06_vhdl

June 19, 2026

1 Tema 6 — VHDL (Lenguaje de descripción de hardware)

> **Electrónica Digital** · Grado en Ing. Electrónica, Robótica y Mecatrónica (GIERM)
> Apuntes propios a partir de las diapositivas oficiales `ED6-VHDL_2019_2020.pdf`.

Estos apuntes están orientados a **examen**: primero el ejemplo resuelto, después la regla. El foco es el **código VHDL** (bloques "'vhdl); el Python (matplotlib) se usa solo para ilustrar un cronograma de simulación.

1.1 Índice

1.2 1. ¿Qué es VHDL?

VHDL = *Very High Speed Integrated Circuit Hardware Description Language*. Es un lenguaje de **descripción de hardware** para modelar el funcionamiento de un bloque digital. El objetivo es poder **sintetizar** (fabricar el circuito) y **simular** (verificarlo) circuitos digitales.

Idea clave que cae en examen: - Todo código **sintetizable** se puede **simular**, pero **NO** todo código simulable es sintetizable. - A diferencia de un lenguaje de programación (C, Python), **VHDL es concurrente**: las sentencias se ejecutan **en paralelo**, igual que los componentes reales de un circuito funcionan a la vez.

Está regulado por el **IEEE**: - **IEEE 1076**: define las versiones del lenguaje (VHDL-87, -93, -02, -08). - **IEEE 1164**: añade los valores lógicos multievaluados (`std_logic`).

1.2.1 Flujo de diseño

Etapa	Qué ocurre
Descripción funcional en HDL	Se describe el comportamiento en VHDL. Aquí ya se puede **simular**.
Síntesis	Se traduce a "cajas negras" reconociendo bloques básicos (registros, sumadores, comparadores…).
Análisis y elaboración	Se sustituye la tecnología genérica por la real (FPGA o ASIC).
Optimización	Se minimiza el circuito según las restricciones de diseño.
Place & Route	Se obtiene la *netlist* a bajo nivel (puertas y biestables / recursos de la FPGA).

1.3 2. Estructura básica de un fichero VHDL

Todo diseño se compone de tres partes:

1. **Librerías** (`library` / `use`): almacenan tipos, operadores, componentes y funciones, organizados en *packages*. Hay que hacerlas "visibles" para usarlas. Por defecto ya son visibles `std` y `work`.
2. **Entidad** (`entity`): es la **interfaz** del circuito con el exterior — la "caja negra" con sus terminales de entrada/salida (`ports`).
3. **Arquitectura** (`architecture`): **define el funcionamiento** de la entidad. Una entidad puede tener varias arquitecturas, pero solo una está activa.

1.3.1 Ejemplo mínimo: puerta AND de 2 entradas

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;        -- contiene el tipo std_logic

ENTITY and2 IS
    PORT (
        a, b : IN  std_logic;       -- puertos de entrada
        s    : OUT std_logic        -- puerto de salida
    );
END and2;

ARCHITECTURE and2_arq OF and2 IS
BEGIN
    s <= a AND b;                   -- asignación concurrente
END and2_arq;
```

```
   a
            and2      s
   b
```

1.3.2 La entidad en detalle

```
ENTITY nombre_entidad IS
    GENERIC (generic_list);    -- opcional
    PORT (port_list);
END nombre_entidad;
```

Cada **puerto** tiene tres campos: **nombre**, **dirección** (modo) y **tipo de dato**.

Dirección	Descripción
`IN`	Solo entrada. Se puede **leer** pero no escribir.
`OUT`	Solo salida. Se puede **escribir** pero no leer.
`INOUT`	Bidireccional (entrada o salida según la aplicación).
`BUFFER`	Salida que se realimenta internamente. **Se recomienda NO usarla.**

> **Truco de examen:** como un `OUT` no se puede leer, si necesitas usar el valor de una salida dentro de la arquitectura, usa una **señal interna** para operar y al final asígnala al puerto de salida.

1.3.3 `generic`: parametrizar un diseño

`generic` es un campo **opcional** que permite parametrizar (nombre, tipo, valor por defecto). Útil, p.ej., para un sumador de anchura `N` configurable:

```
ENTITY full_adder IS
    GENERIC (
        N : NATURAL := 2
    );
    PORT (
        A   : IN  std_logic_vector(N-1 DOWNTO 0);
        B   : IN  std_logic_vector(N-1 DOWNTO 0);
        SUM : OUT std_logic_vector(N-1 DOWNTO 0);
        C   : OUT std_logic_vector(N-1 DOWNTO 0)
    );
END full_adder;
```

1.4 3. Tipos de datos

1.4.1 3.1 Tipos predefinidos

Paquete `standard` (librería `std`) — escalares:

Tipo	Valores
`bit`	`'0'`, `'1'`
`boolean`	`false`, `true`
`character`	ASCII: `'a'`, `'b'`, …
`integer`	enteros: `137`, `-38` (rango `-MAXINT..MAXINT`)

Tipo	Valores
`natural`	enteros 0
`positive`	enteros > 0
`real`	reales: `0.05`, `1e-9`
`time`	tiempo: `100 fs`, `2 ns` (unidades: fs, ps, ns, us, ms, sec, min, hr)

Compuestos: `bit_vector` (`"000101"`), `string` (`"Hello"`).

Paquete `std_logic_1164` (librería `ieee`) — datos **multievaluados**, los más usados en síntesis: - `std_logic` / `std_ulogic`: además de `'0'` y `'1'`, permiten representar señales no inicializadas (`'U'`), alta impedancia (`'Z'`), no definido (`'X'`), valores débiles pull-up/pull-down, etc.

Paquete `std_logic_arith` (librería `ieee`): tipos `signed` / `unsigned` y funciones de conversión.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
```

1.4.2 Valores de `std_logic` (IEEE 1164)

Valor	Significado
`'0'` / `'1'`	nivel lógico fuerte
`'Z'`	alta impedancia (tri-state)
`'X'`	desconocido / conflicto
`'U'`	no inicializado
`'-'`	*don't care*
`'L'` / `'H'`	pull-down / pull-up (débil)
`'W'`	desconocido débil

1.4.3 3.2 Vectores: `DOWNTO` vs `TO`

```
SIGNAL a : std_logic_vector(7 DOWNTO 0);    -- a(7) = MSB, a(0) = LSB  (lo habitual)
SIGNAL b : std_logic_vector(0 TO 7);        -- b(0) = MSB

a <= "10110100";           -- literal binario
a <= X"B4";                -- literal hexadecimal (equivale a "10110100")
b <= (OTHERS => '0');      -- todos los bits a 0
```

El operador de **concatenación** es `&`:

```
SIGNAL temp : std_logic_vector(1 DOWNTO 0);
temp <= X & Y;             -- crea un vector de 2 bits a partir de X e Y
```

1.4.4 3.3 Tipos enumerados (definidos por el usuario)

```
TYPE estado      IS (reset, libre, ocupado);
TYPE semana      IS (L, M, X, J, V, S, D);
TYPE tipo_puerta IS (andg, org, norg, nandg);
```

Sintaxis general: `TYPE nombre_tipo IS definición_de_tipo;` Son la base de las **máquinas de estados** (ver §10).

1.4.5 3.4 Subtipos (restringir un rango)

```
SUBTYPE valor_bus  IS INTEGER RANGE 0 TO 255;
SUBTYPE laborables IS semana  RANGE L TO V;
PORT ( Qa : OUT INTEGER RANGE 0 TO 7 );      -- subrango directo en un puerto
```

1.4.6 3.5 Records y arrays

```
-- RECORD: agrupa elementos de distinto tipo (se accede con ".")
TYPE switch_info IS RECORD
    status   : bit;
    IDnumber : integer;
END RECORD switch_info;

VARIABLE switch : switch_info;
-- switch.status   := '0';
-- switch.IDnumber := 3;

-- ARRAY: grupo de elementos del mismo tipo
TYPE nibble IS ARRAY (3 DOWNTO 0) OF BIT;
TYPE ram    IS ARRAY (0 TO 31) OF INTEGER RANGE 0 TO 255;

SIGNAL a_bus : nibble;
SIGNAL ram_0 : ram;
```

1.5 4. Objetos: señales, variables y constantes

```
OBJETO nombre_objeto : tipo [ := valor ];
         signal | variable | constant
```

Objeto	Dónde se declara	Operador de asignación	Cuándo se actualiza
signal	en la arquitectura (fuera de process)	<=	al **final** del process
variable	**dentro** de process/function/procedure	:=	**inmediatamente**
constant	donde haga falta (solo lectura)	:= (al declarar)	nunca cambia

1.5.1 Señales

Representan **conexiones físicas** (cables) del circuito. Son locales a la arquitectura. Los puertos de la entidad actúan como señales dentro de su arquitectura.

```
SIGNAL b_datos : bit_vector(15 DOWNTO 0);   -- definición
b_datos <= X"3FB4";                          -- asignación con <=
```

Clave de examen: dentro de un `process`, **una señal NO cambia su valor hasta que termina de evaluarse todo el process** (semántica de actualización diferida). Una **variable** sí cambia en el acto.

1.5.2 Variables (se actualizan al instante)

```
ARCHITECTURE ok OF count_ones IS
BEGIN
    PROCESS(din)
        VARIABLE temp : INTEGER RANGE 0 TO 8;
    BEGIN
        temp := 0;
        FOR i IN 0 TO 7 LOOP
            IF ( din(i) = '1' ) THEN
                temp := temp + 1;            -- ':=' actualiza ya mismo
            END IF;
        END LOOP;
        ones <= temp;                        -- '<=' a la señal de salida
    END PROCESS;
END ok;
```

1.5.3 Constantes

```
CONSTANT set_bit     : BIT := '1';
CONSTANT data_memory : memory := (('0','0','0','0'),
                                  ('0','0','0','1'),
                                  ('0','0','1','1'));
```

1.6 5. Descripción concurrente vs. secuencial

Al describir una arquitectura hay **dos estilos** (combinables en la misma arquitectura):

- **Sentencias concurrentes** → se ejecutan **en paralelo**. El **orden no importa**. El sintetizador las convierte en **lógica combinacional**.
- **Sentencias secuenciales** → van **dentro de un `process`** y se ejecutan en orden. Facilitan la descripción; el sintetizador infiere el mejor circuito.

1.6.1 El orden NO importa (concurrente)

```
-- Estas dos versiones son EQUIVALENTES:
d <= a OR b;        z <= c OR d;
z <= c OR d;        d <= a OR b;
```

Prohibido: no se puede asignar dos valores a la misma señal con dos sentencias concurrentes (`z <= a AND b;` y `z <= m OR c;` a la vez → conflicto/cortocircuito).

1.6.2 Estructura `process`

El `process` es la **unidad básica de ejecución secuencial**. Visto desde fuera, **equivale a una sentencia concurrente** (aunque contenga muchas líneas). Partes:

1. **Lista de sensibilidad**: señales que, al cambiar, **disparan** la reevaluación del proceso.
2. **Declaración de variables** (opcional).
3. **Comportamiento**.

```
mux2to1 : PROCESS(e1, e2, sel)   -- lista de sensibilidad
    -- (declaración de variables si hicieran falta)
BEGIN
    IF ( sel = '0' ) THEN
        o1 <= e1;
    ELSE
        o1 <= e2;
    END IF;
END PROCESS;
```

1.6.3 Lista de sensibilidad: el detalle que suspende

```
proc1 : PROCESS(a, b, c)
BEGIN
    x <= a AND b AND c;
END PROCESS;
```

```
proc2 : PROCESS(a, b)   -- falta 'c'
BEGIN
    x <= a AND b AND c;
END PROCESS;
```

Aunque parecen iguales, **NO** lo son: - `proc1` describe una **AND combinacional** correcta. - `proc2`, al faltar `c` en la lista, **no se reevalúa** cuando cambia `c`. El sintetizador necesita **memorizar** el valor de `c` → **infiere memoria (latch)**, lo cual es un error si querías combinacional.

Reglas: - Para un bloque **combinacional**, en la lista de sensibilidad deben aparecer **TODAS las entradas**. - Un `process` **sin** lista de sensibilidad se ejecuta continuamente → el simulador no avanza.

1.6.4 Semántica de actualización diferida (ejemplo de simulación)

```
PROCESS(B, M)
BEGIN
    M <= B;      -- (1)
    Z <= M;      -- (2)  usa el valor ANTIGUO de M, no el de la línea (1)
END PROCESS;
```

Partiendo de `B = M = Z = 0`, si llega el **evento** `B = 1`:

Paso	Qué pasa	B	M	Z
Inicio	—	0	0	0
Evento	`B` pasa a 1	**1**	0	0
Fin 1ª pasada	`M<=B` (M=1); `Z<=M` usa M **antiguo** (0)	1	**1**	0
Reactivación	cambia `M` se reevalúa	1	1	0
Fin 2ª pasada	ahora `Z<=M` usa M=1	1	1	**1**

Resultado final: `B = M = Z = 1`, pero hizo falta **dos pasadas** porque las señales se actualizan al final de cada evaluación.

1.7 6. Sentencias concurrentes (NO van dentro de `process`)

1.7.1 `WHEN ... ELSE` (asignación condicional)

```
señal_destino <= exp_1 WHEN cond_1 ELSE
                 exp_2 WHEN cond_2 ELSE
                 ...
                 exp_N;                  -- caso por defecto (sin WHEN)
```

1.7.2 `WITH ... SELECT ... WHEN` (asignación selectiva)

```
WITH exp_seleccion SELECT
    señal_destino <= exp_1 WHEN valor_1,
                     exp_2 WHEN valor_2,
                     ...
                     exp_N WHEN OTHERS;    -- OTHERS cubre el resto (obligatorio)
```

1.7.3 Mismo mux 2→1 de tres formas concurrentes

```
-- (a) flujo de datos puro (álgebra de Boole)
ARCHITECTURE arq OF mux_2to1 IS
BEGIN
    o1 <= (e1 AND NOT(sel)) OR (e2 AND sel);
END arq;

-- (b) WHEN ... ELSE
ARCHITECTURE arq OF mux_2to1 IS
BEGIN
    o1 <= e1 WHEN sel = '0' ELSE
          e2;
END arq;

-- (c) WITH ... SELECT
ARCHITECTURE arq OF mux_2to1 IS
BEGIN
    WITH sel SELECT
        o1 <= e1 WHEN '0',
              e2 WHEN OTHERS;
END arq;
```

1.8 7. Sentencias secuenciales (SOLO dentro de `process`)

1.8.1 `IF ... THEN ... ELSIF ... ELSE`

```
IF cond_1 THEN
    sentencias;
ELSIF cond_2 THEN
```

```
    sentencias;
ELSE
    sentencias;
END IF;
```

1.8.2 CASE ... WHEN

```
CASE objeto IS
    WHEN valor_1 => Código_1;
    WHEN valor_2 => Código_2;
    WHEN valor_3 => Código_3;
    WHEN OTHERS  => Código;       -- OTHERS para cubrir todos los casos
END CASE;
```

1.8.3 El mismo mux con IF y con CASE

```
-- IF
ARCHITECTURE arq OF mux_2to1 IS
BEGIN
    mux2to1 : PROCESS(e1, e2, sel)
    BEGIN
        IF ( sel = '0' ) THEN
            o1 <= e1;
        ELSE
            o1 <= e2;
        END IF;
    END PROCESS;
END arq;
```

```
-- CASE
ARCHITECTURE arq OF mux_2to1 IS
BEGIN
    mux2to1 : PROCESS(e1, e2, sel)
    BEGIN
        CASE sel IS
            WHEN '0'    => o1 <= e1;
            WHEN OTHERS => o1 <= e2;
        END CASE;
    END PROCESS;
END arq;
```

El error de los muxes incorrectos (cae mucho): dentro de un `process`, si asignas dos veces a la misma señal **gana la última asignación**. Por eso esta arquitectura **siempre devuelve `e1`**, ignorando `sel`:

```
PROCESS(e1, e2, sel)
BEGIN
    IF ( sel = '1' ) THEN
        o1 <= e2;
```

```
        END IF;
        o1 <= e1;        -- ¡esta SIEMPRE se ejecuta al final y machaca lo anterior!
    END PROCESS;
```

1.9 8. Circuitos combinacionales — ejemplos resueltos

1.9.1 8.1 Multiplexor 4→1 (buses de 8 bits) con CASE

```
ENTITY mux_4_1 IS
    PORT (
        seleccion                                : IN  BIT_VECTOR (1 DOWNTO 0);
        Entrada0, Entrada1, Entrada2, Entrada3 : IN  BIT_VECTOR (0 TO 7);
        Salida                                   : OUT BIT_VECTOR (0 TO 7)
    );
END mux_4_1;

ARCHITECTURE mux_4_1_arq OF mux_4_1 IS
BEGIN
    PROCESS (seleccion, Entrada0, Entrada1, Entrada2, Entrada3)
    BEGIN
        CASE seleccion IS
            WHEN "00" => Salida <= Entrada0;
            WHEN "01" => Salida <= Entrada1;
            WHEN "10" => Salida <= Entrada2;
            WHEN "11" => Salida <= Entrada3;
        END CASE;
    END PROCESS;
END mux_4_1_arq;
```

1.9.2 8.2 Conversor BCD → 7 segmentos con WITH ... SELECT

```
ENTITY bcd_7seg IS
    PORT (
        codigo    : IN  BIT_VECTOR (3 DOWNTO 0);
        segmentos : OUT BIT_VECTOR (7 DOWNTO 0)
    );
END bcd_7seg;

-- Display de ánodo común: se activa con nivel BAJO.
-- Orden de segmentos (7..0): a-b-c-d-e-f-g-dp
ARCHITECTURE convertidor OF bcd_7seg IS
BEGIN
    WITH codigo SELECT
        segmentos <= "00000011" WHEN "0000",   -- 0
                     "10011111" WHEN "0001",   -- 1
                     "00100101" WHEN "0010",   -- 2
                     "00001101" WHEN "0011",   -- 3
                     "10011001" WHEN "0100",   -- 4
                     "01001001" WHEN "0101",   -- 5
```

```
                    "01000001" WHEN "0110",    -- 6
                    "00011111" WHEN "0111",    -- 7
                    "00000001" WHEN "1000",    -- 8
                    "00011001" WHEN "1001",    -- 9
                    "11111111" WHEN OTHERS;    -- apagado
END convertidor;
```

1.9.3 8.3 Comparador de dos registros de 8 bits

```
ENTITY comparador IS
    PORT (
        A, B    : IN  BIT_VECTOR(7 DOWNTO 0);
        EQ, AGB : OUT BIT
    );
END comparador;

ARCHITECTURE comparador_arq OF comparador IS
BEGIN
    EQ  <= '1' WHEN (A = B) ELSE '0';      -- A igual que B
    AGB <= '1' WHEN (A > B) ELSE '0';      -- A mayor que B (A "Greater" B)
END comparador_arq;
```

1.9.4 8.4 Tabla de verdad con concatenación (&)

```
ENTITY ejemplo1 IS
    PORT ( X, Y : IN  BIT;
           F    : OUT BIT );
END ejemplo1;

ARCHITECTURE A1_ejemplo1 OF ejemplo1 IS
    SIGNAL temp : BIT_VECTOR (1 DOWNTO 0);    -- señal interna de 2 bits
BEGIN
    temp <= X & Y;                             -- concatenación X|Y
    F <= '0' WHEN temp = "00" ELSE
         '0' WHEN temp = "01" ELSE
         '1' WHEN temp = "10" ELSE
         '0';
END A1_ejemplo1;
```

1.9.5 8.5 Semisumador de 2 bits (suma + acarreo en un vector)

```
ENTITY sumador_2bits IS
    PORT ( A, B        : IN  BIT;
           SUMA_CARRY  : OUT BIT_VECTOR(1 DOWNTO 0) );
END sumador_2bits;

ARCHITECTURE sumar OF sumador_2bits IS
BEGIN
    SUMA_CARRY <= "00" WHEN (A = '0' AND B = '0') ELSE
```

```
                "10" WHEN (A = '0' AND B = '1') ELSE
                "10" WHEN (A = '1' AND B = '0') ELSE
                "01";                -- (A='1' AND B='1')
END sumar;
```

1.10 9. Diseño jerárquico (component + port map)

Cuando un diseño es grande conviene **dividirlo en bloques** e instanciarlos en una entidad de **jerarquía superior** (TOP). Para usar un bloque de jerarquía inferior:

1. **Antes** del `begin` de la arquitectura → declarar el bloque con `COMPONENT`.
2. **Después** del `begin` → instanciarlo con:
   - `GENERIC MAP`: da valor a los parámetros del bloque.
   - `PORT MAP`: conecta los puertos entre entidades.

1.10.1 Sumador completo a partir de dos semisumadores

```
-- ===== Bloque inferior: semisumador =====
ENTITY semisumador IS
    PORT ( ai, bi   : IN  BIT;
           si, cout : OUT BIT );
END semisumador;

ARCHITECTURE semi OF semisumador IS
BEGIN
    si   <= ai XOR bi;
    cout <= ai AND bi;
END semi;

-- ===== Bloque superior (TOP): sumador completo =====
ENTITY sumador IS
    PORT ( ai, bi, cin : IN  BIT;
           si, cout    : OUT BIT );
END sumador;

ARCHITECTURE suma OF sumador IS
    SIGNAL i1, i2, i3 : BIT;                  -- señales internas de interconexión
    -- Declaración del componente que se va a instanciar
    COMPONENT semisumador
        PORT ( ai, bi   : IN  BIT;
               si, cout : OUT BIT );
    END COMPONENT;
BEGIN
    u1 : semisumador PORT MAP (ai, bi,  i1, i2);   -- 1er semisumador
    u2 : semisumador PORT MAP (i1, cin, si, i3);   -- 2º semisumador
    cout <= i3 OR i2;                              -- acarreo de salida
END suma;
```

La conexión por **posición** (`PORT MAP (ai, bi, i1, i2)`) asocia en orden a `(ai, bi, si, cout)`

del semisumador. También existe la conexión por **nombre**: `PORT MAP (ai => ai, bi => bi, si => i1, cout => i2)`.

1.11 10. Circuitos síncronos

Un circuito síncrono se separa en: - **Lógica combinacional**: calcula salidas y el **estado futuro** a partir del estado actual y las entradas. - **Lógica secuencial**: almacena el **estado actual** (biestables), sincronizada por el **reloj**.

Esto permite **dividir la descripción en dos `process`** (combinacional + secuencial), lo que da diseños más eficientes. Se usan **dos señales**: `p_state` (presente/actual) y `n_state` (próximo/futuro).

1.11.1 10.1 Proceso secuencial: reset asíncrono vs. síncrono

```
-- RESET ASÍNCRONO: el reset está en la lista de sensibilidad
seq : PROCESS(clk, reset)
BEGIN
    IF reset = '1' THEN
        p_state <= idle;
    ELSIF clk'event AND clk = '1' THEN      -- flanco de subida
        p_state <= n_state;
    END IF;
END PROCESS;

-- RESET SÍNCRONO: solo clk en la lista; el reset se evalúa CON el flanco
seq : PROCESS(clk)
BEGIN
    IF clk'event AND clk = '1' THEN
        IF reset = '1' THEN
            p_state <= idle;
        ELSE
            p_state <= n_state;
        END IF;
    END IF;
END PROCESS;
```

> La diferencia clave: en el **asíncrono**, `reset` aparece en la lista de sensibilidad y actúa en cuanto se activa. En el **síncrono**, `reset` se comprueba **dentro** del `IF` del flanco de reloj. `clk'event AND clk = '1'` y `rising_edge(clk)` son **equivalentes** (flanco de subida).

1.11.2 10.2 Biestable D con flanco de subida y `clear` asíncrono (activo a nivel bajo)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY FF_D IS
    PORT (
        CLK : IN  std_logic;
        CLR : IN  std_logic;
```

```
        D   : IN  std_logic;
        Q   : OUT std_logic
    );
END FF_D;

ARCHITECTURE A_FF_D OF FF_D IS
BEGIN
    PROCESS (CLK, CLR)
    BEGIN
        IF CLR = '0' THEN                       -- clear asíncrono activo a nivel bajo
            Q <= '0';
        ELSIF rising_edge(CLK) THEN             -- flanco de subida
            Q <= D;
        END IF;
    END PROCESS;
END A_FF_D;
```

1.11.3 10.3 Registro de 3 bits con clear asíncrono (estilo dos procesos)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY FF_D3 IS
    PORT ( CLK, CLR : IN  std_logic;
           D        : IN  std_logic_vector(2 DOWNTO 0);
           Q        : OUT std_logic_vector(2 DOWNTO 0) );
END FF_D3;

ARCHITECTURE A_FF_D3 OF FF_D3 IS
    SIGNAL n_q, p_q : std_logic_vector(2 DOWNTO 0);
BEGIN
    -- Proceso SECUENCIAL: memoriza el estado
    seq : PROCESS (CLK, CLR)
    BEGIN
        IF CLR = '0' THEN
            p_q <= "000";
        ELSIF rising_edge(CLK) THEN
            p_q <= n_q;
        END IF;
    END PROCESS;

    -- Proceso COMBINACIONAL: estado futuro y salida
    comb : PROCESS (p_q, D)
    BEGIN
        n_q <= D;
        Q   <= p_q;
    END PROCESS;
END A_FF_D3;
```

1.11.4 10.4 Contador reversible módulo 10 (reset asíncrono, precarga síncrona, habilitación)

Cuenta de 0 a 9. `U_D` elige sentido (1 = ascendente). `LOAD` precarga `DATA_IN`. `EN` habilita.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;      -- permite "+"/"-" sobre std_logic_vector

ENTITY CONTADOR IS
    PORT ( CLK      : IN  STD_LOGIC;
           RST      : IN  STD_LOGIC;
           EN       : IN  STD_LOGIC;
           U_D      : IN  STD_LOGIC;
           DATA_IN  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
           LOAD     : IN  STD_LOGIC;
           CONT_OUT : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) );
END CONTADOR;

ARCHITECTURE SYNT OF CONTADOR IS
    SIGNAL CONTEO : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
    PROCESS (CLK, RST)
    BEGIN
        IF RST = '1' THEN                          -- reset ASÍNCRONO
            CONTEO <= "0000";
        ELSIF CLK = '1' AND CLK'EVENT THEN         -- flanco de subida
            IF LOAD = '1' THEN                     -- precarga SÍNCRONA
                CONTEO <= DATA_IN;
            ELSIF EN = '1' THEN                    -- habilitación
                IF U_D = '1' THEN                  -- cuenta ASCENDENTE
                    IF CONTEO = "1001" THEN        -- 9 -> 0 (módulo 10)
                        CONTEO <= "0000";
                    ELSE
                        CONTEO <= CONTEO + "0001";
                    END IF;
                ELSIF CONTEO = "0000" THEN         -- cuenta DESCENDENTE: 0 -> 9
                    CONTEO <= "1001";
                ELSE
                    CONTEO <= CONTEO - "0001";
                END IF;
            END IF;
        END IF;
    END PROCESS;
    CONT_OUT <= CONTEO;
END SYNT;
```

1.11.5 10.5 Autómata síncrono tipo Moore (máquina de estados, FSM)

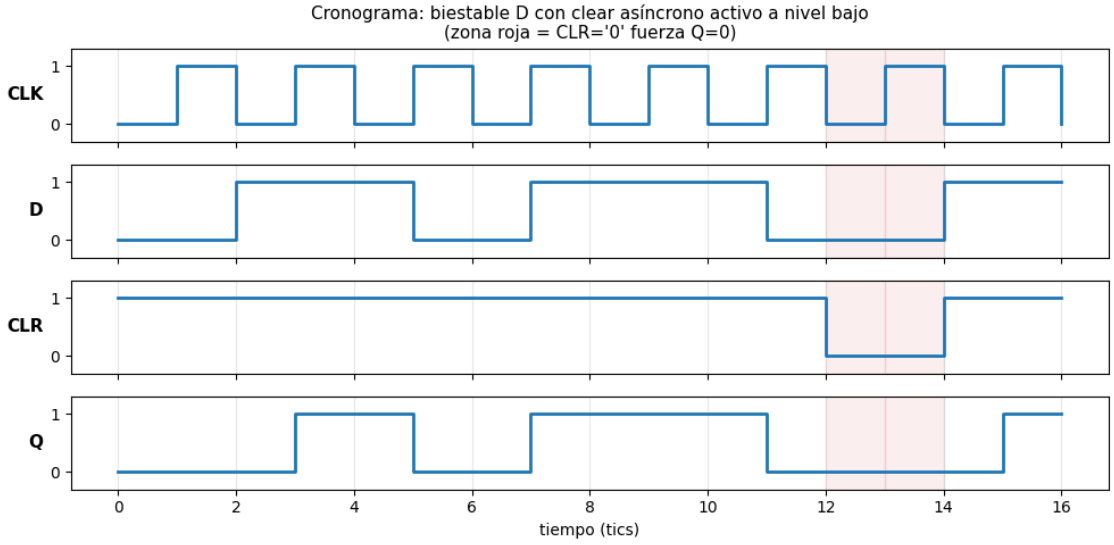
Patrón clásico de examen: **3 bloques**. 1. `process` combinacional del **estado futuro** (`EF`) — un `CASE` sobre el estado presente `EP`. 2. `process` secuencial que **actualiza** `EP <= EF` con el reloj (reset asíncrono). 3. Sentencia concurrente combinacional para la **salida** (en Moore depende **solo** de `EP`).

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Sistema IS
    PORT ( CLK, RESET : IN  Std_logic;
           E          : IN  Std_logic;
           F          : OUT Std_Logic_vector(1 DOWNTO 0) );
END Sistema;

ARCHITECTURE EJ1_ARCH OF Sistema IS
    TYPE   State_Type IS (Inicio, S0, S1, S2, S3, S4);    -- tipo enumerado
    SIGNAL EF, EP : State_Type;                           -- estado futuro / presente
BEGIN

    -- (1) Lógica del PRÓXIMO estado EF (combinacional)
    PROCESS (E, EP)
    BEGIN
        CASE EP IS
            WHEN S1     => EF <= S2;
            WHEN S2     => EF <= S3;
            WHEN S3     => EF <= S4;
            WHEN S0     => IF E = '0' THEN EF <= S0; ELSE EF <= S1;     END IF;
            WHEN Inicio => IF E = '0' THEN EF <= S0; ELSE EF <= Inicio; END IF;
            WHEN S4     => IF E = '0' THEN EF <= S0; ELSE EF <= Inicio; END IF;
            WHEN OTHERS => EF <= S0;
        END CASE;
    END PROCESS;

    -- (2) Registro de estado (secuencial, reset asíncrono activo a nivel alto)
    PROCESS (RESET, CLK)
    BEGIN
        IF RESET = '1' THEN
            EP <= Inicio;
        ELSIF rising_edge(CLK) THEN
            EP <= EF;
        END IF;
    END PROCESS;

    -- (3) Circuito combinacional de SALIDA (Moore: solo depende de EP)
    WITH EP SELECT
        F <= "01" WHEN S2,
             "11" WHEN S3,
```

```
            "01" WHEN S4,
            "00" WHEN OTHERS;

END EJ1_ARCH;
```

1.12 11. Cronograma de simulación (figura con matplotlib)

El VHDL describe hardware, pero en el laboratorio se verifica **simulando** y mirando el **cronograma** (waveform). Abajo dibujamos, con Python, el cronograma del **biestable D con clear asíncrono activo a nivel bajo** (§10.2): `Q` captura `D` en cada flanco de subida de `CLK`, salvo cuando `CLR = '0'`, que fuerza `Q = '0'` de forma asíncrona.




```
Q final: [0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1]
```

1.13 12. Resumen y errores típicos de examen

1.13.1 Chuleta rápida

Quiero…	Uso…	¿Dentro de `process`?
Lógica combinacional simple	`<=` (asignación concurrente)	no
Asignación condicional	`WHEN ... ELSE`	**no**
Asignación selectiva (tipo mux)	`WITH ... SELECT ... WHEN`	**no**
Decisión secuencial	`IF / ELSIF / ELSE`	**sí**
Selección secuencial (mux/FSM)	`CASE ... WHEN`	**sí**
Memoria / biestable	`process` con `rising_edge(clk)`	**sí**

1.13.2 Errores que cuestan puntos

1. **Latch no deseado:** dejar una sentencia condicional **incompleta** (un `IF` sin `ELSE`, o un `CASE` sin `OTHERS`) hace que el sintetizador **conserve el valor anterior** infiriendo un **latch**. → más área, menor frecuencia, sensible a transitorios. **SIEMPRE cubre todas las condiciones.**
2. **Lista de sensibilidad incompleta** en un bloque combinacional → simulación incorrecta / latch.
3. **Asignar dos valores a la misma señal** en concurrente → conflicto (prohibido). En `process`, gana la **última** asignación (causa de los muxes que "siempre devuelven e1").
4. **Confundir `signal` (`<=`, diferida) con `variable` (`:=`, inmediata).**
5. **Leer un puerto `OUT`** → ilegal; usa una señal interna.
6. **`process` sin lista de sensibilidad** → el simulador no avanza.

1.13.3 Equivalencias útiles

- `clk'event AND clk = '1'` ≡ `rising_edge(clk)` (flanco de subida).
- `X"B4"` ≡ `"10110100"` (literal hex ≡ binario).
- `(OTHERS => '0')` inicializa un vector entero a ceros.

Apuntes generados a partir de `ED6-VHDL_2019_2020.pdf` (Electrónica Digital, GIERM, Univ. de Sevilla).