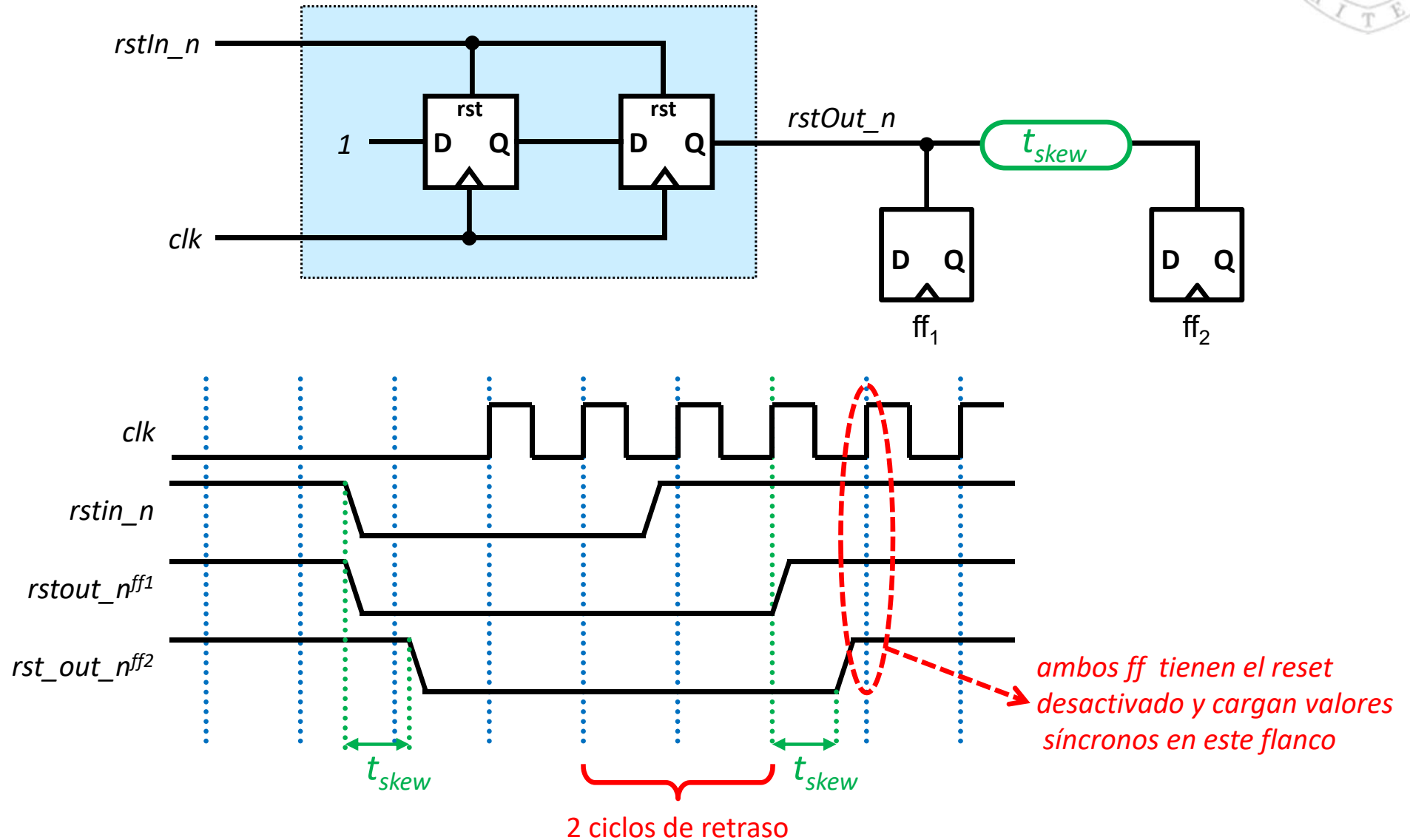
- Una posible solución sería **añadir un sincronizador al reset**
  - De este modo, **el reset se activaría y desactivaría síncronamente**.
    - La desactivación se produce poco después del flanco de reloj, y **tiene un ciclo completo para alcanzar a todos los biestables**.
  - Sin embargo para que la activación del reset se propague a los biestables:
    - **Es necesario que el reloj funcione** (para que la activación se propague por el sincronizador)
    - **Lo hace algunos ciclos después de que el reloj comience a funcionar**, ciclos durante el cual el sistema ha tenido un comportamiento síncrono.
- La solución mejor es **adaptar un sincronizador para que tenga una activación asíncrona y desactivación síncrona**.
  - Mientras que la señal de reloj no funcione, el sistema estará reseteando
  - Una vez funcione, seguirá reseteando hasta que la desactivación alcance la salida del sincronizador





# Señal asíncrona de reset

## sincronizador de reset de 2 etapas (ii)





# Señal asíncrona de reset

## synchronizer.vhd

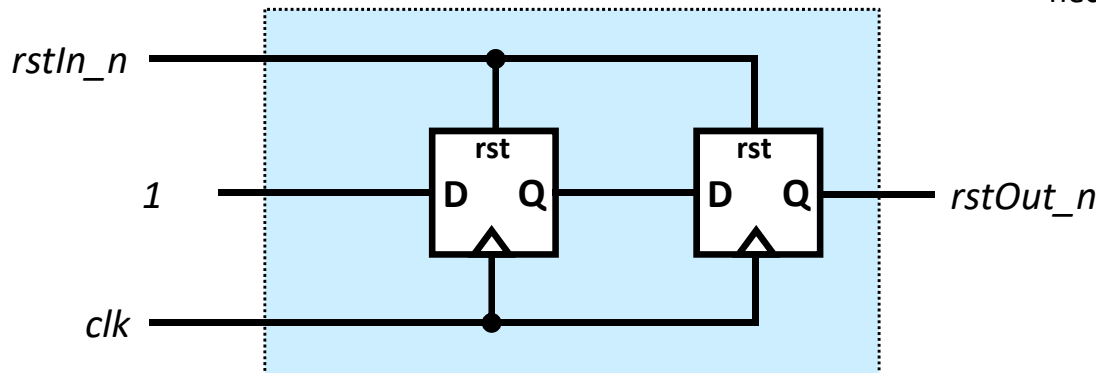
- El sincronizador diseñado para sincronizar señales externas puede reutilizarse también para sincronizar el reloj.

- o Instanciación para sincronizar un pulsador en reposo los pulsadores valen '1'

```
pbSynchronizer : synchronizer
generic map ( STAGES => 2, INIT => '1' )
port map ( rst_n => rst_n, clk => clk, x => pb_n, xSync_n => pb_n );
```

- o Instanciación para sincronizar el reset (aunque venga de un pulsador)

```
rstSynchronizer : synchronizer
generic map ( STAGES => 2, INIT => '0' )
port map ( rst_n => rstIn_n, clk => clk, x => '1', xSync_n => rstOut_n );
```



necesario para la activación asíncrona del reset

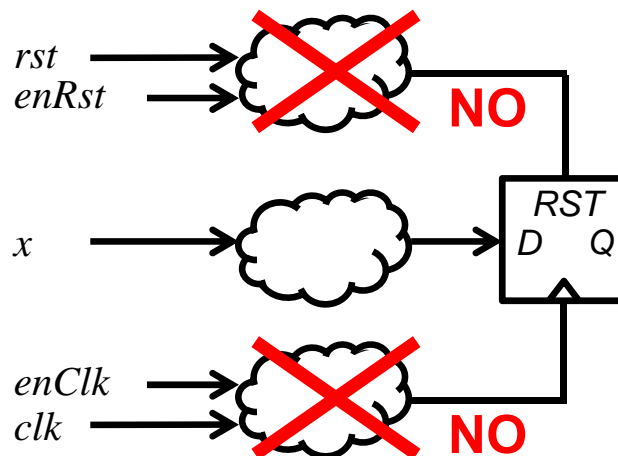




# Reloj y reset asíncrono

## evitar malas prácticas

- Es **mala práctica que el reloj atraviese lógica** (clock gating):
  - Introduce un **skew variable el reloj** debido a la incertidumbre de la red de puertas.
  - Puede producir glitches que provoquen **cambios espurios de estado**.
- Es **mala práctica que el reset asíncrono atraviese lógica** (reset gating):
  - Introduce un **skew variable en el reset** debido a la incertidumbre de la red de puertas
  - Puede producir glitches que provoquen **inicializaciones espurias**.
- Cuando se usan estas técnicas requieren un diseño manual de la red.



```
process( rst, clk )
begin
    if rst='1' and enRst='1' then
        q <= ...;
    if rising_edge(clk) and enClk='1' then
        q <= ...;
    end if;
end process;
```

# Reset o no reset

## esa es la cuestión



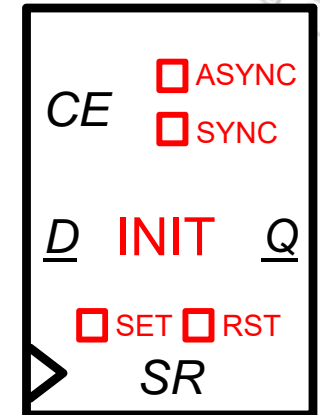
- Un **diseño ortodoxo VLSI** dispone de **una única señal global de reset asíncrono acondicionada** que alcanza **todos los biestables** del circuito
  - Dicho reset se debe activar **tras power-on** y **bajo demanda** para llevar al sistema completo a un estado inicial conocido.
- Sin embargo, esta **técnica es costosa**:
  - En **ASICs**, los biestables con reset son mayores que sin reset y el rutado de la señal global ocupa espacio.
  - En **FPGAs**, todos los biestables tienen reset, pero uso puede requerir más recursos que cuando no se usa.
- Así, para reducir costes, **cuando el valor de un biestable tras power-on no afecta el funcionamiento** del sistema, **se especifica sin reset**:
  - Si forman parte de un pipe-line (tras algunos ciclos alcanzan valores validos)
  - Si están conectados a lógica que los inicializa explícitamente (síncronamente)

# Reset asíncrono vs. síncrono

## diseños sobre FPGAs



- Los **biestables de las FPGAs** tienen ciertas peculiaridades:
  - Tienen entradas de inicialización (SR) que pueden configurarse para que **inicialicen el biestable asíncrona o síncronamente**.
  - El **valor inicial del biestable forma parte de la configuración**, luego no requieren ser reseteados explícitamente tras power-on (la FPGA implícitamente resetea tras la carga, activado la **señal GSR**).



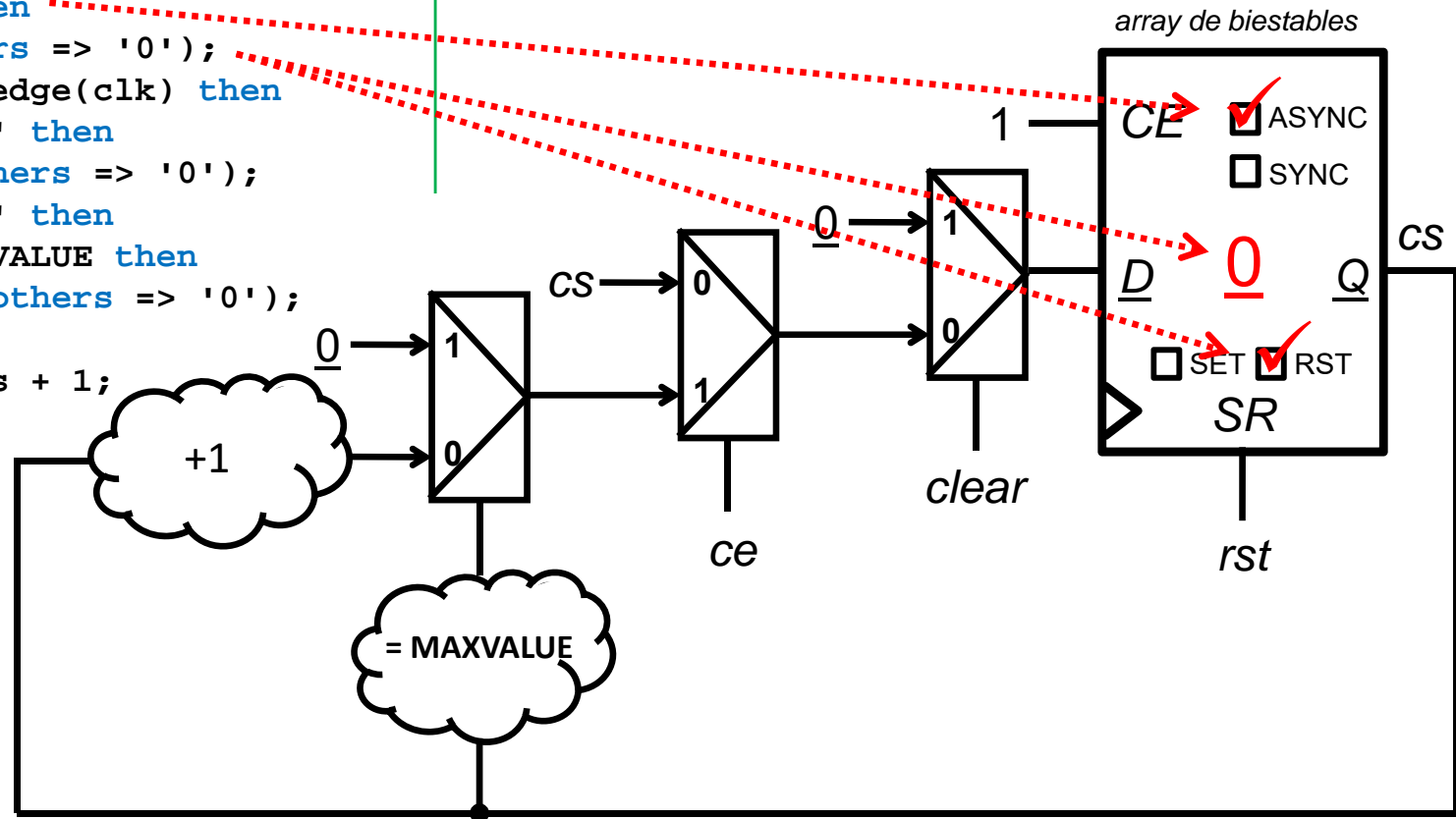
- Por ello, es común ver **diseños proyectados sobre FPGAs** con:
  - Lógica secuencial **con reset síncrono**.
  - Lógica secuencial **sin reset**.
  - En ambos casos consiguen circuitos con un **uso menor de recursos**.
- En **ASICs el reset síncrono no se usa** por ser una técnica que no resuelve el problema de la inicialización:
  - Ya que requiere la existencia de una señal de reloj válida (cosa no siempre cierta tras power-on).



# Reset asíncrono vs. síncrono

## diseños sobre FPGAs con reset asíncrono

```
architecture rstAsync of counter is
    signal cs : unsigned(n-1 downto 0);
begin
    process (rst_n, clk)
    begin
        if rst='1' then
            cs <= (others => '0');
        elsif rising_edge(clk) then
            if clear='1' then
                cs <= (others => '0');
            elsif ce='1' then
                if cs=MAXVALUE then
                    cs <= (others => '0');
                else
                    cs <= cs + 1;
                end if;
            end if;
        end if;
    end process;
end rstAsvnc;
```



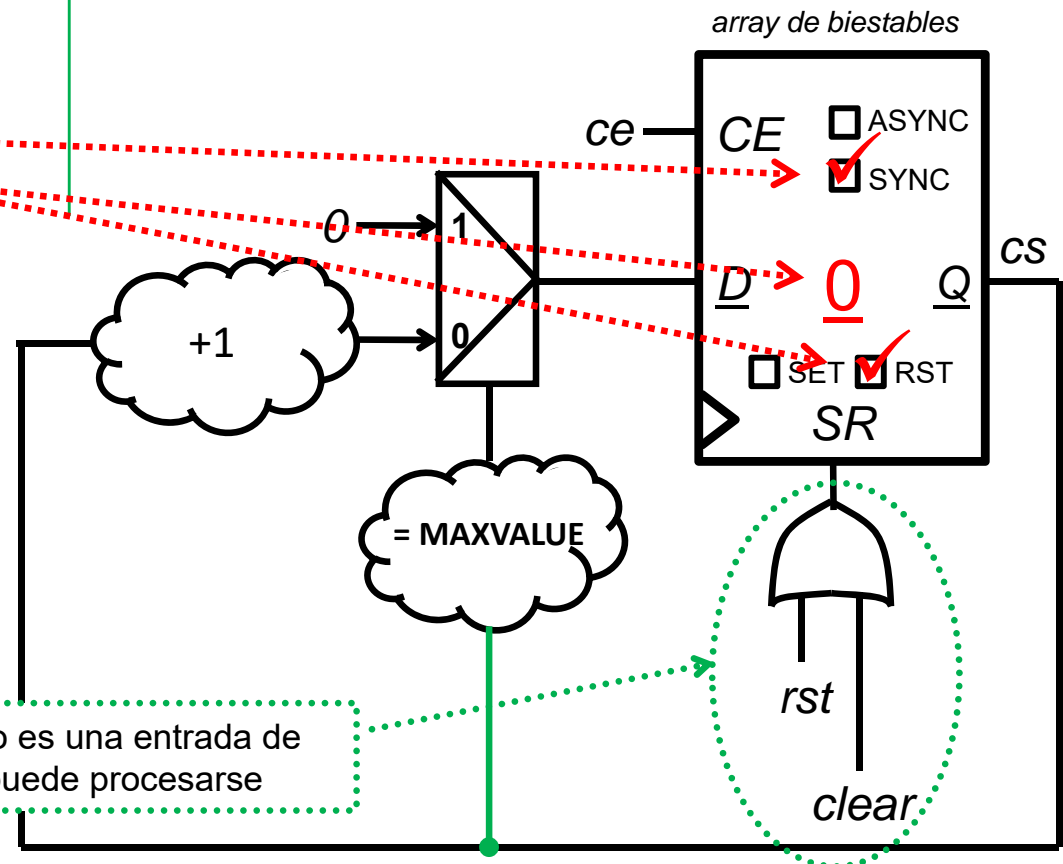


# Reset asíncrono vs. síncrono

## diseños sobre FPGAs con reset síncrono

```
architecture rstSync of counter is
    signal cs : unsigned(n-1 downto 0);
begin
    process (rst_n, clk)
    begin
        if rising_edge(clk) then
            if rst='1' then
                cs <= (others => '0');
            elsif clear='1' then
                cs <= (others => '0');
            elsif ce='1' then
                if cs=MAXVALUE then
                    cs <= (others => '0');
                else
                    cs <= cs + 1;
                end if;
            end if;
        end if;
    end process;
end rstSync;
```

El reset síncrono es una entrada de datos más y puede procesarse

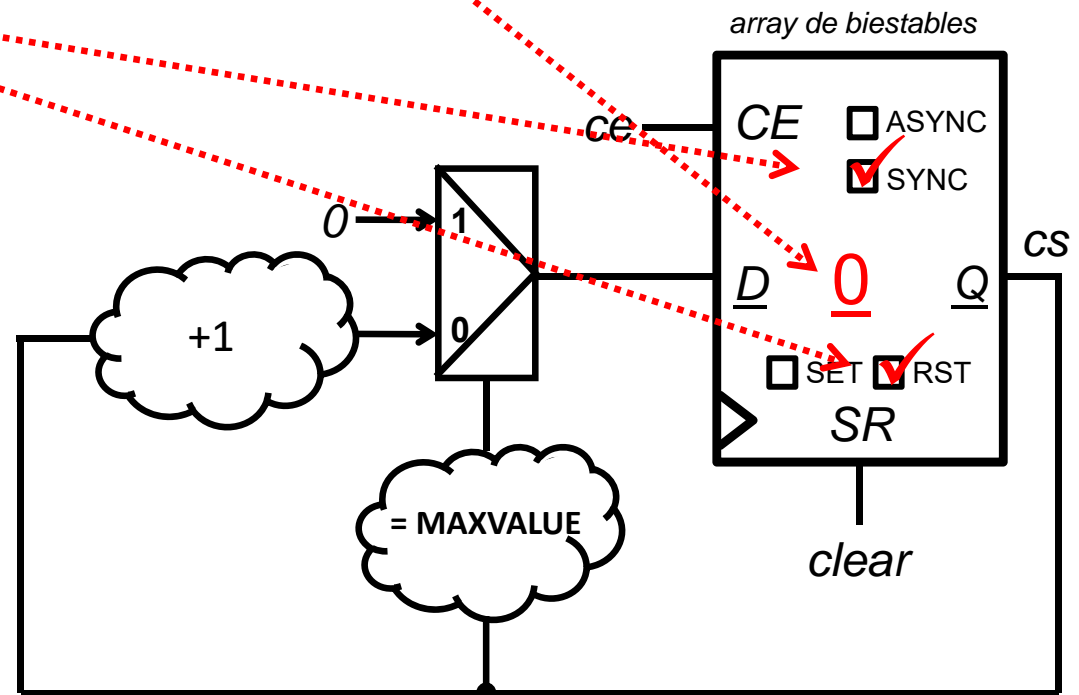




# Reset asíncrono vs. síncrono

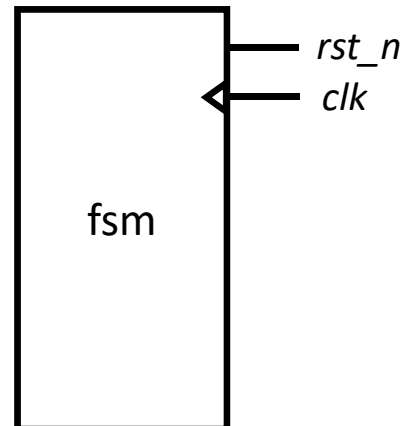
## diseños sobre FPGAs sin reset

```
architecture noRst of counter is
  signal cs : unsigned(n-1 downto 0) := (others => '0');
begin
  process (rst_n, clk)
  begin
    if rising_edge(clk) then
      if clear='1' then
        cs <= (others => '0');
      elsif ce='1' then
        if cs=MAXVALUE then
          cs <= (others => '0');
        else
          cs <= cs + 1;
        end if;
      end if;
    end if;
  end process;
end noRst;
```



# Diseño principal

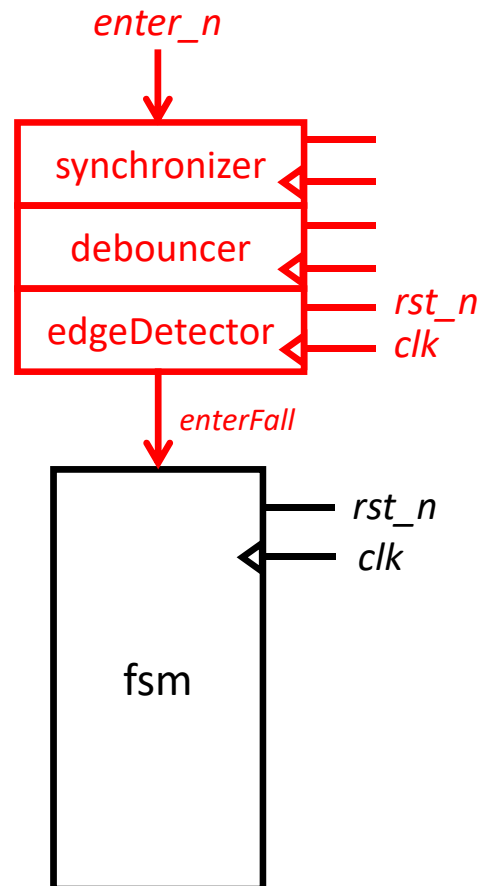
## diagrama RTL (i)





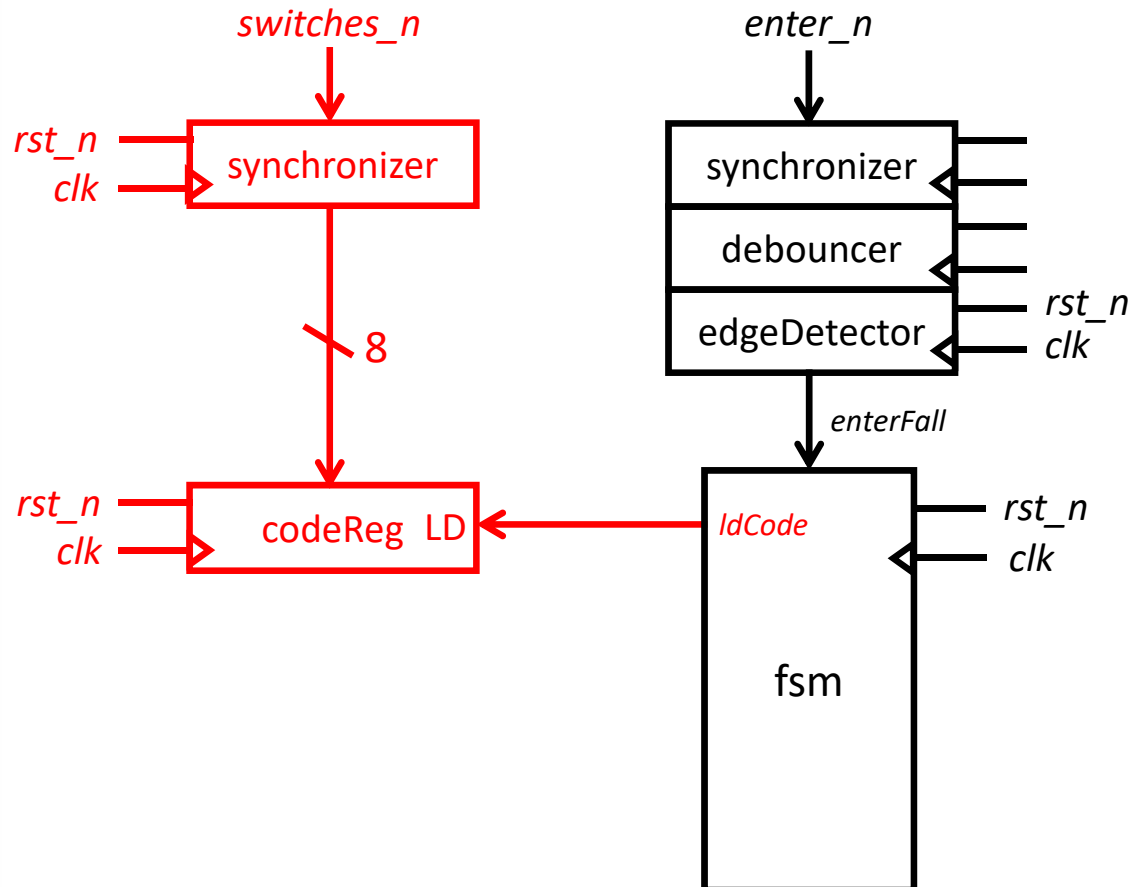
# Diseño principal

## diagrama RTL (i)



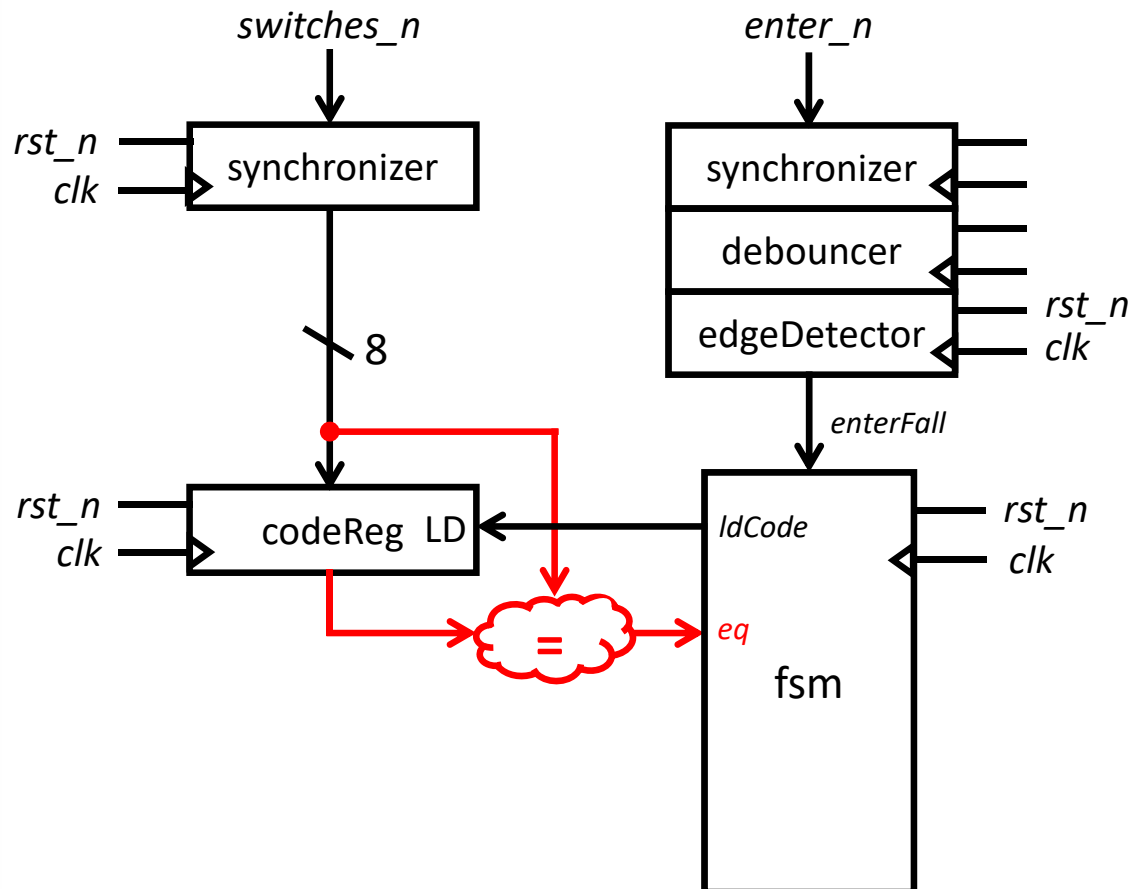
# Diseño principal

## diagrama RTL (i)



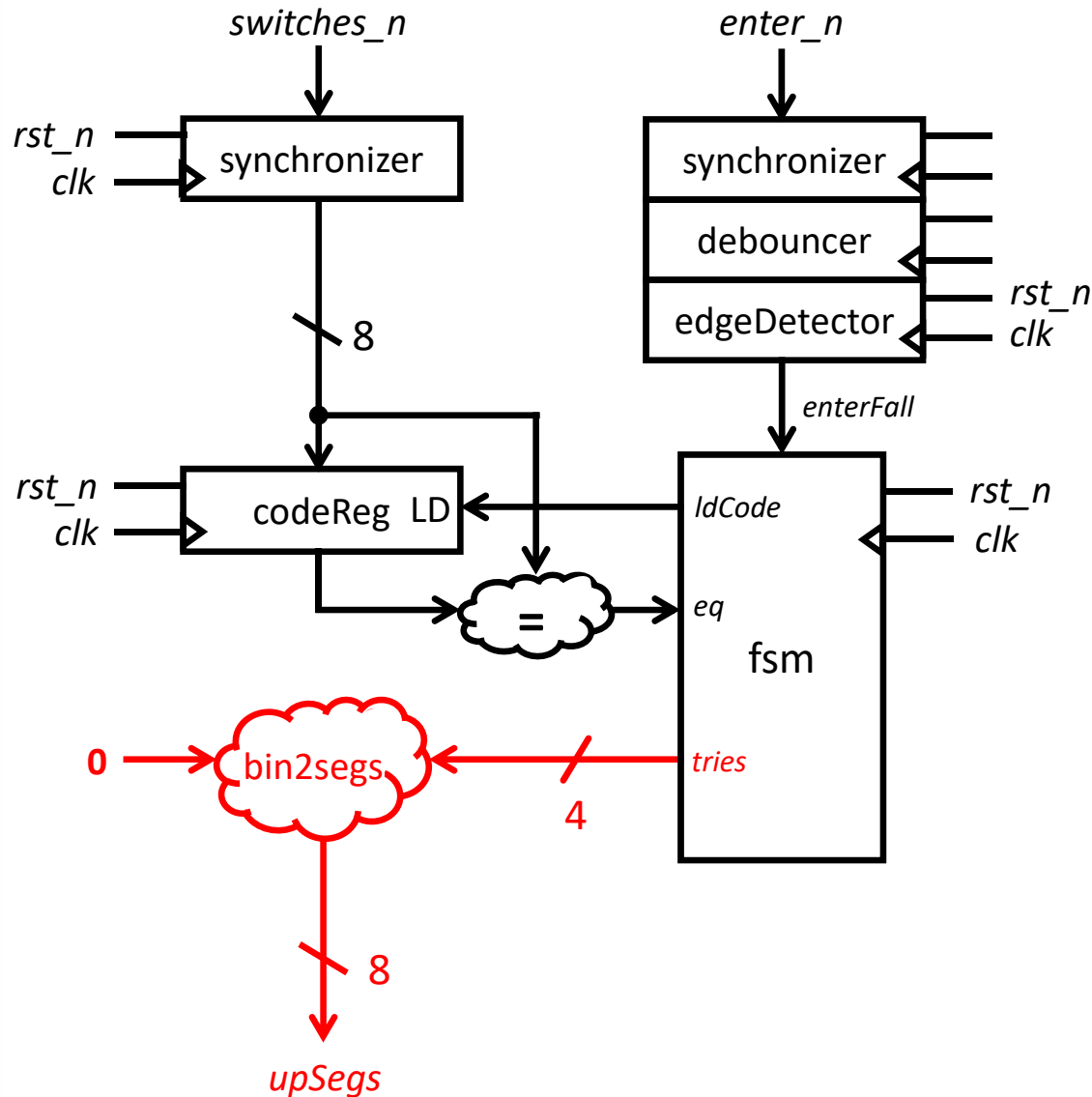
# Diseño principal

## diagrama RTL (i)



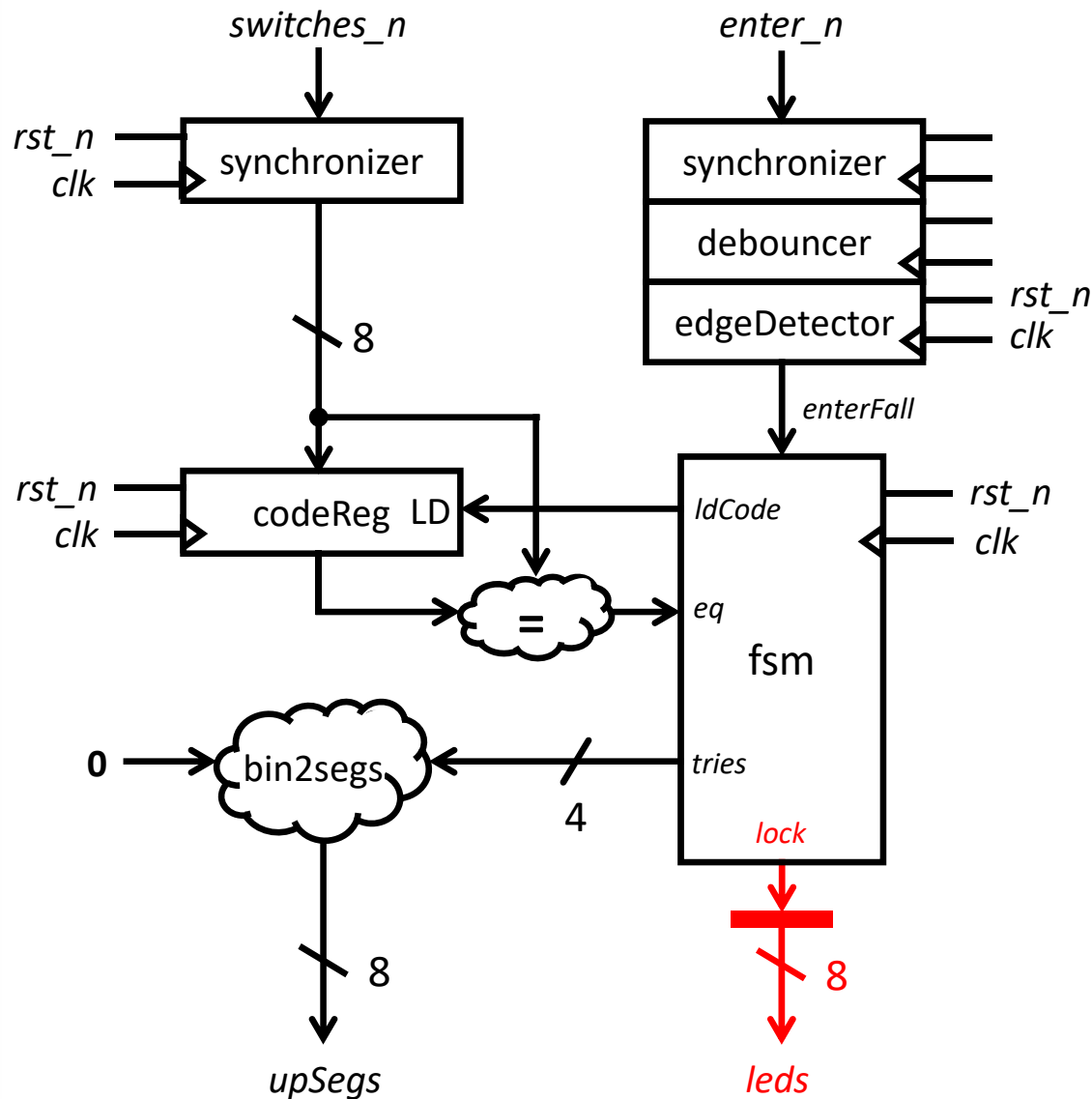
# Diseño principal

## diagrama RTL (i)



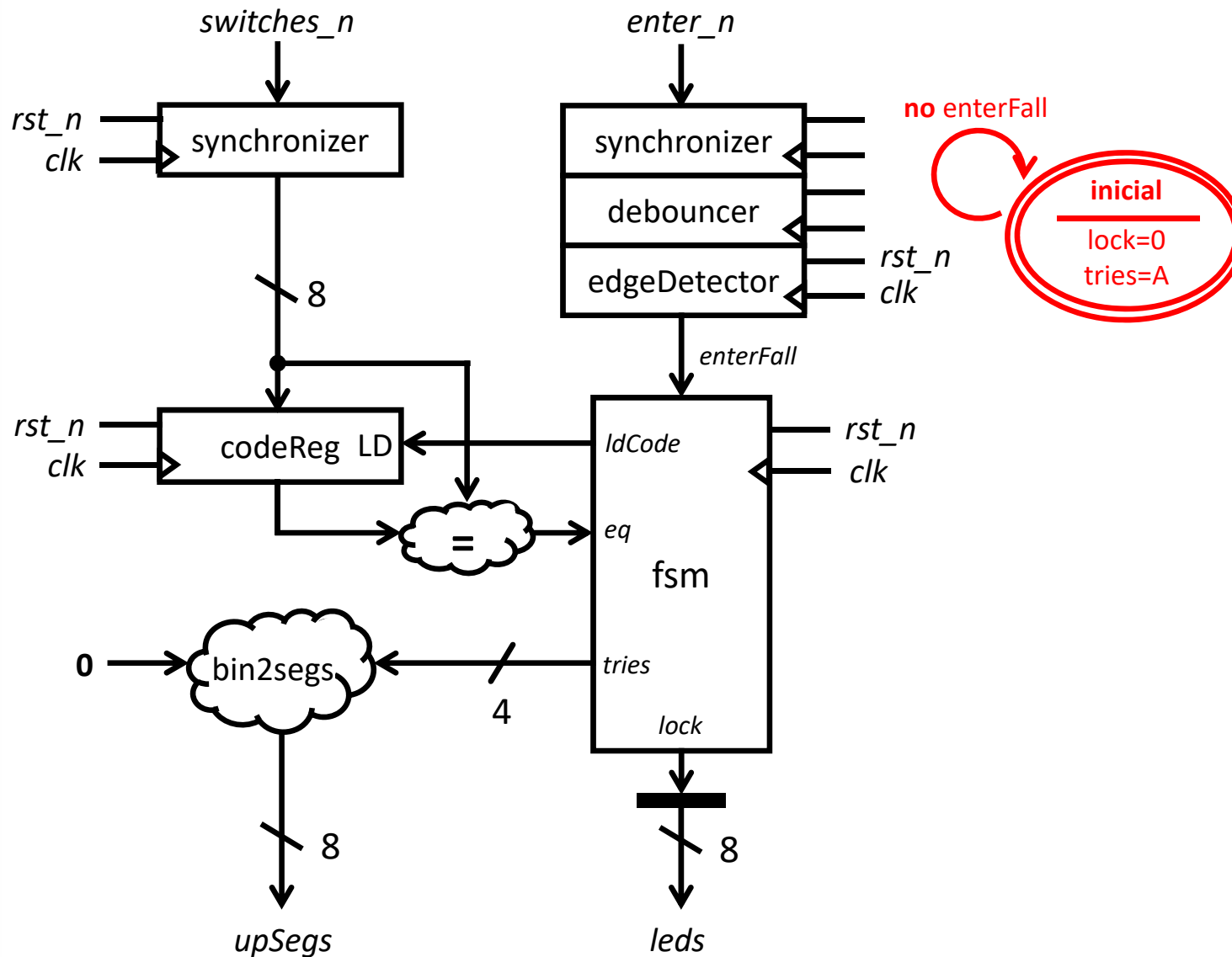
# Diseño principal

## diagrama RTL (i)



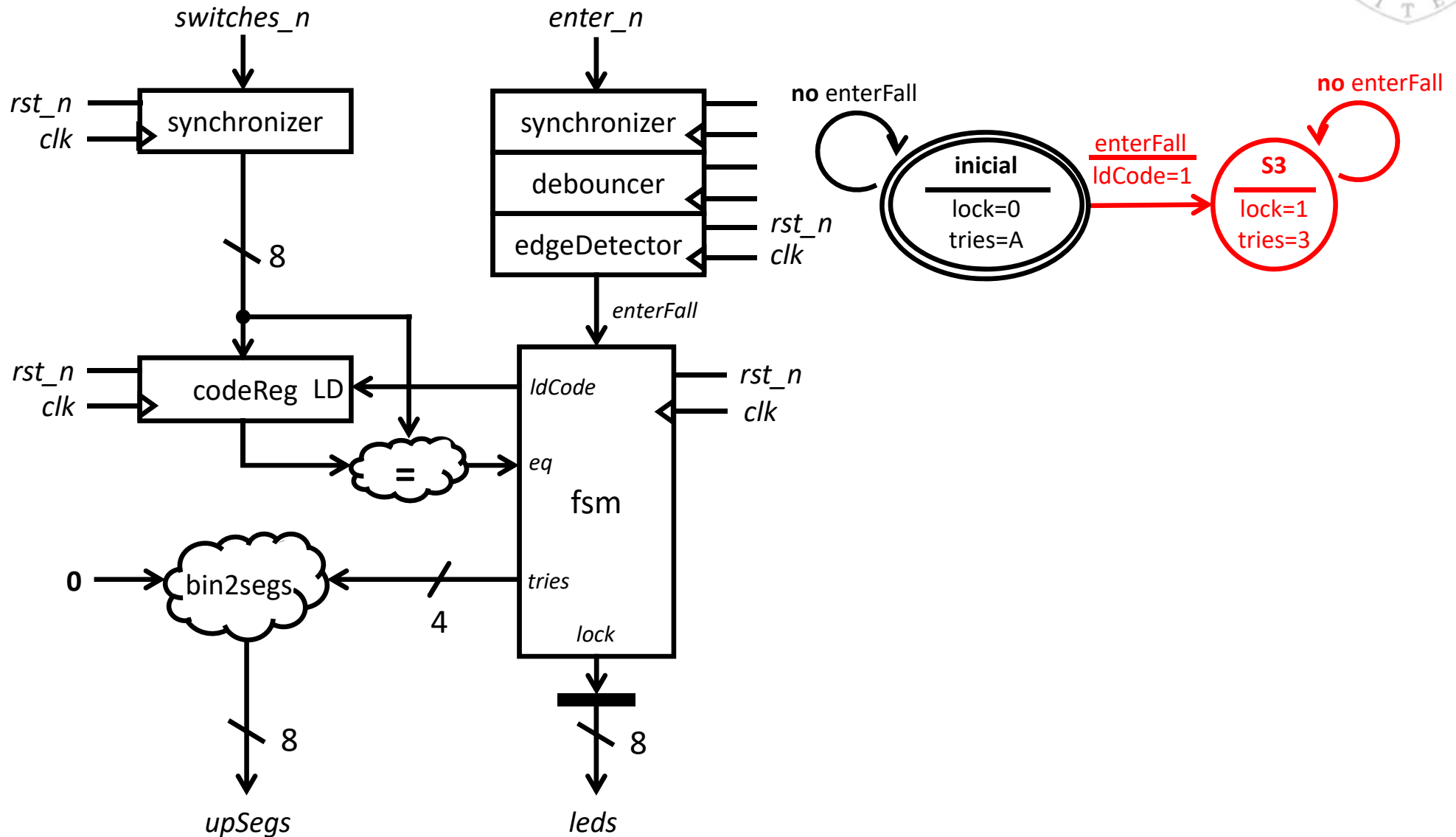
# Diseño principal

## diagrama RTL (i)



# Diseño principal

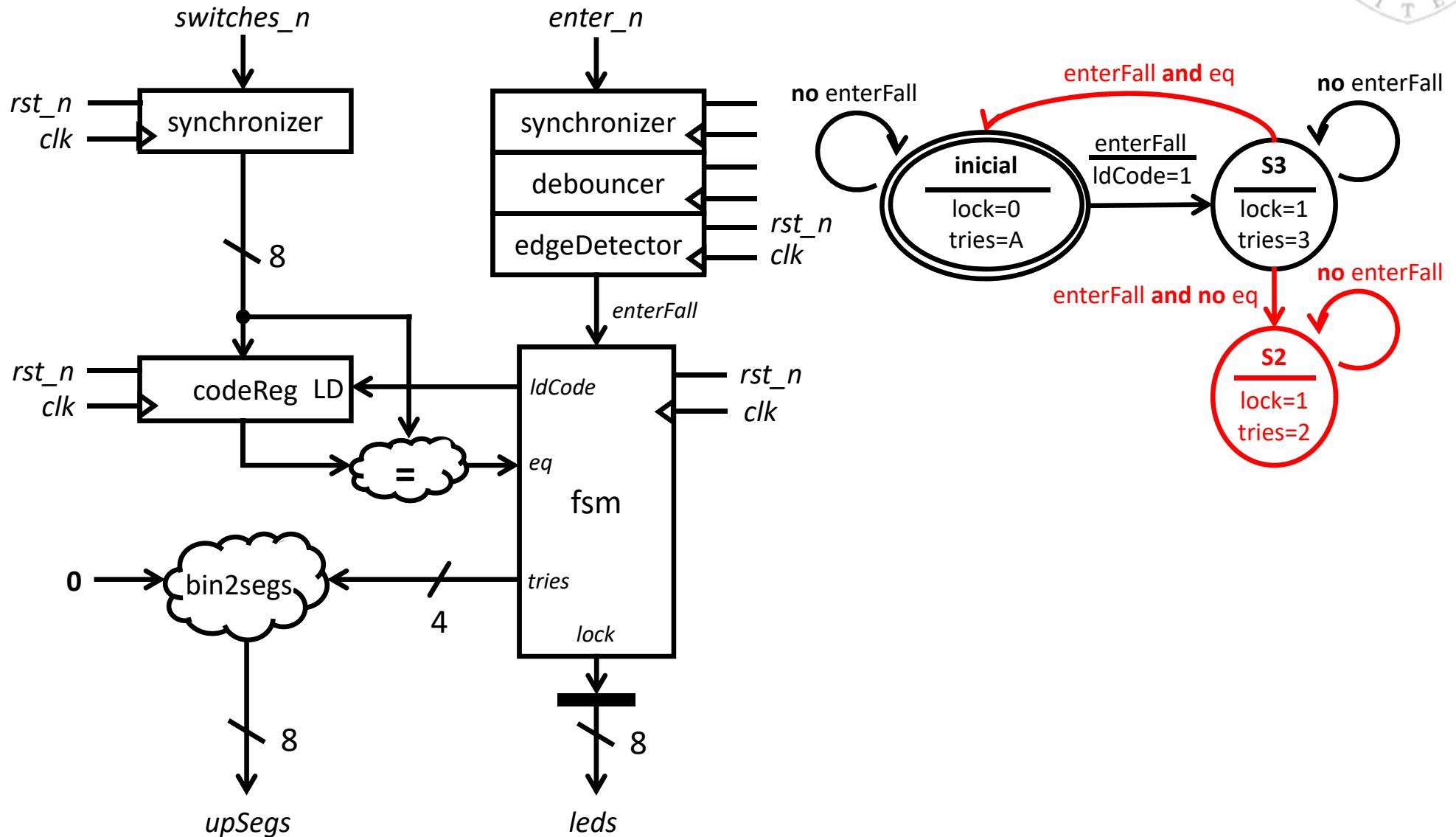
## diagrama RTL (i)





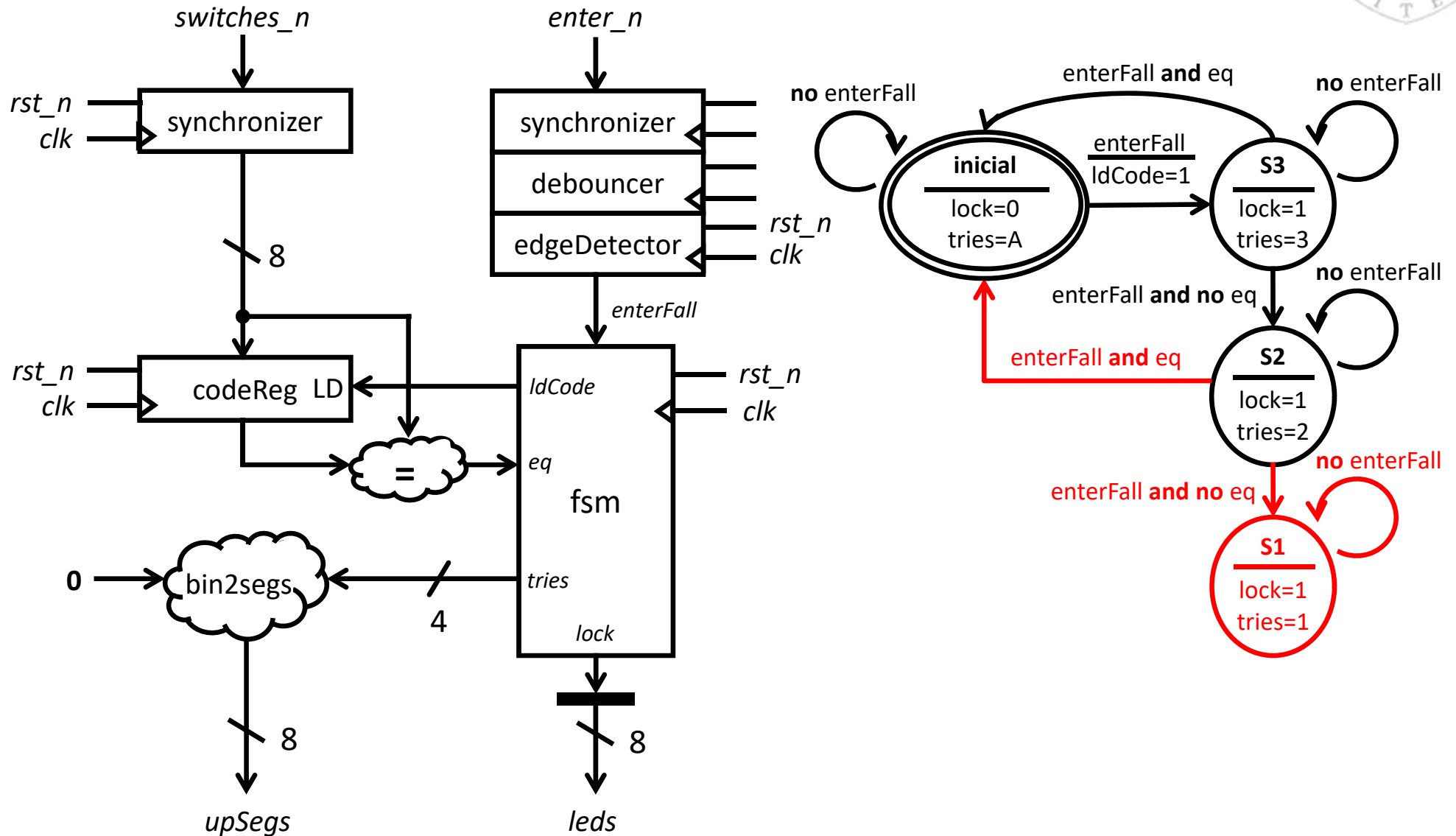
# Diseño principal

## diagrama RTL (i)



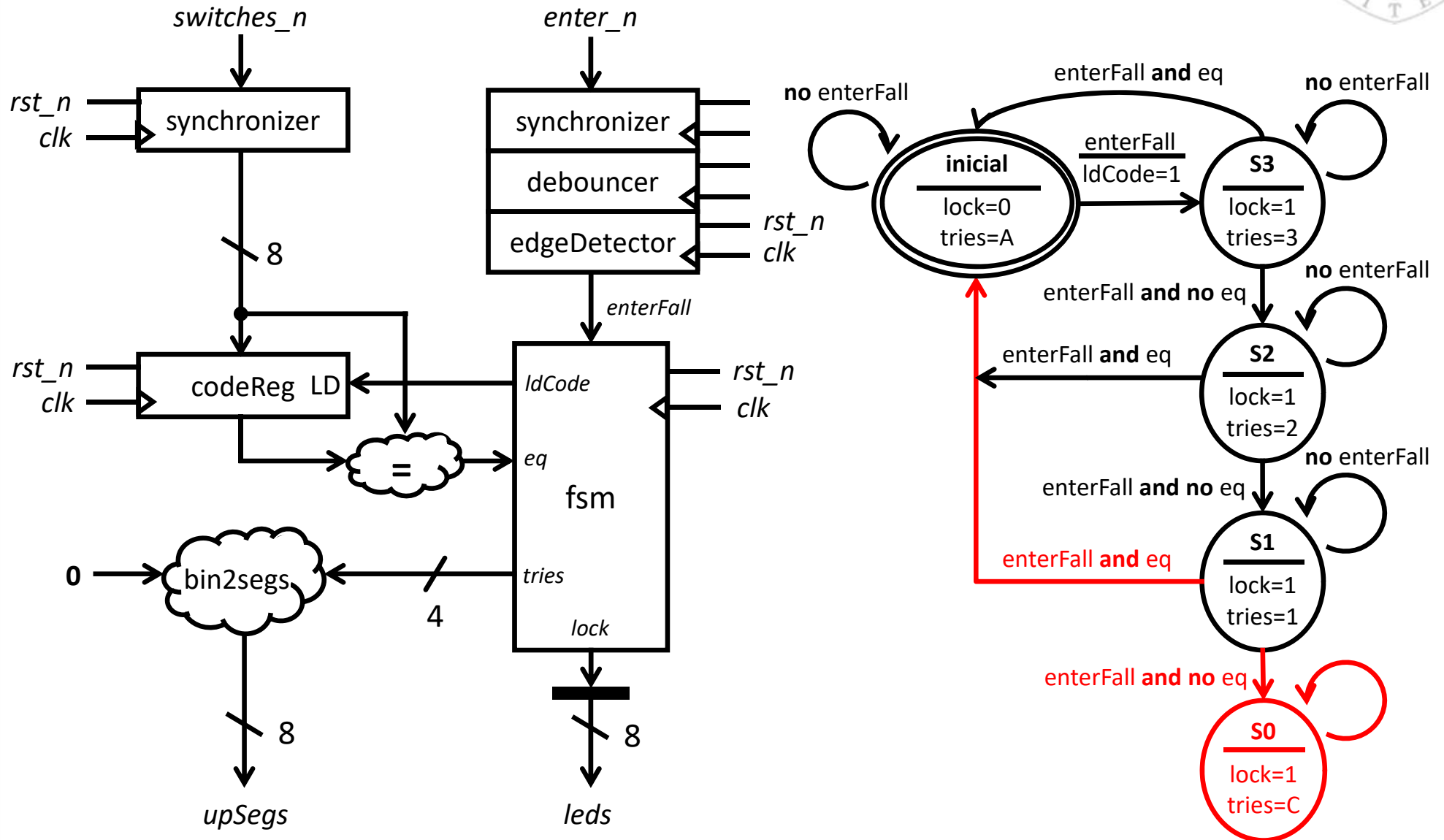
# Diseño principal

## diagrama RTL (i)



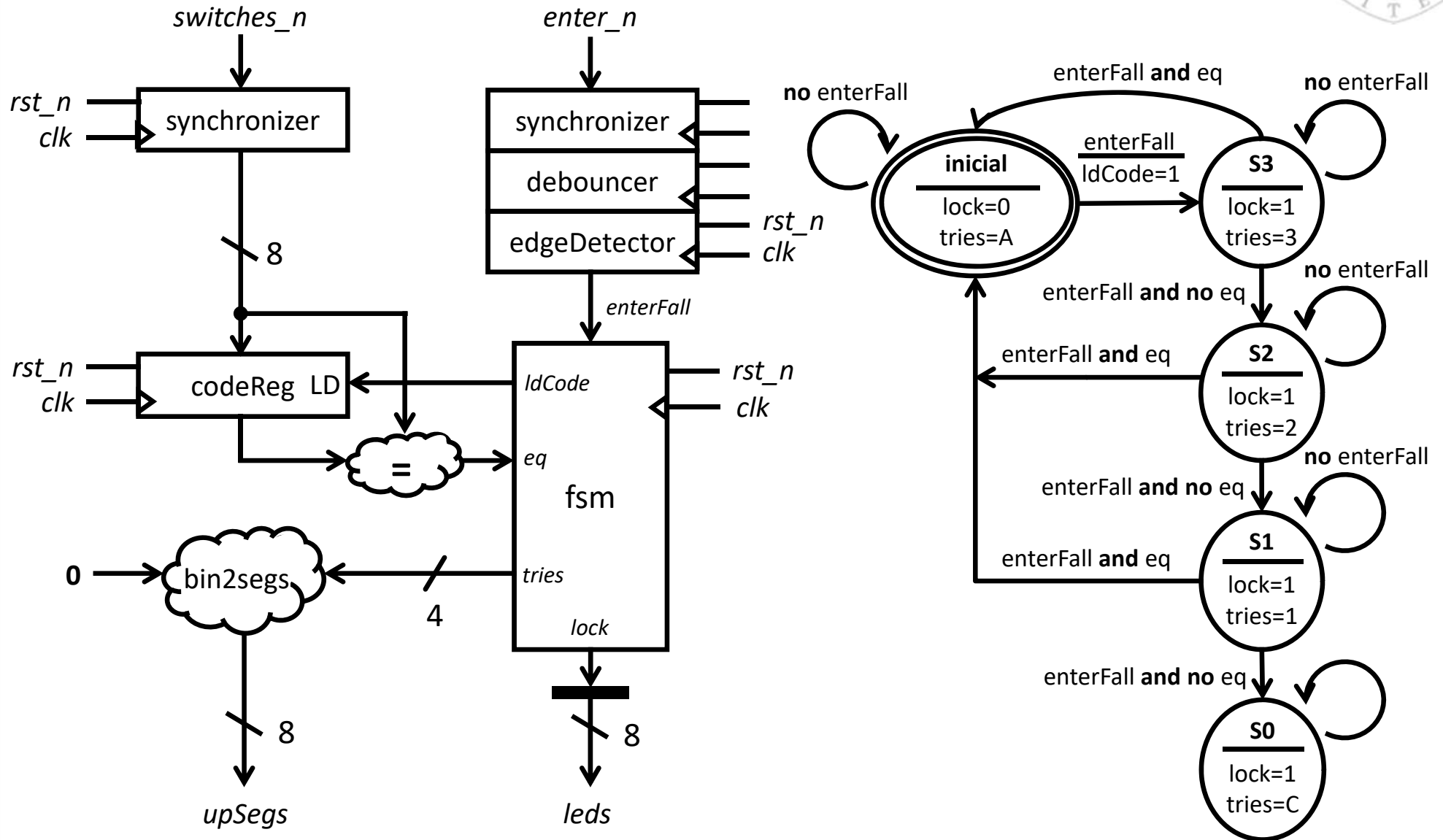
# Diseño principal

## diagrama RTL (i)



# Diseño principal

## diagrama RTL (i)

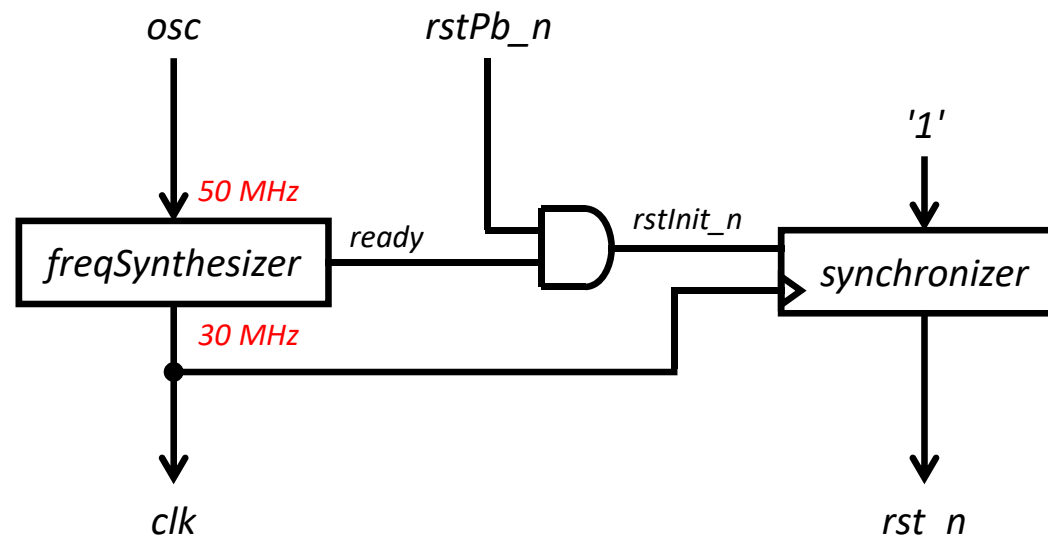


# Diseño principal

## diagrama RTL (ii)



- Adicionalmente a este diseño añadiremos:
  - Un **sintetizador de frecuencias** para reducir a 30 MHz la frecuencia de reloj.
  - Un **sincronizador** (activación asíncrona) a la señal de reset
    - El reset deberá estar activo hasta que el sintetizador alcance la frecuencia deseada.
  - La señal de **reloj externa** conectada al oscilador (50 MHz) se llamará **osc** y la señal de **reloj interna** (30 MHz) conectada a todos los biestables se llamará **clk**
  - La señal de **reset externa** conectada a un pulsador se llamará **rstPb\_n** y la señal de **reset interna** conectada a todos los biestables se llamará **rst\_n**



# Diseño principal

## lab3.vhd



```
library ieee; use ieee.std_logic_1164.all;
entity lab3 is
port
(
    rstPb_n      : in  std_logic;
    osc          : in  std_logic;
    enter_n      : in  std_logic;
    switches_n   : in  std_logic_vector(7 downto 0);
    leds         : out std_logic_vector(7 downto 0);
    upSegs       : out std_logic_vector(7 downto 0)
);
end lab3;

library ieee; use ieee.std_logic_unsigned.all;
use work.common.all;
architecture syn of lab3 is
    signal clk, rst_n : std_logic;
    signal ready, rstInit_n : std_logic;
    signal enterSync_n, enterDeb_n, enterFall : std_logic;
    signal switchesSync_n : std_logic_vector(7 downto 0);
    signal ldCode, eq, lock : std_logic;
    signal code : std_logic_vector(7 downto 0);
    signal tries : std_logic_vector(3 downto 0);

begin
    ...
end syn;
```

# Diseño principal

## lab3.vhd



begin

```
rstInit_n <= rstPb_n and ready;
```

```
resetSynchronizer : synchronizer
```

```
  generic map ( STAGES => 2, INIT => '0' )
```

```
  port map ( rst_n => rstInit_n, clk => clk, x => '1', xSync => rst_n );
```

```
clkGenerator : frequencySynthesizer
```

```
  generic map ( FREQ => 50_000, MODE => "LOW", MULTIPLY => 3, DIVIDE => 5 )
```

```
  port map ( ... );
```

```
enterSynchronizer : synchronizer
```

```
  generic map ( ... )
```

```
  port map ( ... );
```

```
enterDebouncer : debouncer
```

```
  generic map ( FREQ => 30_000, BOUNCE => 50 )
```

```
  port map ( ... );
```

```
enterEdgeDetector : edgeDetector
```

```
  port map ( ... );
```

```
switchesSynchronizer :
```

```
  for i in switches_n'range generate
```

```
  begin
```

```
    switchsynchronizer : synchronizer
```

```
      generic map ( ... )
```

```
      port map ( ... );
```

```
  end generate;
```

la frecuencia externa es 50 MHz

sincroniza, elimina rebotes y detecta flancos de enter

la frecuencia interna es 30 MHz

sincroniza switches



# Diseño principal

## lab3.vhd



```
fsm:
process (rst_n, clk, enterFall)
    type states is (initial, S3, S2, S1, S0);
    variable state: states;
begin
    if ... then
        ldCode <= ...;
        ...
    end if;
    case state is
        when initial =>
            tries <= ...;
            lock <= ...;
            ...
        end case;
    if rst_n='0' then
        state := ...;
    elsif rising_edge(clk) then
        case state is
            when initial =>
                ...
            end case;
        end if;
    end process;
```

```
codeRegister :
process (rst_n, clk)
begin
    ...
end process;

comparator:
eq <= ...;

righthConverter : bin2segs
    port map( ... );

leds <= ...;

end syn;
```

# Tareas



1. Crear el proyecto **lab3** en el directorio **DAS**
2. Descargar de la Web en el directorio **common** los ficheros:
  - **frequencySynthesizer.vhd**
3. Descargar de la Web en el directorio **lab3** los ficheros:
  - **lab3.vhd** y **lab3.ucf**
4. Completar el fichero **common.vhd** con la declaración del nuevo componente reusable.
5. Completar el código omitido en el fichero **lab3.vhd**
6. Añadir al proyecto los ficheros:
  - **common.vhd**, **bin2segs.vhd**, **synchronizer.vhd**, **debouncer.vhd**, **frequencySynthesizer.vhd**, **edgedetector.vhd**, **modcounter.vhd**, **lab3.vhd** y **lab3.ucf**
7. Sintetizar, implementar y generar el fichero de configuración.
8. Conectar la placa y encenderla.
9. Descargar el fichero **lab3.bit**



# Acerca de *Creative Commons*



## ■ Licencia CC (*Creative Commons*)

- Ofrece algunos derechos a terceras personas bajo ciertas condiciones. Este documento tiene establecidas las siguientes:



**Reconocimiento** (*Attribution*):

En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.



**No comercial** (*Non commercial*):

La explotación de la obra queda limitada a usos no comerciales.



**Compartir igual** (*Share alike*):

La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Más información: <https://creativecommons.org/licenses/by-nc-sa/4.0/>