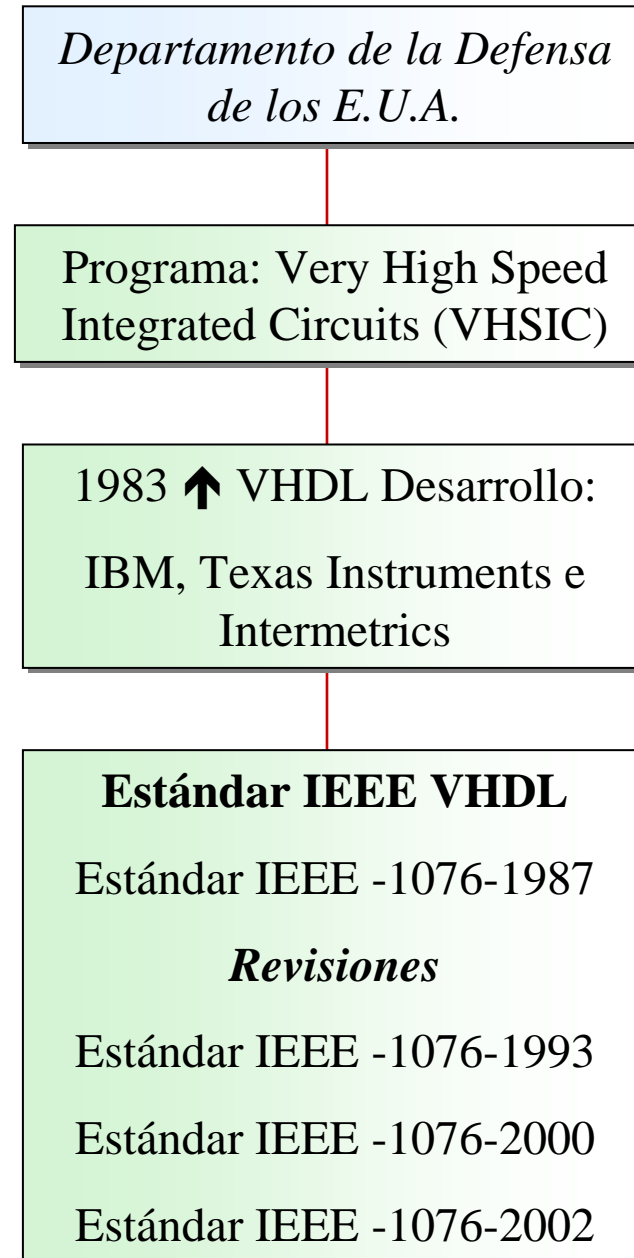
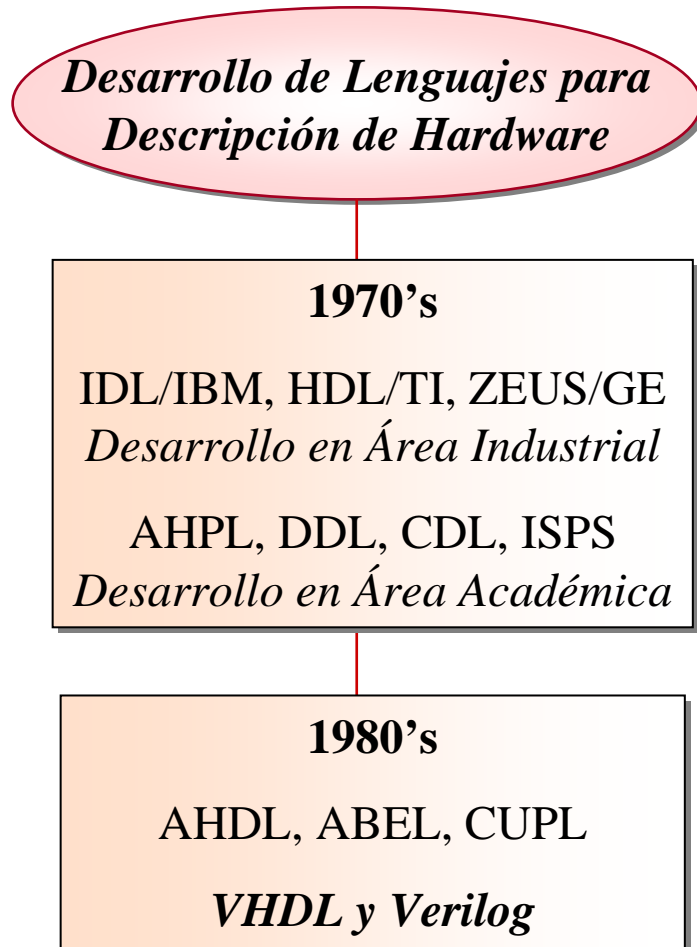




Curso:

VHDL (VHSIC Hardware Description Language)

VHSIC – Very High Speed Integrated Circuit





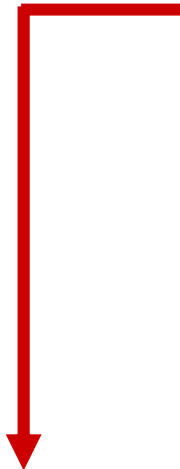
<i>Ventajas de VHDL</i>
Notación Estandarizada
Disponibilidad al Público
Independencia del Sistema de Desarrollo (con algunas excepciones)
Independencia de la Metodología de Diseño (PLDs, ASICs, FPGAs)
Independencia de la Tecnología y Proceso de Fabricación (CMOS, Bipolar, BiCMOS)
Reutilización de Código
Capacidad descriptiva del comportamiento del sistema en distintos niveles de abstracción: <i>Algoritmo, RTL</i> (Register Transfer Logic), <i>Lógico y Compuerta</i>
Facilitar la Verificación/Prueba del Sistema siendo diseñado.
Adición de la extensión analógica (IEEE1076.1) que permite la especificación, simulación y síntesis de sistemas digitales, analógicos y mixtos



<i>Elementos sintácticos del VHDL</i>	
Comentarios	Se consideran comentarios después de dos guiones medios seguidos "--"
Símbolos especiales	Existen caracteres especiales sencillos como (&, #) o dobles como (:=, <=)
Identificadores	Es lo que se usa para dar nombre a los diferentes objetos del lenguaje
Números	Se considera que se encuentra en base 10, se admite la notación científica convencional es posible definir números en otras bases utilizando el símbolo # : 2#11000100#
Caracteres	Es cualquier letra o carácter entre comillas simples: '3', 't'
Cadenas	Son un conjunto de caracteres englobados por comillas dobles: "hola"
Cadenas de bits	Los tipos bit y bit_vector son en realidad tipo carácter y arreglo de caracteres respectivamente, se coloca un prefijo para indicar la base : O"126", X"FE"
Palabras reservadas	Son las instrucciones, órdenes y elementos que permiten definir sentencias.



Identificadores	
Nombres o etiquetas que se usan para referirse a: Variables, Constantes, Señales, Procesos, Entidades, etc.	
Están formados por números, letras (mayúsculas o minúsculas) y guión bajo “_” con las reglas especificadas en la tabla siguiente.	
Longitud (Número de Caracteres): Sin restricciones	
Palabras reservadas por VHDL no pueden ser identificadores	
En VHDL, un identificador en mayúsculas es igual a su contraparte en minúsculas	



Reglas para especificar un identificador	<i>Incorrecto</i>	<i>Correcto</i>
Primer carácter debe ser siempre una letra mayúscula o minúscula	4Suma	Suma4
Segundo carácter no puede ser un guión bajo (_)	S_4bits	S4_bits
Dos guiones bajos no son permitidos	Resta__4	Resta_4_
Un identificador no puede utilizar símbolos especiales	Clear#8	Clear_8

***Lista de palabras reservadas en VHDL***

abs	downto	library	postponed	subtype
access	else	linkage	procedure	then
after	elsif	literal	process	to
alias	end	loop	pure	transport
all	entity	map	range	type
and	exit	mod	record	unaffected
architecture	file	nand	register	units
array	for	new	reject	until
assert	function	next	rem	use
attribute	generate	nor	report	variable
begin	generic	not	return	wait
block	group	null	rol	when
body	guarded	of	ror	while
buffer	if	on	select	with
bus	impure	open	severity	xnor
case	in	or	shared	xor
component	inertial	others	signal	
configuration	inout	out	sla	
constant	is	package	sra	
disconnect	label	port	srl	



Mayor



Menor

Precedencia de operadores						
NOT	ABS	**				
*	/	MOD	REM			
+ (signo)	- (signo)					
+	-	&				
sll	srl	sla	sra	rol	ror	
=	/=	<	<=	>	>=	
AND	OR	NAND	NOR	XOR	XNOR	

La precedencia de operadores se encuentran ordenados de mayor (arriba) a menor (abajo), los operadores que se encuentran en la misma fila tienen la misma precedencia.



PRECEDENCIA DE OPERADORES

Los operadores anteriores se definen de la siguiente manera:

1. Operadores lógicos binarios: **and or nand nor xor xnor**.
2. Operadores relacionales: **= /= < <= > >=**.
3. Operadores de desplazamiento: **sll srl sla sra rol ror**.
4. Operadores de adición: **+** **-** **&** (concatenación).
5. Operadores de signo: **+** **-**
6. Operadores de multiplicación: ***** **/** **mod rem**.
7. Operadores misceláneos: **not abs ****

Cuando no se usan los paréntesis, los operadores de la clase 7 tienen la mayor precedencia y se aplican primero, seguidos en ésta por los de la clase 6, luego la 5 y así sucesivamente. Los de la clase 1 tienen la menor precedencia y se aplican al último. Los operadores de la misma precedencia se aplican de izquierda a derecha en la expresión.

El orden de la precedencia puede cambiarse mediante los paréntesis.

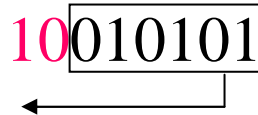


PRECEDENCIA DE OPERADORES

Ejercicio:

Realizar las siguientes operaciones al vector $A = \text{"10010101"}:$

A **sll 2** (desplazamiento lógico hacia la izquierda llenando con ceros "0"):



$$A \text{ sll } 2 = 01010100$$

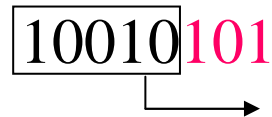


PRECEDENCIA DE OPERADORES

Ejercicio:

Realizar las siguientes operaciones al vector $A = \text{"10010101"}:$

$A \text{ srl } 3$ (desplazamiento lógico hacia la derecha llenando con ceros "0"):



$$A \text{ srl } 3 = 00010010$$



PRECEDENCIA DE OPERADORES

Ejercicio:

Realizar las siguientes operaciones al vector $A = \text{"10010101"}:$

A **sla** 3 (desplazamiento aritmético hacia la izquierda llenando con el bit a la derecha):

10010101

A **sla** 3 =

10101111

Bit a la derecha



PRECEDENCIA DE OPERADORES

Ejercicio:

Realizar las siguientes operaciones al vector $A = \text{"10010101"}:$

A sra 2 (desplazamiento aritmético hacia la derecha llenando con el bit a la izquierda):

10010101

A sra 2 =

1100101

Bit a la izquierda

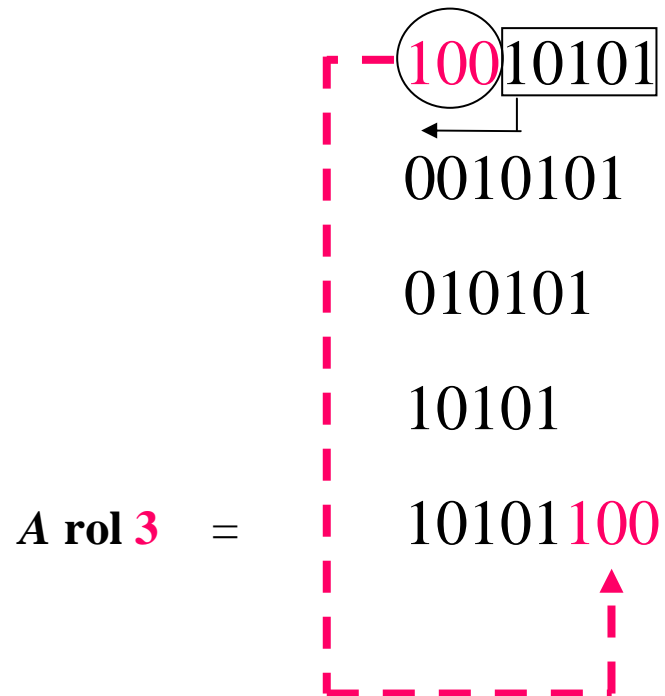


PRECEDENCIA DE OPERADORES

Ejercicio:

Realizar las siguientes operaciones al vector $A = \text{"10010101"}:$

$A \text{ rol } 3$ (rotación a la izquierda):



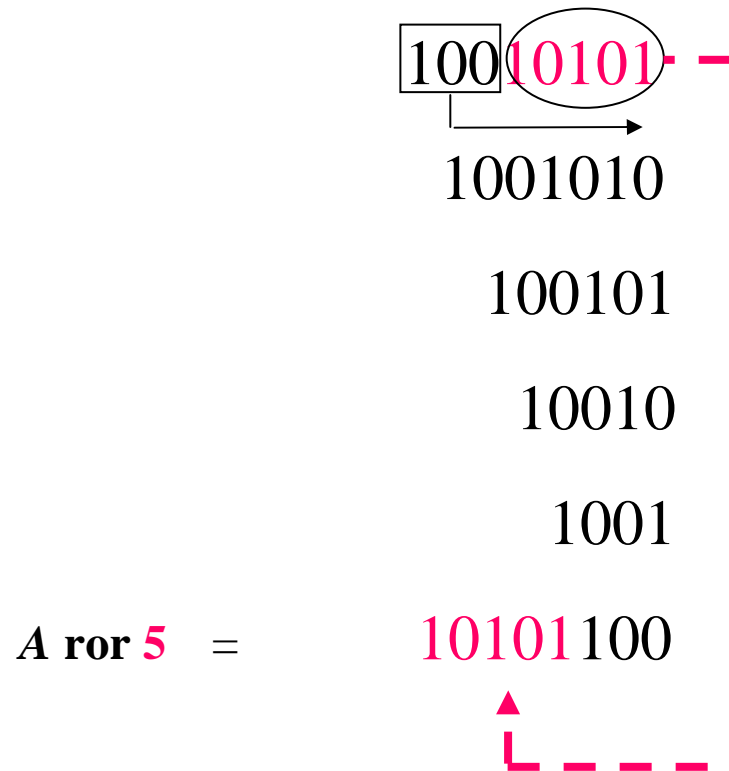


PRECEDENCIA DE OPERADORES

Ejercicio:

Realizar las siguientes operaciones al vector $A = \text{"10010101"}:$

$A \text{ ror } 5$ (rotación a la derecha):





PRECEDENCIA DE OPERADORES

Ejercicio:

En la siguiente expresión, *A*, *B*, *C* y *D* son del tipo *bit_vector*.

$$(A \ \& \ \text{not } B \ \text{or } C \ \text{ror } 2 \ \text{and } D) = \text{"110010"}$$

Entonces, los operadores se aplicarán en el siguiente orden:

not, **&**, **ror**, **or**, **and**, **=**

Si *A*= "110", *B*= "111", *C*= "011000" y *D*= "111011", las operaciones se realizan como se muestra a continuación:

not B = "000" (complemento bit por bit)

A & not B = "110000" (concatenación)

C ror 2 = "000110" (rotación a la derecha dos lugares)

(A & not B) or (C ror 2) = "110110" (operación **or** bit por bit)

(A & not B or C ror 2) and D = "110010" (operación **and** bit por bit)

[(A & not B or C ror 2 and D) = "110010"] = TRUE (el paréntesis fuerza a la prueba de igualdad al final, resultando en verdadero (**TRUE**))

Op. lógicos bin.: **and or nand nor xor xnor**.

Operadores relacionales: **= /= < <= > >=**.

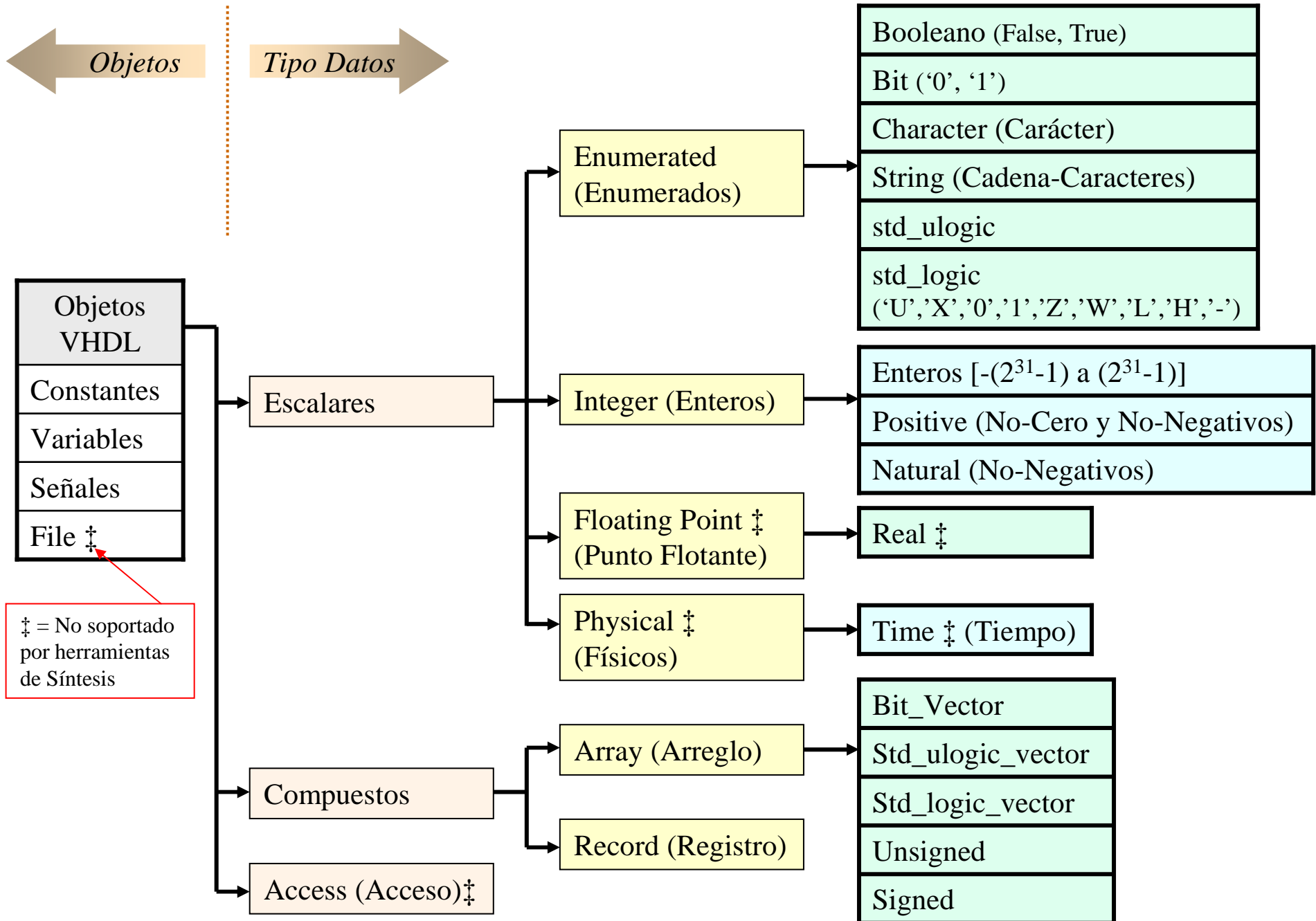
Op. de desplazamiento: **sll srl sla sra rol ror**.

Operadores de adición: **+** **-** **&** (concatenación).

Operadores de signo: **+** **-**

Operadores de multiplicación: ***** **/** **mod** **rem**.

Operadores misceláneos: **not abs ****





Objetos de Datos

Un objeto de datos en VHDL es un elemento que toma un valor de algún tipo de dato determinado. Según sea el tipo de dato, el objeto poseerá un conjunto de propiedades. En VHDL los objetos de datos son generalmente una de las tres clases siguientes:

Constantes

Una constante es un elemento que puede tomar un único valor de un tipo dato, las constantes pueden ser declaradas dentro de entidades, arquitecturas, procesos y paquetes.

CONSTANT identificador : tipo := valor;

Ejemplo

CONSTANT byte: integer := 8;

Variables

Las variables pueden ser modificadas cuando sea necesario, pueden ser declaradas solamente dentro de los procesos y subprogramas.

VARIABLE identificador : tipo [:= valor];

Ejemplo

VARIABLE aux1, aux2: bit;

Señales

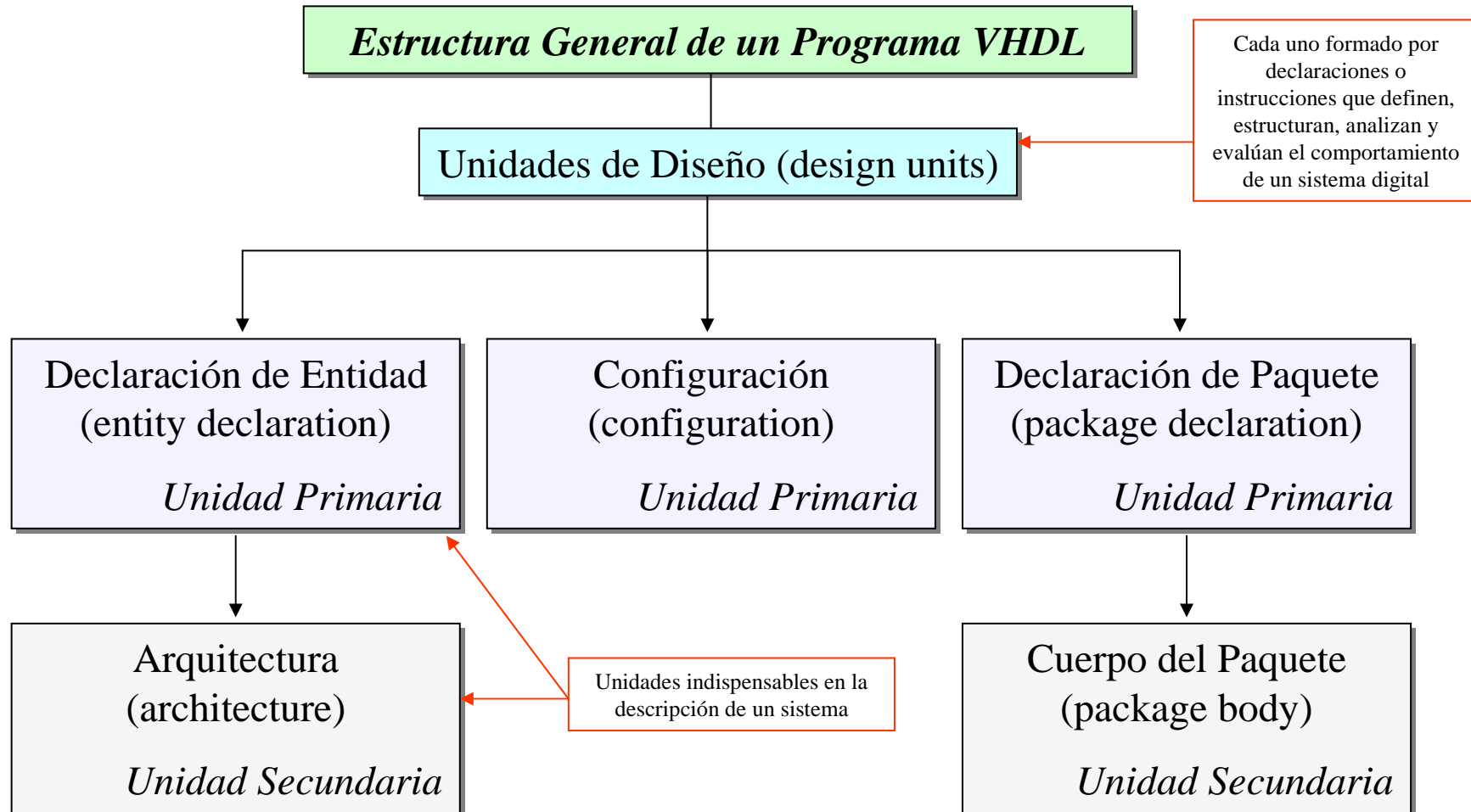
Las señales sí pueden almacenar o pasar valores lógicos, por lo tanto, representan elementos de memoria o conexiones y si pueden ser sintetizadas. Son declaradas en las arquitecturas antes del **BEGIN**.

SIGNAL identificador : tipo [:= valor];

Ejemplo

SIGNAL A, B : bit := '0';

SIGNAL dato: bit_vector (7 **downto** 0);



entidad (**entity**) → Bloque elemental de diseño



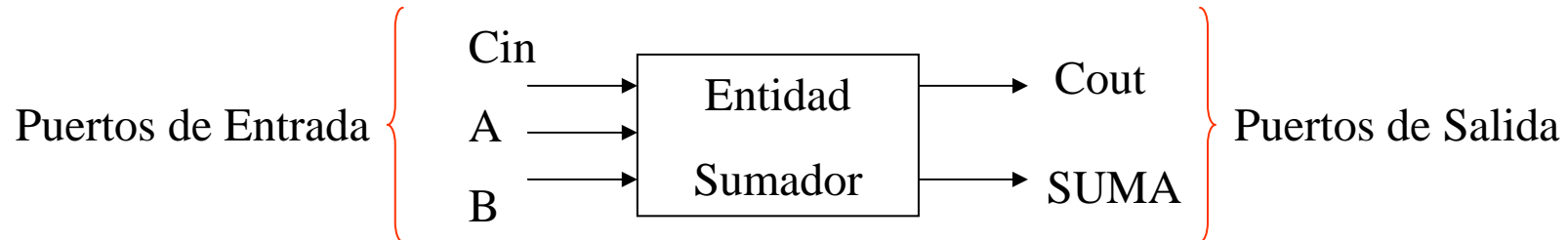
Circuitos elementales digitales que forman de manera individual o en conjunto un sistema digital



Ejemplos: Compuertas, Flip-Flops, Sumadores/Restadores, Multiplexores, Contadores, Multiplicadores, ALUs, Neurona-Digital, etc.



Ejemplo-1

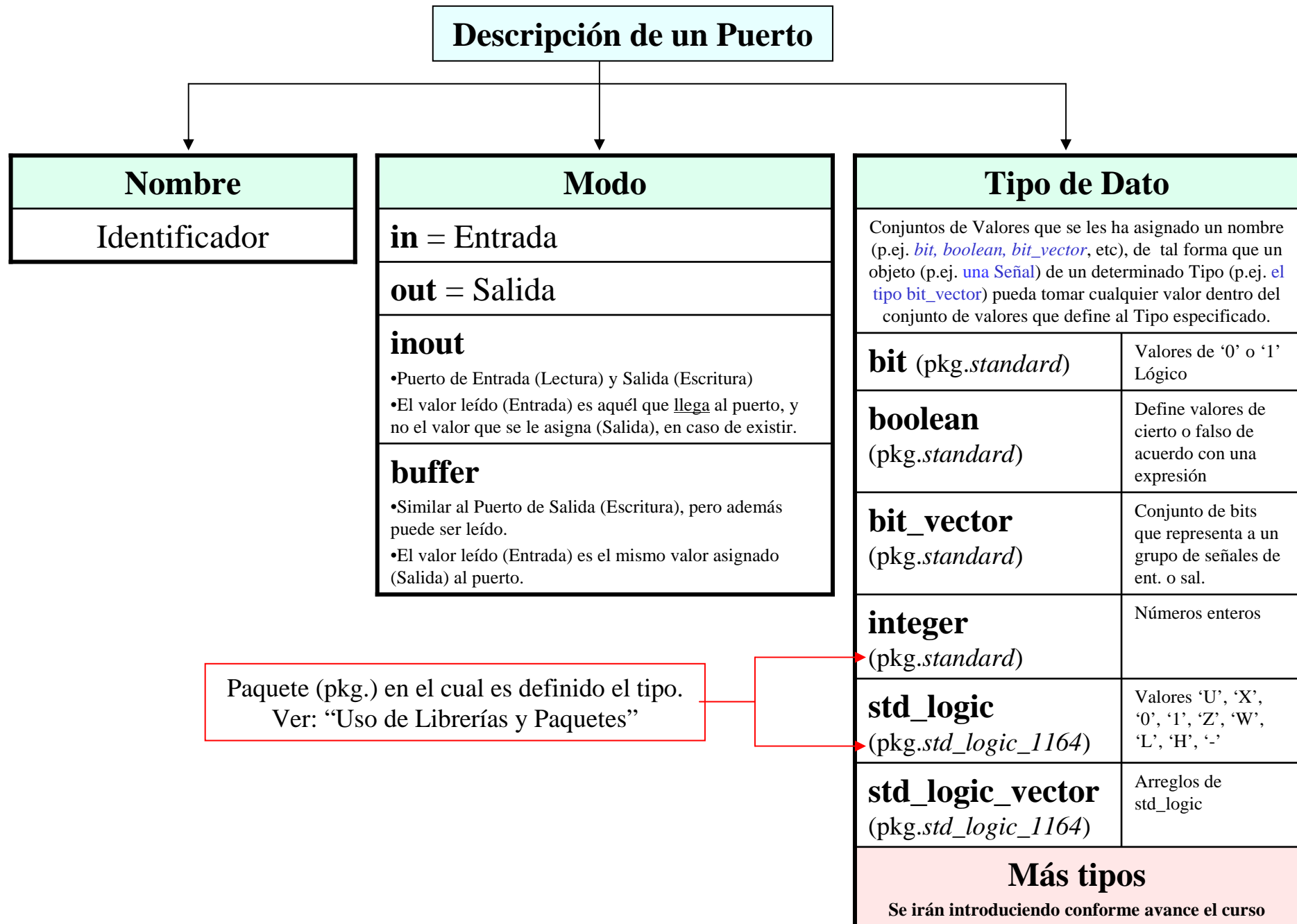


Declaración de una entidad → Consiste en la descripción de los puertos de entrada o salida de un circuito, el cual es identificado como una entidad (**entity**)

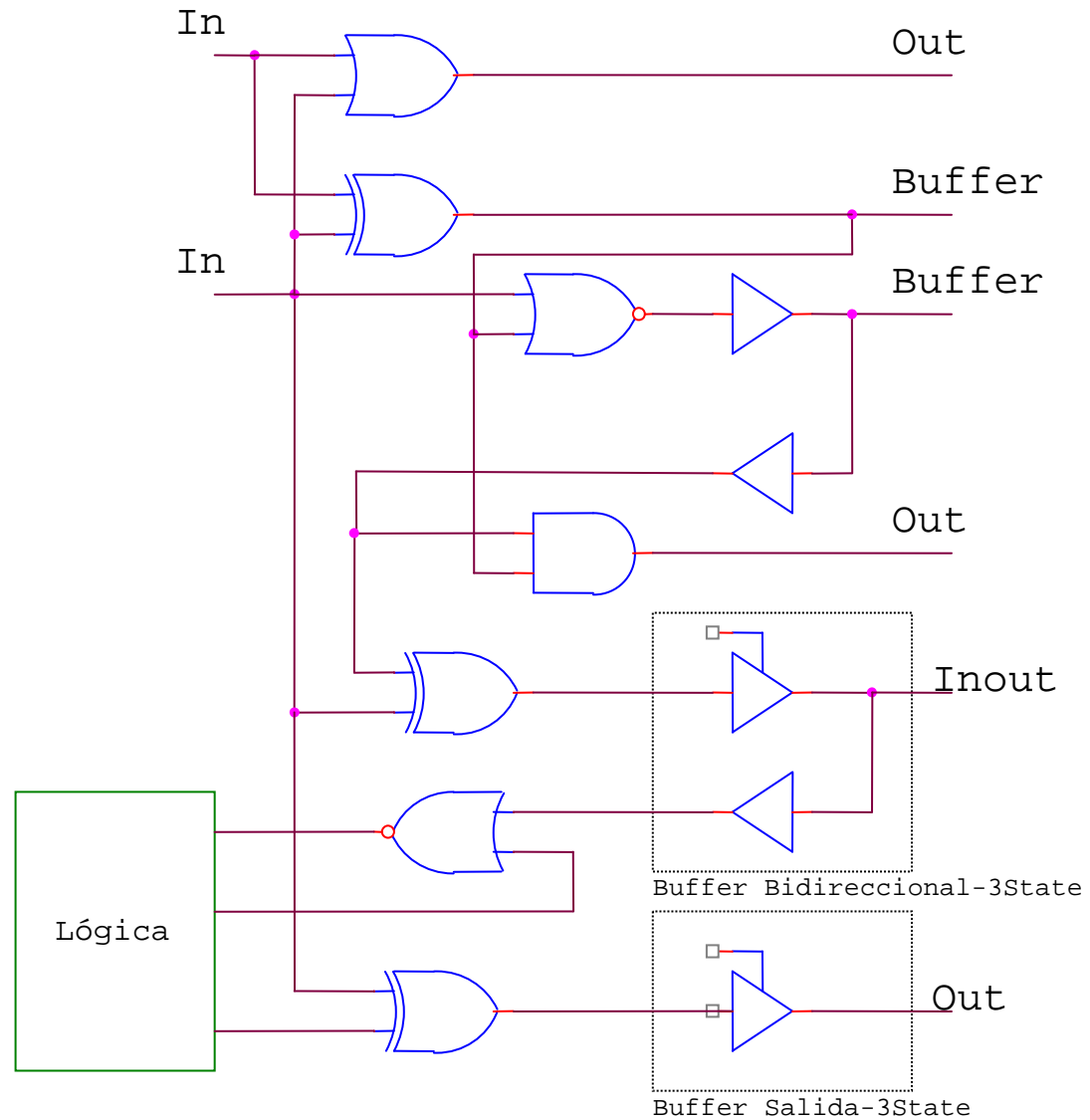


¡Importante!

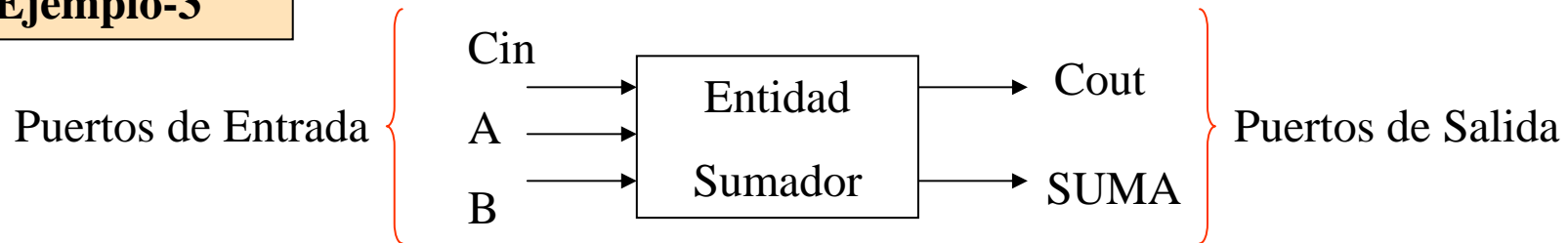
No se describe cómo será realizado o implementado el circuito, es decir, su Arquitectura



Ejemplo-2a



Ejemplo-3



Línea N°.	Sumador-completo de dos datos con longitudes de 1-bit (Declaración de Entidad)
1	--Declaración de la entidad de un circuito sumador
2	entity sumador is
3	port (A, B, Cin: in bit;
4	SUMA, Cout: out bit);
5	end sumador;

(entity) Inicia declaración de la entidad

(--) Indica Comentario

Identificador de la entidad

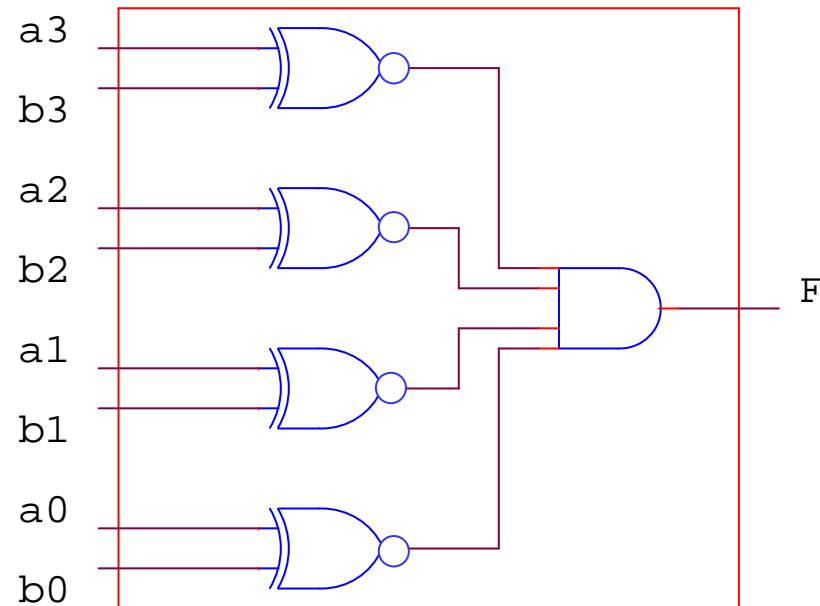
Nombres de los puertos

Modo de Operación

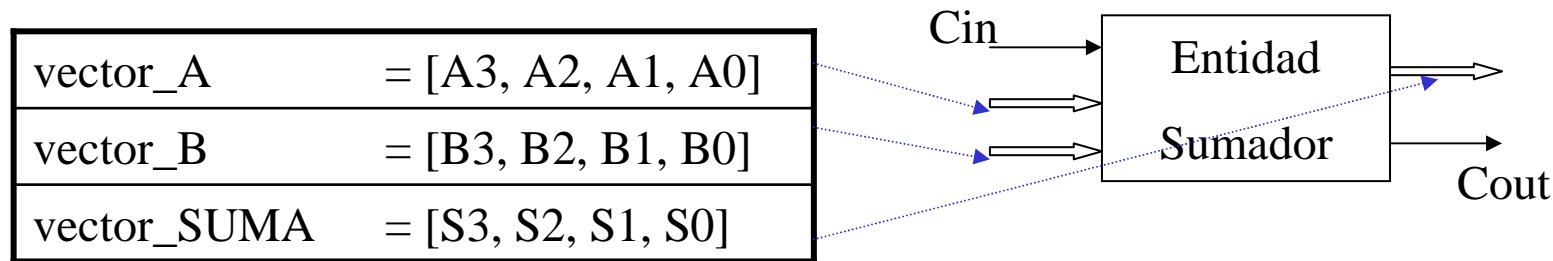
Tipo de Dato

(;) Finaliza declaración o subdeclaración

(end) Finaliza declaración de la entidad

**Ejemplo-4**

Línea N°.	Comparador – Uso de dos datos con longitudes de 4-bit (Declaración de Entidad)
1	--Declaracion de la entidad
2	entity circuito is
3	port (a3, b3, a2, b2, a1, b1, a0, b0: in bit;
4	F: out bit);
5	end circuito;



Declaración de Puertos Tipo-Vector

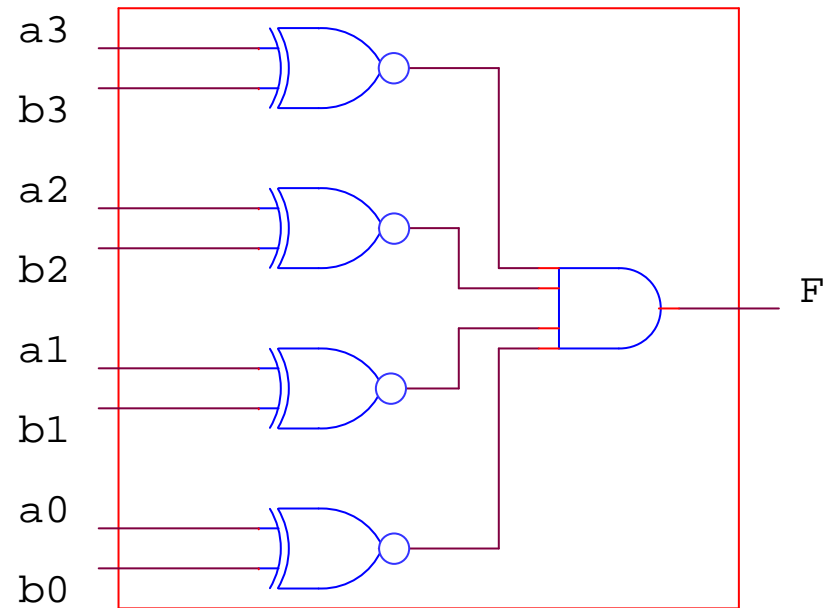
```
port (vector_A, vector_B: in bit_vector (3 downto 0);  
      vector_SUMA: out bit_vector (3 downto 0));
```

Para ordenar en forma ascendente utilizar **to** en lugar de **downto** (p.ej. 0 **to** 3)

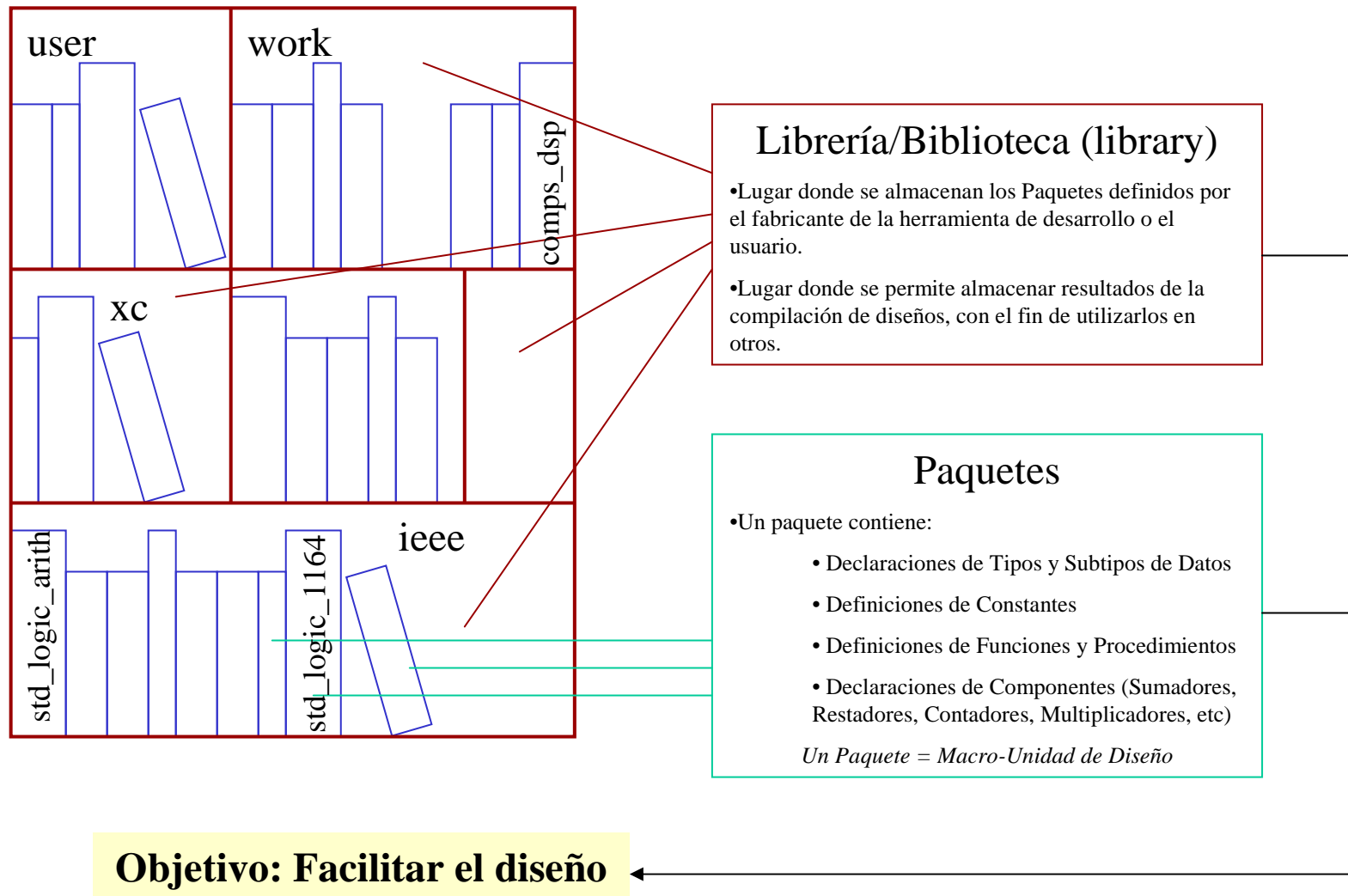
Sumador-completo de dos datos con longitudes de 4-bit (Declaración de Entidad – Uso de Vectores)

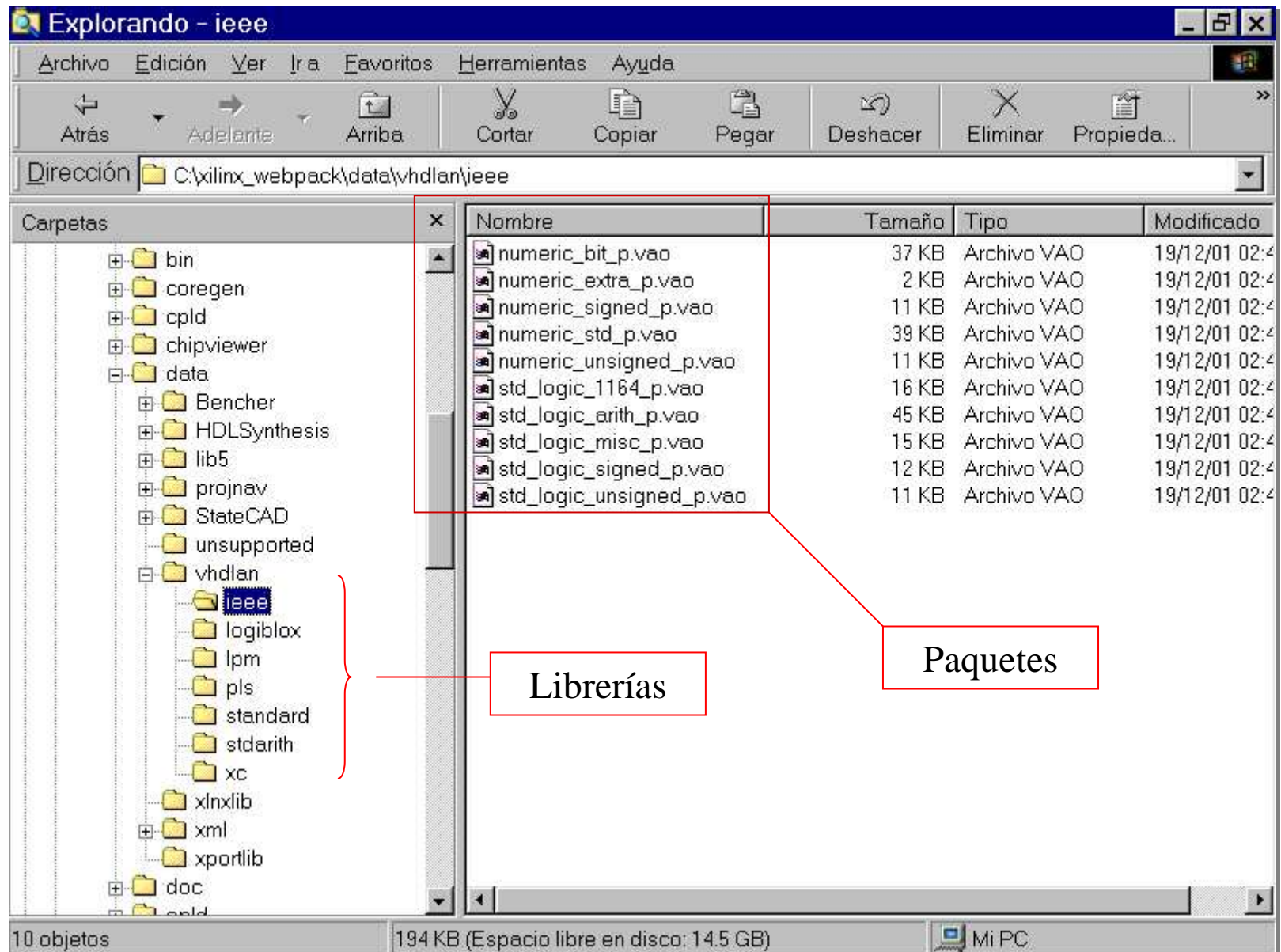
```
entity sumador is  
  port (A, B: in bit_vector (3 downto 0);  
        Cin: in bit;  
        Cout: out bit;  
        SUMA: out bit_vector (3 downto 0));  
end sumador;
```

Ejemplo-5

**Ejemplo-6**

Línea N°.	Comparador – Uso de dos datos con longitudes de 4-bit (Declaración de Entidad – Uso de Vectores)
1	--Declaracion de la entidad
2	entity circuito is
3	port (a, b: in bit_vector (3 downto 0);
4	F: out bit);
5	end circuito;







Para llamar un paquete es necesario llamar a la librería/biblioteca que lo contiene
(donde ha sido compilado)

Sintaxis: **use** nombre_librería.nombre_paquete.**all**;

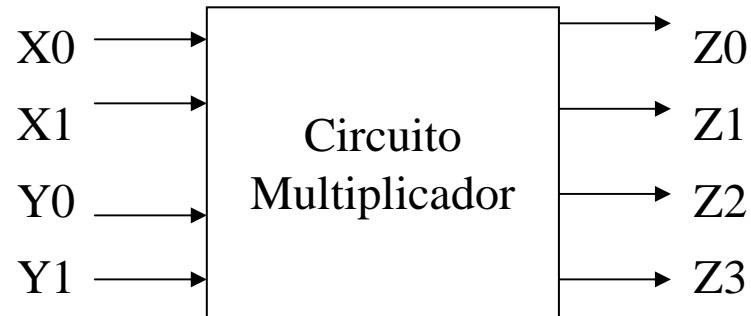
Ejemplo: **use** ieee.std_logic_1164.**all**;

Uso del paquete **std_logic_1164**
incluido en la biblioteca **ieee**

Permite el uso de todos los componentes
almacenados en el paquete



<i>Paquetes predefinidos comúnmente utilizados</i>	
Standard	
standard	<ul style="list-style-type: none"> • Contiene tipos básicos: bit, bit_vector, integer • Paquete incluido por omisión.
IEEE	
std_logic_1164	<ul style="list-style-type: none"> • Define los tipos: std_logic, std_ulogic, std_logic_vector, std_ulogic_vector • Define funciones de conversión basadas sobre estos tipos.
numeric_bit	<ul style="list-style-type: none"> • Define tipos de vectores signados y no-signados basados en el tipo bit y todos los operadores aritméticos sobre estos tipos. • Define funciones extendidas y de conversión para dichos tipos.
numeric_std	Define tipos de vectores signados y no-signados basados en el tipo std_logic. Paquete equivalente al Paquete std_logic_arith
Synopsys	
std_logic_arith	<ul style="list-style-type: none"> • Define tipos de vectores signados y no-signados, y todos los operadores aritméticos sobre estos tipos. • Define funciones extendidas y de conversión para dichos tipos.
std_logic_unsigned	• Define operadores aritméticos sobre el tipo std_ulogic_vector y los considera como operadores no-signados.
std_logic_signed	• Define operadores aritméticos sobre el tipo std_logic_vector y los considera como operadores signados.
std_logic_misc	• Define tipos, subtipos, constantes y funciones complementarios para el paquete std_logic_1164.

**Ejemplo-7**

Multiplicador de dos datos con longitudes de 2-bit
(Declaración de Entidad – Uso de Biblioteca y Paquete)

```
library ieee;  
use ieee.std_logic_1164.all;  
entity multiplica is  
    port (X0, X1, Y0, Y1: in std_logic;  
          Z3, Z2, Z1, Z0: out std_logic);  
end multiplica;
```

**arquitectura (*architecture*)**

Unidad de Diseño Secundaria que describe el comportamiento interno de una entidad.



¿Cómo? - A través de la programación de varios procedimientos que permitan que la entidad (**entity**) cumpla con las condiciones de operación o comportamiento deseadas.



Niveles de *Descripción* utilizados

Nivel Algoritmo

Nivel de Transferencia entre Registros (*RTL*)

Nivel Lógico

Nivel Compuerta

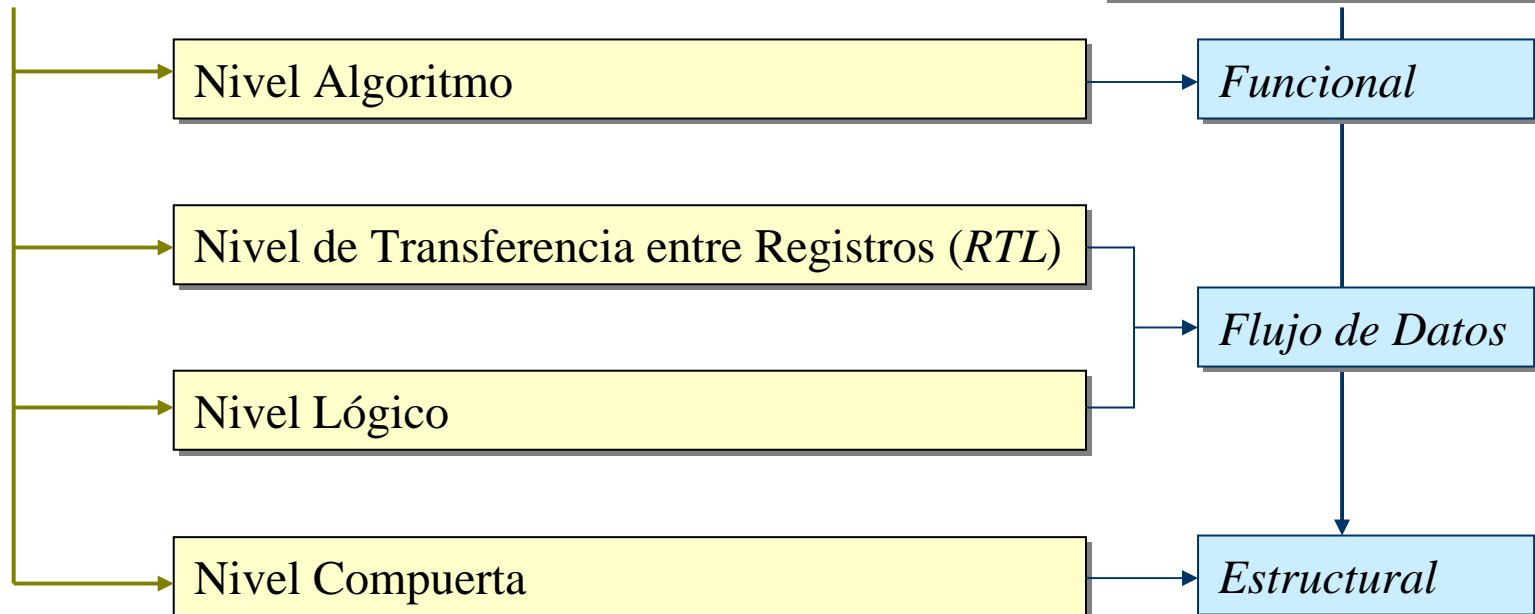
Nivel Transistor (Topología / Layout)

Estilo Programación o Modelización

Funcional

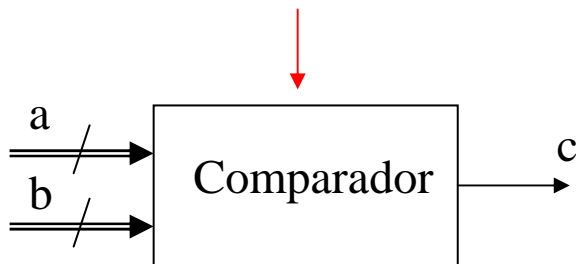
Flujo de Datos

Estructural



Funcional - En este caso, se describen las relaciones entre las entradas y salidas, sin importar la estructura o implementación física del sistema o circuito.

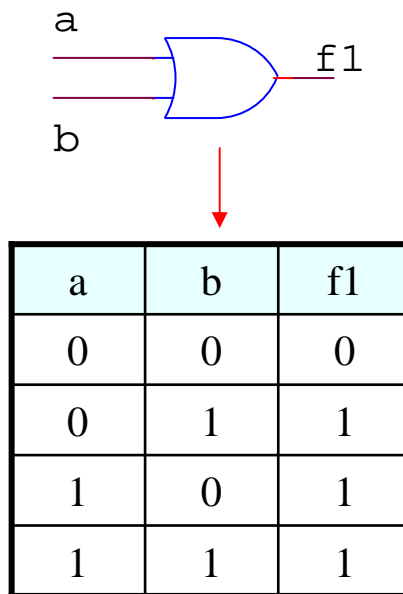
Ejemplo-8



Uso de **if-then-else**
(construcción secuencial)

si $a = b$ entonces $c = 1$
si $a \neq b$ entonces $c = 0$

Línea Nº	Arquitectura - Comparador de Igualdad de dos Datos de Long. = 2Bits
1	--Ejemplo de una descripción abstracta (funcional)
2	library ieee;
3	use ieee.std_logic_1164.all;
4	entity comp is
5	port (a,b: in bit_vector (1 downto 0);
6	c: out bit);
7	end comp;
8	architecture funcional of comp is
9	begin
10	compara: process (a,b)
11	begin
12	if a = b then
13	c <= '1';
14	else
15	c <= '0';
16	end if ;
17	end process compara;
18	end funcional;

Ejemplo-9

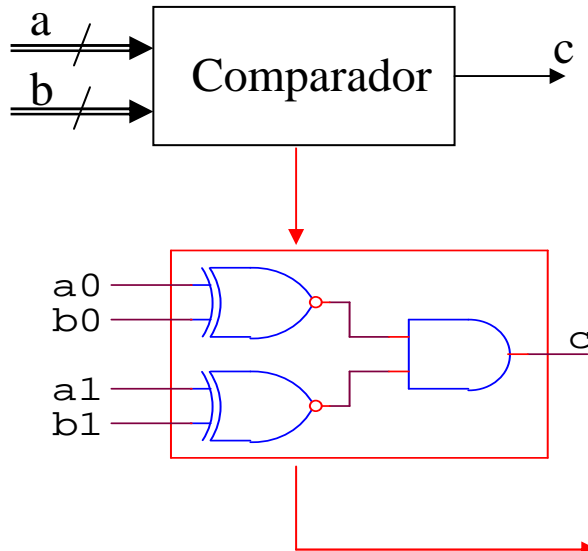
Línea Nº	Arquitectura - Compuerta OR de dos entradas
1	--Ejemplo de una descripción abstracta (funcional)
2	library ieee;
3	use ieee.std_logic_1164.all;
4	entity com_or is
5	port (a,b: in std_logic;
6	f1: out std_logic);
7	end com_or;
8	architecture funcional of com_or is
9	begin
10	process (a,b) begin
11	if (a = '0' and b='0') then
12	f1 <= '0';
13	else
14	f1 <= '1';
15	end if ;
16	end process ;
17	end funcional;



Flujo de Datos - En este caso, se describe la forma en la que los datos se pueden transferir entre los diferentes módulos operativos que constituyen la entidad (sistema o circuito)

➤ La construcción **when-else**

Ejemplo-10
Comparador
(Ejemplo-8)

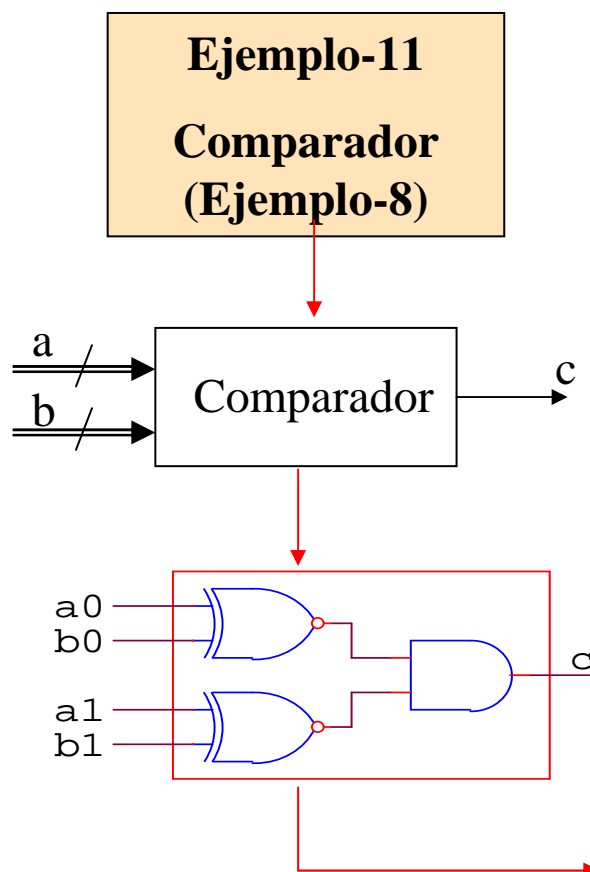


Línea Nº	Arquitectura - Comparador de Igualdad de dos Datos de Long. = 2Bits
1	--Ejemplo de una arquitectura usando when-else
2	library ieee;
3	use ieee.std_logic_1164.all;
4	entity comp is
5	port (a,b: in bit_vector (1 downto 0);
6	c: out bit);
7	end comp;
8	architecture f_datos of comp is
9	begin
10	c <= '1' when (a = b) else '0';
11	end f_datos;
12	



Flujo de Datos - En este caso, se describe la forma en la que los datos se pueden transferir entre los diferentes módulos operativos que constituyen la entidad (sistema o circuito)

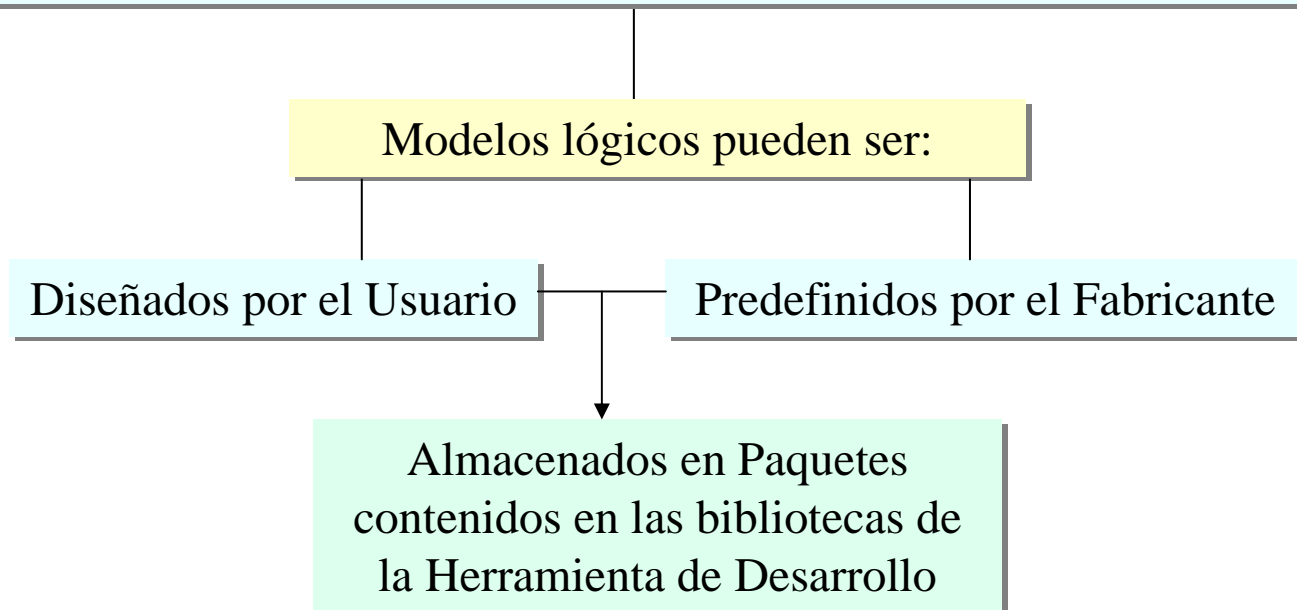
➤ Uso de ecuaciones booleanas

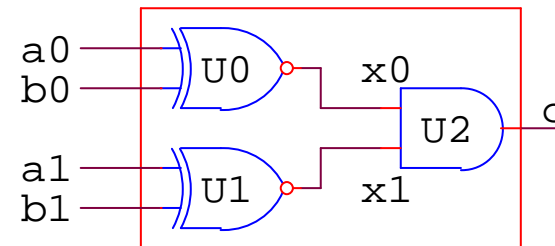


Línea Nº	Arquitectura - Comparador de Igualdad de dos Datos de Long. = 2Bits
1	--Ejemplo de una arquitectura usando ecs. booleanas
2	library ieee;
3	use ieee.std_logic_1164. all ;
4	entity comp is
5	port (a,b: in bit_vector (1 downto 0);
6	c: out bit);
7	end comp;
8	architecture booleana of comp is
9	begin
10	c <= (a(1) xnor b(1)) and (a(0) xnor b(0));
11	end booleana;
12	



Estructural - En este caso, el comportamiento de un sistema o circuito es descrito mediante modelos lógicos establecidos de los componentes que conforman al sistema o circuito, como son: Compuertas, Sumadores, Contadores, etc.



**Ejemplo-12****Comparador
(Ejemplo-8)**

La instrucción **signal** se declara dentro de la arquitectura y no en la entidad, debido a que no representan a una terminal externa y sólo se utilizan para conectar bloques de manera interna a la entidad.

Línea Nº	Arquitectura - Comparador de Igualdad de dos Datos de Long. = 2Bits
1	library ieee;
2	use ieee.std_logic_1164. all ;
3	use work.compuertas. all ;
4	entity comp is
5	port (a,b: in bit_vector (0 to 1);
6	c: out bit);
7	end comp;
8	architecture estructural of comp is
9	signal x: bit_vector (0 to 1);
10	begin
11	U0: xnor2 port map (a(0), b(0), x(0));
12	U1: xnor2 port map (a(1), b(1), x(1));
13	U2: and2 port map (x(0), x(1), c);
14	end estructural;



En resumen, se puede decir que:

La descripción **funcional** se basa principalmente en el uso de procesos y de declaraciones *secuenciales*. Esta descripción es similar a la hecha en un lenguaje de programación de alto nivel, por su alto nivel de abstracción.

Más que especificar la estructura o la forma en que se deben conectar los componentes de un diseño, nos limitamos a describir su comportamiento.

Una descripción **funcional** consiste de una serie de instrucciones, que ejecutadas *secuencialmente*, modelan el comportamiento del circuito.

La ventaja de este tipo de descripción, es que no se requiere enfocar a un nivel de compuerta para implementar un diseño.

En **VHDL** una descripción **funcional** necesariamente implica el uso de por lo menos un bloque **PROCESS (if-then-else)**

*Definición de instrucción secuencial:*

Las instrucciones secuenciales son aquellas que son ejecutadas serialmente, una después de otra. La mayoría de los lenguajes de programación, como C o Pascal, utilizan este tipo de instrucciones.

En **VHDL** las instrucciones *secuenciales* son implementadas únicamente dentro del bloque **PROCESS**.

COMENTARIO:

Dentro de una arquitectura en **VHDL**, no existe un orden específico de ejecución de las asignaciones. El orden en el que las instrucciones son ejecutadas depende de los eventos ocurridos en las señales, similar al funcionamiento del circuito.



Descripción por flujo de datos:

La descripción por flujo de datos indica la forma en que los datos se pueden transferir de una señal a otra sin necesidad de declaraciones *secuenciales*.

Este tipo de descripciones permite definir el flujo que tomarán los datos entre módulos encargados de realizar operaciones: **when-else**.

Esta forma de descripción, puede realizarse también mediante ecuaciones booleanas, en donde se emplean los operadores correspondientes: **or**, **and**, **nand**, **nor**, **xor**, **xnor**.

Descripción estructural:

Este tipo de descripción basa su comportamiento en modelos lógicos establecidos (*compuertas, sumadores, contadores, etc.*).

El usuario puede diseñar estas estructuras y guardarlas para su uso posterior o tomarlas de los *paquetes* contenidos en las *librerías* de diseño del software que se esté utilizando.



Comparación entre los estilos de diseño.

El estilo de diseño utilizado en la programación del circuito depende del diseñador y de la complejidad del proyecto. Por ejemplo, un diseño puede describirse por medio de **ecuaciones booleanas**, pero si es muy extenso quizá sea más apropiado emplear **estructuras** jerárquicas para dividirlo.

Ahora bien, si se requiere diseñar un sistema cuyo funcionamiento dependa sólo de sus entradas y salidas, es conveniente utilizar la descripción **funcional**, la cual presenta la ventaja de requerir menos instrucciones y el diseñador no necesita un conocimiento previo de cada componente del circuito.



Las instrucciones *concurrentes* (*flujo de datos y estructural*) se utilizan fuera de un bloque **PROCESS**, a diferencia de las instrucciones *secuenciales*, que únicamente se utilizan dentro del bloque **PROCESS**.

Entonces, las descripciones se pueden distinguir entre **secuenciales** (*funcional*) y **concurrentes** (*flujo de datos y estructural*).



Para la siguiente declaración:

```
library ieee;  
use ieee.std_logic_1164.all;  
entity selección is port (  
  x: in std_logic_vector(0 to 3);  
  f: out std_logic);  
end selección;
```

Indicar:

nombre de la entidad: _____

los puertos de entrada: _____

los puertos de salida: _____

el tipo de dato: _____



Para la siguiente declaración:

```
library ieee;  
use ieee.std_logic_1164.all;  
entity seleccion is port (  
  x: in std_logic_vector(0 to 3);  
  f: out std_logic);  
end selección;
```

Indicar:

nombre de la entidad: _____

seleccion

los puertos de entrada: _____

x

los puertos de salida: _____

f

el tipo de dato: _____

std_logic_vector



Señale cuál de los siguientes identificadores son correctos o incorrectos:

1logico_____

Desp_laza_____

con_trol_____

N_ivel_____

Pagina_____

architecture_____

registro_____


S_uma#_____

2Suma_____

Res__ta_____



Señale cuál de los siguientes identificadores son correctos o incorrectos:

1logico___  ___

Desp_laza___  _


con_trol___  ____

N_ivel_____  _


Pagina___  ____

architecture_  _

registro___  ____

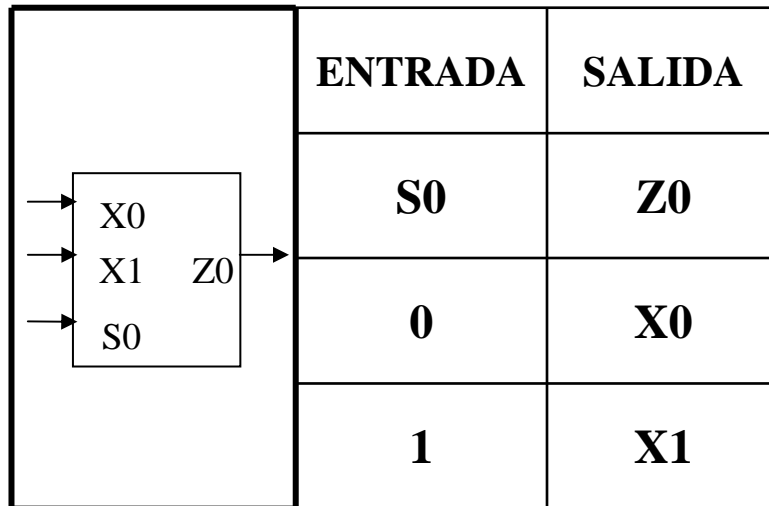
S_uma#_____  _

2Suma___  ____

Res__ta_____  _

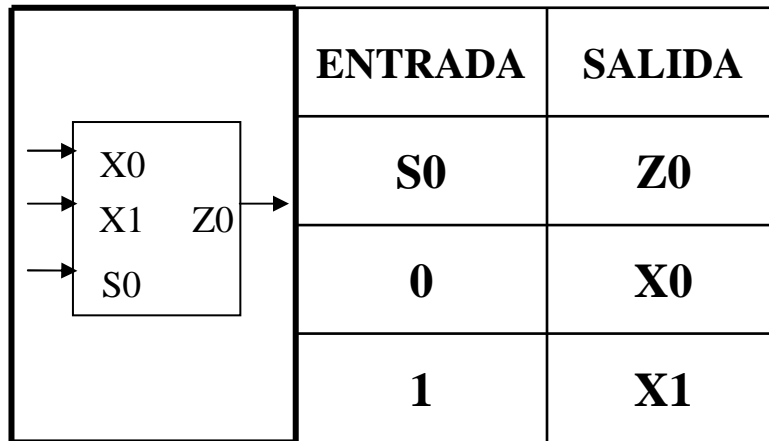


Describir la siguiente función por VHDL.



Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

MULTIPLEXOR



ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL temp: bit_vector (2 to 0);

BEGIN

```

z0 <= '0' WHEN temp = "000" ELSE
      '0' WHEN temp = "001" ELSE
      '1' WHEN temp = "010" ELSE
      '1' WHEN temp = "011" ELSE
      '0' WHEN temp = "100" ELSE
      '1' WHEN temp = "101" ELSE
      '0' WHEN temp = "110" ELSE
      '1' ;

```

```

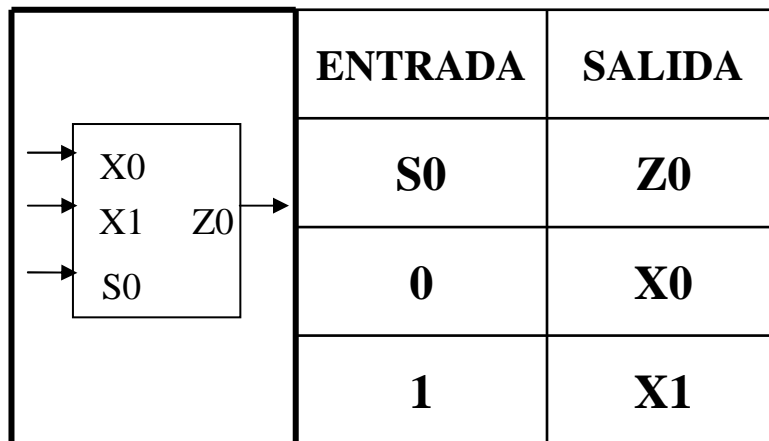
temp <= so & x0 & x1; -- concatenación
                      -- de las entradas en un
                      -- solo bus

```

END data_flow;

Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Para la primera descripción que se mostrará, se empleará el código de la tabla extendida, resultando en el siguiente listado VHDL.



ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL temp: bit_vector (2 to 0);

BEGIN

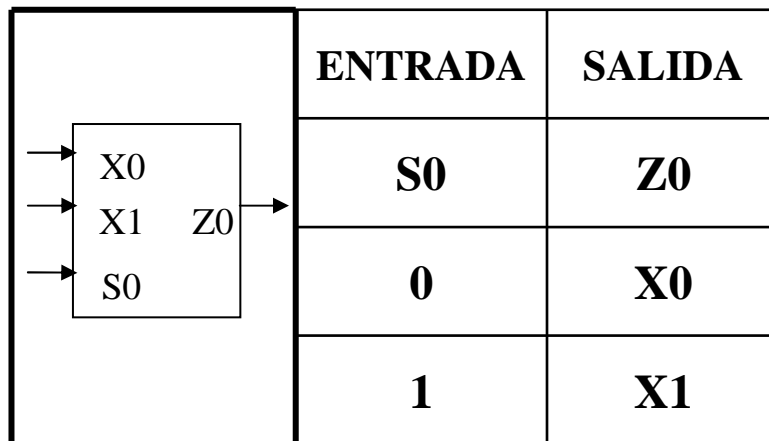
```
z0 <= '0' WHEN temp = "000" ELSE
      '0' WHEN temp = "001" ELSE
      '1' WHEN temp = "010" ELSE
      '1' WHEN temp = "011" ELSE
      '0' WHEN temp = "100" ELSE
      '1' WHEN temp = "101" ELSE
      '0' WHEN temp = "110" ELSE
      '1' ;
```

```
temp <= s0 & x0 & x1; -- concatenación
                      -- de las entradas en un
                      -- solo bus
```

END data_flow;

Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Los valores asignados al tipo `bit_vector` deber ser especificados con comillas dobles (“_”) y los valores asignados al tipo `bit simple`, son asignados con comillas simples (‘_’).



Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL temp: bit_vector (2 **to** 0);

BEGIN

```

z0 <= '0' WHEN temp = "000" ELSE
      '0' WHEN temp = "001" ELSE
      '1' WHEN temp = "010" ELSE
      '1' WHEN temp = "011" ELSE
      '0' WHEN temp = "100" ELSE
      '1' WHEN temp = "101" ELSE
      '0' WHEN temp = "110" ELSE
      '1';

```

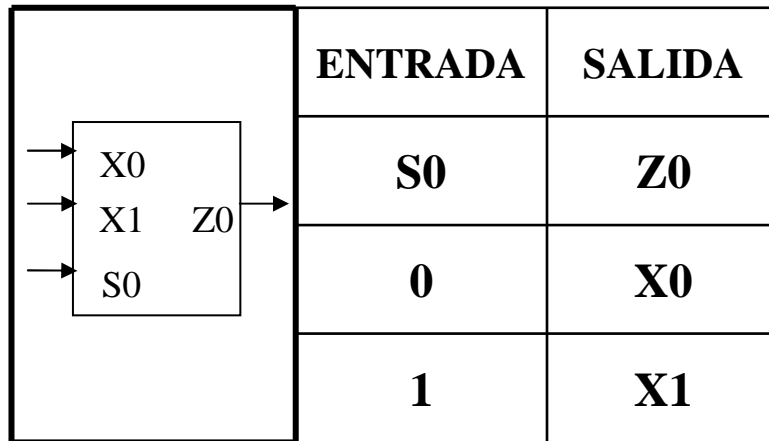
```

temp <= s0 & x0 & x1; -- concatenación
                      -- de las entradas en un
                      -- solo bus

```

END data_flow;

Se empleó el objeto de datos **SIGNAL** para crear el bus “temp” y concatenar “s0”, “x0” y “x1” en un solo objeto de datos y así facilitar la descripción.



Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL temp: bit_vector (2 **to** 0);

BEGIN

```

z0 <= '0' WHEN temp = "000" ELSE
      '0' WHEN temp = "001" ELSE
      '1' WHEN temp = "010" ELSE
      '1' WHEN temp = "011" ELSE
      '0' WHEN temp = "100" ELSE
      '1' WHEN temp = "101" ELSE
      '0' WHEN temp = "110" ELSE
      '1' ;

```

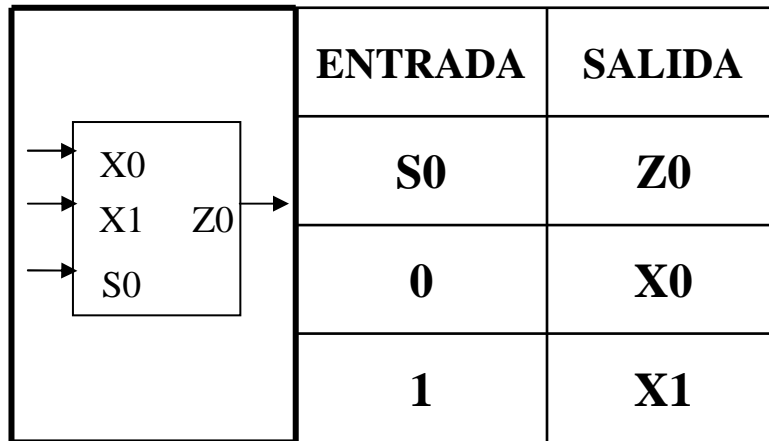
```

temp <= s0 & x0 & x1; -- concatenación
                      -- de las entradas en un
                      -- solo bus

```

END data_flow;

¿Qué tipo de descripción se realizó en este multiplexor?



Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL temp: bit_vector (2 **to** 0);

BEGIN

```

z0 <= '0' WHEN temp = "000" ELSE
      '0' WHEN temp = "001" ELSE
      '1' WHEN temp = "010" ELSE
      '1' WHEN temp = "011" ELSE
      '0' WHEN temp = "100" ELSE
      '1' WHEN temp = "101" ELSE
      '0' WHEN temp = "110" ELSE
      '1' ;

```

```

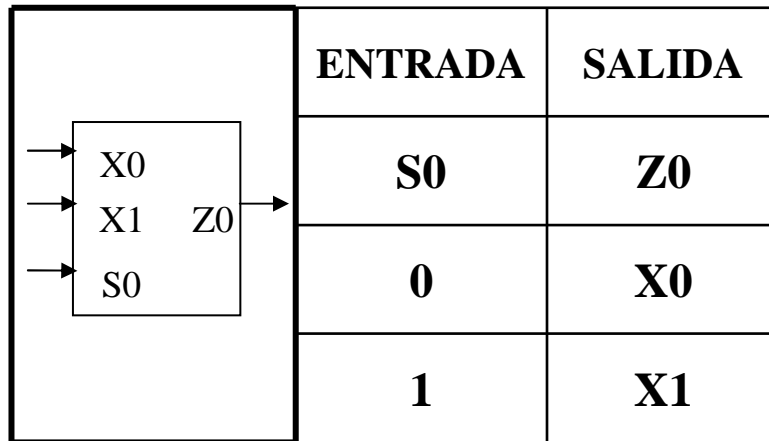
temp <= s0 & x0 & x1; -- concatenación
                      -- de las entradas en un
                      -- solo bus

```

END data_flow;

¿Qué tipo de descripción se realizó en este multiplexor?

FLUJO DE DATOS



```

ENTITY multiplexor IS
  PORT (s0, x0, x1: IN bit;
        z0: OUT bit);
END multiplexor;

```

```

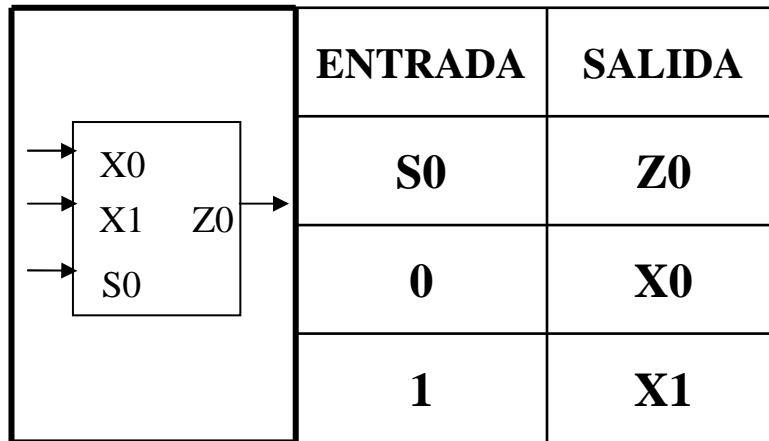
ARCHITECTURE data_flow OF multiplexor IS
BEGIN
  z0 <= x0 WHEN s0 = '0' ELSE x1
END data_flow;

```

Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

La siguiente descripción se deriva de la tabla simplificada, ya que se ve que Z0 depende solamente del estado de S0.

Por lo tanto, la descripción resulta más sencilla.



```

ENTITY multiplexor IS
  PORT (s0, x0, x1: IN bit;
        z0: OUT bit);
END multiplexor;

```

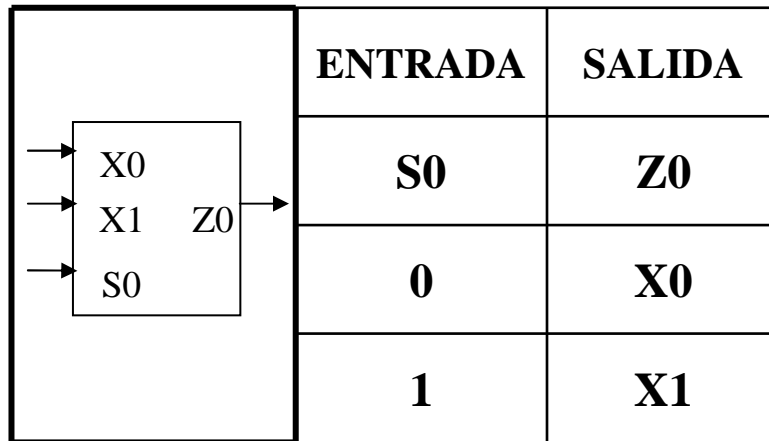
```

ARCHITECTURE data_flow OF multiplexor IS
BEGIN
  z0 <= x0 WHEN s0 = '0' ELSE x1
END data_flow;

```

Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

¿Qué tipo de descripción se realizó en este multiplexor?



```

ENTITY multiplexor IS
  PORT (s0, x0, x1: IN bit;
        z0: OUT bit);
END multiplexor;

```

```

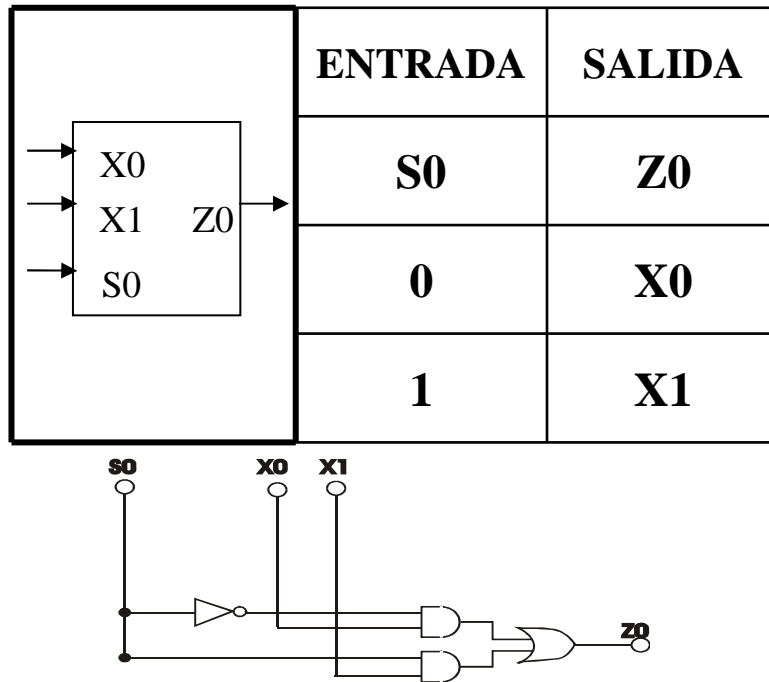
ARCHITECTURE data_flow OF multiplexor IS
BEGIN
  z0 <= x0 WHEN s0 = '0' ELSE x1
END data_flow;

```

Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

¿Qué tipo de descripción se realizó en este multiplexor?

FLUJO DE DATOS



Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL not_s0, and1, and2: bit;

BEGIN

z0 <= and1 **OR** and2;

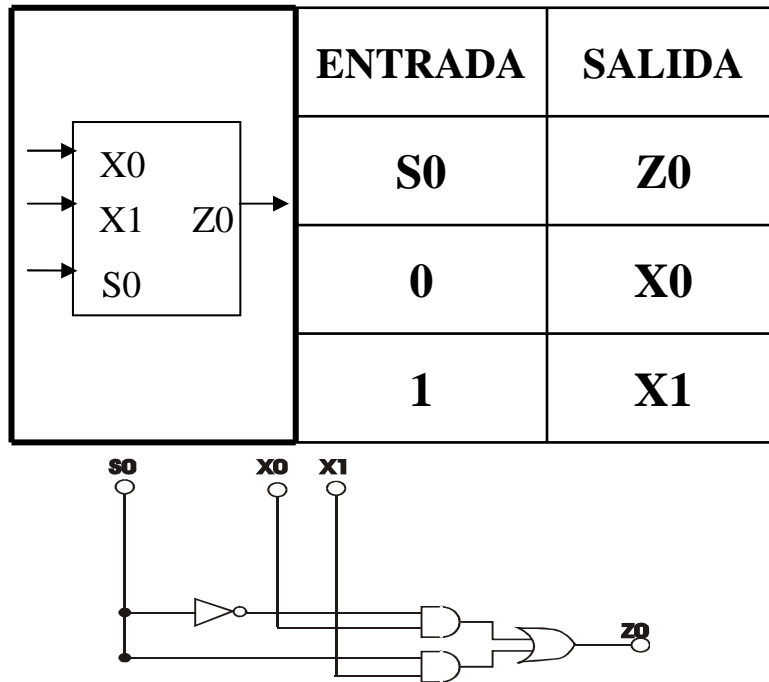
and1 <= not_s0 **AND** x0;

not_s0 <= **NOT** s0;

and2 <= s0 **AND** x1;

END data_flow;

La descripción mostrada a continuación, hace uso del siguiente diagrama explícito del multiplexor.



Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL not_s0, and1, and2: bit;;

BEGIN

z0 <= and1 **OR** and2;

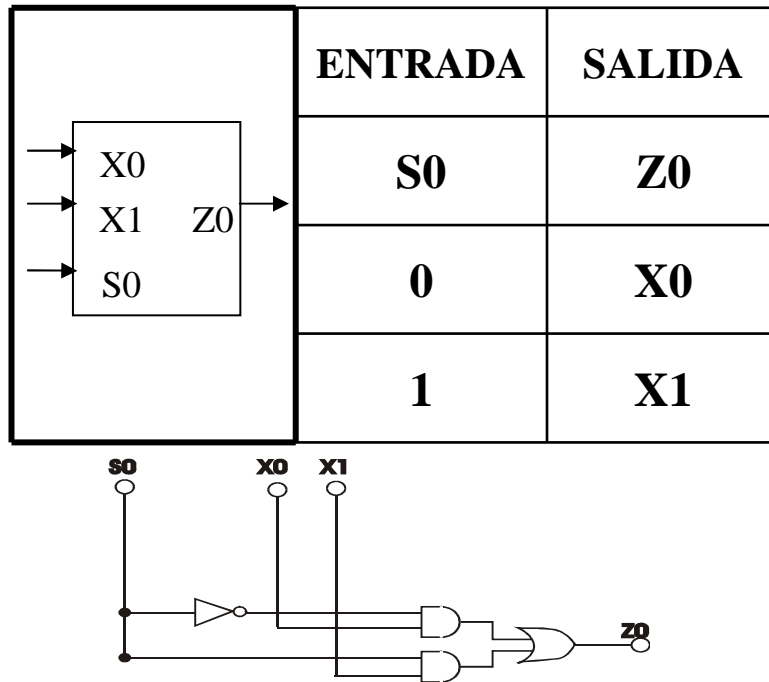
and1 <= not_s0 **AND** x0;

not_s0 <= **NOT** s0;

and2 <= s0 **AND** x1;

END data_flow;

¿Qué tipo de descripción se realizó en este multiplexor?



Entradas			Salidas
S0	X0	X1	Z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

ENTITY multiplexor **IS**

PORT (s0, x0, x1: **IN** bit;
z0: **OUT** bit);

END multiplexor;

ARCHITECTURE data_flow **OF** multiplexor **IS**

SIGNAL not_s0, and1, and2: bit;;

BEGIN

z0 <= and1 **OR** and2;

and1 <= not_s0 **AND** x0;

not_s0 <= **NOT** s0;

and2 <= s0 **AND** x1;

END data_flow;

¿Qué tipo de descripción se realizó en este multiplexor?

**FLUJO DE DATOS CON ECUACIONES
BOOLEANAS.**