

VHDL 3: Diseño de circuitos síncronos

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hguzman@us.es

Acknowledgement to Fernando Muñoz,
Universidad de Sevilla

Contexto docente

BT01: Introducción a VHDL y su aplicación en FPGAs

- Tema 1: Estructura de un fichero VHDL
- Tema 2: Describiendo la funcionalidad
- Tema 3: Diseño de circuitos síncronos
- Tema 4: Simulación con testbenches

Conocimientos previos requeridos:

- Electrónica digital básica (puertas lógicas y biestables)
- Temas 1 y 2: Estructura de un fichero VHDL y Describiendo la funcionalidad

Objetivos de aprendizaje

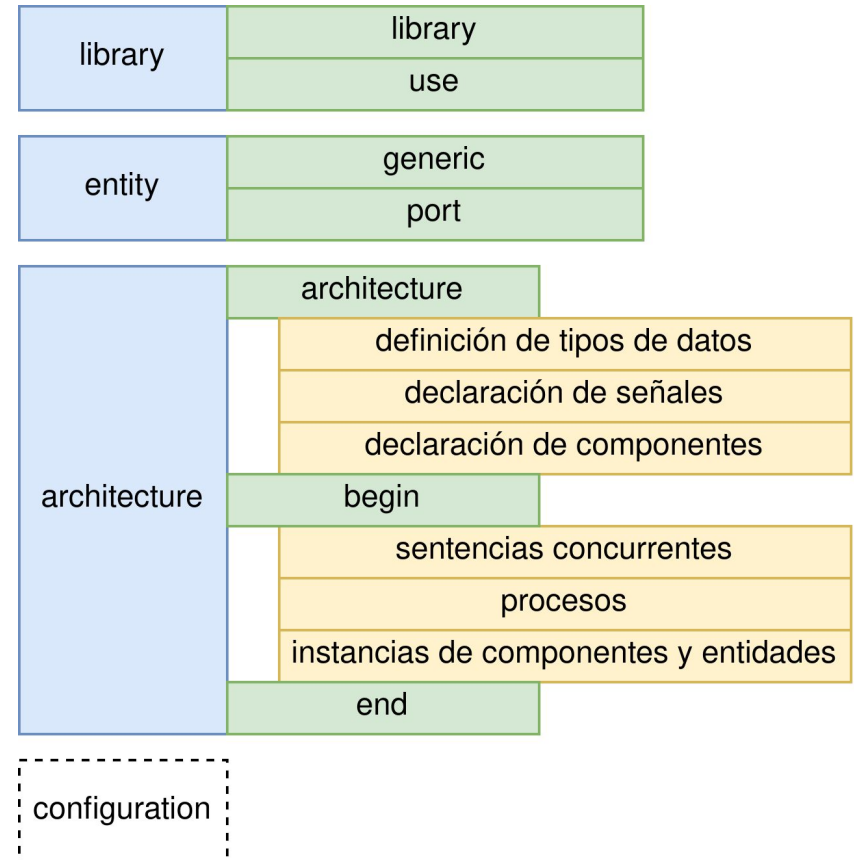
- Aprender cómo se describen circuitos síncronos en VHDL
- Comprender la diferencia entre procesos combinacionales y procesos síncronos
- Adquirir las herramientas necesarias para describir circuitos de cierta complejidad como máquinas de estados

Contenido

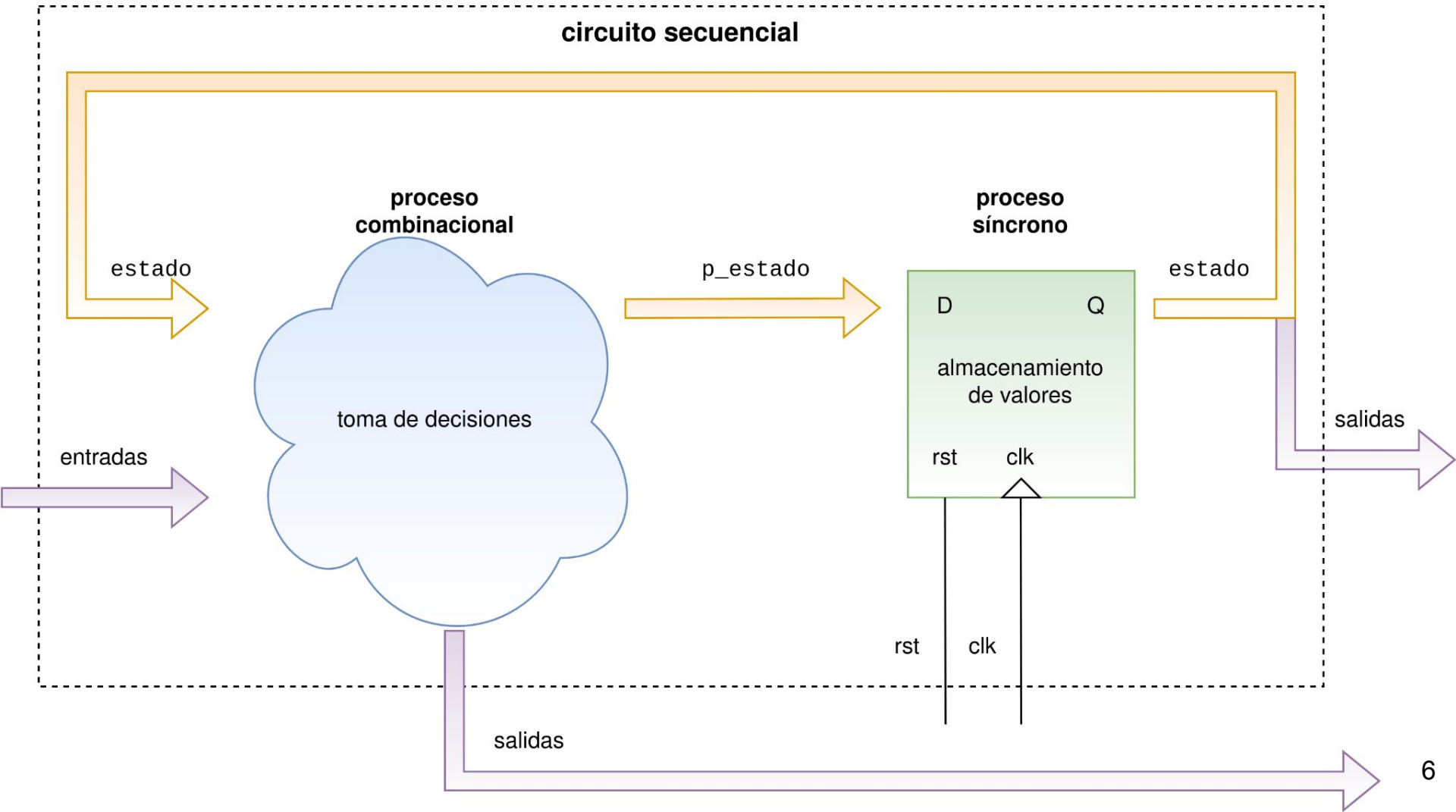
- Repaso
- Descripción en dos procesos
- Definición de tipos de datos
- Máquinas de estados
- Variables
- Atributos

Repaso

- **Library**
 - Inclusión de librerías y paquetes
- **Entity**
 - Descripción de caja negra
- **Architecture**
 - Descripción de la funcionalidad
- **Configuration**
 - Normalmente no se utiliza



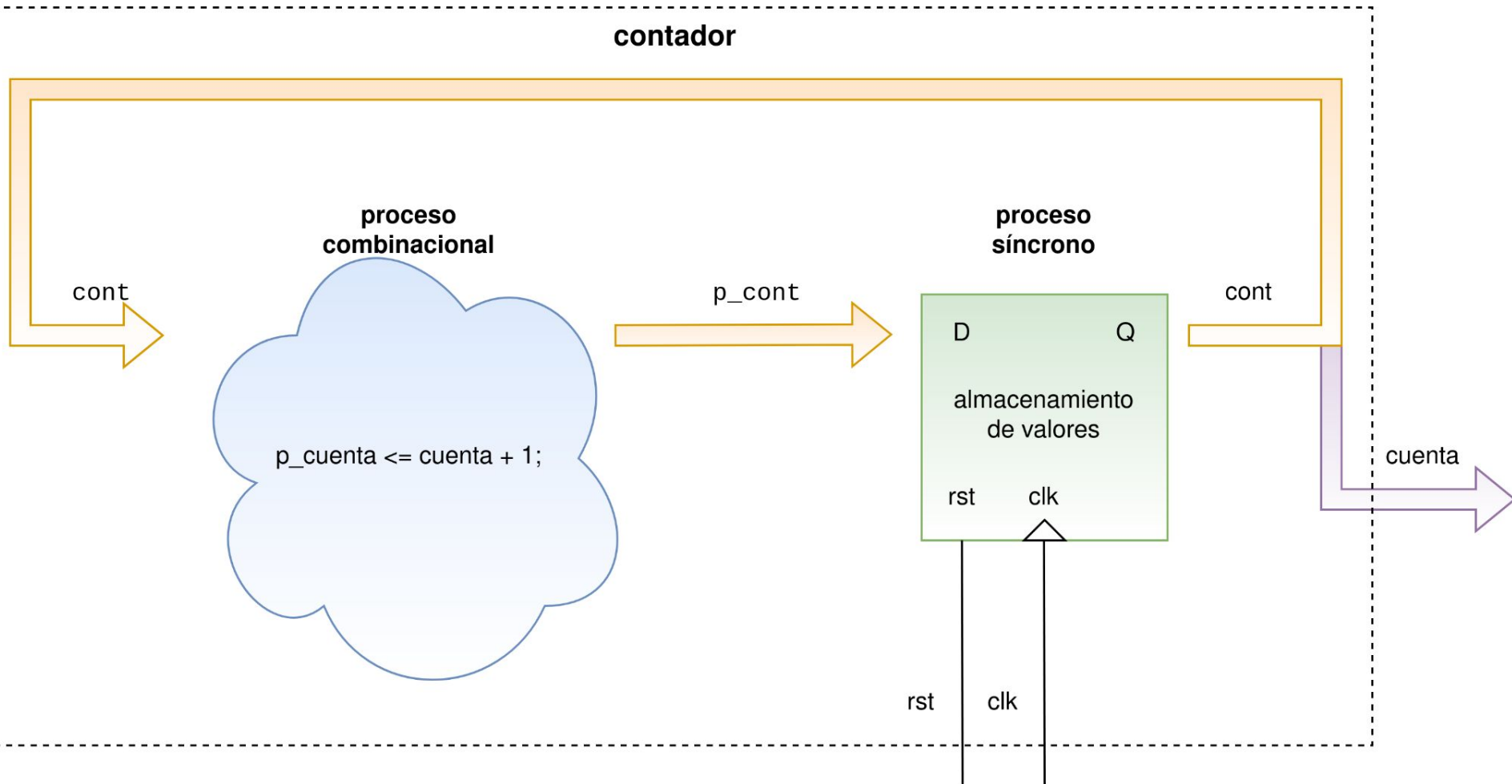
Combinacional + Síncrono



¿Cómo haríamos un contador?

- Un contador necesita saber almacenar el valor por el que va contando
- Ese será el *estado interno* del circuito
- La *toma de decisiones* es tan simple como decidir cuál es el *siguiente* valor de la cuenta

Planteamiento del contador



Codificamos el VHDL

```
architecture cont_arch of contador is
```

```
    signal cont, p_cont: unsigned(7 downto 0);
```

```
begin
```

```
    comb: process(cont)
```

```
    begin
```

```
        p_cont <= cont + 1;
```

```
    end process;
```

```
    cuenta <= cont;
```

```
    sync: process(clk, reset)
```

```
    begin
```

```
        if rst = '1' then
```

```
            cont <= (others => '0');
```

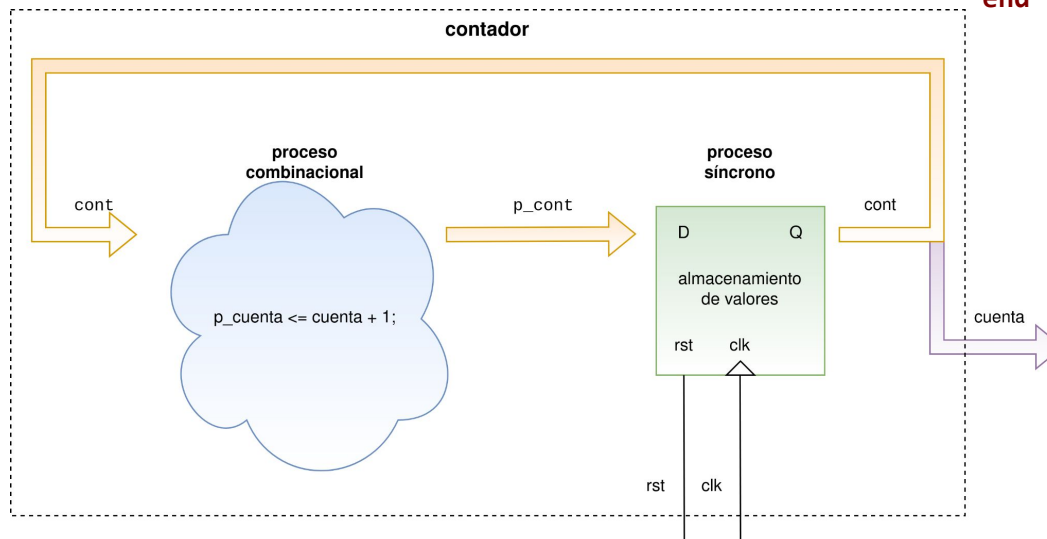
```
        elsif rising_edge(clk) then
```

```
            cont <= p_cont;
```

```
        end if;
```

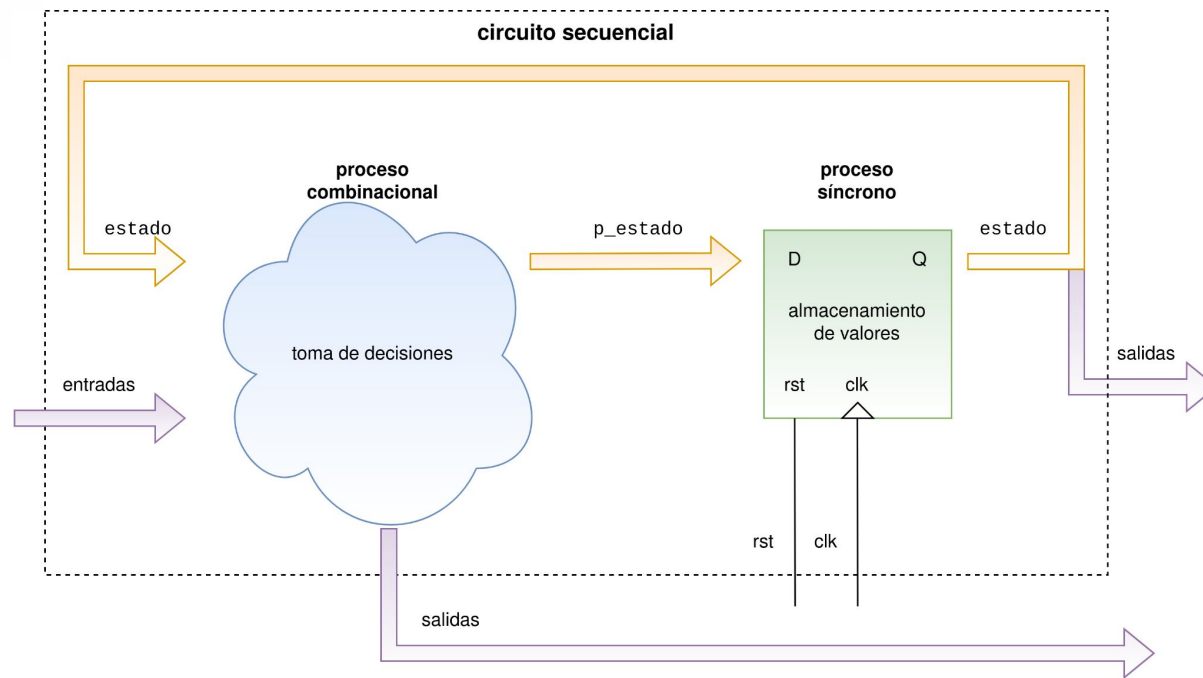
```
    end process;
```

```
end cont_arch;
```



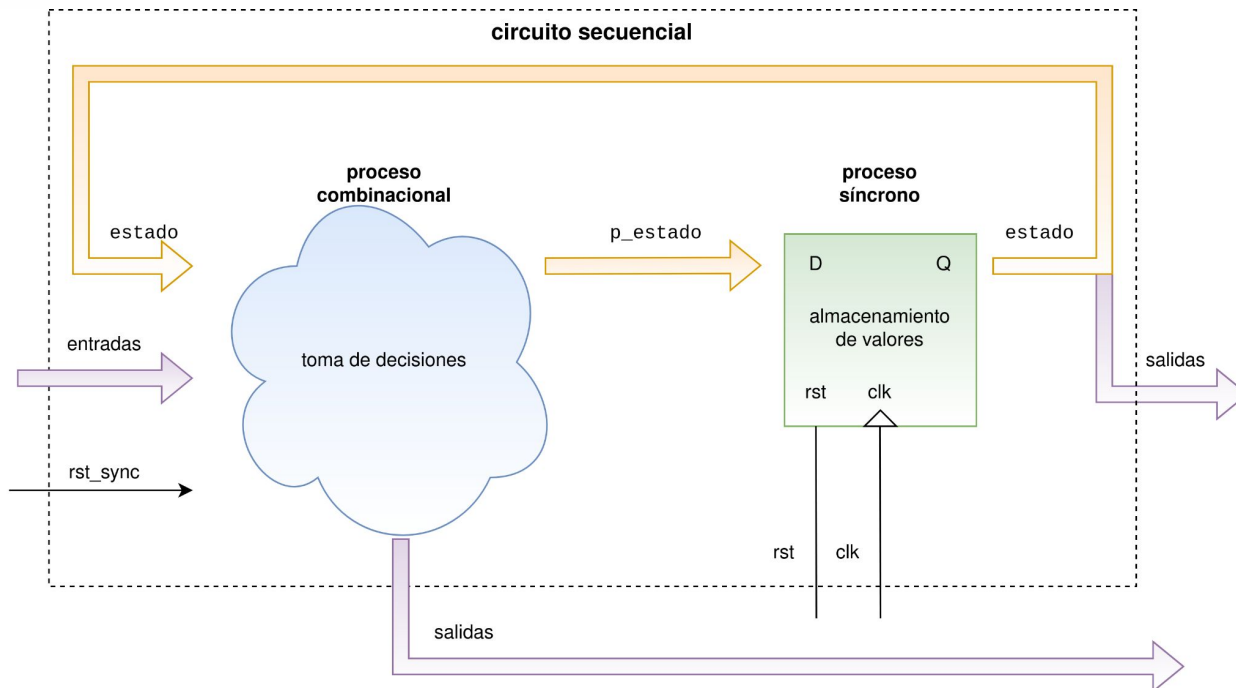
Reset asíncrono

- El reset asíncrono se define en el proceso síncrono
- El motivo es que este reset debe actuar de manera instantánea sobre el registro a resetear, no puede esperar al reloj



Reset síncrono

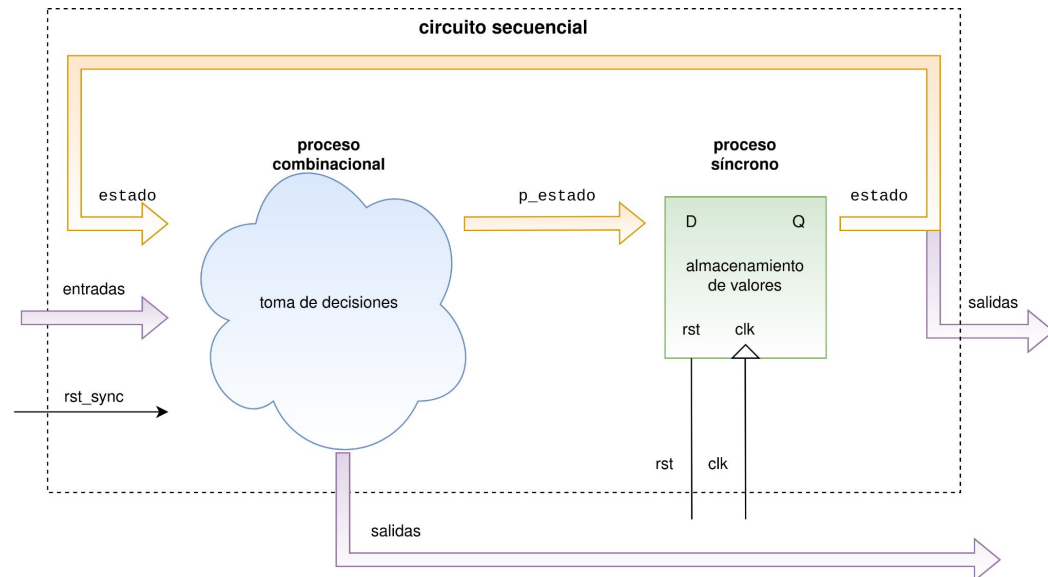
- El reset síncrono se define en el proceso combinacional
- El proceso combinacional asigna un valor a p_estado (por ejemplo poniendo todos sus bits a cero)
- El proceso combinacional (biestable(s)) traslada ese cambio a estado en el siguiente flanco activo de reloj



Reset síncrono

- Definido en el proceso combinacional:

```
comb: process(cont, rst_sync)
begin
  if (rst_sync = '1') then
    p_cont <= (others => '0');
  else
    p_cont <= cont + 1;
  end if;
end process;
```



Reset síncrono

- También es posible definirlo en el proceso síncrono
- Para ello tiene que actuar sólo si hay un flanco activo de reloj
- Ese proceso no sería sensible al reset

```
-- Reset asíncrono:
```

```
sync: process(clk, rst)
begin
  if rst = '1' then
    cont <= (others => '0');
  elsif rising_edge(clk) then
    cont <= p_cont;
  end if;
end process;
```



```
-- Reset síncrono:
```

```
sync: process(clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      cont <= (others => '0');
    else
      cont <= p_cont;
    end if;
  end if;
end process;
```

¿Dónde definimos los resets?

- Como regla general, es buena idea tomar un criterio y seguirlo siempre
- El reset asíncrono sólo puede definirse en el proceso síncrono
- Para el reset síncrono, lo normal es definirlo en el proceso combinacional
 - Eso sí, saber que puede hacerse otra manera nos ayuda a entender código de terceros
- No se deben poner ambos resets en el proceso síncrono
 - Porque las FPGAs no tienen biestables con dos resets a la vez
 - Los biestables sólo tienen un reset que se puede configurar como síncrono o asíncrono
 - Si queremos tener resets de los dos tipos tenemos que poner el reset síncrono en el proceso combinacional y el reset asíncrono en el proceso síncrono

Reset asíncrono vs reset síncrono

Reset asíncrono

- Sensible a glitches
- El uso más común es como **reset global del diseño**
- Utilizado una única vez al principio, cuando se programa la FPGA
- No se vuelve a utilizar durante el funcionamiento del circuito
- “Asynchronously asserted, synchronously deasserted”, pero es la FPGA la que se encarga de manera transparente al usuario

Reset síncrono

- Sincronizado con el reloj
- Por ello, es predecible y no es sensible a glitches
- Utilizado localmente, para poner a cero módulos o estados específicos
- De manera ordenada y sin problemas de temporización
- Puede utilizarse y se utiliza en cualquier momento del funcionamiento del circuito, no sólo en “tiempo cero”

Ejemplo: contador

Hagamos un contador:

- De tamaño genérico (**N** bits)
- Con entrada de habilitación (**ena**)
- Con entrada de sentido de la cuenta (**updown**)
 - Si updown = '1', cuenta hacia arriba
 - Si updown = '0', cuenta hacia abajo

Definición de tipos de datos

- VHDL permite al usuario crear nuevos tipos de datos
- Una vez definidos, cualquier objeto puede ser creado con ese nuevo tipo
- Recordemos que, al ser de tipado duro, VHDL no nos permite realizar asignaciones entre objetos de tipos diferentes

```
type nombre_del_tipo is definicion_del_tipo;
```

Definición de tipos de datos

type nombre_del_tipo **is** definicion_del_tipo;

- Tres formas:
 - Tipo enumerado
 - Array
 - Record

Tipos enumerados

- Consisten en una lista de valores que podrá tomar el objeto (señal, variable ...)

```
type t_enumerado is (valor1, valor2, valor3, valor4);
```

```
type estado is (reposo, lento, medio, rapido);
```

- Normalmente se utilizan para implementar máquinas de estado
- VHDL le asigna un número a cada posible valor, por orden de declaración
 - Por ejemplo, podemos decir que rápido > reposo
 - Pero no lo hagáis! Resulta en código ofuscado, más difícil de entender y mantener, en el que un cambio en el orden de la definición puede cambiarte el funcionamiento del diseño
- std_ulogic de ieee.std_logic_1164 es un tipo enumerado

```
TYPE std_ulogic IS ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-' );
```

Arrays

- Vector de elementos

- Constrained:

```
type ram_type is array (255 downto 0) of std_logic_vector (15 downto 0);  
[...]  
signal ram: ram_t;
```

- Unconstrained: se restringe al declarar el objeto

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;  
type SIGNED is array (NATURAL range <>) of STD_LOGIC;  
[...]  
signal my_unsigned: unsigned(range 7 downto 0);
```

Records

- Compuesto por otros tipos de datos
- Análogo al struct de C

```
type complex10 is
  record
    re : signed(9 downto 0);
    im : signed(9 downto 0);
  end record;
```

```
type fsm_context is
  record
    state      : fsm_state;
    index      : integer range 0 to 8;
    data       : std_ulogic_vector(63 downto 0);
    has_data   : std_ulogic;
  end record;
```

Subtypes: type + restriction

```
-- Antes del begin de la architecture
```

```
type dia_mes is integer range 1 to 31;    -- ERROR: Illegal
```

```
type otro_tipo is integer range 1 to 31;  -- ERROR: Illegal
```

```
subtype dia_mes is integer range 1 to 31; -- Correcto
```

```
subtype otro_tipo is integer range 1 to 31; -- Correcto
```

```
signal dia_hoy, dia_manyana: dia_mes;
```

```
signal otro_dato: otro_tipo;
```

```
begin
```

```
-- Después del begin de la architecture
```

```
dia_manyana <= dia_hoy + 1;  -- OK! Mismo tipo de dato
```

```
otro_dato   <= dia_hoy;      -- OK! Distintos subtype del mismo type
```

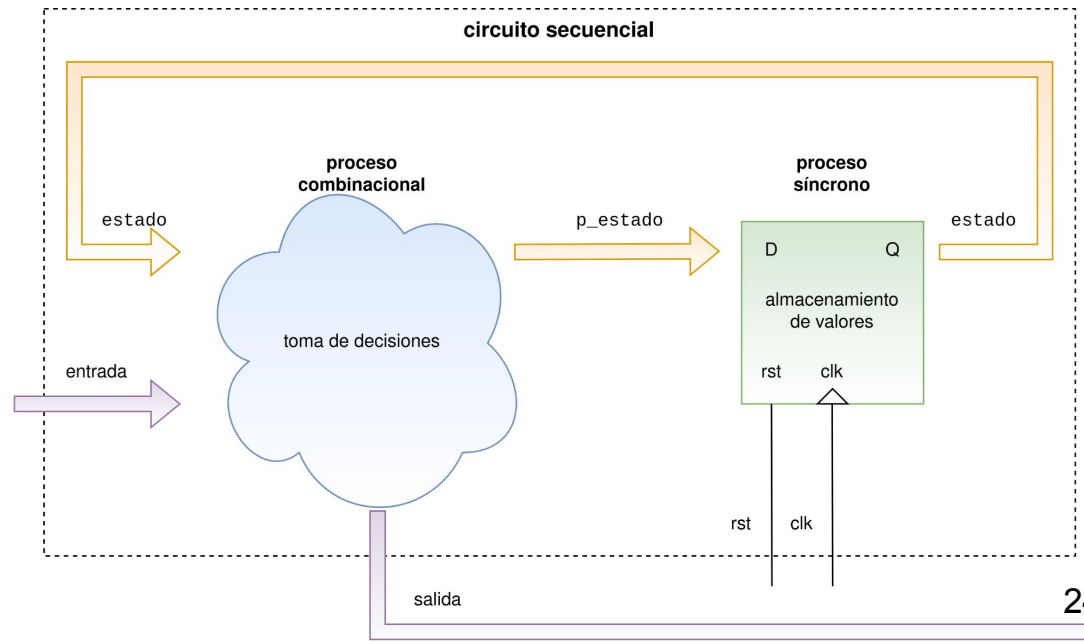
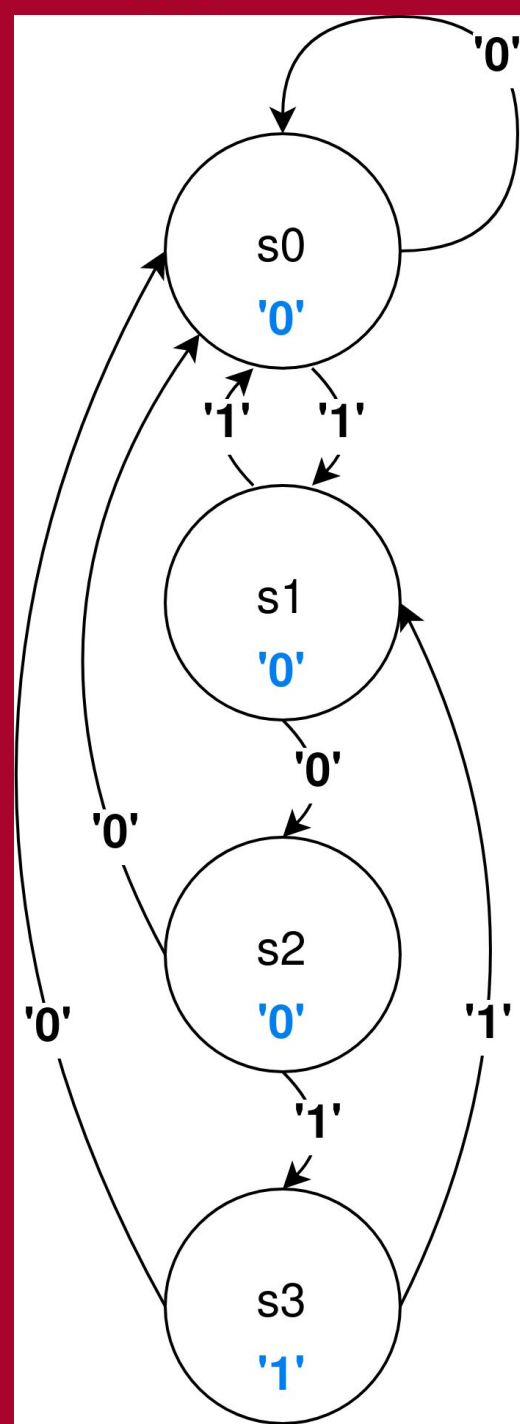
Máquinas de estados

FSM, por sus siglas en inglés (Finite State Machine)

- Se define un tipo enumerado con los estados posibles
- El proceso síncrono almacenará:
 - El estado de la FSM
 - Cualquier otro estado interno del circuito que deba almacenarse
- El proceso combinacional debe decidir:
 - Cuál será el siguiente estado de la FSM
 - Qué valor tomarán las salidas

Ejemplo FSM

Es una máquina de Moore porque la salida (en **azul**) sólo depende del estado y no de la entrada (en **negro**)



FSM: tipos y proceso síncrono

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

entity fsm is
  port (
    rst      : in  std_ulogic;
    clk      : in  std_ulogic;
    entrada  : in  std_ulogic;
    salida   : out std_ulogic
  );
end fsm;
```

```
architecture arch of fsm is
```

```
  type t_estado is (s0, s1, s2, s3);
  signal estado, p_estado: t_estado;
  -- (En inglés las llamaríamos
  -- state y n_state)
```

```
begin
```

```
  sync: process(rst, clk)
  begin
    if rst = '1' then
      estado <= s0;
    elsif rising_edge(clk) then
      estado <= p_estado;
    end if;
  end process;
```

FSM: proceso combinacional

```
comb: process(estado, entrada)
```

```
begin
```

```
  case estado is
```

```
    when s0 =>
```

```
      salida <= '0';
```

```
      if entrada = '0' then
```

```
        p_estado <= s0;
```

```
      else
```

```
        p_estado <= s1;
```

```
      end if;
```

```
    when s1 =>
```

```
      salida <= '0';
```

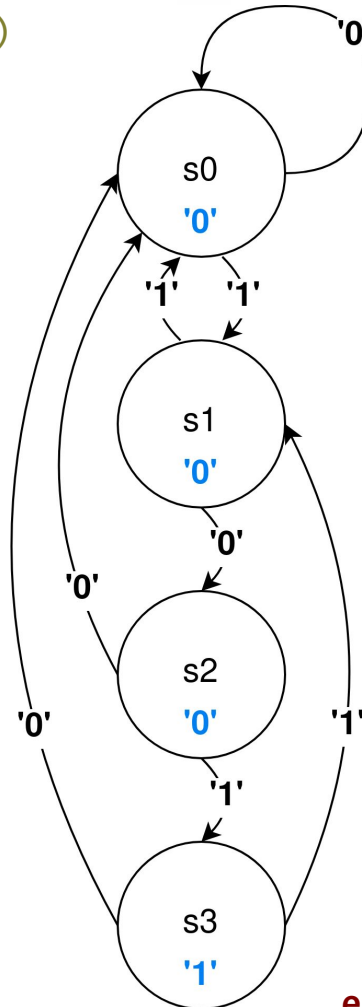
```
      if entrada = '1' then
```

```
        p_estado <= s0;
```

```
      else
```

```
        p_estado <= s2;
```

```
      end if;
```



```
  when s2 =>
```

```
    salida <= '0';
```

```
    if entrada = '0' then
```

```
      p_estado <= s0;
```

```
    else
```

```
      p_estado <= s3;
```

```
    end if;
```

```
  when s3 =>
```

```
    salida <= '1';
```

```
    if entrada = '0' then
```

```
      p_estado <= s0;
```

```
    else
```

```
      p_estado <= s1;
```

```
    end if;
```

```
  end case;
```

```
end process;
```

```
end arch;
```

Variables

- No son específicas de diseños síncronos, de hecho suelen aparecer en los procesos combinacionales y también en testbenches
- Las variables son locales a un **process**, es decir que no existen fuera de él
 - Sólo existen dentro del **process** en el que han sido definidas
- Las asignaciones a variables se hacen con **:=** y no con **<=**
- Las variables se actualizan inmediatamente tras ser asignadas
 - No como los signals que se actualizan al llegar al **end process** o a sentencias **wait**

Variables en código sintetizable

- A pesar de lo que su nombre sugiere, en síntesis **no se utilizan para guardar valores!**
 - Si quieres almacenar un valor, utiliza un proceso síncrono
- Se utilizan más bien para realizar cálculos intermedios
- Veamos un ejemplo

Ejemplo: variable en código sintetizable

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

entity mult is
  port (
    op1 : in  unsigned(7 downto 0);
    op2 : in  unsigned(7 downto 0);
    res : out unsigned(11 downto 0)
  );
end mult;

architecture arch of mult is

begin
```

```
  process (op1, op2)
    variable mult16 : unsigned(15 downto 0);
  begin
    mult16 := op1 * op2;
    res <= mult16(15 downto 4);
  end process;

end arch;
```

Queremos multiplicar 2 números de 8 bits, descartando los 4 bits menos significativos del resultado

Variables en testbenches

- En código específico para simulación, sí podemos hacer prácticamente lo que queramos con variables
- Las metodologías de verificación modernas y otras librerías hacen un fuerte uso de las variables
- Aquí sí se parecen más a lo que entendemos por variable en un lenguaje de programación
- Siguen siendo locales a los **process**

Atributos

- Un atributo es una característica asociada a un elemento (tipo de dato, señal, entidad, ...) que proporciona información adicional.
- Atributo $\neq$ Valor
- Un objeto tiene un solo valor y puede tener múltiples atributos.
- VHDL proporciona una serie de atributos predefinidos.

Ejemplo

```
signal mysig: std_ulogic_vector(7 downto 2);
```

Atributo	Valor que toma
mysig'LEFT	7
mysig'RIGHT	2
mysig'LOW	2
mysig'HIGH	7
mysig'ASCENDING	false (ya que el rango es “downto”)
mysig'RANGE	7 downto 2
mysig'REVERSE_RANGE	2 to 7

Atributos predefinidos de tipos

Atributo	Significado
<code>typ'BASE</code>	Tipo base de typ . Sólo se permite como prefijo para otro atributo
<code>typ'LEFT</code>	Límite izquierdo de typ
<code>typ'RIGHT</code>	Límite derecho de typ
<code>typ'LOW</code>	Límite inferior de typ
<code>typ'HIGH</code>	Límite superior de typ
<code>typ'ASCENDING</code>	true si typ tiene un rango “LSB to MSB”, false si el rango es “MSB downto LSB”
<code>typ'IMAGE(X)</code>	Representación como string del valor de X
<code>typ'VALUE(X)</code>	El valor del tipo typ cuya representación como string es X
<code>typ'POS(X)</code>	Posición (índice) de X en typ
<code>typ'VAL(X)</code>	Valor que está en la posición X en typ
<code>typ'SUCC(X)</code>	Sucesor: valor en posición_de_X + 1 ($T'VAL(T'POS(X)+1)$)
<code>typ'PRED(X)</code>	Predecesor: valor en posición_de_X - 1 ($T'VAL(T'POS(X)-1)$)
<code>typ'LEFTOF(X)</code>	Valor a la izquierda de X en typ
<code>typ'RIGHTOF(X)</code>	Valor a la derecha de X en typ

Atributos predefinidos de arrays

Atributo	Significado
<code>arr'LEFT</code>	Valor a la izquierda del intervalo de índices
<code>arr'RIGHT</code>	Valor a la derecha del intervalo de índices
<code>arr'LOW</code>	Valor máximo del intervalo de índices
<code>arr'HIGH</code>	Valor mínimo del intervalo de índices
<code>arr'RANGE</code>	Intervalo completo de índices
<code>arr'REVERSE_RANGE</code>	Intervalo de índices en sentido inverso
<code>arr'LENGTH</code>	Longitud del intervalo de índices (número de elementos en el array)
<code>arr'ASCENDING</code>	true si el rango es de tipo “LSB to MSB”, false si es de tipo “MSB downto LSB”

Atributos predefinidos de arrays

Atributo	Significado
arr'LEFT[(N)]	Mismos que los atributos de los arrays, pero para el elemento N del array arr . Se utilizan en arrays multidimensionales.
arr'RIGHT[(N)]	
arr'LOW[(N)]	
arr'HIGH[(N)]	
arr'RANGE[(N)]	
arr'REVERSE_RANGE[(N)]	
arr'LENGTH[(N)]	
arr'ASCENDING[(N)]	

Atributos predefinidos de **signal**

- Generalmente no se usan en síntesis!
- Salvo **'EVENT**, y **'LAST_VALUE**, que son usados internamente por `rising_edge()`

Atributo	Significado
sig' DELAYED[(T)]	Valor que tenía sig hace T tiempo
sig' STABLE[(T)]	true si no ha ocurrido ningún evento (cambio) en sig en el último T tiempo
sig' QUIET[(T)]	true si no ha ocurrido ninguna transacción (asignación) en sig en el último T tiempo
sig' TRANSACTION	Tipo bit , que cambia cada vez que hay una transacción en sig
sig' EVENT	true si ha ocurrido un evento en sig en el instante actual
sig' ACTIVE	true si ha ocurrido una transacción en sig en el instante actual
sig' LAST_EVENT	Tiempo desde el último evento en sig
sig' LAST_ACTIVE	Tiempo desde la última transacción en sig
sig' LAST_VALUE	Valor de sig antes del último evento
sig' DRIVING	false si nada está asignando valor en el process actual a sig
sig' DRIVING_VALUE	Valor que se está asignando a sig en el process actual

Atributos predefinidos generales

- Muy útiles en simulación para imprimir mensajes de ayuda a la depuración

Atributo	Significado	Ejemplo
<code>element'SIMPLE_NAME</code>	Representación como string del nombre de <code>element</code>	<code>switch_predictor</code>
<code>element'PATH_NAME</code>	Representación como string del nombre jerárquico de <code>element</code> , sin incluir los nombres de las entidades instanciadas	<code>:tb_switch_top:predictor_inst:</code>
<code>element'INSTANCE_NAME</code>	Representación como string del nombre jerárquico de <code>element</code> , incluyendo los nombres de las entidades instanciadas y sus arquitecturas	<code>:tb_switch_top(behavioral):predictor_inst@switch_predictor(switch_predictor_arch):</code>

Conclusiones

- Diseñar circuitos secuenciales es saber diseñar con dos procesos
- El proceso combinacional:
 - Lee **entradas** y **estado**
 - Escribe **salidas** y **p_estado**
 - Es sensible a todas sus entradas
- El proceso síncrono:
 - Lee **rst**, **clk**, **p_estado**
 - Escribe **estado**
 - Es sensible a **rst** (opcionalmente) y **clk** (siempre)
 - NO es sensible a **p_estado** / **n_state**

Conclusiones

- El estado de un circuito pueden ser varias señales
 - Se actualizan todas en el mismo `if rising_edge(clk) ...`
 - Se inicializan todas en el mismo `if rst = '1' ...`
- En inglés en lugar de `p_` se utiliza `n_` para indicar el próximo valor

Conclusiones

- En síntesis, ningún proceso lee donde escribe!!
 - Si en el mismo proceso leemos y escribimos la misma señal, lo estamos haciendo mal
- Si una salida es igual a un estado interno, conectamos el estado interno a la salida utilizando una asignación concurrente

Bibliografía

- Brian Mealy, Fabrizio Tappero, [*Free Range VHDL*](#). Free Range Factory, 2018
- *The VHDL Golden Reference Guide*. Doulos, 1995
- Ricardo Jasinski, *Effective Coding with VHDL: principles and best practice*. The MIT Press, 2016
- Jiri Gaisler, “A Structured VHDL Design Method” ([texto](#) y [transparencias](#))

Resultados de aprendizaje

- Saber estructurar un diseño VHDL, dividiendo la funcionalidad en procesos combinacionales y síncronos
- Conocer cómo se definen tipos enumerados en VHDL
- Ser capaz de describir máquinas de estados en VHDL
- Entender para qué pueden ser útiles las variables en VHDL
- Conocer las diferencias entre resets síncronos y asíncronos, así como saber implementarlos y reconocerlos en VHDL
- Conocer los atributos más comunes en VHDL