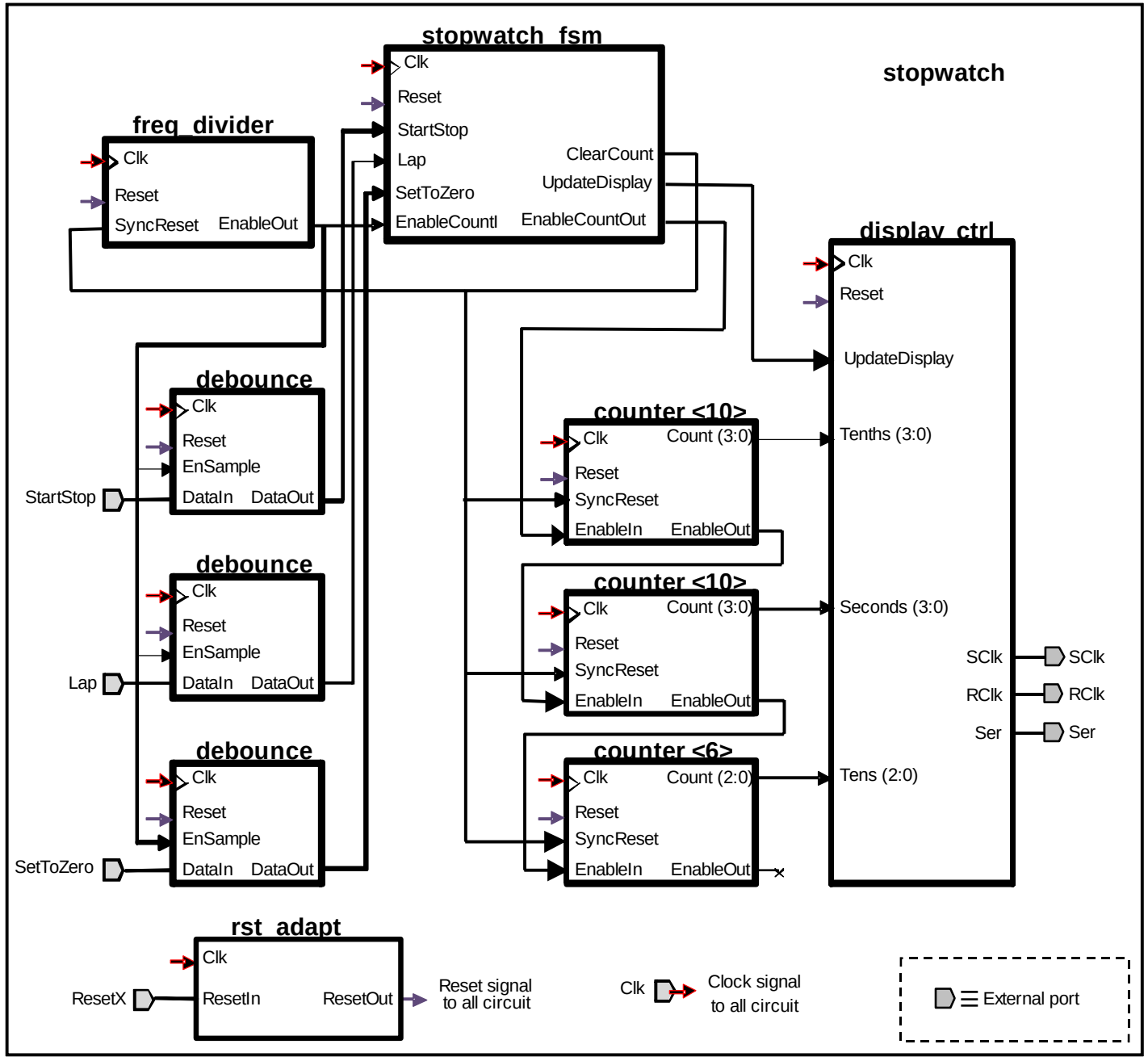
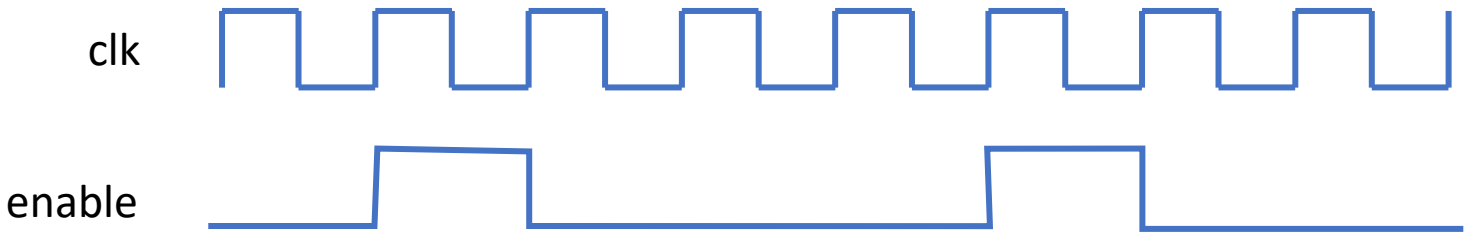
Práctica 2

Diapositivas utilizadas en las explicaciones

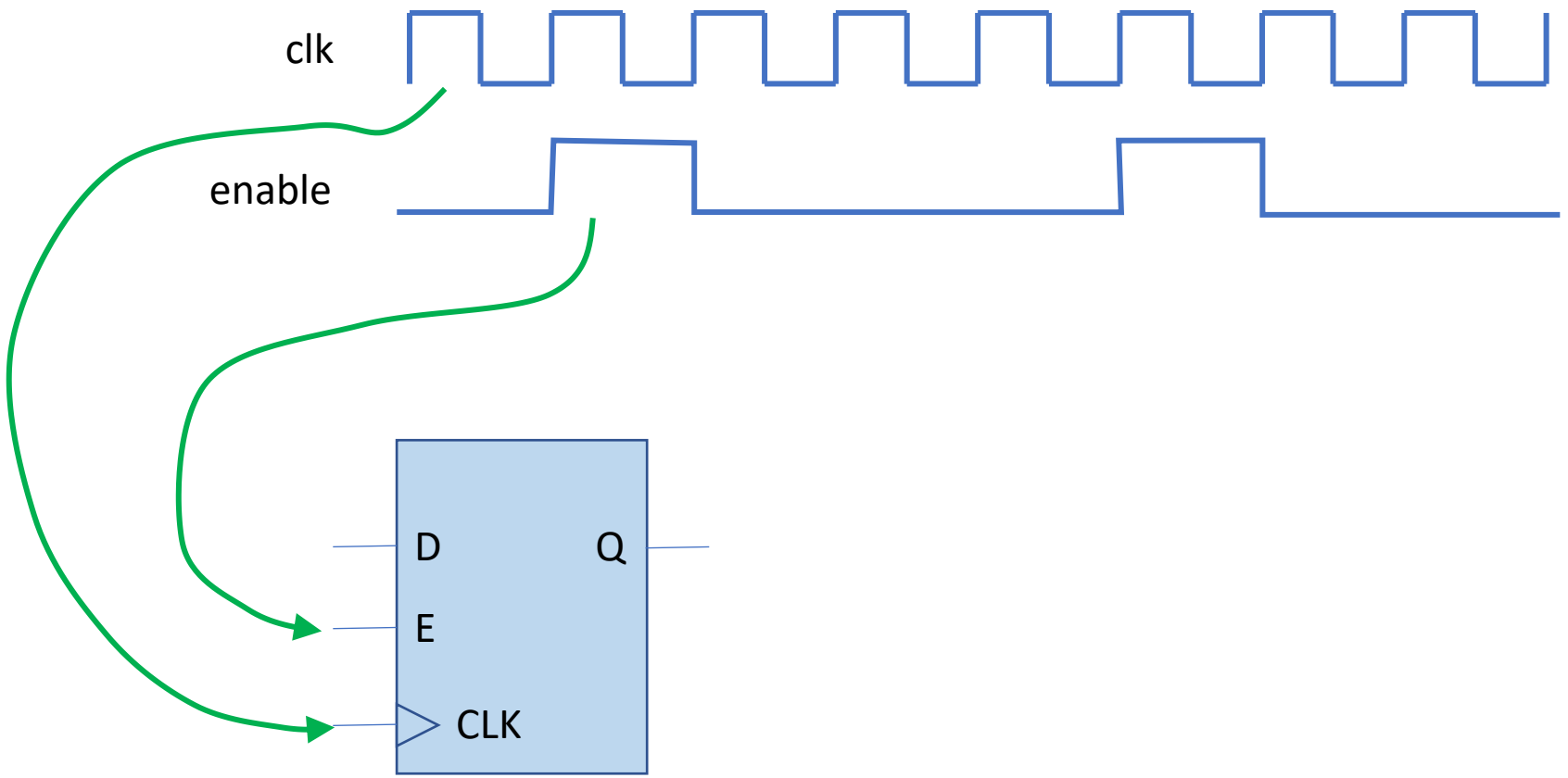


Esquema
stopwatch

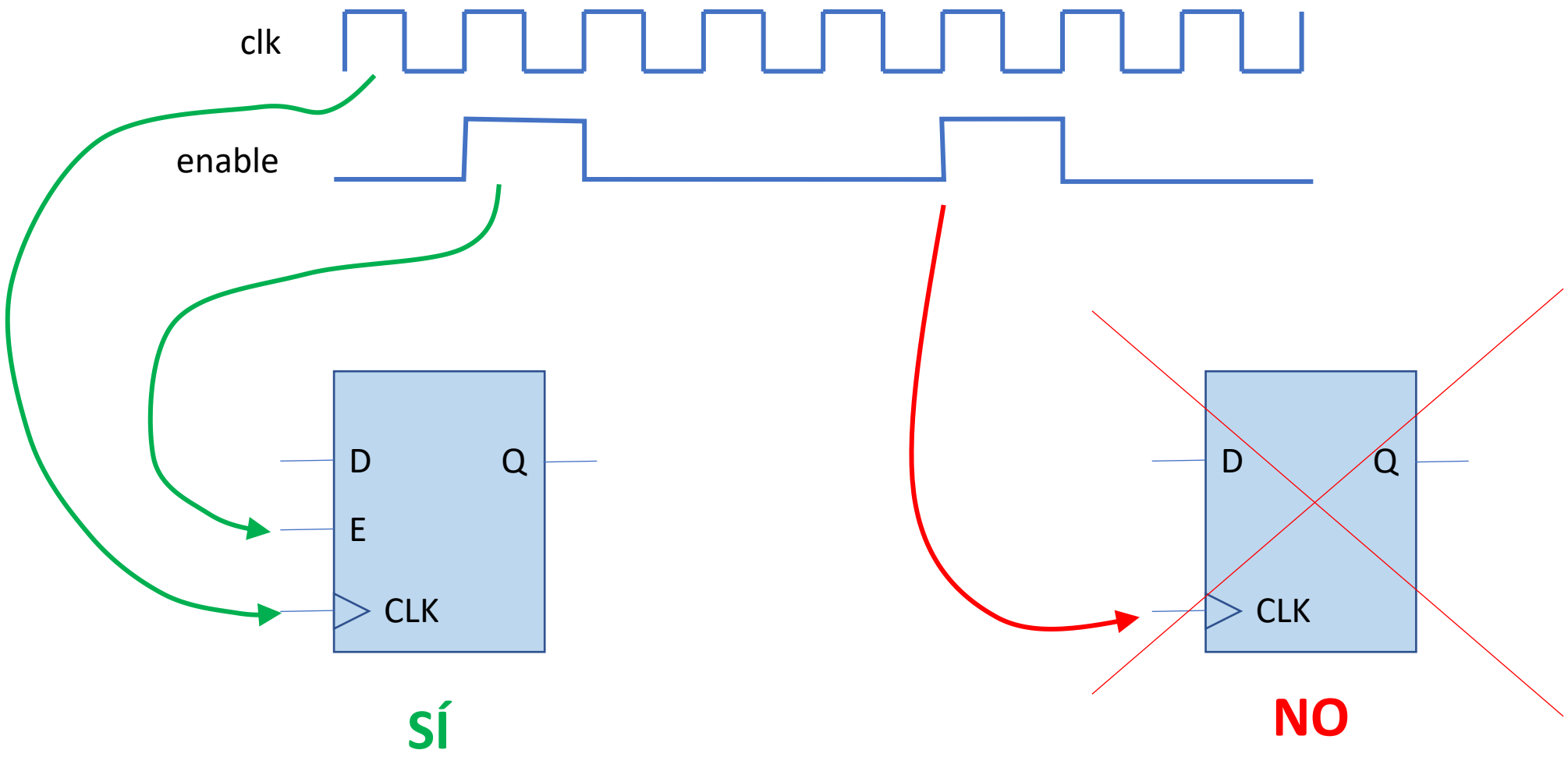
Base de tiempos = enable



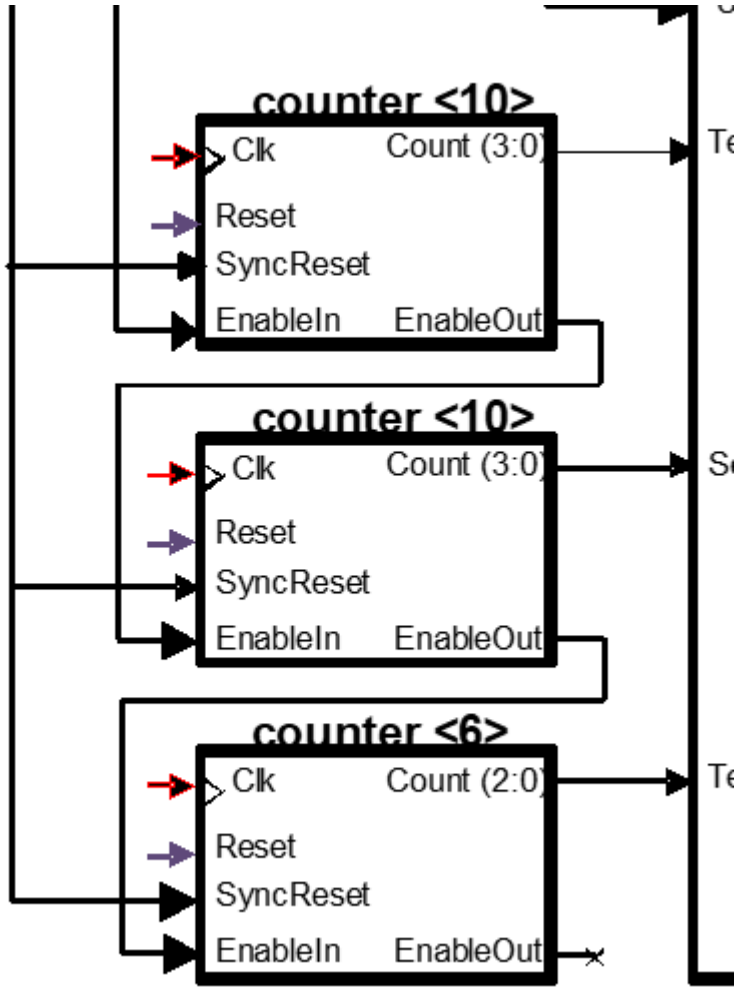
Base de tiempos = enable



Base de tiempos = enable



Generics

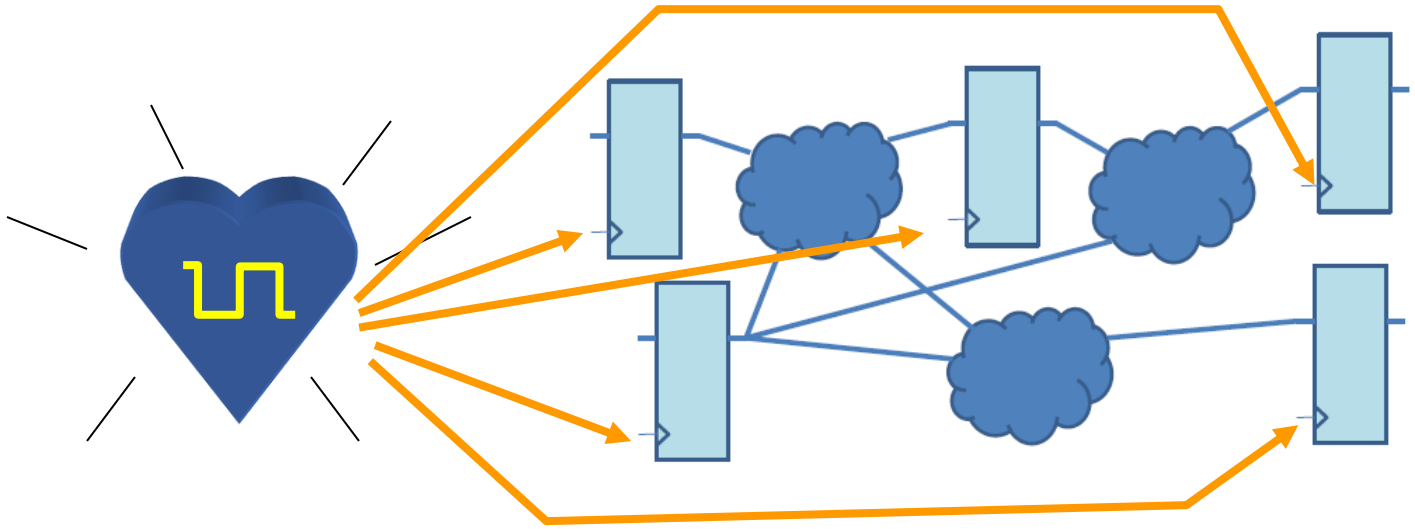



```
entity counter is
   generic (
      COUNT_LENGTH : integer := 100
   );
   port (
      Clk : in std_logic;
      ...
   );
end counter;
```

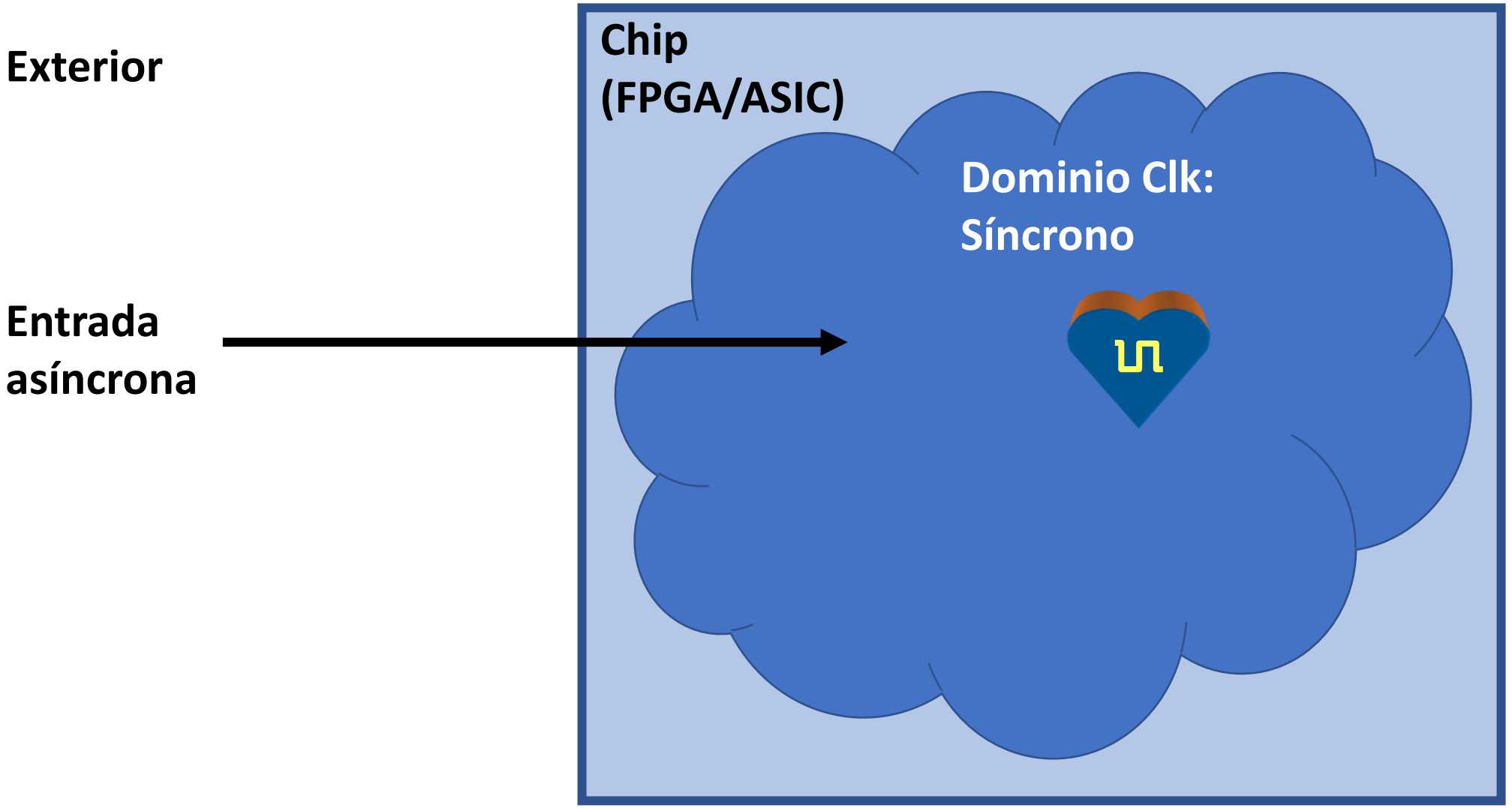
```
i_counter_tenths : counter
   generic map (
      COUNT_LENGTH => 10
   )
   port map (
      Clk => Clk,
      ...
   );
```

Diseño síncrono

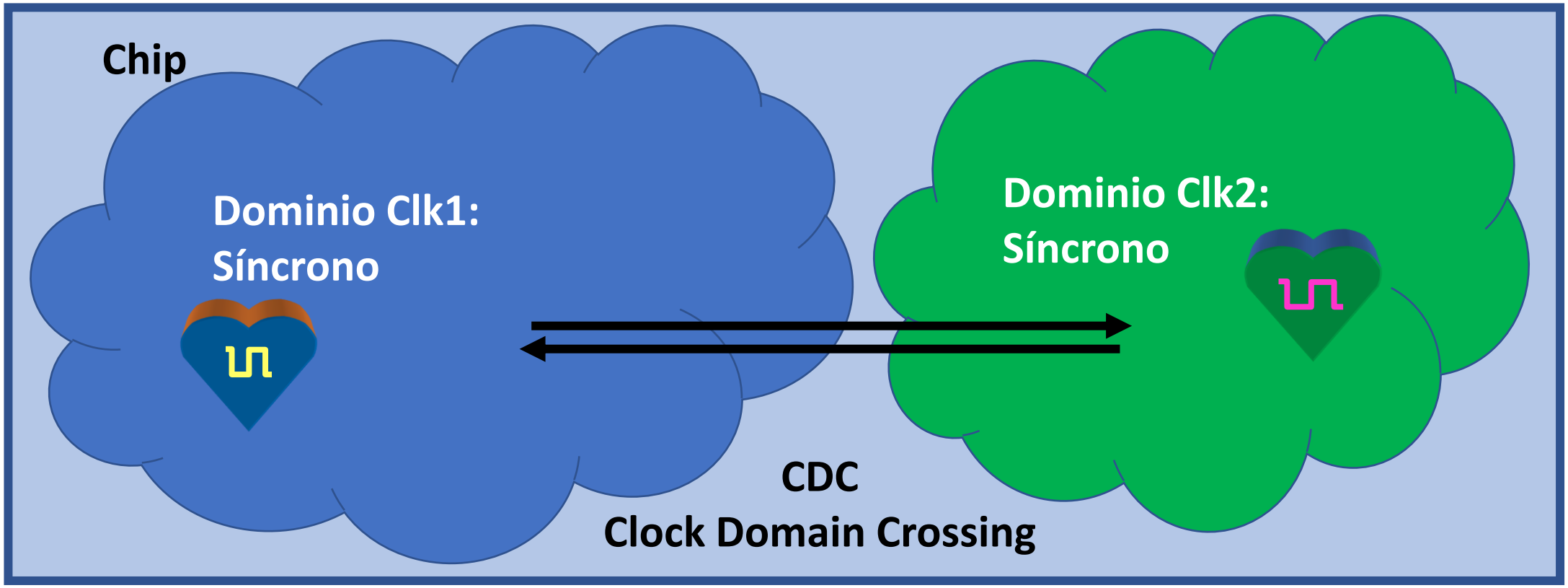
- **Ideal : reloj único** para todo el diseño
  - Soporte del concepto ***RTL***
  - Simplifica el *Static Timing Analysis*



Situación asíncrona 1: interfaz con el exterior



Situación asíncrona 2: diferentes “dominios de reloj”



***Clock Domain Crossing* (CDC)**

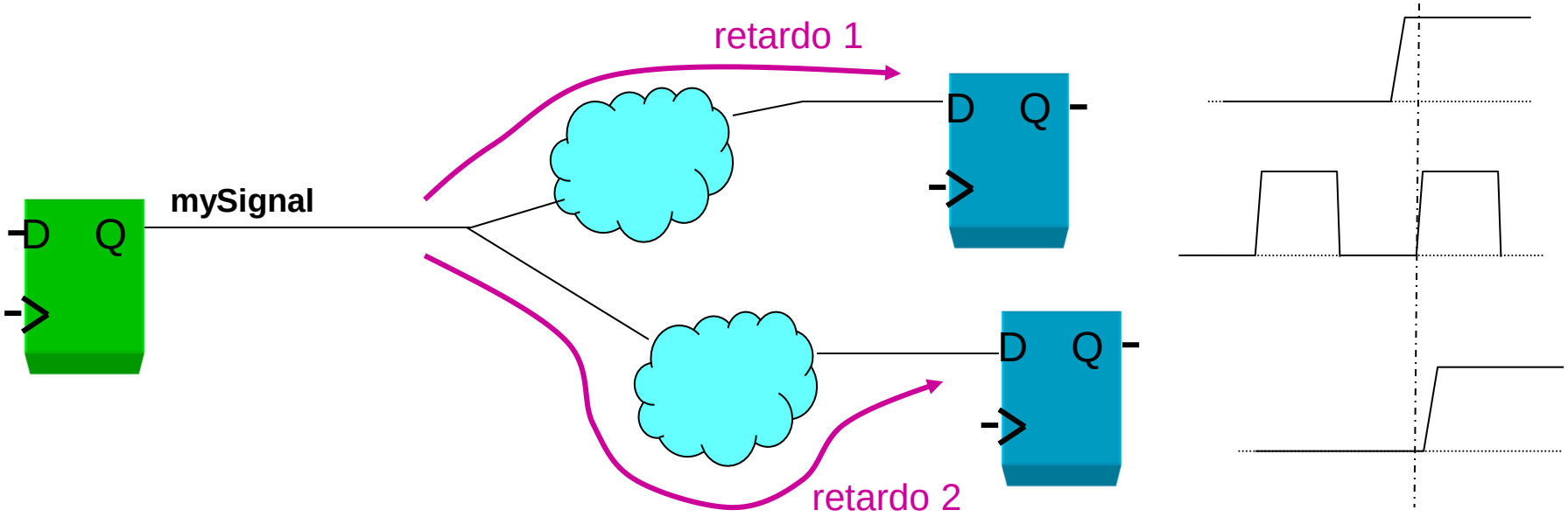
= Transferencia entre 2 dominios de reloj

→ Hay que asegurarse de que se haga de un modo seguro

(No confundir con el otro CDC, tener más de un dominio de reloj puede ser una enfermedad, pero también algo normal)

Muestreo múltiple

Si una señal de un dominio va a dos FFs de otro dominio totalmente asíncrono, debido a los diferentes retardos, uno de ellos puede considerar la señal antes y el otro después de una transición.

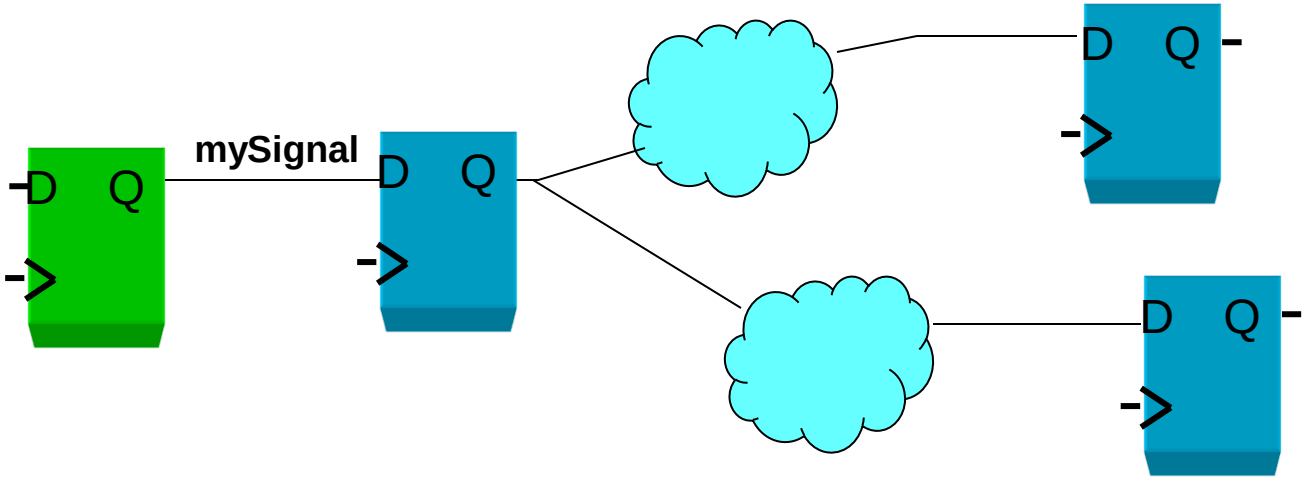


(situación similar a lo que puede pasar dentro de un dominio si no cumplimos la *constraint* de periodo de reloj)

¡ 1 señal en nuestro RTL, múltiples valores en realidad !

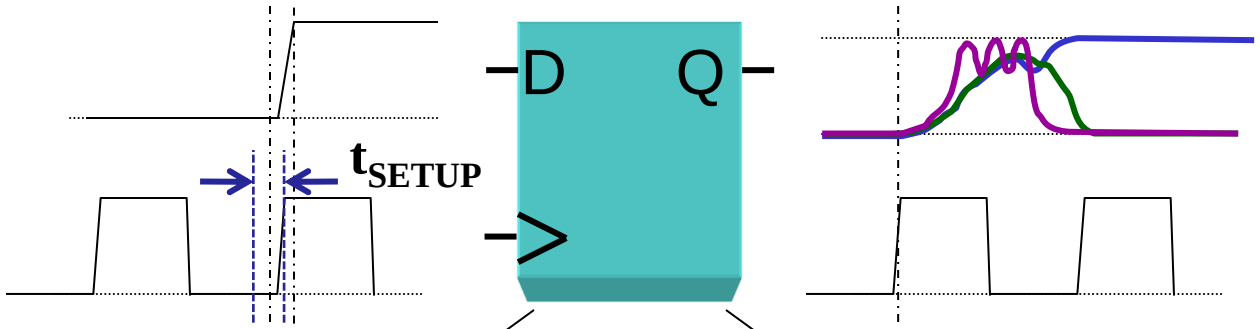
El muestreo ha de ser único

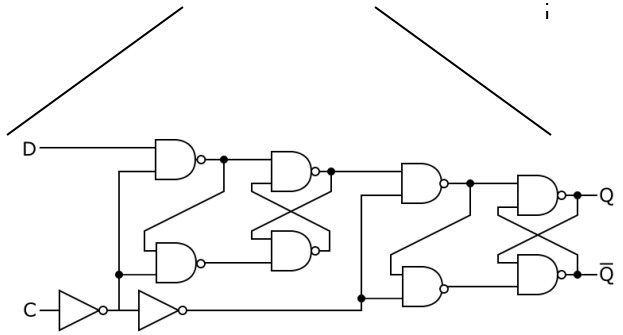
Mejor si capturamos la señal con un único FF y luego la usamos en donde haga falta:

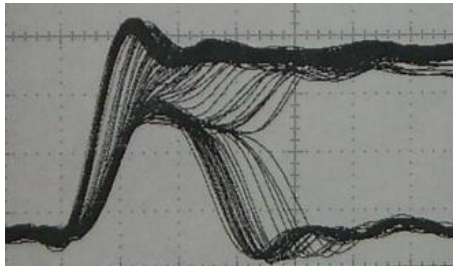


Metaestabilidad

- Cuando el dato de entrada cambia “a la vez” que el reloj, el FF entra en un estado *metaestable*.

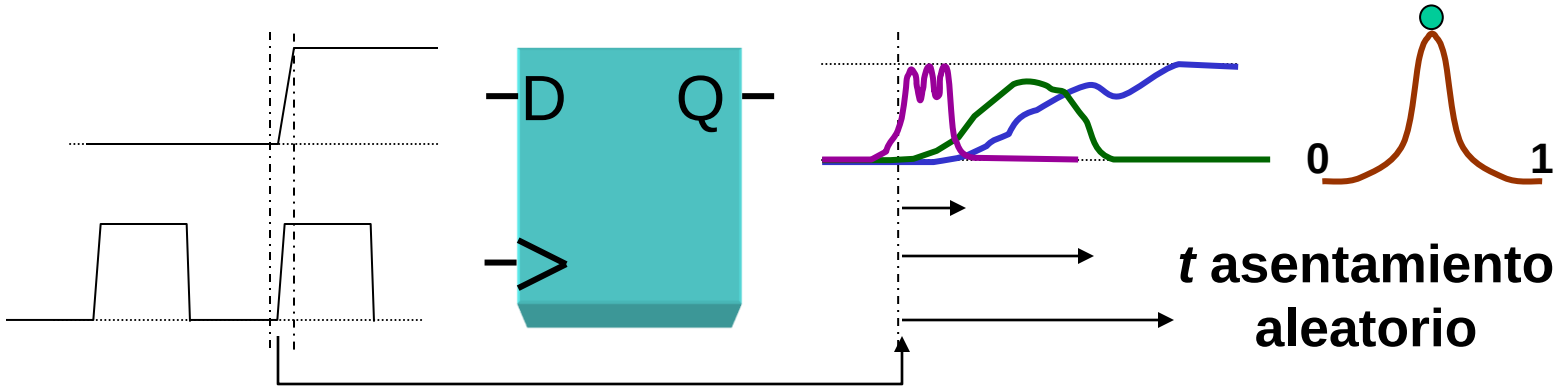






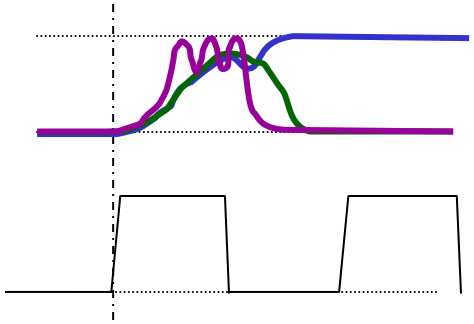
Metaestabilidad (2)

- La salida del estado metaestable finalmente se produce, pero no está acotado cuánto puede tardar.
  → Tiempo de asentamiento: distribución probabilística.



- Si la salida del FF es capturada por otros FFs cuando ya se ha estabilizado, no hay un problema en el funcionamiento de la lógica
  - Para esto hay que contar con cualquier retardo de lógica e interconexión hacia esos FFs destino

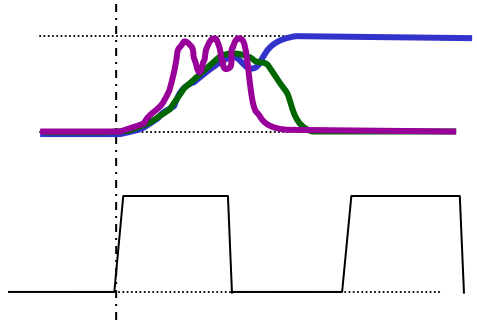
Metaestabilidad: efectos



- Si la salida del FF metaestable llega a múltiples sitios podría causar corrientes elevadas, siendo peligroso.
- Si la señal va a varios destinos, cada uno la puede interpretar de una manera con la llegada del flanco de reloj.

¡ 1 señal en nuestro RTL, múltiples valores en realidad !

Metaestabilidad: efectos (2)



Comportamiento exponencial

- Tiempo medio entre fallos:

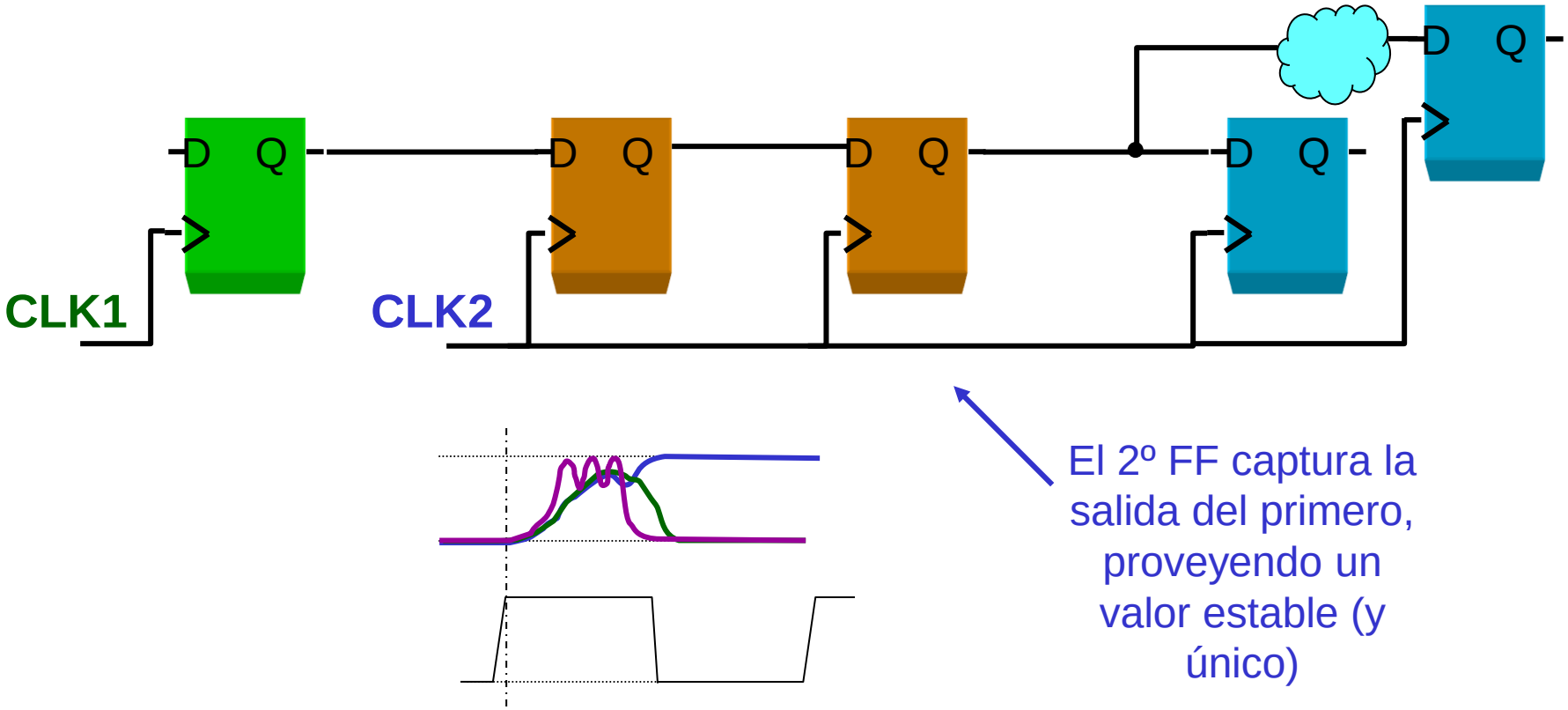
$$MTBF = \frac{e^{T/\tau}}{f_{Clk} \cdot f_{IN} \cdot T_w}$$

Cuanto más **tiempo haya disponible**, antes del siguiente flanco de reloj, para que la señal se asiente, menor probabilidad de fallo.

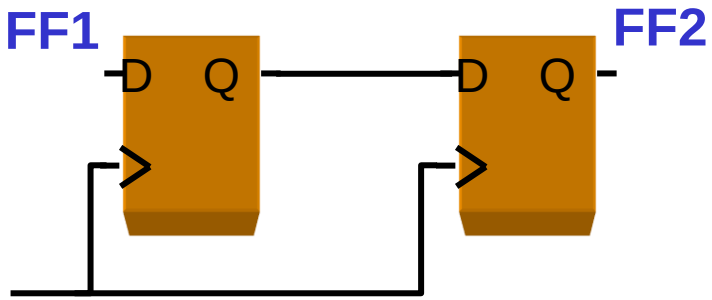
- f_{Clk} : Frecuencia de reloj
- f_{IN} : Frecuencia de transiciones en la entrada
- T_w : Ventana de susceptibilidad del FF
- τ : Constante de tiempo de asentamiento del FF
- T : Ventana de asentamiento

Tolerancia a la metaestabilidad

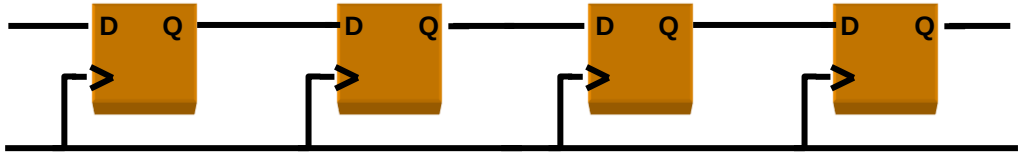
- No podemos evitar la metaestabilidad pero si tolerarla.
- Solución simple: utilizar 2 FFs de sincronismo



Sincronizador de cadena de FFs



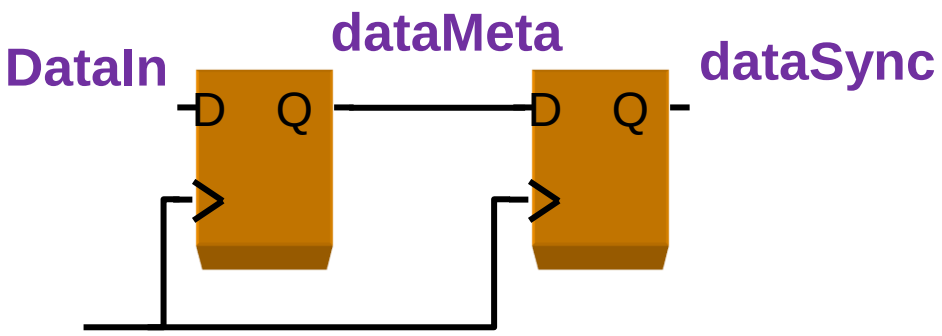
- Cuanta menor frecuencia, más fácil será que FF2 capture el dato ya estable.
- Cuanto más rápido el camino entre FF1 y FF2, más facilitaremos que FF2 capture el dato ya estable.
- Si aumentamos el número de FFs, introducimos un factor multiplicativo en el MTBF: interesante si la frecuencia es alta (donde “alta” depende de la tecnología utilizada).



Según la circunstancias puede ser recomendable usar más de 2 FFs

Sincronizador en VHDL: 2 FFs

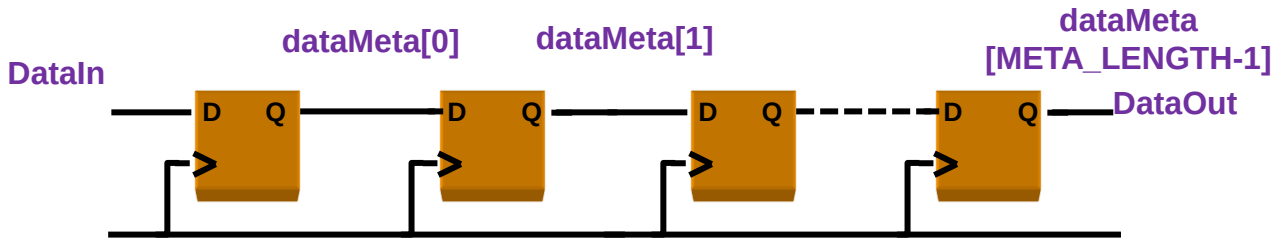
```
entity my_block is
   port (
      Clk      : in  std_logic; -- Reloj
      DataIn   : in  std_logic; -- Entrada de dato de otro dominio
      ...
   );
end my_block;

architecture rtl of my_block is
   signal dataMeta : std_logic;
   signal dataSync : std_logic;
begin

   process (Clk)
   begin
      if rising_edge(Clk) then
         dataMeta <= DataIn;
         dataSync <= dataMeta;
      end if;
   end process;
   ...
   end architecture rtl;
```



Barrera de antimetaestabilidad

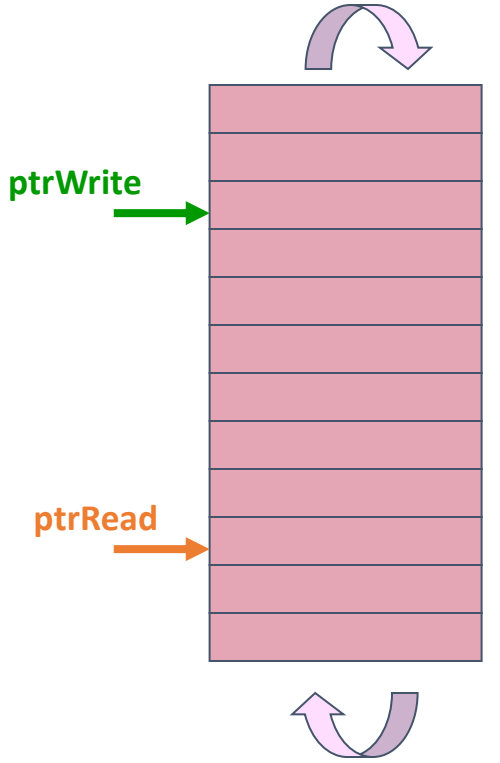
Sincronizador en VHDL: N FFs

```
entity sync is
   generic (
      META_LENGTH : integer := 2 -- Numero de FFs en la cadena (min=2)
   );
   port (
      DataIn   : in  std_logic; -- Entrada de dato
      Clk      : in  std_logic; -- Reloj con el que sincronizar la salida
      DataOut  : out std_logic  -- Salida de dato
   );
end sync;

architecture rtl of sync is
   signal dataMeta : std_logic_vector (META_LENGTH-1 downto 0);
begin
   process (Clk)
   begin
      if rising_edge(Clk) then
         dataMeta <= dataMeta (META_LENGTH-2 downto 0) & DataIn;
      end if;
   end process;

   DataOut <= dataMeta (META_LENGTH-1);
end architecture rtl;
```



Otros mecanismos de sincronización entre dominios de reloj

- En circuitos con más de un dominio de reloj, se aplican diversas técnicas para transferir señales entre dominios:
  - Cadenas de antimetaestabilidad
  - FIFOs
  - Código *Gray* en buses
  - *Handshaking*
  - …

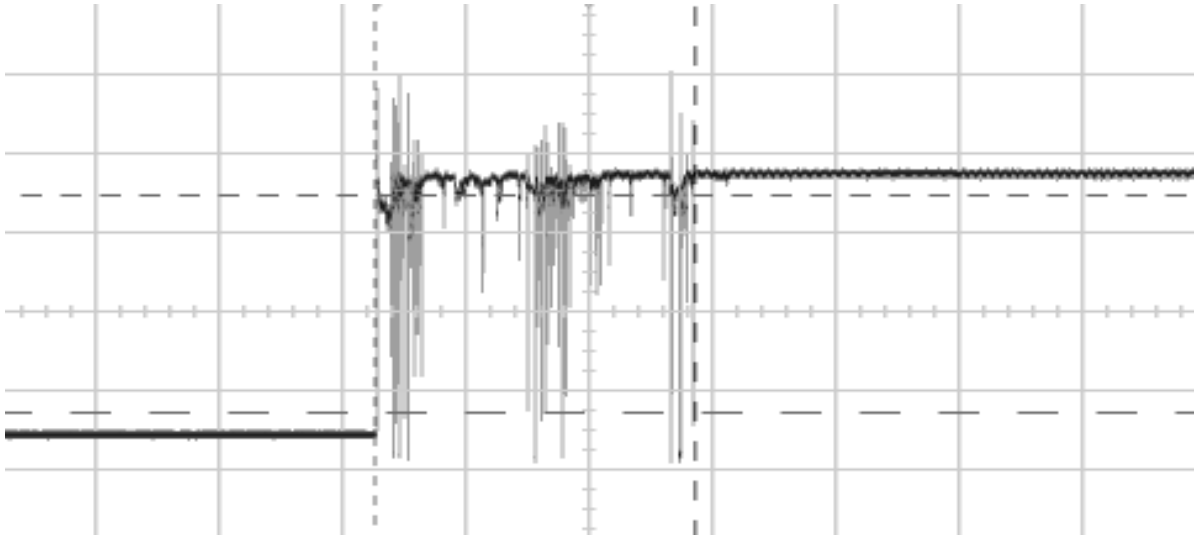


Ej. FIFO entre dos dominios de reloj

- En el laboratorio de DIE solo usaremos 1 dominio interno y solo veremos las cadenas de antimetaestabilidad.

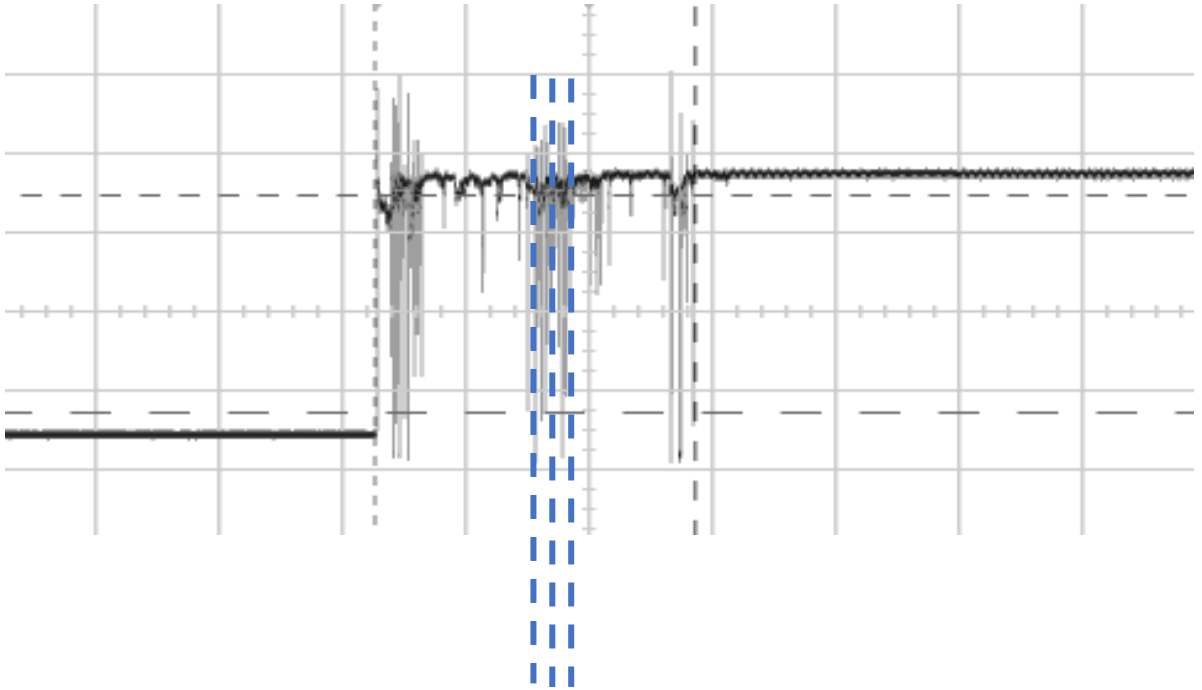
Debouncing de pulsadores

- El pulsado de un botón no produce una señal limpia, hay “rebotes” entre 0 y 1 hasta que la señal finalmente se estabiliza.



Debouncing de pulsadores

- El pulsado de un botón no produce una señal limpia, hay “rebotes” entre 0 y 1 hasta que la señal finalmente se estabiliza.

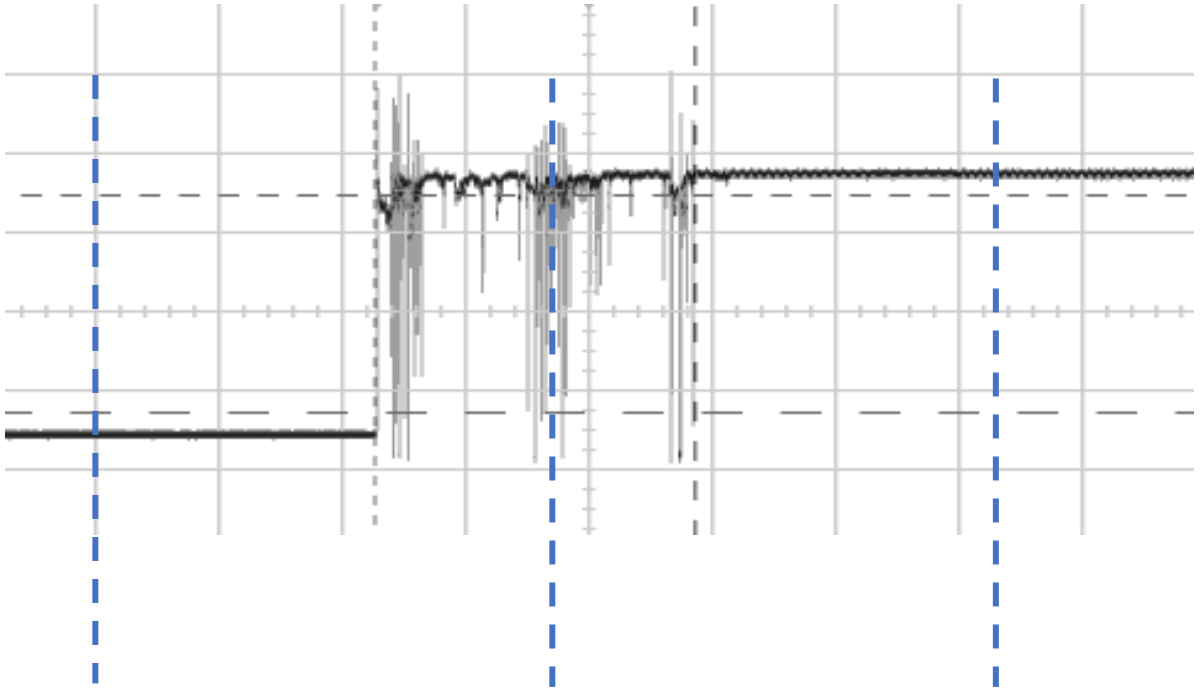


Si muestreamos la entrada del pulsador muy rápidamente, veremos las subidas y bajadas:

…….-0-0-0-0-0-1-0-1-0-1-1-1-1-0-1-1-0-…….-1-1-1-1-1-1-1-1……

Debouncing de pulsadores

- El pulsado de un botón no produce una señal limpia, hay “rebotes” entre 0 y 1 hasta que la señal finalmente se estabiliza.



Si espaciamos los muestreos, solo veremos una subida:

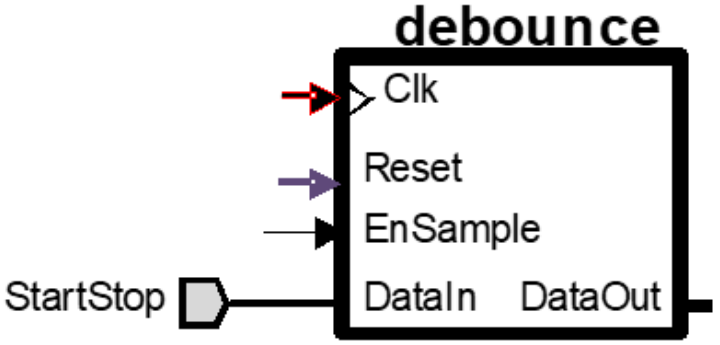
… 0 0 1 1 1 … o bien:

… 0 0 0 1 1 …

Bloque *debounce*

```
entity debounce is
   port (
      Clk      : in  std_logic;  -- Clock (rising edge)
      Reset    : in  std_logic;  -- Async reset, active high
      EnSample : in  std_logic;  -- Sample DataIn when high
      DataIn   : in  std_logic;  -- Data input
      DataOut  : out std_logic   -- Filtered data output
   );
end debounce;

------------------------------------------------------------
-- Architecture

architecture Behavioral of debounce is

   -- Anti-metastability FF chain for the input:
   signal dataInMeta : std_logic_vector (1 downto 0);

begin

   process (Clk, Reset)
   begin
      if Reset = '1' then
         dataInMeta <= (others => '0');
      elsif rising_edge (Clk) then
         dataInMeta <= dataInMeta(0) & DataIn;
         if EnSample = '1' then
            DataOut <= dataInMeta(1);
         end if;
      end if;
   end process;

end Behavioral;
```

Enable = base de tiempos 0.1 s



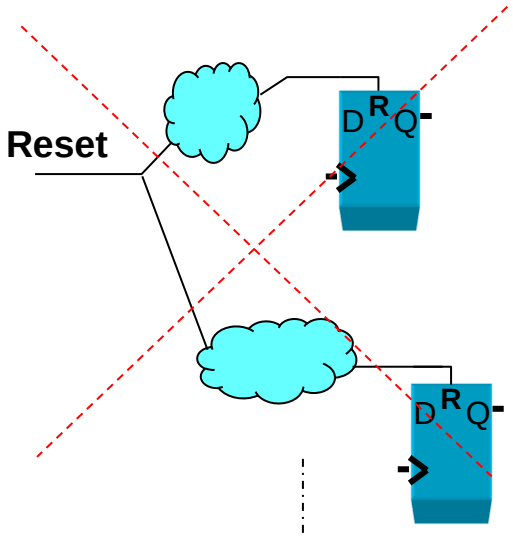
Antimetaestabilidad

Muestreo espaciado del dato ya sincronizado

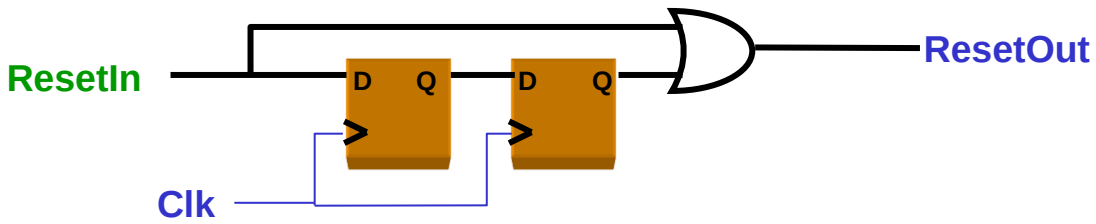
Sincronización del reset

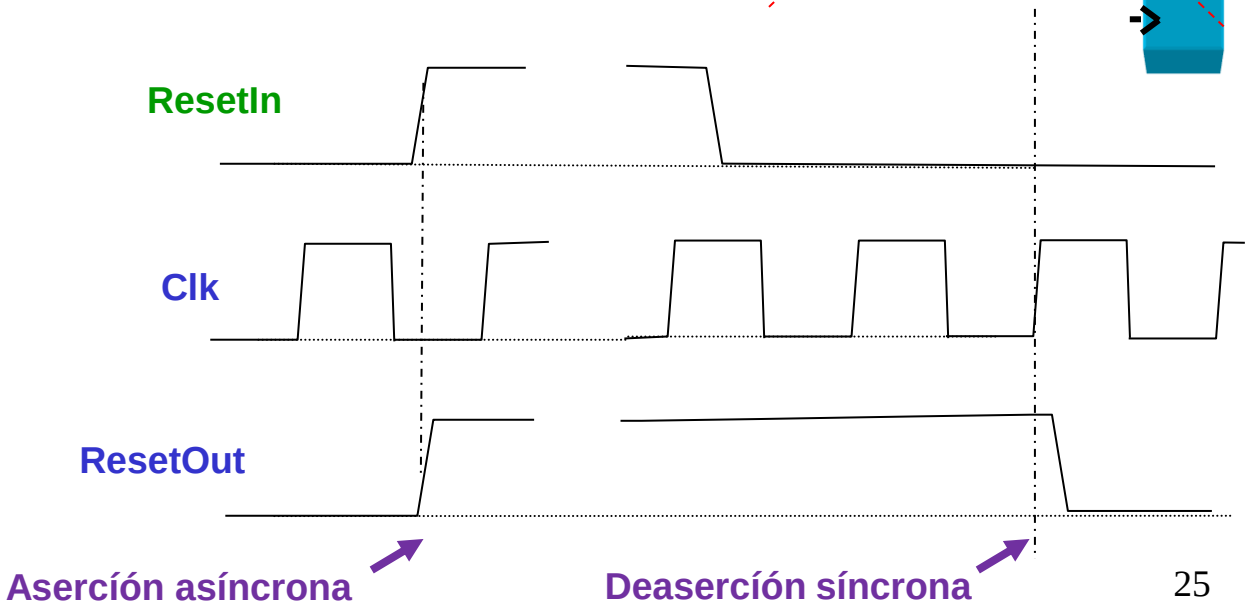
- Cuando utilizamos un reset para un circuito, este puede venir del exterior o de otro dominio de reloj.
- En tal caso, es necesario sincronizarlo.
- Lo importante es sincronizar la "**de-aserción**", no la "aserción":

  Nos importa que todo el circuito **salga** de reset a la vez.



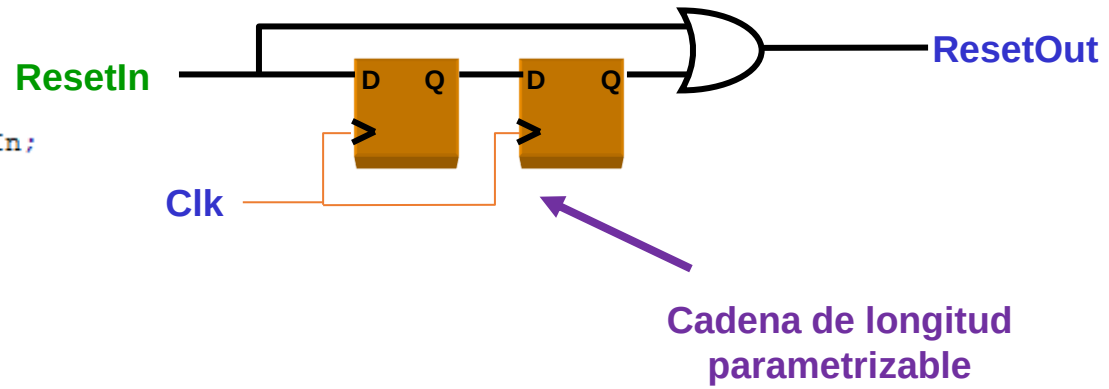
En ASICs es típico un esquema como el siguiente:



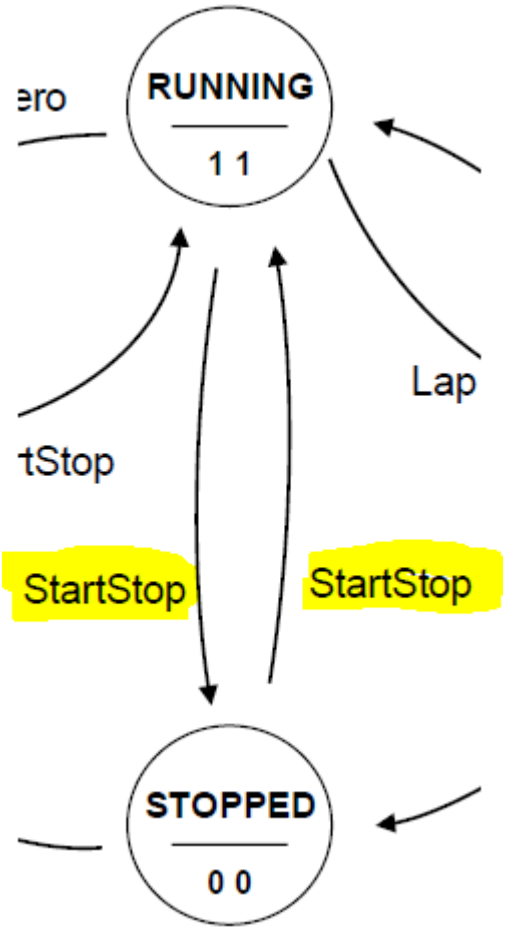


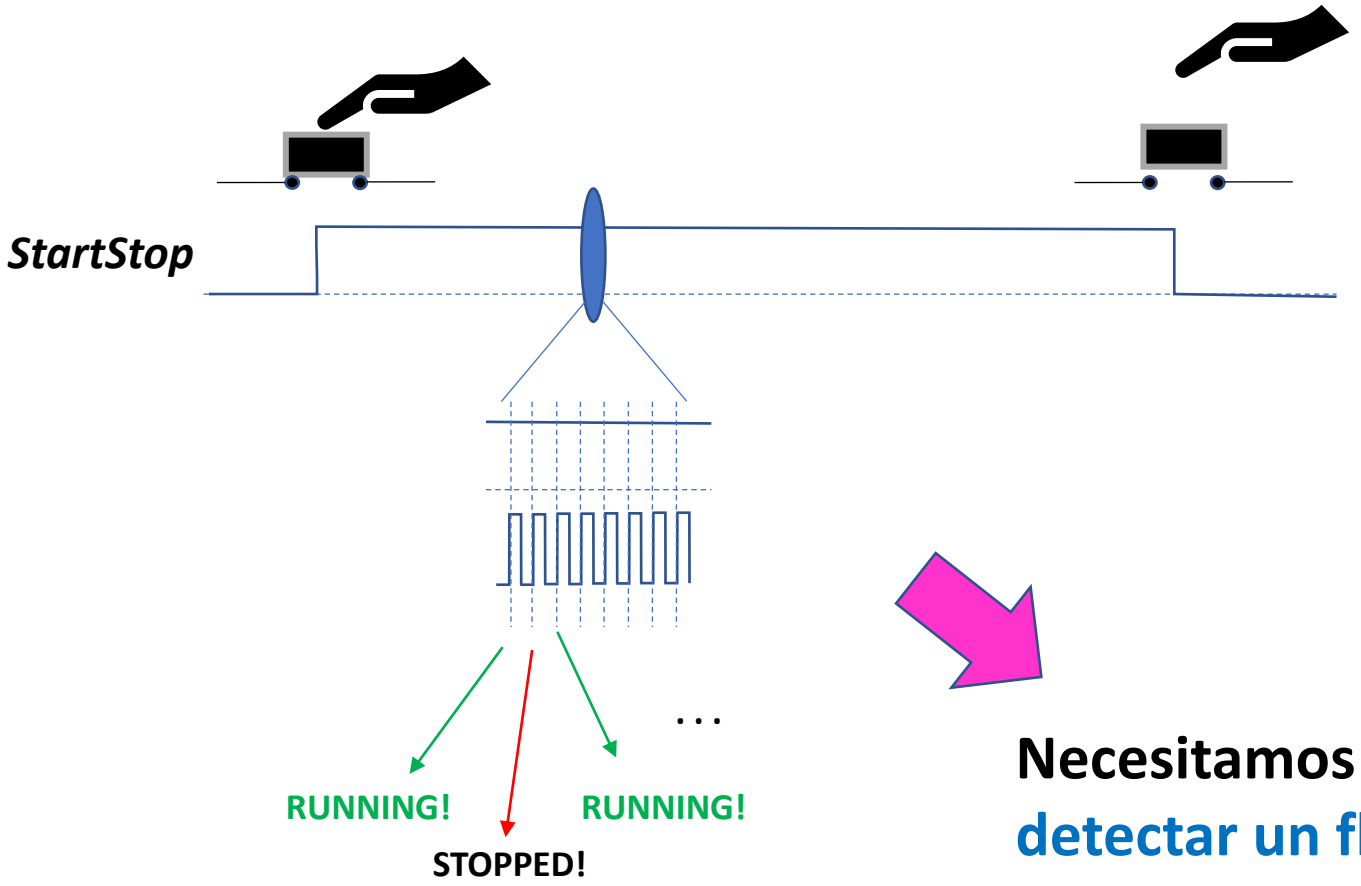
Bloque *rst_adapt*

```
entity rst_adapt is
   generic (
      META_DEPTH : integer := 3  -- Number of flops in anti-metastability
                                 --   chain. Optimum configuration
                                 --   depends on clock frequency.
   );
   port (
   -- Clock and external reset inputs:
      Clk      : in  std_logic;
      ResetIn  : in  std_logic;
      -- Synchronized reset output:
      ResetOut : out std_logic
   );
end rst_adapt;

-------------------------------------------------------------------------------

architecture rtl of rst_adapt is
   -- Signal declaration for anti-metastability flip-flop chain:
   signal resetMeta : std_logic_vector (META_DEPTH-1 downto 0);
begin

   -- Anti-metastability (shift reset bit in):

   process (Clk)
   begin
      if Clk'event and Clk = '1' then
         resetMeta <= resetMeta(META_DEPTH-2 downto 0) & ResetIn;
      end if;
   end process;

   -- Assertion is asynchronous, deassertion is synchronous:

   ResetOut <= ResetIn or resetMeta(META_DEPTH-1);

end rtl;
```



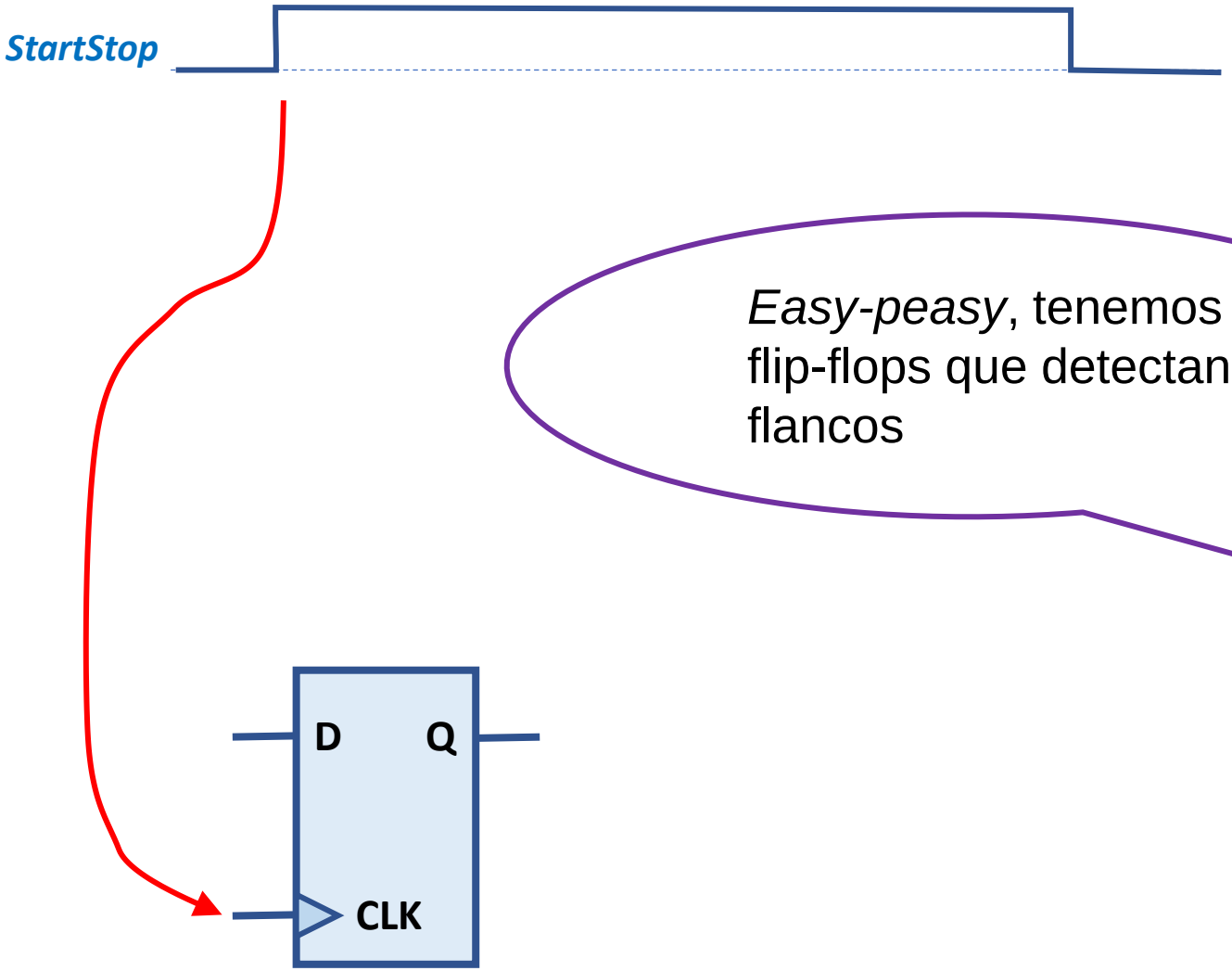
Detección de botones en la máquina de estados





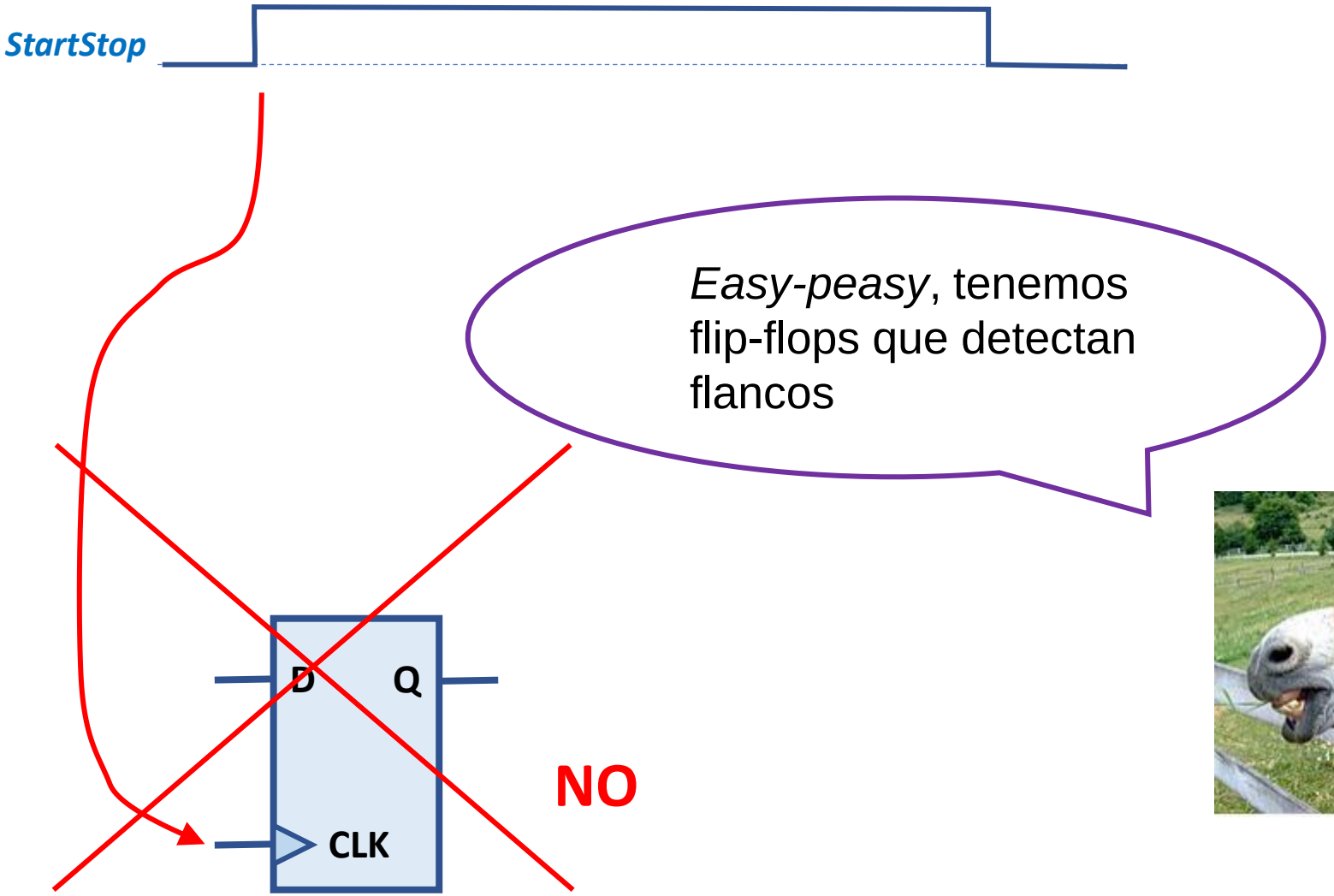
Necesitamos
detectar un flanco

Detectar un flanco (p.ej. de subida)

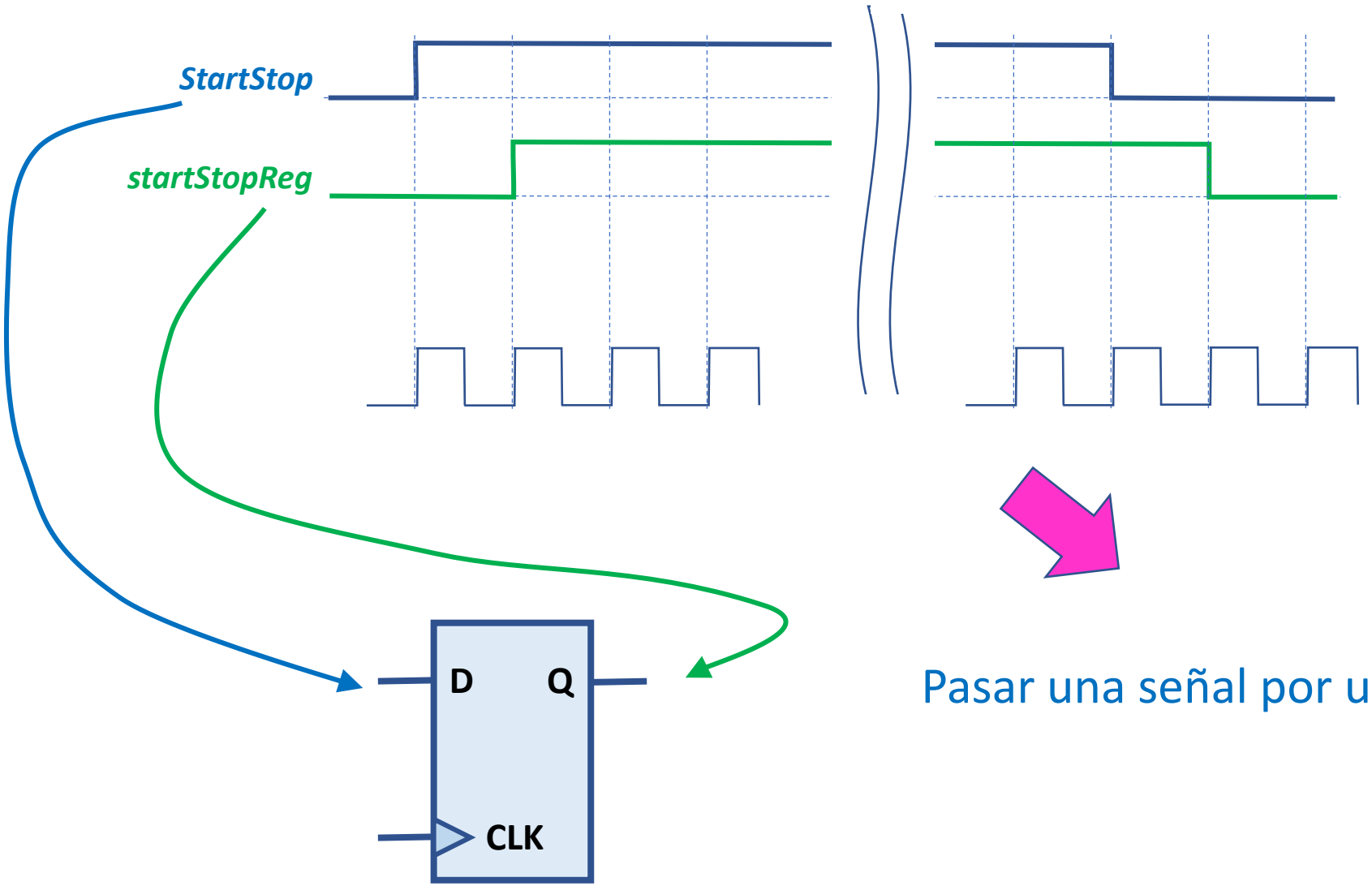


Easy-peasy, tenemos flip-flops que detectan flancos

Detectar un flanco (p.ej. de subida)

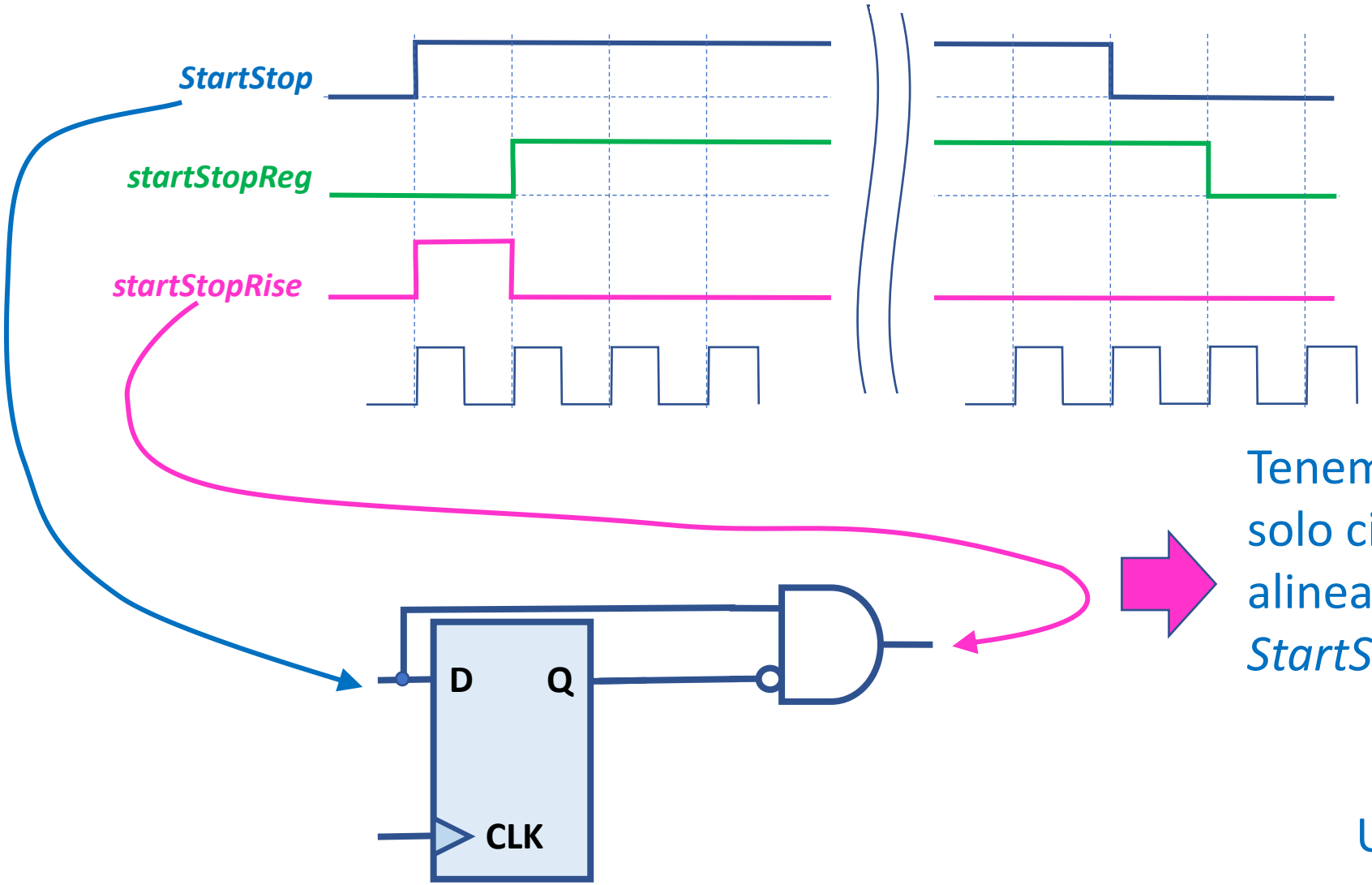


Detectar un flanco (p.ej. de subida) bien (I)



Pasar una señal por un FF es retrasarla un ciclo

Detectar un flanco (p.ej. de subida) bien (II)

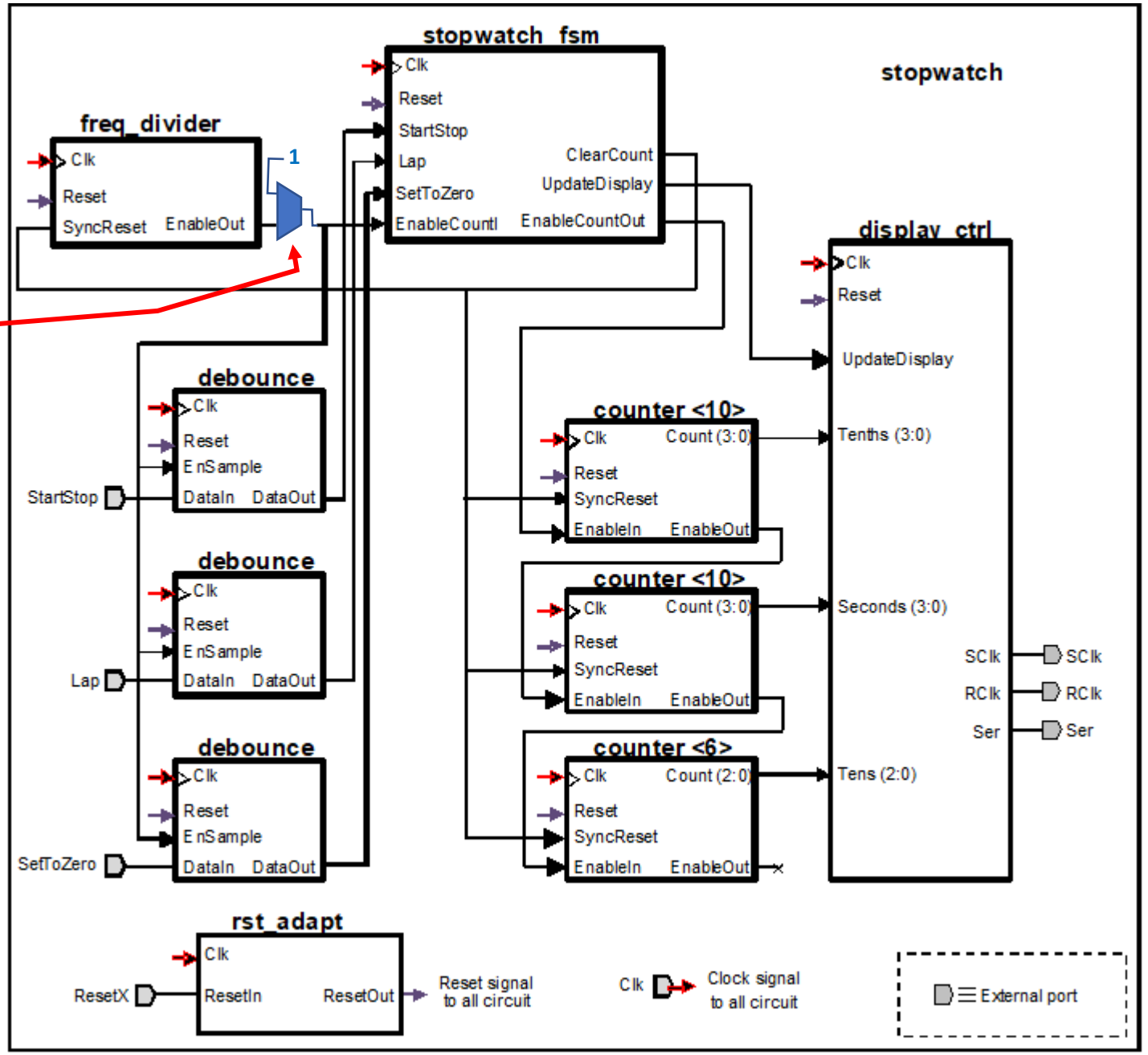


Tenemos una señal que ocurre un solo ciclo por pulsación del botón, alineada al flanco de subida de *StartStop*

Usar ésta en la FSM

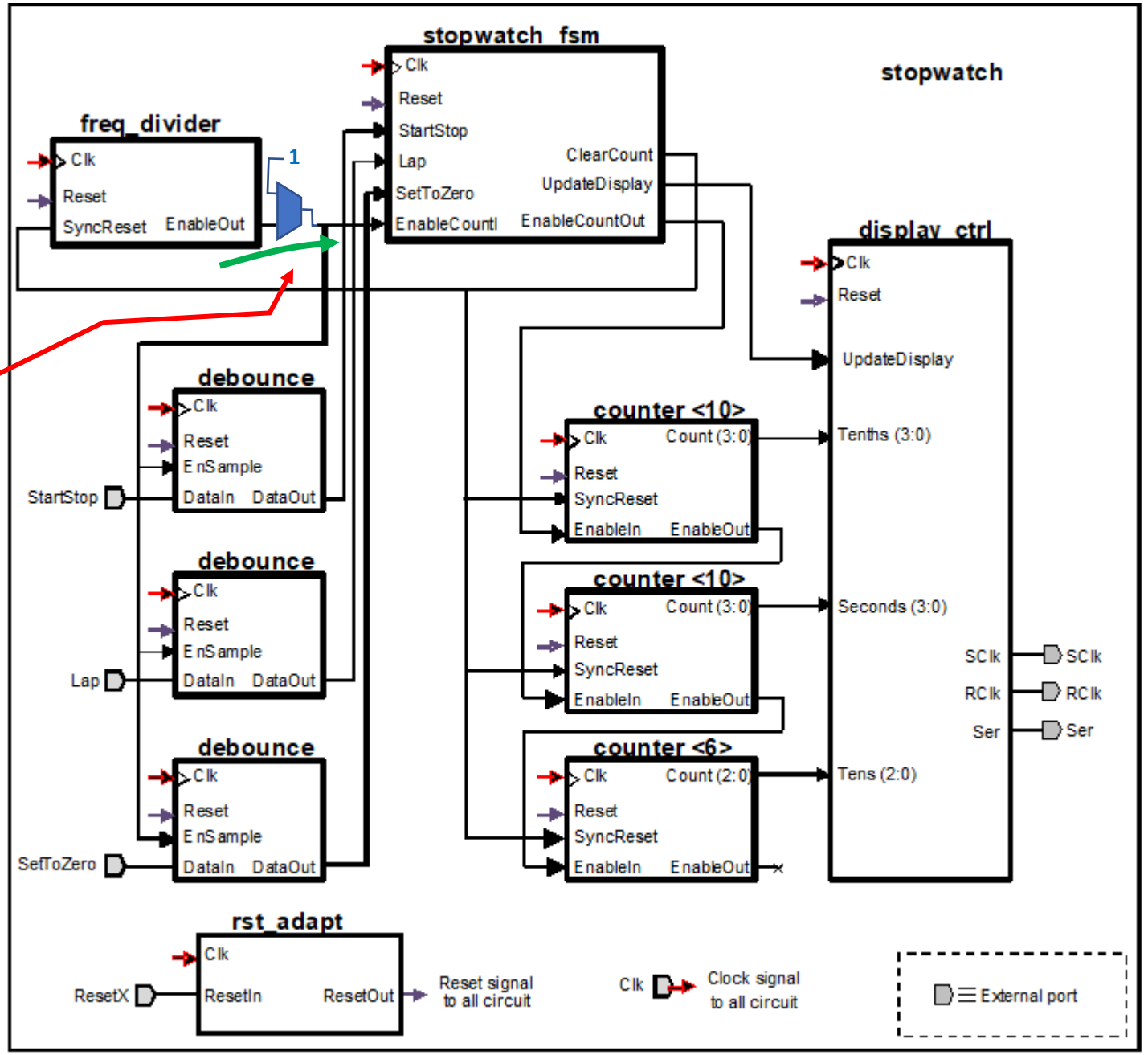
FAST_SIMULATION

- Truco para acelerar simulaciones mientras se depura el circuito.
- Añadimos multiplexor.



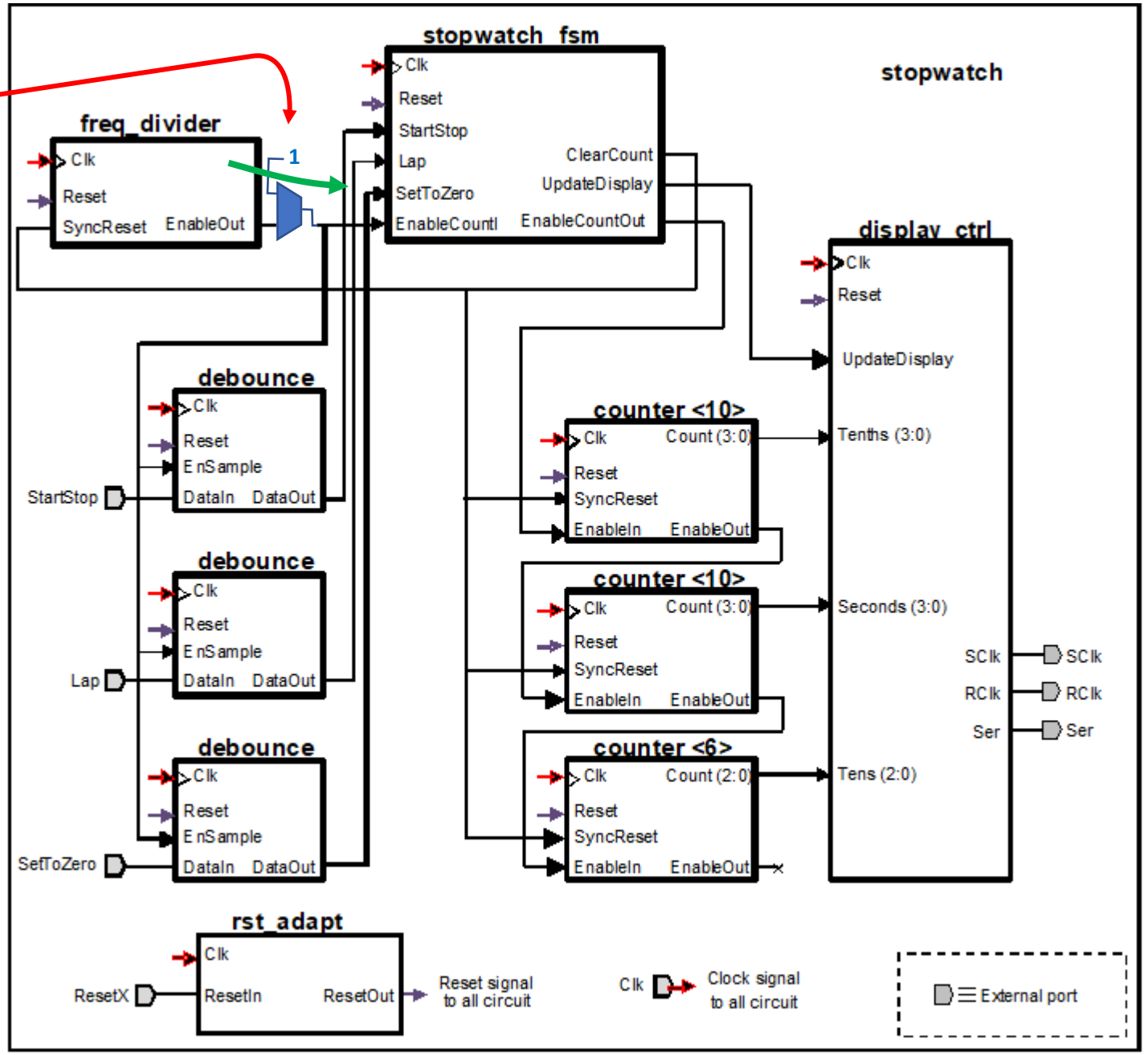
FAST_SIMULATION

- Truco para acelerar simulaciones mientras se depura el circuito.
- Añadimos multiplexor.
- En esa posición, funcionamiento normal.



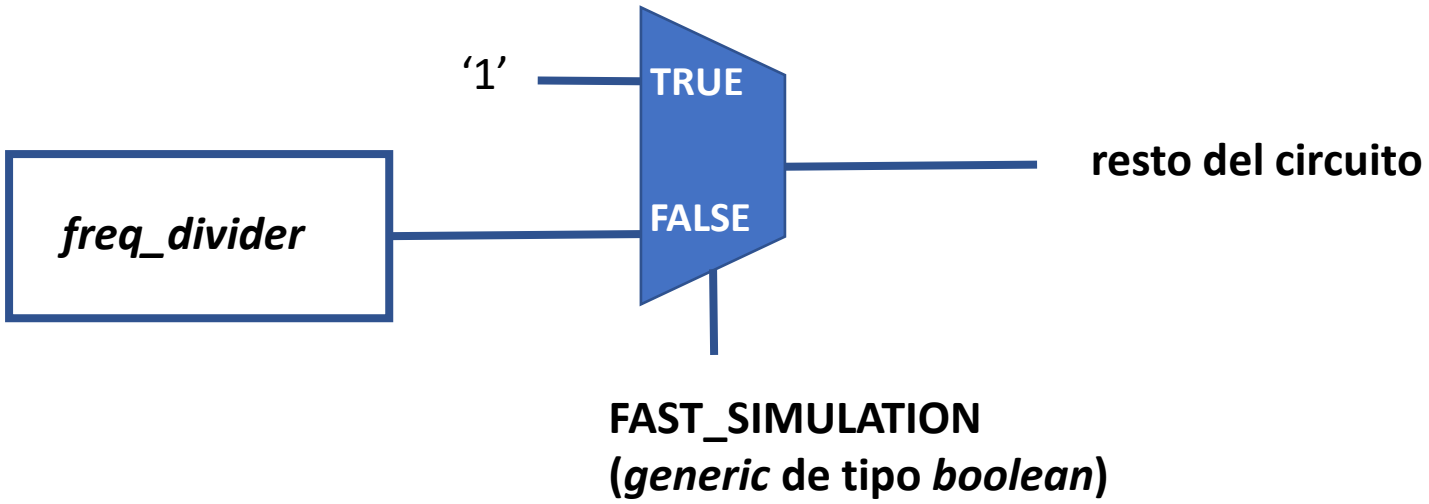
FAST_SIMULATION

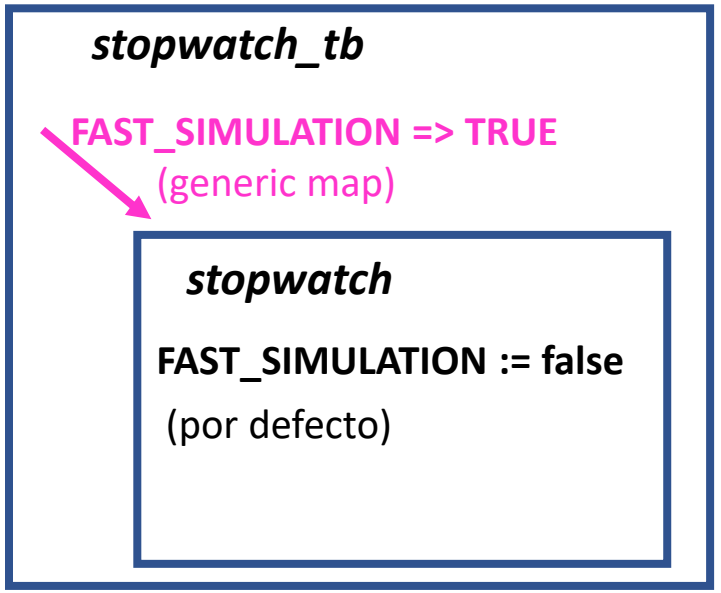
- Truco para acelerar simulaciones mientras se depura el circuito.
- Añadimos multiplexor.
- En esa posición, funcionamiento normal.
- En esa otra posición, el tren de pulsos que era la base de tiempos es ahora un ‘1’ fijo -> el contador de décimas de segundo se moverá con cada ciclo de reloj.



FAST_SIMULATION (II)

- Controlamos el multiplexor mediante un ***generic***.
  - Su valor por defecto pondrá al multiplexor en funcionamiento normal → Síntesis
  - Desde el testbench, un *generic map* podrá acelerar las simulaciones.





Testbench (I)

entity stopwatch_tb …

No tiene puertos ni *generics*

end …

architecture ...

Declaraciones:
- *component stopwatch* --> igual que *entity stopwatch*
- Constantes que el enunciado pide definir
  (periodo de reloj, tiempo de la fase principal de la simulación)
- Señales:
  - Las necesarias para estimular *stopwatch* y ver sus salidas
  - *endSimulation*, de tipo *boolean*

begin

Testbench (II)

architecture ...

...

begin

Instancia de *stopwatch*
- Con *generic map* para *FAST_SIMULATION* y señales conectadas

process para generar un reloj con periodo *CLK_PERIOD*
- En bucle continuo hasta que *endSimulation* sea *TRUE*
- Para con un *"wait;"*

process para el "programa principal" del test:
- Dar valor a entradas de *stopwatch* (no al reloj)
- Retirar el reset *tras* N periodos de reloj (*wait for*)
- Hacer un "pulsado" de *StartStop* de duración T_P
- Esperar el tiempo definido por la constante *SIM_TIME*
- Activar *endSimulation*
- *wait;*

N , T_P, SIM_TIME → ver enunciado

estímulos

control fin de la simulación

end ...