

VHDL

Hortensia Mecha López

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

ESQUEMA GENERAL

Entidad

```
entity identificador_entidad is  
[generic(lista de genéricos);]  
[port (lista_puertos);]  
end [entity] [identificador];
```

```
    entity counter is  
    generic( n : integer := 8 );  
    port(  
        clk, rst, ld, inc, dec : in std_logic;  
        din : in std_logic_vector( n-1 downto 0 );  
        tca, tcd : out std_logic;  
        dout : out std_logic_vector( n-1 downto 0 ) );  
    end counter;
```

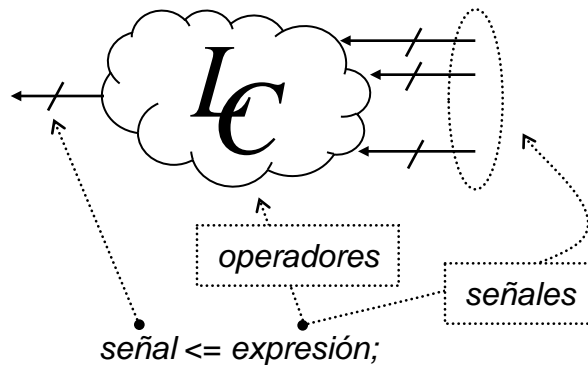
Arquitectura

```
architecture identificador of identificador_entidad is  
[declaraciones]  
begin  
[sentencias concurrentes]  
end [architecture] [identificador];
```

implementación de asignaciones concurrentes (i)

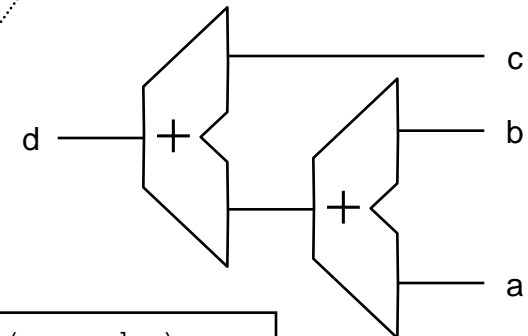
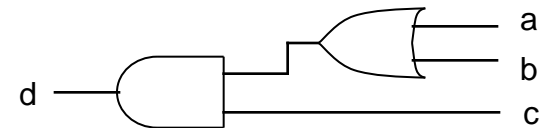
☒ Toda **asignación de señal** (o su proceso equivalente) se implementa como un bloque de lógica combinacional:

- Con un único puerto de salida (que puede ser vectorial en modelos de nivel RT).
- Con tantos puertos de entrada como señales diferentes aparezcan en la expresión.
- Con una funcionalidad especificada por los operadores que forman la expresión.



Una posible implementación de la sentencia ya que la implementación definitiva siempre la decide la herramienta

$d \leq (a \text{ or } b) \text{ and } c$

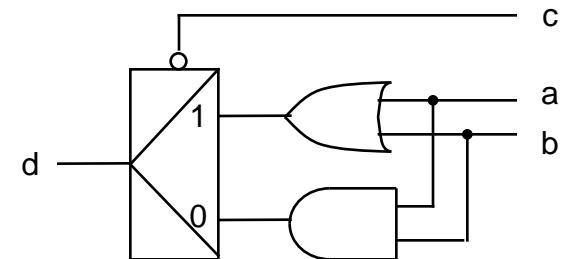
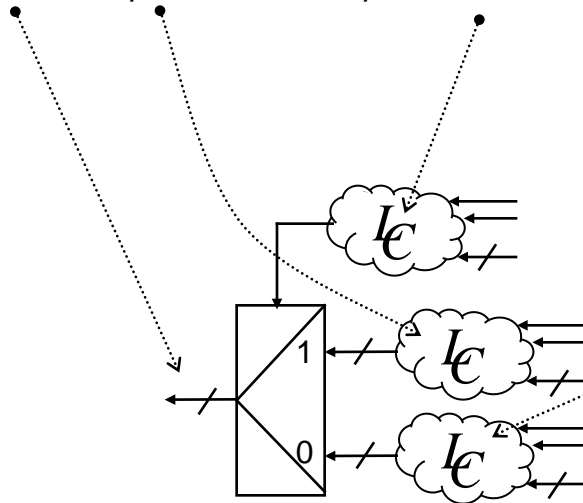


$d \leq (a + b) + c$

implementación de asignaciones concurrentes (ii)

- ☒ Toda **asignación condicional de señal** (o su proceso equivalente) se implementa como un bloque de lógica combinacional:
- Con un único puerto de salida (que puede ser vectorial en modelos de nivel RT).
 - Con tantos puertos de entrada como señales diferentes aparezcan en el lado derecho de la asignación (independientemente de la expresión en la que ocurran).
 - Con un comportamiento que se corresponde con el de un multiplexor 2 a 1 cuyas 3 entradas están conectadas a las salidas de 3 bloques combinacionales. La funcionalidad de dichos bloques queda especificada por los operadores que forman cada una de las 3 expresiones de la sentencia.

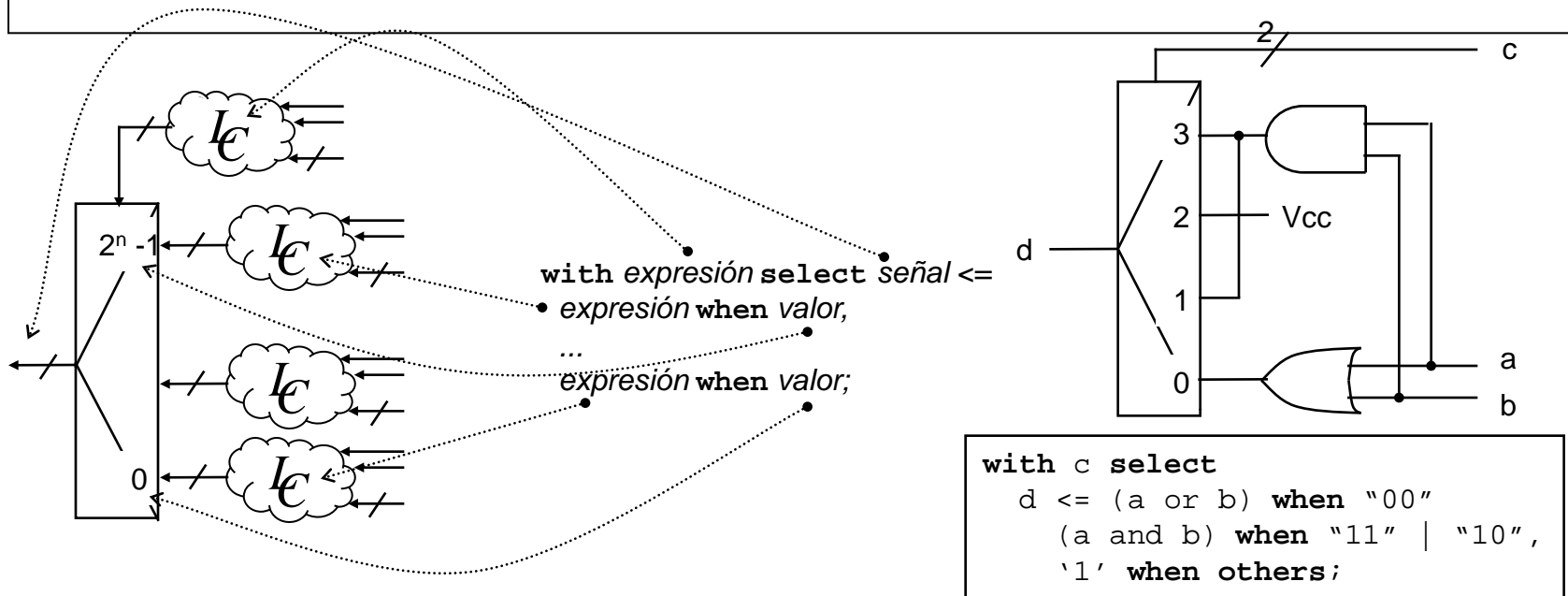
señal <= expresión when expresión_booleana else expresión;



`d <= (a or b) when (c = '0') else (a and b);`

implementación de asignaciones concurrentes (iii)

- ☒ Toda **asignación selectiva de señal** (o su proceso equivalente) se implementa como un bloque de lógica combinacional:
- Con un único puerto de salida (que puede ser vectorial en modelos de nivel RT).
 - Con tantos puertos de entrada como señales diferentes aparezcan en el lado derecho de la asignación (independientemente de la expresión en la que ocurran).
 - Con un comportamiento que se corresponde con el de un multiplexor 2^n a 1 cuyas $2^n + 1$ entradas (2^n son de datos y 1 de control) están conectadas a las salidas de $2^n + 1$ bloques combinacionales. La funcionalidad de dichos bloques queda especificada por los operadores que forman cada una de las $2^n + 1$ expresiones de la sentencia. Siendo n el número de bits con los que se codifica la selección entre 2^n expresiones de datos distintas.



implementación de procesos (i)

- ☒ Un proceso especifica **HW combinacional** si todas las señales o variables escritas dentro del mismo se asignan al menos una vez bajo cualquier condición de ejecución.
- ☒ Por el contrario un proceso especifica **HW secuencial** si las señales o variables escritas dentro del mismo no se asignan por defecto, sólo bajo ciertas condiciones de ejecución
 - Típicamente tras la detección de un nivel o un evento en cierta señal

```

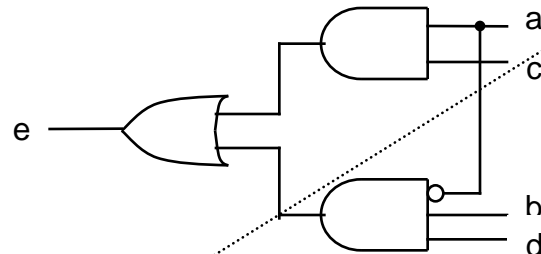
process( a, b, c, d )
begin
  if (a = '1') then
    e <= c;
  elsif (b = '1') then
    e <= d;
  else
    e <= '0';
  end if;
end process;

```

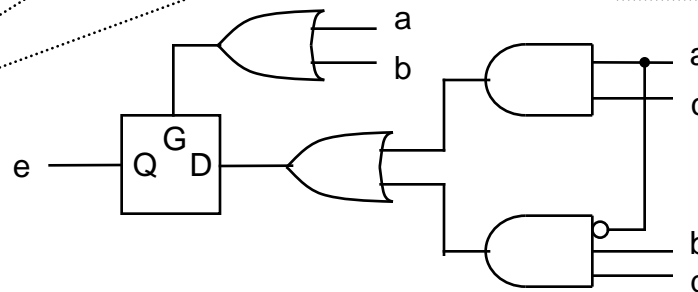
```

process( a, b, c, d )
begin
  if (a = '1') then
    e <= c;
  elsif (b = '1' ) then
    e <= d;
  end if;
end process;

```



El tipo de expresiones utilizadas (de nivel o de flanco) determinan el tipo de HW secuencial

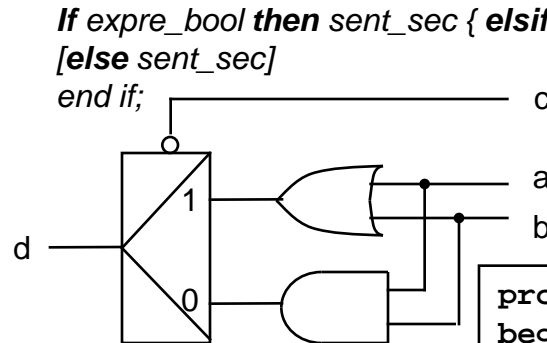


cualquier sentencia *if* o *case* incompletamente especificada implica la necesidad de almacenar los valores no modificados

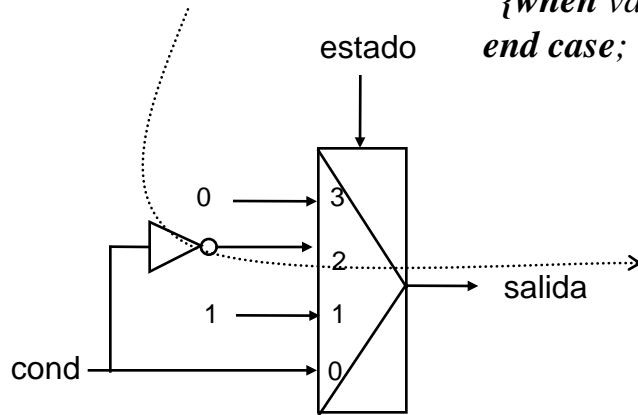
Implementación de procesos (ii)

⊗ Selección condicional

```
process( a, b, c )
begin
  if (c = '0') then
    d <= a or b;
  else
    d <= a and b;
  end if;
end process;
```



⊗ Selección múltiple



case *expresión* **is**
 when *valor* => *sentencias_secuenciales*;
 {*when* *valor* => *sentencias_secuenciales*;}
end case;

```
process( estado, cond )
begin
  case estado is
    when '0' =>
      if (cond = '0') then
        salida <= '0';
      else salida <= '1';
      end if;
    when '1' =>
      salida <= '1';
    when '2' =>
      if (cond = '0') then
        salida <= '1';
      else salida <= '0';
      end if;
    when '3' =>
      salida <= '0';
  end case;
end process;
```

implementación de procesos (iii): ejemplos

Latch tipo SR

```
process( set, rst )
begin
  if (set = '1') then
    q <= '1';
  elsif (rst = '1') then
    q <= '0';
  end if;
end process;
```

Latch tipo D

```
process( gate, d )
begin
  if (gate = '1') then
    q <= d;
  end if;
end process;
```

Latch tipo D con *reset* asíncrono

```
process( gate, set, rst, d )
begin
  if (rst = '1') then
    q <= '0';
  elsif (gate = '1') then
    q <= d;
  end if;
end process;
```

Flip-flop tipo D

```
process( clk )
begin
  if (clk'event and clk = '1') then
    q <= d;
  end if;
end process;
```

Flip-flop tipo D con reset síncrono

```
process( clk, rst )
begin
  if (clk'event and clk = '1') then
    if (rst = '1') then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

Flip-flop tipo D con reset asíncrono

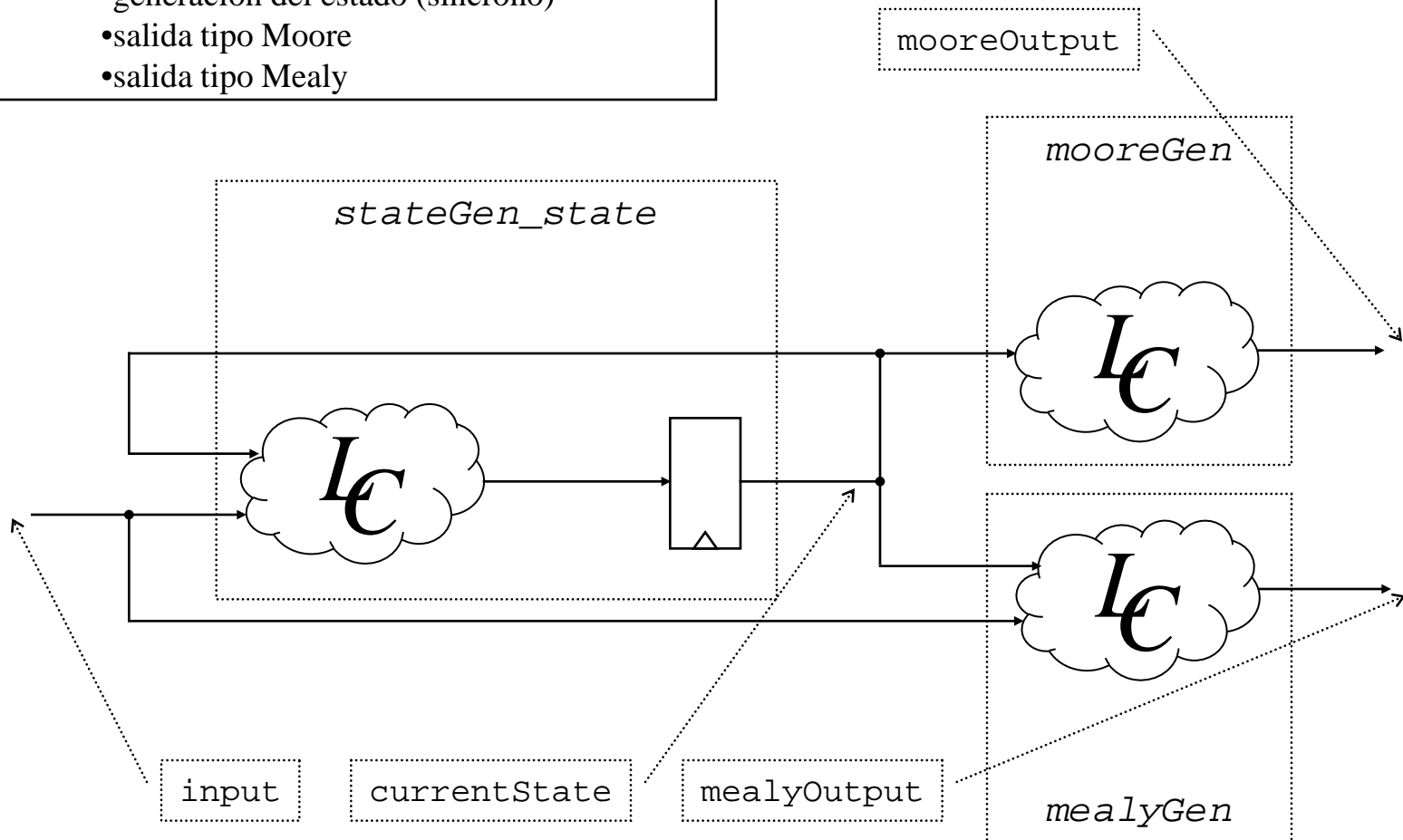
```
process( clk, rst )
begin
  if (rst = '1') then
    q <= '0';
  elsif (clk'event and clk = '1') then
    q <= d;
  end if;
end process;
```


especificación de HW secuencial (i) : método

alternativa

3 procesos:

- generación del estado (síncrono)
- salida tipo Moore
- salida tipo Mealy



especificación de HW secuencial (ii): método

```
stateGen_state:
process( clk, rst, input )
begin
  if (rst = '1') then
    currentState <= ...;
  elsif (clk'event and clk='1') then
    currentState <= ...;
    case currentState is
      when ... =>
        if (input ...) then
          currentState <= ...;
        elsif (input ...) then
          ...
        else
          ...
        end if;
      ...
    end case;
  end if;
end process;
```

```
mealyGen:
process( currentState, input )
begin
  mealyOutput <= ...;
  case currentState is
    when ... =>
      if (input ...) then
        mealyOutput <= ...
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

podría suprimirse

estado al que se salta por defecto (puede ser sobrescrito por posteriores asignaciones)

el calculo del nuevo estado actual en función del estado actual (*currentState* es una señal) debe formar parte de la rama *then* de la especificación de flanco

```
mooreGen:
process( currentState )
begin
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...
    ...
  end case;
end;
```

especificación de HW secuencial (iii): ejemplo

```

Reconoc_101:
process( clk, rst, input )
begin
  if (rst = '1') then
    currentState <= S0;
  elsif (clk'event and clk='1') then
    case currentState is
      when S0 =>
        if (input = '0') then
          currentState <= S0;
        else currentState <= S1;
        end if;
      when S1 =>
        if (input = '0') then
          currentState <= S2;
        else currentState <= S3;
        end if;
      when S2 =>
        if (input = '0') then
          currentState <= S0;
        else currentState <= S1;
        end if;
      when S3 =>
        if (input = '0') then
          currentState <= S2;
        else currentState <= S3;
        end if;
    end case;
  end if;
end process;

```

```

mealyGen:
process( currentState, input )
begin

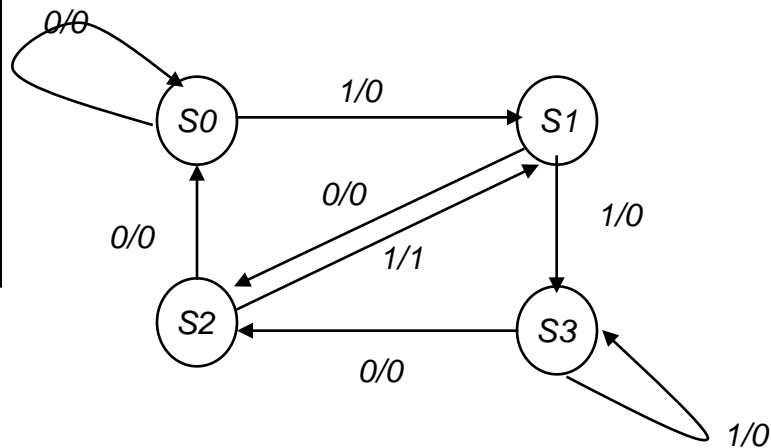
  if (currentState=S2 and input="1") then
    mealyOutput <= 1;
  else
    mealyOutput <= 0;
  end if;
end process;

```

```

type estados_t is (S0, S1, S2, S3);
signal currentState : estados_t;
signal mealyOutput: std_logic;

```

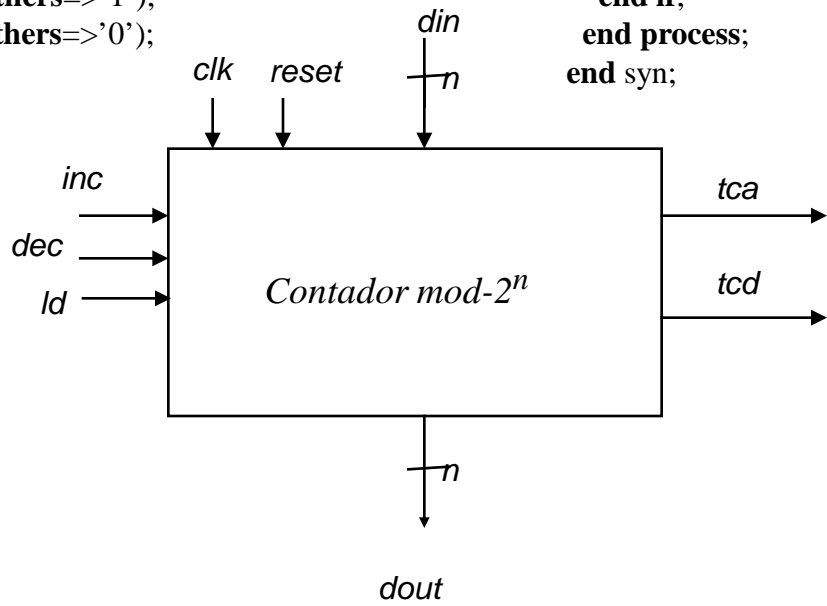


especificación de HW secuencial (iii): ejemplo

```
entity counter is
  generic( n : integer := 8 );
  port(
    clk, rst, ld, inc, dec : in std_logic;
    din : in std_logic_vector( n-1 downto 0 );
    tca, tcd : out std_logic;
    dout : out std_logic_vector( n-1 downto 0 ) );
end counter;

architecture syn of counter is
  signal cs : std_logic_vector( n-1 downto 0 );
begin
  tca <= inc and cs = (others=>'1');
  tcd <= dec and cs = (others=>'0');
  dout <= cs;
end syn;
```

```
process( clk, rst, cs, ld, inc, dec, din )
begin
  if rst='1' then
    cs <= (others=>'0');
  elsif clk'event and clk='1' then
    if ld='1' then
      cs <= din;
    elsif inc='1' then
      cs <= cs + 1;
    elsif dec='1' then
      cs <= cs - 1;
    end if;
  end if;
end process;
end counter;
```



Esquema general de un diseño

```
library IEEE;
use IEEE.std_logic_1164.all;
--use IEEE.std_logic_signed.all;
use IEEE.std_logic_unsigned.all;
entity mi_entidad is
  port (
    ent1,ent2, ...entn: in STD_LOGIC_VECTOR (N downto 0);
    in1, in2, ..., inn: in STD_LOGIC;
    sal1,sal2, ...saln: out STD_LOGIC_VECTOR (K downto 0);
    out1, out2, ..., outn: out STD_LOGIC
  );
end mi_entidad;
architecture mi_entidad_arch of mi_entidad is
  type estados_t is (estado1, estado2, estado3,..., estadoL);
  signal estado : estados_t;
  signal signal1, signal2, ..., signalh : std_logic_vector(H downto 0);
  signal señal1, señal2, señalg: std_logic;
begin
  --Aquí se coloca toda la lógica combinacional. Por ejemplo
  señal<= '1' when signalm=ent1 else '0';
  signal2<= "1110110" when in1='1' else "0111000";
  --También se coloca aquí la lógica combinacional correspondiente a las salidas
  --de las máquinas de estados, como por ejemplo
  sal5 <= signal2(7 downto 0) when estado=estado1 or estado=estado2 or
  estado=estado3 else "ZZZZZZZZ";
```

```
-- A continuación vienen las máquinas de estados.
process( rst, clk )
begin
  if rst='1' then
    estado <= estado1;
  elsif clk'event and clk='1' then
    case estado is
      when estado1 =>
        estado <= estado2;
      when estado2 =>
        estado <= estado3;
      when estado3 =>
        if señal2= '1' then
          estado <= estadon;
        else
          estado <= estadoj;
        end if;
      ....
      when estadon =>
        if inj = '1' then
          estado <= estadoh;
        else
          estado <= estadok;
        end if;
      end case;
    end if;
  end process;
end mi_entidad_arch;
```

Librerías y paquetes

Un **paquete** en VHDL contiene una colección de elementos utilizados habitualmente, como tipos de datos, componentes, etc

Una **librería** es un lugar donde se almacenan las unidades de diseño. Normalmente se usa la librería **work**. Dentro de una librería puede haber varios paquetes definidos. La declaración de librerías y paquetes se hace al principio del código VHDL.

```
library: library_name;
use library_name. package_name.item_name o bien use library_name. package_name.all
```

Librerías y paquetes habituales de los estándares del IEEE 1164 y 1076.3 : `ieee.std_logic_1164`, `ieee.numeric_std.all` y `ieee.std_logic_arith.all`;

Paquete `ieee.std_logic_1164`

En este paquete se definen los tipos **std_logic** and **std_logic_vector**.

Para usarlo debe hacerse la siguiente declaración al principio del fichero VHDL

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

También se define el tipo entero, pero sin número de bits predefinido. Por eso no debe usarse para síntesis. En su lugar debe usarse el paquete **numeric_std** del IEEE standard 1176.3.

Paquete `ieee.numeric_std`

En él se definen dos tipos nuevos de datos: **signed** and **unsigned**, que son arrays de elementos.

```
signal x, y: signed(15 downto 0);
```

Para usarlos, debe realizarse la siguiente declaración al principio del fichero VHDL

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

Librerías y paquetes

También pueden usarse los siguientes paquetes:

std_logic_arith
std_logic_unsigned
std_logic_signed

Para usarlos se declaran al principio:

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

El tipo de datos `std_logic_vector` se interpreta como números binarios de tipo `unsigned` and `signed` en los paquetes `std_logic_unsigned` and `std_logic_signed` respectivamente. Es mejor utilizar el paquete `ieee.numeric_std`.



Diseño Jerárquico

Es muy útil en la gestión de diseños muy grandes y facilita el reúso. Está basado en la utilización de componentes.
Declaración de componentes: *proporciona información sobre el interfaz de los diferentes módulos que forman el diseño.*

```
component component_name
generic(
generic_declaration;
generic_declaration;
...
);
port(
port_declaration;
port_declaration;
...
);
end component;
```

Por ejemplo, si definimos la siguiente entidad, y queremos reusarla, declararíamos la siguiente componente:

<pre>entity dec_counter is port(clk, reset: in std_logic; en: in std_logic; q: out std_logic_vector(3 downto 0); pulse: out std_logic); end dec_counter;</pre>	<pre>component dec_counter port(clk, reset: in std_logic; en: in std_logic; q: out std_logic_vector(3 downto 0); pulse: out std_logic); end component;</pre>
---	---

La declaración de componentes se hace en el cuerpo de la arquitectura, antes del begin



Diseño Jerárquico

Instanciación de componentes: Se realiza dentro de la arquitectura después del begin

```
instance_label: component_name
generic map(
generic_association;
generic_association;
...
)
port map(
port_association;
port_association;
...
);
```

```
one_digit: dec_counter
port map (
    clk=>clk,
    reset=>reset,
    en=>en,
    pulse=>p_one,
    q=>q_one);
ten_digit: dec_counter
port map (
    clk=>clk,
    reset=>reset,
    en=>p_one,
    pulse=>p_ten,
    q=>q_ten);
```