Tema 2. Funciones Lógicas

- Algebra de Conmutación.
- Minimización de funciones Lógicas.
- Introducción al VHDL.

Introducción al VHDL

- Definición de las estructuras básicas. Entidades: genéricos y puertos. Tipos básicos.
- Arquitecturas: operadores básicos. Sentencias concurrentes.
- El proceso: variables, sentencias de control y de lazo.
- Descripción estructural: componentes. El tipo std_logic.

Lenguajes HDL

- Los lenguajes HDL (Hardware Descripction Language) han surgido como una representación alternativa a la representación esquemática tradicional. Son descripciones de tipo texto, con sintaxis similar a la de un lenguaje de programación alto nivel, y cuya finalidad es la descripción del funcionamiento de circuitos digitales integrados.
- El VHDL (Very High-Speed Integrated Circuits Description Language), basado en el lenguaje de programación ADA es uno de los lenguajes (junto a Verilog, similar al lenguaje C) de uso más extendido.

Lenguajes HDL

- Las ventajas de una descripción mediante un HDL son:

  - Portable: la descripción de un circuito realizada para un entorno de diseño puede ser compilada por otro entorno.

  - Permite describir el circuito a varios niveles:

  Nivel de comportamiento y funcional. Cercano a las especificaciones, soporta sentencias de alto nivel, operaciones lógicas y aritméticas que podrían ser sintetizadas automáticamente en un circuito lógico (dependiendo de la capacidad del compilador).

  Nivel estructural. Cercano a la implementación final. Equivale a una descripción esquemática en la que el circuito se describe en base a la conexión de componentes.

  - Pueden aplicarse a estas descripciones herramientas CAD de simulación y de síntesis.

VHDL

- El lenguaje VHDL es muy amplio y tiene una alta complejidad y permite la definición de funciones, de procedimientos, definición de estructuras complejas, punteros, manejo de ficheros, etc. Esta introducción al VHDL se centra en las ideas y construcciones básicas del lenguaje, adecuadas para la representación en VHDL de los circuitos digitales típicos y su síntesis.
- Construcciones principales de VHDL:

  Entity: vista externa del circuito.

  Architecture: asociada a una *entity* ,contiene la descripción del comportamiento del circuito. Una *entity* puede tener asociadas varias *architectures*.

  Package: se utiliza para agrupar definiciones, funciones, etc, que pueden ser necesarios para definir una entity. Por ejemplo todo lo relacionado con un tipo de datos como el std_logic.

  Configuration: asociada a una *entity* se utiliza para caracterizar los módulos y parámetros de un circuito digital desde fuera de su entity. Por ejemplo para una *entity* dada con qué *entitys* y con qué *architectures* se realizan sus componentes internos.

Entity

- El formato básico de esta construcción es:

entity *nombre* **is**
generic (*definiciones*);
port (*definiciones*);
end *nombre*;

```
entity mux2 is
port ( I0, I1, S: in bit;    -- Entradas
       Z: out bit);          -- Salidas
end mux2;
```

- Dentro de la *entity* se definen valores genéricos y puertos de entrada y de salida. Se pueden definir otros elementos comunes a cualquier *architecture* asociada a la *entity*, pero esta introducción se limita a estos elementos.
- Los genéricos definen valores que se pueden luego utilizar en todos los elementos relacionados con la *entity*, por ejemplo el número N de bits de las entradas. El formato de los elementos genéricos es:

generic (*nombre1*: tipo (:= valor);
nombre2: tipo (:= valor);
…….
nombrex: tipo (:= valor));

Entity

- El formato de los puertos es:

  port (*nombrep1*: dirección tipo (:= valor);
  nombrep2: dirección tipo (:= valor);
  ……
  nombrepx: dirección tipo (:= valor));

  Varios puertos con el mismo tipo y dirección pueden indicarse juntos, separados por comas.

- La dirección de los *port* toma los valores:

  in para entradas, el puerto sólo pueden aparecer en el lado derecho de una asignación.

  out para salidas, el puerto sólo pueden aparecer en el lado izquierdo de una asignación).

  inout (entrada-salida).
- El tipo de un *generic* o un *port* indica el rango de valores que puede tomar el elemento.
- Es conveniente asignar un valor por defecto a un genérico, aunque es opcional tanto en los genéricos como en los puertos.

Entity

- Los tipos estándar en VHDL son:

  boolean. Valores TRUE y FALSE.

  bit. Valores ‘0’, ‘1’ (entre comilla simple).

  bit_vector. Conjunto o array de bits del tipo “011”, “1000100” (entre comillas). El número de bits se indica mediante un formato del tipo **bit_vector**(3 **downto** 1): tres bits que se referencian para un nombre a como a(3) (más significativo), a(2) y a(1) (menos significativo), ó **bit_vector**(1 **to** 3): tres bits que se referencian como a(1) (más significativo), a(2) y a(3) (menos significativo).

  integer. Números enteros entre -2147483647 y +2147483647. Se puede crear un rango dentro del tipo entero usando una sentencia del tipo; **integer range** 0 **to** 15, que indica que el dato sólo puede tomar estos valores.

  real. Números reales en formato real 2.35, -4.2E -3. También se pueden crear rangos.

  character. Valores del tipo ‘a’, ‘X’, ’;’, etc, correspondiente al código ASCII de 7 bits.

  string. Conjunto de caracteres entre comillas como “Hola”.

Entity

- Se pueden definir tipos físicos con unidades. Como tipo predefinido se tiene:

  time. Variable física que describe el tiempo. Se indica con valores numéricos seguida de una unidad, por ejemplo 10 ns (10 nanosegundos), 2 min (dos minutos).

  Las unidades son fs (femto seg, 10E-15), ps (pico seg, 10E-12), ns (nano seg, 10E-9), us (micro seg, 10E-6), ms (mili seg, 10E-3), seg, min y hr.

- En las construcciones VHDL, en especial en *package* y *architecture* se pueden definir otros tipos o subtipos (orden **subtype**), algunos complejos como estructuras (**record**) o **arrays**. Sentencias del tipo:

  type valores **is** (‘X’, ‘0’, ‘1’); -- Para poder definir don‘t cares
  type conjunto **is array** (8 **downto** 1) **of integer**; -- 8 enteros
  type semaforo **is** (rojo, amarillo, verde); -- tipo enumerado

Architecture

- Define la operación de una entity a la que está asociada. Su estructura básica es:

```
architecture nombre_arch of nombre_entidad is
-- declaraciones de tipos, señales, componentes, etc
begin
-- Sentencias concurrentes
z1 <= (I0 and (not S)) or (I1 and S);
……
process (I0, I1, S)
-- declaraciones de tipos, variables, etc
begin
    if ( S = '0' ) then    -- Sentencias secuenciales
      z2 <= I0;
    else z1 <= I1;
    end if;
……
end process;
……
U0: and2 port map (a, b, x);  -- Llamadas a otros módulos
……
end nombre_arch;
```

Architecture

- Una definición básica de una *architecture* puede hacerse a base de señales internas y de sentencias concurrentes (se realizan en paralelo, no importa el orden en el que se escriban) formadas por asignaciones y operaciones entre puertos y señales.

- Las señales internas de la *architecture* se definen en la zona de declaraciones mediante la siguiente sentencia, donde el tipo corresponde a un tipo estándar o a un tipo definido anteriormente. Se pueden definir varias señales en la misma línea, separadas por comas, si son del mismo tipo. El valor es opcional.

  signal *nom_signal* : tipo (:= valor);

- Las asignaciones se realizan en sentencias del tipo:

  signal1 **<=** *signal2* op *signal3*;

  donde *op* es un operador, y los operandos y el resultado son señales o puertos. En una sentencia se pueden encadenar operadores y operandos.

Architecture

- Los operadores se aplican sobre unos tipos determinados, un tipo definido por el usuario debería ser acompañado de la definición de operadores para operar con él. Los operadores estándar son:

  Lógicos: **not**, **and**, **nand**, **or**, **nor**, **xor**. Operan con los tipos bit, bit_vector y boolean.

  Relacionales (o de comparación). **=,** **/=** (distinto que), **>**, **<**, **>=**, **<=**. Utilizable por todos los tipos. El resultado es un dato de tipo booleano.

  Aritméticos: (**+**, **-**), (*, /), (** (exponenciación), **rem** (resto), **mod** (módulo)). Operan sobre tipos numéricos como el integer y el real, y tipos físicos como el time.

  Concatenación: **&**. Une datos para formar un array: C <= A & B; une dos señales, A y B, de tipo bit para formar una señal z de tipo bit_vector (2 downto 1) donde C(2) es A, y C(1) es B.

Architecture

- El orden de precedencia de los operadores de mayor a menor es:

  **, ABS, NOT
  *, /, MOD, REM
  +, - (signo)
  +, -, & (operaciones)
  =, /=, <, <=, >, >=
  AND, OR, NAND, NOR, XOR

```
architecture mux2_5 of mux2 is
begin
Z <= (I0 and (not S) ) or (I1 and S);
end mux2_5;
```

  Se deben usar paréntesis para establecer la prioridad en asignaciones complejas.

- VHDL es un lenguaje fuertemente tipado. La compilación de las sentencias de asignación produce error cuando los tipos de los operandos no son correctos.

- A las asignaciones se las puede añadir un retraso temporal, útil en simulación mediante la construcción **after** *tiempo*.

  C <= A **and** B **after** 10 ns;

Architecture

- Se pueden hacer generaciones de sentencias condicionales. La falta de alguna condición en la sentencia genera un elemento secuencial (se mantiene el valor de la señal).

```
señal <= expresión1 when condición1 else
         expresión2 when condición2 else
         ……
         expresiónx when condiciónx else
         expresiónz;

with expresión select  -- Se obtiene el valor
señal <= expresión1 when valor1,
         expresión2 when valor2,
         ……
         (expresiónx when others;) -- Opcional
```

```
architecture mux2_2 of mux2 is
begin
Z <= I0 when S = '0' else I1;
end mux2_2;
```

```
architecture mux2_2b of mux2 is
begin
with S select
Z <= I0 when '0',
     I1 when '1';
end mux2_2b;
```

Architecture

- Se pueden utilizar sentencias de asignación condicionales o controladas por una variable de lazo mediante las sentencia generate.

```
etiq: if ( condicion ) generate
          -- sentencias
         A <= B and C;
      end generate etiqueta;

etiq: for i in vinicial to vfin generate
          -- sentencias
          A(i) <= B(i) and C(i-1);
      end generate etiqueta;
```

(o **downto**)

- Normalmente estas sentencias se utiliza para situar módulos en una descripción estructural.

Architecture

- Ejemplo: Desarrollo de un semisumador de 4 bits en base a sentencias generate

```
entity parasum is
generic (N: integer := 4);
port (A,B:  in bit_vector(N downto 1);
      S:    out bit_vector(N downto 1);
      Cout: out bit);
end parasum;

architecture genera of parasum is
signal C: bit_vector(N downto 1);
begin
G1: for i in 1 to N generate
  G2: if ( i = 1 ) generate
    S(i) <= A(i) xor B(i);
    C(i) <= A(i) and B(i);
  end generate G2;
  G3: if ( i /= 1 ) generate
    S(i) <= A(i) xor B(i) xor C(i-1);
    C(i) <= (A(i) and B(i)) or (A(i) and C(i-1)) or (B(i) and C(i-1));
  end generate G3;
end generate G1;
Cout <= C(4);
end genera;
```

Architecture

- Existen especificaciones que son más fáciles describir mediante sentencias que se realizan una después de otra de forma ordenada, que mediante sentencias en paralelo. Las sentencias ordenadas se describen dentro de la *architecture* en sentencias de tipo *process*. Pueden existir varios *process* dentro de una *architecture* operando en paralelo entre ellos:

  process (lista de sensibilidad)
  -- declaraciones de tipos, variables, etc
  begin
  -- Sentencias secuenciales
  end process;

- Lista de sensibilidad: grupo de señales (separadas por comas). Los cambios de valor (o activación de eventos) en cualquiera de estas señales producen la activación del proceso y la ejecución de sus sentencias. Los cambios en las señales que no estén en la lista no activan el proceso, aunque se usen dentro de él.

Architecture

- En la zona de declaraciones del *process* se definen tipos, variables, etc, sólo reconocibles dentro del *process*. No se definen señales sino variables:

  variable *nombre* : tipo (:= valor);

- Las asignaciones sobre variables se realizan con el operador := en vez del operador <= usado sobre las señales, en sentencias del tipo:

  variable1 **:=** *signal2* op *port3 op variable4*;

- Dentro del *process* se puede operar con los terminales (port), las señales definidas en la arquitectura (signal) y las variables definidas en el *process*. Las señales y puertos tienen sentido físico y no se actualizan hasta el final del *process*, aunque se hagan asignaciones sobre ellas en mitad del mismo. Las variables no son afectadas por tipos de tipo físico (*time*) y se actualizan en el momento que la sentencia es ejecutada dentro del *process*.

- La forma normal de operar sería realizar los cálculos intermedios sobre variables y cargar los resultados en señales antes de finalizar el *process*.

Architecture

- Sentencias en los *process:*

  - Sentencias de asignación en base a operadores usando señales o variables.

  - Sentencias del tipo IF-ELSE:

```
if condición then
   sentencias;
end if;
```

```
if condición then
   sentencias;
else
   sentencias;
end if;
```

```
if condición1 then
   sentencias;
elsif condición2 then
   sentencias;
……
else sentencias;
end if;
```

```
architecture mux2_1 of mux2 is
begin
process (I0, I1, S)
begin
if ( S = '0' ) then Z <= I0;
else Z <= I1;
end if;
end process;
end mux2_1;
```

Architecture

- Sentencias de tipo CASE

```
case expresión is
  when valor1 => sentencias1;
  when valor2 => sentencias2;
  ……….
 (when others => sentenciasx;)
end case;
```

- Sentencias de tipo LAZO

```
for i in vinicial to vfin loop
   -- en función de i
   sentencias;
end loop;

while condición loop
   sentencias;
end loop;
```

(o **downto**)

```
architecture mux2_3 of mux2 is
begin
process (I0, I1, S)
variable inter:bit_vector(3 downto 1);
begin
inter := S & I0 & I1;
case inter is
   when "000" => Z <= '0';
   when "001" => Z <= '0';
   when "100" => Z <= '0';
   when "110" => Z <= '0';
   when others => Z <= '1';
end case;
end process;
end mux2_3;
```

```
architecture uno of DHgen is
begin
process (A,B)
variable inter:integer range 0 to N;
begin
inter := 0;
for i in 1 to N loop
   if ( A(i) /= B(i) ) then
      inter := inter + 1;
   end if;
end loop;
DH <= inter;
end process;
end uno;
```

Architecture

- En VHDL también pueden realizarse descripciones de tipo estructural formada por *componentes* conectados entre si, como en una descripción esquemática. Estos componentes corresponden a otras descripciones VHDL.

- Los componentes deben declararse en la zona de declaraciones de la arquitectura para poder ser reconocidos. En esta declaración se usa el siguiente formato, donde la definición de los genéricos y de los puertos debe coincidir con los de la entidad:

  component *nombre*
  generic(*nom_g*: tipo); -- Definiciones de genericos si hay
  port(*nom_pin*: in *tipo*; *nom_pout*: out *tipo*); -- Definiciones de puertos
  end component;

- La forma más común en VHDL de realizar la declaración de componentes es utilizando paquetes (**package**), declarando varios componentes conjuntamente. Usando (**use**) el paquete en una entidad todos los componentes quedan reconocidos, por lo que ya pueden usarse en la descripción.

Architecture

- Para incluir un componente en una *architecture* se usa un formato como el siguiente:

  etiqueta: *nombre_entity*
  generic map(*valor1*, *valor2*, …)
  port map(*señal1*, *señal2*, …);

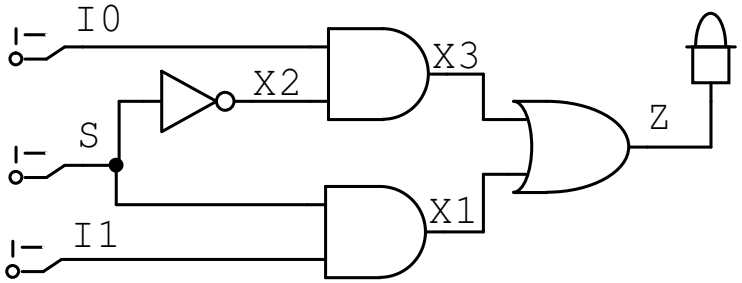
- La asociación de valores a los genéricos y puertos puede hacerse siguiendo directamente el orden en que están declarados (separados por comas), o asociando los datos directamente a cada uno mediante un formato del tipo nom_gen => valor. Se puede dejar un puerto desconectado usando la palabra **open**.

```
architecture usaDH of pru is
signal D1,D2: bit_vector(6 downto 1);
signal num: integer range 0 to 6;

component DHgen
generic(N:integer);
port (A,B:in bit_vector(N downto 1);
      DH: out integer range 0 to N);
end component;

begin
U0: DHgen generic map(6)    --o N => 6
          port map(D1, D2, num);
    -- o port map(A =>  D1,B => D2, DH => num);
end usaDH;
```

Architecture




```
architecture mux2_6 of mux2 is
component and2 port (A,B: in bit;
                     Z: out bit);
end component;
component or2 port (A,B: in bit;
                    Z: out bit);
end component;
component inv port (A: in bit;
                    Z: out bit);
end component;
signal X1, X2, X3: bit;
begin
U0: and2 port map(I1, S, X1);
U1: inv port map(S, X2);
U2: and2 port map(I0, X2, X3);
U3: or2 port map(X1, X3, Z);
end mux2_6;
```

```
entity inv is
port (A: in bit; Z: out bit);
end inv;
architecture uno of inv is
begin
Z <= not A after 10 ns;
end uno;
```

```
entity or2 is
port (A, B: in bit; Z: out bit);
end or2;
architecture uno of or2 is
begin
Z <= A or B after 10 ns;
end uno;
```

```
entity and2 is
port (A, B: in bit; Z: out bit);
end and2;
architecture uno of and2 is
begin
Z <= A and B after 10 ns;
end uno;
```

Tipo std_logic

- Los tipos estándar *bit*, *integer*, etc, no son capaces de describir todas las situaciones lógicas que se producen en un circuito, por ejemplo los *don´t cares*, ni situaciones electrónicas como la desconexión de un nudo. Las descripciones VHDL utilizan normalmente el tipo *std_logic* que sustituye al tipo bit y puede tener asignado uno de los siguientes nueve valores:

- '0': 0 Lógico.
- '1': 1 Lógico.
- '-': don't care.
- 'Z': alta impedancia. Nudo desconectado.
- 'U': no inicializado. El nudo está a 0 ó a 1 pero no se sabe a qué valor. Se utiliza en circuitos secuenciales cuando comienza a funcionar el circuito.
- 'L': 0 débil. Valor de tensión bajo en circuitos electrónicos.
- 'H': 1 débil. Valor de tensión alto en circuitos electrónicos.
- 'X': Valor desconocido. Zona de incertidumbre no válida en circuitos electrónicos.
- 'W': desconocido débil.

Tipo std_logic

- Al igual que el tipo bit se substituye por el tipo std_logic, el tipo bit_vector es substituido por el tipo std_logic_vector.

- La definición del tipo std_logic se realiza en distintos paquetes contenidos en la librería ieee. Para usar este tipo en una entidad hay que utilizar al menos el paquete std_logic_1164 mediante estas dos sentencias:

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

- Existen otros paquetes relacionados con el tipo std_logic que permiten utilizarlo con operadores aritméticos como el + y el -. Cuando se usa el paquete std_logic_signed se puede operar con los tipos std_logic y std_logic_vector como si fueran datos numéricos con signo en complemento-2; si se usa el paquete std_logic_unsigned se pueden usar estos tipos como datos numéricos sin signo.

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

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

Tipo std_logic

- El paquete std_logic_arith permite definir dos nuevos tipos: *signed* y *unsigned* similares al tipo std_logic_vector que operan directamente como números con signo (c-a-2) o sin signo respectivamente.

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

- Estos paquetes incluyen funciones de conversion *to_unsigned()*, *to_integer()*, etc, que permiten cambiar el tipo de los datos cuando es necesario.

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

entity sumauns is
generic(N: integer:=4);
port (A, B: in std_logic_vector(N downto 1);
      Sum: out std_logic_vector(N+1 downto 1));
end sumauns;

architecture uno of sumauns is
begin
Sum <= ('0' & A) + ('0' & B);
end uno;
```

Tipo std_logic

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

entity sumasig is
generic(N: integer:=4);
port (A, B: in std_logic_vector(N downto 1);
      Sum: out std_logic_vector(N+1 downto 1));
end sumasig;

architecture uno of sumasig is
begin
Sum <= (A(N) & A) + (B(N) & B);
end uno;
```

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

entity sumasig2 is
generic(N: integer:=4);
port (A, B: in signed(N downto 1);
      Sum: out signed(N+1 downto 1));
end sumasig2;

architecture uno of sumasig2 is
begin
Sum <= (A(N) & A) + (B(N) & B);
end uno;
```