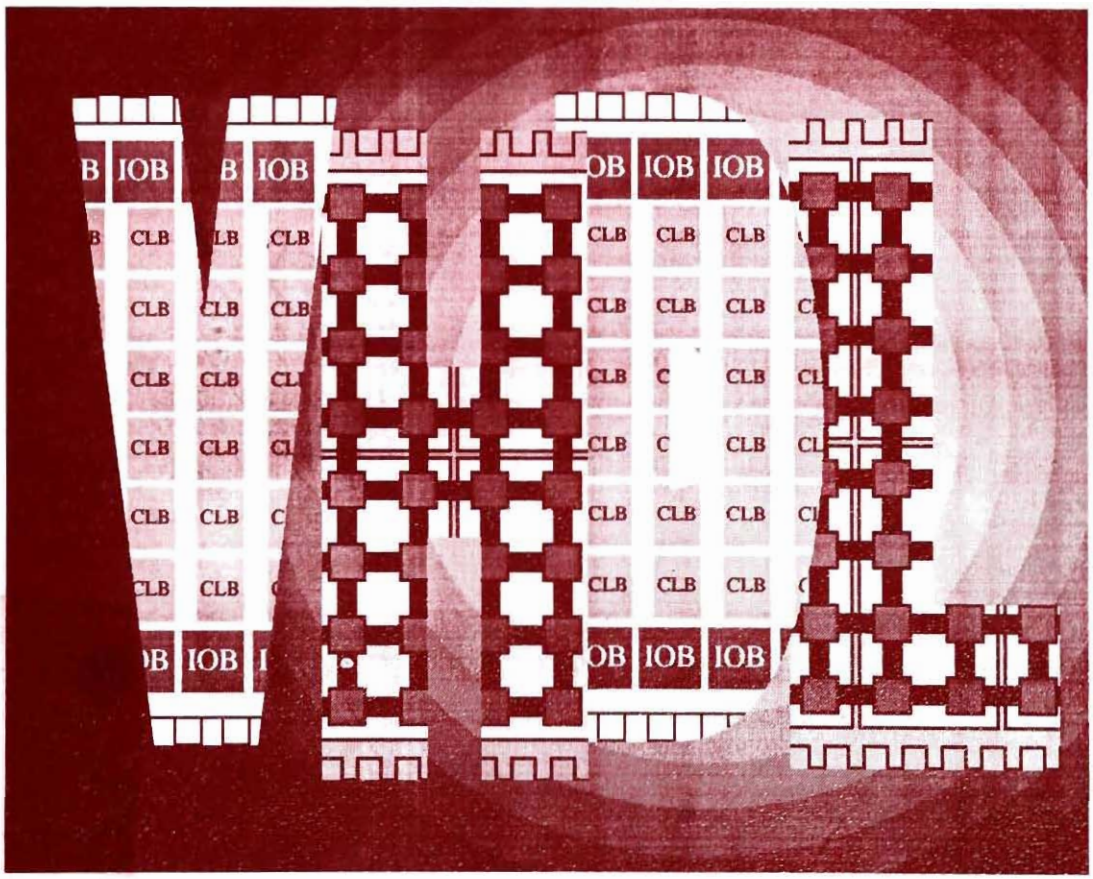
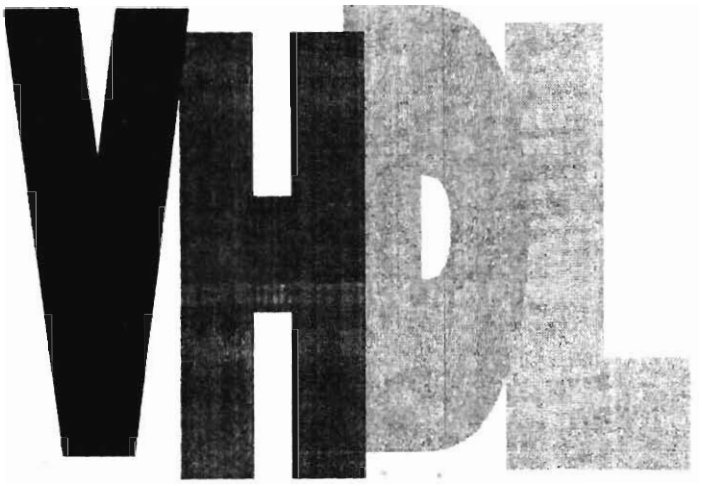
Introducción al lenguaje

Victor Gonzalo Rodriguez Tapia
Maria Antonieta Garcia Galván
José Francisco Cosme Aceves
Francisco Javier Sánchez Rangel



UNIVERSIDAD AUTONOMA METROPOLITANA
Azcapotzalco

Introducción al lenguaje



Introducción
al
lenguaje

Este material fue dictaminado y aprobado por el
Consejo Editorial de la División de Ciencias Básicas
e Ingeniería, el 20 de septiembre de 2000.

Introducción al lenguaje

Víctor Gonzalo Rodríguez Tapia
María Antonieta García Galván
José Francisco Cosme Aceves
Francisco Javier Sánchez Rangel

UNIVERSIDAD AUTÓNOMA METROPOLITANA
Casa abierta al tiempo Azcapotzalco

Coordinación de Extensión Universitaria
Sección de Producción y Distribución Editoriales

División de Ciencias Básicas e Ingeniería
Departamento de Ciencias Básicas

UAM-AZCAPOTZALCO

Rectora
Mtra. Mónica de la Garza Malo
Secretario
Lic. Guillermo Ejea Mendoza
Coordinador de Extensión Universitaria
Lic. Enrique López Aguilar
Jefa de la Sección de Producción y Distribución Editoriales
Lic. Silvia G. Lona Perales

ISBN 970-654-755-X

Corrección
Marisela Juárez Capistrán
Portada y Preliminares
d.c.g. Hugo Adrián Ábrego García

Universidad Autónoma Metropolitana
Unidad Azcapotzalco
Av. San Pablo 180
Col. Reynosa Tamaulipas
Delegación Azcapotzalco
C.P. 02200
México, D.F.
Tel. 5318-9222 y 23
Fax 5318-9222

Primera edición, 2001
Hecho en México

PRÓLOGO

El objetivo principal de esté trabajo es dar un panorama general de lo que es y para qué se usa el VHDL, así como analizar los pasos de la síntesis e implementación para los dispositivos lógicos programables, usando los dispositivos Xilinx en particular.

Después de la lectura del trabajo es de esperarse que los lectores no puedan por sí mismos escribir código, sino que tan sólo tengan una visión global de las potencialidades del VHDL para producir hardware digital: tener un primer contacto con algunas características del lenguaje, comprender algunas de sus limitaciones actuales, sus tendencias de desarrollo y de que conozcan algunos de los dispositivos en los que preferentemente se plasman fisicamente los diseños, y cual es el flujo de diseño típico de la herramienta *Foundation Express Series Software*.

El material aqui presentado deberá servir como la primera lectura sobre el VHDL, una especie de "por aquí empezar el estudio del VHDL". Se supone que los lectores deberán estar familiarizados con las técnicas de diseño digital hasta el nivel de transferencia entre registros o RTL, y además poseer buenos fundamentos de programación con lenguajes de alto nivel como el C 0 PASCAL.

El trabajo justifica su existencia actualmente debido a que los tópicos por él cubiertos son sumamente novedosos en este momento, a la escasez de bibliografía disponible en español, ya que a los estudiantes de electrónica en diseño digital en la UAM-Azc se les ha pedido como parte de su formación el cursar tanto teórica como prácticamente un curso de diseño de sistemas digitales mediante software con lenguajes de descripción de hardware y en especial con el VHDL.

El trabajo está dividido en tres grandes partes; constan a su vez de capítulos, cada uno de ellos subdividido en varias lecciones o tópicos generales presentados en forma de láminas, esto con la finalidad de tener un mayor impacto visual con los lectores, haciendo según creemos, más amena la exposición.

En la primera parte 'conceptos básicos del lenguaje' trataremos sobre generalidades acerca del lenguaje, sus áreas de aplicación, sus principales construcciones y el modelo de simulación VHDL. Consta de seis capítulos, siendo el primero 'introducción al lenguaje y sus aplicaciones' en donde se dará un panorama general de lo qué es el VHDL y qué papel juega dentro del flujo de diseño de los sistemas digitales. El segundo capítulo 'tipos de datos y señales' estará dedicado a estudiar los diferentes portadores de datos y sus distintos tipos de valores que pueden llevar. Para el capítulo tres 'operadores VHDL' se deja el estudio de los operadores predefinidos en el lenguaje. El capítulo cuatro 'postulados secuenciales y concurrentes' se dedica a mostrar cómo es la estructura de los archivos *.vhd, en qué posición dentro de ellos van los postulados concurrentes y los secuenciales, y sobre todo la diferencia cualitativa entre ellos. Dada la importancia para la síntesis de los procesos VHDL, será en el capítulo quinto en donde se verán con mayor detalle los postulados secuenciales, ya que estos son propios de dichos procesos. En el

capítulo que cierra la primera parte se trata sobre algunas cuestiones relacionadas con la síntesis del VHDL.

La segunda parte 'información de VHDL para Xilinx' versará sobre cuestiones prácticas del VHDL para cuando el código se escriba ex profeso para PLDs de Xilinx, ya que como se verá en algunas ocasiones conviene perder generalidad propia del VHDL y ganar eficiencia al escribir con determinados estilos cuando se apunta a un dispositivo programable con una arquitectura específica. Los capítulos que componen esta parte son tres. El capítulo siete trata de dar 'un panorama general' sobre la síntesis del VHDL para los dispositivos programables en general y los FPGA de Xilinx en particular. El capítulo octavo es donde se hablará de los diferentes estilos de escritura del VHDL para sacar provecho de las diferentes arquitecturas de los dispositivos Xilinx. En el noveno analizaremos los diferentes pasos a seguir en una situación de diseño típica de diseños VHDL-top desde la herramienta de entrada del código, la síntesis, la simulación funcional, la implementación en sus diferentes fases, la generación del archivo de configuración y finalmente la programación misma del chip.

En la tercera parte llamada 'Información básica sobre los proveedores de software' versará sobre algunas cuestiones a considerar cuando se trata de seleccionar alguna herramienta de síntesis y/o de simulación desde alguno de los muchos proveedores posibles y de acuerdo con las necesidades actuales y a mediano plazo del usuario final. Esta parte esta formada por un solo capítulo, el décimo 'cuestiones sobre el uso del VHDL para diseño lógico programable' y se hacen notar un conjunto de cuestiones que consideramos esenciales de tomar en cuenta para cuando se está en la disyuntiva de cuál sintetizador, cuál simulador y bajo que plataforma trabajar.

La presentación está pensada para emplearse preferentemente con medios visuales como la PC, o mediante una copia dura de ellas, aunque debe tenerse en cuenta que el trabajo consta de más de 550 láminas. La información aquí tratada es de carácter general y no es una presentación formal del curso de VHDL que se ha empezado a impartir en la licenciatura en electrónica en la UAM-Azc, aunque claro está se abordan muchas más cuestiones que las del curso, aunque de una manera tan sólo con carácter cualitativo, por lo que consideramos indispensable que los alumnos que se están entrenando como diseñadores de hardware digital con las novisimas herramientas del diseño VHDL deben de empezar leyendo este trabajo, e ir releyendo conforme avanzan en su curso, ya que al paso del tiempo se les harán claros muchos de los conceptos aquí manejados desde un principio para tratar de dar un panorama general.

Este trabajo es un primer resultado del grupo de investigación de Arquitectura de computadoras del área de Sistemas Digitales del Dpto. de Electrónica de la UAM-Azc, mediante el cuál en un principio los integrantes del grupo obtuvimos la intelección de la metodología del diseño digital mediante el VHDL a partir de una escasa bibliografía y qué surgió ante la necesidad del uso de los dispositivos lógicos programables como chips destino de sus diseños. Es de notar que nuestra experiencia en diseño digital se circunscribía al diseño esquemático, por lo qué buena parte del tiempo lo dedicamos a diseñar mediante esta técnica, hasta que "descubrimos" que la metodología seguida hasta aquí era sencillamente ineficiente y que mediante software se podía moldear cuál

plastilina el hardware digital. Nos aplicamos desde entonces a aprender el VHDL como nuestra herramienta de diseño principal y a tener como dispositivos destino a los FPGAs de modo natural.

Una vez cumplida la primera misión de este trabajo, era nuestra propia comprensión del VHDL, se fué a los estantes, hasta que se vio la necesidad de introducir la nueva herramienta como parte de la curricula de nuestros alumnos de electrónica, materia en la que consideramos somos los primeros en hacerlo en las universidades mexicanas, y por lo tanto nos vimos en la obligación de desempolvar el trabajo, darle una revisión y corrección de estilo y lanzarlo a su publicación para ayuda de los profesores y alumnos que enseñan o cursan tópicos de VHDL.

ATENTAMENTE

Víctor Gonzalo Rodríguez Tapia
María Antonieta García Galván
José Francisco Cosme Aceves
Francisco Javier Sánchez Rangel

ÍNDICE

Primera parte

'Conceptos básicos del lenguaje'

CAPÍTULO

Segunda parte

'Información de VHDL para XILINX'

CAPÍTULO

ÍNDICE

Tercera parte

'Información básica sobre los proveedores de software'

CAPÍTULO

PRIMERA PARTE
Conceptos básicos del lenguaje

1. Introducción al uso del lenguaje VHDL y sus aplicaciones

1.1.0 ¿De que trata la siguiente presentación?

Introducción al uso del lenguaje

VHDL

y sus aplicaciones

Mediante esta primera presentación aprenderemos cómo se usa el VHDL en el diseño de proyectos electrónicos para sistemas digitales, y a comprender algunas de las principales construcciones del lenguaje.

1.1.1 De que trata la presente lección?

Aplicación: que es VHDL, estilos de uso

Aplicaciones del VHDL

Primero veremos algunas cuestiones sobre las aplicaciones del VHDL, cubriendo brevemente sus fundamentos, algunas de sus bondades y defectos, y un resumen de los diferentes estilos de uso del lenguaje.

1.1.2 ¿De que trata la presente lección?

Conceptos tras el lenguaje, principales construcciones

Conceptos, Construcciones

Enseguida se discutirán los principales conceptos que encierra el lenguaje, luego resumiremos las principales construcciones disponibles. Esto con el fin de obtener un panorama general de los diferentes tipos de facilidades disponibles, asi como de la terminología más importante.

1.1.3 ¿De que trata la presente lección?

Aplicación: que es VHDL, estilos de uso...

Panorama general

Finalmente cubriremos como se usan las principales construcciones del lenguaje en situaciones tipicas de diseño para tener una visión global, de modo que conforme avancemos los detalles puedan ir siendo introducidos según se vayan requiriendo dentro de la presentación.

1.2.0 ¿Qué significa VHDL?

¿Qué es VHDL?

¿Qué quiere decir VHDL?

El primer tópico trata la cuestion más básica por comprender: ¿qué es o que significa VHDL?

1.2.1 ¿Qué significa VHDL?

VHSIC: Hardware Description Language

VHSIC	**L**enguaje de
Hardware	**D**escripción de **H**ardware para
Description	Circuitos Integrados de muy Alta
Language.	**V**elocidad

Es un acrónimo compuesto: la **V** es el acrónimo de VHSIC (*Very High Speed Integrated Circuit*) y **HDL** es otro acrónimo de *Hardware Description Language.*

1.2.2 ¿Qué es VHDL?

Desarrollado a principios de los 80's

- Desde principios de los 80's
- Para descripciones independientes de la implementación

El lenguaje VHDL se desarrolló en los 80's para describir sistemas electrónicos, buscando fueran una descripción independiente del método de implementación. Fue desarrollado bajo pedido del gobierno de USA.

1.2.3 ¿Qué es VHDL?

IEEE 1076 :1987, & 1993

IEEE 1076:1987

Nuevo: 1993

Las construcciones del VHDL fueron definidas como **estándar** con número 1076 por la IEEE en 1987. Una nueva revisión fue definida en 1993, a la cuál se añadieron algunas nuevas capacidades, removiéndose igualmente algunas ambiguedades, y permaneciendo compatible hacia arriba.

1.2.4 ¿Qué significa VHDL?

Esta lección soporta ambas versiones

Versiones
1987 & 1993

En el caso de las construcciones VHDL más empleadas en la **síntesis** de Hardware, las diferencias entre las dos versiones es mínima. Todas las construcciones usadas en adelante son validas para ambas versiones.

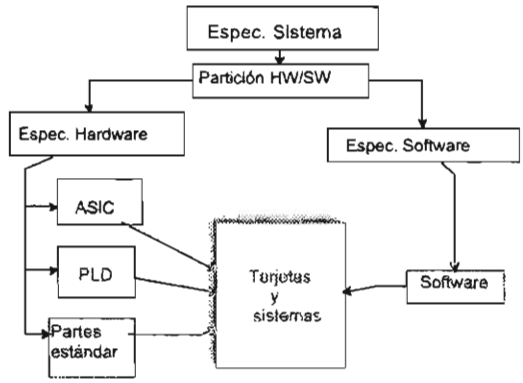
1.3.0 Áreas de aplicación

Áreas de aplicación

Veamos los diferentes pasos más generales incluidos en el diseño de cualesquier sistema electrónico, y a partir de ellos veremos en donde están las principales áreas de aplicación del VHDL...

1.3.1 Áreas de aplicación

Proceso de diseño electrónico

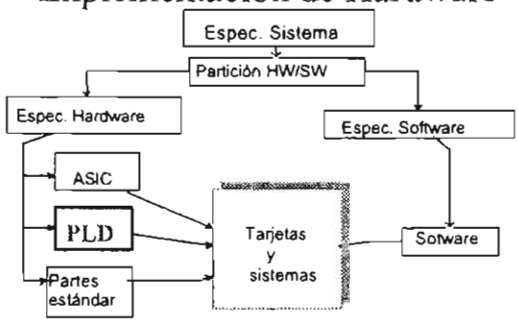


Pasos de diseño en los Sistemas Electrónicos

El diagrama muestra un proceso general, completo, del diseño de un sistema electrónico, desde la especificación del sistema, a través de particionar el hardware y software, hasta la especificación e implementación del hardware y software.

1.3.2 Áreas de aplicación

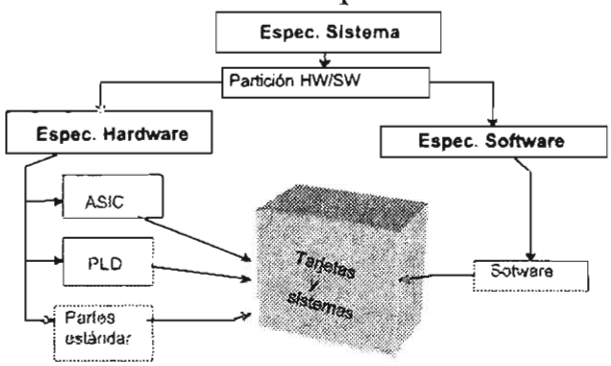
Implementación de Hardware



A inicios de los 90's, el VHDL fue usado principalmente para diseñar ASIC complejos, usando herramientas de síntesis que automáticamente crean y optimizan la implementación. A mediados de esos años la corriente principal del uso del VHDL se dirigió hacia la síntesis de diseño de lógica programable PLD.

1.3.3 Áreas de aplicación

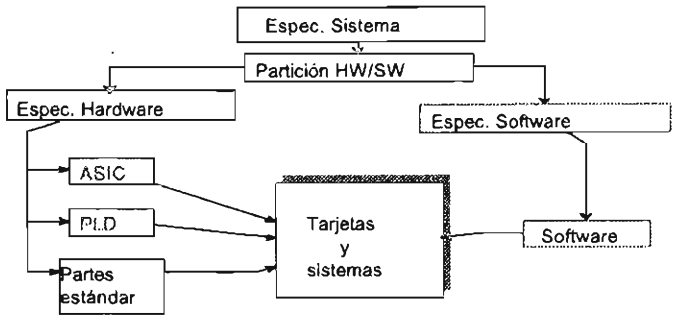
Modelado de especificaciones



Actualmente existe también un incremento en el uso de VHDL en el modelado de especificaciones, tanto para la parte de hardware del sistema, como para el mismo sistema completo. Ello requiere que VHDL sea escrito en diferentes estilos según sea la aplicación.

1.3.4 Áreas de aplicación

Nuestro enfoque se centrará en: VHDL para síntesis, diseño de Lógica Programable y ASIC (PLD / ASIC)



Esta presentación explica los conceptos principales del lenguaje VHDL, y cómo escribir VHDL para síntesis, en un estilo el cual es adecuado para diseñar tanto ASICs, como dispositivos de lógica programable o PLDs.

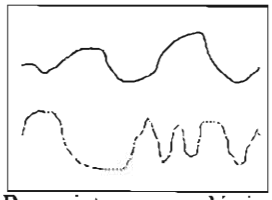
1.4.0 Limitaciones del lenguaje VHDL

Limitaciones....

Resumamos algunas limitaciones del lenguaje VHDL, antes de ver los diferentes estilos de uso de dicho lenguaje, para no crear falsas expectativas.

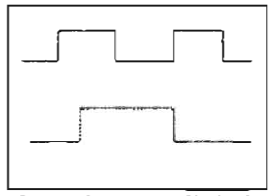
Descripciones analógicas

Analógico



Para sistemas analógicos
NO es adecuado actualmente,
aunque pronto lo será.

Digital



Para sistemas digitales
SI es una herramienta
adecuada.

VHDL es un lenguaje principalmente adecuado para usarse en el diseño digital. Tiene capacidades limitadas en el área analógica, más sin embargo existe una gran cantidad de trabajo tendiente a alcanzar construcciones estándar para versiones analógicas.

IEEE 1076 define lenguaje: construcciones y su sintaxis, no estilos de uso

construcciones & sintaxis

VHDL
espec.
IEEE
1076

sintaxis

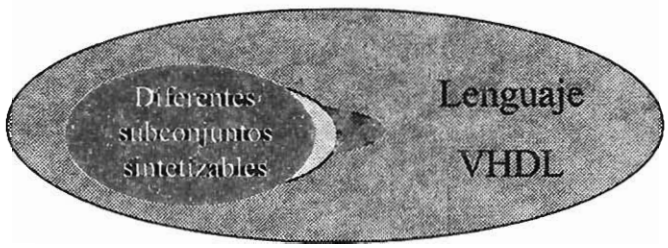
otros estándares
definen el estilo

Estilo A
Estilo
IEEE
¿¿¿¿¿¿¿

estilos de uso

El estándar IEEE 1076 define las construcciones del lenguaje y su sintaxis sin describir ningun estilo de cómo debe ser usado en el diseño de proyectos. Existen otros estándares para definir 'estilos consistentes' de uso del lenguaje en ciertas áreas específicas.

Soporte para síntesis del lenguaje VHDL



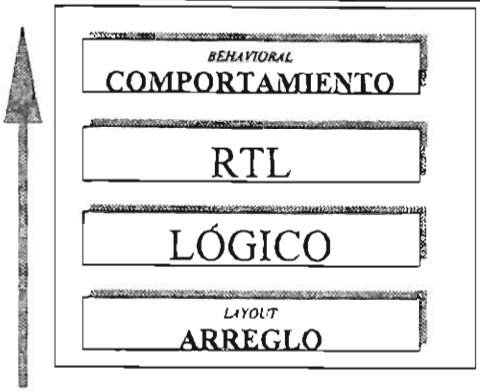
Únicamente es posible sintetizar lógica desde un subconjunto de las contrucciones del VHDL. Actualmente, cada herramienta define su propio subconjunto sintetizable ligeramente diferente entre ellas. Por ello es muy probable que algún código escrito para una herramienta tenga que ser ligeramente modificado si se ha de emplear otra herramienta de síntesis.

Estilos de modelado

Estilos de escribir VHDL...

Habiendo conocido someramente hasta aquí algunas virtudes y defectos del VHDL, ahora veremos algunos estilos diferentes de como puede escribirse código...

1.5.0 Niveles de abstracción

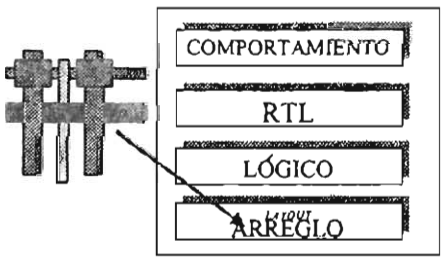


La **abstracción**: define el detalle

Los diferentes estilos de escribir código VHDL tienen que ver con un concepto llamado **abstracción**. El nivel de abstracción define con cuánto detalle es especificado un diseño para una descripción en particular. En nuestro caso tenemos principalmente cuatro niveles de abstracción.

1.5.1 Niveles de abstracción

Nivel Arreglo (*Layout*)



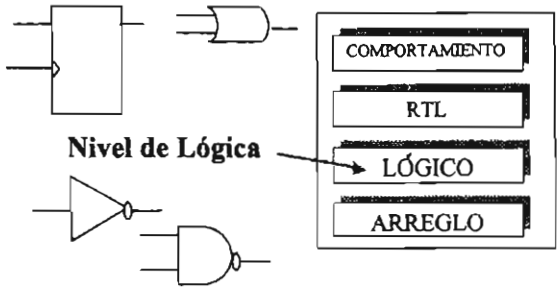
Nivel ARREGLO (*LAYOUT*):

- Disposición sobre el Silicio
- Temporización, efectos analógicos

El nivel más bajo que definiremos es el Nivel Disposición o Arreglo (*LAYOUT*). Se requiere especificar información acerca de la disposición o colocado del diseño en el Silicio, pudiendose especificar información detallada de tiempos o de efectos analógicos.

1.5.3 Niveles de abstracción

Nivel de lógica



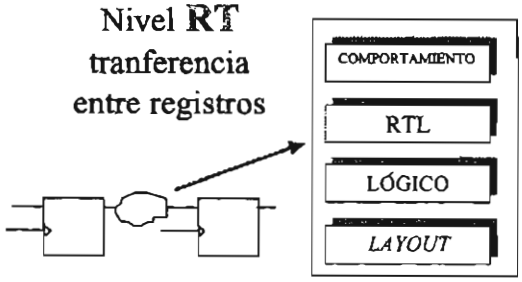
- funciones
- arquitectura
- tecnología
- temporización detallada

Por arriba del nivel Disposición o Arreglo está el nivel de Lógica, donde interconectamos compuertas y registros. La información del nivel disposición y los efectos analógicos no se toman en cuenta, y el diseño contiene información acerca de funciones lógicas, arquitectura, tecnología y de tiempos .

1.5.4 Niveles de abstracción

Nivel RT o RTL

Nivel **RT** tranferencia entre registros



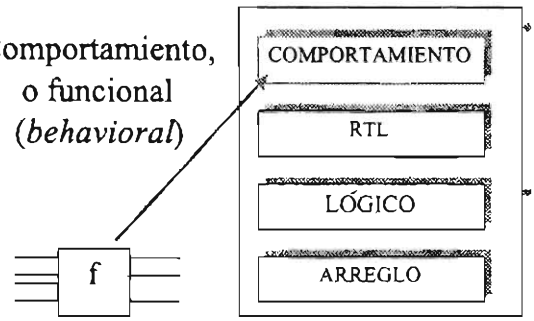
- Arquitecturas de funciones y registros
- Temporizaciones basadas en ciclos

Al nivel de transferencia entre registros, el uso de VHDL se apega a estilos estrictos para definir cada uno de los registros en el diseño y la lógica combinacional entre ellos. Entonces nuestros diseños contienen información de la arquitectura, pero sin detalles de la tecnología. Los tiempos de retardo absoluto no son especificados.

1.5.4 Niveles de abstracción

Comportamiento (*behavioral*)

Comportamiento, o funcional (*behavioral*)



- Únicamente funciones, sin ninguna arquitectura
- Se requiere temporización detallada

Por encima de RTL tenemos el nivel de comportamiento (*BEHAVIORAL*). Este nivel usa VHDL para describir el funcionamiento del diseño sin especificar detalles de la arquitectura entre registros. El código comportamiento puede contener tanta información de tiempos como el diseñador lo requiera para representar su función.

1.6.0 Niveles funcional vs RTL

COMPORTAMIENTO

VS

RTL

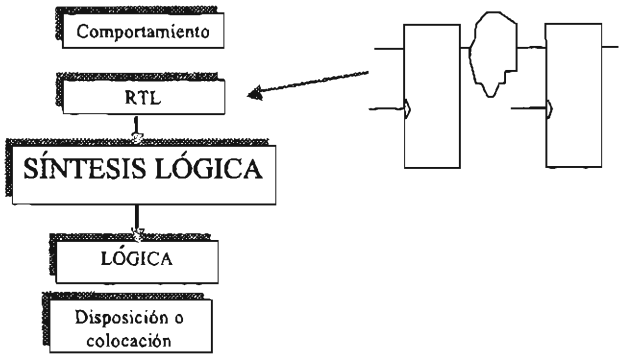
A partir de lo anterior podemos ver que al menos hay dos estilos diferentes de escribir VHDL:

Comportamiento y RTL.

¿Cuándo usar uno u otro?

1.6.1 Comportamientol vs RTL

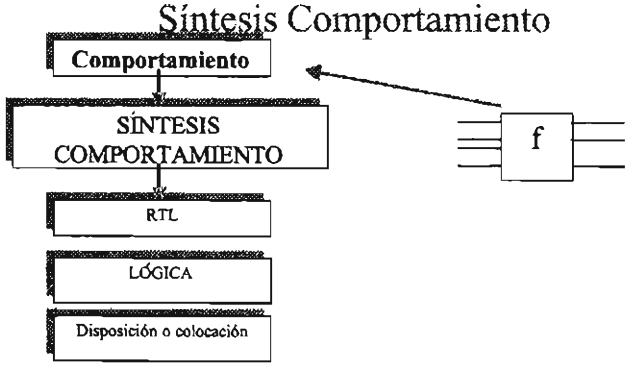
RTL para síntesis



Actualmente la mayoría de código escrito en VHDL lo es a nivel RTL por la sencilla razón de que prácticamente todas las herramientas de síntesis requieren que así sea escrito. A este nivel el diseñador mantiene control sobre la arquitectura de los registros que intervienen en su diseño.

1.6.2 Comportamiento vs RTL

Síntesis Comportamiento

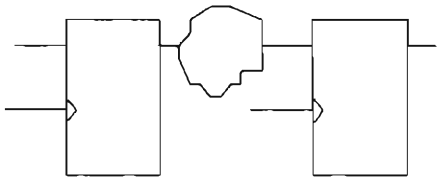


Las herramientas de síntesis-comportamiento automáticamente generan arquitecturas de registros y lógica desde una descripción 'VHDL comportamiento'. Actualmente ya están disponibles para prueba varias de estas herramientas y lo común es usarlas cuando se implementan algoritmos para aplicaciones de procesamiento de señales.

1.6.3 Comportamiento vs RTL

RTL para síntesis

¡VHDL-RTL para síntesis!



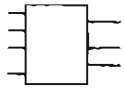
Así entonces la mayoría de veces se requiere escribir los diseños en VHDL a nivel de tranferencia de registros (RTL) para poder usar las herramientas de síntesis comerciales más comunes.

1.6.4 Comportamiento vs RTL

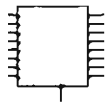
Funcional para estímulos y especificaciones

VHDL-
-Comportamiento
para:

estímulos



Partes estándar



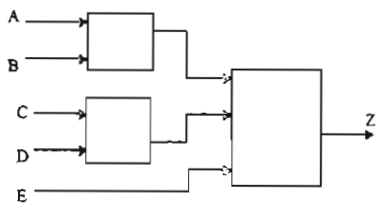
Especificaciones

VHDL-comportamiento es usado para crear estímulos, para modelar partes estándar del sistema, o para crear especificaciones simulables de dichos sistemas. Luego se verá como crear *test bench,* es decir un banco de estímulos junto con sus modelos VHDL simulables.

1.7.0 Conceptos principales del lenguaje

Conceptos principales del lenguaje VHDL

Ahora se introducirán los principales conceptos del lenguaje VHDL, asi como las principales construcciones sintetizables disponibles para el usuario.



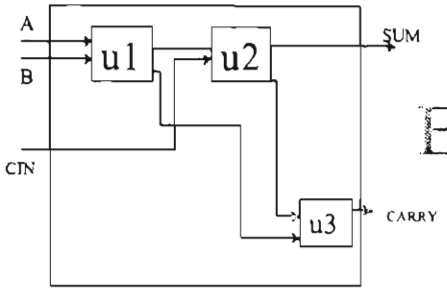
1.7.1 Conceptos principales del lenguaje

Concurrencia

Concurrencia

Ya que VHDL es un lenguaje que nos permite describir Hardware, debe por lo tanto ser capaz de describir actividades que estan sucediendo en paralelo. Tales actividades se dice que son "concurrentes".

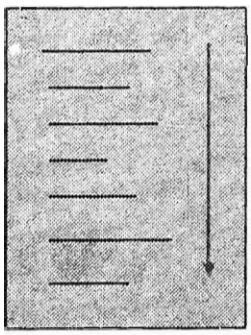
Estructura



Estructura

Un aspecto específico de la descripción de concurrencia es la capacidad para poder describir estructuras o jerarquías. VHDL permite esto, y las descripciones de estructura pueden ser libremente mezcladas con descripciones de comportamiento. Es también posible tener justo una descripción exclusivamente estructural, i.e. una *netlist* en VHDL.

Postulados secuenciales

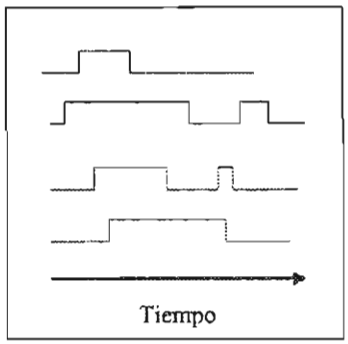


Secuencial

Una descripción VHDL puede también contener postulados que se ejecutan uno tras otro en secuencia, semejante a los programas de software tradicional ('C', 'Pascal', 'ADA'). Estos se conocen como postulados secuenciales.

1.7.4 Principales conceptos del lenguaje

Tiempo



Tiempo

Y finalmente, VHDL permite modelar el concepto de tiempo, el cual realmente es una parte importante de la descripción de los sistemas electrónicos.

1.8.0 Entidad (*Entity*)

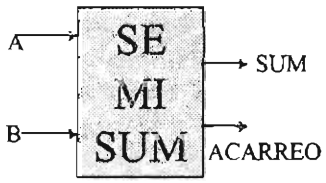
Entidad (*entity*)

Justo hasta ahora volveremos la atención hacia algunas cuestiones básicas de la sintaxis del lenguaje VHDL. Iniciaremos con el bloque principal de cualquier diseño VHDL: la ENTIDAD.

1.8.1 Entidad

Entity: interfaz al diseño

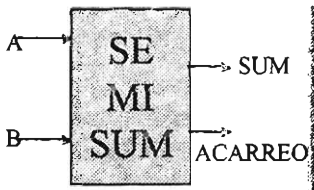
```
entity SEMISUM is
   port (A, B : in bit ;
         SUM, ACARREO : out bit ) ;
end SEMISUM ;
```



La entidad (*entity*) en VHDL describe la interfaz a un bloque jerárquico, sin definir su comportamiento. La entidad es equivalente a un símbolo en los diseños basados en esquemas. Veamos la sintaxis para el caso de un medio sumador...

1.8.2 *Entity*

Entity: sintaxis....



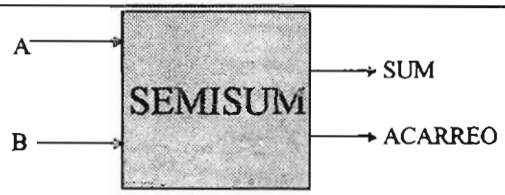

```
entity SEMISUM is
     port (A,B: in bit;
           SUM,ACARREO: out bit ) ;
end SEMISUM;
```

La sintaxis de una entidad se inicia con la palabra clave "entity", seguida por el nombre de la entidad, y enseguida la palabra clave "is". Las entradas y salidas son contenidas dentro del postulado "port". En el listado anterior se establece la dirección "in" u "out", y el que cada una de las señales o puertos es de un sólo "bit"

1.8.3 Entity

Entidad : .....mas sintaxis....

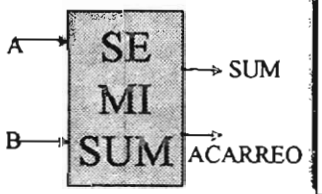
```
entity SEMISUM is
    port ( A,B: in bit ;
           SUM, ACARREO: out bit) ;
end SEMISUM ;
```



Por último tenemos la palabra clave "end", y el nombre de la entidad, seguida de un punto y coma (;) con lo cuál termina la declaración de la entidad.

1.8.4 Entity

Nótese el uso de los puntos y comas




```
entity SEMISUM is
    port ( A, B : in bit ;
           SUM , ACARREO : out bit ) ;
end SEMISUM ;
```

Tengase cuidado con el uso de los puntos y comas: no debe ir el punto y coma, para la lista final de port, antes de cerrar el paréntesis que termina dicho postulado.

1.9.0 Arquitectura

Arquitectura (architecture)

Ahora el interés se centrará en comprender como describir el funcionamiento, descripción o comportamiento (*behavior*) de una entidad (*entity*). Para esto se necesita una construcción VHDL conocida como architecture (arquitectura).

1.9.1 Arquitectura

sintaxis de la Arquitectura

```
architecture COMPORTA of SEMISUM is
begin
        SUM <=  A xor B ;
        ACARREO <=  A and B ;
end COMPORTA ;
```

En el listado presente, se tiene la arquitectura de la entidad SEMISUM. La regla sintáctica dicta empezar con la palabra reservada 'architecture', seguida por un nombre definido por el usuario para identificar la arquitectura misma. Ésta, en este ejemplo se le llama COMPORTA.

1.9.2 *Architecture*

..asociada con una *entity*

```
architecture COMPORTA of SEMISUM is
begin
        SUM <=  A xor B ;
        ACARREO <=  A and B ;
end COMPORTA ;
```

Una arquitectura siempre va asociada con una entidad mediante la clave 'of'. En el caso actual la arquitectura COMPORTA es de la entidad ('of') SEMISUM. Luego siguen las palabras reservadas 'is' y 'begin'.

1.9.3 *architecture*

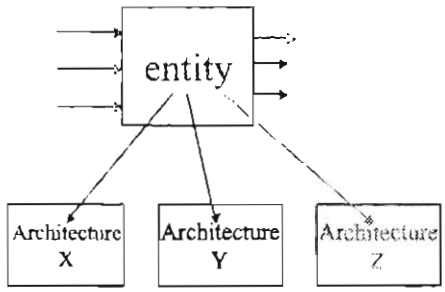
Asignación de señales

```
architecture COMPORTA of SEMISUM is
begin
        SUM <= A xor B ;
        ACARREO <= A and B ;
end COMPORTA ;
```

Después del 'begin' de la arquitectura, tenemos los postulados que definen el comportamiento actual.En este caso tenemos la asignación de dos "señales": el resultado de "A xor B" se asigna a la señal SUM, y el resultado de "A and B" a la señal ACARREO.

1.9.4 architecture

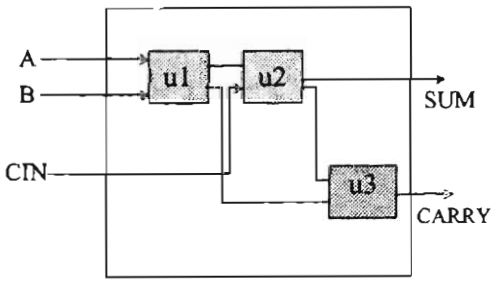
Una entidad puede tener múltiples arquitecturas



Es importante hacer notar que una entidad puede tener más de una arquitectura. La utilidad de esto se manifiesta cuando un diseño tiene que ser descrito a varios niveles de abstracción: pueden existir descripciones de comportamiento, RTL o a nivel de compuertas para el mismo diseño.

1.10.0 VHDL permite representación jerarquica

Jerarquía (*hierarchy*)



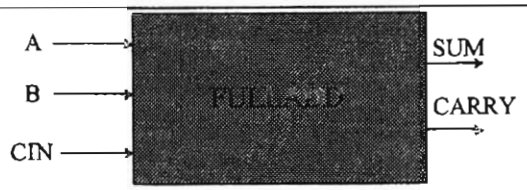
U1 : semisumador
U2 : semisumador
U3 : or

La entity y la architecture son los bloques de construcción principales a partir de los cuales se contruye la jerarquía. Veamos como se describe la jerarquía en VHDL, sirviéndonos de un sumador completo hecho a partir de dos semisumadores y una componente or,...

1.10.2 Representación jerárquica

Entidad

```
entity FULLADD is
   port ( A, B, CIN : in bit ;
          SUM , CARRY : out bit ) ;
end FULLADD ;
```

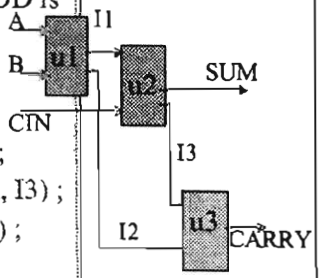


Se lista el código VHDL para la *entity* FULLADD. Como se puede ver, los cinco puertos se declaran de un solo bit, tal como se hizo para el medio sumador.

1.10.2 Representación jerárquica

Arquitectura

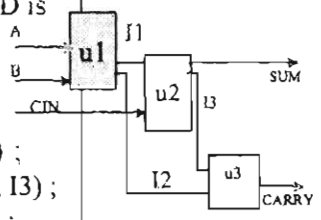
```
architecture ESTRUCTURAL of FULLADD is
        signal I1 , I2 , I3 : bit ;
        -- otras declaraciones
begin
  u1 : SEMISUM port map(A, B , I1 , I2 ) ;
  u2 : SEMISUM port map( I1, CIN, SUM, I3) ;
  u3 : ORGATE  port map(I3, I2, CARRY) ;
end ESTRUCTURAL ;
```



Se muestra la arquitectura de FULLADD. Vease las tres líneas de código entre los postulados *begin* y *end*. Cada uno de ellos crea una instancia desde otra entidad, lo cuál es similar a colocar símbolos en un diagrama esquemático.

Instanciación de componentes

```
architecture ESTRUCTURAL of FULLADD is
        signal I1 , I2 , I3 : bit ;
        -- otras declaraciones
begin
  u1 : SEMISUM port map (A, B , I1 , I2 ) ;
  u2 : SEMISUM port map( I1, CIN, SUM, I3) ;
  u3 : ORGATE  port map(I3, I2, CARRY) ;
end ESTRUCTURAL ;
```

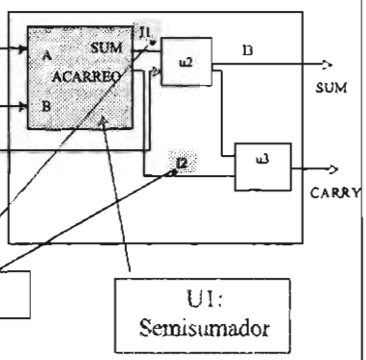


Revisando la sintaxis vemos primero el nombre de la instancia (u1), y después de los dos puntos tenemos el nombre de la entidad que estamos instanciando. Enseguida la palabra reservada "port map", y luego la lista de las señales dentro de FULLADD que están conectadas a los puertos del diseño de SEMISUM.

Conexiones por posición

```
entity SEMISUM is
port (A, B :  in bit ;
   SUM, ACARREO : out bit ) ;
end SEMISUM ;
```

```
u1 : SEMISUM port map(A, B, I1, I2) ;
```



Las conexiones de la entidad SEMISUM han sido hechas por posición: el primer puerto listado en SEMISUM (puerto A) es conectado a la señal A dentro de FULLADD,

....

1.11.0 Declaraciones locales

Declaraciones de:

-Señales locales

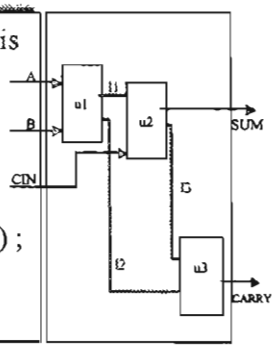
-Componentes

Existen dos declaraciones que hasta aquí han sido hechas localmente dentro de la arquitectura. Ellas son las "signal" locales, y una referencia a las entidades SEMISUM y ORGATE, conocidas como una "component". Veamos primero las señales locales,...

1.11.1 declaraciones locales

Señales locales

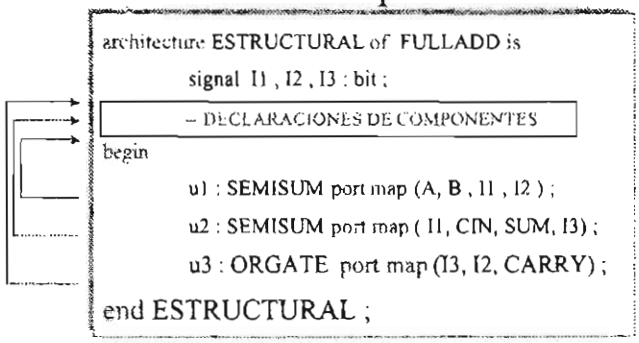
```
architecture ESTRUCTURAL of FULLADD is
        signal I1 , I2 , I3: bit ;
        -- otras declaraciones
begin
  u1 : SEMISUM port map (A, B , I1 , I2 ) ;
  u2 : SEMISUM port map( I1, CIN, SUM, I3) ;
  u3 : ORGATE  port map(I3, I2, CARRY) ;
end ESTRUCTURAL ;
```



El diseño FULLADD, tal como se ha hecho, contiene las señales I1, I2, e I3, que son locales al diseño, i.e. no son entradas o salidas. Las señales locales pueden ser declaradas dentro de la arquitectura antes de la palabra clave "begin",..

1.11.2 Declaraciones locales

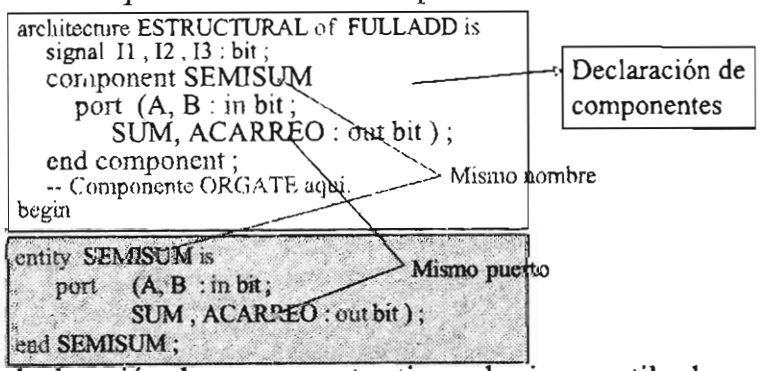
Declaración de componentes



Antes de que podamos instanciar una entidad tal como SEMISUM, es necesario declarar localmente la interfaz a la entidad en la arquitectura FULLADD. Esto se hace con lo que se conoce como 'declaración de la componente'.

1.11.3 Declaraciones locales

component: lo mismo que la entidad



La declaración de componentes tiene el mismo estilo de sintaxis que la declaración de su entidad. En el 99% de los casos, los nombres de las componentes declaradas tendrán el mismo nombre y la misma lista de puertos que el de la entidad. No hay vuelta a esto: la declaración de la componente debe ser hecha y estar presente.

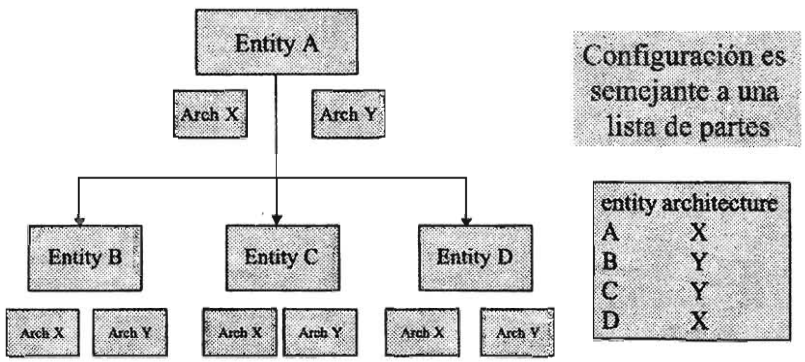
1.11.4 Declaraciones locales

Código VHDL completo para FULLADD

```
entity FULLADD is
        port (A, B, CIN,  : in bit ;
              SUM, CARRY : out bit ) ;
end FULLADD ;

architecture ESTRUCTURAL of FULLADD is
        signal  I1, I2, I3 : bit;
        component SEMISUM
                port (A, B, : in bit ;
                      SUM, ACARREO : out bit ) ;
        end component ;
        component ORGATE
                port ( A, B : in bit ;
                       Z : out bit ) ;
        end component ;
begin
        u1 : SEMISUM port map (A, B, I1, I2) ;
        u2 : SEMISUM port map (I1, CIN, SUM, I3) ;
        u3 : ORGATE  port map (I3, I2, CARRY) ;
end ESTRUCTURAL ;
```

El código completo para FULLADD es mostrado. Nótese que se declararon las señales locales y las componentes antes del "begin" de la arquitectura, y que la descripción de la estructura del diseño esta entre el "begin" y el "end".

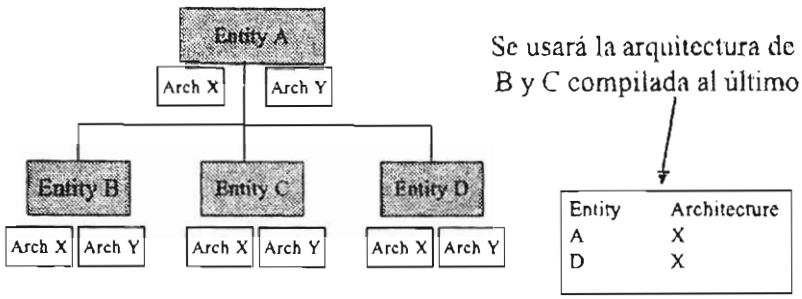
1.12.0 Configuraciones



El siguiente objeto VHDL que estudiaremos se llama la configuración. Considere que un diseño jerárquico está constituido de un conjunto de entidades, cada una de las cuales puede tener múltiples arquitecturas. La configuración es semejante a un listado de partes, especificando cuáles arquitecturas serán usadas por cada una de las entidades.

1.12.1 Configuración

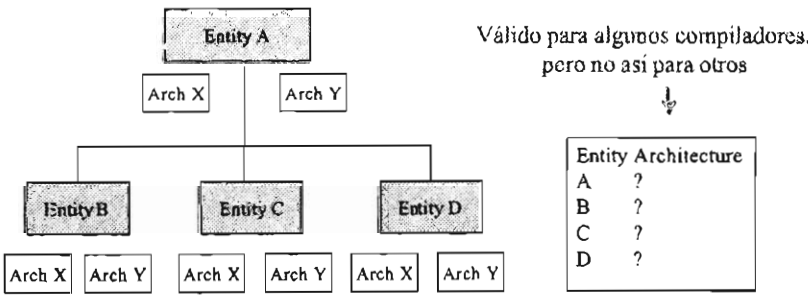
No se requiere de toda la información



No se requiere dar toda la información de la "lista de partes" para todas las jerarquías del diseño. Si no se da información de la configuración de una entidad, entonces la última arquitectura usada en la compilación es la seleccionada para ser usada durante la simulación.

1.12.2 Configuraciones

Puede no requerirse dar una configuración default

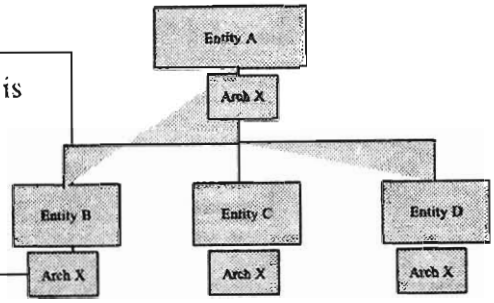


Lo común es tener tan sólo una arquitectura por entidad para un diseño completo, por lo que en teoría no se requeriría de ninguna información de configuración adicional. Algunos simuladores permiten lo anterior, pero algunos otros necesitan que se de la "configuración por default". El usar la configuración por defecto hace al código resultante más portable.

1.12.3 Configuraciones

Sintaxis para configuración

```
configuration CFG_A of A is
        for X
        end for ;
end CFG_A ;
```



CONFIGURACION POR DEFAULT

Veamos la sintaxis para dar la configuración por default: una configuración tiene nombre propio y se asocia con la entidad al más alto nivel de jerarquía tal como se muestra en el código. Internamente en la configuración por default se nombran las arquitecturas de los niveles más altos de la jerarquía entre el "for" y "end for".

1.12.4 Configuraciones

Siempre usar la configuración por default

¡Usar la configuración por default!

Tal como se ha dicho, es recomendable construir al menos una configuración por default para cada diseño. Este es un tema avanzado... muy usual para simulación y poco para síntesis.

1.13.0 Procesos y tipos

Process

y

Type

¿Qué es un **proceso** en VHDL?

¿Qué relación hay entre los tipos y las señales?

1.13.1 Procesos y tipos

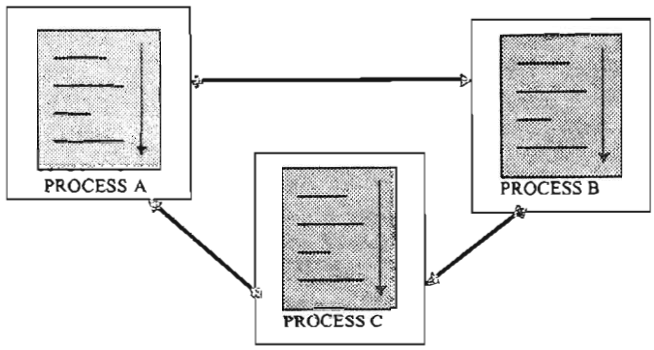
Los procesos contienen postulados secuenciales

```
entity OR_FUN is
        port (A, B : in bit ;
              Z : out bit ) ;
end OR_FUN ;
architecture CMPRTMNT of OR__FUN is
begin
        OR_GATE : process ( A,B)
        begin
              if ( A='1' or B='1' ) then
                    Z <= '1' ;
              else
                    Z <= '0' ;
              end if ;
        end process OR_GATE ;
end CMPRTMNT ;
```

Los **procesos** son una construcción del VHDL, dentro de los cuáles se escribe código cuyos postulados se ejecutan en secuencia. Un proceso existe dentro de una arquitectura, y múltiples procesos interactúan uno al otro concurrentemente.

1.13.2 Procesos y tipos

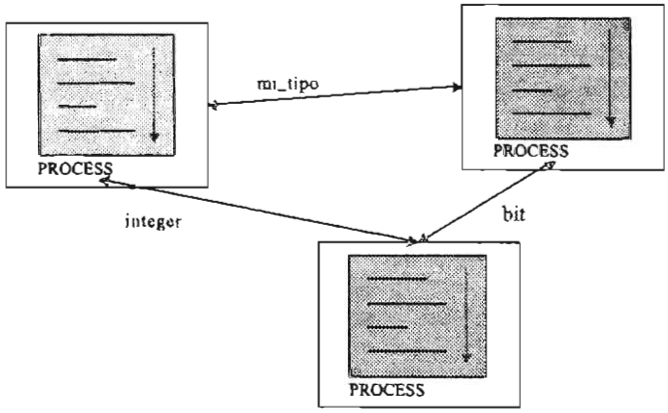
Simulación VHDL: múltiples procesos



Cualquier modelo VHDL es visto por un simulador como una colección de procesos. Cada uno de los procesos ejecuta sus postulados en secuencia y los múltiples procesos están interactuando entre ellos concurrentemente, tal como se ve en el diagrama. Puede haber cualquier número de procesos dentro de una misma arquitectura.

1.13.3 Process y tipos

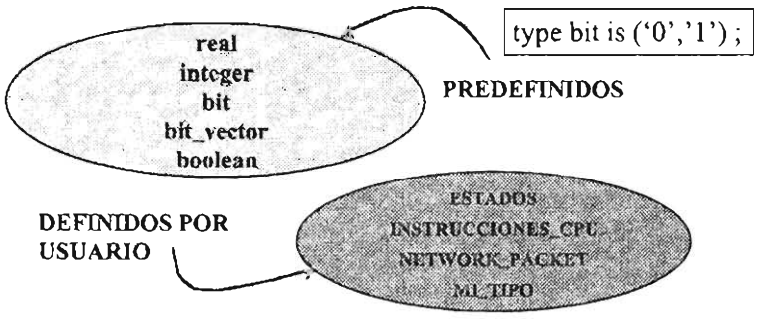
Tipo: conjunto de valores que pueden asignarse a una señal



Cada una de las señales (*signals*) en un diseño VHDL tiene asociado un tipo al declararla. Un tipo define un conjunto de valores los cuáles se permite asignar a las señales.

1.13.4 Process y tipos

Tipos predefinidos y definidos por el usuario



Existen tipos predefinidos en el lenguaje estándar, y el usuario puede definir tipos propios. Hasta aquí hemos visto el tipo bit, el cual puede tener los valores de '0' y '1'. Se discutirá más sobre ellos..

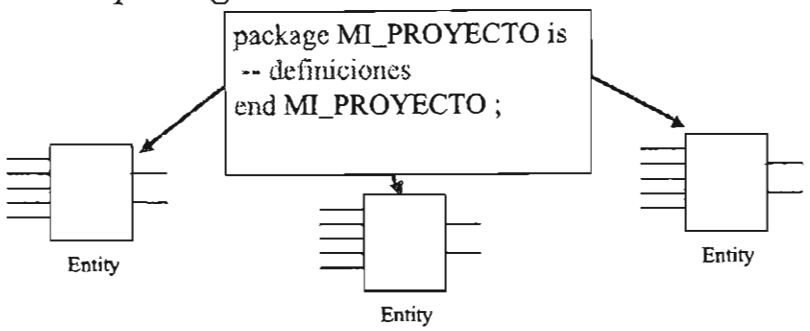
1.14.0 Los *packages*

Paquete (*package*....)

El siguiente objeto VHDL a estudiar es el conocido como *package* .....

1.14.1 *Packages*

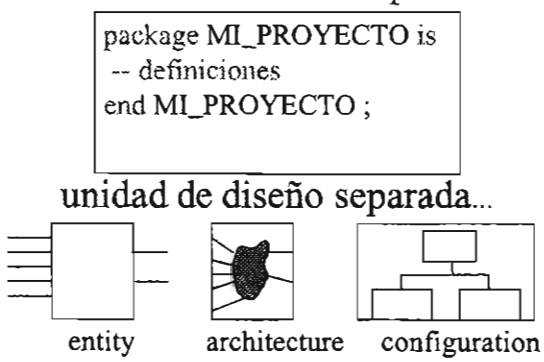
package: colección de definiciones



Un paquete contiene una colección de definiciones las cuáles pueden ser referenciadas por muchos diseños al mismo tiempo. Colocando definiciones que son comunes entre diseños en un paquete, ayuda a que un diseño mancomunado se trabaje con más consistencia.

1.14.2 *Packages*

Unidad de diseño separada



Un paquete VHDL es una unidad de diseño separada. Por lo tanto el paquete existe fuera de las otras unidades de diseño que hemos considerado hasta aquí, tal como: *entity*, *architecture* y *configuration*.

1.14.3 *Packages*

Constantes, tipo de datos, componentes, sub-rutinas.

```
package MI_PROYECTO is
--constantes
-- tipos definidos por el usuario
--declaración de componentes
--subprogramas
end MI_PROYECTO ;
```

Un paquete puede contener definiciones de constantes, tipos de datos definidos por el usuario, declaraciones de componentes, o subprogramas de código VHDL que son compartidos entre diferentes diseños. Una lista completa de los ítems que pueden ser encontradas dentro de un paquete es bastante extensa y se encuentran en el manual de referencia.

1.15 1.0Compilación y bibliotecas

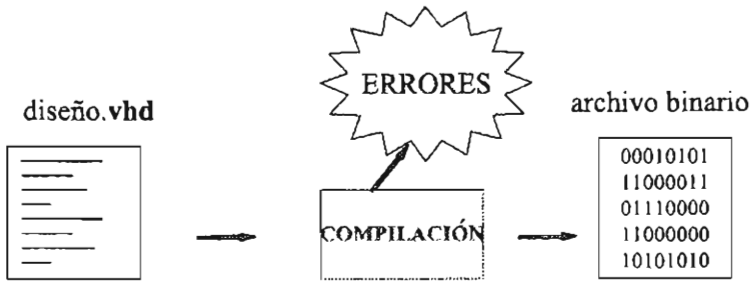
-¿Cómo se compila el código fuente VHDL ?

- biblioteca (*library*) VHDL

Veremos como se compila el código fuente VHDL y como los diseños son agrupados juntos dentro de una biblioteca (*library*),....

1.15.1 Compilación y bibliotecas

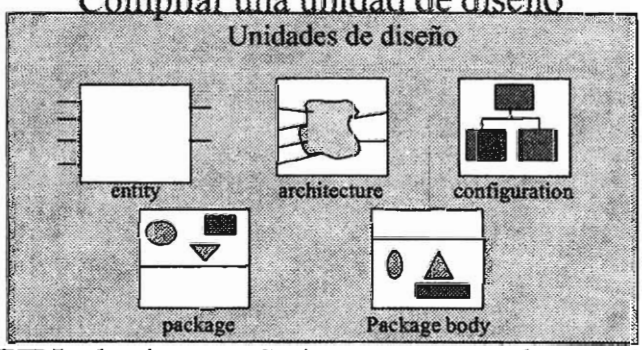
Compilación o análisis



Una vez que el código VHDL ha sido escrito como texto en un archivo, el código es revisado para errores de sintaxis, y entonces un archivo binario es creado. Este proceso se conoce en algunas herramientas como "compilación", y por otras como "análisis". Seguiremos aplicando el término "compilación" en esta presentación.

1.15.2 Compilación y bibliotecas

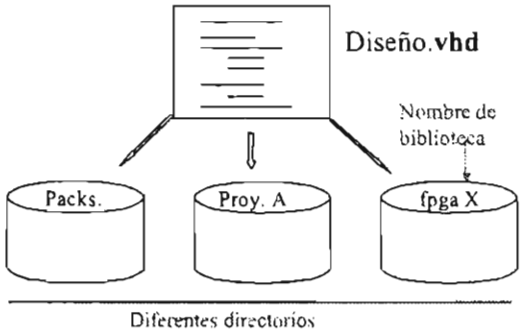
Compilar una unidad de diseño



En VHDL el más pequeño item que se puede compilar es una sola unidad de diseño. Ellas son la entity, architecture, configuration y el package, estos últimos pueden tener una parte separada llamada un package body. Esos cinco objetos son las únicas unidades de diseño definidas en el lenguaje VHDL.

1.15.3 Compilación y bibliotecas

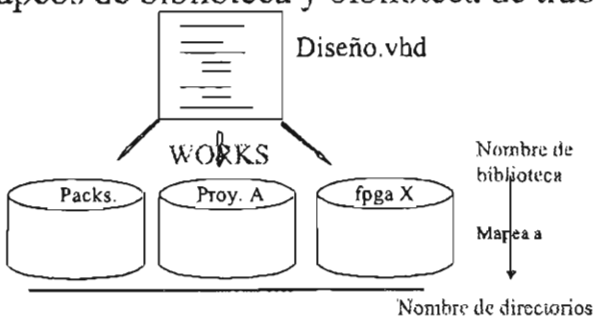
Compilar dentro de una biblioteca



Cuando se compila código fuente VHDL, el archivo binario resultante es almacenado en un directorio específico. A modo de tener referencia a este archivo independiente de una trayectoria a un directorio en específico, a dichos directorios se les da un nombre lógico, conocido como "library".

1.15.4 Compilacion y bibliotecas

Mapeos de biblioteca y biblioteca de trabajo



Todas las herramientas de simulación y síntesis VHDL accesan un archivo que mapea nombres de bibliotecas VHDL a trayectorias de directorio fisicas en su sistema. VHDL también permite que una biblioteca llamada WORK sea definida como la biblioteca dentro de la cuál las unidades de diseño son compiladas si ninguna biblioteca de destino es especificada.

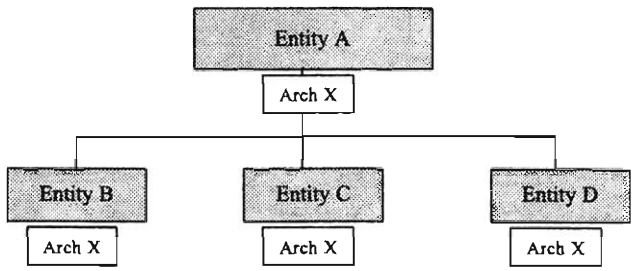
1.16.0 El cuadro completo

...el cuadro completo...

Dado que se presentaron ya los principales bloques de construcción, junto con los principales conceptos del lenguaje VHDL,veamos el cuadro completo de la relación de ellos en una situación de diseño típica.

1.16.1 El cuadro completo

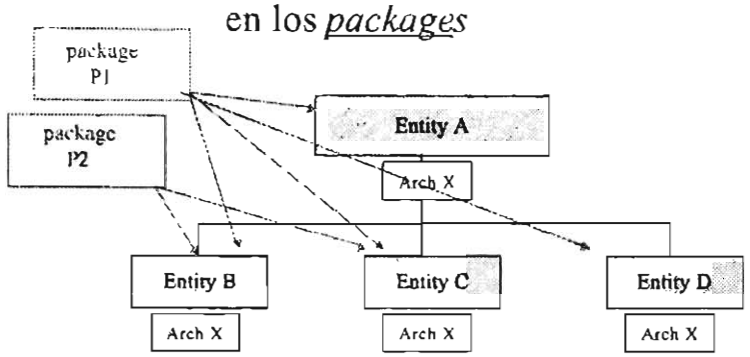
Jerarquía de entidades-arquitecturas



Como se ha visto, un diseño jerárquico completo está definido por múltiples entidades, las cuales tienen al menos una arquitectura. Únicamente una arquitectura es referida para cada una de las entidades en una simulación particular.

1.16.2 El cuadro completo

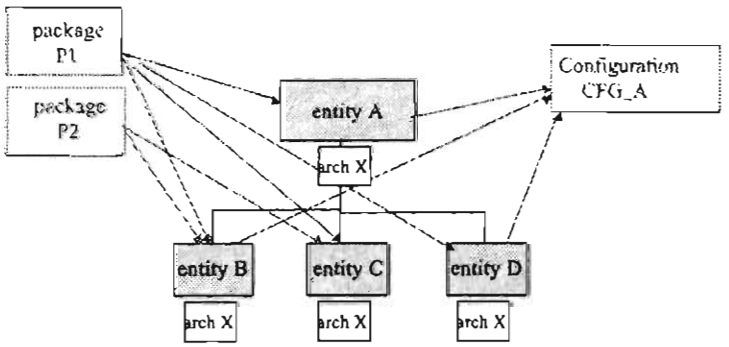
Las definiciones más comunes se colocarán en los *packages*



La mayoría de las entidades o arquitecturas harán referencia a uno o más paquetes de definiciones comunes,....

1.16.3 El cuadro completo

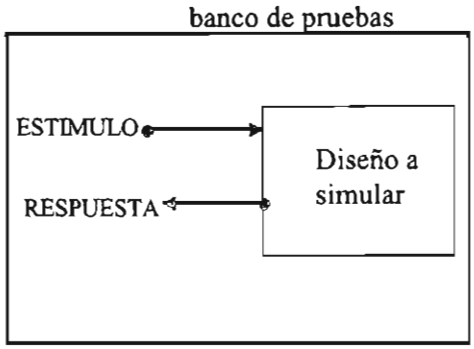
Configuración



La liga entre cada uno de los niveles de jerarquía, y la especificación de cuál arquitectura deberá ser usada, es llevada a cabo por la configuración.

1.16.4 El cuadro completo

Estímulos desde un banco de pruebas



Finalmente, en la jerarquía de más alto nivel usualmente contiene tanto el diseño como los estímulos que se aplicarán durante simulación, los cuales son escritos en VHDL.

2. Tipos de datos y señales

2.1.0 De lo que trata la segunda presentación

La finalidad...

La siguiente presentación tiene como objetivo ver como se efectua la asignación de valores a las señales en VHDL, así como el dar un panorama de las reglas relativas al uso de los tipos (type).

2.1.1 Lo que trata la segunda presentación

Concepto de tipo

- Concepto de 'tipo' (*type*)

- Tipos estándares

Veremos el concepto "tipo de dato", y enseguida estudiaremos los tipos estándar, definidos como parte integral del lenguaje VHDL.

2.1.2 ¿De que trata la lección?

arrays

- arreglos (*array*)

y

- manipulación de señales (*signal*)

Enseguida trataremos el concepto de "arreglo" (*array*), equivalente al bus en los esquemas. Se verán algunos ejemplos de como pueden manipularse en VHDL las señales tanto de un solo bit como de arreglos de bits.

2.1.3 ¿Qué más veremos?

Tipos enumerados

Defina sus propios tipos de datos :

tipos propios o

enumerated types

Enseguida se estudiará como se pueden definir tipos de datos propios, los cuales se conocen en el lenguaje VHDL como 'tipos enumerados'.

2.1.4 ¿Qué más veremos?

Lógica Estándar

Standard_Logic

Veremos finalmente la definición de un tipo conocido como Standard_Logic, tipo al que se le ha considerado como el estándar para la descripción de señales en VHDL.

2.2.0 El concepto de tipo

El concepto de 'tipo'

y

como se especifica

Primero hay que describir el concepto 'tipo de dato', y luego en que lugar del código se especifica dicho 'tipo de dato de una señal'.

2.2.1 El concepto de tipo

Concepto de tipo

'0'
'1'

'0' y '1' son los valores permitidos en el **tipo** bit

```
architecture ESTRUCTURAL of
FULLADD is
        signal N_SUM : bit ;
        --otras declaciones
begin
 -- código
end ESTRUCTURAL ;
```

El lenguaje VHDL define que cada una de las señales (*signal*) debe tener asociado un tipo de dato, el cuál debe darse cuando la señal se declara. El tipo define un conjunto de valores, y la asignación a esa señal debe ser siempre un valor definido por ese conjunto.

Especificar declaración en *port* o en *signal*

```
entity FULLADD is
   port (A, B, CIN : in  bit ;
         SUM, CARRY : out bit );
end FULLADD ;
architecture ESTRUCTURAL of FULLADD is
         signal  N_SUM  :  bit ;
--otras declaciones
begin
--Codigo
end ESTRUCTURAL ;
```

Tipo

Tipo

Tipo

Las señales son generalmente declaradas ya sea en la sección de 'port' de una entidad, o mediante una declaración explícita de la señal dentro de la arquitectura. Una vez que la señal es declarada en cualquiera de esas secciones, el tipo que conlleva asociado debe ser especificado: en el ejemplo para las señales el tipo es 'bit'.

¡Los tipos deberán acoplar!

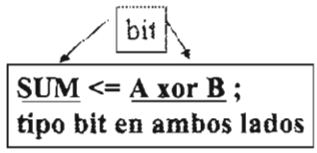
```
entity SEMISUM is
   port (A,B : in bit;
         SUM, ACARREO : out bit );
end SEMISUM;
```

SUM <= A xor B;
Los tipos deben ajustar...

Al hacer una asignación a una señal, los tipos a ambos lados del operador de asignación deben coincidir..

tipo bit en ambos lados

```
entity SEMISUM is
    port (A,B : in bit;
          SUM, ACARREO : out bit );
end SEMISUM;
```



En el ejemplo mostrado, el resultado de 'A xor B' debe ser del tipo bit para que sea correcta la asignación a la señal SUM.

Tipos de datos estándar
Tipos predefinidos...

```
package STANDARD is
type BOOLEAN is (FALSE,TRUE);
type BIT is ('0', '1');
type CHARACTER is (--ascii set );
type INTEGER is range definido_implementación;
type REAL is range definido_implementación;
      -- BIT_VECTOR, STRING, TIME
end STANDARD ;
```

Una vez conocido el concepto de tipos de datos, ahora veremos los tipos estándar predefinidos en el lenguaje VHDL. Todos estos tipos son especificados como código fuente VHDL, y el lenguaje define que todos ellos deberán ser colocados en un paquete llamado STANDARD...

2.3.1 Tipos de datos estandar

boolean, time

boolean: false , true

false

true

time: 10 ns, 200 fs, 2.5 ps

10 ns

200 fs

2.5 ps

El tipo 'boolean' define dos valores: true y false. El tipo boolean es devuelto en la comparación de dos valores ... El tipo 'time' se usa para modelar tiempos en las simulaciones, y consiste tanto de un número como de una unidad ...

2.3.2 Tipos de datos estándar

bit, bit_vector

'0'

'1'

bit

"010101010"

"111"

"0000"

bit_vector

El tipo bit puede usarse para modelar un solo bit y consiste de los valores '0' y '1'.

El tipo bit_vector define una colección de bits, y puede ser usado para modelar una estructura de bus. Note las diferencias al especificar un valor de tipo bit(') y un valor de tipo bit_vector(").

2.3.3 Tipos de datos estandar

character, string

character

'a'
'?'
'A'
'1'

"Hola"
"VHDL"
"UAM_AZC"

string

El tipo character define un solo caracter en VHDL. El tipo string define un arreglo de caracteres. Notese la diferencia entre los apóstrofes ' y las comillas " para los tipos.

2.3.4 Tipos de datos estándar

integer, real

0
-50
294,..

Integer

1.02
-37.5
1.0,...

Real

El tipo integer debe ser cualquier número entero, el tipo real puede ser cualquier número con cualquier cantidad de decimales. El punto debe de ir para los tipos reales, mientras que no lo debe llevar en el caso de los tipos enteros.

2.4.0 *Signal y Driver*

Señales (*signal*) y sus manejadores (*drivers*)

Asignación de valor a las señales (*signal*)

Una vez presentados los tipos estándar, veremos los conceptos escondidos tras la 'asignación a una señal' (*signal*).

2.4.1 *Signal y Drivers*

Postulado de asignación a una señal

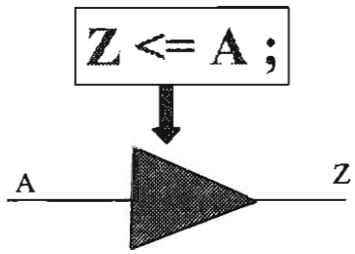
```
signal A, B, Z : bit ;
signal X_INT : integer ;
```

postulado de asignación de señal ⟹ **Z <= A ;**

La asignación de un nuevo valor a una señal en VHDL se hace mediante el llamado 'postulado de asignación de señal'.

controlador de señal

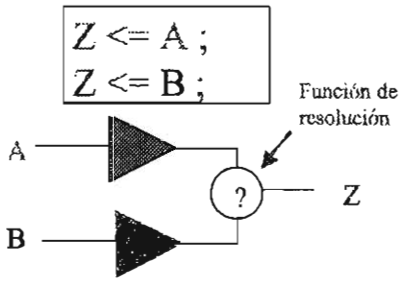
```
signal A, B, Z : bit ;
signal X_INT : integer ;
```



La asignación a una señal define un *driver* o manejador para esa señal. El concepto de *driver* es exactamente el mismo que en hardware. La herramienta de síntesis, para el caso presente no necesariamente inferirá usar un buffer, a menos que sea necesario manejar una gran carga.

Múltiples *drivers*

```
signal A, B, Z : bit ;
signal X_INT : integer ;
```

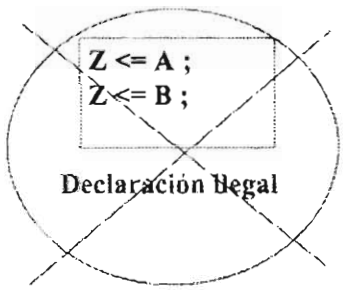


En caso que más de una asignación sea hecha a una sola señal, la señal tendrá más de un driver. En este caso la señal deberá ser declarada de una clase de tipo especial, conocido como tipo *resolved*.

Un tipo *resolved* tiene asociada una llamada a una funcion la cual "resuelve" el valor final de la señal,...

Todos los tipos son unresolved para los casos tratados aquí

```
signal A, B, Z : bit ;
signal X_INT : integer ;
```



Los tipos que veremos se conocen como *unresolved*, esto quiere decir que los objetos de este tipo tan soló pueden tener un driver.

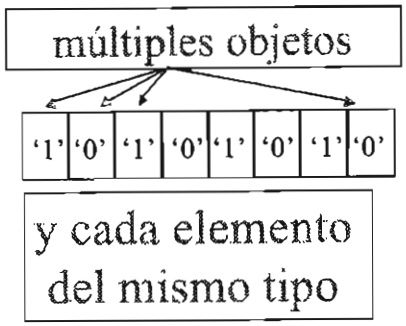
Arreglos
(*arrays*...)

Ahora es tiempo de ver como se definen y como se manipulan en VHDL las señales tipo arreglos (*arrays*) ...

2.5.1 Arreglos

Elementos del mismo tipo

Arreglo



Un *array* es una colección de objetos, siendo cada uno de ellos del mismo tipo.

El lenguaje VHDL define dos tipos estándar de arreglos: bit_vector y string.

2.5.2 Arreglos

El tamaño se da cuando se declara

Declaraciones legales:

```
signal Z_BUS : bit_vector (3 downto 0) ;
signal C_BUS : bit_vector (1 to 4) ;
```

Declaraciones ilegales:

```
signal Z_BUS : bit_vector(0 downto 3) ;
signal C_BUS : bit_vector(3 to 0) ;
```

Justo al momento de declarar un arreglo se tiene que definir su rango: en el ejemplo anterior el rango es de un ancho de 4 bits. El rango se declara usando la notación: 'downto' o 'to'. Sin embargo debe escogerse la palabra clave apropiada .

2.5.3 Arreglos

Asignación de *arrays*

```
signal Z_BUS : bit_vector(3 downto 0) ;
signal C_BUS : bit_vector(1 to 4) ;
Z_BUS <= C_BUS ;
```

es lo mismo que:

```
Z_BUS(3) <= C_BUS(1) ;
Z_BUS(2) <= C_BUS(2) ;
Z_BUS(1) <= C_BUS(3) ;
Z_BUS(0) <= C_BUS(4) ;
```

¡ La asignación es por la posición !

Dos objetos array pueden asignarse uno al otro pero debe cuidarse que ambos sean del mismo tipo y del mismo tamaño. Deberá notarse que la asignación es por posición y no mediante el número de índice.
En el lenguaje VHDL no existe en sí el concepto de Bit Más Significativo (BMS).

2.5.4 Arreglos

secciones de *array*

```
signal Z_BUS : bit_vector(3 downto 0) ;
signal C_BUS : bit_vector(1 to 4) ;
```

Válido o legal:

```
Z_BUS (3 downto 2)  <= "00" ;
C_BUS (2 to 4)  <=  Z_BUS(3 downto 1) ;
```

Ilegal:

```
Z_BUS(0 to 1)  <= "11" ;
```

Se puede hacer referencia tan solo a un 'pedazo' (*slice*) de un arreglo, inclusive a un solo elemento.
La dirección del 'pedazo' (i. e. 'downto' o 'to') debe ajustar con la dirección según se haya declarado el arreglo.

2.6.0 Concatenación y agregados

Concatenación

y

Agregados

La concatenación y los agregados son dos métodos para asociar 'juntando' las señales, a modo de asignar a ellas objetos de arreglos.

2.6.1 Concatenación y agregados

Concatenación

```
signal Z_BUS : bit_vector (3 downto 0) ;
signal A,B, C,D : bit ;
signal BYTE : bit_vector (7 downto 0) ;
signal A_BUS : bit_vector(3 downto 0) ;
signal B_BUS : bit_vector(3 downto 0) ;

Z_BUS <= A & B & C & D ;
```

operador de concatenación

```
BYTE <= A_BUS & B_BUS ;
```

Mediante VHDL es posible asociar bits solos y vectores de ellos, en uno solo, para formar estructuras de arreglos. Esto es lo que se llama concatenación, la cual usa el operador ampersand (&) .

Agregados

```
signal Z_BUS : bit_vector(3 downto 0) ;
signal A, B, C, D : bit ;
signal BYTE : bit_vector(7 downto 0) ;

Z_BUS <= (A , B , C , D) ;
es igual a :
Z_BUS(3) <= A ;
Z_BUS(2) <= B ;
Z_BUS(1) <= C ;
Z_BUS(0) <= D ;
```

Agregado

Otro método de asignar elementos a un arreglo es mediante la agregación. Un agregado es contenido dentro de paréntesis redondos y la asignación a cada elemento esta separada por comas (,).

Especificando elemento por elemento

```
signal X : bit_vector (3 downto 0) ;
signal A, B, C, D : bit ;
signal BYTE : bit_vector(7 downto 0) ;

X <= ( 3 => '1' , 1 downto 0 => '1' , 2 => B)
```

Asignación por nombre

Es posible para fines de asignación, especificar por nombre el elemento de un arreglo, al igual que por su posición dentro del arreglo. Puede igualmente especificarse un rango del arreglo tan pronto como el mismo valor vaya a ser asignado a cada uno de los elementos dentro de ese rango.

others, también soportado por la síntesis

```
signal X : bit_vector (3 downto 0) ;
signal A, B,C, D : bit ;
signal BYTE : bit_vector(7 downto 0) ;

X <= (3 => '1' , 1 => '0' , others => B) ;
```

Al usar agregados, puede emplearse el postulado "others" para asignarle algun valor a todos los otros elementos del arreglo que hasta aquí no han sido especificados. No todas las máquinas de síntesis soportan agregados, aunque si todas soportan concatenación en la manipulación de arreglos.

Definición de tipo (*type*)

Se verá como los usuarios pueden definir en el lenguaje VHDL sus tipos propios.

2.7.1 Definiciones de tipo

tipos enumerados

Las definiciones de tipo se efectuan por lo común en:

En VHDL un '*enumerated type*' son aquellos definidos por el usuario. Los tipos enumerados son por lo común definidos dentro de los *package*, *architecture* o *process*, y muchas herramientas VHDL los pueden sintetizar.

2.7.2 Definicion de tipo

tipos enumerados

declaración de tipo

```
type MI_ESTADO is
        (RESET, ESPERA, RW_CICLO, INT_CICLO )
```

Veamos la sintaxis para definir un '*enumerated type*'. Primero hay que especificar el nombre del tipo, y luego listar todos los valores que un objeto del tipo puede tomar, con cada uno de ellos separado por comas.

2.7.3 Tipos definidos

Señales de tipos enumerados

```
type MI_ESTADO is (RESET, RW_CICLO,  INT_CICLO) ;
...
signal ESTADO : MI_ESTADO ;
signal DOS_BITS : bit_vector (0 to 1) ;
...
        ESTADO <= RESET ;  --legal
        ESTADO <= "00" ;   --ilegal
        ESTADO <= DOS_BITS ;--ilegal
```

Una vez definido el tipo, solo entonces podemos definir señales que sean de ese tipo. En el ejemplo, la señal ESTADO es de tipo MI_ESTADO.

Hay que observar estrictamente las reglas de asignación. A la señal ESTADO no puede asignarsele más que valores de tipo MI_ESTADO.

2.7.4 Tipos definidos

síntesis de tipos enumerados

```
type MI_ESTADO is ( RESET, ESPERA, RW_CICLO, INT_CICLO) ;
        -- codificado de izquierda a derecha en secuencia binaria
```

RESET	"00"
ESPERA	"01"
RW_CICLO	"10"
INT_CICLO	"11"

Usando el número mínimo de bits

La mayoría de las herramientas de síntesis pueden construir lógica desde una señal de tipo enumerado. Lo usual es que la señal sea codificada con el número mínimo de bits para representar todo el conjunto de posibles valores; esto para la implementación de hardware. El ejemplo muestra como se codifica cada uno de los valores en la implementación de hardware.

2.8.0 Lógica multivalor

Lógica multivalor

Veremos como se pueden representar en VHDL los sistemas lógicos con más valores que justo el '0' y '1', y entonces trataremos de un estándar el cual ha sido definido en esta área, como Lógica Estándar (Standard_Logic).

2.8.1 Lógica multivalor

¡Se necesitan más que el '0' y '1'!

- **Desconocido** Valor que fue conocido, pero ya no lo es más.
- **Sin inicializar** ¡El valor nunca fue conocido en el primer instante !
- **Alta impedancia** La conexión o *net* no tiene *driver*
- **Casos de multiples *drivers*** Manipula diferentes *drivers* de salida.
- **No importa** Tómese como mejor convenga para optimizar la implementación de síntesis

El tipo BIT tiene algunas limitaciones. A saber, que tan solo puede representar '0' y '1's. En simulación puede ser útil representar algunos otros valores, v.gr. desconocidos, sin inicializar, alta impedancia, y diferentes situaciones de multiples *drivers*. Para síntesis, resulta útil representar adicionalmente condiciones de no importa.

2.8.2 Lógica multivalor

Lógica multivalor

0 lógico **'0'**	**1 lógico** **'1'**
Alta impedancia **'Z'**	**No importa** **'-'**
Sin inicializar **'U'**	**Desconocido** **'X'**

Esto trae la idea de sistemas lógicos multivalor, o MVL. Existe un tipo enumerado en donde se definen caracteres para poder representar estos otros valores, junto con un conjunto de funciones las cuales efectuan operaciones lógicas tales como la 'and' y la 'or' sobre objetos de este tipo.

2.8.3 Lógica multivalor

Cada herramienta VHDL tiene su propio MVL

Hasta principios de los 92's no existía ningún estándar en esta área, de modo que cada una de las herramientas de simulación y síntesis usaba su sistema MVL propio, yendo desde uno con 4 hasta otro con 126 diferentes valores. Esto hace que el código sea fuertemente dependiente del proveedor, y de difícil intercambio.

Actualmente hay una MVL estándar

Standard_Logic IEEE 1164

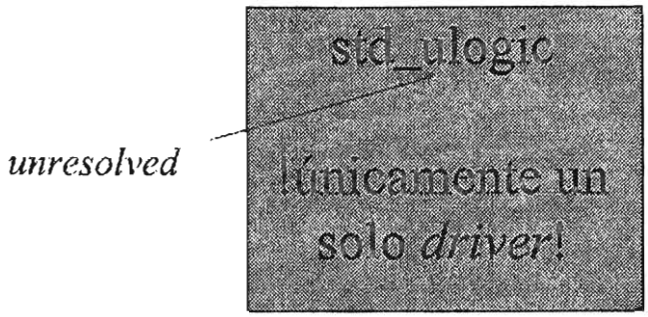
Actualmente en esta área se ha definido un estándar, en el cuál se tiene un sistema MVL con nueve valores o 'estados'. Este es un estándar aparte del que define el lenguaje mismo, y es el estándar IEEE con número 1164. Veremos con mayor detalle este tipo de datos, conocidos como **Standard_Logic**.

	type std_ulogic is (	
***Drive* fuerte**	**'u',**	sin inicializar
	'x',	desconocido
	'0',	0 lógico
	'1',	1 lógico
	'Z',	Alta impedancia
***Drive* débil** **(usado sólo ocasionalmente)**	**'W',**	Desconocido
	'L',	0 lógico
	'H',	1 lógico
	'-') ;	no importa

El estándar IEEE 1164 esta definido en código fuente VHDL puro, y describe un tipo de dato llamado 'std_ulogic' (lease estándar-u-logic), el cuál define los nueve posibles estados. La tabla muestra la definición de cada uno de dichos valores.

2.9.1 *Standard Logic*

std_ulogic

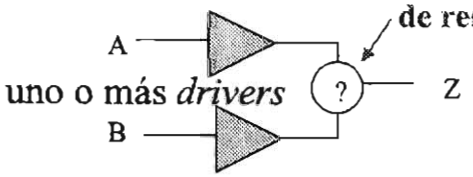


El tipo std_ulogic es un tipo de dato *unresolved*, y por lo tanto solo puede tener un *driver*. En efecto, la 'u' de su nombre significa 'unresolved'.

2.9.2 *Standard Logic*

std_logic

std_logic **lleva asociada una función de resolución**



Está definido en el mismo paquete otro tipo llamado '**std_logic**', el cuál es una versión *resolved* del std_ulogic, es decir una señal de este tipo puede tener mas de un *driver*. El tipo std_logic tiene los mismos nueve estados que el tipo std_ulogic.

Paquete standard_logic_1164

Package :

standard_logic_1164

Los tipos std_ulogic y std_logic estan definidos y contenidos dentro del paquete llamado standard_logic_1164. Más adelante veremos como hacer referencia a este paquete.

std_ulogic_vector
std_logic_vector

std_ulogic_vector | '1' | '0' | '1' | '0' | '1' |

arreglo de std_ulogic

std_logic_vector | '1' | '0' | '1' | '0' | '1' |

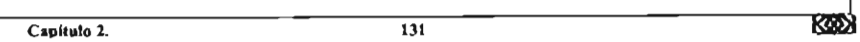
arreglo de std_logic

El paquete contiene además la definición de arreglos de std_logic y de std_ulogic, conocidos como std_logic_vector y std_ulogic_vector respectivamente. Estos tipos se usan para describir estructuras de bus en la misma forma en que lo hace el tipo bit_vector.

2.10.0 Usando *Standard Logic*

Usando standard_logic

Una vez vistas las definiciones de los tipos de datos basados en standard_logic, veamos como emplearlos.



2.10.1 Uso de *standard logic*

Bibliotecas y cláusulas de uso

hace visible la biblioteca

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MVLS is
    port (A, B : in std_logic;
          Z    : out std_logic );
end MVLS;
```

hace visible todo contenido del paquete

El paquete de definiciones está contenido dentro de una biblioteca llamada IEEE. Es necesario hacer referencia a ambos, la biblioteca y el paquete, tal como se muestra. Se deberán usar esas líneas de código antes de cada entidad que se escriba y que use standard_logic.

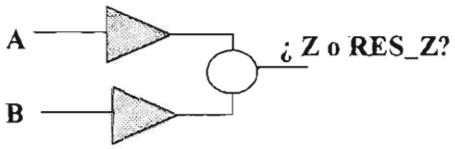
Asignar std_logic a std_ulogic

```
signal A, B, Z : std_ulogic ;
signal RES_Z : std_logic ;

Z <= A ;        legal

RES_Z <= A ;        legal

A <= RES_Z ;        legal
```

Tal como se muestra, es válido asignar libremente objetos de tipo std_logic a objetos de tipo std_ulogic en cualesquier sentido.

standard_logic para múltiples *drivers*

```
signal A, B, Z : std_ulogic ;
signal RES_Z : std_logic ;
```

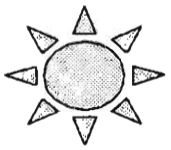


`Z <= A ;`	`RES_Z <= A ;`
`Z <= B ;`	`RES_Z <= B ;`
Ilegal	**Legal**

Si para una sola señal existen múltiples *drivers*, tendrá que usarse el tipo **std_logic**, esto debido a que es ilegal tener más de un *driver* para un dato tipo *unresolved*.

Usar std_logic

std_logic
recomendado



std_ulogic
no recomendado

Las siguientes presentaciones se basan en el uso de tipos std_logic y std_logic_vector, los cuales son los recomendados a usar en todos los diseños. La razón más poderosa es que en el VHDL existen otros estándares definidos en base a estos tipos, no así en los std_ulogic y std_ulogic_vector.

3. Operadores VHDL

3.1 ¿De que trata la tercera presentación?
3.2 Operadores Lógicos
3.3 Operadores Relacionales
3.4 Operadores Aritmeticos

3.1.0 ¿Que se presentara en esta tercera sección?

Operadores

VHDL

Trataremos sobre los operadores VHDL

3.1.1 ¿De que trata la lección?

Veremos cuales son los principales operadores

Operadores VHDL

Como hasta aquí ya conocemos lo básico sobre las señales y sus tipos asociados, es tiempo de ver los principales operadores que nos permitirán efectuar ... ¡aunque no lo crea!... operaciones entre esas señales.

3.1.2 ¿Qué más estudiaremos?

Lógicos

Operadores Lógicos

Veremos los operadores lógicos que usa el lenguaje, tal como 'and', 'or', 'xor' ...

3.1.3 ¿Qué veremos?

Relacionales

Relacionales

También veremos los operadores que se usan para la comparación entre valores, ...

3.1.4¿Y qué mas?

Aritméticos

Aritméticos

Veremos también los operadores que permiten efectuar cálculos aritméticos entre los valores que toman las señales, ...

3.2.0 Operadores lógicos

Lógicos

Para empezar, veamos algunos operadores lógicos...

3.2.1 Lógicos

and, or, nand, nor, xor, not

Los operadores lógicos que maneja el lenguaje son los mostrados aquí: **and**, **or**, **nand**, **nor**, **xor**, los cuales tienen la misma precedencia, y se ejecutan de izquierda a derecha a lo largo del postulado. El operador **not** posee la más alta precedencia, y por lo tanto es ejecutado antes que cualquier otro dentro de una expresión.

ejemplo muy sencillo

```
library IEEE ;
use IEEE.std_logic_1164.all ;

entity MultiVL is
port (A, B, C : in std_logic ;
      Z       : out std_logic ) ;
end MultiVL ;
architecture Arq of MultiVL is
begin
   Z <= A and not ( B or C) ;
end Arq ;
```

Cuidado con el uso de paréntesis

El ejemplo muestra el uso de los operadores lógicos. Se debe ser cuidadoso en el uso de los paréntesis ...

operadores lógicos estan predefinidos para...

Los operadores lógicos estan predefinidos para objetos de cualquiera de los tipos de datos que se muestran arriba.

3.2.4 Operadores lógicos

en arreglos operan sobre elementos que coinciden

```
signal A_BUS, B_BUS, Z_BUS : std_logic_vector  (3 downto 0) ;
Z_BUS <= A_BUS and B_BUS ;
equivalente a:
Z_BUS (3) <= A_BUS (3) and B_BUS (3) ;
Z_BUS (2) <= A_BUS (2) and B_BUS (2) ;
Z_BUS (1) <= A_BUS (1) and B_BUS (1) ;
Z_BUS (0) <= A_BUS (0) and B_BUS (0) ;
```

Las operaciones lógicas en el caso de los arreglos sólo tienen sentido si se aplican entre arreglos del mismo tipo y tamaño. Los operadores lógicos se aplican entonces a los elementos que coinciden por posición (*bitwise*), devolviendo un arreglo del mismo tipo y longitud.

3.3.0 Operadores relacionales

Operadores relacionales

Veamos ahora los operadores relacionales o de comparación

3.3.1 Operadores relacionales

>, >=, <, <=, /=, =

menor que → <

mayor que → >

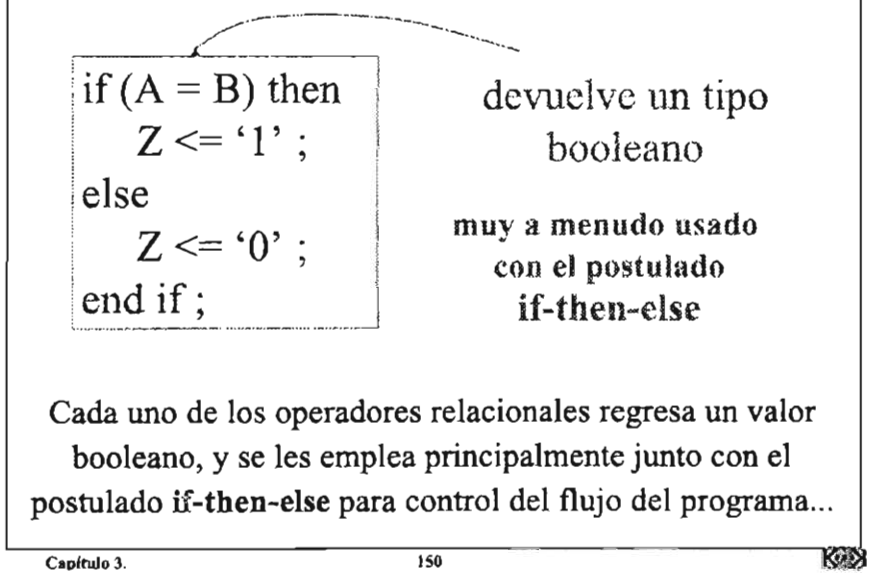
idéntico a → =

menor o igual que → <=

mayor o igual que → >=

diferente a → /=

Los operadores relacionales en el VHDL son los archiconocidos.....



3.3.2 Operadores relacionales

la comparación regresa un tipo booleano

```
if (A = B) then
    Z <= '1' ;
else
    Z <= '0' ;
end if ;
```

devuelve un tipo booleano

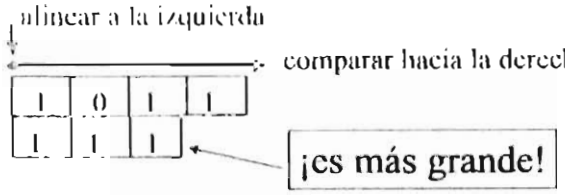
muy a menudo usado con el postulado if-then-else

Cada uno de los operadores relacionales regresa un valor booleano, y se les emplea principalmente junto con el postulado **if-then-else** para control del flujo del programa...

operandos del mismo tipo

operandos sólo del mismo tipo

arreglos de diferentes longitudes:



Las reglas de uso de los operadores relacionales exigen que los operandos sean del mismo tipo. En el caso de los arreglos los operandos pueden ser de diferente longitud: los operandos son alineados por la izquierda y comparados hacia la derecha. Así por ejemplo, "111" es mayor que "1011". ¡**Cuidado con las sorpresas**!.

los arreglos no tienen ningun significado numérico

Sin significado numérico

no es '7' sino tan solo 'tres '1's'

En VHDL no existe ningún significado numérico asociado con un vector: tan sólo es una colección de objetos del mismo tipo. Existen formas de darle la vuelta a la restricción anterior, lo que se verá más adelante en las cuestiones de la síntesis...

3.4.0 Operadores aritméticos

Operadores aritméticos

Es tiempo de que veamos los operadores aritméticos de que dispone el lenguaje VHDL...

3.4.1 Operadores aritméticos

+, -, *, /, **, rem, mod, abs

suma	+	exponenciación	**
resta	-	valor absoluto	abs
multiplicación	*	módulo	mod
division	/	resto	rem

Los operadores aritméticos del VHDL son los enlistados arriba. Estan predefinidos para los tipos *integer*, *real*, (excepto el módulo y resto), y el tipo *time*. Los operadores aritméticos no pueden ser usados con los tipos bit_vector, std_ulogic_vector, o std_logic_vector, ya que los vectores **no** pueden representar valores numéricos.

3.4.2 Operadores aritméticos

generalmente los operandos son del mismo tipo

```
signal A, B, Z : integer ;
...
  Z <=  A + B ;
```

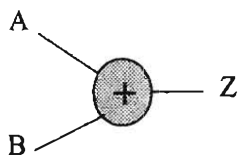
tipos '*integer*'

Por lo general los operandos necesitan ser del mismo tipo. Asi por ejemplo, la suma de dos *integer* regresa otro *integer*.

3.4.3 Operadores aritméticos

checa el rango de los enteros

```
entity SUM is
port (A, B : in integer range 0 to 7;
      Z : out integer range 0 to 15 );
end SUM;
architecture ARITMETICA of SUM is
begin
   Z <= A + B;
end ARITMETICA;
```

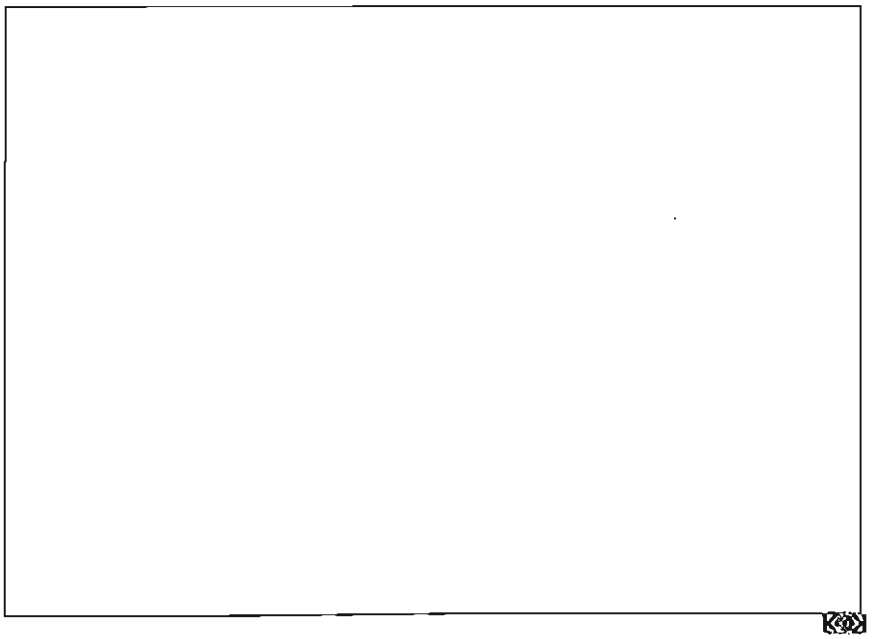


El ejemplo muestra la 'habilidad' para especificar que un *integer* caerá dentro de cierto rango. En este caso tanto A, B, y Z son del mismo tipo, pero el simulador checa que sus valores estén dentro del rango especificado al momento de la ejecución, dando un error si no se cumple lo anterior.

tiempo : multiplica /divide por un *integer* o un *real*

```
signal CLK : std_logic ;
constant PERIODO : time := 50 ns ;

wait for 5 * PERIODO ;
-- espera por 250 ns

wait for PERIODO * 1.5 ;
-- espera por 75 ns

CLK <= not CLK after PERIODO/2 ;
-- asignación despues de 25 ns
```

Es posible multiplicar o dividir un objeto de tipo *time* por un entero o por un real y regresar un valor de tipo *time*. Esto es muy útil cuando se crean estímulos con el VHDL, tal como se muestra en el ejemplo.



4. Postulados concurrentes y secuenciales

4.1.0 ¿Qué aprenderemos en esta cuarta lección?

Postulados 'Concurrentes' y 'Secuenciales'

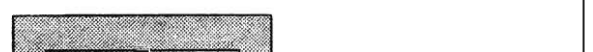
Presentación que trata sobre los postulados **concurrentes** y **secuenciales**. Veremos que la palabra concurrente es para los postulados que se ejecutan fuera de un proceso, mientras que los secuenciales, son aquellos que se ejecutan en secuencia dentro de un proceso.

Postulados Concurrentes

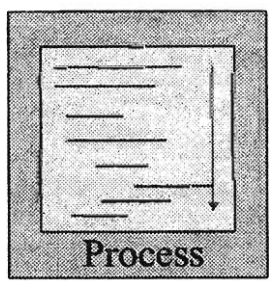
Postulados

'concurrentes'

En primer lugar revisaremos brevemente los conceptos que encierra la palabra concurrente ...



Postulados secuenciales



... y enseguida estudiaremos lo que es un **proceso** (*process*), y como el significado de los postulados cambia cuando ellos son descritos dentro de un proceso, a cuando son descritos fuera del mismo.

4.1.3 Aprenderemos que...

postulados if, case, for loop

```
if SEL = '1'
then
   Z <= A ;
else
   Z <= B ;
end if ;
```

if

```
case X is
    when 0 to 4 =>
        Z <= B ;
    when ....
```

case

```
for I in 0 to 3 loop
  if (A = I) then
    Z(I) <= '1' ;
  end if ;
end loop;
```

for loop

Por último, introduciremos algunos postulados que en VHDL únicamente pueden ser usados dentro del mundo secuencial: el postulado '*if*', el 'case', y el '*for loop*'.

4.2.0 Postulados de asignación concurrente

- Postulados concurrentes:
  se ejecutan al mismo tiempo,
  'en paralelo'
- Postulados secuenciales:
  se ejecutan uno a la vez,
  'en secuencia'

Así pues, veamos primero algunos conceptos subyacentes tras los postulados concurrentes. Un postulado concurrente es aquel que se ejecuta al mismo tiempo que otros de su misma naturaleza, es decir en paralelo. Es diametralmente opuesto a los postulados secuenciales, los cuales solo pueden ejecutarse uno a la vez, en secuencia.

Ejecución en paralelo, independencia del orden

Da lo mismo cualquier orden de escritura

```
X <= A + B ;
Y <= D + Z ;
Z <= C + X ;
```

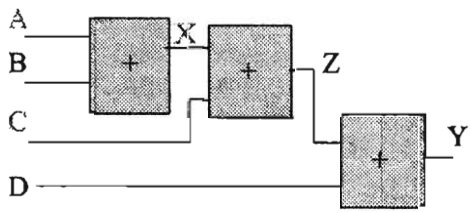
```
Y <= D + Z ;
Z <= C + X ;
X <= A + B ;
```

El comportamiento de los postulados concurrentes es independiente del orden en que ellos son escritos. Tal como se muestra en el ejemplo, los postulados pueden escribirse en uno u otro orden, obteniendose el mismo resultado. El resultado de A + B se usa como entrada en el sumador que maneja Z,...

¡tracese una representación gráfica!

```
X <= A + B ;
Y <= D + Z ;
Z <= C + X ;
```

```
Y <= D + Z ;
Z <= C + X ;
X <= A + B ;
```

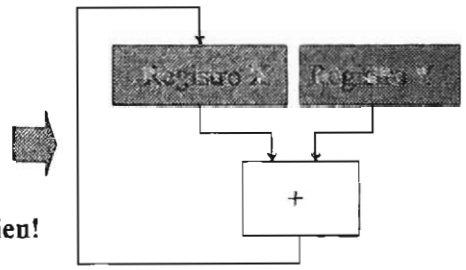


La forma más fácil de percibir la concurrencia es trazando un diagrama de las operaciones; ellos son empleados para representar hardware, de modo que representan símbolos interconectados trabajando concurrentemente. En el ejemplo anterior los tres sumadores responden inmediatamente cuando alguna de sus entradas cambia, es decir actuan en paralelo, aunque la salida de uno puede afectar a otro.

Asignación a sí mismo, ¡correcto para software...!

X <= X + Y ;

en software, un programador pensaría en el siguiente esquema



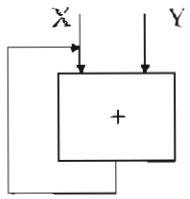
¡en software está muy bien!

Hay algunas cosas que un programador de software está acostumbrado a hacer aunque sin sentido en la descripción de hardware. Consideremos el siguiente postulado: X<=X+Y. En software X y Y son registros: tomamos el contenido de X, le sumamos el contenido de Y, y almacenamos el resultado en el mismo registro X. Todo muy bien ...

¡... no para hardware!

X <= X + Y ;

para un programador VHDL la situación es diferente

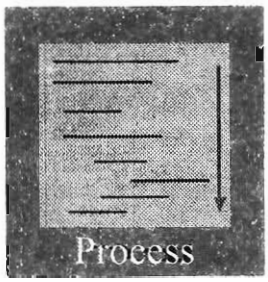


¡en hardware esta muy mal!

¡piense siempre en hardware!

Como un postulado concurrente, la misma línea de código X<=X+Y está describiendo un sumador, pero sin inferir ningún registro de almacenamiento. ¡Con eso hemos descrito retroalimentación alrededor de lógica combinacional!. Como se ve es realmente importante el hecho de que cuando escribimos código VHDL estamos en realidad describiendo hardware.

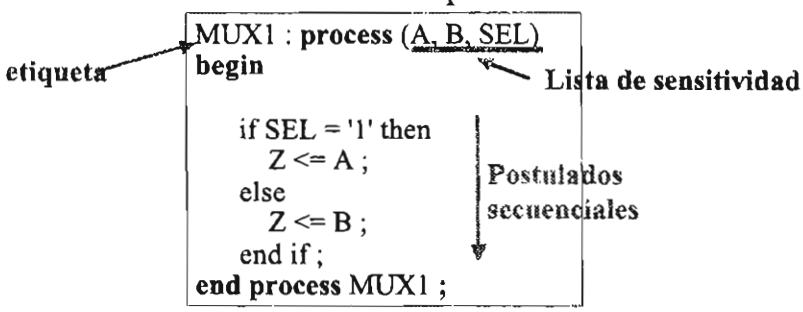
4.3.0 El proceso



Una vez aclarados algunos conceptos subyacentes en los postulados concurrentes y secuenciales, veamos cómo esos tipos de postulados interaccionan en el lenguaje VHDL. Lo hacen a través del **proceso**.....

4.3.1 El proceso

Sintaxis del proceso



Tal como se ha dicho, el proceso contiene postulados secuenciales. Lleva opcionalmente una etiqueta de identificación, seguida de la palabra clave '***process***', y luego una lista de sensitividad. Enseguida viene el '***begin***' del proceso, después de lo cuál están los postulados secuenciales hasta la palabra reservada '***end process***' seguida del nombre de la etiqueta.

4.3.2 El proceso

Eventos sobre las señales en la lista de sensitividad

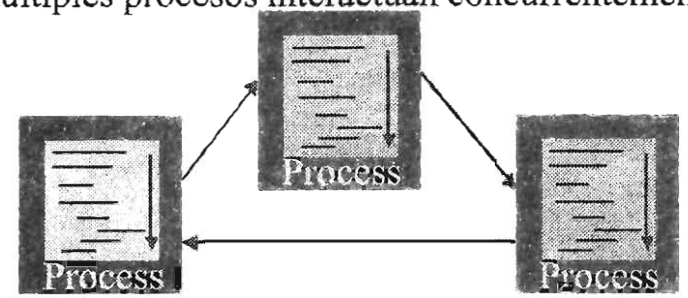
```
MUX1 : process (A, B, SEL)
begin

 if SEL = '1' then
    Z <= A ;
 else
    Z <= B ;
 end if ;
end process MUX 1;
```

Se ejecuta después que **sucede un evento** en A, B, o SEL

Un proceso no se ejecuta continuamente: él es invocado cuando alguna de las señales dentro de la lista de sensitividad cambia de valor, o como se dice en lenguaje VHDL, cuando sucede un 'evento'.

4.3.3 El proceso

Múltiples procesos interactúan concurrentemente



Mientras que cada uno de los procesos ejecuta sus propios postulados en estricta secuencia, múltiples procesos interactúan entre ellos concurrentemente, tal como se ve en el diagrama. Cada uno de los procesos se ejecuta cuando existe un 'evento' sobre alguna de las señales en su lista de sensitividad, causando que probablemente nuevos eventos ocurran en las señales asignadas por ellos.

4.3.4 El proceso

Múltiples procesos, misma arquitectura.

```
architecture A of E is
begin
        -- postulados concurrentes
        P1 : process
        begin
        -- postulados secuenciales
        end process P1 ;
        -- postulados concurrentes
        P2 : process
        begin
        -- postulados secuenciales
        end process P2 ;
        -- postulados concurrentes
end A ;
```

Un proceso debe escribirse siempre dentro de una arquitectura, y el VHDL permite que dentro de la misma puedan ser escritos cualquier número de procesos.

4.4.0 Ejecución de procesos

Ejecución de 'procesos' (*process*)

Una vez revisados los principios básicos de los procesos, ahora veremos en mayor detalle cuál es la forma de ejecución de esos procesos.

4.4.1 Ejecución de procesos

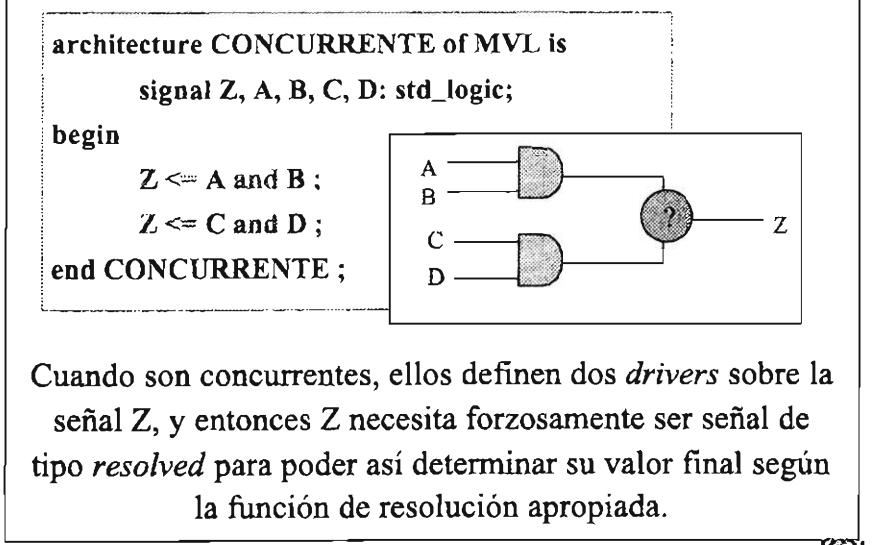
Asignaciones a la misma señal: 'concurrente' vs 'secuencial'

¿Qué pasa si son concurrentes...?

```
Z <= A and B ;
Z <= C and D ;
```

¿... y qué si son secuenciales?

Hay que cuidar que la ejecución de un proceso sea descrita en el contexto apropiado. Veamos las consecuencias de los siguientes dos postulados Z <=A and B y Z<=C and D, en donde cada uno hace una asignación a la misma señal. Consideraremos primero la conducta considerándolos postulados concurrentes, y luego como postulados secuenciales.



4.4.2 Ejecución de procesos

Concurrente: dos *drivers*

```
architecture CONCURRENTE of MVL is
        signal Z, A, B, C, D: std_logic;
begin
        Z <= A and B ;
        Z <= C and D ;
end CONCURRENTE ;
```

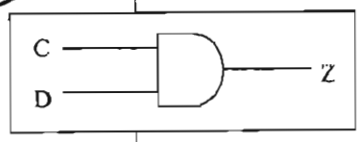

Cuando son concurrentes, ellos definen dos *drivers* sobre la señal Z, y entonces Z necesita forzosamente ser señal de tipo *resolved* para poder así determinar su valor final según la función de resolución apropiada.

4.4.3 Ejecución de procesos

Secuencial: un solo *driver*, la última asignación

```
architecture SECUENCIAL of MVL is
        signal  Z, A, B, C, D: std_logic ;
begin
        process (A, B, C, D)
        begin
                Z <= A and B ;
                Z <= C and D ;
        end process ;
end SECUENCIAL ;
```

Valor asignado finalmente



Cuando los postulados son secuenciales, su conducta cambia. Debido a que un proceso es por sí mismo un solo postulado concurrente, entonces un proceso 'coloca' solo un *driver* sobre una señal. El valor de esa señal es actualizado con el último valor asignado a ella dentro de la ejecución del proceso .

4.4.4 Ejecución de procesos

La señal se actualiza cuando el proceso se suspende

```
architecture SECUENCIAL of MVL is
        signal Z, A, B, C, D : std_logic;
begin
        process (A, B, C, D)
        begin
                Z <= A and B;
                Z <= C and D;
        end process;
end SECUENCIAL;
```

Z se actualiza tan solo después de que el proceso se suspende

Para asegurar que la señal se actualice en una transición limpia, las señales asignadas dentro de un proceso tan solo se actualizan a su nuevo valor hasta que el proceso se suspende al ejecutar el *end process*. Así en el ejemplo de arriba, Z será actualizada únicamente con el valor C and D, y NUNCA se actualiza con el valor de A and B.

4.5.0 Lista de sensitividad completa

Listas

de

sensitividad

Ahora que nos hemos familiarizado con la forma en que se ejecutan los procesos, veamos algunas cuestiones relativas a su lista de sensitividad

4.5.1 Listas de sensitividad completa

Ejemplo de MUX

```
MUX : process (A, B, SEL)
begin
        if SEL = '1' then
                Z <= A ;
        else
                Z <= B ;
        end if ;
end process MUX ;
```

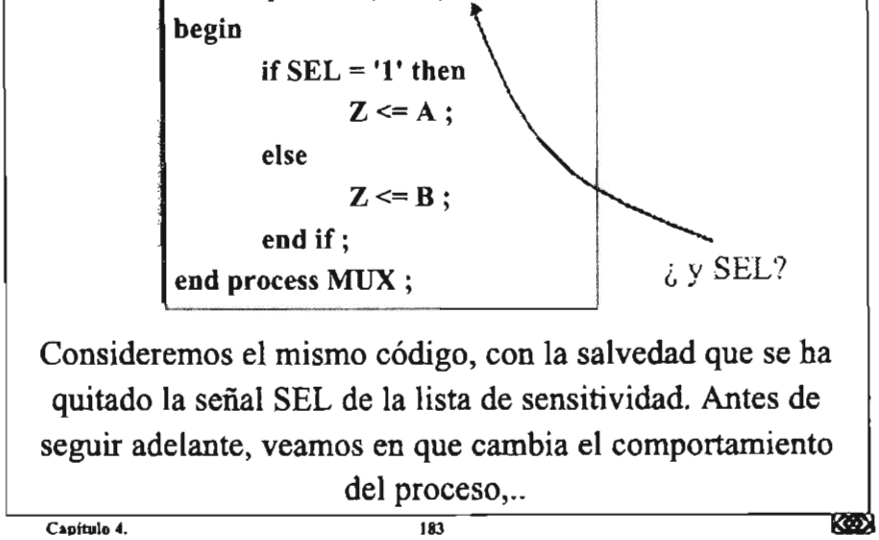
El código describe un MULTIPLEXOR 2a1: Si existe un evento sobre A, B, o SEL entonces se llama al proceso MUX, y dependiendo del valor de SEL la asignación a Z será A o B.

Si se pierde SEL: ¿qué pasa?

```
MUX : process (A, B )
begin
        if SEL = '1' then
                Z <= A ;
        else
                Z <= B ;
        end if ;
end process MUX ;
```

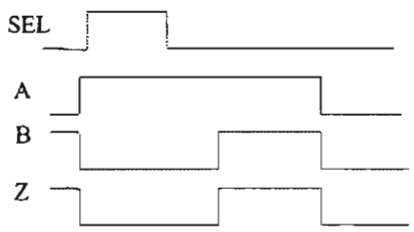
¿ y SEL?

Consideremos el mismo código, con la salvedad que se ha quitado la señal SEL de la lista de sensitividad. Antes de seguir adelante, veamos en que cambia el comportamiento del proceso,..



La Z falla algunas veces

```
MUX: process (A, B )
begin
        if SEL = '1' then
                Z <= A ;
        else
                Z <= B ;
        end if ;
end process MUX ;
```



Un proceso únicamente se ejecuta cuando un evento sucede sobre A o B. Cuando un evento ocurre sobre SEL, cualquier cambio correspondiente a Z fallará, ya que tan solo los eventos sobre A y B son tomados en cuenta. Este error puede ser muy difícil de depurar, ya que Z algunas veces posee el valor correcto y otras veces no.

4.5.4 Listas de sensitividad completa

Lógica combinacional: lista de sensitividad completa

```
MUX : process (A, B, SEL )
begin
        if SEL = '1' then
                Z <= A ;
        else
                Z <= B ;
        end if ;
end process MUX ;
```

Síntesis de lógica combinacional: ¡Lista de sensitividad completa!

Luego entonces, podemos tener una regla acerca de la descripción de lógica **combinacional** mediante procesos: la lista de sensitividad siempre deberá estar completa, es decir deberá listarse en ella cada una de las señales cuyos valores son leidos dentro del proceso. Esta es una regla que debe ser observada estrictamente en todas las herramientas de síntesis disponibles actualmente.

4.6.0 El postulado if

El postulado 'if'

Una vez vistas las bases sobre la ejecución de procesos, ahora veremos tres postulados que únicamente pueden ser usados dentro de código VHDL secuencial: primero el postulado '**if**', luego el '**case**', y por último '**for loop**'.

4.6.1 El postulado if

sintaxis de if-then-else

```
if CONDICION then
        -- postulados secuenciales
end if ;
```

```
if CONDICION then
        -- postulados secuenciales
else
        -- postulados secuenciales
end if ;
```

El postulado if prueba una condición, y dependiendo del resultado puede ejecutar diferentes conjuntos de postulados. Las formas más socorridas son mostradas aquí: la forma if-then-end if y la if-then-else-end if. Notese que es *end if* no *endif*.

4.6.2 El postulado if

estructura if-elsif

```
if CONDICION1 then
        -- postulados secuenciales
elsif CONDICION2 then
        -- postulados secuenciales
elsif CONDICION3 then
        -- postulados secuenciales
else
        -- postulados secuenciales
end if ;
```

se ejecuta únicamente una sola rama

El postulado if-elsif permite que una serie de condiciones sean probadas, siendo la primer condición verdadera en donde se entra a proseguir la ejecución de los postulados secuenciales dentro de ella. Esto sucede tan solo para una rama de la estructura: el flujo entonces pasa al postulado final 'end if'.

If-elsif: el primer postulado verdadero es ejecutado

```
if CONDICION1 then
        -- postulados secuenciales
elsif CONDICION2 then
        -- postulados secuenciales
elsif CONDICION3 then
        -- postulados secuenciales
else
        -- postulados secuenciales
end if ;
```

la primera rama verdadera es la única que se ejecuta

El orden en el cual se escriben los postulados dentro de la estructura if-elsif es muy importante. Puede suceder que más de una de las condiciones sea verdadera, siendo la primer condición verdadera encontrada la seleccionada para que los postulados que contiene sean los únicos en ser ejecutados.

ejemplo: if-elsif

```
process (A, B, C, X)
begin
        if (X = "0000") then
                Z <= A ;
        elsif (X = "0000") then
                Z <= B ;
        else
                Z <= C ;
        end if ;
end process ;
```

si X="0000" esta es la rama que se ejecuta

Si X tiene un valor "0000", entonces ambos conjuntos de condiciones son ciertos. Sin embargo, ya que la condición X="0000" se cumple en la primer rama, entonces a Z se le asigna el valor de A y nunca el de B. De lo anterior podemos ver que el postulado if-elsif tiene interconstruida una prioridad.

4.7.0 El postulado case

El postulado *'case'*

Ahora presentamos un segundo postulado secuencial: el '*case*'

4.7.1 El postulado case

Prueba posibles valores de un objeto

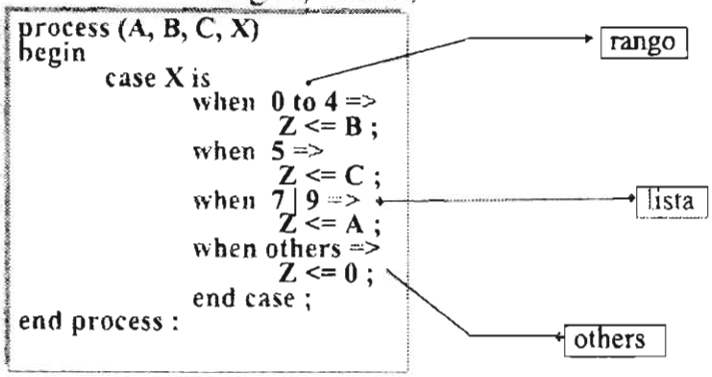
```
case OBJETO is
        when VALOR_1 =>
                --postulados secuenciales
        when VALOR_2 =>
                --postulados secuenciales
        when VALOR_3 =>
                --postulados secuenciales
        --etc....
end case ;
```

objeto

valores posibles

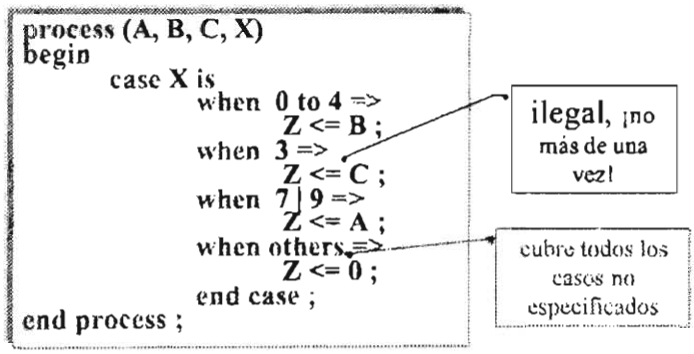
El postulado 'case' considera todos los valores posibles que un objeto puede tomar, y ejecuta los postulados secuenciales dentro de una sola rama dependiendo del valor actual de dicho objeto.

Rangos, listas , *others*



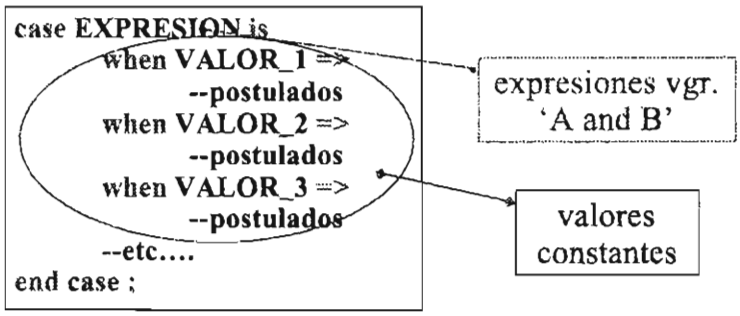
La sintaxis del postulado *case* esta hecha para proporcionar mayor control sobre la forma de especificar los valores posibles. El ejemplo muestra un rango '0 to 4', una lista de valores separados por el simbolo '|' (barra), y la cláusula '***others***' para cubrir todos los posibles valores de X que hasta aquí no hayan sido especificados.

Cubrir todas los valores, pero sólo una vez



En VHDL las reglas de uso del postulado ***case*** son muy estrictas: no se permite de ningún modo especificar más de una sola vez uno de los posibles valores, y todos los valores deben ser cubiertos, sea explícitamente, o mediante la cláusula '*others*'.

Los objetos pueden ser expresiones, las opciones son constantes



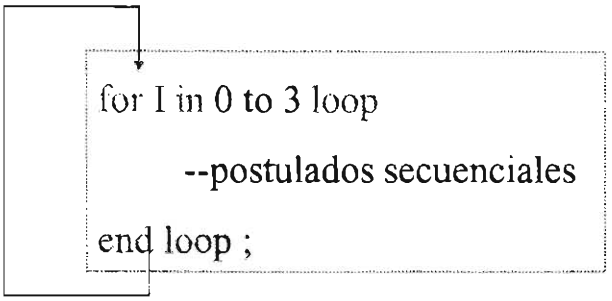
Finalmente, el objeto de referencia puede ser efectivamente cualquier expresión válida en VHDL. Los valores listados en las ramas del postulado *case* deben ser valores constantes, y del mismo tipo que el de la expresión.

Postulado *'for loop'*

Por último otro postulado secuencial que veremos será el *for loop*

4.8.1 El for loop

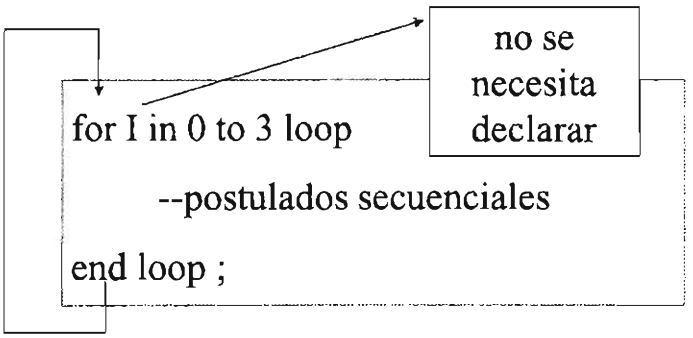
iteraciones de *loop*, valor incrementándose



La escencia del postulado *for-loop* es la iteración alrededor de un bucle (*loop*) un número fijo de veces, mientras se incrementa o decrementa el valor I con cada iteración a través de un rango fijo. En el ejemplo, se hacen cuatro pasadas alrededor del *loop*.

4.8.2 El *for loop*

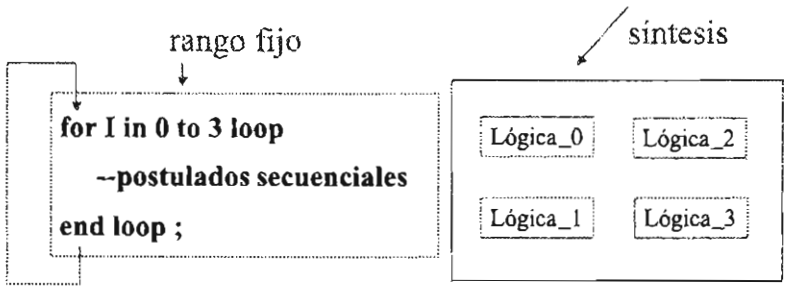
No hay necesidad de declarar la variable de *loop*



No es necesario declarar la variable de *loop* (I) en ninguna parte del código: implícitamente se declara con el solo hecho del uso del postulado for-loop.

4.8.3 El *for loop*

Es adecuada para la síntesis



La mayoría de las herramientas de síntesis VHDL soportan el postulado *for-loop* cuando el rango de iteración es fijo para combinacionales. La síntesis expande ('*unfold*') el *loop*, y replica la lógica dentro de ella una vez por cada iteración, aunque posteriormente optimice la implementación al simplificar algebraicamente.

4.8.4 El *for loop*

Ejemplo

```
entity EJEMPLO is
port (A : in std_logic_vector (0 to 15) ;
        SEL : in integer range 0 to 15 ;
        Z : out std_logic ) ;
end EJEMPLO ;
architecture RTL of EJEMPLO is
begin
        Desconocido : process (A, SEL)
        begin
                for I in 0 to 15 loop
                        if SEL = I then
                                Z <= A(I) ;
                        else
                                Z <= '0' ;
                        end if ;
                end loop ;
        end process Desconocido ;
end RTL ;
```

He aquí un ejemplo que usa el postulado *for loop*. El comportamiento que sigue será colocar en cascada dieciséis Mux2a1. El proceso será ejecutado dieciséis iteraciones alrededor del *loop*.

5. Postulados secuenciales

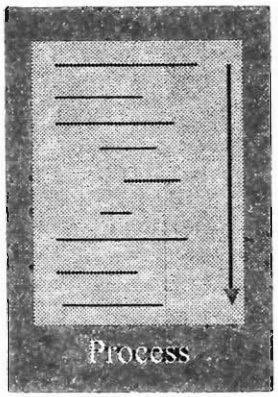
5.1.0 ¿Que veremos ?

Postulados 'secuenciales',....

Empecemos viendo cuales serán los puntos en que centraremos nuestro interés en esta presentación,...

5.1.1 ¿Qué veremos?

Finalidad

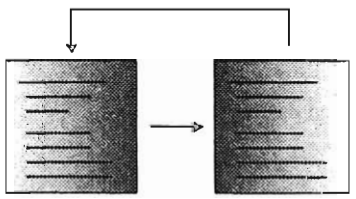


Process

Nuestro objetivo es estudiar algunas carácterísticas adicionales de los procesos, y cómo utilizarlas en la generación de código,...

5.1.2 ¿Qué más aprenderemos?

Ejecución de procesos



actualización de señales

ejecución de procesos

ejecución de un proceso

Se mostrará como se ejecuta un solo proceso, y después tratar el caso de múltiples procesos, con la finalidad de tener un pictograma de cómo se efectua una simulación completa en VHDL.

5.1.3 ¿Qué más aprenderemos?

Postulado *'wait'*

```
ESTIMULOS : process
begin
        SEL <= '0' ;
        BUS_B <= "0000" ;
        BUS_A <= "1111" ;
        wait for 10 ns ;
                SEL <= '1' ;
        wait for 10 ns ;
        --etc,....
end process ESTIMULOS ;
```

Finalmente veremos más en detalle el postulado 'wait', y como afecta la ejecución de un proceso.

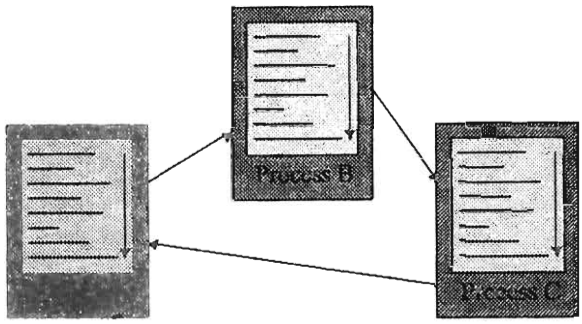
5.2.0 Concurrente vs secuencial

Conceptos previos

Recordemos algunos conceptos relacionados, que serán de utilidad ahora...

5.2.1 Concurrente vs secuencial

modelo de proceso



Recordemos que nuestro modelo de simulación VHDL consiste de múltiples procesos ejecutándose cada uno de ellos secuencialmente, e interactuando entre ellos concurrentemente.

5.2.2 Concurrente vs secuencial

Procesos dentro de la arquitectura

```
architecture A of E is
begin
        --postulados concurrentes
        P1 : process
        begin
        --postulados secuenciales
        end process P1 ;
        --postulados concurrentes
        P2 : process
        begin
        --postulados secuenciales
        end process P2 ;
        --postulados concurrentes
end A ;
```

Se había visto que este modelo se mapea al código VHDL escribiendo cualquier número de procesos dentro de una misma arquitectura...

5.2.3 Concurrente vs secuencial

Concurrente dentro de la arquitectura

```
architecture A of E is
begin
        --postulados concurrentes
        P1 : process
        begin
        --postulados secuenciales
        end process P1 ;
        --postulados concurrentes
        P2 : process
        begin
        --postulados secuenciales
        end process P2 ;
        --postulados concurrentes
end A ;
```

....y que cada proceso se ve tan solo como un postulado concurrente dentro de la arquitectura.

5.2.4 Concurrente vs secuencial

Multiples procesos dentro de *architecture*

```
architecture A of E is
begin
        --postulados concurrentes
        P1 : process
        begin
                --postulados secuenciales
        end process P1 ;
        --postulados concurrentes
        P2 : process
        begin
                --postulados secuenciales
        end process P2 ;
        --postulados concurrentes
end A ;
```

Como ya se dijo, una arquitectura dada puede contener cualquier número de postulados concurrentes, y por lo tanto cualquier número de procesos.

5.3.0 Asignación de señales en un proceso

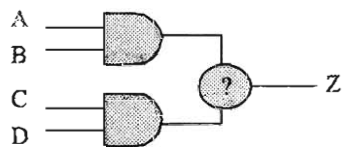
Asignación de señales dentro de un proceso

Otro concepto que debemos tener presente, es cómo se actualizan dentro de un proceso los valores asignados a las señales.

5.3.1 Asignación de señales dentro de un proceso

Postulados concurrentes: múltiples *drivers*

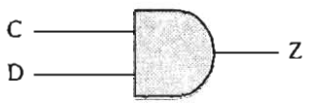
```
architecture CONCURRENTE of MULTIPLE is
        signal  Z, A, B, C, D : std_logic ;
begin
        Z <= A and B ;
        Z <= C and D ;
end CONCURRENTE ;
```



Recuérdese este ejemplo donde a la señal Z se le asignan dos valores desde procesos diferentes. Ya que los postulados declarados fuera de un proceso son concurrentes, por lo tanto en este caso Z tiene dos *drivers* y debe entonces ser de tipo *resolved*, para que su valor final pueda determinarse desde la función de resolución.

Dentro de un proceso: un solo *driver*

```
architecture SECUENCIAL of MULTIPLE is
        signal Z, A, B, C, D : std_logic ;
begin
        process (A, B, C, D)
        begin
                Z <= A and B ;
                Z <= C and D ;
        end process ;
end SECUENCIAL ;
```



Habíamos visto que cuando a la señal Z se le hacen dos o más asignaciones pero dentro de un mismo proceso, a la señal se le asigna un solo *driver*...

La asignación final toma efecto cuando el proceso se suspende

```
architecture SECUENCIAL of MULTIPLE is
        signal Z, A, B, C, D : std_logic ;
begin
        process (A, B, C, D)
        begin
                Z <= A and B ;
                Z <= C and D ;
        end process ;
end SECUENCIAL ;
```

esto toma efecto

cuando el proceso se suspende

El LRM define que dentro de un proceso la asignación a la señal hecha al último es la que toma efecto, pero tan solo en el momento en que el proceso se suspende, que en este ejemplo es cuando se ejecuta la última línea *end process*.

5.3.4 Asignación de señales dentro de un proceso

¡Diferente significado!

```
process ( A, B, C, D)
begin
        Z <= A and B ;
        Z <= C and D ;
end process ;
```

dentro de una arquitectura

≠

dentro de un proceso

```
architecture X of MULTIPLE is
        ...
begin
        Z <= A and B ;
        Z <= C and B ;
end X ;
```

Los mismos postulados tienen significados muy diferentes, dependiendo de sí ellos son ejecutados dentro o fuera de un proceso.

5.3.0 Múltiples llamadas a un proceso (1)

¿ Múltiples llamadas a un solo proceso?

Debido a la forma de actualización de las señales dentro de un proceso, ahora veremos como un proceso puede potencialmente ser ejecutado varias veces antes de que todas las señales sean actualizadas.

5.4.1 Múltiples llamadas a procesos (1)

Ejemplo: señal asignada, y luego leida

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M;
end process EJEMPLO ;
```

primero se asigna

luego se lee

Este ejemplo hace uso de una asignación a la señal M, pero el valor de M es posteriormente leído dentro del mismo proceso. Analicemos este caso para ver más detalladamente como se ejecutan los procesos en VHDL.

5.4.2 Múltiples llamadas a procesos (1)

'Evento' sobre B

llamado al proceso, ejemplo

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M ;
end process EJEMPLO ;
```

A: 0
B: 1
M: 0
Z: 0

evento en B

Supongamos que sucede un evento a la señal B, señal que digamos ha cambiado su valor de 0 a 1. Con eso, el proceso EJEMPLO es llamado, ya que B forma parte de la lista de sensitividad.

5.4.3 Múltiples Llamadas a procesos (1)

M se asigna conforme el proceso se suspende

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M ;
end process EJEMPLO ;
```

deberá asignarse, **¡pero**
ue el proceso se suspenda!

A: 0
B: 1
M: 0
Z: 0

el valor de M aún es 0, ¡no
se ha actualizado hasta aquí!

El valor de M no se actualiza cuando la línea 'M<=B' es leída, sino hasta cuando el proceso se suspende (hasta '*end process*', al fondo).

5.4.4 Múltiples Llamadas a procesos (1)

¡Z no se actualiza!

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M ;
end process EJEMPLO ;
```

a Z se le
asignará 0,
es decir,
no cambiará

M aún contiene 0

A: 0
B: 1
M: 0
Z: 0

Así pues, cuando la línea 'Z <= M' se ejecuta, el valor de M es aún 0, y por lo tanto Z no se actualiza con 1 una vez que el proceso se suspende.

5.5.0 Llamadas multiples a procesos (2)

El proceso se suspende: M se actualiza, Z no lo hace

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M ;
end process EJEMPLO ;
```

el proceso se suspende

M se actualiza

Z no se actualiza

A: 0
B: 1
M: 1
Z: 0

Después de que el proceso se suspende, M se actualiza con su nuevo valor de 1, pero Z permanece en 0.

5.5.2 Llamadas a múltiples procesos (2)

El proceso es llamado de nueva cuenta

el proceso es llamado otra vez

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M ;
end process EJEMPLO ;
```

A: 0
B: 1
M: 1
Z: 0

evento en M

El proceso EJEMPLO es llamado de nueva cuenta ya que recién ha sucedido un evento sobre M, y dicha señal está presente en la lista de sensitividad.

5.5.3 Llamadas a procesos múltiples (2)

Z cambia a 1 en la segunda llamada

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M;
end process EJEMPLO ;
```

cuando el proceso se suspenda, Z será actualizada con M

Z vale 0 mientras no se suspenda el proceso

A: 0
B: 1
M: 1
Z: 0

Ahora en la segunda llamada, Z<=M producirá que Z se actualice con el valor 1 cuando el proceso se suspende en la línea del fondo.

5.5.4 Llamadas múltiples a un proceso (2)

¡Por fin Z se actualiza!

```
EJEMPLO : process (A, B, M)
begin
        Y <= A ;
        M <= B ;
        Z <= M ;
end process EJEMPLO ;
```

al suspenderse el proceso

A: 0
B: 1
M: 1
Z: 1

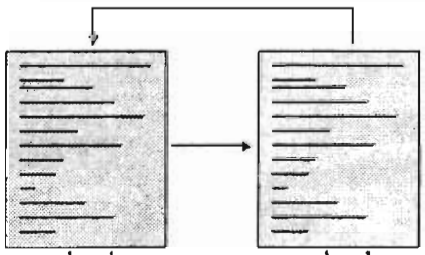
la señal Z se actualiza con 1

Esta vez cuando el proceso EJEMPLO se suspende, la señal Z sufre un evento y finalmente se actualiza con el valor 1.

¿Cómo se ejecuta un ciclo de simulación VHDL?

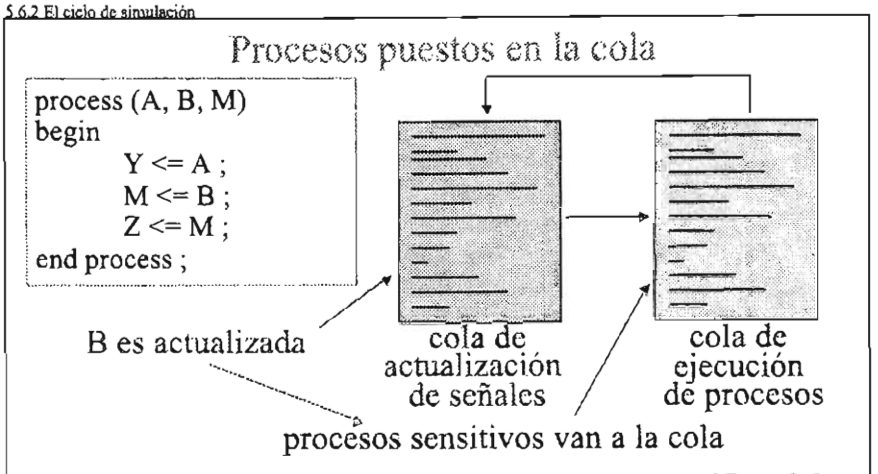
Se presentará mediante un pequeño ejemplo 'el como' se lleva a cabo una simulación VHDL. Este ejemplo ajusta en un cuadro más general...

Mecanismo de cola



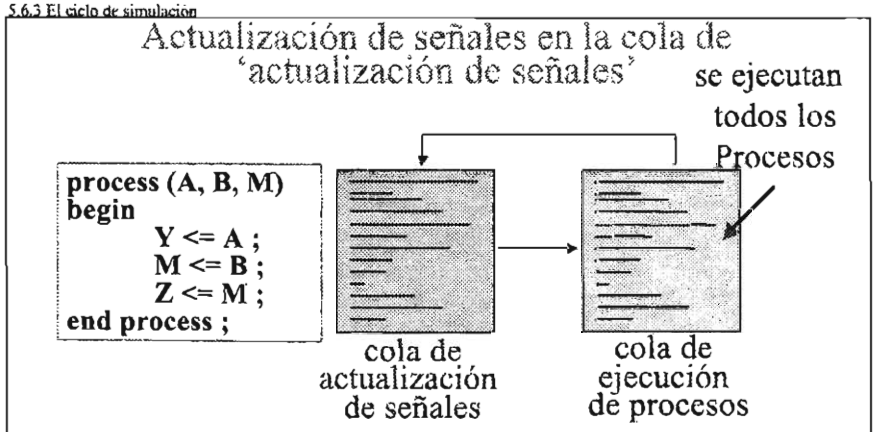
En un punto dado del **tiempo de simulación** existirán dos colas: una cola de **'señales a ser actualizadas'**, y otra de **'procesos a ser ejecutados'**.

5.6.2 El ciclo de simulación



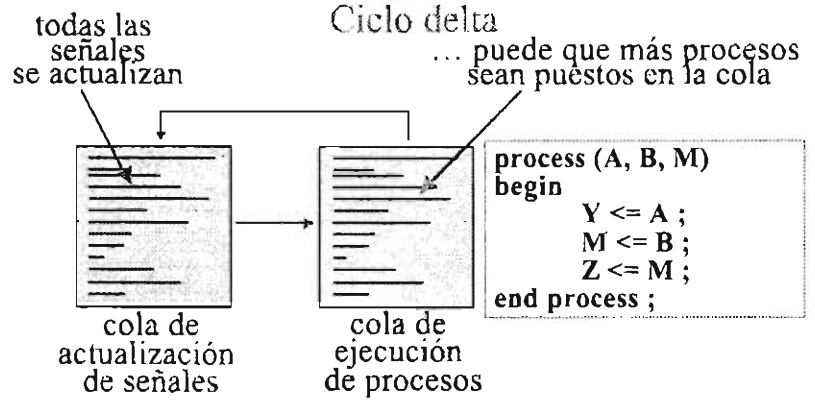
Cuando una señal se actualiza en un punto específico del tiempo de simulación, es decir sucede un evento sobre ella, todos los procesos que son sensitivos a esa señal se colocan en la cola 'Ejecución de Procesos'.

5.6.3 El ciclo de simulación



Cada uno de los procesos puestos en la cola se ejecuta por turno, y las señales a ser asignadas en cada uno de los procesos no se actualizan en cuanto el proceso se suspende, sino que son colocadas en la cola 'Actualización de Señales'. Cuando todos los procesos activos se han ejecutado, solo entonces las señales se actualizan.

5.6.4 El ciclo de simulación



Como resultado de la actualización de las señales en la cola 'Actualización de Señales', algunos procesos adicionales pueden ser colocados en la cola de 'Ejecución de Procesos'. A la iteración alrededor de esta secuencia se le conoce como 'ciclos delta'.

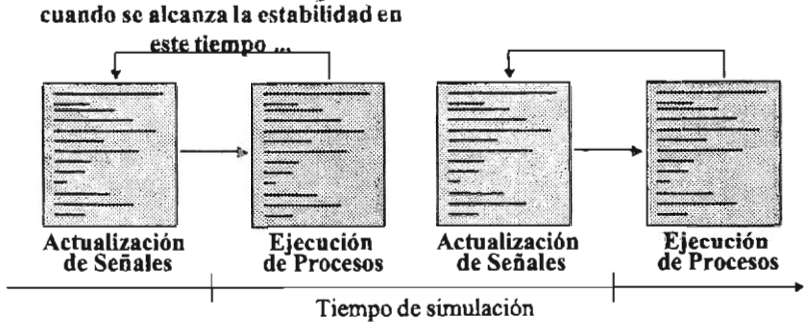
5.7.0 El ciclo delta y el tiempo de simulación

Ejecución sobre el tiempo....

Es importante recordar que múltiples ciclos delta están sucediendo en el **mismo punto del tiempo de simulación**: veamos como este modelo puede ser extendido para incluir el tiempo de simulación....

5.7.1 El ciclo delta y el tiempo de simulación

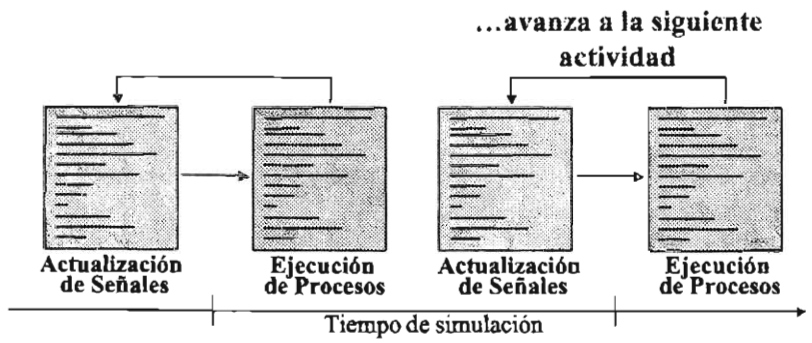
Múltiples deltas en cada uno de los puntos del tiempo de simulación



El modelo completo para una simulación VHDL consiste de múltiples ciclos delta en cada uno de los puntos del tiempo de simulación: ejecución en un solo punto del tiempo de simulación, hasta que no existan en cola ni más asignaciones de señales, ni más ejecución de procesos que hacer.

5.7.2 El ciclo delta y el tiempo de simulación

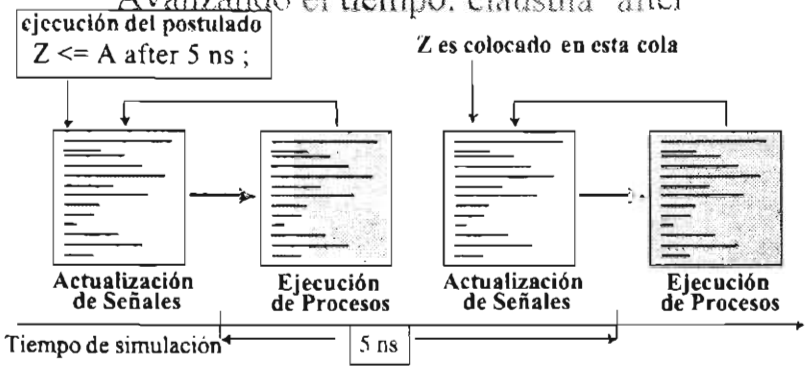
El tiempo avanza



Solo entonces el tiempo de simulación avanza al siguiente tiempo, en donde hay tanto procesos por ejecutar, como señales a actualizar.

5.7.3 El ciclo delta y el tiempo de simulación

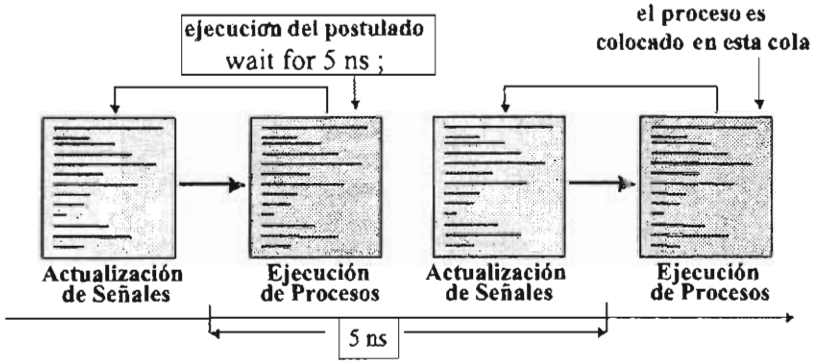
Avanzando el tiempo: cláusula 'after'



En VHDL tan solo existen dos postulados que ocasionan que el tiempo avance. Uno de ellos es la asignación de señales junto con el uso de la cláusula '*after*', ocasionando con esto que la asignación sea puesta en la cola de actualización de señales para un tiempo en el futuro en lugar del tiempo presente.

5.7.4 El ciclo delta y el tiempo de simulación

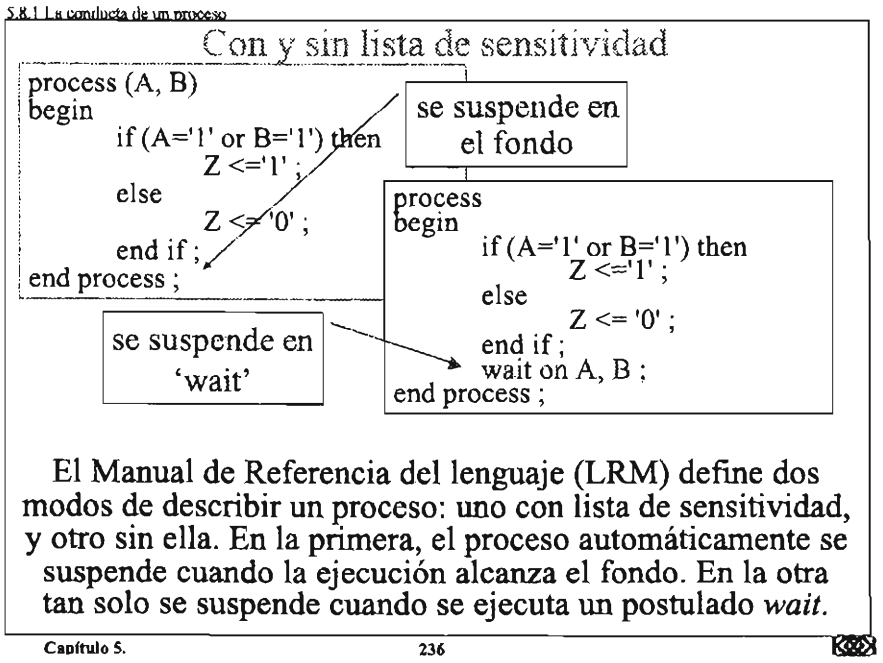
Avanzando el tiempo: el postulado '*wait for*'



La segunda vía para avanzar en el tiempo, es cuando un proceso se suspende hasta un tiempo en el futuro: esto puede lograrse usando el postulado '*wait for*'. Veremos después este postulado con mayor detalle.

Modelo completo del proceso ...

Hasta aquí hemos considerado el comportamiento de procesos con lista de sensitividad: ahora ampliremos el modelo para cubrir más capacidades de los procesos VHDL.



El Manual de Referencia del lenguaje (LRM) define dos modos de describir un proceso: uno con lista de sensitividad, y otro sin ella. En la primera, el proceso automáticamente se suspende cuando la ejecución alcanza el fondo. En la otra tan solo se suspende cuando se ejecuta un postulado *wait*.

5.8.2 La conducta de un proceso

Bucle infinito

iteración infinita

```
process
begin
        if (A='1' or B='1') then
                Z <='1' ;
        else
                Z <= '0' ;
        end if ;
        wait on A, B ;
end process ;
```

Un proceso **sin** lista de sensitividad se define como uno de bucle infinito, de modo que cuando la ejecución alcanza el fondo, automáticamente arranca a ejecutarse nuevamente desde la primer línea.

5.8.3 La conducta de un proceso

Ningun *wait* si hay lista de sensitividad

```
process (A, B)
begin
        if (A='1' or B='1')
then
                Z <='1' ;
        else
                Z <= '0' ;
        end if ;
end process ;
```

ningun wait..

```
process
begin
        if (A='1' or B='1') then
                Z <='1' ;
        else
                Z <= '0' ;
        end if ;
        wait on A, B ;
end process ;
```

equivalente al caso de lista de sensitividad

El LRM define que un proceso con lista de sensitividad no debe contener postulados '*wait*'.Por lo tanto es equivalente escribir un proceso con lista de sensitividad a uno con un solo *wait* en el fondo, el cuál espera para un evento en una o más de las señales igual que si estuviesen en la lista de sensitividad de su proceso equivalente.

La forma con lista de sensitividad para síntesis

preferido para síntesis

```
process (A, B)
begin
        if (A='1' or B='1')
then
                Z <='1' ;
        else
                Z <= '0' ;
        end if ;
end process ;
```

equivalentes

```
process
begin
        if (A='1' or B='1') then
                Z <='1' ;
        else
                Z <= '0' ;
        end if ;
        wait on A, B ;
end process ;
```

Entonces por definición, los dos procesos del ejemplo son equivalentes. Sin embargo, para la descripción de **lógica combinacional** se recomienda usar la forma **CON** lista de sensitividad, ya que es aceptada por todas las herramientas de síntesis.

El postulado '*wait*' ...

El postulado '*wait*' provoca que un proceso se suspenda, veamos las cuatro formas del postulado '*wait*', y las diferentes maneras en que 'esas formas' causan que un proceso se re-ejecute.

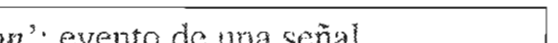
5.9.1 El postulado *wait*

wait for: periodo de tiempo

```
ESTIMULOS : process
begin
        SEL <= '0' ;
        BUS_B <= "0000" ;
        BUS_A <= "1111" ;
        wait for 10 ns ;
        SEL <= '1' ;
        wait for 10 ns ;
        --etc. .
end process ESTIMULOS ;
```

wait for <especificar tiempo>;

La primera forma es '*wait for*', mediante la cuál el proceso se re-ejecuta después de que ha transcurrido cierta cantidad especificada de tiempo. Esto puede ser útil en los *test bench* para crear estímulos, donde se requiere que las señales cambien de valor conforme el tiempo avanza.



5.9.2 El postulado *wait*

'*Wait on*': evento de una señal

```
process
begin
        if (A='1' or B='1')
then
                Z <='1' ;
        else
                Z <= '0' ;
        end if ;
        wait on A, B ;
end process ;
```

wait on <lista de señales> ;

El postulado '*wait on*' espera por un evento en una o más de las señales en su lista de señales antes de que el proceso se re-ejecute.

5.9.3 El postulado *wait*

'*wait until*': condición y evento

```
process
begin
        wait until CLK='1' ;
        Q <= D ;
end process ;
```

```
wait until <condición> ;
```

El postulado '*wait until*' tiene una condición que devuelve un tipo booleano. El postulado espera por un evento en alguna de las señales en la condición, y si como resultado del evento la condición viene a ser '*true*', entonces el proceso se reinicia.

5.9.4 El postulado wait

'*wait*': para siempre

```
ESTIMULOS : process
begin
        SEL <0 '0' ;
        BUS_B <= "0000" ;
        BUS_A <= "1111" ;
        wait for 10 ns ;
        SEL <= '1' ;
        wait ;
end process ESTIMULOS ;
```

```
wait ;
```

Finalmente, es posible decir simplemente '*wait*' para suspender para siempre un proceso. Esto puede ser útil por ejemplo en un *test bench* para evitar que la característica de bucle infinito de un proceso haga que el estímulo se repita indefinidamente una vez que ha completado un ciclo.

5.10.0 Variables

```
process (A, B, C)
        variable M, N : integer ;
begin
        M := A ;
        N := B ;
        Z <= M + N ;
        M := C ;
        Y <= M + N ;
end process ;
```

Veamos un nuevo objeto en VHDL: la **variable**.

5.10.1 Variables

Asignadas inmediatamente

```
process (A, B, C)
        variable M, N : integer ;
begin
        M := A ;
        N := B ;
        Z <= M + N ;
        M := C ;
        Y <= M + N ;
end process ;
```

A + B

C + B

La variable es un objeto VHDL que únicamente puede declararse y usarse dentro de un proceso. A las variables se les asigna un valor en el momento de ejecución de la línea, es decir inmediatamente, tal como en el software tradicional.

5.10.2 Variables

Usada solo en el proceso donde se declara

```
process (A, B, C)
        variable M, N : integer ;
begin
        M := A ;
        N := B ;
        Z <= M + N ;
        M := C ;
        Y <= M + N ;
end process ;
```

alcance

Para que la asignación de variables sea más fácil de reconocer, se usa la notación ':=' en lugar de '<=', la cual es dedicada a la asignación de señales. Una variable solo puede usarse dentro del proceso en que se declara: esto se conoce como el 'alcance' de la variable.

5.10.3 Variables

Retiene su valor

```
process (A, B, C)
        variable M, N : integer ;
begin
        M := A ;
        N := B ;
        Z <= M + N ;
        M := C ;
        Y <= M + N ;
end process ;
```

Retiene su valor.

Una variable retiene su valor entre llamadas a un proceso, y de ese modo es posible implicar la conducta de un elemento de memoria si ello se requiere. Para evitar cualquier problema cuando se describe lógica combinacional, siempre se debe asignar un valor a las variables antes de que su valor sea leído, tal como se hace en este ejemplo.

5.10.4 Variables

Mix con señales: deben ajustar los tipos

```
process (A, B, C)
        variable M, N : integer ;
begin
        M := A ;
        N := B ;
        Z <= M + N ;
        M := C ;
        Y <= M + N ;
end process ;
```

de señal a variable

de variable a señal

Se puede asignar libremente tanto señales a variables, como variables a señales, tal como se ve en el ejemplo. Sin embargo los tipos en ambos lados deben coincidir.

5.11.0 Modelo de uso de variables

Uso de variables...

Veamos algunos casos donde podemos usar ventajosamente las variables al escribir código VHDL.

5.11.1 Modelo de uso de variables

Valores intermedios

```
if MOSTRAR_A = '1' then
        Tiempo_exhibición := Tiempo_alarma ;
else
        Tiempo_exhibición :=Tiempo_horario ;
end if ;
```

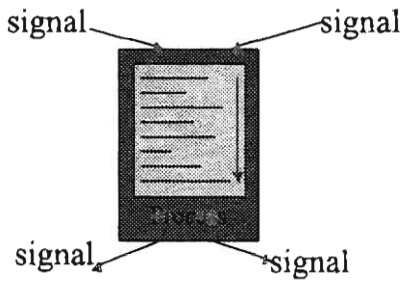
Valor intermedio actualizado

```
case Tiempo_exhibición is
        when 0 =>
        ...
```

Debido a que una variable se actualiza inmediatamente, es común usarlas para calcular valores 'intermedios' dentro de un proceso, los cuales posteriormente se usan en la ejecución de más código dentro del mismo proceso.

5.11.2 Modelo de uso de variables

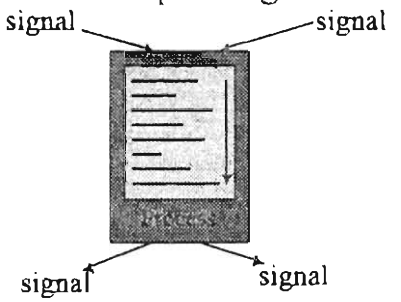
No pueden ser asignadas fuera de un proceso



Ya que el 'alcance' de una variable se circunscribe únicamente dentro del proceso donde fue declarada, es necesario asignar su valor final calculado a una señal, para poderlo comunicar fuera del proceso, si ello fuere menester.

5.11.3 Modelo de uso de variables

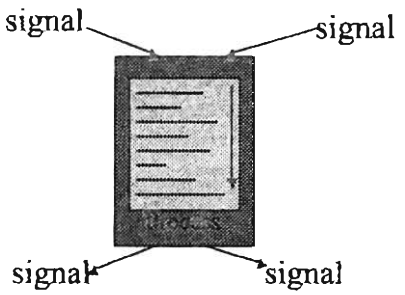
Variables para algoritmos



La tendencia es usar las variables cuando se tienen que realizar cálculos complejos o algoritmos. Los valores se introducen dentro del proceso mediante señales, mientras que el algoritmo efectúa cálculos dentro del mismo haciendo asignaciones a variables, y el resultado final se asigna a señales.

5.11.4 Modelo de uso de variables

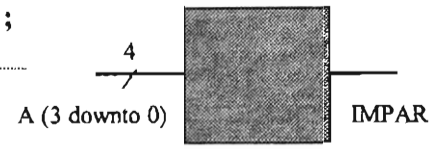
Ejemplo



Veamos un ejemplo de uso de variables, en un sistema empleado para calcular la paridad en una señal de datos de longitud arbitraria.

5.12.0 Ejemplo de un generador de paridad

```
process (A)
        variable TMP: std_logic ;
begin
        TMP := '0' ;
        for I in A'low to A'high loop
                TMP := TMP xor A(I) ;
        end loop ;
        IMPAR <= TMP ;
end process ;
```

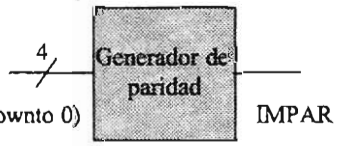


En el código para el generador de paridad, se declara una variable TMP. Al inicio del proceso, le asignamos el valor '0', y luego hacemos asignaciones a dicha variable TMP dentro del bucle.

5.12.1 Ejemplo del generador de paridad

Índice del bucle: atributos

```
process (A)
        variable TMP : std_logic ;
begin
        TMP := '0' ;
        for I in A'low to A'high loop
                TMP := TMP xor A(I) ;
        end loop ;
        IMPAR <= TMP ;
        end process ;
```



Nótese el rango del for-loop: 'A'low' es el índice más bajo del arreglo A (0 en este caso), y 'A'high' es el índice más alto (3 para el ejemplo). Se repetirá el ciclo una vez para cada uno de los elementos de A, y obsérvese que el código esta escrito de modo que resulta independiente del tamaño de A. A'low y A'high en VHDL son atributos de A.

5.12.2 Ejemplo del generador de paridad

Expandir el *for loop*

```
TMP := '0' ;
for I in A'low to A'high loop
        TMP := TMP xor A(I) ;
end loop ;
```

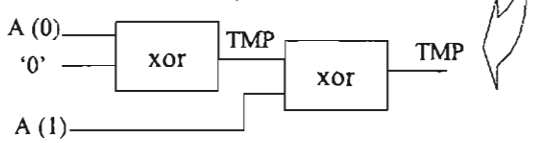
```
TMP := '0' ;
TMP := TMP xor A(0) ;
TMP := TMP xor A(1) ;
TMP := TMP xor A(2) ;
TMP := TMP xor A(3) ;
IMPAR <= TMP ;
```

Para comprender como trabaja este código, expandamos el for-loop sustituyendo los valores para I. El resultado es la secuencia de asignaciones de la variable TMP mostrado en el esquema. Ahora mentalmente tracemos la interconexión resultante de los operadores xor descritos,.....

5.12.3 Ejemplo del generador de paridad

Pensar acerca de las operaciones xor

```
TMP := '0' ;
TMP := TMP xor A(0) ;
TMP := TMP xor A(1) ;
TMP := TMP xor A(2) ;
TMP := TMP xor A(3) ;
IMPAR <= TMP ;
```

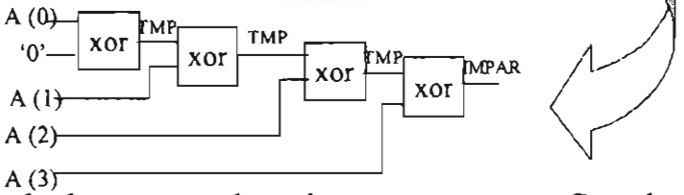


El primer operador xor tiene como entradas A(0) y TMP cuyo valor inicial es '0'. El segundo operador xor tiene como entradas A(1) y TMP, siendo el valor actual de TMP el calculado de la línea anterior. Desde el punto de vista del hardware, ese valor es la salida de la función xor previa,....

5.12.4 Ejemplo del generador de paridad

Trace la función

```
TMP := '0' ;
for I in A'low to A'high loop
        TMP := TMP xor A(I) ;
end loop ;
IMPAR <= TMP ;
```



El resultado es una cadena de compuertas xor. Cuando el código se sintetice, no será necesario reproducir exactamente esas compuertas xor. Sin embargo esta estructura es la arquitectura inicial que damos a la herramienta de síntesis para optimización, influyendo de ese modo en la implementación resultante.

5.13.0 Variables vs señales

variables vs señales

Veamos el diferente comportamiento que muestran las variables y las señales ....

5.13.1 Variables vs señales

Las variables se asignan inmediatamente

```
        signal  A, B, C, Y, Z : integer ;
begin
        process (A, B, C)
                variable M, N : integer ;
        begin
                M := A ;
                N := B ;
                Z <= M + N ;
                M := C ;
                Y <= M + N ;
        end process ;
```

A +B

A una variable se le asigna su valor inmediatamente. En el ejemplo, cuando se ejecuta la línea Z<=M+N, el valor de M+N es el valor de A+B.

5.13.2 Variables vs señales

La señal actualiza su valor posteriormente

```
        signal  A, B, C, Y, Z : integer ;
begin
        process (A, B, C)
                variable M, N : integer ;
        begin
                M := A ;
                N := B ;
                Z <= M + N ;
                M := C ;
                Y <= M + N ;
        end process ;
```

Z se actualiza con A+B

El valor de la señal Z será actualizado cuando el proceso se suspenda, aunque es calculado y almacenado cuando se ejecuta la línea correspondiente. Entonces, en este ejemplo, Z será actualizado con el valor de A+B hasta el *end process*.

5.13.3 Variables vs señales

Lo mismo para Y

```
        signal  A, B, C, Y, Z : integer ;
begin
        process (A, B, C)
                variable M, N : integer ;
        begin
                M := A ;
                N := B ;
                Z <= M + N ;
                M := C ;
                Y <= M + N ;
        end process ;
```

Y se actualiza con C + B

Cuando se alcanza la asignación a Y, M que es variable, se ha actualizado con el valor de C, y de ese modo cuando el proceso se suspenda, a la señal Y le será asignado el resultado de C+B.

5.13.4 Variables vs señales

Considerese lo siguiente para señales....

```
        signal  A, B, C, Y, Z : integer ;
        signal  M, N : integer ;
begin
        process (A, B, C, M, N)
        begin
                M <= A ;
                N <= B ;
                Z <= M + N ;
                M <= C ;
                Y <= M + N ;
        end process ;
```

¿Cuánto valen Y & Z ?

Consideremos el mismo ejemplo, excepto que M y N son señales en lugar de variables. ¿Cómo se ejecuta este proceso si existe un evento en A, B, o C, y ....cuánto valen Y y Z?...¡lo mismo: C+B!.

6. Tópicos sobre la síntesis

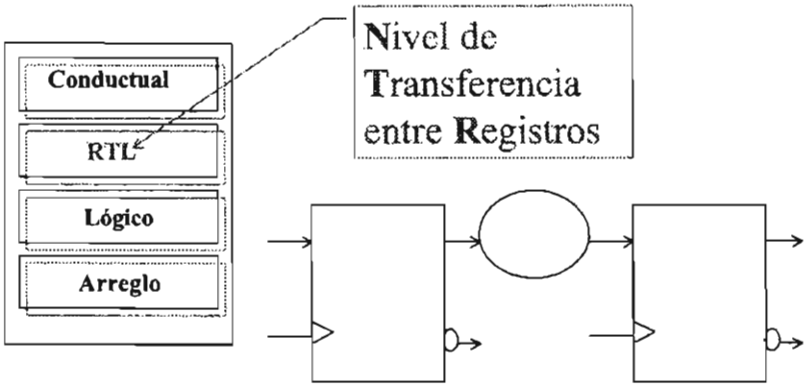
6.1.0 ¿Que se aprenderá en la sexta presentación?

Tópicos sobre síntesis

Hay tres temas principales que se cubren en esta parte: veremos un panorama general sobre inferencia de registros, aritmética vectorial y síntesis de operadores aritméticos.

6.1.1 ¿Qué aprenderemos?

i) ¿Qué es el estilo RTL?

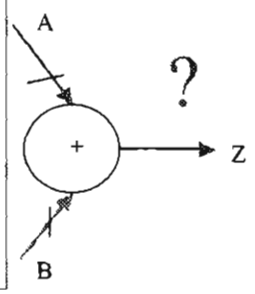


Antes que nada veremos la forma de describir registros en el lenguaje VHDL, y en base a eso definiremos lo que es en VHDL el estilo de descripción al nivel de transferencia entre registros o RTL.

6.1.2 ¿De que trata la lección?

ii) Aritmética vectorial

```
entity Sum is
port (A, B : in bit_vector (0 to 7) ;
   Z : out bit_vector (0 to 7) ;
end Sum ;
architecture ARITMETICA of Sum is
begin
   Z <= A + B ;
end ARITMETICA ;
```



En el lenguaje VHDL tal como está definido por defecto **no** es posible efectuar aritmética con vectores. Sin embargo ello es necesario en diseño digital. Hay forma de darle la vuelta al problema, y eso veremos más adelante.

6.1.3 ...y que más ?

iii) Síntesis de operadores aritméticos

```
Z <= A + B ;
```

⇩

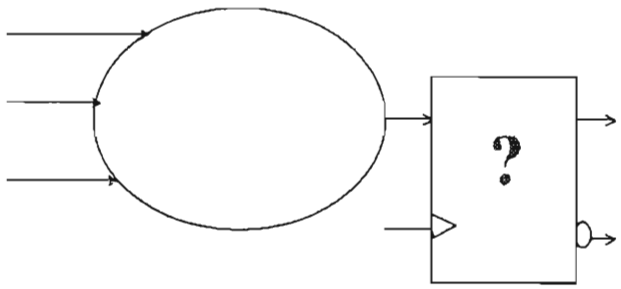
Síntesis

⇩

?

Se verá como en VHDL los operadores aritméticos pueden sintetizarse de diferente forma, y entonces se considerará el efecto de esto en la forma de escribir código para síntesis.

6.2.0 Especificando registros en VHDL



Veamos en principio como describir registros en VHDL...

Proceso síncrono: un proceso disparado por reloj

```
entity FLIPFLOP is
    port (D, CLK : in std_logic ;
          Q : out  std_logic ) ;
end FLIPFLOP ;
architecture A of FLIPFLOP is
begin
    process
    begin
          wait until (CLK'event and CLK='1') ;
          Q <= D ;
    end process ;
end A ;
```

El estilo básico para describir registros es declarando un proceso el cuál se ejecuta únicamente cuando existe un flanco de subida o bajada de la señal de reloj. Aquí Q solo se actualiza con el valor D en el flanco de subida del reloj, y no será modificado en ningún otro instante de tiempo, aunque D cambie. Ello describe la función de un registro en hardware.

procesos síncronos: procesos sensibles al reloj

```
process
begin
  wait until (CLK'event and CLK='1') ;
  Q <= D ;
end process ;
```

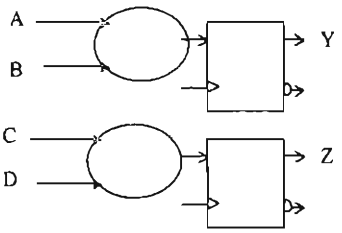
Proceso síncrono o proceso disparado por reloj

En este tipo de procesos, el primer postulado se usa para detectar el flanco de subida del reloj. Se les conoce como 'procesos disparados por el flanco de subida del reloj'. Las líneas de código dentro del proceso por ende, solo serán ejecutadas cuando suceda un flanco de subida en el reloj.

6.2.3 Especificando registros en VHDL

Registros a todas las señales asignadas ...

```
process
begin
  wait until
  Z <= A + B ;
  Y <= C+ D ;
end process ;
```

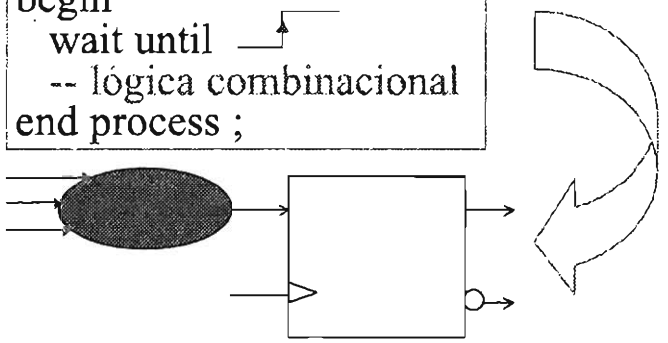


Para las señales cuya asignación se efectúa dentro del proceso, sus nuevos valores únicamente se asignan si sucede un flanco de subida en el reloj. Entonces, **TODAS** las señales que se asignan dentro de ese proceso serán consideradas por el sintetizador como salidas de registros.

6.2.4 Especificando registros en VHDL

Lógica combinacional con salidas a través de registros

```
process
begin
  wait until
  -- lógica combinacional
end process ;
```

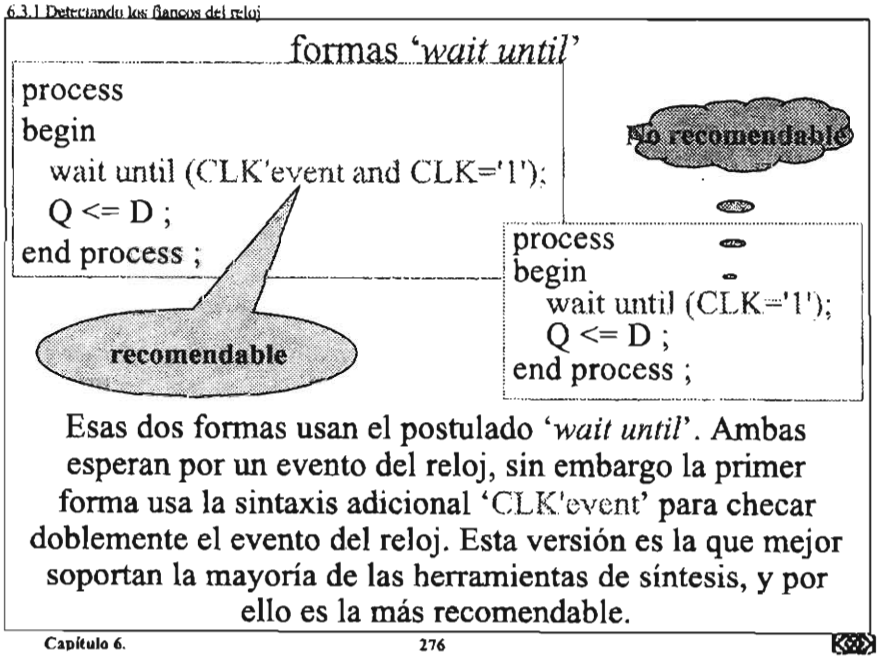


Asi entonces, podemos observar que un proceso síncrono define un bloque de lógica combinacional con sus salidas conectadas a un registro.

6.3.0 Detectando el flanco del reloj

```
process
begin
  wait until  /\
  Z <= A + B ;
  Y <= C + D ;
end process ;
```

En VHDL existen muchas formas para detectar un flanco de subida o bajada de la señal de reloj, y siendo que algunos métodos son mejor soportados por la mayoría de las herramientas de síntesis, mientras que otros no, y analicemos la cuestión de cuál seleccionar cuando se les necesite...



6.3.1 Detectando los flancos del reloj

formas '*wait until*'

```
process
begin
  wait until (CLK'event and CLK='1');
  Q <= D ;
end process ;
```

recomendable

No recomendable

```
process
begin
  wait until (CLK='1');
  Q <= D ;
end process ;
```

Esas dos formas usan el postulado '*wait until*'. Ambas esperan por un evento del reloj, sin embargo la primer forma usa la sintaxis adicional 'CLK'event' para checar doblemente el evento del reloj. Esta versión es la que mejor soportan la mayoría de las herramientas de síntesis, y por ello es la más recomendable.

Formas con lista de sensitividad

```
process (CLK)                    χ
begin
   if (CLK = '1' ) then
        Q <= D ;
   end if ;
end process ;
```

```
process (CLK)                    √
begin
   if (CLK'event and CLK = '1') then
        Q <= D ;
   end if ;
end process ;
```

Estas dos formas usan la lista de sensitividad para detectar un evento en el reloj, seguido de un postulado '*if*' para checar el sentido del evento. Como las herramientas de síntesis ignoran la funcionalidad implicada por un proceso mediante la lista de sensitividad, la mayoría de las herramientas solo soportan la forma con la cláusula '***CLK'event***'.

Forma mediante una llamada a función

```
library IEEE ;
use IEEE.std_logic_1164.all ;
...
process
begin
   wait until RISING_EDGE (CLK) ;
   Q <= D ;
end process ;
```

El paquete **std_logic_1164** de la IEEE, incluye una llamada a función RISING_EDGE o FALLING_EDGE la cual checa por un flanco de subida o bajada de la señal de reloj: la sintaxis es mostrada aquí. No todas las herramientas de síntesis son capaces de detectar los flancos del reloj usando este método.

6.3.4 Detectando flancos del reloj.

Las formas '*if*' y '*wait*' son las comunmente usadas

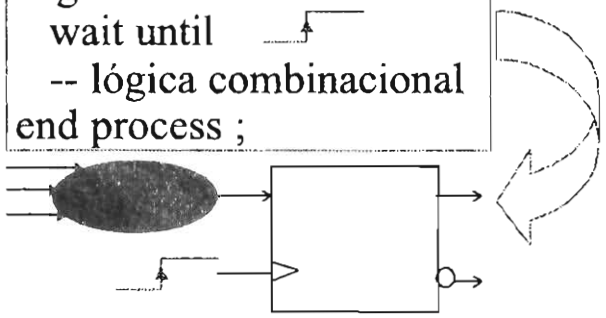
```
process
begin
   wait until (CLK'event and CLK='1') ;
   Q <= D ;
end process ;
```

```
process (CLK)
begin
   if (CLK'event and CLK ='1' ) then
   Q <= D ;
   end if ;
end process ;
```

Los estilos '*if*' y '*wait until*', con la sintaxis '*CLK'event*' son los más comunmente aceptados por los síntetizadores. Se recomienda usar estas formas de detección de flancos cuando se describen registros sin ninguna señal de reset.

6.4.0 Reglas de procesos sincrunos, o disparados por flanco

```
process
begin
   wait until
   -- lógica combinacional
end process ;
```



Veamos algunas reglas adicionales que deben observarse al describir 'procesos disparados por flanco' o 'síncronos'en VHDL...

6.4.1 Reglas de los procesos disparados por flancos

wait es el primer postulado del *process*

```
process
begin
   wait until (CLK'event and CLK='1');
   --lógica combinacional
end process ;
```

primer y único postulado "wait .."

solo lógica combinacional conectada a registros

La regla general acerca del uso del postulado *wait* para detectar los flancos del reloj, es que debe ser el primer y único postulado *wait* dentro del proceso. Todos los otros postulados describen lógica combinacional con salidas conectadas a registros.

6.4.2 Reglas de procesos disparados por flancos de reloj

Forma '*if*': toda la lógica dentro del postulado *if*

```
process (CLK)
begin
   if (CLK'event and CLK = '1') then
      Q <= D ;
   end if ;
end process ;
```

primer postulado

todos los postulados dentro del "if"

La regla para los procesos síncronos considerando la forma '*if*', dice que no deberá haber otros postulados fuera de la estructura *if*, y que la estructura no deberá tener ninguna otra cláusula *else*. Todo el código dentro de los postulados *if* describen lógica combinacional conectada a registros.

6.4.3 Regla de procesos síncronos

Registros 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 ;
```

Para especificar registros con reset asíncrono será necesario usar la estructura de código mostrada aquí. En este caso, un evento en RST(='1') causa que el proceso se ejecute, y Q sea puesta a cero. De otro modo, únicamente un evento sobre el reloj será la causa de que Q se actualice con D.

6.4.4 Regla de procesos síncronos.

Reglas para el 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 ;
```

primero y único

último "else"

La regla para el formato de los procesos síncronos con 'reset asíncrono', es similar a la forma '*if*' sin reset: ningún otro postulado fuera de la estructura *if*, y no debe de haber ninguna otra cláusula *else* después de la detección del flanco.

Inferencia de latches

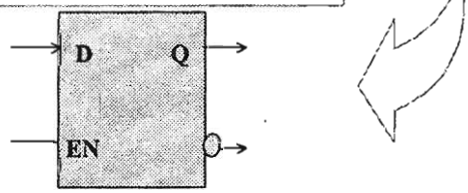
```
process (EN, D )
begin
        if (EN = '1') then
                Q <= D ;
        end if ;
end process ;
```

¿qué implica el código?...

Veamos las implicaciones que trae el escribir código VHDL en la forma descrita arriba.

latches transparentes

```
process (EN, D)
begin
   if (EN = '1' ) then
        Q <= D ;
   end if ;
end process ;
```



El proceso mostrado en este ejemplo es igual al presentado en la lámina anterior, y será sintetizado no como una AND sino como un ***latch* transparente**.

Las asignaciones incompletas provocan latches transparentes

```
process ( A, B )
begin
   if CONDICION then
        X <= A ;
        Y <= B ;
   else
        X <= '0' ;
   end if ;
end process ;
```

asignación incompleta

La regla en las tres últimas descripciones es que algunas señales se asignan dentro de algunas ramas del código, pero no en otras. Esto se conoce como 'asignacion incompleta'. Si esto sucede dentro de un proceso COMBINACIONAL, entonces al sintetizarlo se construirán en hardware *latches* transparente no deseados.

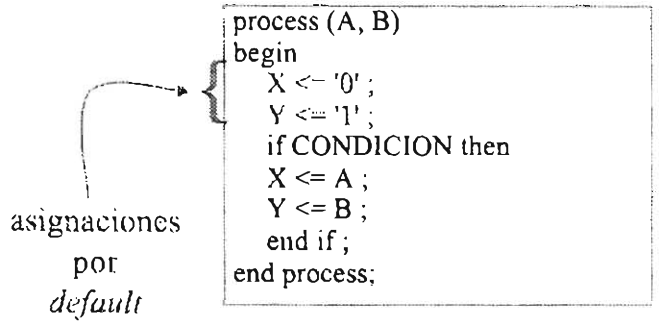
¿Cómo suprimir *latches*?

```
process (A, B)
begin
   if CONDICION then
        X <= A ;
        Y <= B ;
   else
        X <= '0' ;
   end if ;
end process ;
```

fácilmente se olvido la asignación Y <= '1' ;

Es muy fácil que al escribir código se cometan accidentalmente asignaciones incompletas, y por lo tanto **sin querer** se construyan *latches*. En diseños reales, el tratar de asegurar que en todas y cada una de las ramas del código se hagan todas las asignaciones para cada una de las señales que intervienen, es por lo común demasiado trabajo y sumamente tedioso, por lo que ..

Usar asignaciones por *default*



... el método mas confiable consiste en asignar valores a todas las señales por default justo al inicio del proceso combinacional. Esas asignaciones serán sobreescritas por las posteriores asignaciones hechas dentro del mismo proceso, pero se asegura asi que ningún *latch* transparente sea inferido al sintetizar.

¿RTL?

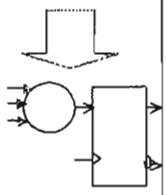
Definamos el significado de escribir código VHDL al estilo de nivel de transferencia de registros o RTL ...

6.6.1 Definición de RTL

RTL son procesos combinacionales y procesos síncronos

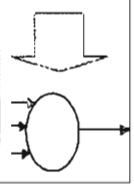
síncrono

```
process
begin
   wait
   --lógica
end process ;
```



combinacional

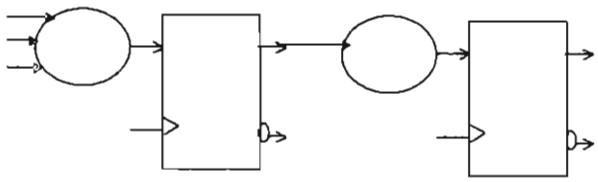
```
process (     )
begin
        --lógica
end process ;
```



En realidad la definición de estilo RTL es muy sencilla: todo el código VHDL que se escriba debe ser particionado entre **procesos combinacionales** y **procesos síncronos**. Entonces, la forma en la cuál se particiona la función del diseño, define donde colocar cada uno de los registros y la lógica combinacional entre ellos.

6.6.2 Definición de RTL

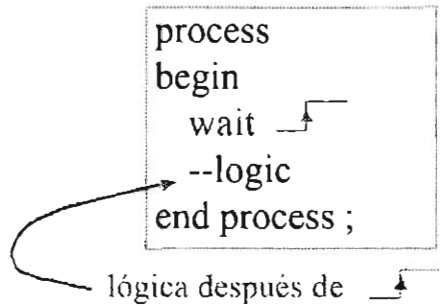
Es responsabilidad del diseñador controlar los registros



De la definición anterior se puede inferir que en este estilo debemos tener control de que y cuanta lógica colocar entre los registros. Vgr. si la herramienta de síntesis no puede alcanzar la velocidad del reloj requerida, entonces es nuestra responsabilidad reescribir el código VHDL para tratar de colocar menos lógica entre los registros...

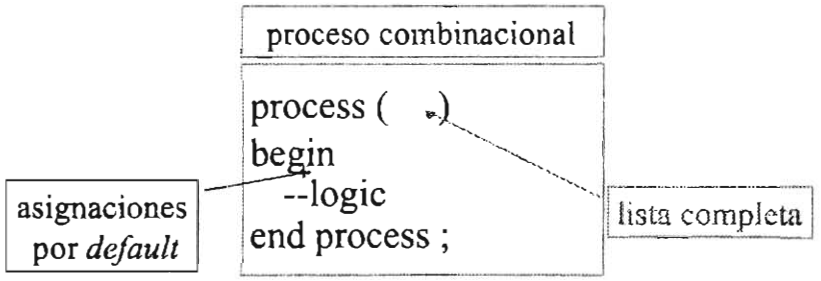
Reglas de procesos síncronos

Procesos síncronos



Resumamos las reglas para los procesos síncronos: toda lógica combinacional definida dentro del proceso debe estar después de la detección del flanco de subida del reloj.

Reglas de los procesos combinacionales



..y para los procesos combinacionales: el proceso debe de tener la lista de sensitividad completa, y de ser posible una asignación de señales por *default* para evitar que se interconstruyan *latches* transparentes no deseados al momento de la síntesis.

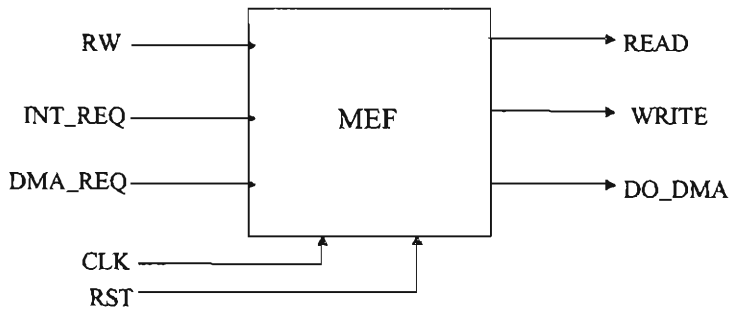
6.7.0 Máquinas de Estado Finitas

Máquinas de Estado Finitas (FSM)...

Ahora veremos como se describe mediante el código VHDL las Máquinas de Estado Finitas (FSM) ...

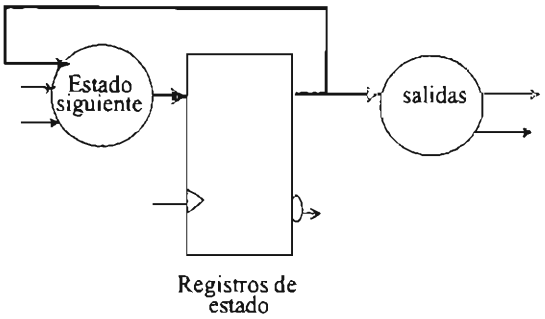
6.7.1 Máquinas de Estado Finitas

Un ejemplo de FSM: controlador de una CPU



Un ejemplo de una FSM es un controlador de un sistema basado en una CPU... Veremos el estilo general de descripción VHDL sintetizable para las Máquinas de Estado Finitas, más que estudiar la funcionalidad específica de este ejemplo...

Arquitectura de las FSM



La arquitectura general de una FSM consiste de un bloque de lógica combinacional para determinar el estado siguiente, registros de estado, y lógica combinacional para la salidas, tal como se muestra en el diagrama.

Tipos enumerados para el vector de estados

```
architecture RTL of MEF is
  type EDOS_TIPO is (ESPERA, RW_CICLO,
                     INT_CICLO, DMA_CICLO ) ;
....
```

En este ejemplo se supone que la FSM tiene cuatro estados, los cuales se describen usando un tipo de dato definido por el usuario. En el caso presente se ha definido un tipo llamado EDOS_TIPO, el cuál se usa para representar los cuatro estados de la máquina en cuestión, llamados a saber: ESPERA, RW_CICLO, INT_CICLO, y DMA_CICLO.

Declaración del vector de estados

```
architecture RTL of MEF is
   type EDOS_TIPO  is (ESPERA, RW_CICLO,
                       INT_CICLO, DMA_CICLO) ;
   signal EDO_SIG, EDO_PRSTE : EDOS_TIPO ;
begin
....
```

Ahora declaramos las señales EDO_PRSTE y EDO_SIG como las entradas y las salidas del vector de estados de la máquina en estudio: como esas señales definen el estado de nuestra FSM, entonces se les declara del tipo EDOS_TIPO.

Lógica del estado siguiente

```
Combinacional: process (EDO_PRSTE, RW, INT_REQ, DMA_REQ)
begin
        case EDO_PRSTE is
                when ESPERA =>
                        --postulados
                when RW_CICLO =>
                        --postulados
                when INT_CICLO =>
                        --postulados
                when DMA_CICLO =>
                        --postulados
        end case ;
end process Combinacional ;
```

La 'lógica para el estado siguiente' se define dentro de un proceso combinacional, el cual contiene el postulado *case* con una rama para cada uno de los estados de la FSM. El proceso es sensitivo a las entradas de función, y a la señal EDO_PRSTE.

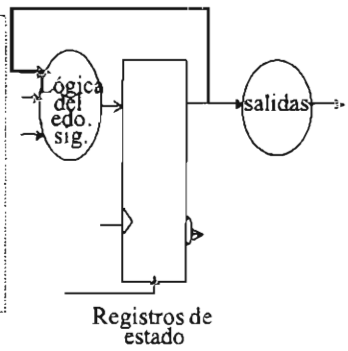
La lógica del estado siguiente

```
Combinacional : process (EDO_PRE, RW_INT, INT_REQ, DMA_REQ)
begin
   EDO_SIG <= EDO_PRSTE ;
   case EDO_PRSTE is
        when ESPERA =>
                if (INT_REQ = '1' ) then
                        EDO_SIG <= INT_CICLO ;
                elsif (DMA_REQ = '1' ) then
                        EDO_SIG <= DMA_CICLO ;
                end if ;
        when RW_CICLO =>
                --postulados
        ...
end process Combinacional ;
```

Dentro de cada rama de los postulados *case* tenemos código mediante el cual se asigna a la señal EDO_SIG la definición de las transiciones de estado. Así entonces, tendremos una transición desde el estado ESPERA si INT_REQ, o DMA_REQ valen '1'.

Registros

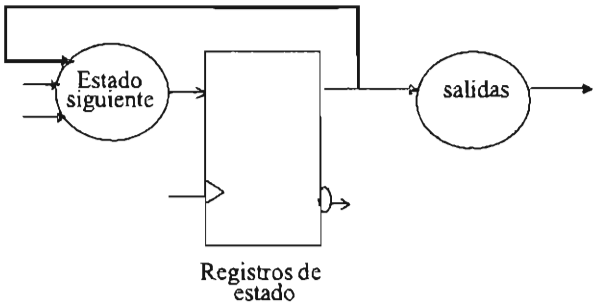
```
REG : process (CLK,RST)
begin
   if RST = '1' then
        EDO_PRSTE <= ESPERA ;
   elsif CLK'event and CLK = '1' then
        EDO_PRSTE <= EDO_SIG ;
   end if ;
end process REG ;
```



Un vector de estados se define dentro de un proceso síncrono separado, el cuál define solo registros, y nada de lógica combinacional. Aquí asignamos el valor de EDO_SIG a EDO_PRSTE solo cuando existe un flanco ascendente del reloj.

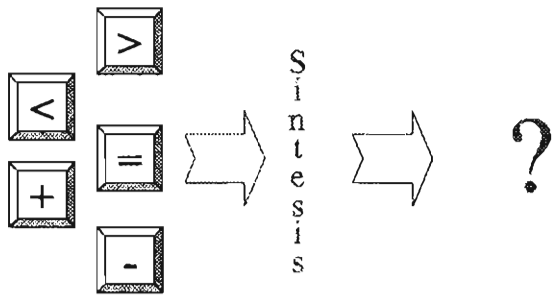
6.8.4 Código de máquinas de estado

Lógica de las salidas



La 'lógica de las salidas' se puede definir ya sea dentro del mismo proceso de la 'lógica del estado siguiente', o dentro de un nuevo proceso combinacional. Si se desea describir una máquina Moore, en donde las salidas sólo son funciones del vector de estados, entonces es recomendable hacerlo en un proceso separado.

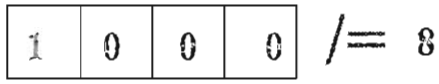
6.9.0 Operaciones sobre vectores



Ahora que se ha definido lo que significa escribir código a nivel de transferencia de registros, estamos listos para estudiar como son sintetizados los operadores relacionales y aritméticos. Veamos primero algunos problemas relacionados al aplicar dichos operadores sobre vectores...

6.9.1 Operaciones sobre vectores

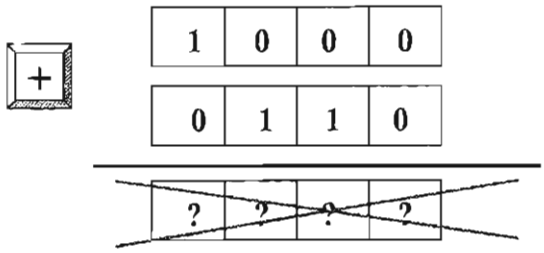
los vectores no representan números



Tal como se había dicho en las láminas de 'operadores VHDL', un vector (tal como un *std_logic_vector*) **no** representa ningún número. VHDL considera que un vector es simplemente una colección de valores de un solo 'bit', los cuales se han asociado juntos por conveniencia.

6.9.2 Operaciones sobre vectores

operaciones limitadas sobre vectores



Como un resultado de lo anterior, en el lenguaje VHDL **por default** no es posible efectuar operaciones aritméticas entre arreglos, digamos como la suma o resta de vectores. Las operaciones relacionales, tal como la prueba para igualdad, solo dan resultados satisfactorios si los vectores son de la misma longitud, caso contrario hay que estar preparados para las sorpresas.

¡pero existe una solución!

operadores sobrecargados (*overloading*)

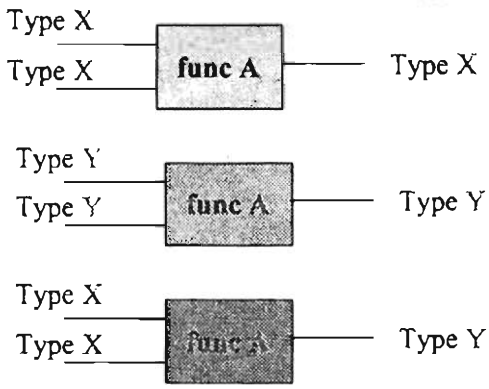
Existe una solución para esta cuestión, la cual es implementada vía una capacidad del VHDL conocida como 'sobrecarga de operadores' ('operador *overloading*'). Discutiremos más adelante en que consiste esta capacidad, y hasta entonces explicaremos como debe usarse para superar nuestras limitaciones,...

concepto de sobrecarga...

Primero veamos el concepto de sobrecarga, considerando algunos pocos ejemplos...

6.10.1 Sobrecarga

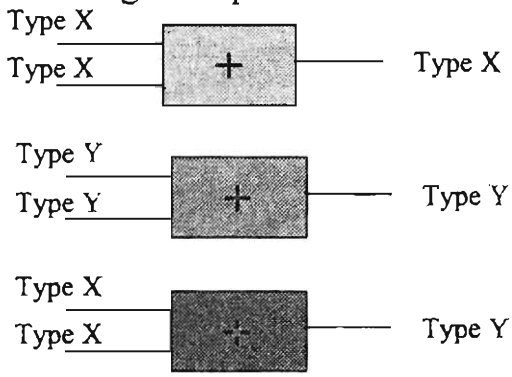
múltiples funciones con diferentes tipos de datos..



El concepto de sobrecarga: una propiedad del VHDL mediante la cual se permite que múltiples funciones sean definidas con el mismo nombre, pero diferentes tipos de datos en sus entradas y salidas.

6.10.2 Sobrecarga

Sobrecarga de operadores estándar

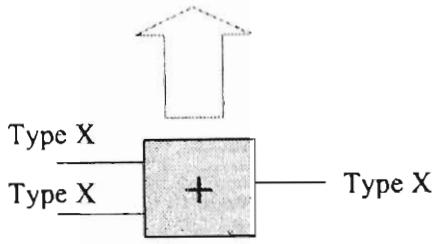


Este concepto se extiende a los operadores estándar: por ejemplo, es legal definir múltiples funciones llamadas "+" tan pronto como ellas tengan diferentes combinaciones de tipos de datos para sus entradas y sus salidas.

6.10.3 Sobrecarga

funciones llamadas según el contexto

```
signal A, B, Z  : X ;
Z <= A + B ;
```



El lenguaje define que un simulador VHDL deberá ser capaz de llamar automáticamente la función apropiada en el contexto en el cual es usada.

6.11.0 Ejercicios de sobrecarga

paquete de funciones

El ejemplo se basa en un caso del mundo real, en donde tenemos un paquete conteniendo varias funciones llamadas con el mismo nombre "+", las cuales deben estar **sobrecargadas** para así poder efectuar aritmética sobre vectores.

El paquete

```
package PAQ_ARITMETICO is
subtype slv is std_logic_vector ;
function "+" (L: slv; R: slv) return integer ;      <-- 1
function "+" (L: slv; R: slv) return slv;           <-- 2
function "+" (L: slv; R: integer) return slv ;      <-- 3
function "+" (L: integer; R: slv) return slv ;      <-- 4
end PAQ_ARITMETICO ;
```

El ejemplo usa el paquete PAQ_ARITMETICO mostrado aquí. El paquete contiene cuatro funciones llamadas "+", cada una con una diferente combinacion de tipos de datos. También se ha definido 'slv' como un subtipo de *std_logic_vector*, lo cual efectivamente hace 'slv' un 'alias' para *std_logic_vector*.

Entidad

```
package PAQ_ARITMETICO is
subtype slv is std_logic_vector ;
function "+" (L:slv; R:slv) return integer ;
function "+" (L:slv; R:slv) return slv ;
function "+" (L:slv; R:integer return slv ;
function "+" (L:integer; R:slv)return slv ;
end PAQ_ARITMETICO ;
```

```
use work.PAQ_ARITMETICO.all ;

entity SOBRECARGADA is
port (    A_BUS: in slv (0 to 3) ;
          A_INT, B_INT: in integer ;
          Y_BUS, Z_BUS: out slv ( 0 to 3) ;
          Y_INT, Z_INT: out integer ) ;
end SOBRECARGADA ;
```

Los puertos de la entidad describen las señales que tratamos de sumar juntas. Cada una de las señales es, ya sea un vector slv (señal de nombre_BUS), o un *integer* (señal de nombre_INT).

6.11.3 Ejercicios de sobrecarga

Arquitectura

```
package PAQ_ARITMETICO is
subtype slv is std_logic_vector ;
function "+" (L: slv ;  R: slv) return integer ;
function "+" (L: slv ; R slv) return slv ;
function "+" (L: slv ; R: integer) return slv ;
function "+" (L: integer ; R: slv) return slv ;
end PAQ_ARITMETICO ;
```

```
architecture A of SOBRECARGADA] is
begin
        Y_INT <= A_INT + B_INT ;
        Z_BUS <= A_BUS + B_BUS ;
        Y_BUS <= A_BUS + A_INT ;
        Z_INT <= A_BUS + B_INT ;
end A ;
```

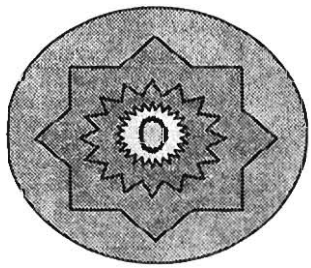
legal por default

legal por sobrecarga

legal por sobrecarga

ilegal

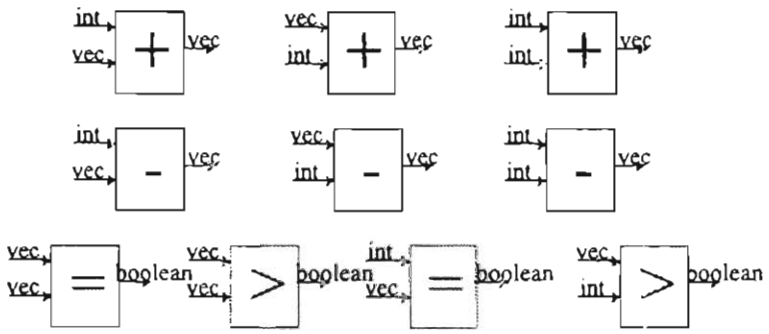
Dentro de la arquitectura se situan las líneas del código VHDL en las que se quiere implementar las sumas. En el ejemplo se explica si cada línea de código VHDL es legal por default en el lenguaje, legal debido a una función sobrecargada, o bien ilegal.



6.12.0 Paquetes de aritmética vectorial

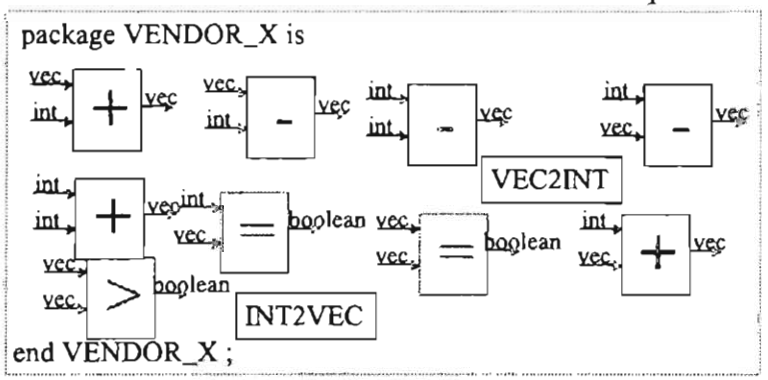
Debemos estar familiarizados con el concepto de sobrecarga u '*overloading*' para comprender la aritmética vectorial. Veamos como este principio es aplicado en las herramientas de síntesis comerciales reales ....

un paquete de funciones sobrecargadas...



Tal como debemos imaginar, lo que se requiere idealmente es un gran paquete VHDL con funciones 'sobrecargadas' para todos los principales operadores aritméticos y relacionales que incluya todas las combinaciones de enteros y vectores. La buena nueva es que no requerimos escribir dicho paquete por sí mismos,...

los distribuidores de sintetizadores nos los proveen

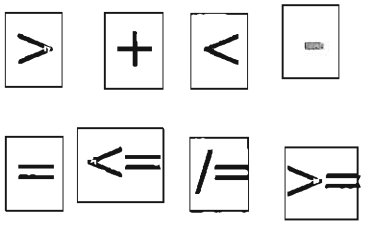


Todas las herramientas de síntesis VHDL proveen tales tipos de paquetes. Típicamente dichos packages contienen todos los operadores aritméticos y relacionales para combinaciones de vectores y enteros, junto con funciones útiles para la conversión entre integers y vectors.

6.12.3 Paquetes de aritmética vectorial

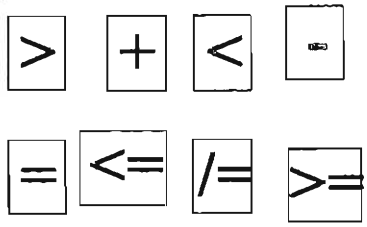
Se estilaba que cada proveedor surtiera diferentes...

package VENDOR_Y is



end VENDOR_Y ;

package VENDOR_X is



end VENDOR_X ;

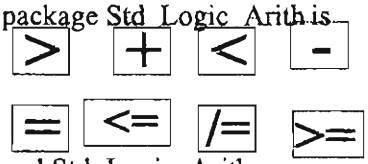
Al principio hubo desacuerdo, ya que cada proveedor entregaba packages diferentes. Ahora es usual escoger únicamente entre uno o dos paquetes usados típicamente en prácticamente todos los proyectos de diseño.

6.12.4 Paquetes de aritmética vectorial

paquetes estándar de Synopsys e IEEE.

package Synopsys:
estándar 'de facto'

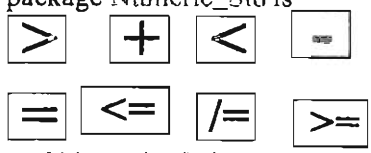
package Std_Logic_Arith is



end Std_Logic_Arith ;

package IEEE:
estándar 'oficial'

package Numeric_Std is



end Numeric_Std ;

El primer paquete debe su origen a Synopsys, un proveedor de herramientas de síntesis. Ellos dieron al dominio público el código fuente de su paquete '*Std_Logic_Arith*', eso lo convirtió en el estándar 'de-facto'. La IEEE ha definido otro estándar, el paquete 'Numeric_Std', el cual es el paquete 'oficial'.

6.13.0 Sistemas numéricos en los paquetes aritméticos estándar

Sistemas numéricos

Veamos brevemente como esos paquetes aritméticos nos permiten representar diferentes sistemas numéricos a usar en nuestros diseños...

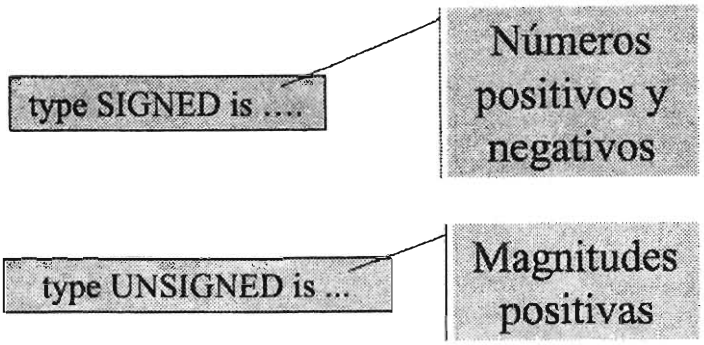
6.13.1 Sistemas numéricos en los paquetes aritméticos estándar

binarios y complemento a dos

En los sistemas digitales un número puede ser representado en al menos dos formatos: usando una representación 'binaria' llana, la cual permite representar tan solo números positivos o magnitudes, o mediante el uso de la representación 'complemento a dos', la cual permite la representación de números positivos y negativos.

6.13.2 Sistemas numéricos en los paquetes aritméticos estándar

tipos *signed y unsigned*



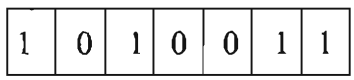
Para permitirnos el uso de ambos sistemas numéricos, los paquetes definen dos nuevos tipos de datos llamados *SIGNED* (para números en complemento a dos) y *UNSIGNED* (para números binarios sencillos).

6.13.3 Sistemas numéricos para paquetes aritméticos estándar

Arreglos de *std_logic*

type SIGNED is ...

type UNSIGNED is ...

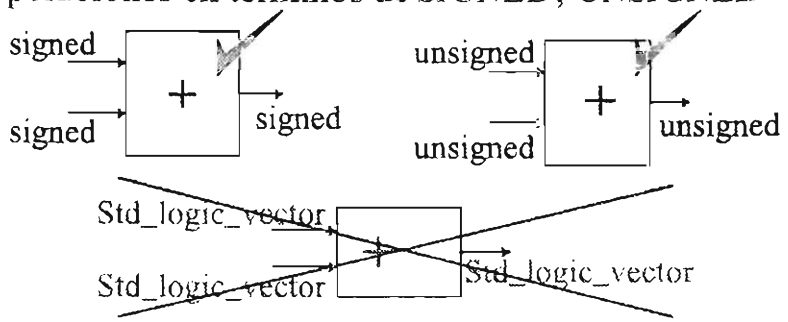


arreglo de *std_logic*

Los tipos de datos *SIGNED* y *UNSIGNED* se declara que sean **arreglos** de elementos del tipo ***std_logic***, de modo que ellos representan la misma clase de estructura de bus o vector como los tipo *std_logic_vector*.

6.13.4 Sistemas numéricos para paquetes de aritmética estándar

operaciones en términos de *SIGNED, UNSIGNED*



En cada uno de los paquetes, las operaciones aritméticas y relacionales se definen en términos de los tipos de datos *SIGNED* y *UNSIGNED* exclusivamente. Los packages no definen operaciones basadas en los tipos *std_logic_vector* para evitar confusiones de que la señal contiene una representación binaria o en complemento a dos.

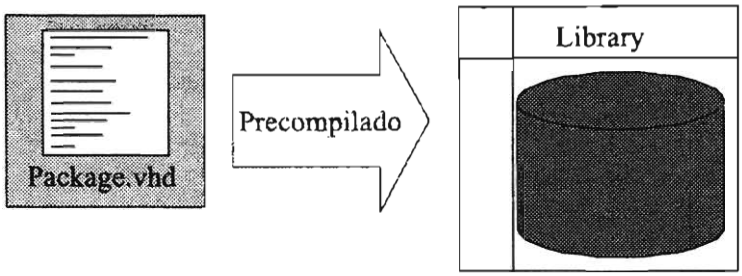
6.14.0 Referenciando los paquetes aritméticos

como hacer referencia a los packages...

Y para terminar esta discusión sobre paquetes aritméticos, veremos como debe hacerse referencia a cada uno de dichos packages, y aclarar la cuestión por resolver cual de ellos emplear.

6.14.1 Referenciando los paquetes aritméticos

Los paquetes deberán ser precompilados

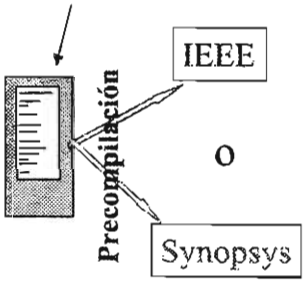


Es de esperarse que la mayoría de las herramientas de síntesis y simulación VHDL soportan bien ambos tipos de paquetes aritméticos que hemos venido discutiendo. Es norma que los paquetes actualmente ya están compilados dentro de una librería VHDL, y todo lo que se requiere para usarlos es hacer referencia a ellos en el código del diseño.

6.14.2 Referenciando los paquetes aritméticos

referenciando *Std_Logic_Arith*

Std_Logic_Arith.vhd



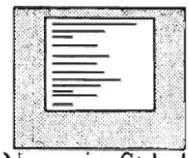

```
LIBRARY IEEE ;
USE IEEE.Std_Logic_1164.ALL ;
USE IEEE.Std_Logic_Arith.ALL ;
ENTITY.....
```

```
LIBRARY IEEE ;
USE IEEE.Std_Logic_1164.ALL ;
LIBRARY Synopsys ;
USE Synopsys.Std_Logic_Arith.ALL ;
ENTITY...
```

Debe encontrarse que el paquete *Std_Logic_Arith* ha sido precompilado, ya sea en una librería llamada 'IEEE', o en una librería de nombre 'Synopsys'. Aunque siempre hay que checar lo anterior en la documentación de la herramienta de síntesis.

6.14.3 Referenciando los paquetes aritméticos

referenciando *Numeric_Std*



Numeric_Std.vhd

Precompilación → IEEE

```
LIBRARY IEEE ;
USE IEEE.Std_Logic_1164.ALL ;
USE IEEE.Numeric_Std.ALL ;
ENTITY .....
```

El paquete *Numeric_Std* esta siempre disponible desde la librería 'IEEE', y es accesado usando las referencias mostradas aquí.

6.14.4 Referenciando los paquetes aritméticos

¿Cuál de ellos usar?

Std_Logic_Arith

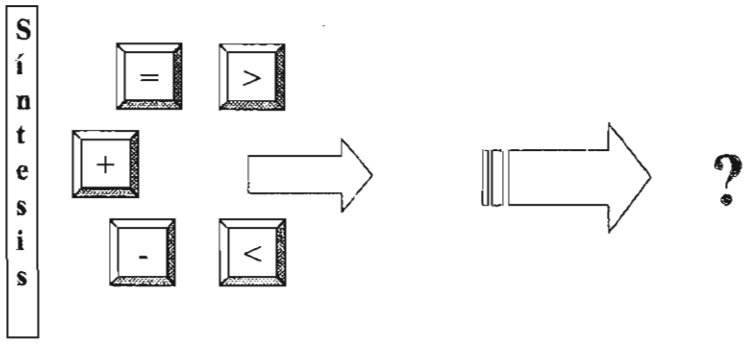
vs

Numeric_Std

- ¿soportado?
- ¿se ha usado en otra parte del diseño?

Para decidir cual de los paquetes usar, se requiere conocer si ambos paquetes están soportados por las herramientas de síntesis y simulación. Si lo anterior es cierto, se puede seleccionar el estándar de facto: *Std_Logic_Arith*, si es que ya se ha hecho uso de este paquete en otras partes del diseño; de otro modo, es mejor usar el paquete *Numeric_Std*.

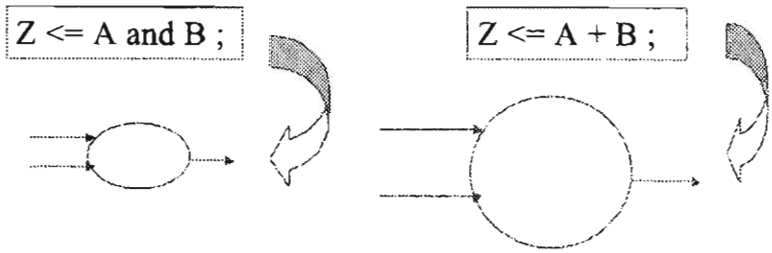
6.15.0 Síntesis de operadores



Finalmente en esta sección de síntesis, nos detendremos un momento más en considerar como los operadores aritméticos y relacionales pueden ser implementados de diferente manera por el sintetizador.

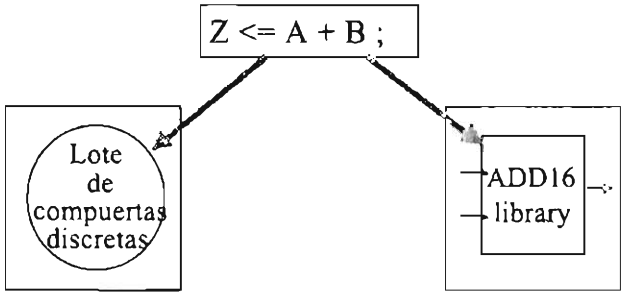
6.15.1 Síntesis de operadores

tamaño del código vs número de compuertas



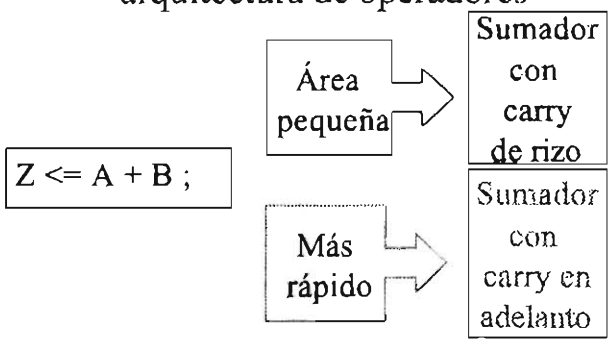
Tratándose de estos operadores, se encuentra que una pequeña cantidad de código puede fácilmente requerir una gran cantidad de área de Silicio, por lo que es importante entender el como puede sintetizarse eficientemente este tipo de operadores.

compuertas vs macroceldas



La primer cuestión es si el operador es construido desde compuertas discretas, o si una macrocelda optimizada es la seleccionada desde la librería de la tecnología que se este empleando. Algunas herramientas de síntesis pueden seleccionar macroceldas automáticamente, mientras que otras requieren que se haga una instanciación tecnológica específica de una celda dentro del código VHDL.

arquitectura de operadores



Con los operadores aritméticos, es importante conocer si la implementación va a ser optimizada para área o velocidad. Algunas herramientas de síntesis pueden tan sólo sintetizar una arquitectura, mientras otras permiten que se especifique cual de las arquitecturas será la empleada, o aún hacen la selección automáticamente.

6.15.4 Síntesis de operadores

compartición de operadores

```
if SEL = '1' then
   Z <= A + B ;
else
   Z <= A + C ;
end if ;
```

¿uno o dos sumadores?

En algunos casos las operaciones no necesitan realizarse al mismo tiempo, y un solo operador puede ser compartido. De nueva cuenta, algunas herramientas de síntesis son capaces de ejecutar lo anterior automáticamente, mientras que para otras es necesario escribir el código VHDL en un estilo diferente para poder inducir la compartición de operadores.

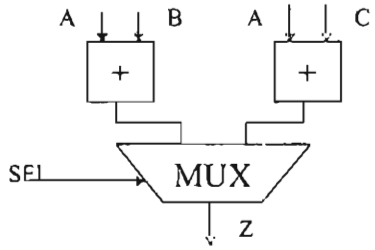
6.16.0 Compartición de recursos

compartición de recursos

El concepto de operaciones compartidas que no son ejecutadas al mismo tiempo se conoce como 'compartición de recursos'. Veamos un ejemplo de ello....

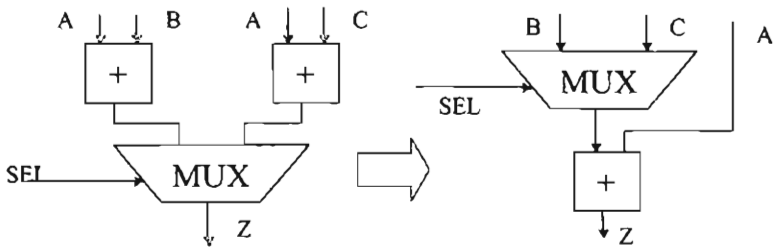
Ejemplo

```
if SEL = '1' then
   Z <= A + B ;
else
   Z <= A + C ;
end if ;
```



Analicemos el código del ejemplo. Representa literalmente dos sumadores, ya que dos operadores "+" han sido usados. La arquitectura que este código representa se muestra, y consiste de dos sumadores y un multiplexor.

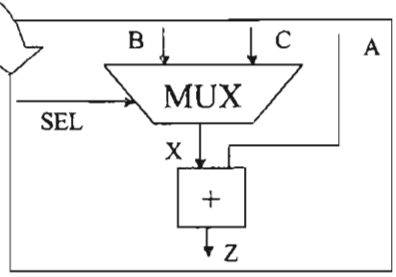
Compartiendo el sumador



Sin embargo, es posible representar esta función con un solo sumador. Podemos multiplexar B o C en un solo sumador, tal como se muestra en el diagrama. Algunas herramientas de síntesis pueden efectuar esta optimización automáticamente, mientras que otras no.

Compartición manual

```
if SEL = '1' then
   X :=B ;
else
   X := C ;
end if ;
Z <= A + X ;
```



En caso de que la herramienta de síntesis por sí misma no pueda efectuar la optimización, el código VHDL puede reescribirse para alcanzar el mismo resultado. El código es mostrado aquí, y puede verse que una variable intermedia ha tenido que definirse para representar la salida del MUX, la cual es entonces usada como entrada del unico sumador.

7. Panorama general sobre síntesis para dispositivos XILINX

7.1.1 Introducción

panorama general sobre síntesis VHDL para PLDs...

A lo largo de las siguientes tres presentaciones iremos revisando paso a paso algunas cuestiones relacionadas con el lenguaje VHDL y cuando en los diseños se emplean como blanco los dispositivos XILINX. En esta séptima trataremos sobre aspectos arquitectónicos de dichos chips, de las características generales de los sintetizadores a emplear y alguna terminología especializada exclusiva de los dispositivos programables.

7.2.0 ¿Qué es lo que cubriremos en este capítulo?

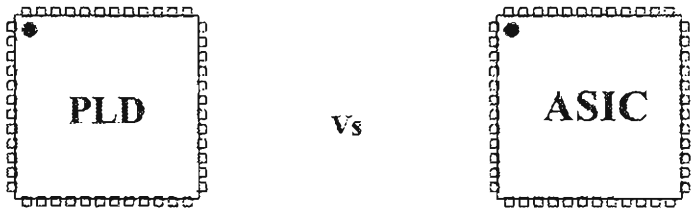
comencemos...

tópicos a tratar en esta presentación

En este séptimo capítulo trataremos de obtener una visión de conjunto sobre la síntesis VHDL enfocada hacia los dispositivos lógicos programables PLDs. Iniciemos planteando los tópicos que trataremos en el...

7.2.1 Lo que se verá en este capítulo

i) tecnologías PLD vs ASIC



Primero discutiremos brevemente las arquitecturas de los dispositivos a los que se pretende bajar los diseños VHDL, veremos en que difieren los PLDs de los ASICs y como esto afecta las metodologías de diseño VHDL para ambos tipos de chips.

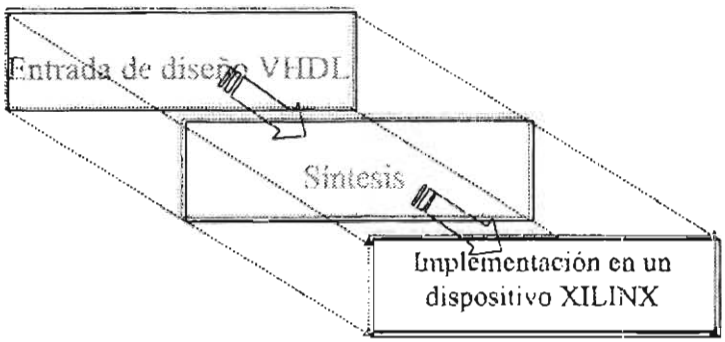
7.2.2 ¿Qué cubriremos en este capítulo?

ii) PLDs Xilinx

Entonces veremos las diferentes arquitecturas de los dispositivos Xilinx, esto es los FPGAs y los CPLDs, ya que estos tipos de chips son los principales dispositivos para los que se diseña con VHDL.

7.2.3 ¿Qué veremos en el presente capítulo?

iii) Arquitecturas Xilinx



Solo entonces es posible visualizar como las diferentes características arquitectónicas de los dispositivos programables impactan sobre el estilo de escribir código ...

7.2.4 Terminología

terminología propia de los dispositivos programables,...

Aclaremos la terminología básica propia de los dispositivos lógicos programables y aplicable para los chips de Xilinx, la cual usaremos a lo largo de lo que resta de las presentaciones

7.3.1 Terminología

terminología Xilinx

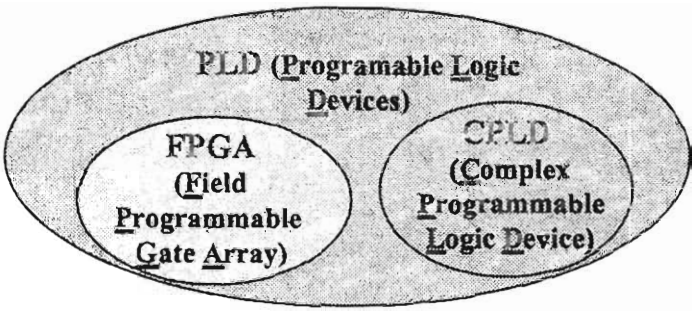
XILINX®

terminologías Xilinx...

A continuación se definirán algunos terminos tal como Xilinx los específica. El significado puede no coincidir con la terminología de lógica programable usada antiguamente o por otros fabricantes del mismo tipo de chips...

7.3.2 Terminología

PLD



El termino PLD ("*Programmable Logic Devices*") abarca actualmente tanto los **FPGA** ("*Field Programmable Gate Array*"), como los **CPLD** ("*Complex PLD*").

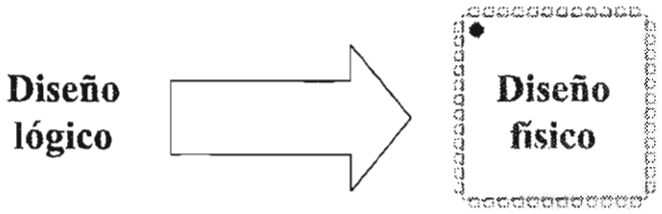
7.3.3 Terminología

diferencias tecnológicas entre FPGAs, y CPLDs

La tecnología subyacente en los **FPGA** es la **SRAM**, mientras que para los **CPLD** es la EPROM o la FLASH. Ejemplos de los primeros son los dispositivos Xilinx de la familia XC4010xl, mientras que de los segundos es la serie XC9000.

7.3.4 Terminología

¿Que quiere decir 'implementación'?



Otro termino más: la "**implementación**". Se refiere a la conversión de un diseño lógico en el diseño físico, i.e. el colocado y enrutado del diseño, y el bajado de bits al dispositivo seleccionado (***bitstream***). En el caso de los CPLD, este proceso también se le conoce como "***fitting***" ("ajuste").

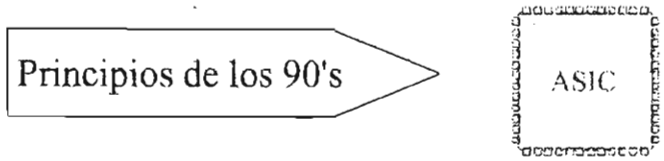
7.4.0 Síntesis de PLD

Síntesis de PLD's..

¿A qué dispositivos apuntar?. ¿Da lo mismo cualquiera de ellos? Empecemos viendo que es lo que se necesita considerar para sintetizar diseños en los PLD's...

7.4.1 Problemas de la síntesis a los PLD

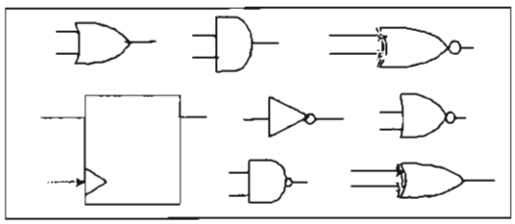
Los primeros usuarios de la Síntesis VHDL fueron los diseñadores de ASIC's



La herramienta de síntesis con HDLs históricamente ha sido usada principalmente por los diseñadores de circuitos integrados personalizados (ASICs) desde principios de los 90's. Se aplicaba desde entonces al diseño de grandes sistemas digitales. En la actualidad, dado que los PLD han crecido en capacidad y prestaciones, el HDL se ha mostrado especialmente adecuado para ser usado con los diseños que tienen como blanco este tipo de dispositivos.

7.4.2 Tópicos relativos a la síntesis para PLD's

ASIC son de grano fino



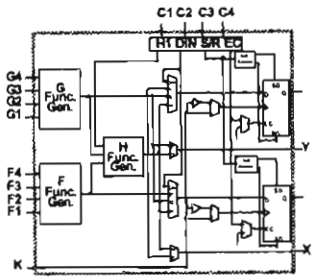
grano fino

Bloques de construcción ASIC

La tecnología ASIC se le clasifica como de '**grano fino**'. Esto simplemente quiere decir que sus bloques primitivos de construcción son muy pequeños, y que pueden ser colocados en cualquier combinación y en cualquier lugar en el dado del Silicio.

7.4.3 Cuestiones de síntesis de PLD's

Los PLD son de tecnología de grano grueso (coarse)



grano grueso

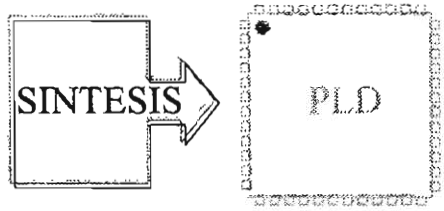
Bloque de construcción de un PLD

La mayoría de las tecnologías de PLDs son de '**grano grueso**', ya que los bloques elementales de construcción son mas grandes, y cuya arquitectura interna esta rígidamente definida.

7.4.4 Tópicos de síntesis PLD's

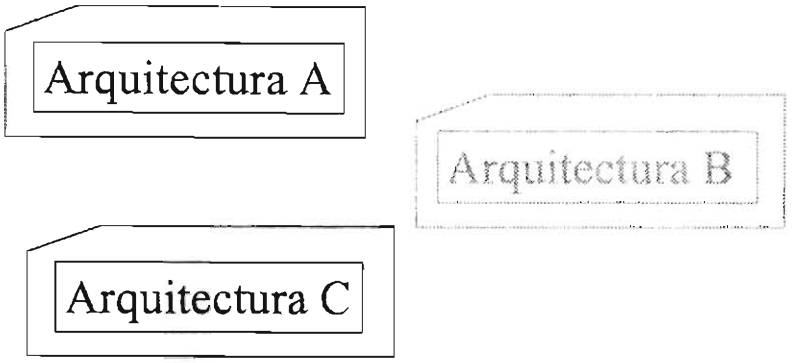
sintetizado a las tecnologías PLD

```
entity SUMADOR is
port (A, B : in BIT ;
        SUM, CARRY : out bit ) ;
.....
.....
.....
end SUMADOR ;
```



¿En qué le incumbe lo anterior a los diseñadores VHDL?. Desde la perspectiva del sintetizador, la herramienta necesita tener para el caso de los PLDs, algoritmos específicos para mapear la lógica dentro de 'bloques de construcción más grandes' en 'combinaciones eficientes', en lugar de simplemente agrupar pequeños bloques de construcción de grano fino como es propio de la tecnología ASIC.

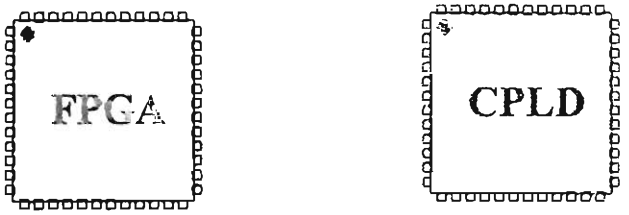
7.5.0 Arquitectura de los dispositivos XILINX



Ahora que ya conocemos en términos generales los diferentes PLDs, veamos un panorama general de las arquitecturas de los dispositivos XILINX, pues ello determinará el estilo de escritura del código…

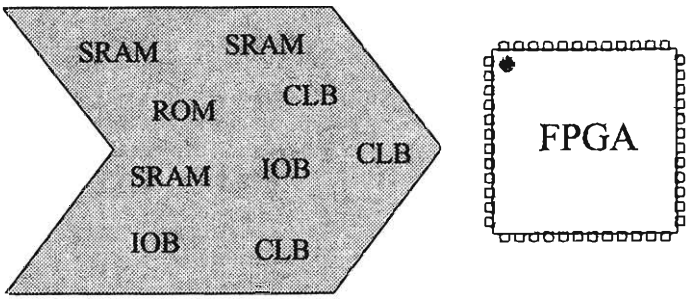
7.5.1 Arquitecturas de Xilinx

FPGA, CPLD.



Xilinx y otros fabricantes de chips poseen dos tipos diferentes de dispositivos de lógica programable: FPGAs y CPLDs. Veamos los principales bloques básicos de estas tecnologías, desde el punto de vista de lo que a la síntesis VHDL le concierne.

7.6.0 Tecnologías FPGA's

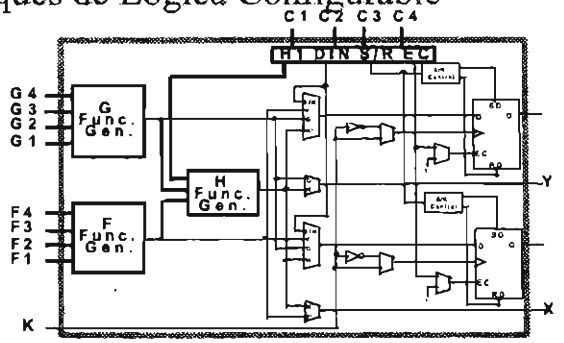


La tecnología Xilinx de los FPGAs esta basada en bloques básicos de construcción SRAM, bloques que configuran su lógica al momento de ser "bajados" al FPGA, esto cada vez que el dispositivo es energizado.

7.6.1 Tecnologías FPGA

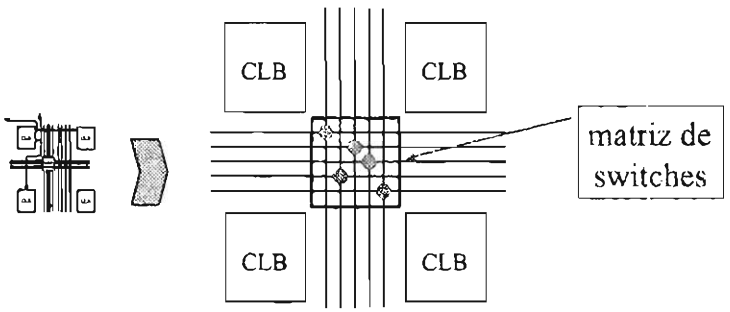
Bloques de Lógica Configurable

CLB de la serie XC4000



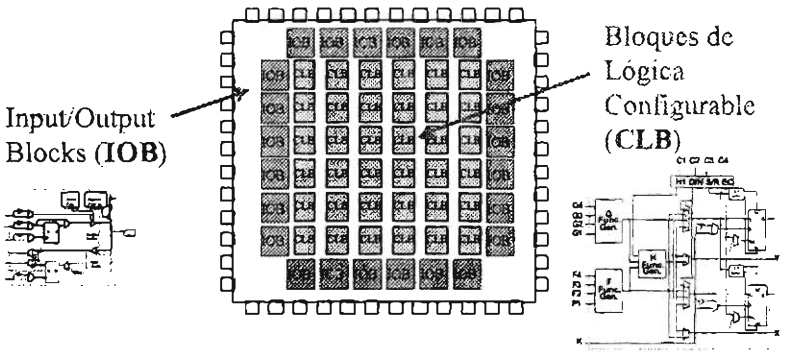
En los dispositivos **FPGA** de **Xilinx**, el bloque básico de construcción se le llama **CLB** (*Configurable Logic Block*), bloques de 'grano grueso'. Cada CLB tiene típicamente más de ocho entradas, tres o más salidas, lógica combinacional, y dos o más flip-flop y puede configurarse de muchos modos.

conexiones mediante switches programables



Los CLBs son colocados dentro del chip formando una matriz fija de CLBs; y en filas y columnas rodeándolos existen gran cantidad de pistas metálicas de conexiones, pudiéndose mediante switches programables, hacerse las conexiones entre CLBs, I/Os,...

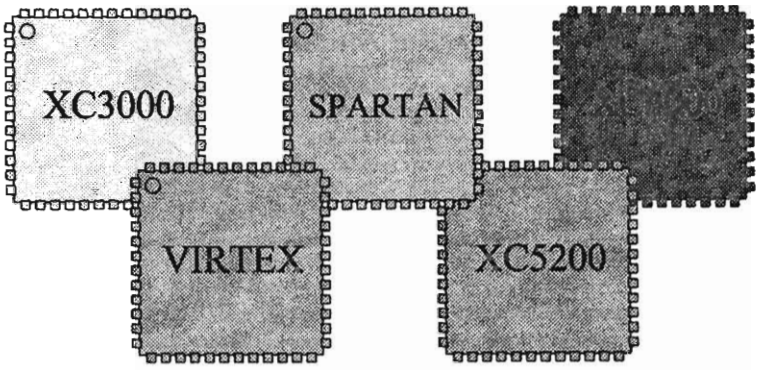
IOB proveen las conexiones de entrada-salida



La matriz de **CLB**s esta rodeada en la periferia por celdas de I/O llamadas **IOB**. La estructura de los IOBs es diferente de las de los CLBs, y puede ser configurada de diferentes formas para darnos gran cantidad de tipos de interfaz de I/O.

7.6.4 Tecnologías FPGA

Las principales tecnologías: XC4000, XC3000, XC 5200, Spartan, Virtex,..



Las principales tecnologías Xilinx basadas en SRAM son la XC3x00A, XC4000E/EX/XL/XV, XC5200, Spartan, Virtex.

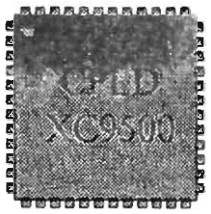
7.7.0 Tecnologías CPLD

Tecnologías

CPLD

Las tecnologías CPLD tienden a ser más útiles en el caso de aplicaciones pequeñas. Su comportamiento de tiempos esta bien caracterizado y por lo tanto es predecible. Ellos tienen una relación de 'registros a lógica' baja y entonces no son adecuadas para diseños donde se requieran gran cantidad de registros (vrg. pipelined) . Los FPGA son adecuados para grandes diseños y poseen más capacidades que los CPLD.

Familias CPLD de XILINX



La familia Xilinx de CPLD incluye las series XC9500 y la XC9000XL.

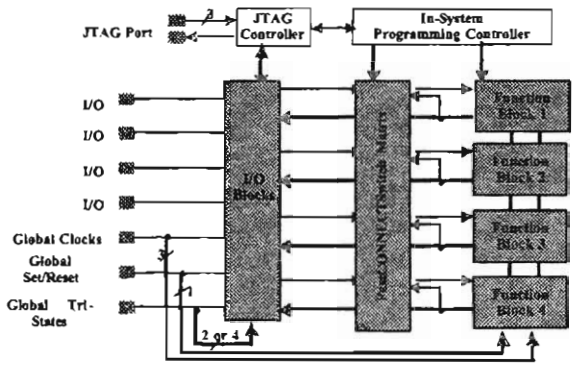
bloque de función

	FPGA	CPLD
Bloque de construcción	CLB (Bloque de Lógica Configurable)	FB (Bloque de Función)

Los CPLD de Xilinx se basan en un tipo diferente de bloque de construcción de los usados por los FPGA. En un CPLD, cada uno de los bloques de construcción se le conoce como Bloque de Función o FB (*Function Block*).

7.7.3 Tecnologías CPLD

FBs y macroceldas



Cada uno de los FBs provee capacidades lógicas programables con 36 entradas y 18 salidas. Cada uno de los FBs esta formado de macroceldas, cada una de ellas capaces de implementar tanto funciones combinacionales o con registros. Esos bloques de función son conectados vía una matriz de switches.

7.7.4 Tecnologías CPLD

terminología

la presentación actual tiende a usar terminología de

FPGA

Para el resto de las presentaciones, tenderemos a usar el término CLB más bien que el FB. Esto no significa que el contexto sea únicamente aplicable a las tecnologías FPGA.

7.8.0 ¿Dónde ocurren las cuestiones específicas a los PLD?

¿Que tiene que ver la síntesis con cuestiones específicas a las diferentes arquitecturas de los PLDs?

Así pues, habiendo visto por dentro la arquitectura de los dispositivos PLDs, podemos pasar ahora a estudiar las principales cuestiones relativas al estilo de escritura y síntesis del código VHDL para ese tipo de chips en específico.

7.8.1 Donde ocurren las cuestiones específicas a los PLD's

Eficiente mapeado a la arquitectura de los PLD

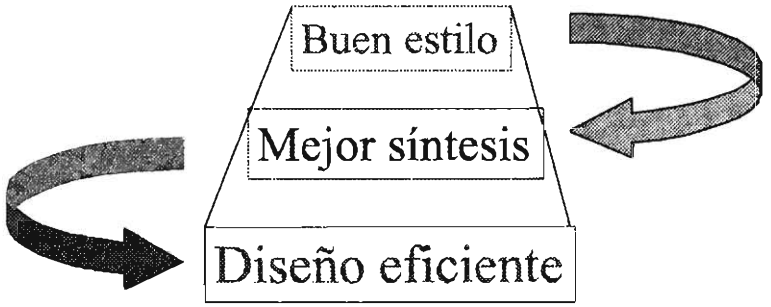
```
entity .... is
port (.....;
      .....;
      .....);
end ....;
architecture ..... of ..... is
begin
      process (.....)
      begin
            for .....
                .....
            end ....;
      end process ....;
end .....;
```

Diseños eficientes

Una buena herramienta de síntesis debe ser capaz de mapear el diseño RTL en una utilizacion eficiente de los CLBs. Sin embargo, el estilo en el cual se escribe el código puede ayudar a que la herramienta de síntesis obtenga mejores resultados.

7.8.2 Donde las cuestiones específicas a los PLD's ocurren

Código con buen estilo



Un código con buen estilo significa que la herramienta de síntesis pueda identificar construcciones dentro del código que fácilmente pueda mapear a características tecnológicas. Al escribir el código de esta forma, puede haber confianza acerca de como sera estructurada la lógica sintetizada.

7.8.3 Donde suceden las cuestiones específicas a los PLD's

usando recursos específicos a las familias PLDs.

¡Pensar mientras se escribe código VHDL en el 'hardware' específico a cada una de las arquitecturas!

7.8.4 Donde las cuestione específicas de los PLD's ocurren

mapeado a grandes bloques de construcción

Librería Xilinx	RAM
ROM	CONTADORES
RESTADORES	REGISTROS DE CORRIMIENTO
SUMADORES	FIFO

Para la mayoría de las tecnologías, Xilinx ha identificado funciones usadas comúnmente y optimizado las configuraciones de CLBs para implementarlas. Típicamente se trata de funciones como operaciones aritméticas, funciones de memorias estilo RAM y ROM, buses *3-state*, etc ,. . .

7.9.0 Cómo se manejan las cuestiones específicas a los PLD's

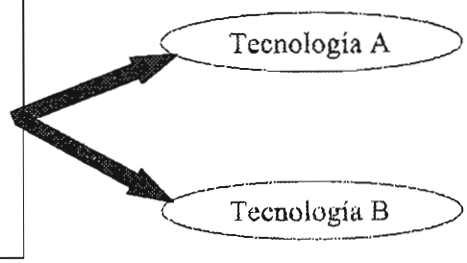
```
entity .... is
port (.....;
      .....;
      .....);
end ....;
architecture ..... of ..... is
begin
      process (.....)
      begin
            for .....
                .....
            end ....;
      end process ....;
end .....;
```

Diseño eficiente

La clave para usar eficientemente los recursos PLD es escribir el código VHDL de modo que haga 'el mejor uso' de las arquitecturas disponibles dentro de los dispositivos a usar. Sin embargo, los métodos para alcanzar esto deberán ser considerados cuidadosamente.

Independencia de la arquitectura

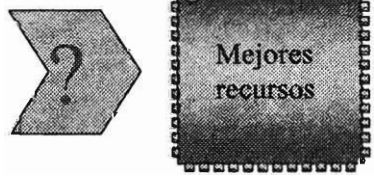
```
entity .... is
port (.....;
      .....;
      .....) ;
end ....;
architecture ..... of ..... is
begin
        process (.....)
        begin
                for .....
                .....
                end ....;
        end process ....;
end .....;
```



Una ventaja importante de diseñar mediante VHDL es que la descripción puede ser independiente de la arquitectura, y puede ser reusable a una nueva arquitectura si ello fuere menester. Sin embargo, el código que es **puramente genérico** puede hacer un uso deficiente de las cuestiones específicas de la arquitectura a la que se 'apunte'.

Independencia de la arquitectura -vs- eficiencia

```
entity .... is
port (.....;
      .....) ;
end ....;
architecture ..... of ..... is
begin
        process (.....)
        begin
                for .....
   Código independiente de la tecnología;
                end ....;
        end process ....;
end .....;
```



Se tiene que satisfacer un compromiso: podríamos querer mantener el código independiente de la arquitectura, dejando que la herramienta de síntesis infiera los mejores recursos para el dispositivo a emplear, o podríamos usar construcciones específicas a la arquitectura explotando funciones pre-optimizadas a expensas de la independencia de la arquitectura.

7.9.3 Como manejar las cuestiones específicas a los PLD's

la inferencia puede ser difícil

```
entity .... is
port (.....;
      .....;
      .....) ;
end ....;
architecture ..... of ..... is
begin
      código independiente de la tecnología;
      código específico a la tecnología;
      código independiente de la tecnología;
end .....;
```

En efecto, no todas las funciones pueden ser inferidas automáticamente por la herramienta de síntesis- por ejemplo, algunas herramientas no infieren RAMs eficientemente. Entonces, algunas veces se requiere hacer referencia a esas partes específicas de la tecnología dentro del código. Esto se conoce como una **instanciación**.

7.9.4 Como manejar las cuestiones específicas a los PLD's

Los requerimientos de la instanciación varían

```
entity .... is
port (.....;
      .....;
      .....) ;
end ....;
architecture ..... of ..... is
begin
      código independiente de la tecnología;
      código específico a la tecnología;
      código independiente de la tecnología;
end .....;
```

Para mantener la independencia de la arquitectura es preferible instanciar tan pocas celdas específicas de la arquitectura como sea posible. El número que debe instanciarse varía entre las herramientas de síntesis y las arquitecturas.

7.10.0 Resumen

Resumen

En esta presentación hemos visto una introducción a las diferentes tecnologías de Xilinx, y algunas cuestiones relacionadas a la síntesis a esas tecnologías como dispositivos destino, partiendo desde el HDL. Hagamos un resumen de los puntos tratados.

7.10.1 Resumen

Xilinx terminología

definición de la terminología...

Hemos discutido la terminología que emplearemos en el resto de las presentaciones.

7.10.2 Resumen

tecnologías de los FPGA y de los CPLD

CPLD

FPGA

Vimos aspectos generales de las arquitecturas de las tecnologías Xilinx y discutimos algunos puntos sobre sus implicaciones para la síntesis.

7.10.3 Resumen

Uso eficiente de los recursos

```
entity .... is
port (.....;
      .....;
      .....) ;
end ....;
architecture ..... of ..... is
begin
.....
   Código independiente de la tecnología
.....
....;
end .....;
```

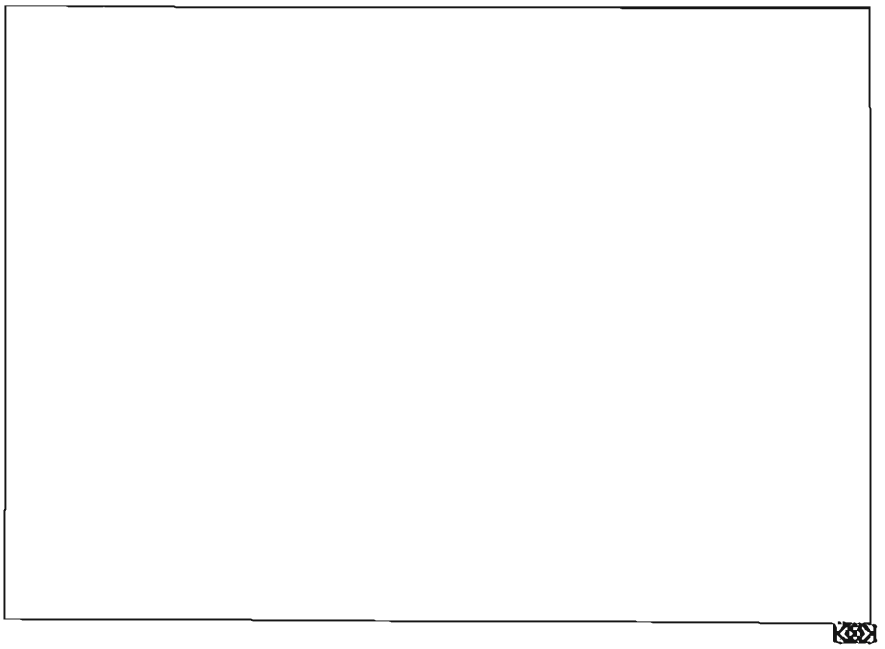
Discutimos que el ideal es usar los recursos tecnológicos del dispositivo destino de la mejor manera posible, pero debemos ser cuidadosos con eso, ya que podemos comprometer la independencia de la arquitectura, el cual es el ideal de los diseños VHDL.

7.10.4 Resumen

Conocer las características de la arquitectura

¡Pensar en el hardware específico para la arquitectura!

Concluímos que con el fin de alcanzar eficiencia es necesario conocer las "características" optimizadas de cada una de las arquitecturas tecnológicas y usarlas sea referenciandolas directamente o mediante el uso de código escrito con un estilo que pueda inferirlas fácilmente.



8. Estilos de codificado

estilos de escritura del código VHDL para uso eficiente de las características Xilinx.

Ahora veremos algunas de las características tecnológicas disponibles en las arquitecturas Xilinx. Veremos los estilos de escribir código para sacar ventaja de esos recursos.

8.1.1Estilos de la escritura de código y las características de Xilinx

terminología para el resto de la presentación

esta presentación usa terminología de los FPGAs.

Aclaremos el alcance de la siguiente presentación: es de notar que aunque la terminología se dirige hacia los FPGAs de Xilinx, sin embargo, muchas de las cuestiones también son aplicables a los CPLDs. Donde la diferencia sea notable, se hará la distinción.

8.1.2 Estilos de escribir código y las características Xilinx.

cuestiones más importantes por resaltar

resaltar las cuestiones que hay que saber segun la familia seleccionada,....

Tampoco se trata de discutir cuestiones particulares de ciertas familias individuales de dispositivos programables, sino tan solo hacer notar algunas cuestiones que deben ser consultadas tratandose de la serie de familias seleccionadas como destino del diseño VHDL ...

8.2.0 Usando módulos

uso de módulos predefinidos

Un factor muy importante para el uso eficiente de los recursos de los dispositivos FPGA o CPLDs es lo relacionado con la utilizacion de módulos. Xilinx hasta aquí ha identificado la mayoría de las funciones digitales complejas empleadas usualmente por la mayoría de los diseñadores y ha desarrollado módulos optimizados que pueden llevar a cabo esas tareas.

8.2.1 Usando módulos

Uso de módulos

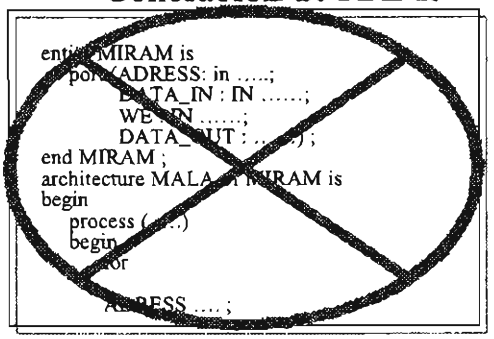
no se requiere describir en el código el comportamiento del módulo

la optimización ya ha sido hecha

Esto trae dos ventajas para los diseñadores VHDL: no necesitan describir el comportamiento de la función dentro de su código, y no necesitan preocuparse de optimizar la implementación de esa parte de su diseño, ya que esto ha sido hecho por Xilinx. Todo lo que se necesita es **instanciar** esos módulos dentro de su proyecto VHDL-*top*.

8.2.2 Usando módulos

Generación de RAMs



Un ejemplo de esto es cuando el diseño requiere un bloque de RAM. El comportamiento de las RAM puede fácilmente describirse en VHDL, pero la síntesis del código así escrito deberá resultar en una implementación deficiente de la estructura RAM pues seguramente obtendremos una configuración a base de latches o registros...

8.2.3 Usando módulos

Uso de funciones optimizadas

Libreria Xilinx

RAM

FIFO

Sumadores

Contadores

Registros de corrimiento

Restadores

ROM

... esto claro será una implementación ineficiente en términos del uso de los recursos del FPGA y del desempeño. **Xilinx** nos provee de funciones complejas usadas constante y comúnmente tales como RAM, ROM, operadores matemáticos, y funciones de corrimiento, en forma de bloques optimizados listos para usarse, de modo que basta con simplemente instanciarlos en el código.

8.3.0 Facilidad LogiBLOX

LogiBLOX

Por si lo anterior no fuese suficiente, **Xilinx** entrega un generador de módulos llamado *LogiBLOX*, el cual es una herramienta gráfica interactiva para crear módulos parametrizables, tales como contadores, registros de corrimiento, multiplexores,... *LogiBLOX* incluye tanto una librería de módulos genéricos y un conjunto de herramientas para personalizarlos.

8.3.1 Facilidad LogiBLOX

Generación de módulos personalizados

```
Library Su_Libreria ;
use Su_libreria.all ;
entity ... is
  port ( ..... ;
         ..... ;
         .....) ;
end ... ;
architecture ... of ..... Is
begin
        process (.....)
        begin
                for .....
                    .....
                end .....;
        end process ;
end .... ;
```

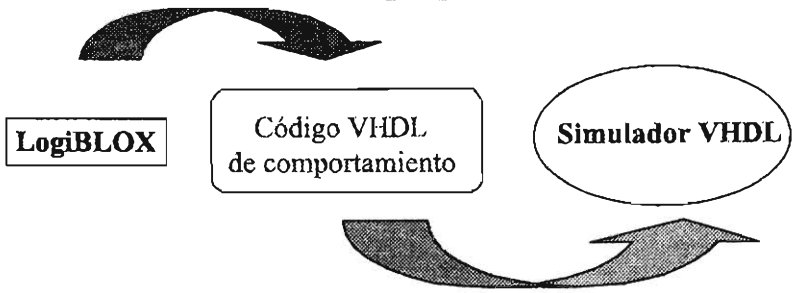
Su_libreria

LogiBLOX

Esto significa que se pueden crear librerías propias de funciones que están optimizadas hacia la arquitectura y tecnología que se prefiera como destino. Se pueden instanciar esas funciones en el código tanto y cuando se necesite.

8.3.2 Facilidad LogiBLOX

Generando sus propias librerías



LogiBLOX también genera al mismo tiempo modelos VHDL **simulables**, de modo que se pueden simular las funciones en un ambiente VHDL. Se requiere checar las funciones *LogiBLOX* disponibles para la serie de familias del chip objetivo, y crear implementaciones optimizadas de las funciones escogidas para la arquitectura específica.

8.3.3 Facilidad LogiBLOX

Inferencia de los módulos LogiBLOX

Z <= A + B ;

Inferencia

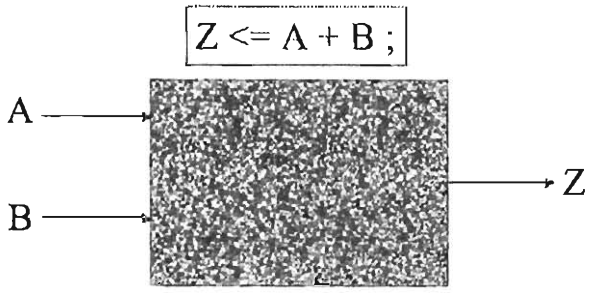
U1 : ADD16 portmap (. . . . .) ;

Instanciación

La mayoría de las herramientas de síntesis entienden los operadores aritméticos y pueden inferir estructuras simples o bien módulos *LogiBLOX* desde ellos. Sin embargo, si se require designar una estructura hardware particular a la función, entonces necesitaremos usar *LogiBLOX* para generarla...

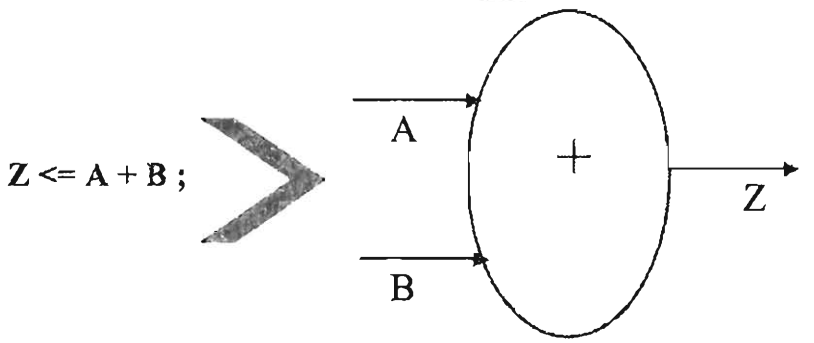
8.3.4 Facilidad LogiBLOX

los módulos LogiBLOX son generados como 'caja negra'



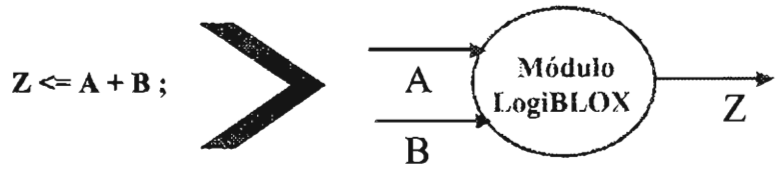
Una cuestión a recordar de los módulos *LogiBLOX*: ellos son módulos para arquitecturas específicas, implementados como 'cajas negras'. Esto significa que no es posible examinar o cambiar su estructura dentro de un editor esquemático o VHDL.

8.4.0 Síntesis de operadores aritméticos



En algunas ocasiones partes de código VHDL escritas en forma compacta resultan al ser sintetizadas en grandes piezas de lógica, y esto es cierto para el caso de los operadores aritméticos. Idealmente debe tenerse la manera de como controlar la implementación de tales funciones.

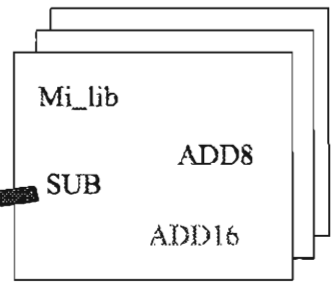
LogiBLOX para funciones aritméticas



Las funciones aritméticas se pueden generar de mejor manera mediante módulos *LogiBLOX*. Esto nos permitirá especificar una arquitectura particular según nuestras necesidades, tal como un sumador con acarreo en adelanto más bien que un sumador con acarreo de rizo.

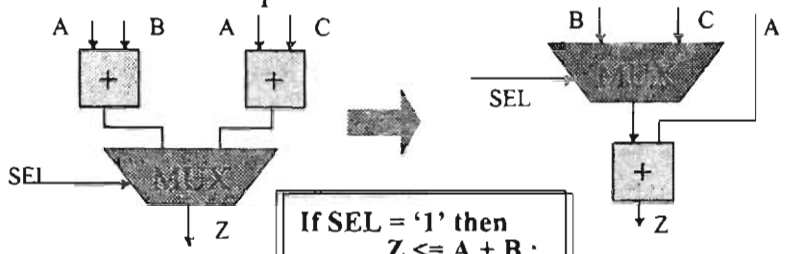
Instanciar módulos *LogiBLOX*

```
library Mi_Lib
use Mi_lib.all ;
entity ... is
  port ( ..... ;
         ..... ;
         .....) ;
end ... ;
architecture ... of ..... Is
component SUB port . . .
begin
U1 : SUB portmap . . . .
         process (.....)
         begin
                  for .....
                      .....
                  end .....;
         end process ;
end .... ;
```



Habiéndo generado su propia librería de módulos *LogiBLOX*, se puede 'instanciar' dentro del código.

Compartición de recursos

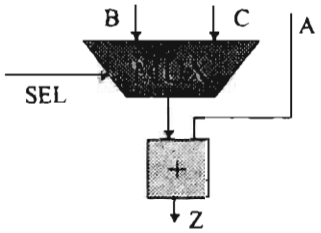



```
If SEL = '1' then
        Z <= A + B ;
else
        Z <= A + C ;
end if ;
```

Tal como se ha discutido previamente en las presentaciones anteriores, algunos operadores pueden ser compartidos entre operandos para usar justo un sólo operador en diferente tiempo. Esto se conoce como 'Compartición de Recursos'.

Compartición de recursos automática y manual

```
If SEL = '1' then
        X := B ;
else
        X := C ;
end if ;
Z <= A + X ;
```



La herramienta de síntesis Foundation Express puede implementar 'compartición de recursos' automáticamente, aunque no a través de las fronteras jerárquicas. Para las herramientas de síntesis que no poseen esta capacidad, se debe efectuar compartición de recursos manualmente, i.e. escribir el código con el mínimo número de operadores matemáticos.

8.5.0 Codificado de maquinas de estado

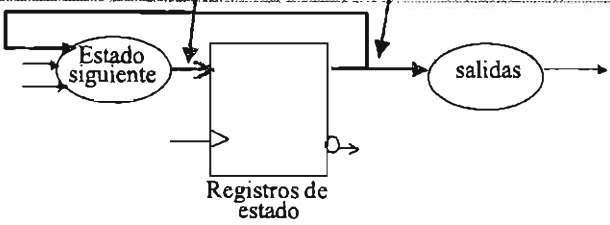
codificado de las máquinas de estado

Habiendo discutido como generar y usar módulos dentro de los diseños basados en VHDL, ahora veremos como se pueden implementar eficientemente máquinas de estado en dispositivos Xilinx a partir del VHDL...

8.5.1 VHDL para máquinas de estado

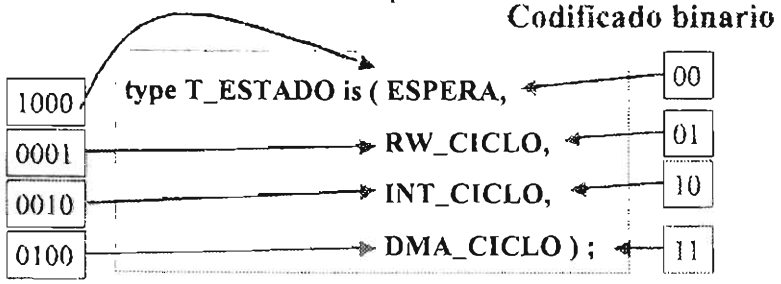
Uso de tipos enumerados

```
architecture RTL of MEF is
        type T_ESTADO is (ESPERA, RW_CICLO,
                          INT_CICLO, DMA_CICLO ) ;
        signal ESTADO_SIG, ESTADO_P : T_ESTADO ;
begin
...
```



La mayoría de las máquinas de estado escritas en VHDL usan los tipos enumerados para definir las señales de estado, tal como se muestra en el código de arriba.

Codificado por default

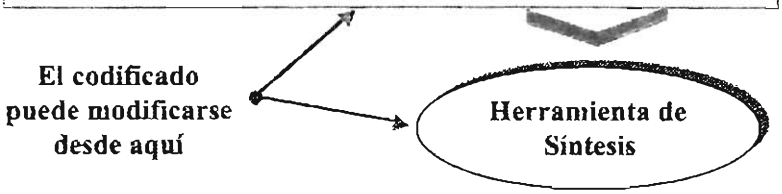


Codificado uno-por-estado

Por default en VHDL, un tipo enumerado es codificado como una secuencia binaria, mapeando desde el valor 0 a través de la secuencia de valores según son listados en la definicion del tipo. Sin embargo, en la herramienta ***Foundation Express***, el codificado por default es el estilo un flip-flop por estado (*one-hot* ~ un ff-por-estado).

Cambiando la codificación

```
architecture RTL of MEF is
        type T_ESTADO is ( ESPERA, RW_CICLO,
                           INT_CICLO, DMA_CICLO ) ;
        signal ESTADO_SIG,ESTADO_P  : T_ESTADO ;
```



En la mayoría de las herramientas de síntesis es posible cambiar las opciones de codificado ya sea con el código VHDL o mediante la selección de opciones dentro de la herramienta de síntesis misma. El método de modificar el codificado desde el mismo código no necesariamente es portable entre las diferentes herramientas de síntesis.

8.5.4 Codificado para máquinas de estado

Cambiando el codificado en Foundation

```
type ESTADO_TIPO is ( S1, S2, S3, S4) ;
attribute ENUM_ENCODING : STRING ;
attribute ENUM_ENCODING of ESTADO_TIPO :
type is "0001 0010 0100 1000" ;

signal C_EDO, N_EDO : ESTADO_TIPO ;
```

Dentro de las herramientas **Xilinx Foundation** puede escogerse entre codificado binario o un flip-flop por estado (uno-por-estado) desde la interfaz del usuario o bien especificar un codificado diferente desde dentro del código VHDL. Un ejemplo de la sintaxis VHDL la cual puede ser usada para cambiar el codificado se muestra aquí.

8.6.0 Uso de máquinas de estado con un flip-flop por estado (uno-por-estado)

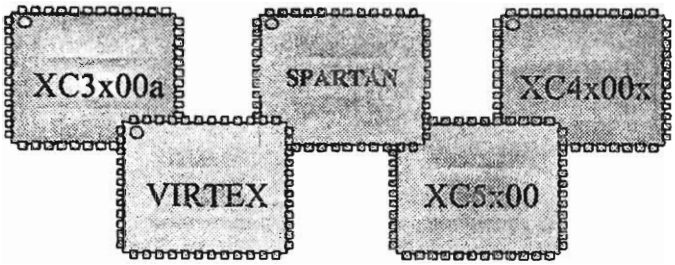
máquinas de estado uno-por-estado (1FF por estado)

El factor que produce un efecto mayor sobre la implementación eficiente de máquinas de estado en los FPGA es el uso o no de la codificación uno-por-estado o *one-hot*. Este tipo de codificación usa un registro para representar cada uno de los estados, comparado al número mínimo de registros que se usan en las codificaciones binarias o de Gray.

8.6.1 Uso de uno-por-estado en las máquinas de estado

Cuando usar uno-por-estado

Arquitecturas ricas en registros =>



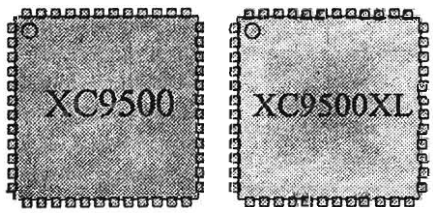
=> arquitecturas adecuadas para uno-por-estado

El codificado uno-por-estado da el mejor rendimiento para una máquina de estados, pero también usa la mayor cantidad de registros. Para las arquitecturas ricas en registros (XC3x00a, XC4x00x, XC5x00, Spartan, Virtex), la codificación uno-por-estado es la recomendada.

8.6.2 Uso de máquinas de estado uno-por-estado

Cuándo no usar uno-por-estado

Las arquitecturas con pocos registros =>



=> NO son las adecuadas para codificado uno-por-estado

Para las arquitecturas que poseen menos registros por compuerta lógica (típicamente las familias XC9500, y 9500XL), el codificado uno-por-estado puede no ser la mejor solución.

8.6.3 Uso de máquinas de estado uno-por-estado

Su decisión

El diseñador ¡decide!

Al final de cuentas, se requiere evaluar los compromisos y a partir de ahí hacer la decisión de cual codificado usar. Finalmente, en esta discusión de máquinas de estados, verémos como asegurarnos de que la operación de la máquina de estados sea "segura"....

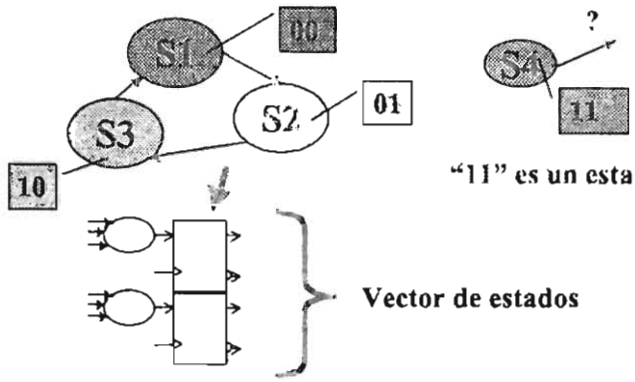
8.7.0 Máquinas de estados seguras

Máquinas de estados 'seguras'

Considere una máquina de estados con tres estados. En hardware, esta será implementada con al menos dos registros, los cuales por sí mismos pueden representar hasta cuatro estados diferentes.

8.7.1 Máquinas de estado seguras

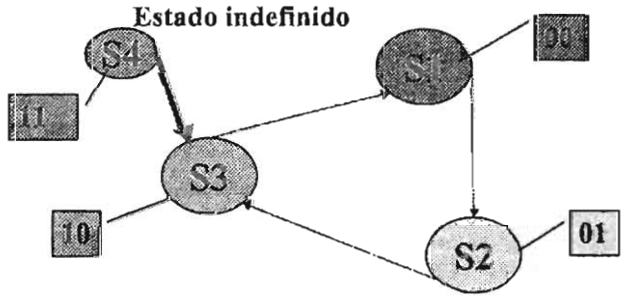
Estados ilegales



Esto significa que alguno de los posibles estados de los registros es un estado no especificado, y la operación de la máquina de estados no esta definida si por alguna razón los registros entran precisamente en ese estado.

8.7.2 Máquinas de estado seguras

Máquinas de estado seguras



Una máquina de estados segura es aquella que define como deberá comportarse el hardware si se entra en cualquiera de los estados indefinidos. Veamos las diferentes opciones para hacer seguras las máquinas de estados incompletamente especificadas....

Usando la cláusula '*others*'

Toma en cuenta a todos los estados no definidos

```
when OTHERS =>
        next_state <= s3 ;
```

Una forma rápida de hacer eso es usar la cláusula *others* en los postulados **case** y definir un estado siguiente legal en esta cláusula. De esto, la herramienta de síntesis **Foundation Express** construye lógica que envía la máquina de estados desde un estado ilegal a un estado conocido legal. Nótese que esta técnica no es soportada por todas las herramientas de síntesis.

Especificando estados ilegales

```
architecture RTL of FSM_Segura is
        type TIPO_ESTADO is bit_vector (1 downto 0 );
        signal ESTADO_SIG, ESTADO_P : TIPO_ESTADO ;
    process (ESTADO_P)
    begin
        case ESTADO_P is
                when "00" =>ESTADO _ SIG <= "01" ;
                when "01" => ESTADO_ SIG <= "10" ;
                when "10" => ESTADO _ SIG <= "00" ;
                when "11" => ESTADO _ SIG <= "00" ;
        end case ;
    end process;
```

Se define el comportamiento para el estado ilegal

Un método alternativo es especificar todos los posibles estados ilegales, de modo que su acción pueda ser definida. ¡más sin embargo esto puede volverse un proceso sumamente tedioso!

8.8.0 VHDL para la arquitectura a que se apunte

Pensar en el HARDWARE

Siempre que se escriba VHDL para síntesis, es esencial 'Pensar en Hardware'. Se necesita tener la seguridad de que el código escrito usa características que están disponibles para la arquitectura hacia la que se apunta. Esta sección ve algunos ejemplos específicos de Xilinx sobre esta cuestión....

8.8.1 VHDL para la arquitectura destino

preset y *clear* asíncronos

```
process ( CLK, CLR, PRSET )
        begin
                if (CLR = '1' ) then
                        Q <= '0' ;
                elsif PRSET = '1' then
                        Q <= '1' ;
                elseif CLK'event and CLK = '1' then
                        --acciones
                end if ;
end process ;
```

Checar que los registros en la familia Xilinx escogida soporte los tipos de funciones de set, reset y clear en la manera que se ha implicado su ejecución dentro del código. No todas esas funciones son implicadas en cada una serie de familias.

8.8.2 VHDL para las arquitecturas apuntadas.

latches transparentes

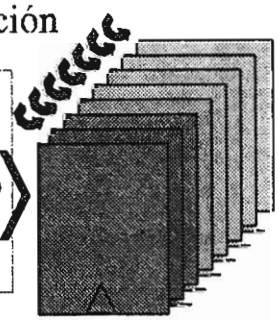
checar que la arquitectura soporta las características...

Si en el diseño se desea implementar latches transparentes, checar que ellos son soportados y como se pueden inferir. Checar el archivo de síntesis '*log file*' o los reportes de la síntesis para asegurarse que accidentalmente no se han inferido *latches*, particularmente para las arquitecturas que no los soportan.

8.8.3 VHDL para la arquitectura que se trate

Postulados de generación

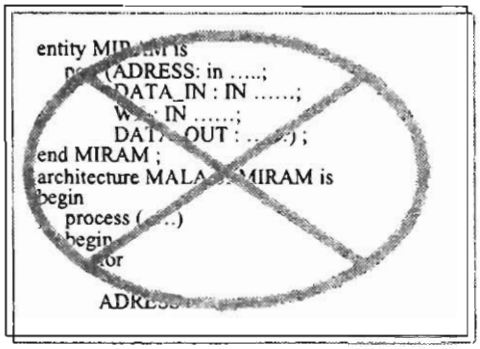
```
GEN_BLOX :
        for i in 0 to 16 generate
                BLOX : REG portmap (. . .)
        end generate GEN_BLOX ;
end GEN ;
```



Los postulados de generación son una construcción VHDL muy útil, pero en ocasiones pueden crear una cantidad excesiva de lógica y tomar gran cantidad de tiempo en el proceso de síntesis. Esto se debe a que la herramienta de síntesis requiere 'desenrollar' o 'replicar' la lógica implicada por los postulados de generación.

8.8.4 VHDL para la arquitectura 'blanco'

Inferencia de RAM



Recuerde que si se desea usar una función RAM, se deberá usar *LogiBLOX* para generarlas y entonces instanciarla. Si se trata de sintetizar RAMs a partir del código, con toda seguridad se obtendrá una implementación deficiente.

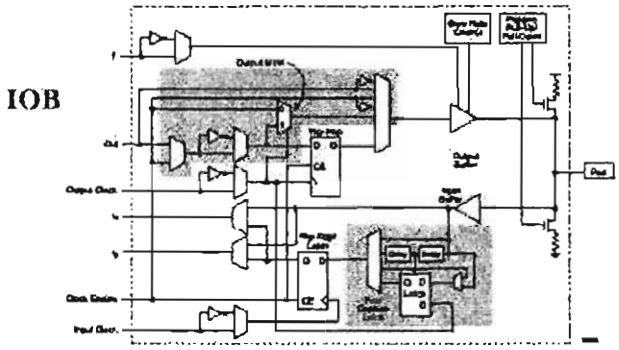
8.9.0 Características I/O

Características I/O de los dispositivos Xilinx

Habiendo visto ya algunos de los módulos de alto nivel disponibles en la tecnología Xilinx, veámos ahora algunas componentes tecnológicas mas pequeñas disponibles en los chips Xilinx, tales como los buffers de reloj y de tercer estado (*3-state*). Empecemos estudiando las celdas de entrada/salida de dichos dispositivos Xilinx.

8.9.1 Características I/O

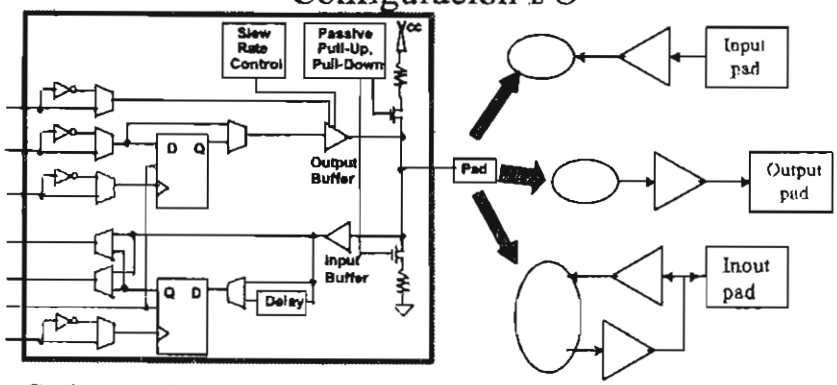
Bloques de Entrada/Salida
características I/O de Xilinx



Los IOBs (*Input/Output Blocks*) bloques de Entrada/Salida son los bloques que proveen las señales de interfaz entre las patillas externas del paquete y las líneas de las señales internas.

8.9.2 Características I/O

Configuración I/O



Cada uno de los IOBs controla un pin del paquete y puede ser configurado para proveer diferentes tipos de buffers como entradas (*in*), salidas (*out*), o *pins* bidireccionales (in/out). En la siguiente sección, nos concentraremos en como se pueden insertar los diferentes tipos de buffers en los diseños.

8.10.0 Definicion de I/O

Definición de I/O

Hay dos pasos incluídos en la definición de I/O. El primero es la definición del tipo de ***buffer*** (*input*, *output*, *tristate*, etc..). El segundo es hacer la **asignación del pin**, el cual es el proceso de asignar una señal I/O a una patilla en particular del paquete que contiene al chip. Veamos en primer término la definición del tipo del ***buffer*** I/O.

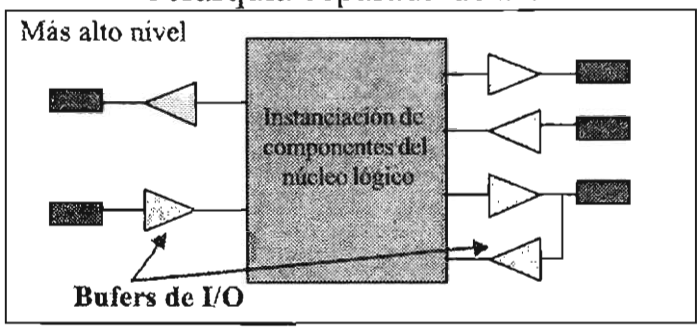
8.10.1 Definición I/O

Definición del tipo en el código inferir o instanciar I/O dentro de Foundation Express

Se puede instanciar o inferir desde el código los tipos de ***buffers*** **I/O**. La herramienta de síntesis ***Foundation Express*** es capaz de inferir automáticamente todos los diferentes tipos de *buffers* I/O. Veremos ejemplos de su instanciación con el código más adelante.

8.10.2 Definición I/O

Jerarquía separada de I/O



Si la herramienta de síntesis no infiere IOBs o si se ha escogido instanciar ellos desde el código, es buena práctica definir los bloques I/O en un nivel de jerarquía separado. Usando este método, en el más alto nivel de jerarquía se instancia el núcleo lógico del diseño al mismo nivel que cada uno de los IOBs.

8.10.3 Definición de I/O

Definición del tipo en la herramienta de síntesis

La definición de I/O en la herramienta de síntesis

- vía menus
- vía archivo de control

En algunas herramientas de síntesis se puede efectuar un paso separado para la definición de I/O. Se permitirá seleccionar tipos de buffers y asignarlos a los puertos del núcleo lógico. Alternativamente, es posible hacer esas definiciones en un archivo de control.

8.10-1 Definicion I/O

Asignación de las patillas I/O

Asignación de los pines en las herramientas Foundation

- vía archivo de control de usuario (*.UCF)

Finalmente, consideremos la **asignación de pines**. Cuando se usan las herramientas Xilinx, la asignación de pines se especifica con lo que se llama Archivo de Control de Usuario (*User Control File* (*.UCF)).

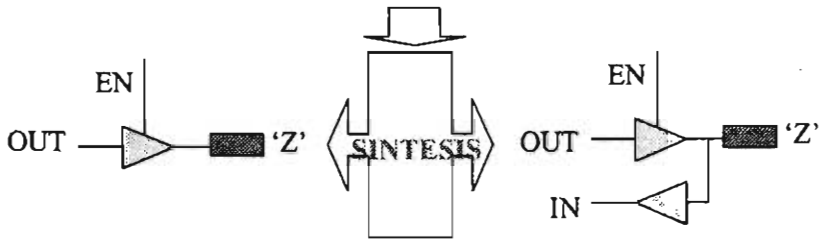
8.11-0 Usando buffer 3-estado

3-estado y E/S bidireccional

La mayoría de las familias de dispositivos Xilinx soportan I/O con registros, 3-estado, I/O bidireccional, o varias combinaciones de ellos. Veamos primero como los elementos 3-estado e I/O bidireccional son manipulados....

Salidas 3-estado

Código VHDL

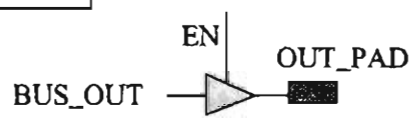


La mayoría de las herramientas de síntesis, incluyendo, claro esta, a ***Foundation Express***, tienen la capacidad para automáticamente inferir tanto salidas 3-estado como celdas I/O bidireccionales desde el código VHDL adecuado.

Estilos de escritura del código para inferir 3-estado

```
If ( EN = '0' ) then
        OUT_PAD <= BUS_OUT ;
else
        OUT_PAD <= 'Z' ;
end if ;
```

SINTESIS

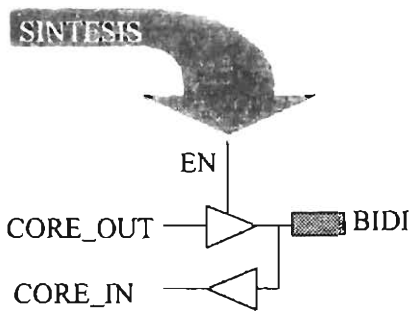


El estilo de escritura apropiado para inferir la función 3-estado en una celda de I/O, usa postulados como los mostrados en el ejemplo presente. Es mediante el uso de la asignación 'Z' que habilita la herramienta de síntesis para inferir el buffer de 3-estado.

Estilo de escritura para inferir I/O bidireccional

```
CORE_IN <= BIDI ;
process ( CORE_OUT, EN )
    if (EN = '0' ) then
        BIDI <= CORE_OUT ;
    else
        BIDI <= 'Z' ;
    end if ;
end process ;
```

Postulado concurrente



El código mostrado aquí genera bloques o celdas de I/O bidireccionales. Nótese de nueva cuenta el uso de la asignación del valor 'Z'.

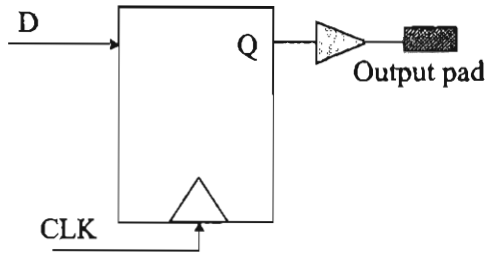
I/O con registros

(I/O registered)

Finalmente en este repaso del acceso de las celdas I/O desde el VHDL, consideremos la implementación de registros manejados directamente con las señales de entrada y salida o I/O síncrona.

8.12.1 Accesando I/O con registros

I/O con registros



Para la mayoría de las herramientas de síntesis, incluída ***Foundation Express***, la implementación de I/O con registros se efectúa automáticamente empleando un registro de la celda I/O apropiada si es que él esta disponible. Pero debe de tenerse cuidado, hay que asegurarse que la arquitectura soporta el uso implicado de la habilitación del reloj y los reset en la celda I/O.

8.12.2 Accesando I/O con registros

inserción de registros durante MAP

El switch "-pr" en MAP habilita a un FF a ser empujado dentro de I/O. (Off por default)

Adicionalmente, las herramientas de implementación de Xilinx tienen un switch que permite empujar los registros dentro de las celdas IOB's cuando es apropiado.

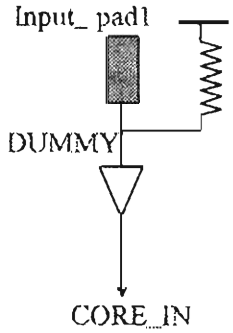
I/O bidireccional y 3-estado con registro

```
CORE_IN <= BIDI ;
process ( CLK )
        if CLK'event and CLK = '1' then
                if (EN = '0' ) then
                        BIDI <= CORE_OUT ;
                else
                        BIDI <= 'Z' ;
                end if ;
        end if ;
end process ;
```

Si se desea hacer uso de registros con I/O, 3-estado, o bidireccional, entonces esta función puede ser inferida desde el código VHDL por las herramientas Foundation.

Pull-up's

```
architecture ESTRUCTURAL of PULLUP_EJ is
  component PULLUP
        port ( O : out std_logic ) ;
  end component ;
  component IBUF
        port (    I : in std_logic ;
                  O : out std_logic ) ;
  end component ;
  signal DUMMY, CORE_IN :std_logic ;
begin
  DUMMY <= INPUT_PAD1 ;
  PU : PULLUP portmap ( DUMMY) ;
  IN : IBUF portmap (DUMMY, CORE_IN ) ;
end ESTRUCTURAL ;
```



Muchas familias de dispositivos Xilinx tienen recursos I/O los cuales permiten que resistencias *pull-up/down* sean conectadas a las pads de IO: Ellas pueden instanciarse, tal como se hace en el ejemplo. Alternativamente, las herramientas ***Foundation Express*** tienen un método gráfico de especificar el uso de *pull-up* o *pull-downs* para cada una de las I/O.

8.13.0 Requerimientos de enrutado especiales

Requerimientos de enrutado especiales

Los recursos de enrutamiento especiales de algunas arquitecturas PLD son a veces requeridos para las conexiones con gran fan-out tales como las señales de *reset* y de *clock*, resultando en una mejor velocidad y menor '*time-skew*'. Para utilizar esos recursos de enrutado especiales se deben especificar las conexiones especiales a la herramienta de colocado y enrutado.

8.13.1 Requerimientos de enrutamiento especiales

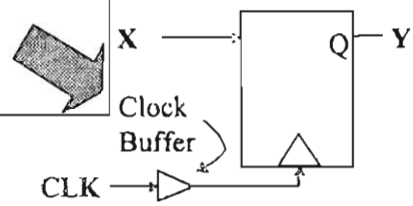
Set/Reset global (*GSR*)

```
entity GSR_EJ is port (CLK, RST  : in std_logic ;
                       Q : out std_logic_vector ( 0 to 3 ) ) ;
end GSR_EJ ;
architecture ESTRUCTURAL of GSR_EJ is
  component STARTUP   port ( GSR : in std_logic ) ;
  end component;
begin
  U1 : STARTUP port map ( GSR => RST ) ;
  process (CLK, RST )
  begin
        if RST = '1' ;      Q <= "0001" ;
        elsif CLK'event and CLK = '1' then
                Q <= Q (1 to 3 ) & Q (0) ;
        end if ;
  end process ;
end ESTRUCTURAL ;
```

Si la herramienta de síntesis no puede inferir la red de *reset* por sí misma, se puede usar mediante la instanciación de la componente *STARTUP*, cuya entrada es conectada a todas las conexiones de *RESET* del dispositivo, vía una componente llamada *GSR*.

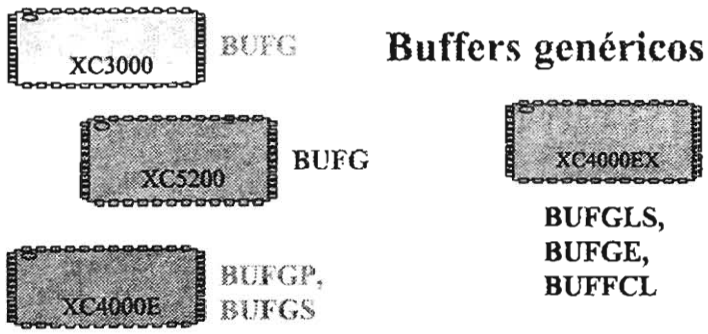
Redes de distribución de relojes

```
If CLK'event and CLK = '1' then
        Z <= X ;
end if ;
```



Algunas herramientas de síntesis, incluyendo ***Foundation Express***, examinan la conectividad de un diseño e instancían automáticamnete unos *buffers* globales sobre las señales de reloj. Si el software no hace esto, entonces se necesita instanciar una celda de ***buffer*** **global** para manejar la red de distribución del reloj.

Usar el *buffer* genérico del reloj



A menos que se requiera usar un buffer global específico, es mejor usar el recurso genérico de enrutado global ***BUFG***. Las herramientas de colocado y enrutado pueden sustituir este buffer por otro recurso disponible mejor. El uso de cualquier otro esquema de buffers significa que el software no es libre de hacer la mejor elección.

8.13.1 Requerimientos de entrada especiales

Cuestiones sobre la portabilidad

checar sobre la disponibilidad de 'buffers'

Debe tenerse en cuenta que muchas arquitecturas Xilinx tienen *buffers* globales los cuales son disponibles sólo a esas arquitecturas, de modo que el uso de *BUFG* permite una transición más fácil entre familias. También, el número de *bufers* de reloj disponibles varía, de modo que hay que revisar cuantos se necesitan y cuantos estamos implicando.

8.14.0 Señales 3 estado internas

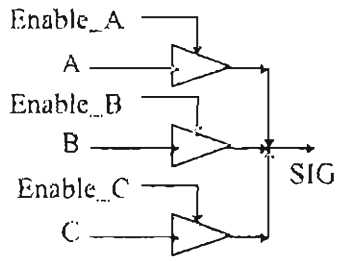
Señales internas '3-estado'

Veámos ahora el uso de buffers 3-state internos, los cuales pueden liberar gran cantidad de recursos dentro del diseño. Ellos son muy útiles para implementar multiplexores o funciones de lógica alambrada. Los *buffers* 3-estado pueden configurarse en tres modos: '3-estado', '*and* alambrada', y '*or-and* alambrada'.

8.14.1 Señales 3-estado internas

Inferencia 3-estado

```
A_OUT <= A when ENABLE_A else 'Z' ;
B_OUT <= B when ENABLE_B else 'Z' ;
C_OUT <= C when ENABLE_C else 'Z' ;
process (A_OUT)
begin
        SIG <= A_OUT ;
end process ;
process (B_OUT)
begin
        SIG <= B_OUT ;
end process ;
process ( B_OUT )
begin
        SIG <= C_OUT ;
end process ;
```



Este código muestra una alternativa de la descripción "if" usual de un convertidor de código 3 a 1 usando inferencia de 3-estado. Resulta en una eficiente implementación de un mux 3 a 1, utilizando buffers de 3-estado.

8.14.2 Señales 3-estado internas

Lógica alambrada

las expresiones

BOOLEANAS

no dan

lógica alambrada

La lógica alambrada puede ser una buena forma de reducir el área y complejidad de algunos tipos de lógica combinacional. Muchas herramientas no son capaces de inferir lógica alambrada desde operadores booleanos de modo que se necesita usar la instanciación para crear esta estructura de lógica.

8.14.3 Señales 3-estado internas

Decodificadores anchos

usar decodificadores de borde ancho

para decodificar

grandes funciones

Una función muy común es la de decodificar direcciones y muchos dispositivos Xilinx tienen circuitería dedicada alrededor del perímetro del dispositivo de modo que las señales de entrada pueden ser decodificadas. Este tipo de 'decodificadores de borde ancho' se implementan como una serie de celdas and-alambrado con una *pull-up* conectada a la señal de 3-estado.

8.14.4 Señales 3-estado internas

Implementación de decodificadores anchos

...consúltese la guía de librerías...

Para sacar provecho de los decodificdores de extremo anchos, se necesita simplemente instanciar los símbolos de las librerías *WAND*. Sin embargo se requiere leer los manuales de las librerías para guiarse en el tipo de decodificadores que están disponibles para la familia seleccionada.

8.15.0 Busqueda de la frontera

Boundary scan

El último tópico que se verá, trata del tópico de la implementación de funciones de ***boundary scan*** con los dispositivos Xilinx.

8.15.1 Busqueda de la frontera

Los pines no son dedicados

Boundary scan

Los dispositivos Xilinx XC4000 y XC5000 contienen registros boundary-scan que son compatibles con el estandar IEEE 1149.1. Se soportan las I/O externas y las pruebas de interconexión; existe también un soporte limitado para autopruebas internas.

8.15.2 Busqueda de la frontera

Los pines boundary scan no son dedicados

los pines de
boundary-scan
pueden usarse como
pines de I/O normales.

Si *boundary scan* no se emplea después de que el dispositivo ha sido configurado, el usuario puede a su gusto emplear los *pads* de *boundary scan* como pines de entrada o de salida. Igual que los IOBs regulares, esos pines de entradas y salidas tienen disponibles resistencias de *pull-up/down*.

8.15.3 Busqueda de la frontera

XILINX®

Página WEB de XILINX
www.xilinx.com

Hasta aquí este tema ... 'mas información de *boundary scan*' en Xilinx....

8.16.0 Resumen

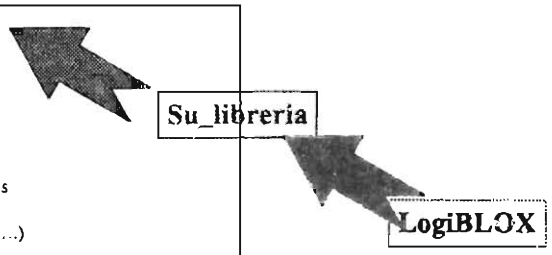
Resumen

En resumen, hemos discutido las siguientes cuestiones considerando los diseños que apuntan a los dispositivos Xilinx usando VHDL:

8.16.1 Resumen

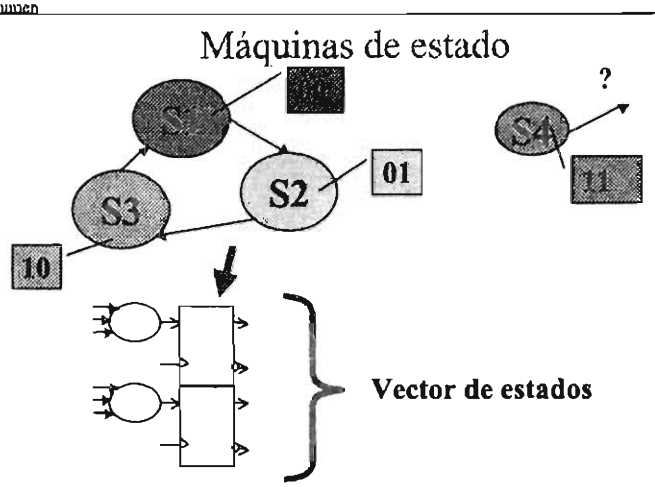
LogiBLOX

```
Library Su_Libreria ;
use Su_libreria.all ;
entity ... is
  port ( ... ..... ;
         ..... ;
         .....) ;
end ... ;
architecture ... of ..... Is
begin
        process (.....)
        begin
                for .....
                    .....
                end .....;
        end process ;
end .... ;
```



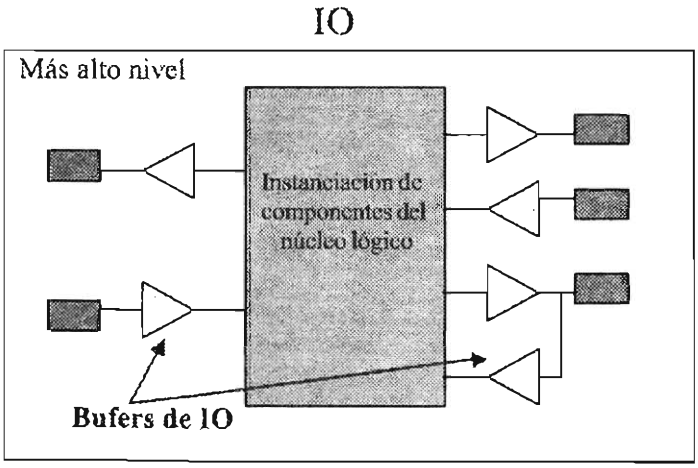
Primero vimos como usar *LogiBLOX*, el cual nos permite generar, usar y crear librerías de módulos útiles propias.

8.16.2 Resumen



Luego discutimos la manera de describir máquinas de estados eficientes, incluso de cómo hacerlas 'seguras'.

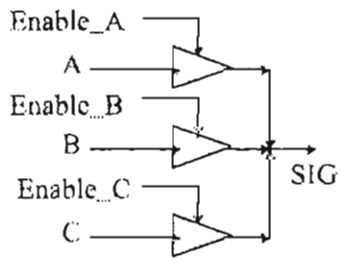
8.16.3 Resumen



Vimos también las diferentes formas de definir las I/O para el dispositivo.

Inferencias 3-estado

```
A_OUT <= A when ENABLE_A else 'Z' ;
B_OUT <= B when ENABLE_B else 'Z' ;
C_OUT <= C when ENABLE_C else 'Z' ;
process (A_OUT)
begin
        SIG <= A_OUT ;
end process ;
process (B_OUT)
begin
        SIG <= B_OUT ;
end process ;
process ( B_OUT )
begin
        SIG <= C_OUT ;
end process ;
```



Finalmente, discutimos formas de usar las características 3-estado para crear implementaciones eficientes de muxes y de decodificadores anchos.

9. Flujo y control de los diseños

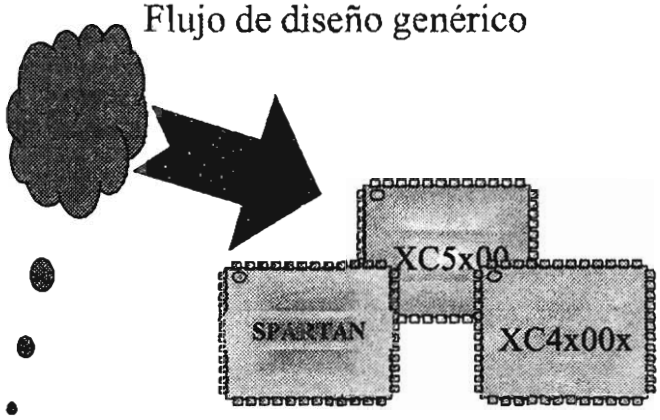
9.1.0 Introducción al flujo de diseño

flujos de diseño VHDL para PLDs Xilinx

En esta novena presentación veremos con mayor detalle el flujo de diseño para los dispositivos Xilinx.

9.1.1 Introducción al flujo de diseño

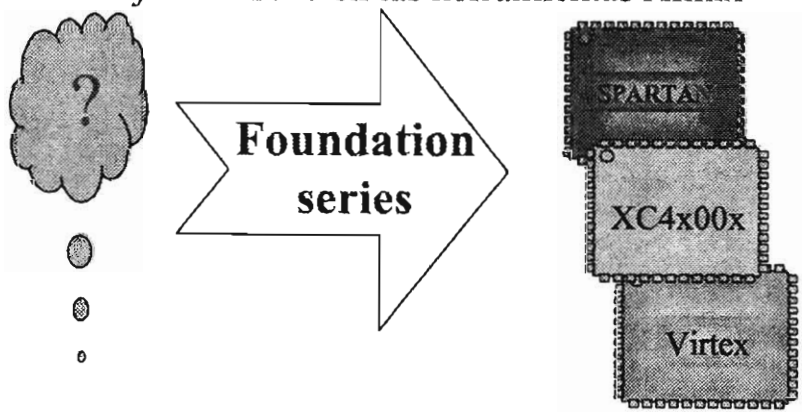
Flujo de diseño genérico



- Empezaremos describiendo el flujo de diseño VHDL genérico, en otras palabras los pasos que se aplican a cualquier diseño VHDL independiente del chip al cual se apunte.

9.1.2 Introducción al flujo de diseño

El flujo de diseño en las herramientas Xilinx



Luego entonces veremos como el flujo ajusta dentro de las herramientas de diseño ***Foundation Series*** y en las tecnologías Xilinx.

9.2.0 El flujo de diseño genérico

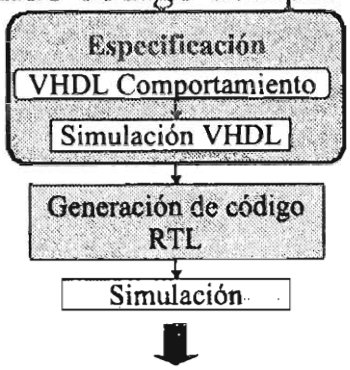
el flujo de diseño genérico

Permítasenos iniciar describiendo el flujo de diseño genérico el cual se aplica a casi cualquier ruta HDL y a cualquier tipo de tecnología PLD.

9.2.1 El flujo de diseño genérico

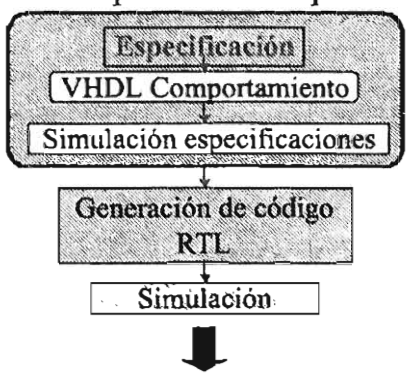
El flujo de diseño deberá iniciar siempre con las especificaciones, ya sea que consten tan sólo de algunos pocos algoritmos simples relacionando las salidas con las entradas, o un gran documento escrito en algún idioma normal o aún VHDL.

Escribiendo código comportamiento



Algunas veces, los trabajos de diseño VHDL se inician produciendo una Descripción VHDL comportamiento para posteriormente convertir cada uno de los bloques funcionales en RTL. Esto por lo general se hace únicamente para nuevos desarrollos de sistemas digitales muy grandes diseños con más de 100k '*gates*'.

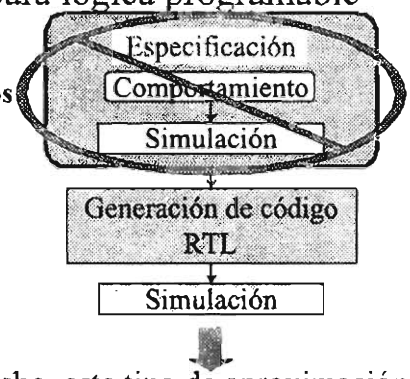
Simulación de comportamiento para comparación



La ventaja de escribir primero un modelo VHDL comportamiento es que se pueden simular efectivamente las especificaciones para asegurar que son correctas, y entonces usar los resultados para posteriores comparaciones con la simulación del modelo RTL sintetizable.

RTL para lógica programable

No se requiere para los diseños pequeños a medianos con PLD



Tal como hemos dicho, este tipo de aproximación es sólo útil para grandes diseños. Para un diseño típico de lógica programable, no es generalmente necesario usar una metodología tan rigurosa. La escritura del código a nivel de RTL es más que suficiente, de modo que en el resto del capítulo nos basaremos en las entradas de diseño RTL.

–Así pues, habiendo concluido que típicamente iniciaremos el flujo de diseño en la etapa de 'Generación de código VHDL al nivel RTL', veamos como debemos proceder en los pasos siguientes.

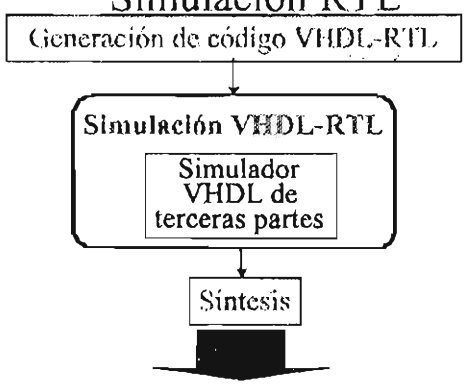
9.3.1 Simulación y Síntesis

Generación de código VHDL-RTL

La entrada de código usualmente se efectúa vía algún editor de textos. Algunos editores son diseñados específicamente para el VHDL y mediante ellos es posible obtener ayuda para la correcta generación del código y para checar sintaxis. Al mismo tiempo en que se está generando código VHDL-RTL, puede estarse generando código VHDL *test bench* para simular la RTL que se ha escrito.

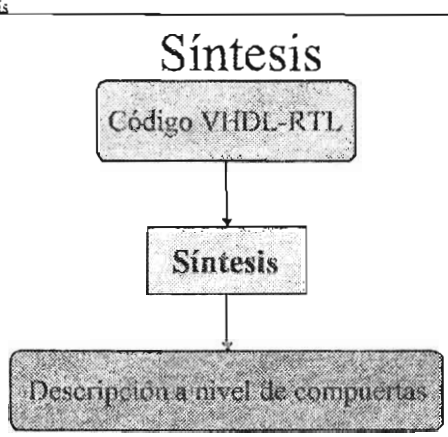
9.3.2 Simulación y Síntesis

Simulación RTL



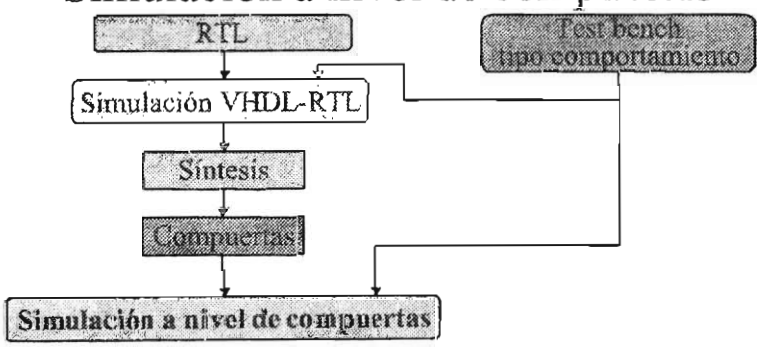
A modo de poder verificar la funcionalidad del código RTL recien generado, se requerirá un simulador VHDL. Algunos diseñadores no ejecutan este paso de simulación RTL y van directo a la síntesis. El esfuerzo empleado en simular RTL puede ser grande y conducir a grandes ciclos de iteración.., pero debe ser satisfactorio para poder seguir adelante.

9.3.3 Simulación y Síntesis



El trabajo de la herramienta de síntesis es traducir la descripción VHDL-RTL del diseño en una descripción lógica. En otras palabras, la herramienta de síntesis toma como entrada la descripción VHDL del diseño y la convierte en una descripción netlist al nivel de compuertas.

9.3.4 Simulación y Síntesis



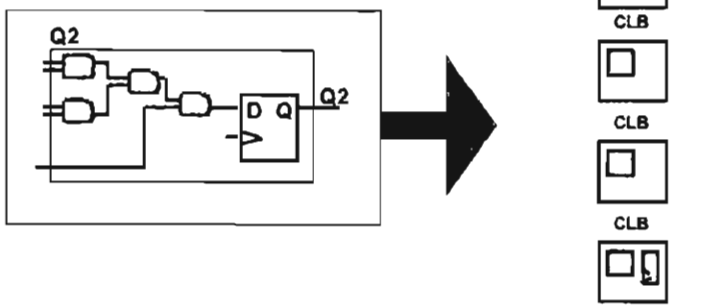
Una vez sintetizado, es necesario checar que la herramienta de síntesis ha generado una descripción lógica que tiene el mismo comportamiento que el RTL. Esto se hace usando el mismo test bench para simular la netlist a nivel de compuertas tal como fue usado para simular la descripción RTL, y comparando los resultados de ambas simulaciones.

9.4.0 Implementación

Una vez que tenemos la netlist lógica funcionalmente correcta, entonces podemos pensar en traducir ésta en una descripción física. Este proceso se llama implementación. El primer paso de la implementación incluye el mapeo, luego el colocado y enrutado, y finalmente la programación del dispositivo al que se bajará el archivo de configuración.

9.4.1 Implementación

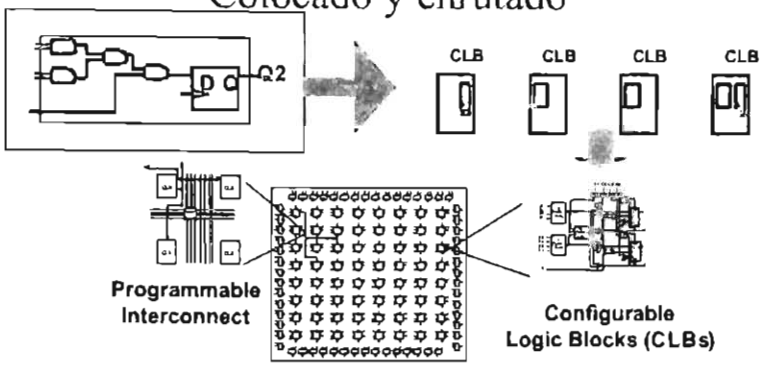
Mapeado



El mapeado es el proceso de convertir la descripción lógica o netlist generada por la herramienta de síntesis en términos de los diferentes recursos propios del dispositivo, tales como los CLBs, FBs, *buffers*, ...

9.4.2 Implementación

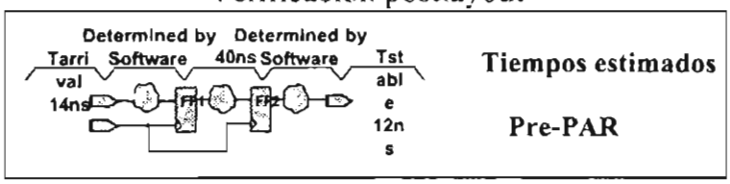
Colocado y enrutado

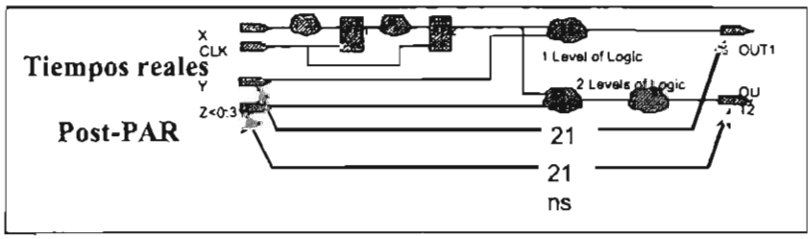


El colocado consiste en la asignación de los recursos mapeados en localidades específicas del dispositivo. El enrutado se efectúa entonces mediante la interconexión entre esos bloques de construcción seleccionados. Si se da algún requerimiento de tiempos a las herramientas de colocado y enrutado, entonces el enrutado será efectuado para tratar de cumplir esas restricciones de tiempo.

9.4.3 Implementación

Verificación postlayout

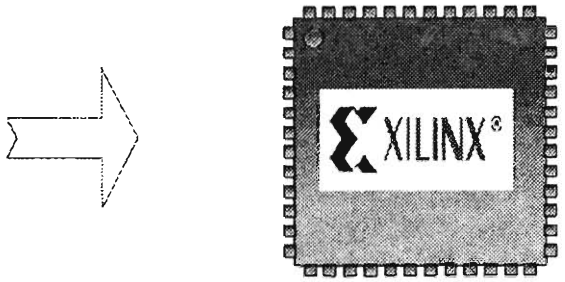




Una vez que tenemos la descripción física efectiva del diseño, los retardos de las trayectorias se conocen con precisión. Para revisar que ellos no cambian la funcionalidad del diseño, debemos verificar que los tiempos están aún dentro de las especificaciones.

9.4.4 Implementación

Programación



El paso final es programar el dispositivo con los archivos de diseño que se han creado. Ahora veremos como se lleva a cabo todo ese flujo de diseño VHDL en particular para los dispositivos Xilinx.

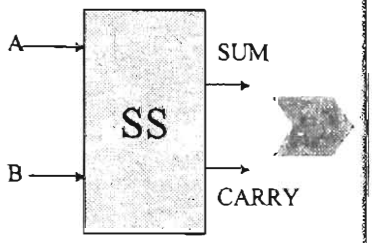
9.5.0 Entrada del código

Entrada del código

La herramienta ***Foundation Series Software*** tiene una buena cantidad de características útiles que nos ayudan a generar el código VHDL. Veamos esas ahora...

9.5.1 Entrada de código

Generación de la entidad




```
entity SEMISUM is
    port (A, B : in bit ;
          SUM, CARRY : out bit ) ;
end SEMISUM ;
```

Cuando creamos un nuevo archivo VHDL, tenemos la opción de seleccionar un '*Design Wizard*'. Debemos introducir información acerca de las I/O de nuestro diseño y el '*Wizard*' crea por nosotros la descripción de la entidad. Podemos especificar puertos de tipo *bit*, *boolean*, *character*, *integer*, o *std_logic*.

9.5.2 Entrada de código

Templetes de código

```
entity is
    generic( );
    port (   : in std_logic_vector( );
             :out std_logic_vector( ) );
end;
```

```
case is
when =>
when =>
when others =>
end case;
```

```
if then
elsif then
elsif then
elsif then
else
end if;
```

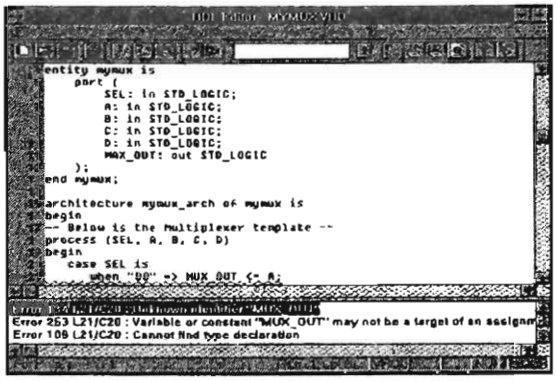
```
if then
else
end if;
```

'formas' de código

Mientras edita, podemos llamar una ventana que contiene templetes de sintaxis – esta característica se llama el 'Asistente de Lenguaje'. Los templetes nos guían acerca de cuestiones del empleo del lenguaje (e.g. sintaxis de declaración de componentes) o con cuestiones de síntesis (e.g. estilo para inferir un flip-flop tipo D). También se pueden personalizar agregando templetes propios.

9.5.3 Entrada de código

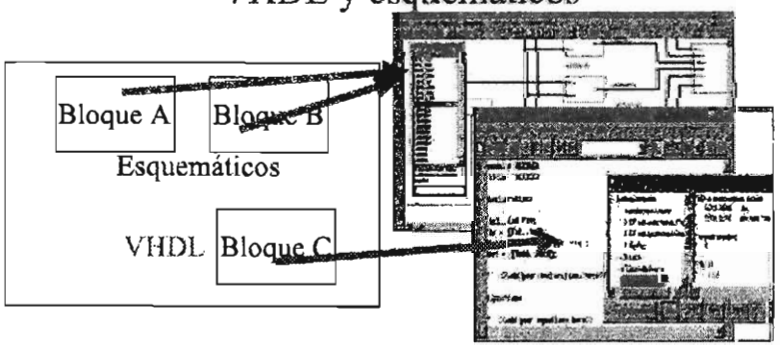
Revisión de sintaxis



Existe también una función revisora de sintaxis. Esto significa que se puede fácilmente checar la calidad del código sin tener que efectuar una síntesis completa. Es importante usar esta característica de vez en vez para ir depurando el código según se va escribiendo.

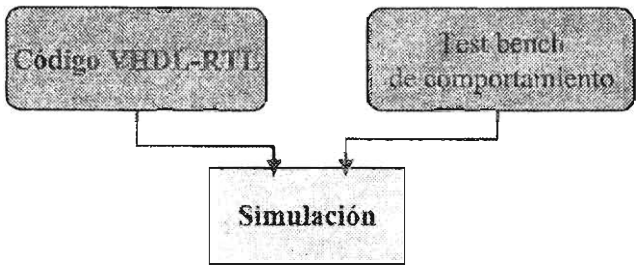
9.5.4 Entrada de código

VHDL y esquemáticos



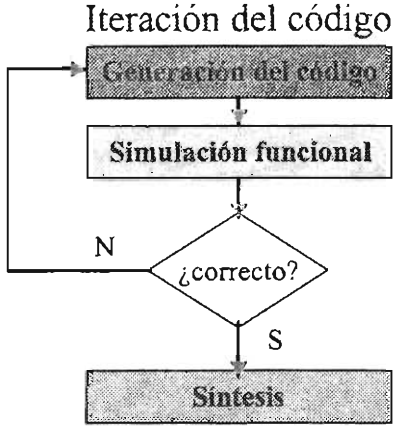
Foundation permite mezclar diseños VHDL-*top* con esquemáticos o a la inversa. Se puede hacer eso, colocando un símbolo dentro de un esquema el cual representa código VHDL. Cuando se "pica" en el símbolo el editor VHDL es llamado. Para crear este tipos de símbolos desde el editor, se debe usar el comando 'Crear Macro' en el menú '*Project*.

9.6.0 Simulación RTL



Habiendo discutido como se captura el código VHDL, pasemos ahora a la simulación del código RTL. Se puede simular el RTL conforme se vaya escribiendo, es decir bloque por bloque. Esto requiere por otro lado escribir un test bench para cada una de las partes del código.

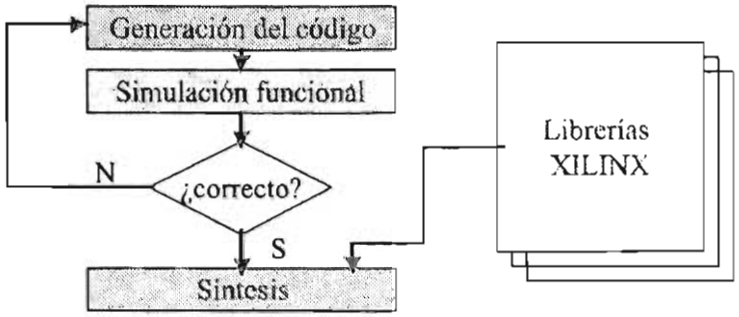
9.6.1 Simulacion RTL



Se puede necesitar iterar varias veces entre la entrada de código y la simulación antes de quedar satisfecho con el código. Ciertamente, el proceso de flujo de diseño completo es iterativo y se pueden requerir repetir muchos de los pasos hasta obtener los resultados buscados.

9.6.2 Simulación RTL

Requerimientos de Simulación VHDL



La herramienta ***Foundation Series*** solo contiene un simulador a nivel de compuertas o de netlist, no VHDL. Por lo que se requerirá emplear un simulador VHDL de una tercera parte para poder simular el código RTL y el VHDL estructural o de netlist con los test bench VHDL. Se requerirán también los modelos simulables VHDL de los varios bloques de construcción Xilinx específicos usados en los diseños.

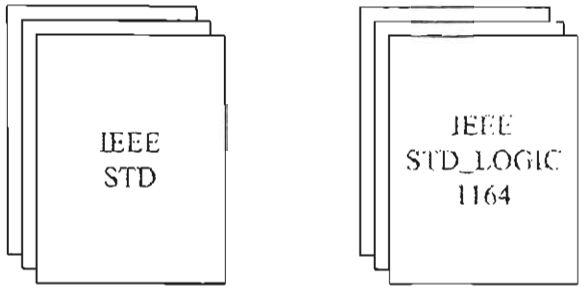
9.7.0 Soporte a la Simulacion para Xilinx

Soporte a la Simulación para Xilinx

Ahora discutiremos las clases de paquetes VHDL y las librerías que provee Xilinx junto con las herramientas ***Foundation Series Software*** para dar soporte a los flujos de diseño basados en VHDL.

9.7.1 Soporte a la Simulacion desde Xilinx

Paquetes VHDL estándar



Los diseños necesitan referenciar los paquetes estándar IEEE: *STD* y *STD_LOGIC_1164*. Esos paquetes son totalmente soportados para la síntesis por las herramientas ***Foundation***, e igual soportados directamente por todos los simuladores VHDL desde las terceras partes.

9.7.2 Soporte a la Simulación desde Xilinx

Librerías específicas de Xilinx

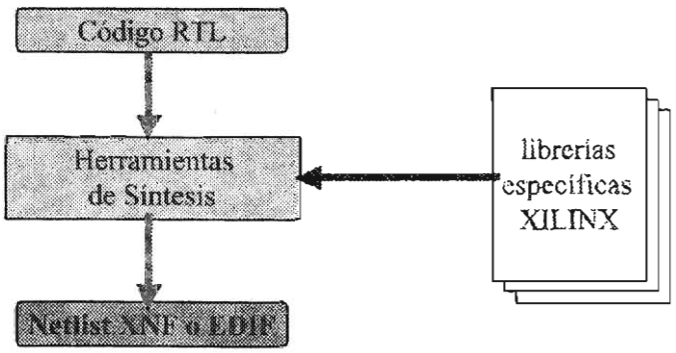
Posteriormente, siguiendo el flujo de diseño, es deseable simular la descripción VHDL a nivel compuerta o de *netlist* resultado de la síntesis, y para poder hacer eso, necesitamos una descripción VHDL de cada uno de los elementos a nivel compuerta.

9.7.3 Soporte a la Simulación desde Xilinx

Soporte 'VITAL'

Xilinx provee esas librerías de modelos VHDL a nivel de compuerta que se requieren para el simulador VHDL de las terceras partes. Las librerías son escritas conforme al formato estándar industrial VITAL. Las librerías son entregadas tanto para funciones macro como también para primitivas de nivel más bajo.

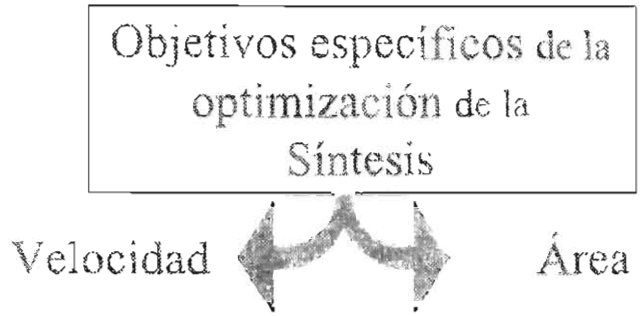
9.8.0 Síntesis



La máquina de síntesis provista con las herramientas ***Foundation Serier Software*** se llama ***Foundation Express*** o ***Base express***. Las librerías que se requieren para soportar estas máquinas sintetizadoras son entregadas e instaladas con esos productos. Además, Xilinx soporta otras herramientas de síntesis con librerías tecnológicas.

9.8.1 Síntesis

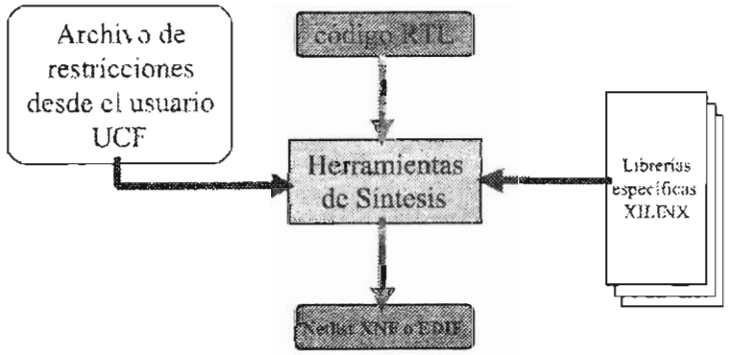
Puesta manual de las restricciones



Si se requiere alcanzar objetivos de síntesis específicos, i.e. optimización tanto para área como para velocidad, o si se requiere limitar la longitud de alguna trayectoria crítica, se pueden establecer manualmente esas restricciones usando opciones de menú en la herramienta de síntesis.

9.8.2 Síntesis

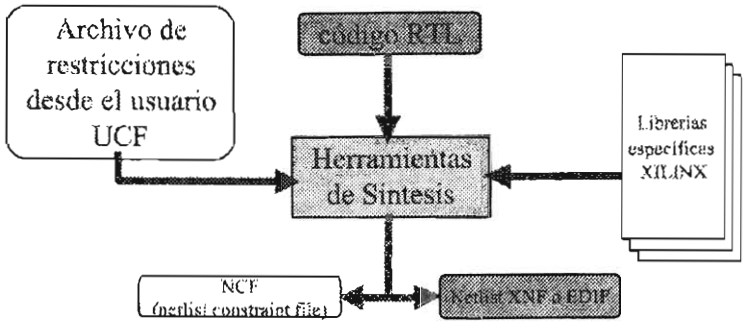
UCF- archivos de restricciones del usuario



También es posible colocar esas especificaciones de restricción en un único archivo de control que puede ser accesado consistentemente cada vez que se efectúa la Síntesis para un diseño específico.

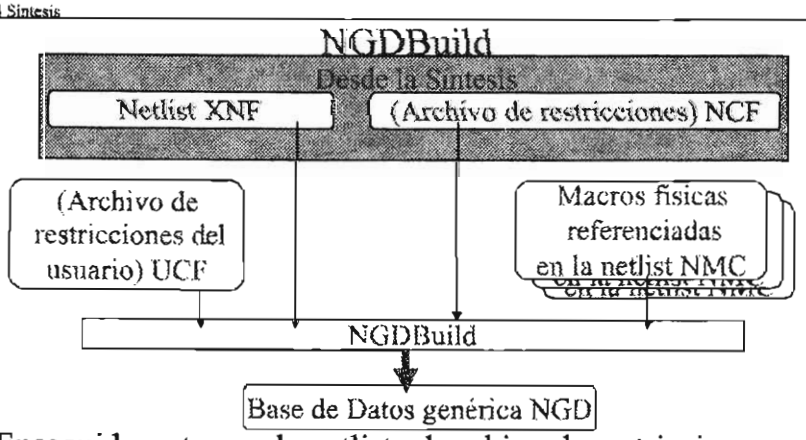
9.8.3 Síntesis

NCF- archivos de restricciones de netlist



Una vez que se ha completado la síntesis, el resultado es la creación de una *netlist* y puede opcionalmente haberse creado un archivo de restricciones conteniendo toda la información de tiempos y de restricciones especificada durante la síntesis. A éste se le conoce como el archivo de restricciones de la *netlist*, y se usa para pasar restricciones sobre el manejo de los tiempos a las herramientas de Implementación.

9.8.4 Síntesis



Enseguida se toman la netlist, el archivo de restricciones, y junto con las descripciones de las macros referenciadas en la *netlist* se usa una herramienta llamada *NGDBuild* para crear una base de datos genérica representando completamente el diseño a base de las 'primitivas Xilinx de más bajo nivel'. Esta base de datos se le conoce como la *NGD* o *Generic Database.*

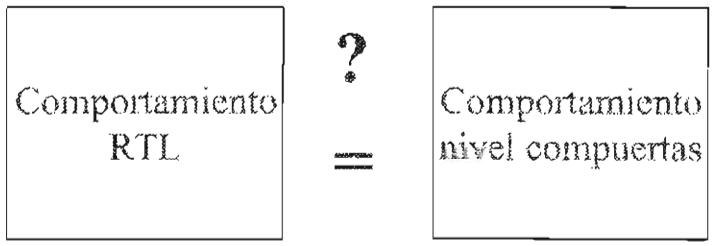
9.9.0 Simulación al nivel compuertas

Simulación al nivel compuertas

Después de la síntesis, se tiene un diseño descrito en primitivas Xilinx las cuales pueden ser simuladas con un simulador VHDL de terceras partes.

9.9.1 Simulación al nivel compuertas

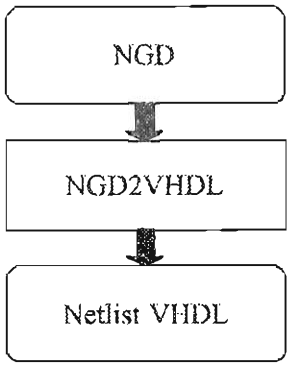
Test bench reusables



El mismo test bench usado para la simulación RTL puede aplicarse para simular la descripción al nivel compuertas. Esto asegura que ningún error funcional ha sido introducido por el proceso de síntesis, y que el estilo del código RTL permitió crear un diseño equivalente a nivel compuertas.

9.9.2 Simulación al nivel compuerta

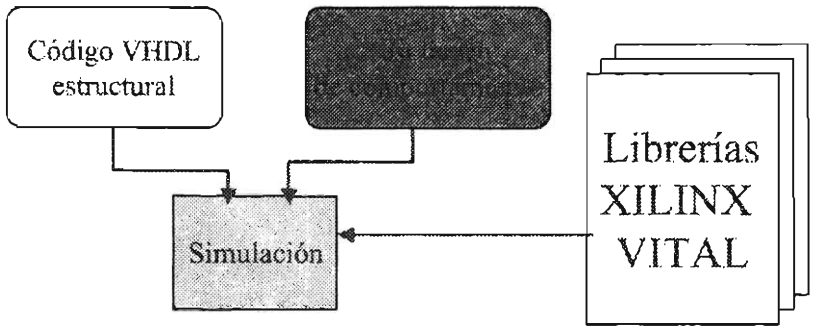
NGD2VHDL



Antes de que se pueda efectuar la simulación basada en VHDL de netlist o a nivel compuerta, se necesita usar un traductor de la netlist llamado NGD2VHDL el cual traduce el archivo NGD en una *netlist* VHDL.

9.9.3 Simulación al nivel compuertas

Puesta de las librerías VITAL



También será necesario tener las librerías VHDL basadas en VITAL para que las celdas a nivel compuerta se activen en el simulador VHDL. Una vez que eso esta heho, entonces se puede efectuar la simulación a nivel compuertas usando el *test bench* VHDL para generar los estímulos.

9.10.0 Implementacion

Implementación

9.10.1 Implementación

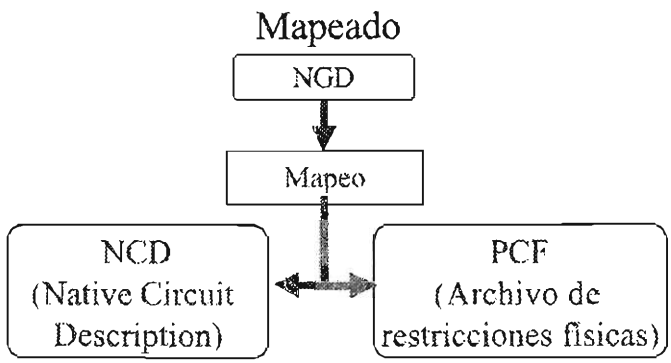
NGD es independiente de la familia

NGD
(Native Generic Database)

Cualquier familia XILINX

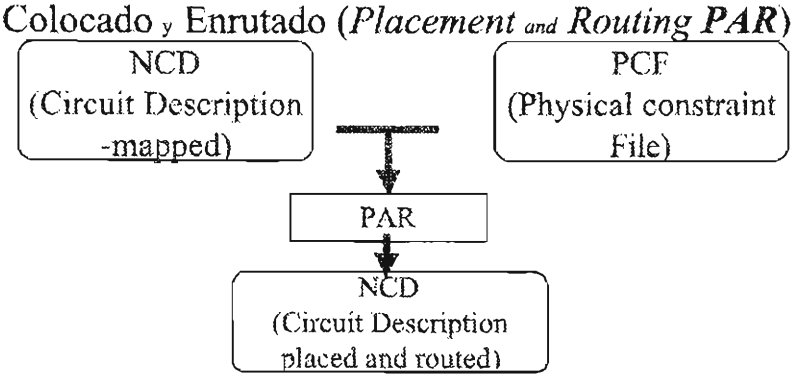
El archivo NGD resultado de la síntesis es una *netlist* (lista de partes interconectadas) de 'primitivas Xilinx' las cuales en este punto podrían ser mapeadas en cualquier familia de dispositivos. Debido a que los recursos disponibles difieren entre las diferentes tecnologías de Xilinx, el programa MAP escoge diferentes implementaciones, dependiendo sobre cual dispositivo va a ser usado.

9.10.2 Implementacion



El software de mapeo distribuye el diseño sintetizado a través de los bloques de construcción disponibles en la tecnología destino, y genera el archivo (***NCD file***) de descripción nativa del circuito (*Native Circuit Description*). También genera otro archivo (*PCF file*) de restricciones físicas (*Physical Constrains*) el cual es usado en el siguiente paso, que es el colocado y enrutado (PAR).

9.10.3 Implementación



El PAR o colocado y enrutado se efectúa como un sólo paso. Colocado incluye la asignación de los CLBs en sus respectivas localidades físicas, y el enrutado es el procedimiento para establecer la correcta conectividad entre ellos. El control del PAR se efectúa especificando restricciones durante la entrada del diseño o en los archivos de control.

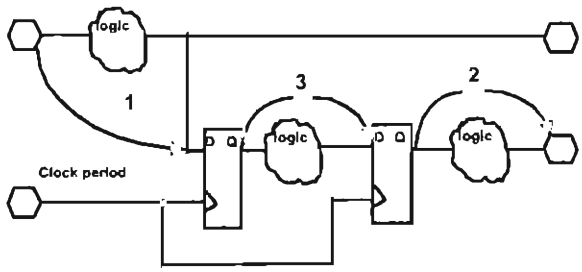
9.11.0 Revisión verificando (*Back Annotation*)

Revisión verificando (*Back Annotation*)

Una vez que el diseño esta colocado y enrutado, se necesita verificar que su funcionalidad no ha sido modificada por la introducción de los retardos reales impuestos por la implementación física del diseño.

9.11.1 Revisión verificando

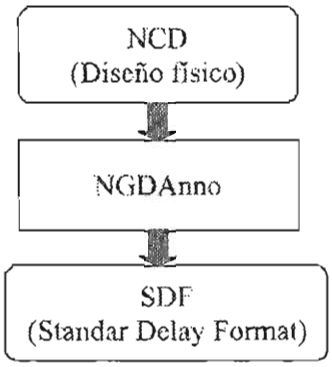
Información física



Para poder verificar los retardos del diseño del flujo de diseño, necesitamos una descripción donde la información de la implementación física sea usada para definir con mayor seguridad las características de tiempos del diseño. Este proceso es conocido como *back annotation*.

9.11.2 *Back Annotation*

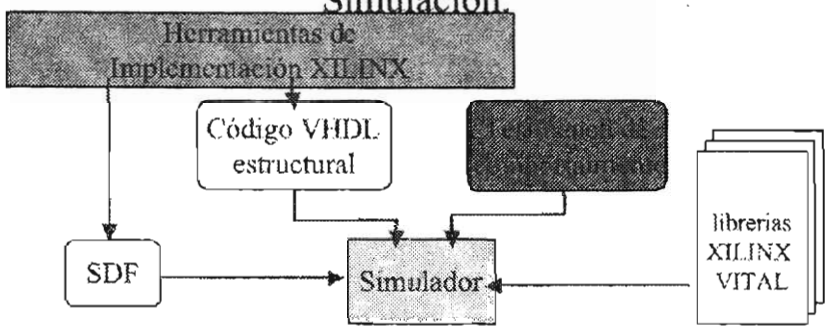
NGDAnno



Los datos de ***Back Annotation*** son generados por un programa llamado *NGDAnno*. Él extrae información de tiempos desde la *database* física y genera tanto un archivo de formato de retardos estándar SDF (*Standar Delay Format*), o anota los retardos directamente sobre la *netlist*.

9.11.3 *Back Annotation*

Retardos en el archivo SDF usados durante la Simulación.



Los simuladores VHDL que cumplen con VITAL tienen la opción de expandir la database del diseño usando datos provenientes del archivo SDF. Los retardos especificados en el archivo SDF son entonces usados en la simulación. Si se está usando el simulador ***Foundation*** nivel-compuertas, los retardos son colocados directamente en la *netlist* EDIF.

9.11.4 *Back Annotation*

Análisis de tiempos

Verificación de tiempos

Partiendo de que el simulador se ha sido inicializado correctamente, deberán tenerse mensajes de precaución y de error si se tienen violaciones al verificar los tiempos, tales como esos de mantenimiento y de puesta (*setup* y *hold times*).

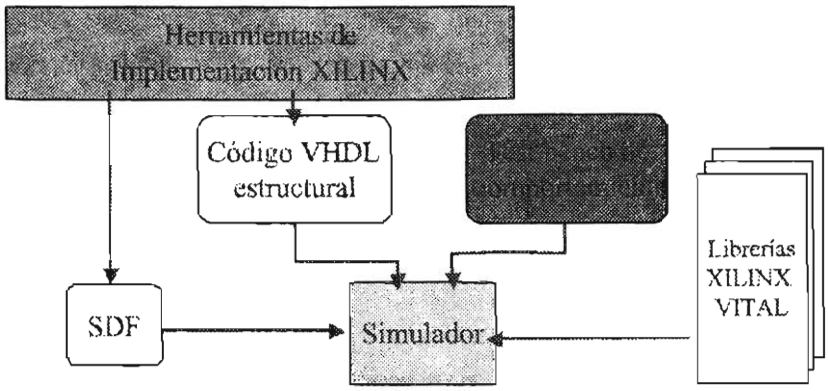
9.12.0 Verificación *Post Layout*

Verificación *Post Layout*

Hasta aquí estamos en la posición donde ya tenemos un archivo VHDL que contiene todos los retardos de la Implementación recién incluidos. Existen las vías para verificar que la funcionalidad está aún dentro de las especificaciones.

9.12.1 Verificación *post layout*

Test bench reusable



Tal como lo hemos discutido, podemos correr la simulación con el mismo *test bench* que el usado para las simulaciones RTL y *pre-layout*, y comparar los resultados.

9.12.2 Verificación *post-layout*

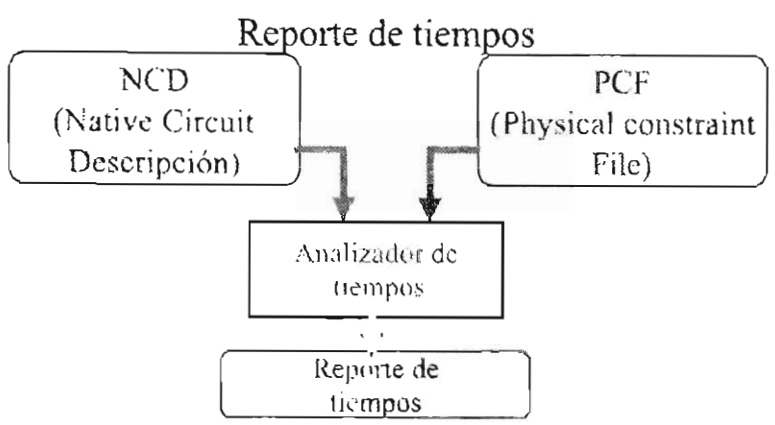
ITA Analizador de Tiempos Interactivo

Analizador de Tiempos Interactivo ITA

(*Interactive Timing Analizer*)

Podríamos en lugar de realizar la simulación de tiempos, usar el 'analizador de tiempos estático' que en las herramientas ***Foundation Series*** se le llama ITA, o simplemente 'analizador de tiempos'. Mediante el uso del ITA podemos determinar todos los retardos en las trayectorias para todas las señales cuya información requerimos desde nuestro diseño.

9.12.3 Verificación post-layout



ITA también genera un reporte enumerado sobre las violaciones de las restricciones temporales. ITA puede correrse sobre diseños sin-colocar, sobre diseños completamente colocados y enrutados, o sobre diseños que son colocados y/o enrutados con cualquier grado de terminación.

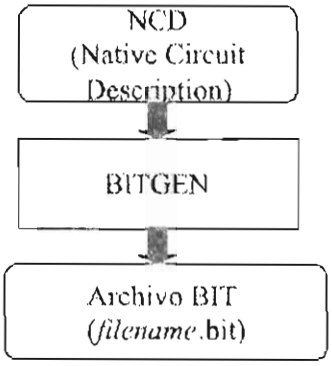
9.13.0 Programación (Programming)

Programación (Programming)

La etapa final en el flujo de diseño VHDL es programar el dispositivo seleccionado como objetivo del diseño. En este paso, necesitamos traducir la información hasta aquí generada por la máquina de Implementación en un formato adecuado para la Programación del dispositivo.

9.13.1 Programación del dispositivo

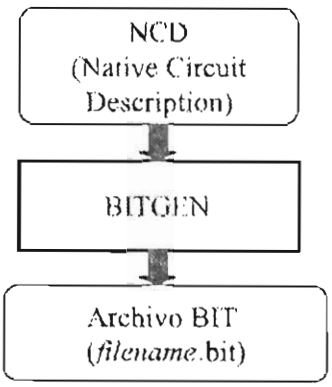
Programa *BitGen*



El programa usado para realizar esta traducción es el software *BitGen*. El parte de tomar el archivo de enrutado completo NCD (*Native Circuit Description*) como su entrada, y produce un *bitstream* (chorro o flujo de bits) de configuración, el cual es un archivo binario con una extensión **.bit*.

9.13.2 Programando

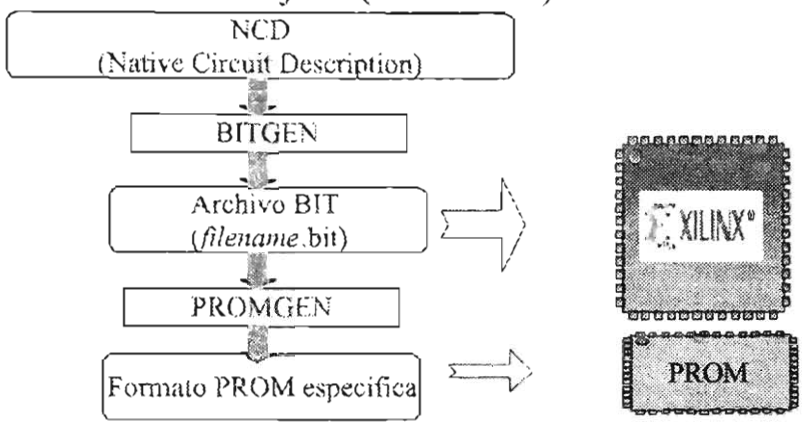
nombre_archivo.bit



Este archivo **.bit* contiene toda la información de la configuración proveniente desde el archivo NCD para definir la lógica interna e interconexiones del chip, más información específica al dispositivo proveniente desde otros archivos asociados con el dispositivo '*target*'.

9.13.3 Programación

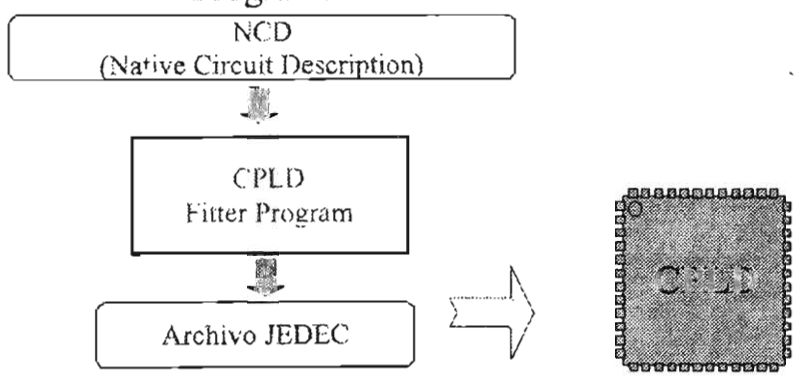
Bajado (*Download*)



Los datos binarios en el archivo **.bit* pueden bajarse directamente (*download*) al dispositivo, o bien pueden pasarse a través del programa llamado PROMGEN, el cual traduce ellos en un formato apropiado para la Programación de PROMs ...

9.13.4 Programación

Programación de los CPLD's



El flujo para programación de un CPLD es muy similar al de la programación de los FPGAs que acabamos de discutir. El archivo NGD se pasa al programa PLD *Fitter*. Esto genera entonces un archivo .JED (archivo JEDEC) el cual puede bajarse directamente al dispositivo, o leído en un programador.

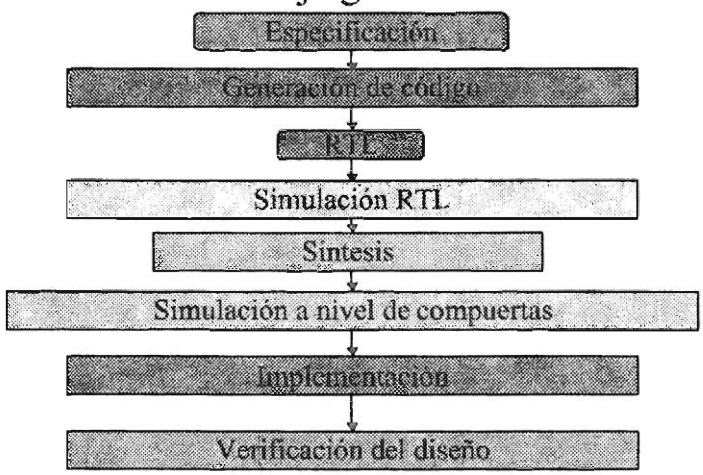
9.14.0 Resumen

Resumen

Con esto hemos completado la segunda parte de las presentaciones: VHDL para diseñar con dispositivos lógicos programables de Xilinx

9.14.1 Resumen

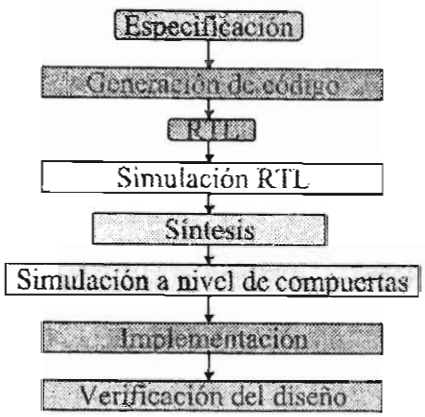
El flujo genérico



Empezamos la presentacion actual viendo cual es el flujo genérico para cualquier diseño empleando cualesquiera herramienta VHDL.

9.14.2 Resumen

Set de herramientas Foundation



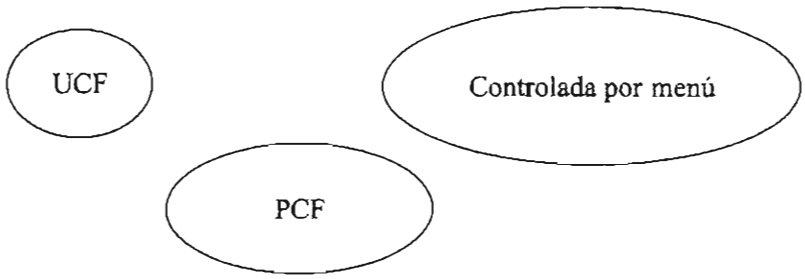
Set de herramientas Foundation Series Software

Enseguida vimos con un poco más de detalle cuales son los pasos a seguir cuando se trabaja dentro del ambiente de las herramientas Xilinx Foundation para los dispositivos FPGA's.

9.14.3 Resumen

Control y restricciones

Metodologías de restricciones



Finalmente vimos acerca de los métodos para controlar y restringir los diseños a través del flujo de diseño.

Información básica sobre los proveedores de software

10. Consideraciones sobre el uso del VHDL en el Diseño Lógico Programable

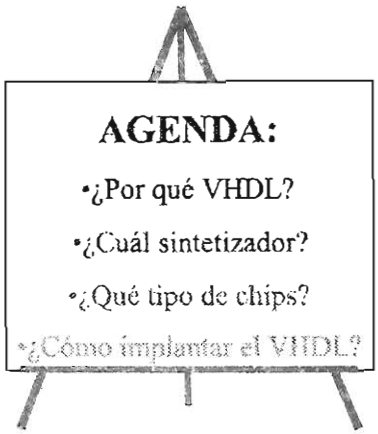
10.1.0 Introducción

VHDL
para
'Lógica Programable'
(PLDs)

Esta es la última lección de la presentación, y en ella se tratan considerandos generales relativos al uso del VHDL en el diseño digital para dispositivos de lógica programable. La lección nos ayudará a entender cómo es que el VHDL trae grandes y revolucionarios beneficios al proceso de diseño y también a ver los problemas potenciales que conlleva su implantación.

10.1.1 Introducción

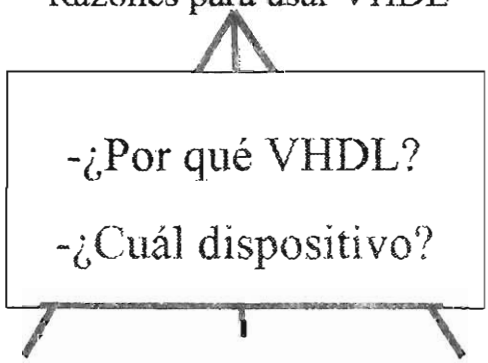
Antes de entrar en detalle



Antes de entrar en materia sobre estas cuestiones, describamos primero los tópicos que serán cubiertos en esta lección…

10.1.2 Introducción

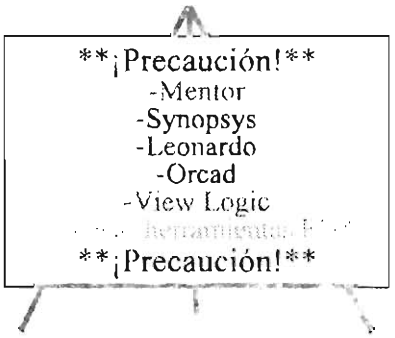
Razones para usar VHDL



Veremos primero algunas razones a considerar que justifiquen el uso del VHDL. ¿Qué beneficios trae? Además ya que el número de productos de lógica programable para diseñar con VHDL está creciendo con gran dinamismo, por lo que discutiremos algunos factores que afectan la decisión al seleccionar alguno de ellos.

10.1.3 Introducción

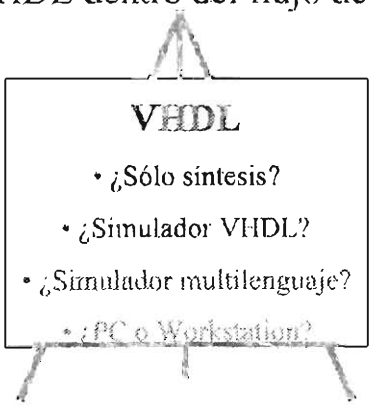
Consideraciones



Enseguida analizaremos con algún detalle cuestiones por sopesar sobre las diferentes herramientas de diseño VHDL desde distintos proveedores para asegurar que la adquisición de software se ajuste a los requerimirentos del tipo de diseños a realizar a mediano plazo.

10.1.4 Introducción

El VHDL dentro del flujo de diseño



Por último, veremos las diferentes formas mediante las cuales se puede implantar el VHDL en el flujo de diseño actual que se tenga, así como la inversión típica requerida en cada uno de los casos.

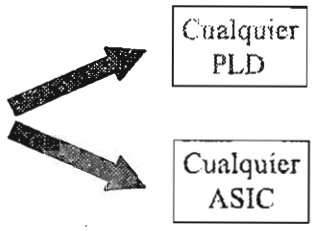
10.2.0 ¿Por qué usar el VHDL?

Algunas razones por las cuales es conveniente el uso del VHDL en los diseños...

10.2.1 ¿Por qué usar VHDL?

Independencia de la implementación

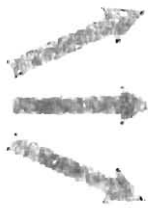
```
Process (A,B,Sel)
begin
    if  Sel ='1' then
        Z <= A;
    else
        Z <= B;
    end if;
end process;
```



Una de las razones por las que el uso del VHDL es conveniente en la descripción de los diseños es porque dicha descripción resulta tecnológicamente independiente de la implementación. Aunque ello aún es posible en lenguajes tales como el ABEL, el VHDL es un lenguaje mucho más poderoso, el cual puede emplearse para todos los dispositivos de lógica programable y las tecnologías ASIC.

Independencia del proveedor

```
Process (A,B,Sel)
begin
    if Sel ='1' then
        Z <= A;
    else
        Z <= B;
    end if;
end process;
```



Herramienta A

Herramienta B

Herramienta C

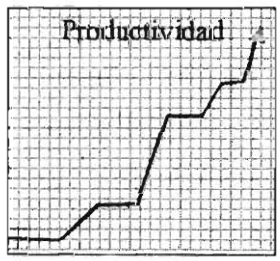
El VHDL actualmente es soportado por la mayoría de los proveedores de herramientas, con lo cual una descripción VHDL es por lo general portable entre diferentes conjuntos de herramientas, dando una gran flexibilidad y más opciones de selección.

Productividad a través de la síntesis

VHDL

Síntesis

Compuertas

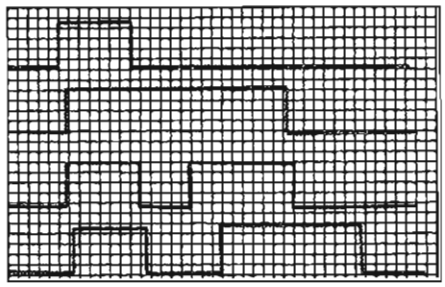


En la actualidad las herramientas de síntesis permiten que un diseño al nivel de compuertas pueda ser construido y optimizado automáticamente a partir de una descripción VHDL. Muchos usuarios experimentados en el uso del VHDL han encontrado que la síntesis VHDL puede incrementar la productividad total en órdenes de dos a cinco veces para los grandes diseños.

10.2.4 ¿Por qué usar VHDL?

El VHDL puede ser simulado

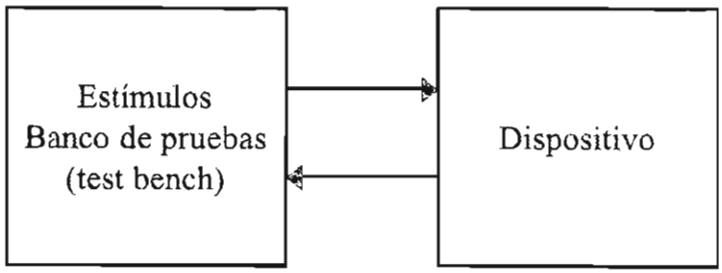
```
Process (A,B,Sel)
begin
    if  Sel ='1' then
        Z <= A;
    else
        Z <= B;
    end if;
end process;
```



Una descripción en VHDL puede ser simulada, lo que no es el caso para lenguajes tales como el ABEL, Palasm,.. Esto es muy importante, ya que trae muchas ventajas: veamos rápidamente algunas de ellas…

10.3.1.- Ventajas de la simulación

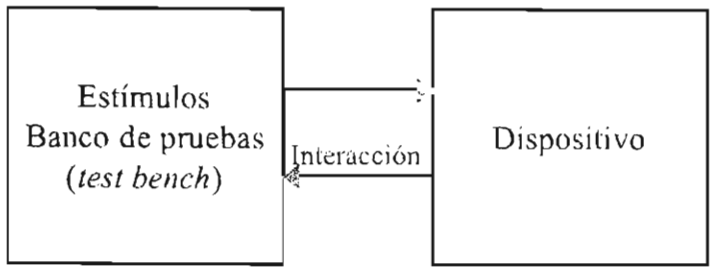
VHDL para estímulos



Una simulación basada en el VHDL por lo común usa el lenguaje VHDL para describir los estímulos, al igual que los dispositivos de diseño. El código que define los estímulos es generalmente llamado el banco de pruebas (test bench), y el poseer una descripción de estímulos portable da una gran flexibilidad a los diseñadores.

10.3.2.- Ventajas de la simulación

Ventajas de los bancos de pruebas (*test bench*)



Hay gran cantidad de ventajas de los test bench. Pueden crearse fácilmente estímulos con alto grado de complejidad, permitiendo interaccionar a dicho banco de pruebas con el modelo del dispositivo, lo que en los simuladores a nivel de compuertas tradicionales no se puede hacer.

10.3.3.- Ventajas de la simulación

Beneficios de los *test bench*

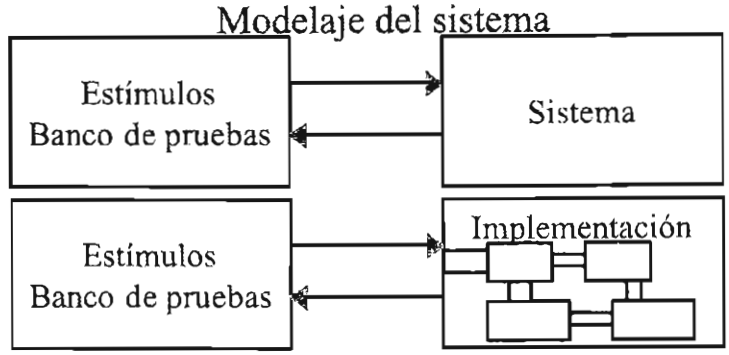
Simulación temprana

Menos iteraciones

Menor tiempo de desarrollo

Mediante el banco de pruebas la interacción entre los distintos dispositivos sobre la tablilla de diseño pueden ser simuladas desde las primeras etapas del diseño, dando por resultado la reducción de las iteraciones al nivel de la tablilla, y por ende una drástica reducción en el tiempo de desarrollo total de los diseños.

10.3.4 .-Ventajas de la simulación



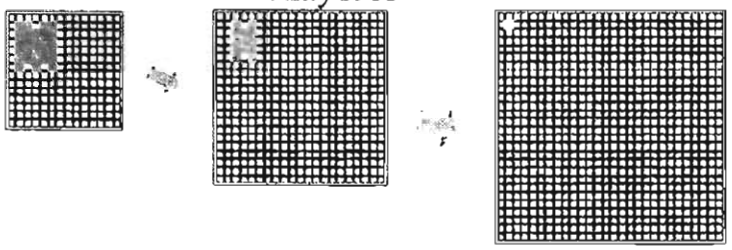
Algunas compañías usan el VHDL para modelar sus sistemas, aún antes del particionado para la implementación. Esto puede ayudar a clarificar las especificaciones desde una etapa temprana del diseño, proveyéndose tanto de una definición precisa del particionado, como de una fuente de estímulos para mayor detalle de los modelos de implementación.

10.4.0.- Ventajas término medio del uso del VHDL

En este apartado sobre las ventajas del VHDL, veremos por último algunas consideraciones, las cuales pueden traer a mediano plazo mayores beneficios a las organizaciones y a sus procesos de diseño digital..

10.4.1.- Ventajas termino medio del uso del VHDL

Densidades de lógica programable cada vez mayores



Desde finales de los 90s estamos siendo testigos de la aparición de dispositivos de lógica programable con decenas, centenas de miles y hoy día millones de '*gates*' usables. La experiencia de antiguos usuarios del VHDL ha mostrado que para sacar ventajas de ello en los diseños de gran complejidad el uso de la síntesis VHDL es de vital importancia ...

10.4.2.- Ventajas termino medio al usar VHDL

El VHDL es un estándar

Más ingenieros

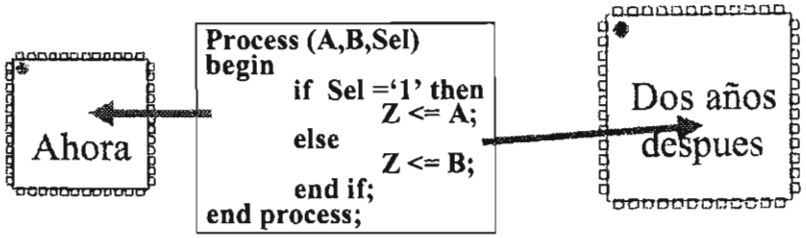
Otros grupos de trabajo

Inversión segura

El VHDL es un lenguaje estándar. Un número cada vez mayor de ingenieros se las tienen que ver con él, y todo mundo lo entiende y sabe exactamente lo que cada construcción significa. El VHDL esta ahora lo suficientemente establecido de modo que la inversión y el esfuerzo de manejarlo es algo seguro. Su desarrollo está igualmente garantizado.

10.4.3.- Ventajas término medio al usar el VHDL.

Reuso de diseños



Es bien sabido que hasta 80% de los nuevos diseños es en realidad rediseño. Al invertir en una tecnología y herramientas de lenguaje estándar independientes y que soportan el paso del tiempo, existe la seguridad de incrementar la productividad a futuro mediante el reuso de código VHDL generado ahora.

10.4.4 Ventajas término medio al usar el VHDL

Resumen..

Resumen
- **Independencia**
- **Productividad**
- **Simulación**
- **Reuso**
- **A prueba del tiempo**

En resumen, hemos considerado las ventajas de la independencia de las herramientas y de la implementación, el incremento de productividad cuando se usa la síntesis con los grandes diseños, y la habilidad de simular al nivel de las tablillas y de sistemas, y la seguridad en la inversión a prueba del futuro mediante el reuso de los diseños actuales.

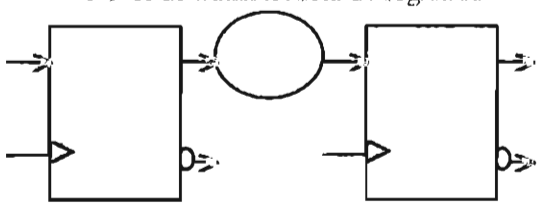
Agenda:

- Cuestiones desventajosas
- Portabilidad ¿mito o realidad?
- ¿Estilo de escritura aún de bajo nivel?
- ¿Set de construcciones sintetizables limitado?

Para balancear, dado que se han señalado las principales ventajas del uso del VHDL en el proceso de diseño lógico, señalemos algunos inconvenientes actuales que puede haber...

Descripción RTL para síntesis

Nivel de transferencia de registros

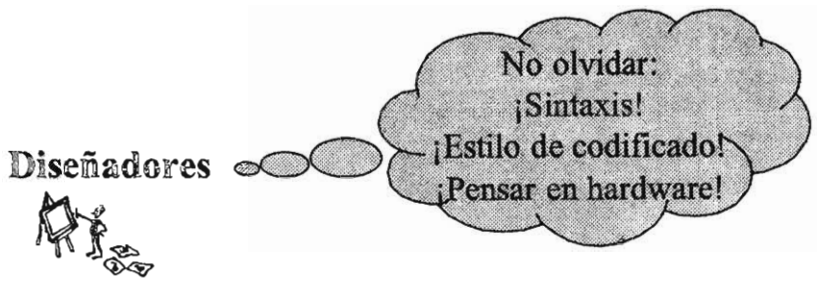


Tan solo un subconjunto del VHDL

Aún es necesario escribir el VHDL enfocado a la síntesis en un estilo, el cual describe cada uno de los registros de que consta el diseño así como de la lógica combinacional entre ellos. Esto es lo que se conoce como una descripción al nivel de transferencia de registros (RTL), y es tan solo un subconjunto de la definición completa del lenguaje.

10.5.2 Las desventajas

El estilo de codificado afecta el resultado de la síntesis



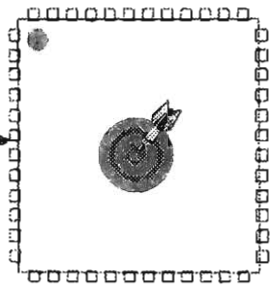
El estilo en el cual se escribe el código VHDL tiene una influencia directa en la calidad del diseño resultante al sintetizar. Los diseñadores VHDL necesitan por lo tanto conocer la sintaxis del lenguaje y el como los diferentes estilos de codificado afectan los resultados de la síntesis antes de que inicien su primer proyecto.

10.5.3 Las desventajas

El código puede ser tecnológicamente específico

```
Process (A,B,Sel)
begin
        if  Sel ='1' then
                Z <= A;
        else
                Z <= B;
        end if;
end process;
```

Tecnológicamente específico



Las tecnologías de lógica programable por lo usual poseen algunas características especificas en sus arquitecturas las cuales permiten realizar diseños más pequeños o más rápidos. Por ello puede ser conveniente hacer parte del código específico a la tecnología que se trate para poder sacar provecho de dichas características.

10.5.4 Las desventajas

Hacer un plan para estos temas

Entrenarse en:

- **Sintaxis**
- **Estilo RTL**
- **Como es que el código afecta el hardware**

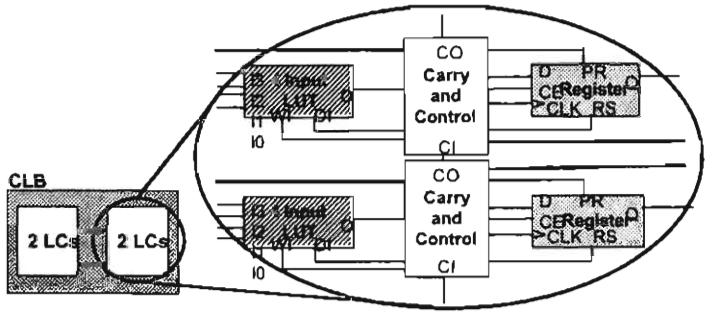
En resumen no hay razones de suficiente peso, ni aún las de los estilos de codificado que disuadan de usar el VHDL para síntesis, aunque hay que planear las cosas antes de iniciar un proyecto.

10.6.0 Cuestiones sobre la calidad de los resultados

¡Cuidado con la habilidad para sintetizar!

Probablemente en la actualidad la principal cuestión a considerar con las herramientas de síntesis es su habilidad para sintetizar eficientemente un diseño en lógica programable. A mayor calidad del sintetizador los diseños ocuparan menos área y correrán a la mayor velocidad del reloj posible.

Arquitecturas de lógica programable

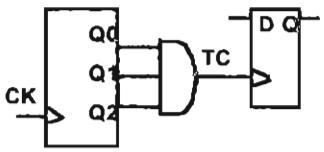


Las diferentes arquitecturas de la lógica programable se caracterizan por los diferentes estilos de sus celdas lógicas. A modo de alcanzar una implementación eficiente para una familia de tecnología específica, la arquitectura de esas celdas lógicas debe de tomarse en consideración por la herramienta de síntesis.

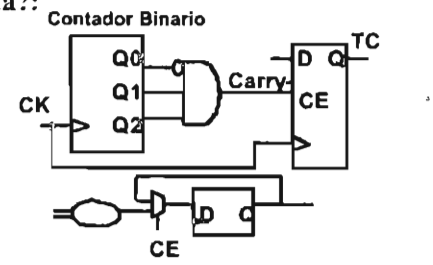
Grandes celdas lógicas

¿Qué resultado da la herramienta?:

Muy mal diseño:



Muy buen diseño:



Las celdas lógicas pueden ser completamente grandes y por ello más difícil para la herramienta de síntesis el colocarla optimamente dentro de un diseño. Si las celdas lógicas no son usadas eficientemente puede resultar una implementación más grande y lenta. Diferentes herramientas pueden dar resultados muy diferentes, siendo esta cuestión un punto a ser evaluado cuidadosamente.

10.6.1 Cuestiones sobre la calidad de los resultados.

Intervención manual a veces

Un diseñador experto

¡Diseños mejor optimizados!

Software para síntesis

¡Beneficio de la productividad!

Aún las herramientas de síntesis más sofisticadas no son siempre capaces de optimizar un diseño del mismo modo a como lo hace un diseñador experto, de modo que es loable esperar que los designers efectúen una pequeña cantidad de trabajo manual de vez en cuando. No obstante, los sintetizadores ofrecen por sí mismos significantes beneficios sobre todo para los grandes diseños.

10.7.0 Cuestiones relativas a la *datapath*

¡Cuidado con la trayectoria de datos!

En los diseños orientados a *datapath*, es decir donde el particionado del sistema se resuelve en grandes unidades de procesamiento de datos por un lado y de control por el otro, pueden ocurrir otro tipo de cuestiones. Veamos cuales son ...

10.7.1 Cuestiones relativas a la *datapath*

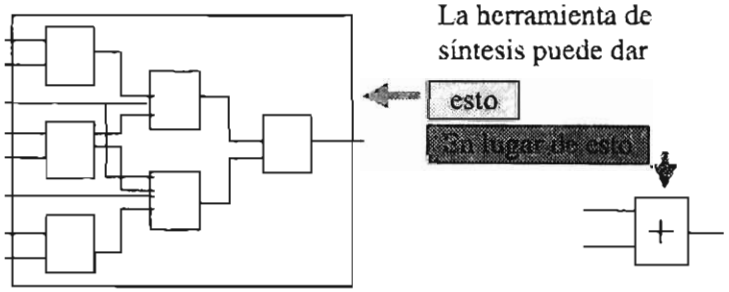
Operadores aritméticos y relacionales

Menor que `<`	Menor o igual `<=`	Igual a `=`
Mayor que `>`	Multiplicación `*`	Mayor o igual `>=`
Diferente a `/=`	Suma `+`	Resta `-`

Las partes de un diseño orientadas a *datapath* contienen muchas estructuras regulares tales como elementos aritméticos y de comparación. A menudo las librerías de los proveedores de silicio o de los PLDs contienen macroceldas de esos elementos las cuales pueden ser construidas eficientemente durante el *device layout* o por los bloques de lógica configurables CLBs.

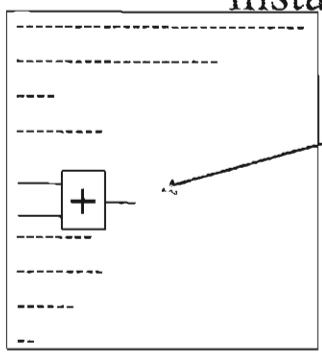
10.7.2 Cuestiones relativas a la datapath

Las macroceldas no pueden ser usadas eficientemente



El beneficio de usar VHDL es que operaciones tales como la suma pueden describirse mediante un solo postulado (vgr. Z <=A+B). Puede suceder que la herramienta de síntesis pueda en este caso construir el sumador desde muchas compuertas primitivas, o que no seleccione la macrocelda de un modo eficiente para construirlo.

instanciación



Código dependiente del proveedor

Más difícil de escribir

A esas cuestiones se les puede dar la vuelta haciendo instanciaciones de las macroceldas específicas del proveedor dentro del código VHDL. Sin embargo, esto hace que el código específico dependa del proveedor y por ende no sea portable, y normalmente más difícil de escribir...

bien, démonos por enterados ...

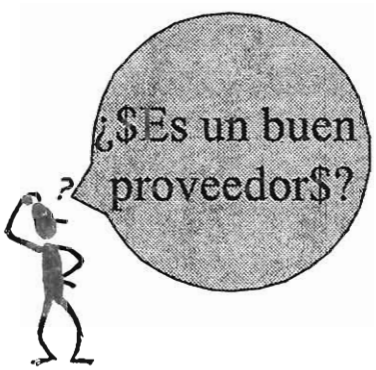
bueno, no todo es como quisiéramos en esta vida...

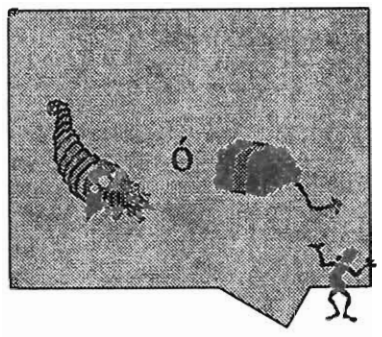
Real engineers design with Xilinx

... en conclusión, la mayoría de las cuestiones que se han discutido al último no aseguran que un diseño sea sintetizado todo lo eficiente que se quiere con solo apretar botones. Se puede ser descuidado en el estilo del código VHDL para ASICs, lo que no es cierto para los PLDs ...

10.8.2 Cosas por considerar: conclusiones

Preguntar al proveedor de la lógica programable





Cuando se planea adquirir una herramienta de síntesis para FPGAs, nos debemos aconsejar entonces con el proveedor de la lógica programable, los cuales deben ser capaces de prevenirnos acerca de las cosas con las que nos debemos enfrentar al usar herramientas específicas.

10.9.0 Integrando el VHDL en el flujo de diseño

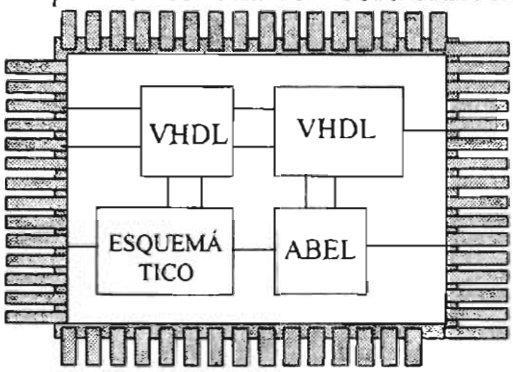
AGENDA:

- **Integración al FLUJO de DISEÑO**
- **¿A partir de diseño esquemático?**
- **¿Se emplea actualmente algún HDL?**

Veamos las diferentes vías mediante las cuales el VHDL puede ser integrado en el flujo de diseño propio y que conjuntos de herramientas y que desembolsos iniciales deberán ser tomados en cuenta.

10.9.1 Integrando el VHDL en el flujo de diseño

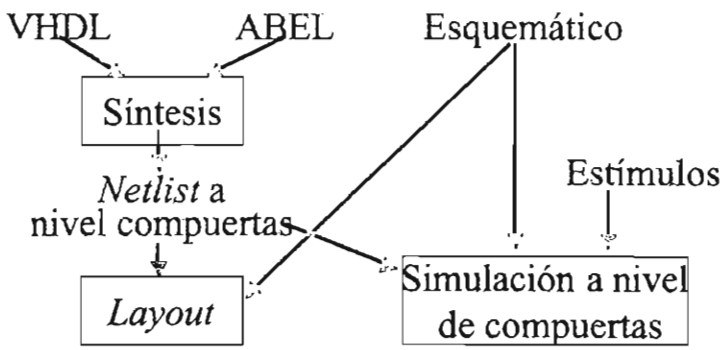
Uso semejante al ABEL o al Palasm
Un primer escenario: 'sólo síntesis'



No es necesario basar todo el diseño para lógica programable en el VHDL. Es posible usar ABEL, Palasm o bien trazado esquemático en las partes del diseño que así se considere conveniente.

10.9.2 Ajustando el VHDL en el flujo de diseño

No hay simulación de VHDL



En este escenario no hay simulación del código VHDL en si. En su lugar la simulación corre al nivel de la *netlist* de las compuertas sintetizadas, probablemente usando la misma herramienta y el mismo método para la creación de estímulos.

10.9.3 Ajustando el VHDL dentro del flujo de diseño

herramientas de síntesis y costos

El más bajo costo

Síntesis basada en PC

Este es el punto de entrada para integrar el VHDL al flujo de diseño con el costo más bajo, requiriendo una herramienta de 'sólo síntesis' VHDL basada en PC. Todas las otras herramientas usadas no necesitan cambiar, y los diseños existentes pueden integrarse dentro del nuevo flujo de diseño tal como hasta aquí existen.

10.9.4 Ajustando el VHDL al flujo de diseño

Cuestiones

Bajo costo,
Bajo riesgo

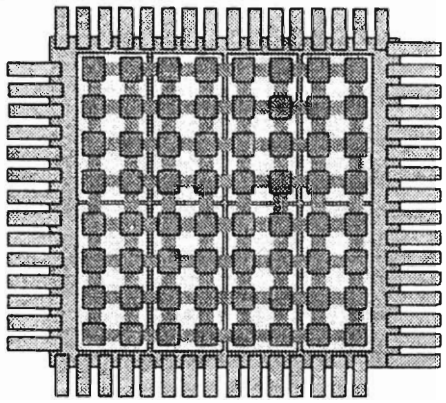
Introducción natural
al
VHDL

Diseños
pequeños
(10-20K '*gates*')

Este método tiene la ventaja de ser de bajo costo y de bajo riesgo, y da una introducción no traumática y conveniente tanto para el uso del lenguaje como a su síntesis. Sin embargo es solo adecuada para el caso de diseños pequeños y los beneficios de la simulación VHDL no se hacen evidentes.

10.10.0 Uso para un dispositivo completo

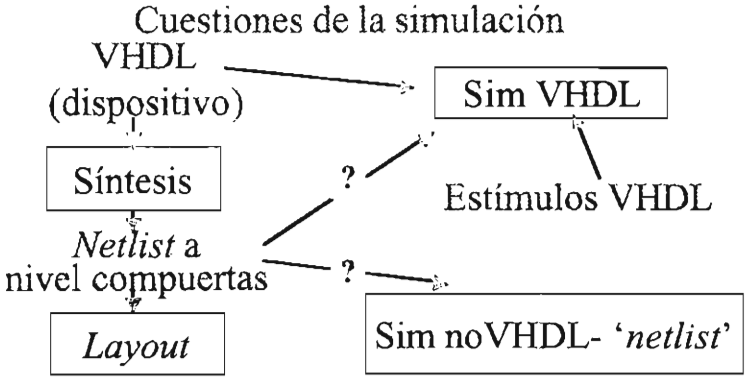
Un segundo escenario: 'simulación y síntesis'



Un segundo método de introducir el VHDL es mediante la descripción con el lenguaje de un dispositivo completo y simular primero el código RTL-VHDL, al igual que posteriormente su sintetizado al nivel de compuertas.

10.10.1 Usar para un dispositivo completo

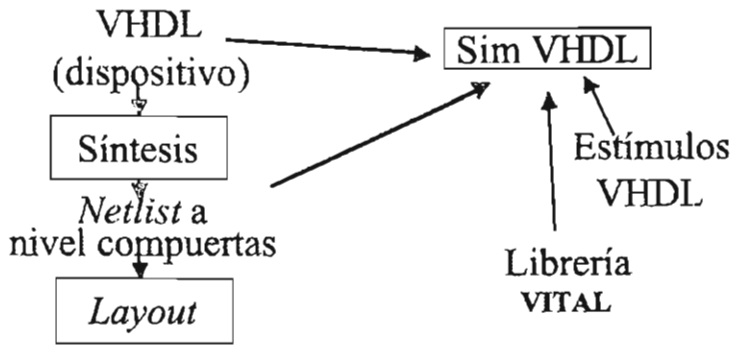
Cuestiones de la simulación



Esta técnica incluye la escritura de un banco de pruebas VHDL para el dispositivo. Idealmente se requiere simular el dispositivo sintetizado a nivel de compuertas usando los estímulos desde el banco de pruebas VHDL. ¿Esto deberá hacerse mediante el simulador VHDL o mediante uno a nivel de compuertas?

10.10.2 Usar para un dispositivo completo

Modelos al nivel de compuertas



En VHDL existe un estándar llamado VITAL para describir celdas al nivel de compuertas, y la mayoría de proveedores de lógica programable tiene disponible librerías basadas en VITAL. Debemos asegurar que las librerías de VITAL son soportadas por el simulador VHDL de *netlist*.

10.10.3 Usar para un dispositivo completo

Herramientas requeridas

Simulación

Síntesis

Esta metodología incluirá la adquisición tanto de las herramientas de síntesis como las de simulación VHDL. Actualmente las herramientas basadas en plataformas PC pueden manejar grandes diseños, requiriéndose grandes estaciones de trabajo tan sólo si los diseños manejan millones de compuertas con tiempos de simulación excesivamente grandes.

10.10.4 Usar para un dispositivo completo

Cuestiones

Simulación VHDL

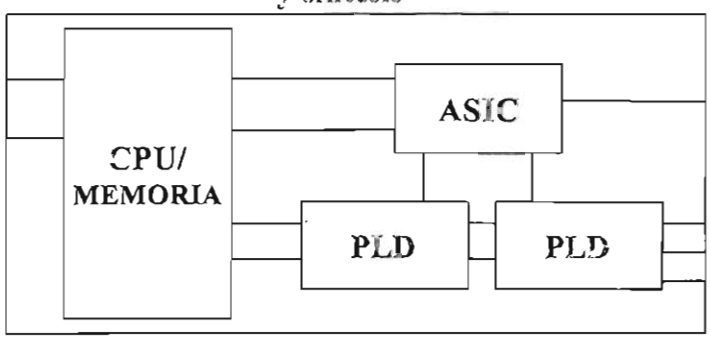
Grandes diseños

Mayor inversión

Este método tiene las ventajas de la simulación con VHDL y debe a mediano plazo ser el preferido sobre todo para cuando haya que realizar grandes diseños. La inversión inicial sera significativamente mayor que en el caso de la metodología "sólo síntesis".

10.11.0 Simulación al nivel de tablilla

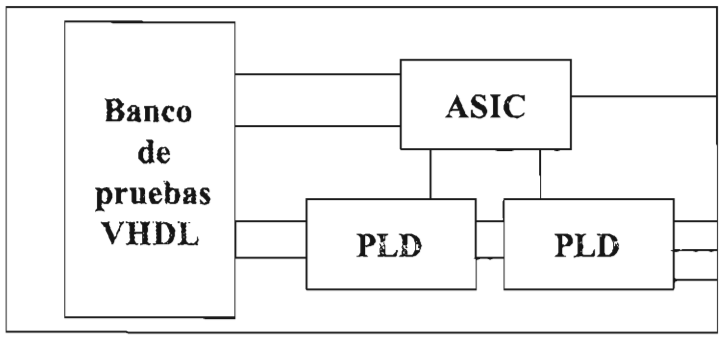
Un tercer método: Simulación VHDL -nivel tablilla y síntesis



Finalmente veremos lo que se requiere para efectuar simulación VHDL al nivel de tablillas en diseños con múltiples dispositivos de lógica programable o ASICs.

10.11.1 Simulación a nivel de tablilla

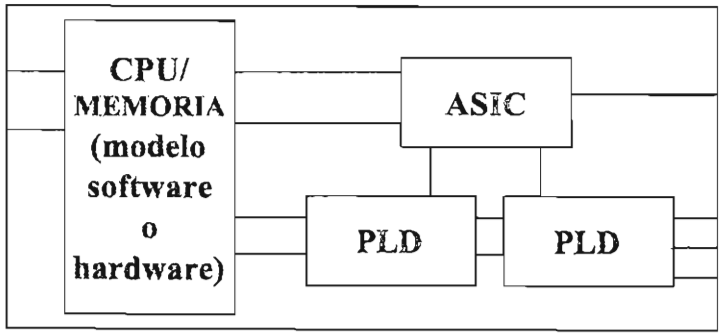
Banco de pruebas emula parte de una tablilla



Una aproximación es usar un banco de pruebas VHDL para "emular" un conjunto de componentes de la tablilla.

10.11.2 Simulación a nivel de tablilla

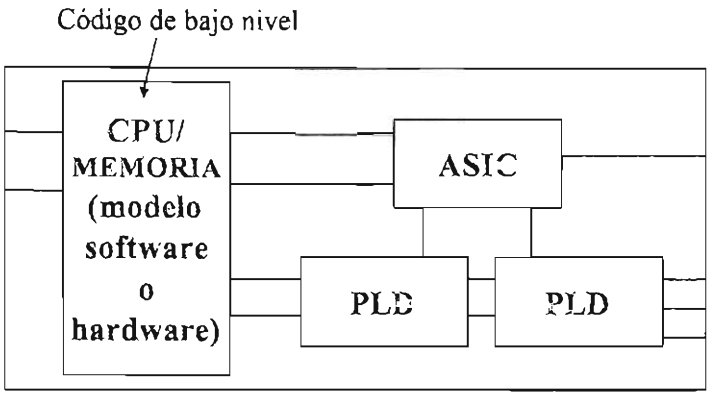
Modelado de partes estándar



Alternativamente algunas *workstation* pueden usar simuladores que pueden mezclar software no VHDL y modelos hardware de partes estándar con el código VHDL para la lógica programable y los dispositivos ASIC. Ese tipo de simuladores y modelos suelen ser demasiado costosos.

10.11.3 Simulación a nivel de tablilla

Corriendo software a bajo nivel



Algunos usuarios de VHDL actualmente ejecutan software de bajo nivel con su simulador VHDL.

10.11.4 Simulación al nivel de tablilla

Beneficios potenciales

Menor número de ciclos en el flujo de diseño a nivel tablilla

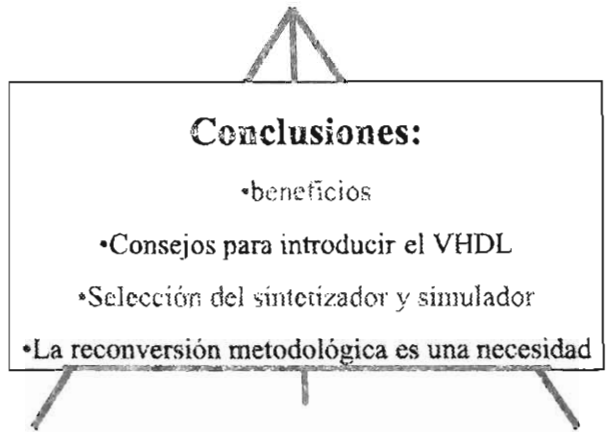
Tiempos de diseño reducido

La más grande inversión

Para resumir, el VHDL tiene la capacidad para reducir el número de ciclos en el flujo de diseño de tablillas y decrecer el tiempo de diseño mediante la síntesis para los diseños complejos basados en lógica programable.

Hay pues que invertir en herramientas, planificación y educación para estar a tono con las nuevas técnicas de diseño digital.

10.12.0 Conclusiones



Por último hagamos algunas conclusiones de los puntos que hasta ahora hemos visto..

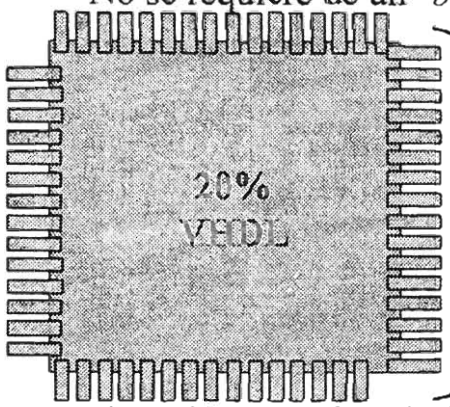
10.12.1 Conclusiones

Beneficios potenciales

Hemos considerado los beneficios del VHDL en términos de la independencia de la implementación y de la herramienta, el incremento de la productividad cuando se usa la síntesis para grandes diseños, la reducción de las iteraciones cuando se usa la simulación de sistemas y tablillas, y de la inversión a prueba del futuro mediante el reuso de los diseños.

10.12.2 Conclusiones

No se requiere de un "*big-bang*"

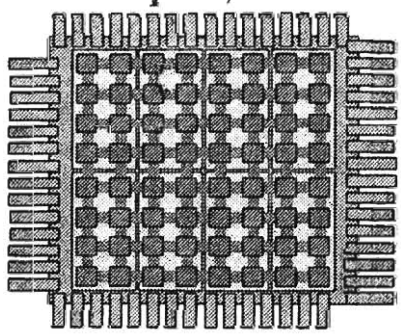


Herramienta de entrada de bajo costo

Con poca inversión se pueden adquirir herramientas de síntesis y entrenar ingenieros para usar el VHDL para diseñar partes de un dispositivo. Conforme se adquiere mayor experiencia, se puede invertir en los simuladores VHDL para aplicarse en diseños más grandes.

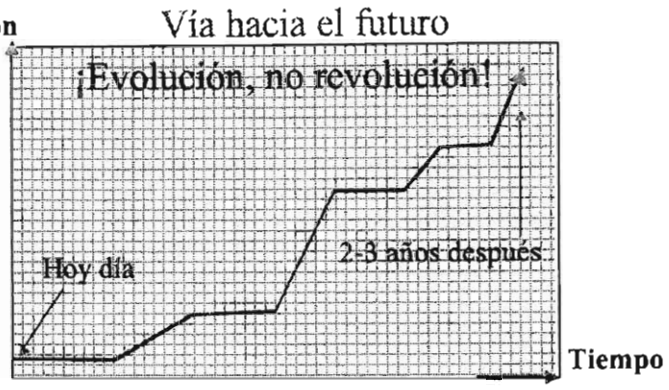
10.12.3 Conclusiones

La síntesis no es de oprimir el botón
Lo más rápido, lo más denso..,

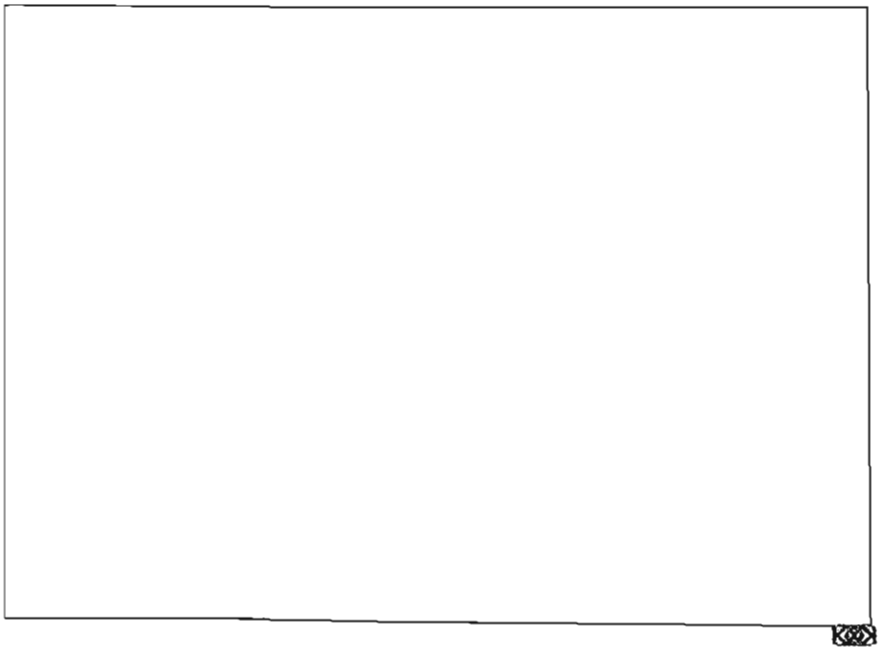


Evaluar las herramientas con sumo cuidado

Si se requiere aprovechar hasta la última compuerta y hasta el último nonasegundo del dispositivo, se deberá ser muy cuidadoso en la elección de la herramienta de síntesis. Puede requerirse especificar celdas manualmente en las partes críticas del diseño.



El VHDL es el camino hacia el futuro en el diseño de hardware. El cambio en la metodología del diseño es bastante radical. Sin embargo el costo económico puede ser gradual si se adquieren primero herramientas "solo síntesis" a modo de obtener experiencia con esta metodología de diseño. Sólo entonces se justifica mayor inversión en herramientas de simulación ....



BIBLIOGRAFÍA

1. The Practical Xilinx Designer Lab Book, Version 1.5
   David Van den Bout
   Ed. Prentice Hall

2. VHDL Modeling For Digital Design Synthesis
   Yu-Chin Hsu, Kevin F. Tsai, Jessie T. Liu and Eric S. Lin
   Ed. Kluwer Academic Publishers

3. VHDL Lenguaje Estándar de Diseño Electronico
   Lluis Terés, Yago Torroja, Serafin Olcoz y Eugenio Villar
   Ed. McGraw Hill

4. Digital Systems Design and Prototyping Using Field Programmable Logic
   Zoran Salcic and Smailagic
   Ed. Kluwer Academic Publishers

Introducción
al
lenguaje

Se terminó
de imprimir
en el mes de abril
del año 2001
en los talleres
de la Sección
de Impresión
y Reproducción de la
Universidad Autónoma Metropolitana,
Unidad Azcapotzalco.

La edición estuvo
a cargo
de la Sección
de Producción
y Distribución Editoriales.

Se imprimieron
300 ejemplares
más sobrantes
para reposición.

UNIVERSIDAD AUTONOMA METROPOLITANA
Casa abierta al tiempo Azcapotzalco

COOSEI
COORDINACIÓN DE SERVICIOS DE INFORMACIÓN

Formato de Papeleta de Vencimiento

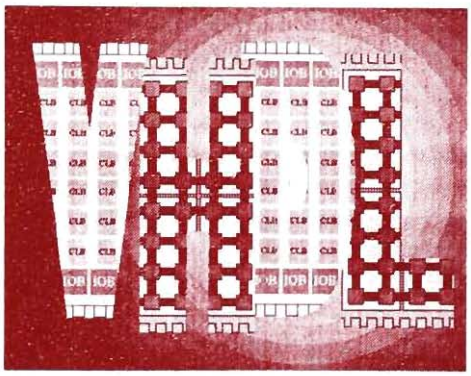
El usuario se obliga a devolver este libro en la fecha señalada en el sello mas reciente

Código de barras. 2893967

FECHA DE DEVOLUCION

- Ordenar las fechas de vencimiento de manera vertical.
- Cancelar con el sello de "DEVUELTO" la fecha de vencimiento a la entrega del libro


2893967



0092101 35298

31.00 - $ 31.00

UNIVERSIDAD AUTONOMA METROPOLITANA
Casa abierta al tiempo Azcapotzalco

Coordinación de Extensión Universitaria/Sección de Producción y Distribución Editoriales–División de Ciencias Básicas e Ingeniería/Departamento de Ciencias Básicas