ABSTRACCIONES DE UN SISTEMA DIGITAL

TOP DOWN

FUNCIONAL	Algoritmos y funciones que indican la relación E/S
ARQUITECTURAL	Componentes funcionales interconectados que definen la arquitectura
FÍSICO	Materialización a nivel eléctrico y geométrico para una determinada tecnología

BOTTOM UP

SOPORTES TECNOLÓGICOS

Puertas discretas y bloques funcionales	Familias Lógicas. Ej.: decodificadores, multiplexores, registros, contadores, puertas lógicas, …
Circuitos programables	Ej.: CPLD, FPGA, PAL,…
Circuitos para aplicaciones específicas	Ej.: ASIC (full-custom, gate-array, standard-cell), SOC,…
Circuitos Microcomputados	Ej.: MICROPROCESADORES, MICROCONTROLADORES, DSP, …

METODOLOGÍA de DISEÑO ELECTRÓNICO

Es el ordenamiento de los procesos que relacionan la complejidad y abstracción por los que atraviesa el diseño de un Sistema Electrónico, orientados a cumplimentar la *reducción de costo* y *tiempo de desarrollo*, garantizar las *prestaciones* y *confiabilidad* del producto final y lograr cierta independencia de las herramientas CAD y de las tecnologías disponibles.

En general involucra las siguientes acciones:

- Definir el nivel de abstracción inicial.
- Realizar una descomposición jerárquica definiendo nuevas abstracciones e interrelaciones entre las mismas.
- Definir la estructuración de los distintos niveles jerárquicos.
- Desarrollar la arquitectura necesaria.
- Seleccionar la tecnología.

Existen dos formas de orientar el orden de las acciones (flujo de diseño):

- BOTTON-UP
- TOP-DOWN

EVOLUCIÓN DEL DISEÑO ELECTRÓNICO

1ra ETAPA	- Desarrollos dentro de la propia fábrica - Diseño a niveles eléctricos y topográficos (Dominio FÍSICO) - Manual y fuertemente ligado a la tecnología - Bottom-up - No existen herramientas CAD
2da ETAPA	- Diseño full-custom y semi-custom - Diseño a nivel de arquitectura (Dominio ARQUITECTURAL) - Ligado a la Tecnología - Bottom-up - Desarrollo de herramientas CAD
3ra ETAPA	- Lógica programable - Diseño a nivel comportamiento (Dominio FUNCIONAL) - Poca relación con la Tecnología Aparición de los HDL - Diseño top-down - Mejoras de herramientas CAD

LENGUAJES DE DESCRIPCIÓN DE HARDWARE (HDL)

- Lenguajes de alto nivel
- Sintaxis y semántica adecuada para el modelado y descripción de circuitos electrónicos
- Permiten descripciones con distintos niveles de abstracción, precisión y estilos de modelado
- Permiten la Simulación global del circuito modelado.

NIVEL DE ABSTRACCION FUNCIONAL Relación funcional entradas-salidas	TIPO DE DATOS ABSTRACTOS	ESTILO DESCRIPTIVO ALGORÍTMICO Similar a un programa de software que indica la funcionalidad
NIVEL DE ABSTRACCIÓN ARQUITECTURAL Partición en bloques funcionales indicando variable tiempo	TIPO DE DATOS COMPUESTOS	ESTILO DESCRIPTIVO FLUJO DE DATOS Ecuaciones y expresiones que indican el flujo de información
NIVEL DE ABSTRACCIÓN LÓGICO Ecuaciones lógicas y elementos de librería (con o sin referencia a una tecnología específica)	TIPO DE DATOS BITS	ESTILO DESCRIPTIVO ESTRUCTURAL Enumeración de componentes y conexiones entre ellos.

VENTAJAS DE LOS HDL

- Si bien los HDL nacen para la descripción y simulación del hardware, su uso se ha generalizado en el diseño y síntesis del mismo.
- Permiten intercambio entendible entre distintos equipos de trabajo.
- Son de disponibilidad pública.
- Soportan descripciones con múltiples niveles de abstracción.
- No dependen de la metodología.
- Proporcionan códigos reutilizables.
- Pueden depender o no de la tecnología.

DESVENTAJAS DE LOS HDL

- Evolución lenta con muchas diferencias entre las versiones sucesivas.
- El no poseer una semántica matemática formal, la portabilidad se ve limitada.

DISEÑO EN VHDL

(Very high speed **H**ardware **D**escription **L**anguage**)**

Consideraciones:

- Las especificaciones deben ser lo más estables posibles.
- Las simulaciones parciales deben ser posibles de realizar.
- Disponer de la biblioteca del fabricante.
- El valor agregado de un circuito electrónico es su funcionalidad, por tanto el costo de diseño es importante.
- Reutilización como una meta.
- La integración de herramientas de diseño (CAD).

FLUJO DE DISEÑO EN VHDL (TOP-DOWN)

REQUISITOS	Documentación preliminar. Viabilidad técnica y económica.
ESPECIFICACIONES	Detalles funcionales del circuito y partición en bloques funcionales.
DISEÑO ARQUITECTURAL	Descripción y simulación en VHDL. De la tecnología elegida depende la biblioteca disponible.
DISEÑO LÓGICO	Síntesis del circuito. Requiere tener en cuenta las herramientas disponibles.
DISEÑO FÍSICO	Ubicación de componentes, conexionado y simulación completa.
FABRICACIÓN	Grabado del dispositivo programable (FPGA, …), Prototipo ASIC, …

VHDL

CARACTERÍSTICAS

GENERALES

- TIPO DE DATOS: Gran flexibilidad que permite descripciones con distintos niveles de abstracción.
- CONTROL DE FLUJO: Condicionales e iteraciones.
- ESTRUCTURACIÓN DEL CÓDIGO: Funciones y Procedimientos
- DESARROLLO Y UTILIZACIÓN DE BIBLIOTECAS

ORIENTADAS A LA DESCRIPCIÓN DEL HARDWARE

- MODELO DE ESTRUCTURA
- MODELO DE CONCURRENCIA
- MODELO DE TIEMPO

MODELO DE ESTRUCTURA

Elementos para describir un dispositivo:

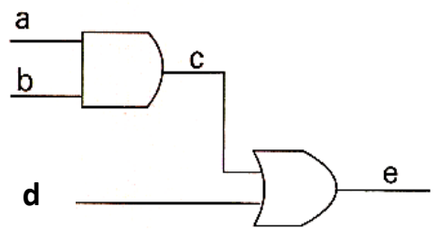
- **ENTITY**: Interfaz del dispositivo con el exterior, define que señales son accesibles desde el exterior: puertos del dispositivo (**PORT**)
- **ARCHITECTURE** : Descripción funcional del dispositivo
  El dispositivo descripto puede ser referenciado para describir dispositivos más complejos bajo el concepto **COMPONENT**.

MODELO DE CONCURRENCIA

Elementos para modelar la concurrencia

- **PROCESS**: Es una descripción del comportamiento de un componente que implica una ejecución paralela con otros procesos, pero que internamente se ejecuta en forma secuencial. Un proceso se ejecutará si una señal (SIGNAL) a la que es sensible, cambia. La ejecución se detiene al encontrar una sentencia WAIT.
- **SIGNAL**: Elemento de comunicación (sincronización) entre procesos, tienen asociadas las colas de eventos.
- **WAIT**: Sentencia que detiene la ejecución de un proceso.

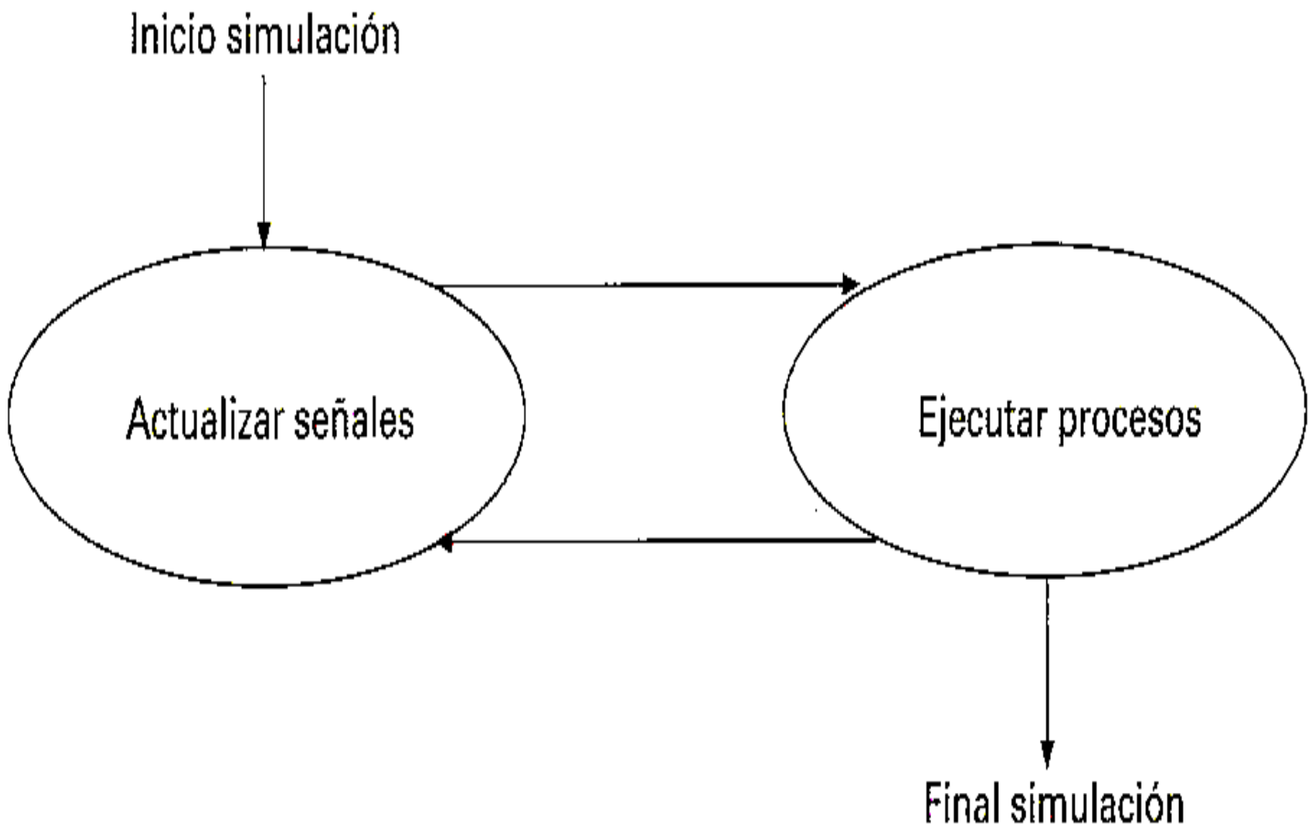
```
AND2 : process
Begin
        c < = a and b;
        wait on a, b;
end process AND2;

OR2 : process
begin
        e <= c or d;
        wait on c, d;
end process OR2;
```



MODELO DE TIEMPO

- Una de las finalidades de VHDL es observar el comportamiento del modelo en el tiempo: **SIMULACIÓN**.
- La **SIMULACIÓN** está dirigida por eventos: cambio en el valor de una **SIGNAL**.
- Ya que un evento puede generar otros eventos, una **SIMULACIÓN** finaliza cuando no existan más eventos o bien se alcance el tiempo de simulación especificado.
- Este mecanismo estímulo-respuesta implica un **CICLO** de simulación de dos etapas:
  1ra Etapa: Las señales actualizan sus valores, desde una cola de eventos.
  2da Etapa: Los procesos sensibles se ejecutan hasta un **WAIT**, colocando, quizás, nuevos valores en las señales a las cuales son sensibles.
- Los **CICLOS** se repetirán hasta tanto no existan más eventos. Esto implica un **DELTA DELAY** (retardo) desde el momento en que los procesos colocan los nuevos valores de las señales en la cola de eventos y el momento en el que las señales adoptan los nuevos valores provocando un nuevo **CICLO**. Este mecanismo permite simular hardware concurrente asegurando el determinismo.

SIMULACIÓN


Inicio simulación
Actualizar señales
Ejecutar procesos
Final simulación

UNIDADES DE DISEÑO

Consideraciones:

- Las UNIDADES de DISEÑO son construcciones VHDL que implican un proceso jerárquico en su análisis a fines de la simulación.
- Un dispositivo se representa mediante una **entidad** que define la interfaz, y una **arquitectura** que define su funcionalidad
- Se pueden definir múltiples arquitecturas para una entidad que se asociarán mediante la **configuración**.
- Los **objetos** (constantes, variables, señales, ficheros) deben declararse antes de ser usados. Para declaraciones usadas en varias unidades de diseño, se utilizan los **paquetes.**

TIPOS

UNIDADES PRIMARIAS

- **Entity declaration**: declaración de entidad
- **Configuration**: configuración
- **Package declaration**: declaración de paquete

UNIDADES SECUNDARIAS (dependen de una primaria para poder analizarlas)

- **Architecture**: arquitectura de una entidad
- **Package body**: cuerpo del paquete

DECLARACIÓN DE ENTIDAD (entity)

Define la visión externa de un dispositivo y no la implementación concreta.

Sintaxis:

```
entity identificador is
        [genéricos]
        [puertos]
        [declaraciones]
        [begin sentencias]
end [entity] [identificador];
```

Identificador: nombre de la entidad

Genéricos: conjunto de parámetros que permiten modificar la funcionalidad al referenciar la entidad.

Puertos: determinan la interfaz con el exterior, debe incluir:

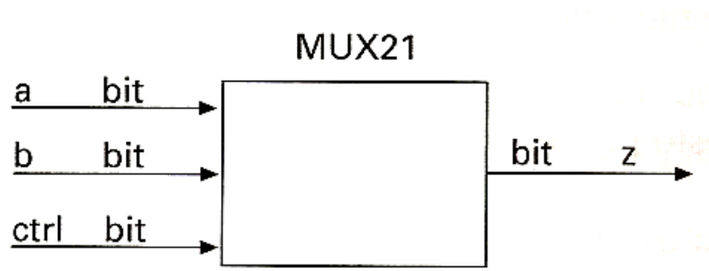
- Nombre: a fin de referenciarlo
- Tipo: clase de información que maneja el puerto
- Modo: define la dirección de la información

Declaraciones: se incluyen normalmente en la **architecture**, sirven para declarar objetos comunes a varias arquitecturas de la misma entidad.

Sentencias: pueden incluirse siempre que no afecten la funcionalidad que se describe en la **architecture.**

Ejemplo: Multiplexor de dos canales MUX21

```
entity Mux21 is
        port  ( a     : in   bit;
               b     : in   bit;

               ctrl   : in   bit;
               z      : out bit);
end ;
```



ARQUITECTURA

- Define la funcionalidad de la entidad a la cual se asocia.
- Describe un conjunto de operaciones sobre las entradas de la entidad asociada y sus salidas.

Sintaxis:

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

Identificador: Nombre de la arquitectura
Identificador-entidad: Nombre de la entidad a la que pertenece
Declaraciones: Declara los elementos necesarios para la descripción de la arquitectura.
Sentencias concurrentes: Describe la funcionalidad del dispositivo.

Dependiendo el tipo de sentencias, se puede describir según tres estilos diferentes:

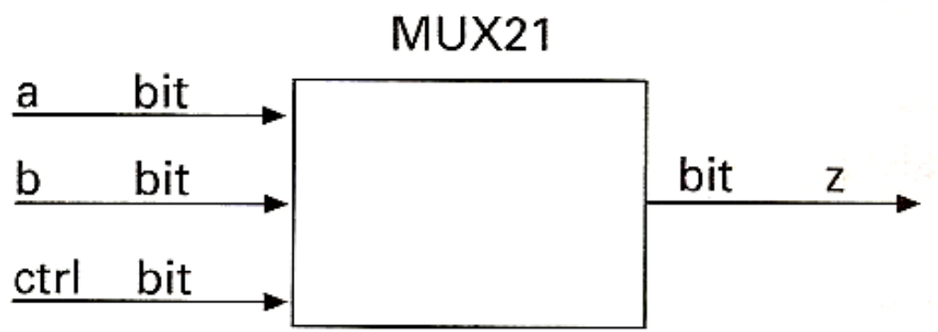
- *Algorítmico*: conjunto de instrucciones que se ejecutan secuencialmente
- *Flujo de datos*: modela la arquitectura como un flujo de datos entre los módulos que la integran.
- *Estructural*: Define la arquitectura como un conjunto de módulos interconectados.

ARQUITECTURA ESTILO *ALGORITMICO*

- Define la funcionalidad del dispositivo mediante un algoritmo que se ejecuta secuencialmente.
- No hay referencia alguna a la estructura de hardware.

Sintaxis:

```
architecture Algoritmico of Mux21 is
begin
        process (a, b, ctrl)
        begin
                if (ctrl = '0') then
                        z <= a;
                else
                        z <= b;
                end if;
        end process;
end Algoritmico;
```



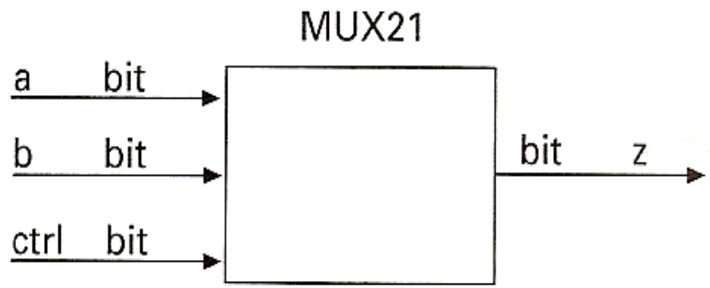
Nota: **process** es una sentencia concurrente que implica la ejecución secuencial del código después de **begin**.

ARQUITECTURA ESTILO *FLUJO DE DATOS*

- Refleja la funcionalidad del dispositivo mediante un conjunto de ecuaciones ejecutadas concurrentemente.
- Existe una correspondencia directa entre el código y la estructura del hardware.
- Esta descripción es funcional y estructural.

Sintaxis:

```
architecture FlujoDatos of Mux21 is
        signal      ctrl_n, n1, n2 : bit;
        begin
                ctrl_n <= not (ctrl) after 1 ns;
                n1 <= ctrl_n and a after 2 ns;
                n2 <= ctrl and b after 2 ns;
                z <= (n1 or n2) after 2 ns;
end FlujoDatos;
```



Nota: Se incorporan tres señales internas para definir la interconexión de los distintos operadores y se asocia un retardo mediante la cláusula **after**.

ARQUITECTURA ESTILO *ESTRUCTURAL*

- Se definen los **componentes** interconectados mediante señales.
- La descripción es estructural, no incluye funcionalidad.
- La funcionalidad de los **componentes** está incluida en la definición de la arquitectura de los mismos.
- Primero: se declaran los **componentes** que formarán la estructura indicando la interfaz de los mismos.
- Segundo: se referencian los **componentes** y se interconectan. (cada referencia debe tener un nombre distinto para diferenciarse de otras al mismo **componente**)

Sintaxis:

```
architecture Estructural of Mux21 is
        signal ctrl_n, n1, n2 : bit;
        component INV
                port   ( y : in bit;
                         z : out bit);
                end component;
        component AND2
                port   (x in bit;
                         y : in bit;
                         z : out bit);
                end component;
        component OR2
                port   (x in bit;
                         y : in bit;
                          z : out bit);
        end component;
        begin
        U0: INV port map (ctrl, ctrl_n);
        U1: AND2 port map (ctrl_n, a, n1);
        U2: AND2 port map (ctrl, b, n2);
        U3: OR2 port map (n1, n2, z);
end Estructural;
```

CONFIGURACIÓN

- Permite seleccionar la arquitectura que se desea utilizar para una entidad
- Como pueden existir más de una arquitectura por entidad, la configuración es la construcción que las relaciona.

Sintaxis:

```
configuration identificador of identificador_entidad is
        for identificador_arquitectura
                for id_componente
                        use entity id_entidad(id_arquitectura);
                end for;
                .
                .
                .

        end for;
end [configuration] [identificador] ;
```

Identificador: nombre de la configuración a fin de referenciarla
Identificador_entidad: nombre de la entidad relacionada
Identificador_arquitectura: nombre de la arquitectura relacionada
Id_entidad: Identificador de la entidad a relacionar
Id_componente: Identificador del componente a asociar a una entidad y arquitectura

- La configuración del Multiplexor Mux21 es, para el tipo descriptivo Flujo de Datos, el siguiente:

```
configuration Mux21_cfg of Mux21 is
    for FlujoDatos
    end for;
end Mux2_cfg;
```

- En el caso de un modelo jerárquico como la arquitectura estilo estructural del Mx21, donde se definen los componentes INV, AND2 y OR2, suponemos que se han definido las entidades ***INV, AND2, OR2*** y la arquitectura para esas entidades es la misma y se llama ***Algorítmico*** guardadas en una biblioteca llamada ***work***. Entonces la configuración sería:

```
configuration Mux21_cfg of Mux21 is
    for Estructural
        for U0 : INV use work.entity INV(Algoritmico);
        end for;
        for all : AND2 use work.entity
    AND2(Algoritmico);
        end for;
        for U3 : OR2 use work.entity OR2(Algoritmico);
        end for;
```

PAQUETES

- Permiten agrupar un conjunto de declaraciones para que puedan ser usadas por varios dispositivos sin ser repetidas en la descripción de cada uno de ellos.
- Dos unidades de diseño:
  Declaración (package): Visión externa de los elementos que se declaran.
  Cuerpo (package body): Implementación

La sintaxis es:

```
package identificador
        [declaraciones]
 end [package] [identificador] ;
```

```
package body identificador is
        [declaraciones cuerpo]
end [package body] [identificador] ;
```

Ejemplo de uso de PAQUETES para definir el retardo de las puertas NOT, AND2 y OR2:

```
package RetardosOp is
        constant RetNOT : time;
        constant RetAND2 : time;
        constant RetOR2 : time;
end RetardosOp;
```

```
Package body RetardosOp is
        constant RetNOT : time := 1 ns;
        constant RetAND2 : time := 2 ns;
        constant RetOR2 : time := 2 ns;
end RetardosOp;
```

Los paquetes y sus cuerpos quedan guardados en una biblioteca, para usarlos habrá que indicar el nombre de la biblioteca, del paquete y del cuerpo. Por ejemplo para usar la constante Ret-NOT declarada en el paquete RetardosOp guardado en una biblioteca llamada Biblio, debe escribirse:

Biblio.RetardosOp.RetNOT

Ejemplo de la construcción arquitectura del Mux21 con estilo flujo de datos, usándolos paquetes definidos:

```
architecture FlujoDatos of Mux21 is
        signal ctrl_n, n1, n2 : bit:
        begin
                ctrl_n <= not (ctrl) after Biblio.RetardosOp.RetNOT;
                n1 <= ctrl_n and a after BibIio.RetardosOp.RetAND2:
                n2 <= ctrl and b after BibIio.RetardosOp.RetAND2;
                z <= (n1 or n2) after BibIio.RetardosOp.RetOR2:
end FlujoDatos:
```

para evitar las repeticiones se puede usar **use** biblio.RetardosOp.RetNOT; o **use** Biblio.RetardosOp.**all**;

```
use. Biblio.RetardosOp.all
architecture FlujoDatos of Mux21 is
        signal ctrl_n, n1, n2 : bit;
        begin
        ctrl_n <= not (ctrl) after RetNOT:
        n1 <= ctrl_n and a after RetAND2:
        n2 <= ctrl and b after RetAND2:
        z <= (n1 or n2) after RetOR2:
end FlujoDatos;
```

BIBLIOTECAS

- Sirven para guardar los resultados del análisis de las unidades de diseño para su reutilización posterior.
- **work** es la biblioteca por defecto.
- El diseñador puede crear bibliotecas según su criterio.
- Existen bibliotecas ya definidas como la **std** que contiene tipos básicos de datos.
- Salvo las bibliotecas **work** y **std**, para tener acceso a una biblioteca se usa la sentencia **library.**

Por ejemplo, si se quiere usar los elementos de un paquete llamado *PaqueteEjemplo* almacenado en una biblioteca llamada *BibliotecaEjemplo*, se debería escribir:

library BibliotecaEjemplo;
use BibliotecaEjemplo.PaqueteEjemplo.**all**;