



Introducción a las FPGAs

Laboratorio 4. Sistemas Empotrados

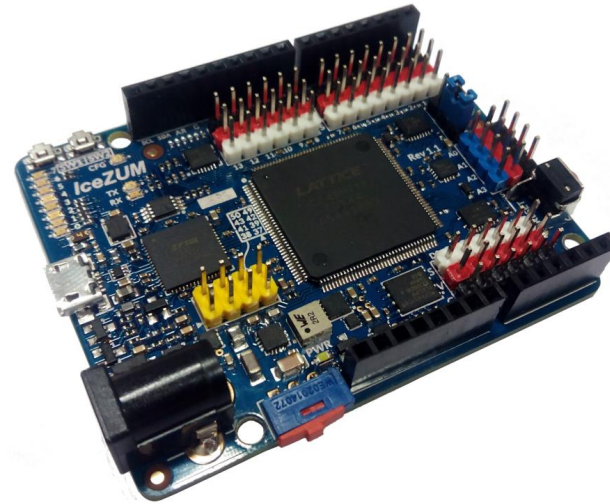
Índice

1. Introducción a la Práctica 4
2. ¿Qué es una FPGA?
3. Verilog HDL
4. Test

Índice

1. Introducció a la Pràctica 4
2. ¿Qué es una FPGA?
3. Verilog HDL
4. Test

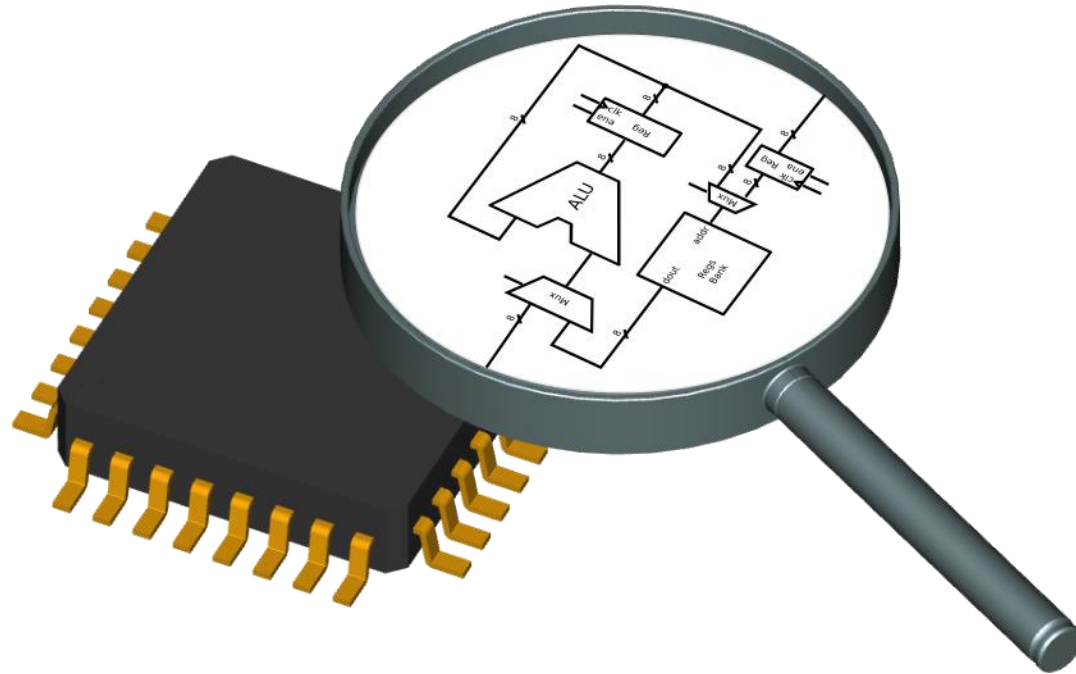
FPGAWars: Explorando el lado libre de las FPGAs



Juan González Gómez (Obijuan)

<https://github.com/Obijuan>

Viaje al interior de los chips digitales

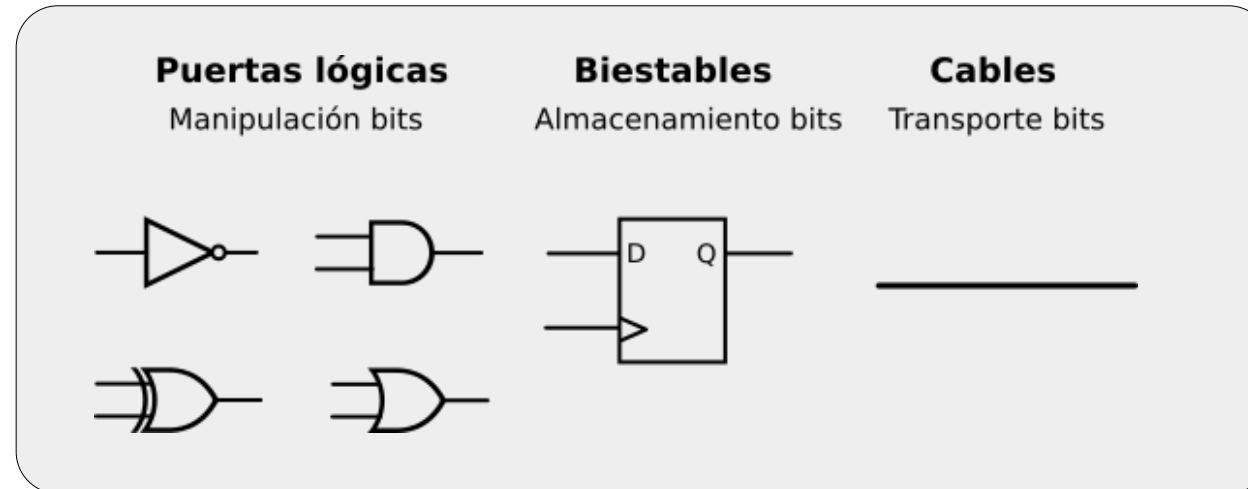


Nivel de electrónica digital

Información: Sólo 1s y 0s (Bits)

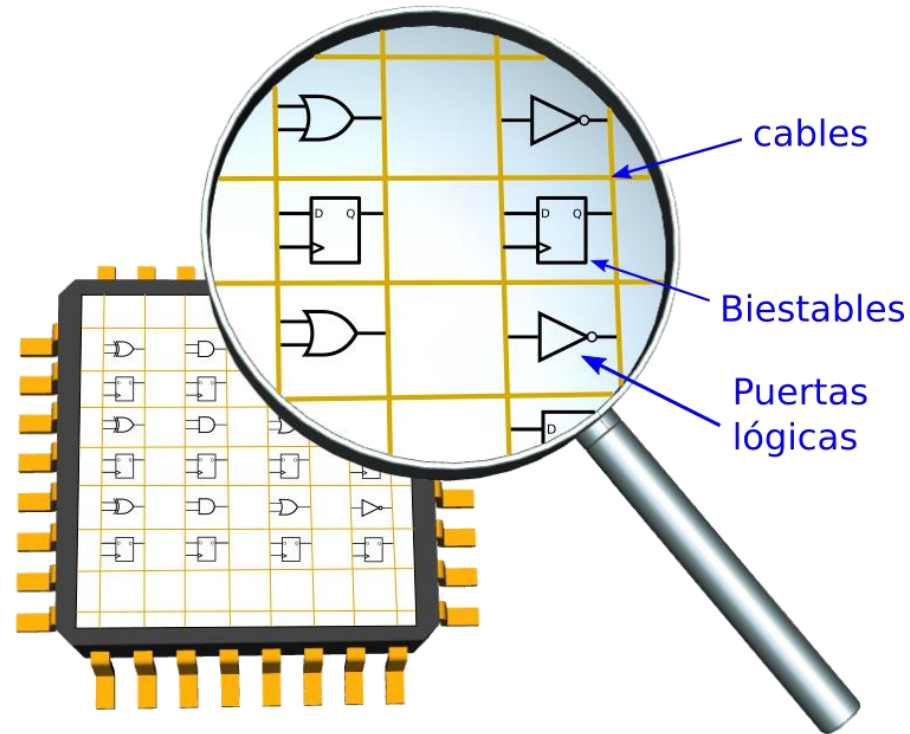
Función: **Manipular, almacenar y transportar bits**

Elementos en circuitos digitales



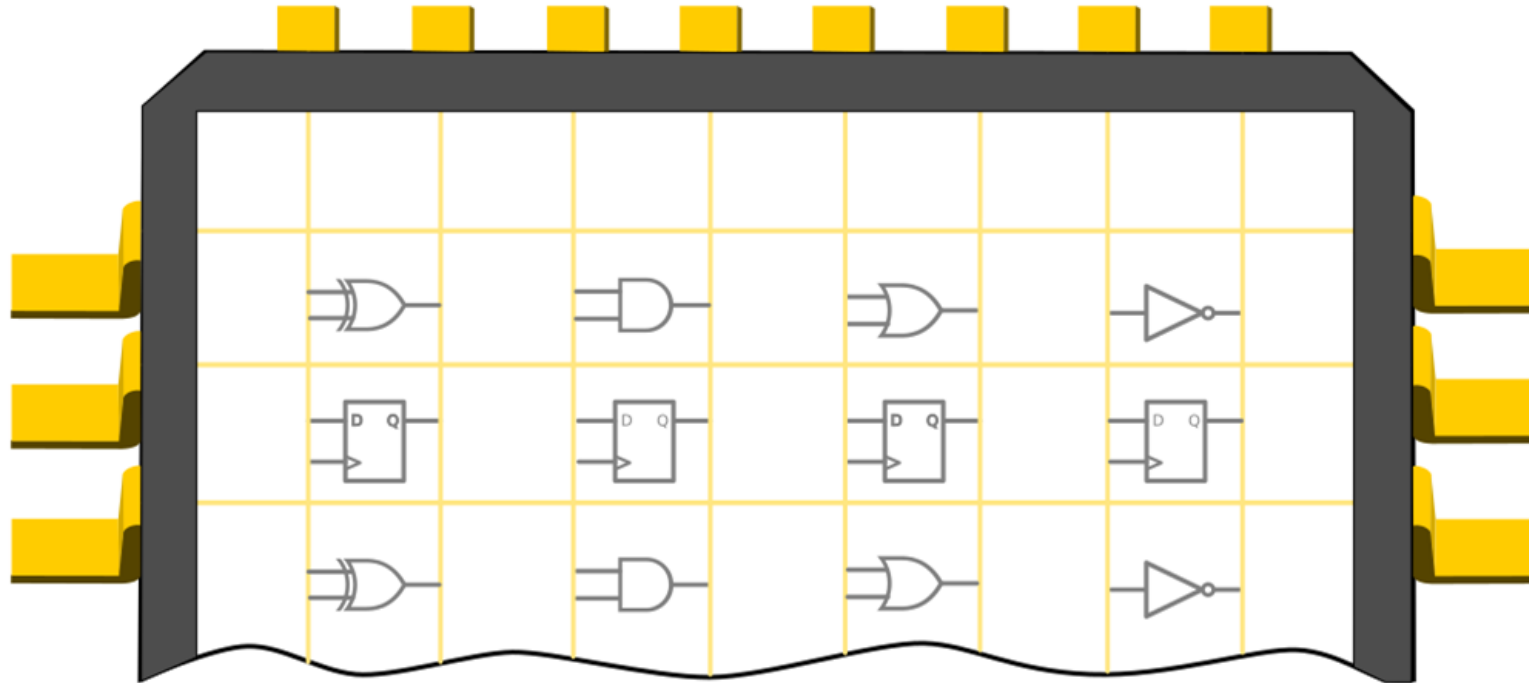
Cualquier circuito digital, por muy complejo que sea, se descompone en estos 3 tipos de **componentes elementales**

Tecnología FPGA



FPGA: Chip “en blanco” que contiene una matriz con los 3 componentes básicos: puertas lógicas, biestables y cables

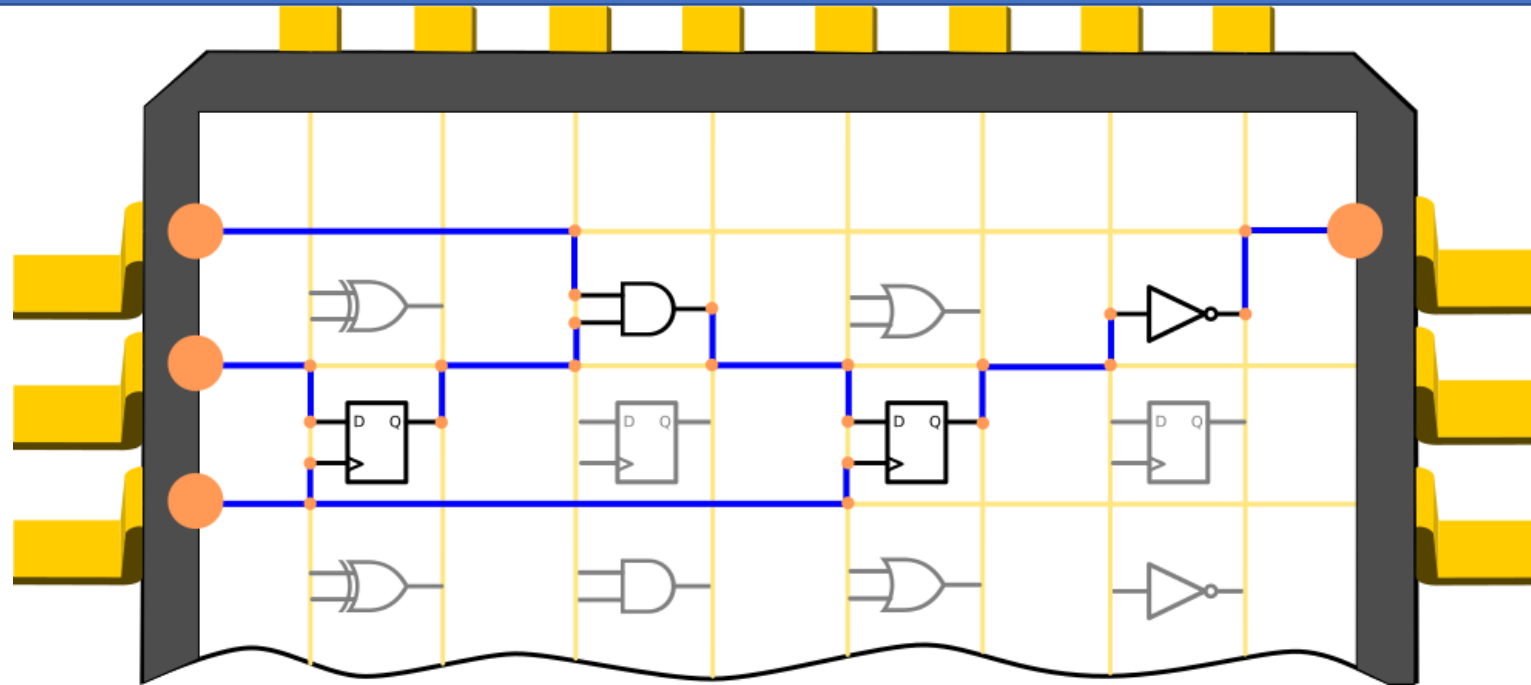
Configuración



Las uniones entre cables son **configurables**

FPGA **no configurada**

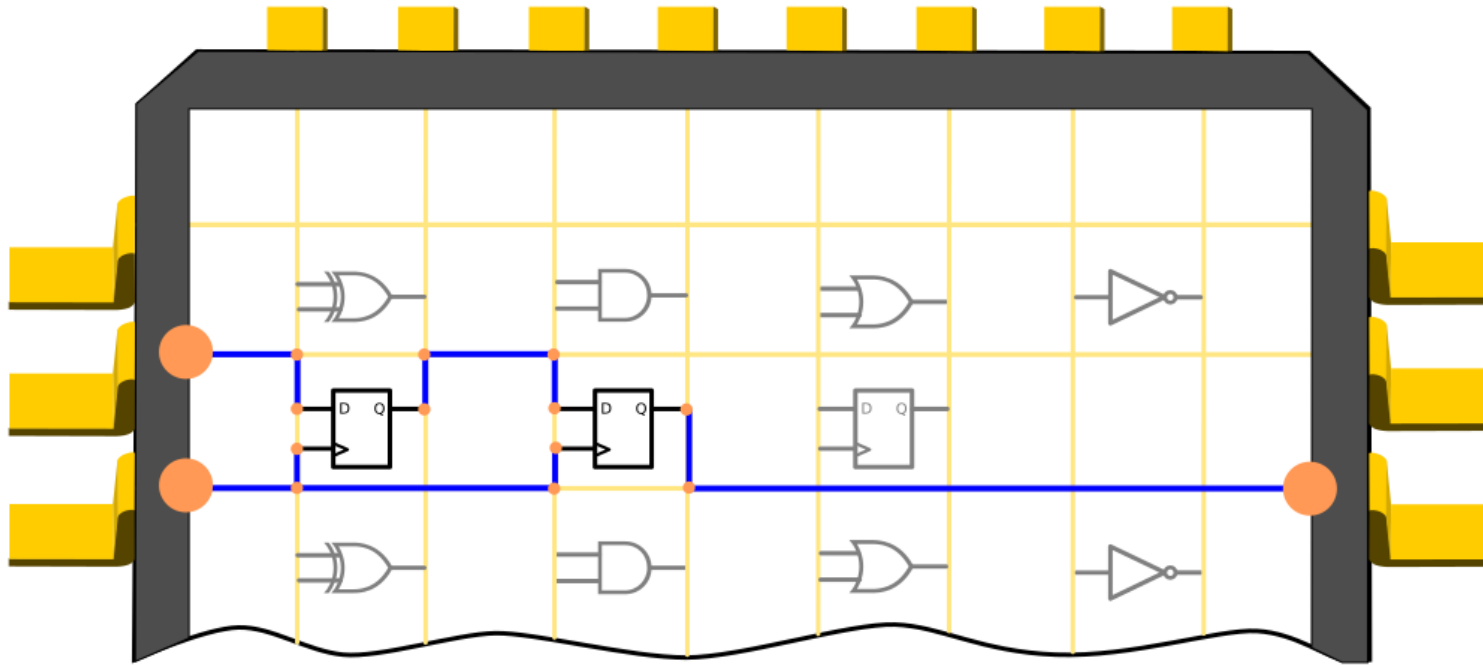
Configuración



Circuito creado configurando las uniones entre los elementos básicos de la FPGA

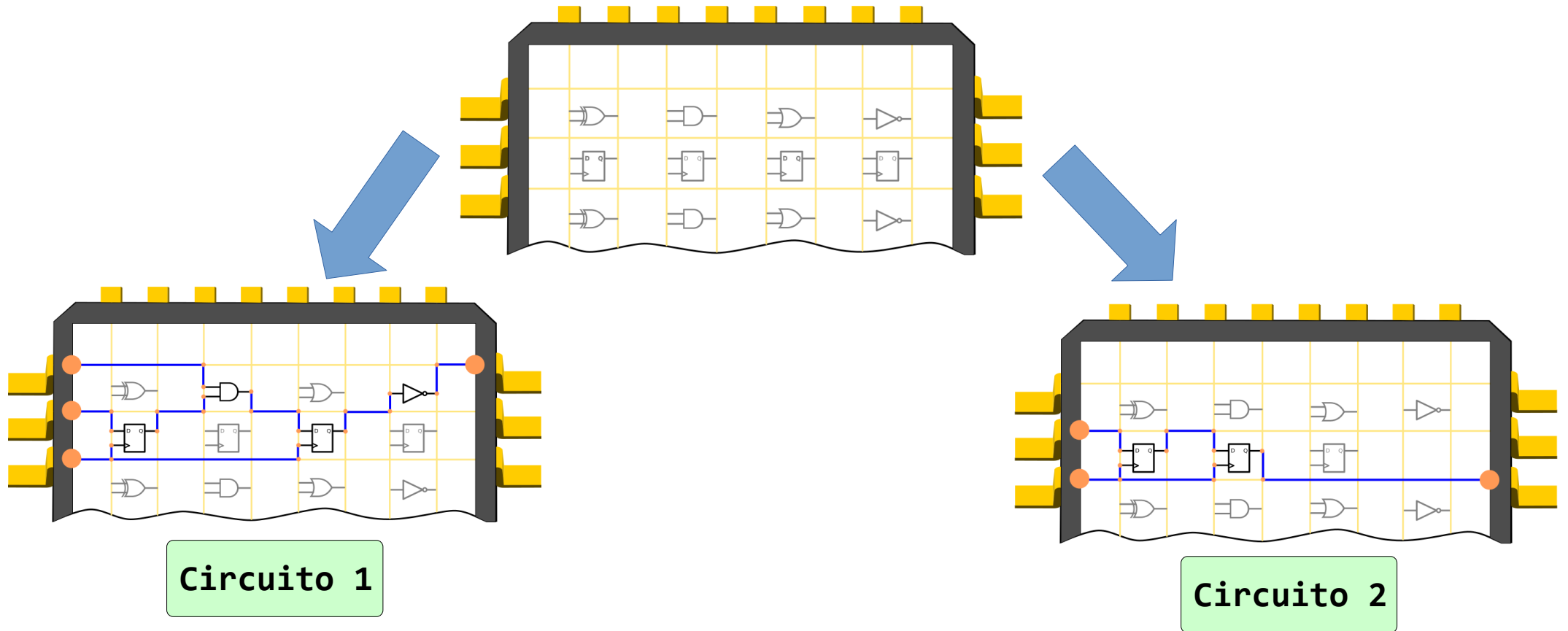
FPGA configurada

Reconfigurar



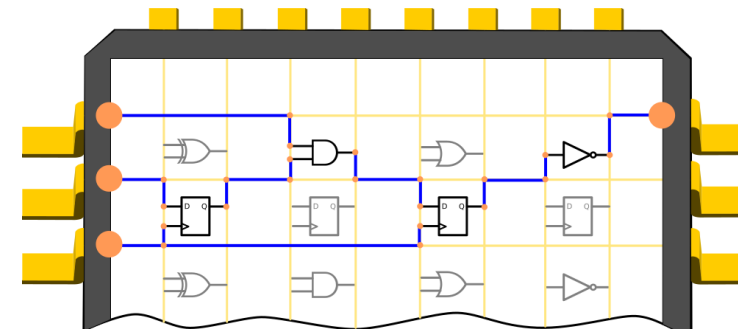
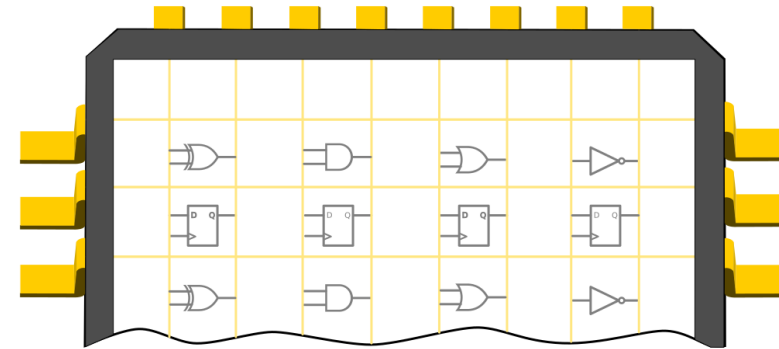

**¡Sólo con cambiar las uniones,
aparece otro circuito diferente!**

Lógica programable



Bitstream

Bitstream
...1001111100010011001010101.....



La configuración se hace cargando un *bitstream* en la FPGA

Estos bits determinan qué cables se conectan y cuales no

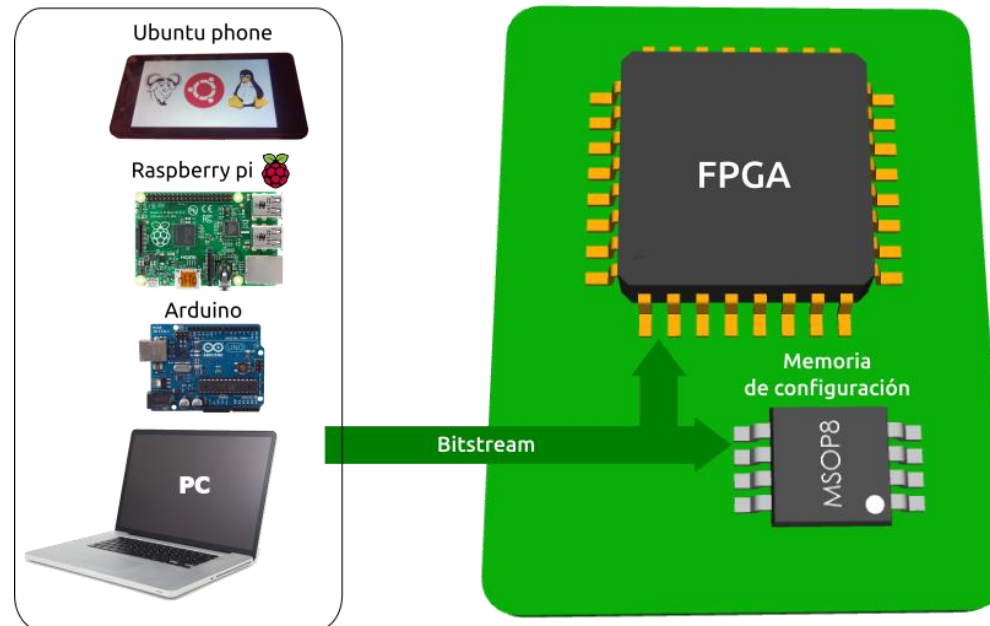
Memoria de configuración

FPGA **volátiles**: pierden su configuración al quitar alimentación

El *bitstream* se guarda en una **memoria externa**: memoria de configuración

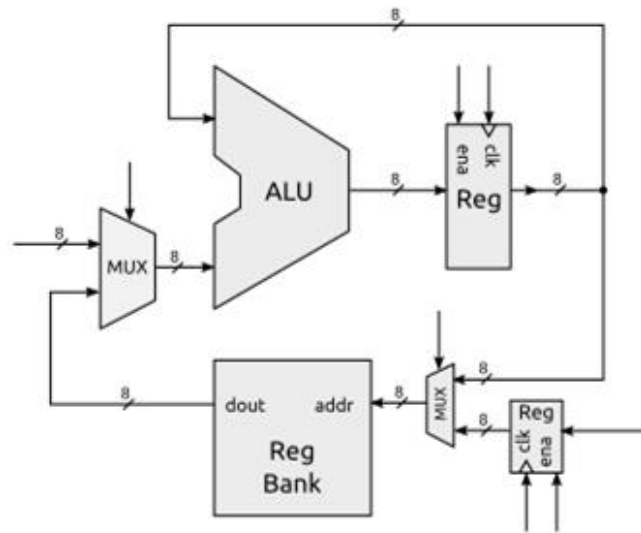
Al arrancar la FPGA se carga con el *bitstream* de la memoria de configuración

Desde un ordenador externo se carga el *bitstream* en la memoria de config.



Diseñando circuitos digitales

Los circuitos digitales modernos se diseñan usando **lenguajes de Descripción Hardware (HDL)**



HDL

```
module simplez #(
    parameter BAUD = `B115200,
    parameter WAIT_DELAY = `T_200ms,
    parameter ROMFILE = "prog.list",
    parameter DEBUG_LEDS = 0
)
(
    input wire clk,
    input wire rstn_ini,
    output wire [3:0] leds,
    output wire stop,
    output wire tx,
    input wire rx
);

reg [DW-1: 0] alu_out;
reg flag_z;

always @(*) begin
    if (alu_op2)
        alu_out = alu_in;
    else if (alu_cir)
        alu_out = 0;
    // Suma de operador 1 + operador 2
    else if (alu_add)
        alu_out = reg_a + alu_in;
    else if (alu_dec)
```

Descripción en lenguajes HDL

Desde una descripción en HDL podemos **simular** el circuito, **generar el bitstream** para FPGAs o **fabricar el circuito integrado**

Fichero HDL

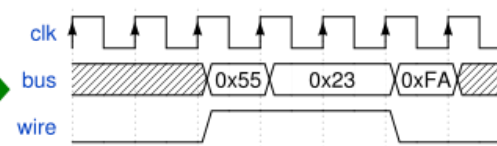
```
module simplez #(
    parameter BAUD = 'B115200,
    parameter WAIT_DELAY = 'T_200ms,
    parameter ROMFILE = "prog.list",
    parameter DEBUG_LEDS = 0
)
{
    input wire clk,
    input wire rstn_ini,
    output wire [3:0] leds,
    output wire stop,
    output wire tx,
    input wire rx
};

reg [DW-1: 0] alu_out;
reg flag_z;

always @(*) begin
    if (alu_op2)
        alu_out = alu_in;
    else if (alu_clr)
        alu_out = 0;

    // Suma de operador 1 + operador 2
    else if (alu_add)
        alu_out = reg_a + alu_in;
    else if (alu_dec)
```

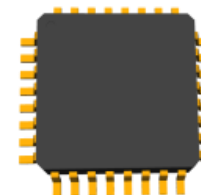
Simulación



Bitstream (FPGA)

... 0110111100101 ...

Fabricación chips



FPGA libre

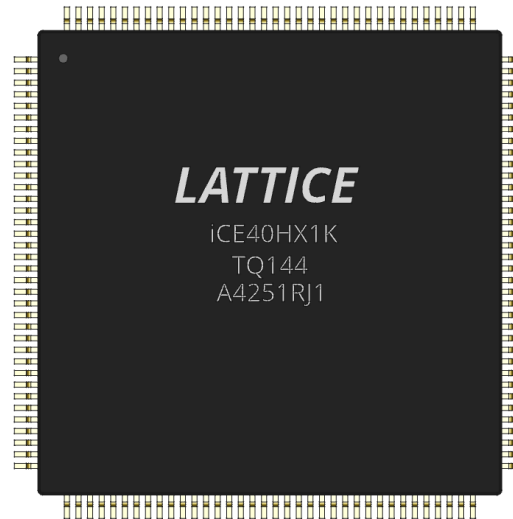
Definición

Denominamos **FPGAs libres** a aquellas FPGAs que disponen de una *toolchain* **totalmente libre**

FPGAs libres actualmente

Familia **Lattice iCE40**

Sólo Lenguaje Verilog



Proyecto Icestrom

Herramientas programadas en C/C++

Bajar del repo y compilar

Línea de comandos

Se usan típicamente junto con make (bajo nivel)

<http://www.clifford.at/icestorm/>

<https://github.com/cliffordwolf/icestorm>

APIO

Autor: **Jesús Arroyo**

Multiplataforma (Linux, Mac, Windows, Raspberry)

Línea de comandos

Programado en python

Multiplaca: icestick, icezum, icoboard, go-board

Comandos



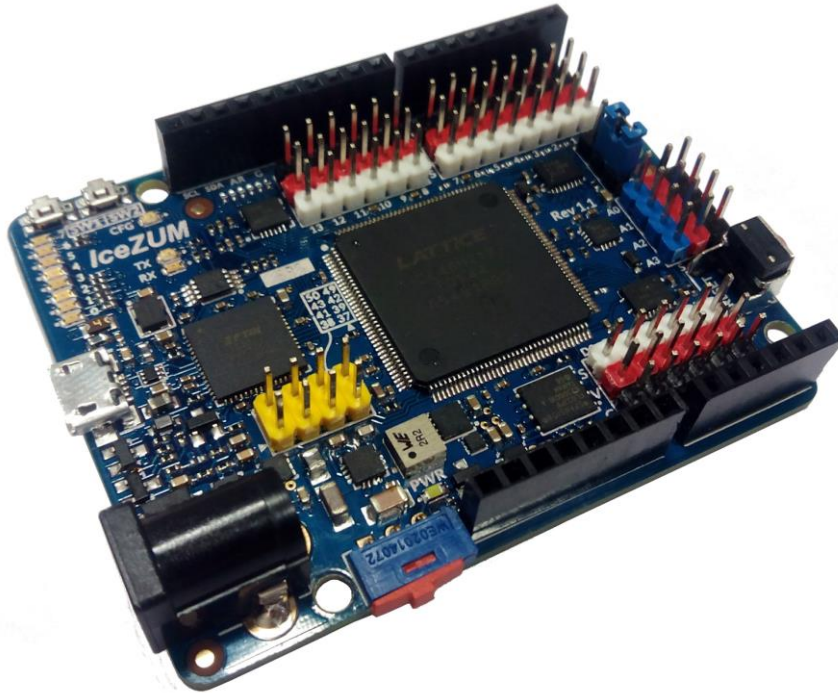
APIO

Icestorm

<https://github.com/FPGAwards/apio>

Demo

Icezum Alhambra v1.1

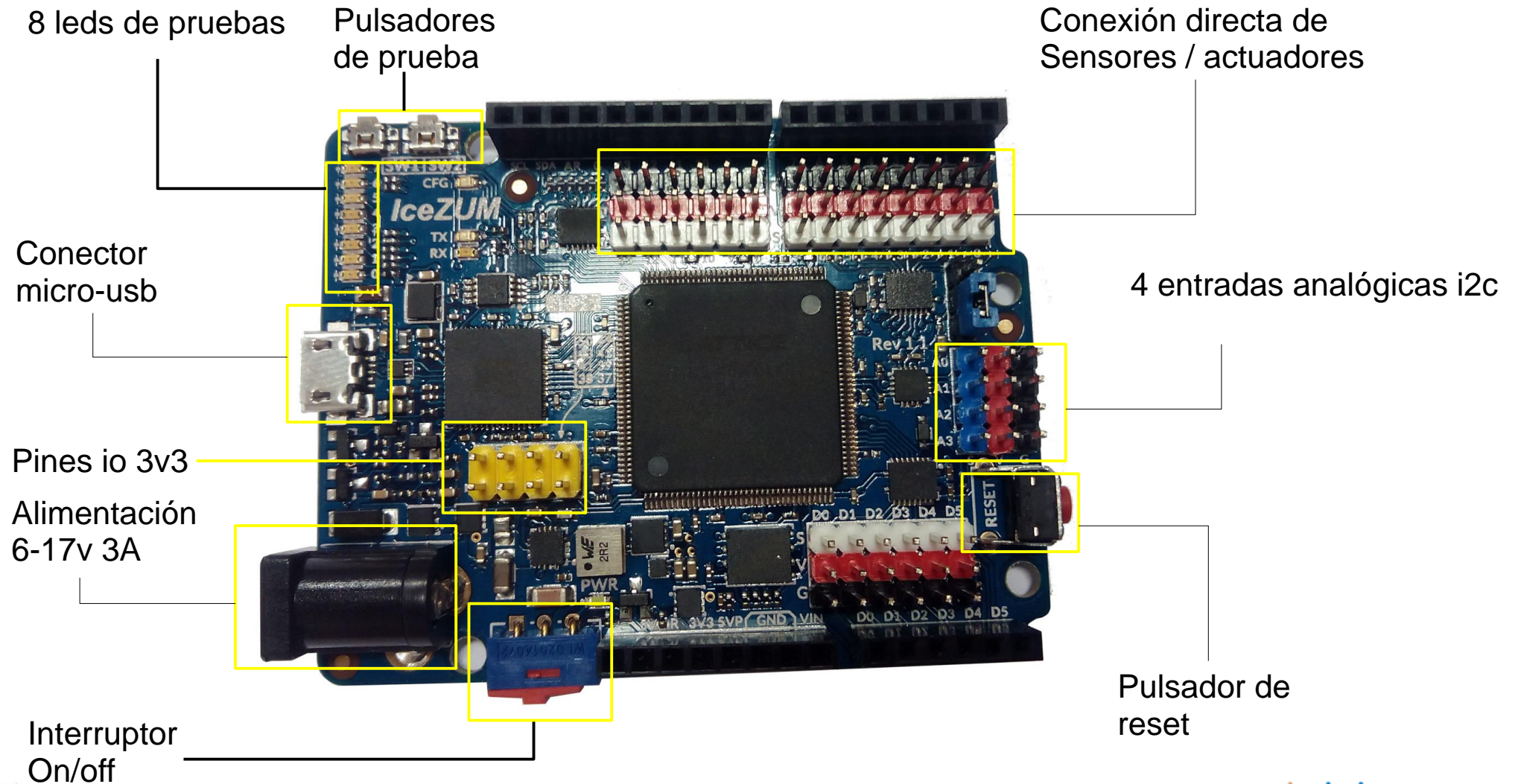


<https://github.com/FPGAwards/icezum/wiki>

Autor: **Eladio Delgado**

- ☐ Arduino de las FPGAs
- ☐ Compatible Arduino
- ☐ Fácil conexión de circuitos externos/sensores/servos
- ☐ Reutilización de los shields de Arduino
- ☐ 20 entradas/salidas de 5v
- ☐ 3A corriente de entrada
- ☐ Perfecta para hacer robots

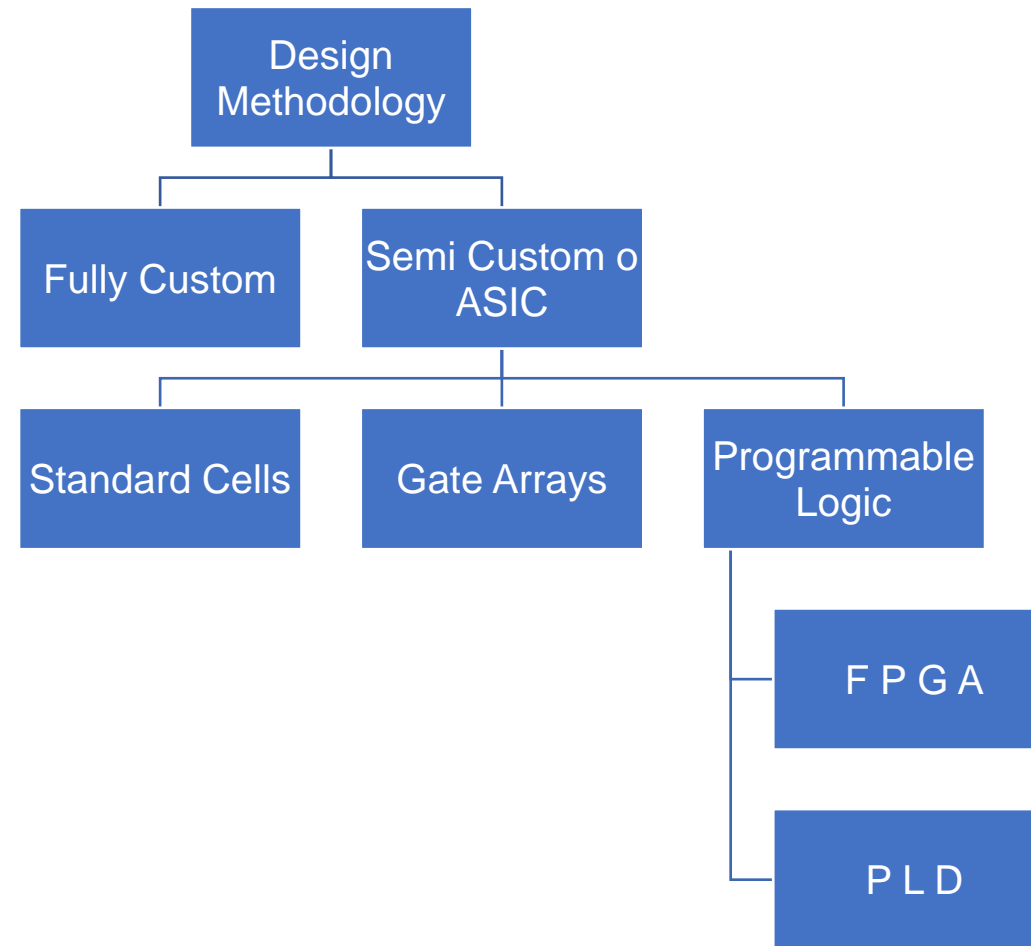
Icezum Alhambra v1.1



Índice

1. Introducción a la Práctica 4
2. ¿Qué es una FPGA?
3. Ciclo de vida en microelectrónica
4. Verilog HDL
5. Test

Metodologías de diseño de Circuitos Integrados (IC)



Diseño de ASIC

Standard Cell Base

- El chip se diseña desde cero.
- El ingeniero diseña algunas o todas las celdas lógicas, circuitos, y distribución del chip de forma específica para un IC full-custom.
- Ventajas: Total flexibilidad, alto grado de optimización en área y rendimiento.
- Desventajas: Es un proceso caro, el esfuerzo requerido para el diseño es enorme.

Full Custom

- Utiliza celdas lógicas prediseñadas, preprobadas y precaracterizadas como bloques de diseño, obtenidas de librerías de celdas estandarizadas.
- La distribución del diseño en el chip es personalizada.
- Ventajas: Ahorra tiempo y dinero, reduce el riesgo en comparación al diseño full-custom.
- Desventajas: Todavía incurre en un alto costo "none-recurring-engineering (NRE)" y amplio tiempo de manufactura.

Gate Array

- Algunas partes del chip están prefabricadas mientras que otras se personalizan para el circuito particular del cliente.
- Se prefabrican una base de celdas idénticas en forma de una matriz bidimensional de puertas (este chip terminado parcialmente se conoce como "gate-array template")
- Las conexiones entre los transistores en el interior de las celdas como entre las mismas celdas se llevan a cabo de forma particular para cada cliente.
- Las máscaras personalizadas se hacen solamente para el cableado.

PLD

- Un PLD es un chip de propósito general para implementar circuitos lógicos.
- Los cables y transistores ya están prefabricados en un PLD.
- Las celdas lógicas y sus conexiones se pueden programar por el usuario final para implementar circuitos específicos.
- No existe la necesidad de crear máscaras personalizadas para cada cliente.

¿Qué es una FPGA?

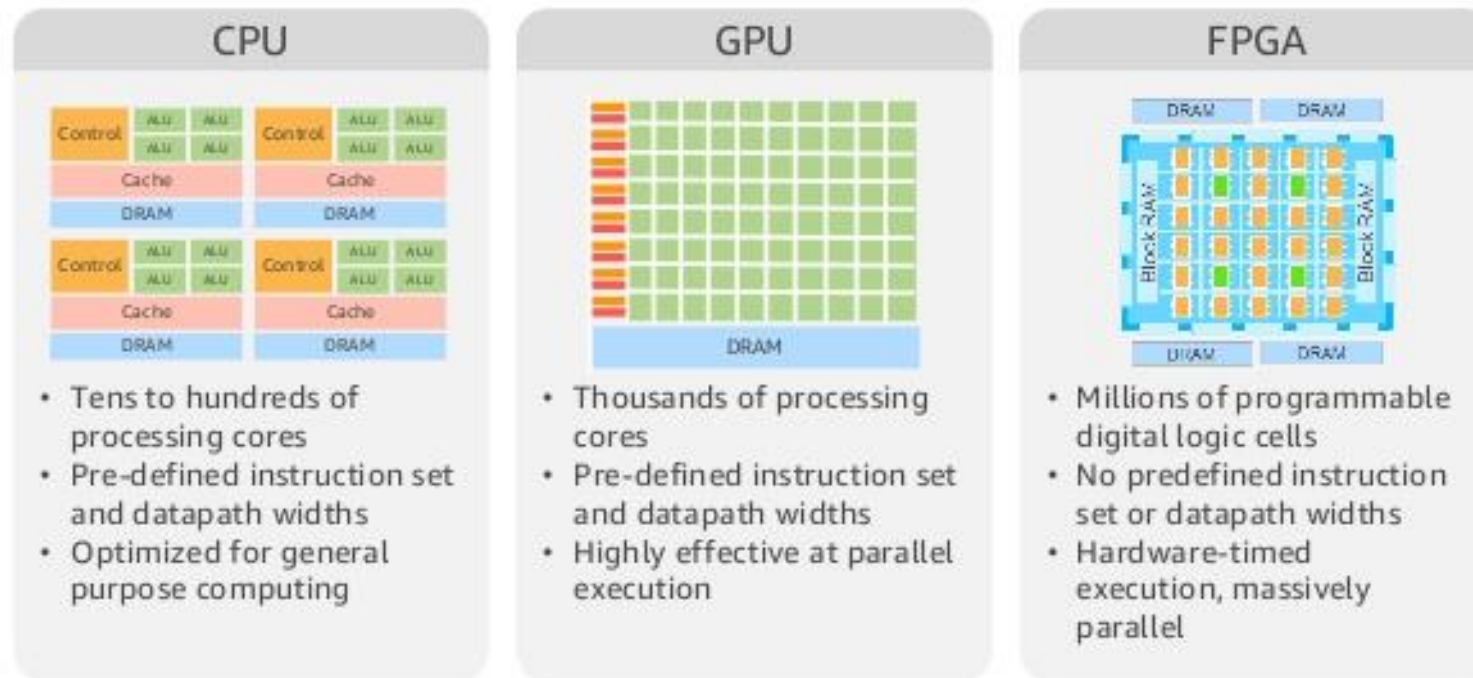
- Siglas de Field Programmable Gate Array.
- Es un dispositivo reprogramable, que permite implementar diversos circuitos digitales en él.
- Hay 3 tipos principales:
 - Basadas en **memorias RAM**: las más comunes, es necesario programarlas cada vez después de su encendido
 - Basadas en **memorias Flash**: de uso en aviación y espacio, están programadas al encenderse, menor consumo
 - Basadas en **fusibles**: sólo se pueden programar una vez, se usan en entornos con alta radiación

¿Por qué usar las FPGA?

- Hay aplicaciones en las cuales usar un microcontrolador no es suficiente, o usar una FPGA posee costos comparables (por ejemplo, codificar y decodificar en MPEG).
- Actualmente hay modelos que incluyen elementos adicionales en el mismo chip.
 - La Zynq de Xilinx posee uno o más cores ARM Cortex embebidos en el chip.

Diferencias con otros tipos de IC

PARALLEL PROCESSING IN GPU AND FPGA



aws
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Ventajas de las FPGA

- Son dispositivos reconfigurables.
- Bajo costo respecto a los ASIC.
- Los circuitos se “ejecutan” más rápido que en otros dispositivos reprogramables.
- Mayor cantidad de entradas y salidas que un microcontrolador.
- La algoritmia implementada en la FPGA se ejecuta más rápido que en un microcontrolador.

Ventajas de las FPGA

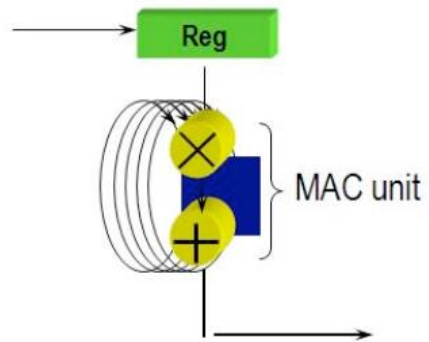
- Al ser circuitos digitales, la “ejecución” de cada bloque es en paralelo, no así en un microcontrolador.
- Son útiles para realizar prototipos que luego serán llevados a ASIC si es necesario.

Desventajas de las FPGA

- Las FPGAs basadas en RAM, pierden su configuración al suprimir la energía → es necesario almacenar su configuración en una memoria no volátil y configurar la FPGA al arrancar.
- Poseen retardos de propagación mayores a los existentes en ASIC o standard cells
 - Un procesador de alta velocidad (~GHz) se ejecuta mucho más rápido en ASIC que en una FPGA.

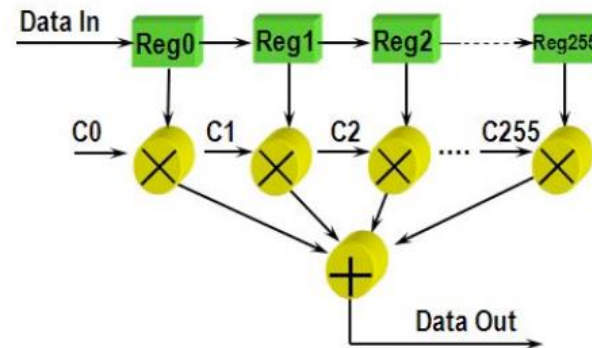
Balance entre área y rendimiento

Conventional DSP Device (Von Neumann architecture)



256 Loops needed to process samples

FPGA



All 256 MAC operations in 1 clock cycle

Fig. 3: FIR filter comparison between DSP and FPGA

La principal ventaja de las FPGAs reside en su paralelismo natural, lo cual permite realizar operaciones mucho más rápido que con una arquitectura secuencial convencional.

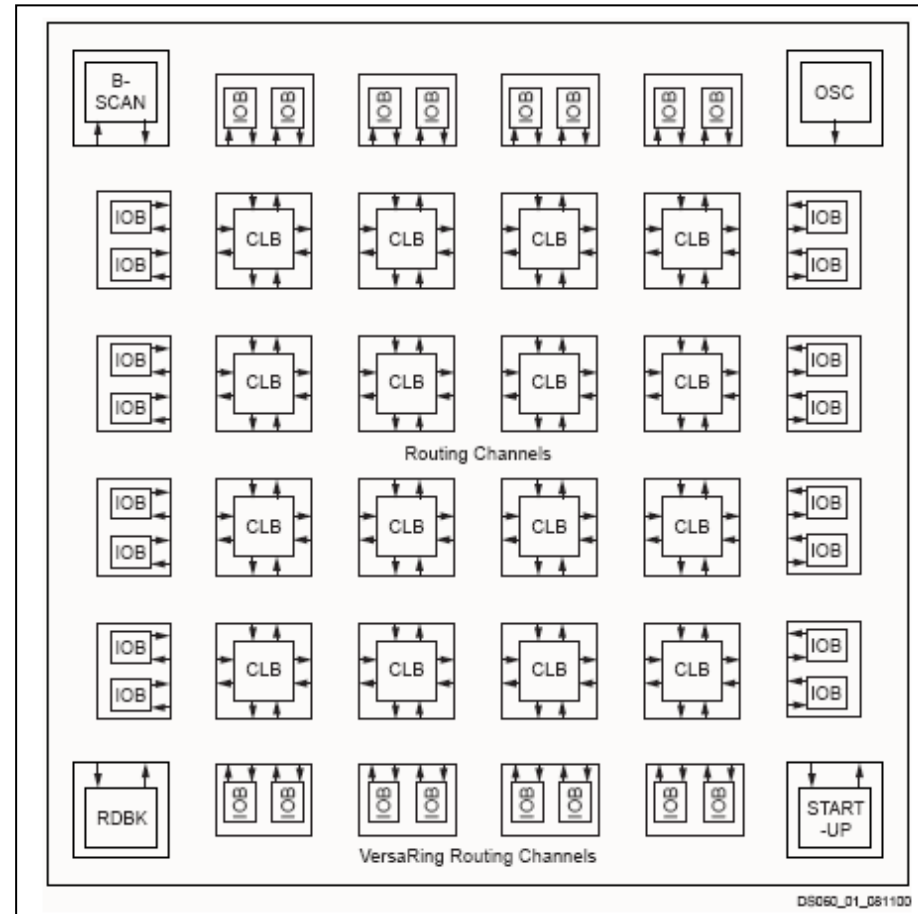
De este modo, siempre existe una contrapartida entre cuán rápido debe ser el diseño y qué tanta área de la FPGA se requiere para implementarlo.

Fabricantes de FPGAs

- Xilinx/AMD
- Altera/Intel
- Microchip/Microsemi
- Lattice Semiconductor
- NanoXplore
- Cypress Semiconductor
- Achronix Semiconductor
- QuickLogic

(fuente: Wikipedia.org)

Arquitectura de una FPGA

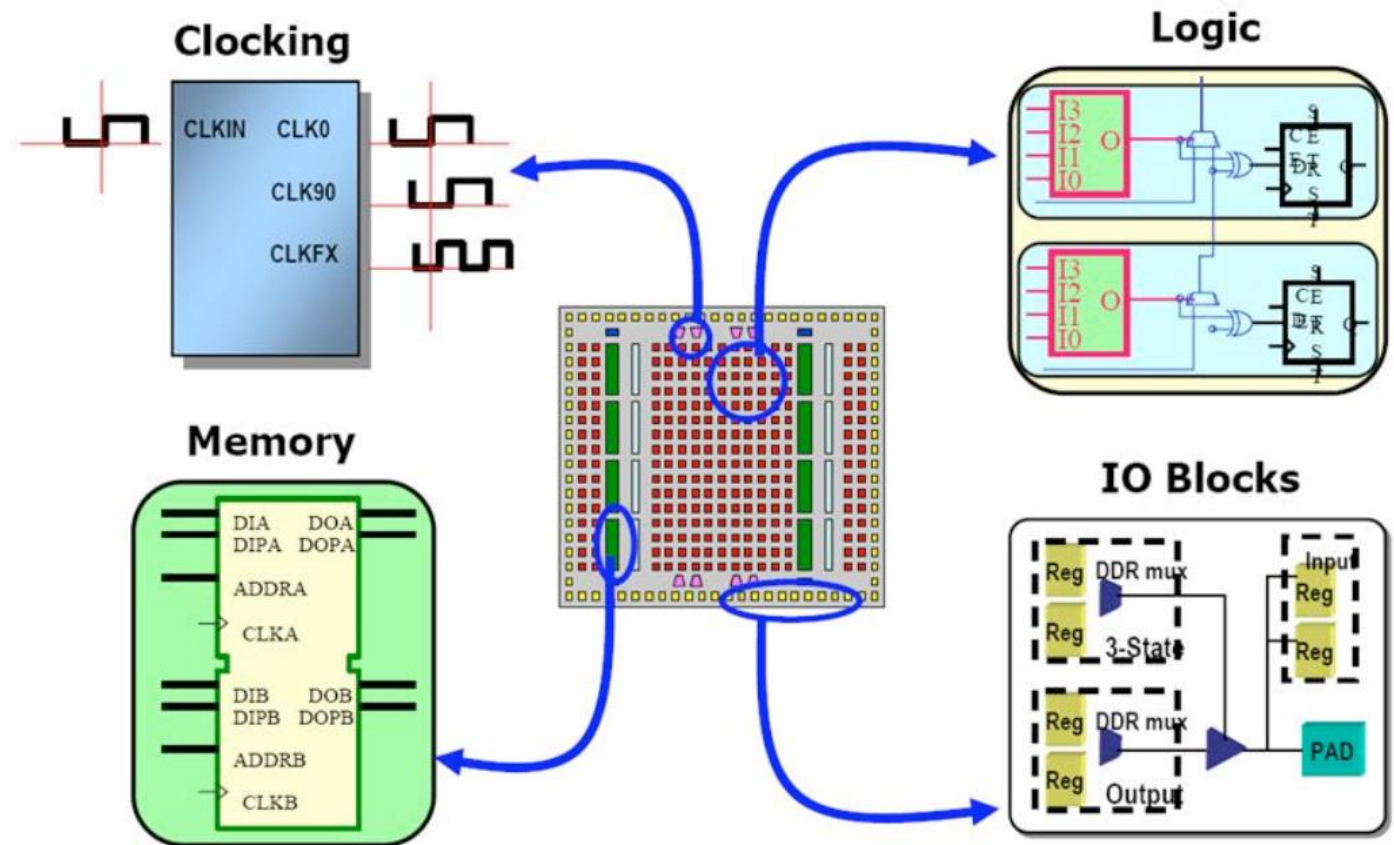


- Una FPGA básica (Xilinx Corp.)

Arquitectura de una FPGA

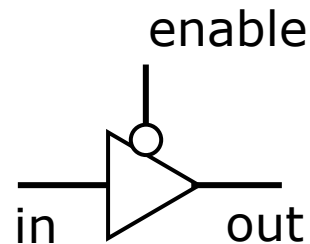
Una FPGA posee al menos 3 bloques:

- **CLB:** Configurable-logic blocks, donde se implementan los circuitos lógicos.
- **IOB:** Input-output block, donde se conectan las configuraciones internas con pines de entrada y salida.
- **DCM:** Digital Clock Managers, permiten entregar señales de reloj a toda la FPGA.

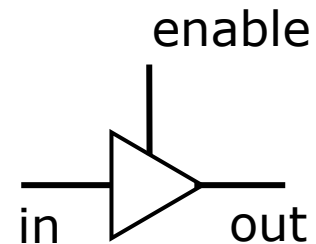


Elementos básicos

- Buffer tri-estado (TRI)
 - Permite o impide el paso de datos de acuerdo a una señal de control.



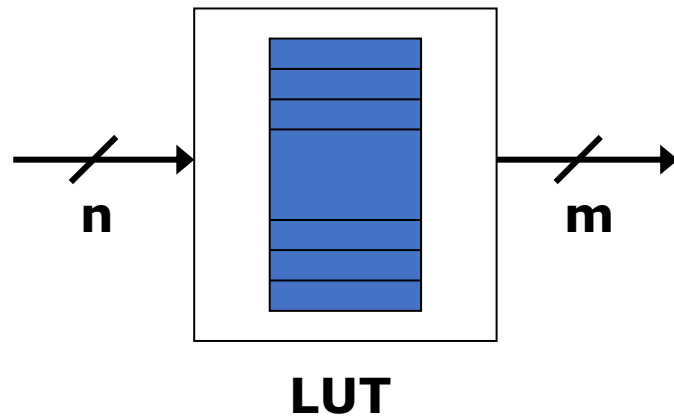
Low-active



High-active

Elementos básicos

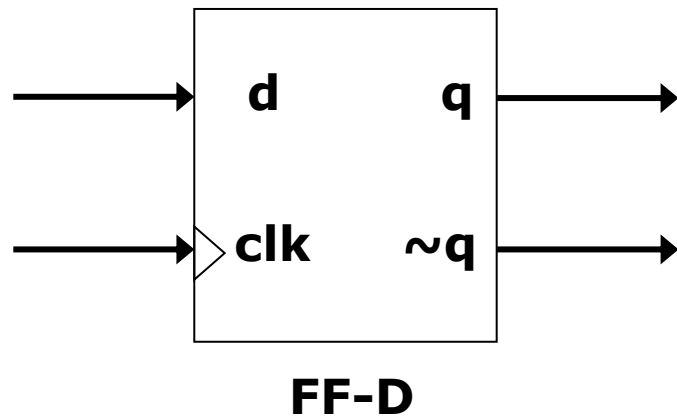
- Look-up Table (LUT)
 - En esencia es una memoria RAM con valores predefinidos.



En los DSP de punto fijo se utilizan Look-up Tables para implementar funciones trigonométricas

Elementos básicos

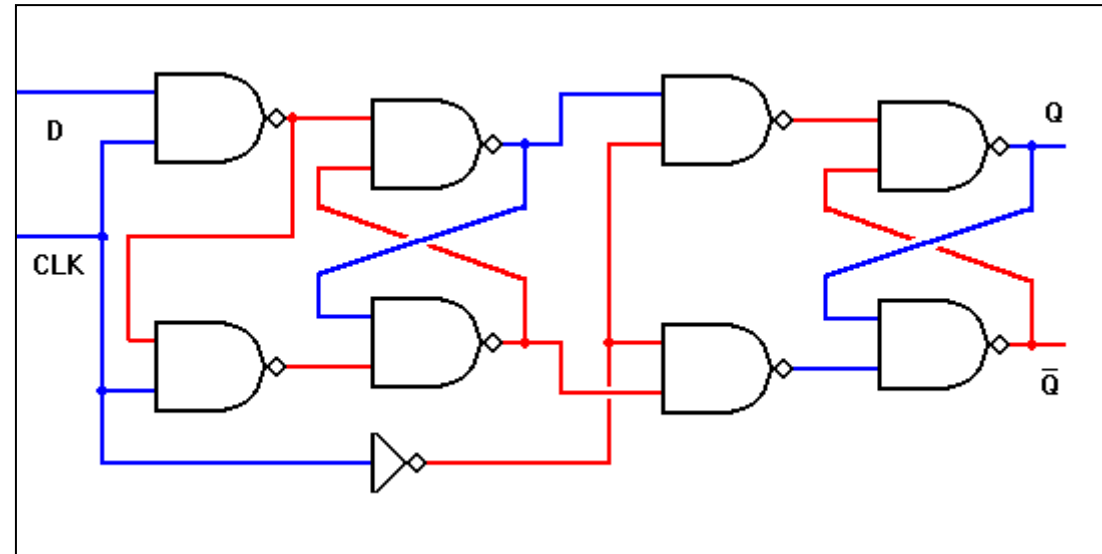
- Flip-Flop (FF)
 - Elemento que almacena un bit de acuerdo a una señal de reloj de entrada.



**Los más utilizados son los
Flip-flop tipo-D**

Elementos Básicos

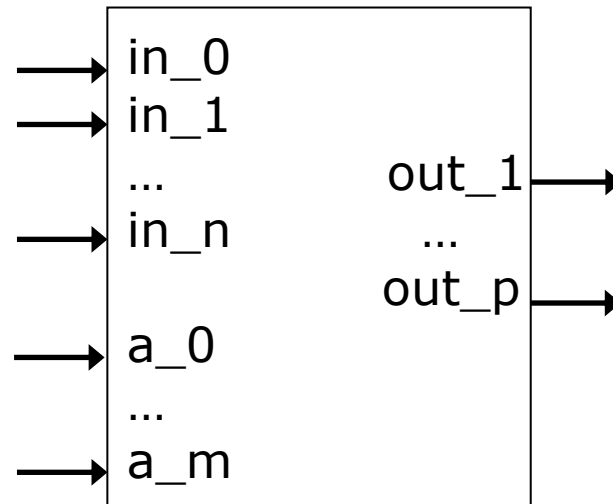
- Flip-Flop tipo D



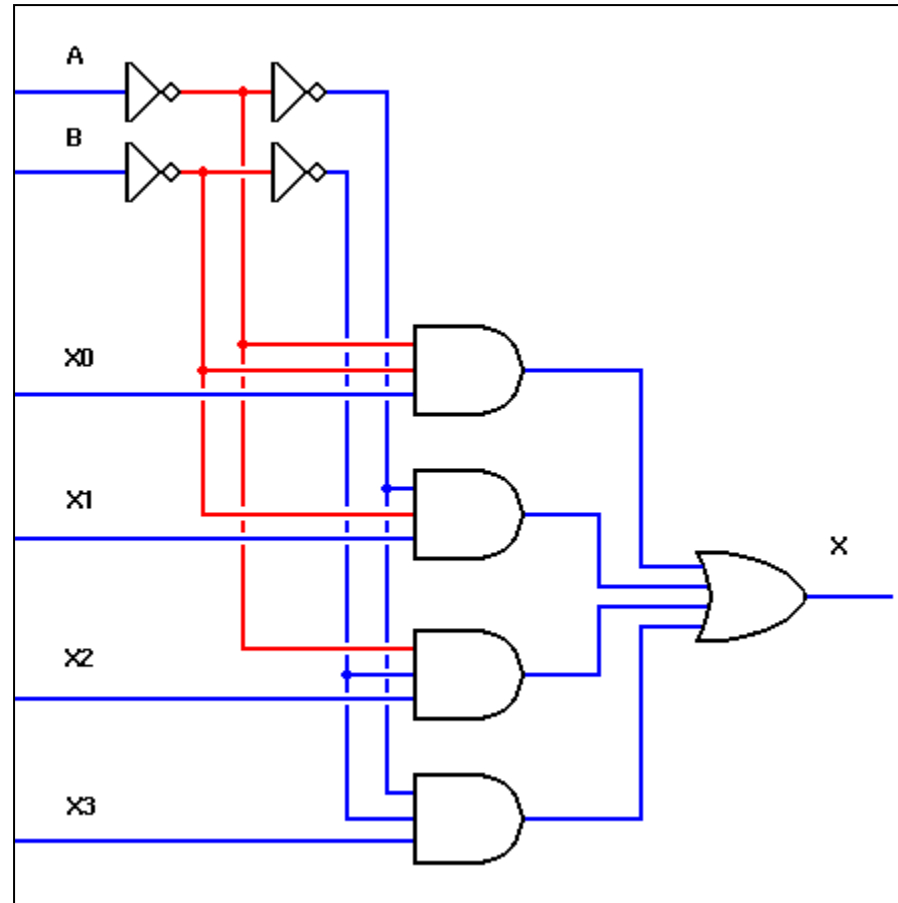
Elementos Básicos

- Multiplexor

- Permite seleccionar una o más señales de entrada a partir de bits de selección



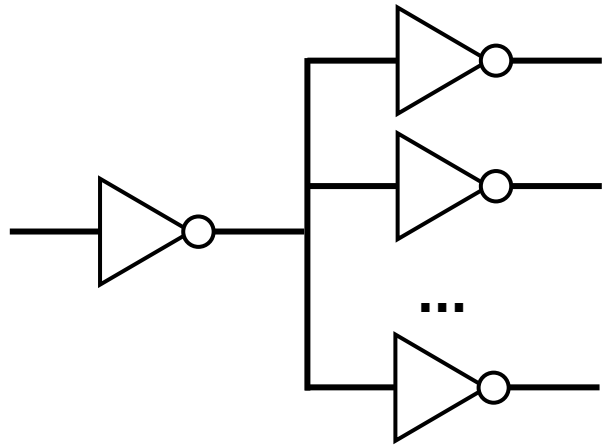
Elementos Básicos



Multiplexor 4 a 1

Conceptos básicos

- Fanout
Cuántas entradas de una familia lógica pueden conectarse a una misma salida.



$$fanout = \frac{I_{salida_{m\acute{a}x}}}{I_{entrada_{m\acute{a}x}}}$$

Conceptos básicos

- Niveles de voltaje

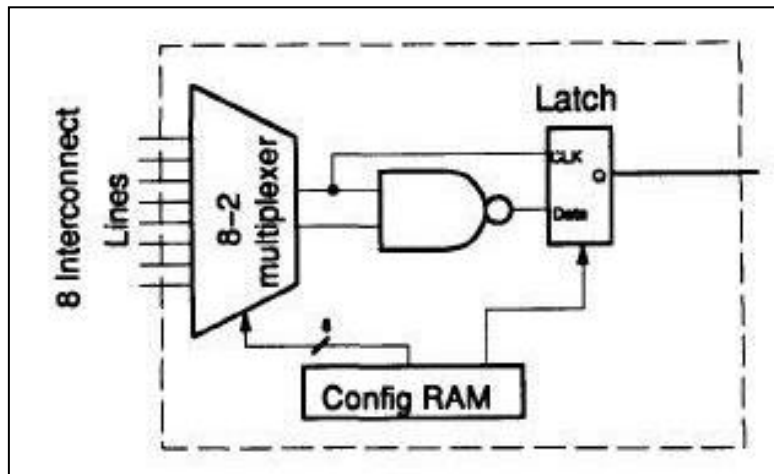
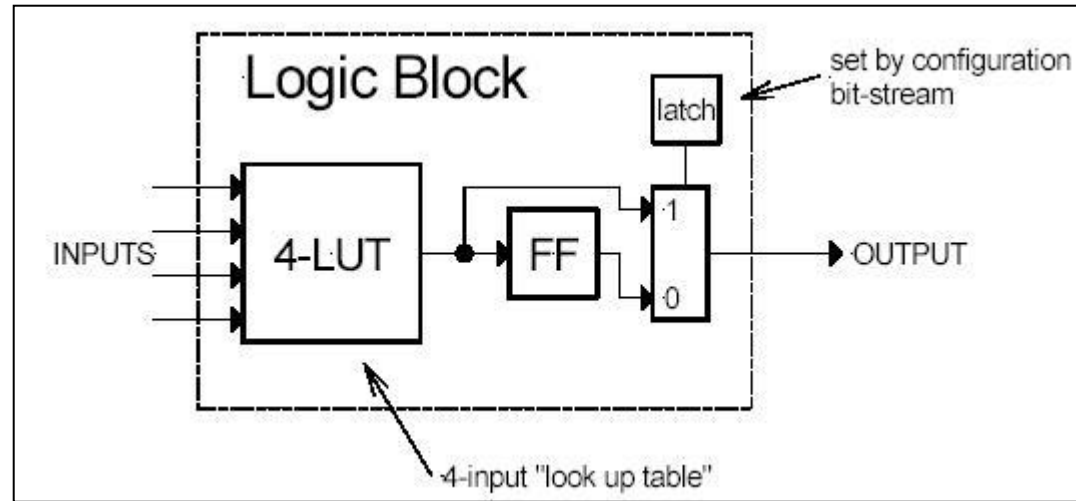
Un 1 y 0 lógico se pueden representar de diversas maneras. Ejemplos:

- TTL 1 = 5V 0 = 0V
- LVTTL 1 = 3.3V 0 = 0V
- En I2C 1 = High-Z 0 = 0V
- USB usa voltajes diferenciales

Cómo es un slice

- Una **slice** es la base de las CLB.
- En esencia es un bloque de **look-up tables enlazadas con una salida que puede ser registrada (flip-flop)** o directa, pudiendo implementarse diversas funciones lógicas.
- También hay implementaciones con multiplexores.

Cómo es un slice



Implementar funciones

- Para implementar funciones combinacionales, se programan las tablas de verdad en las LUT y se usa la salida directa.
- Para implementar funciones secuenciales, se utiliza el flip-flop y se realimenta la salida secuencial en una de las entradas si es necesario.

Implementar funciones

- Implementar la operación XOR2

Tabla de Verdad

A	B	C	D	Y
0	0	X	X	0
0	1	X	X	1
1	0	X	X	1
1	1	X	X	0



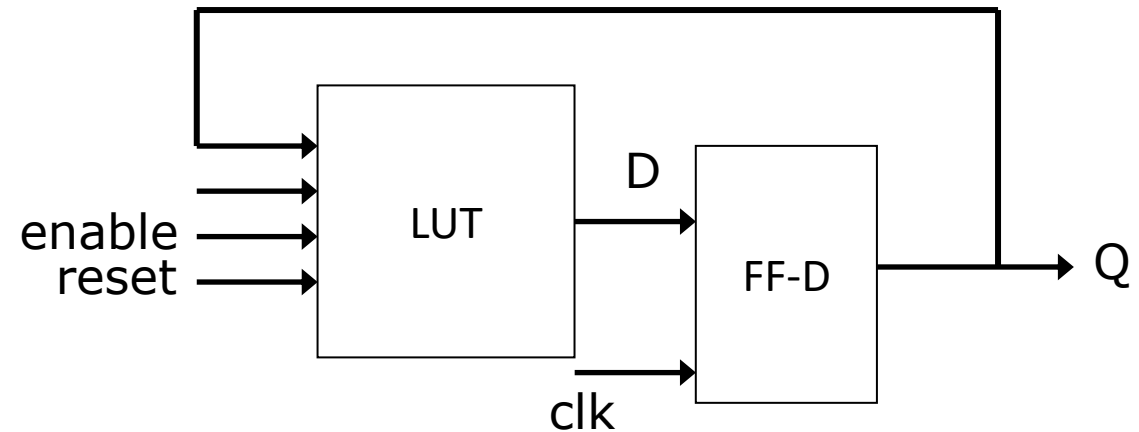
LUT

Entradas (ABCD)	Salida
00XX	0
01XX	1
10XX	1
11XX	0

Implementar funciones

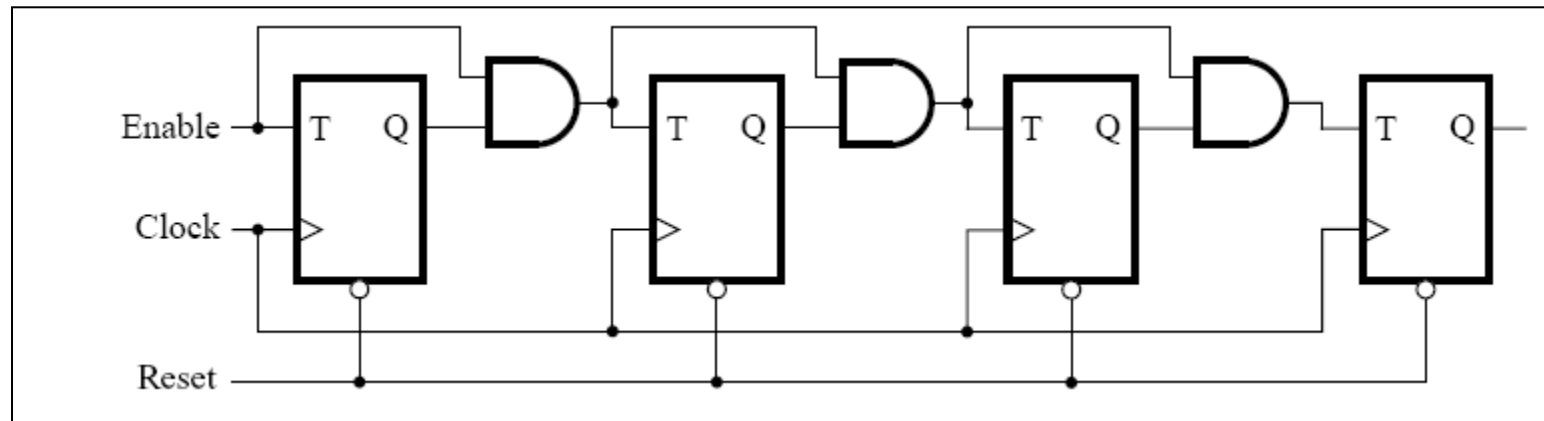
- Implementar Flip Flop T en un bloque:

reset	enable	Q _k	Q _{k+1}
0	X	X	0
1	0	Q	Q
1	1	Q	$\sim Q$



Implementar funciones

- Implementar un contador de 4 bits síncrono



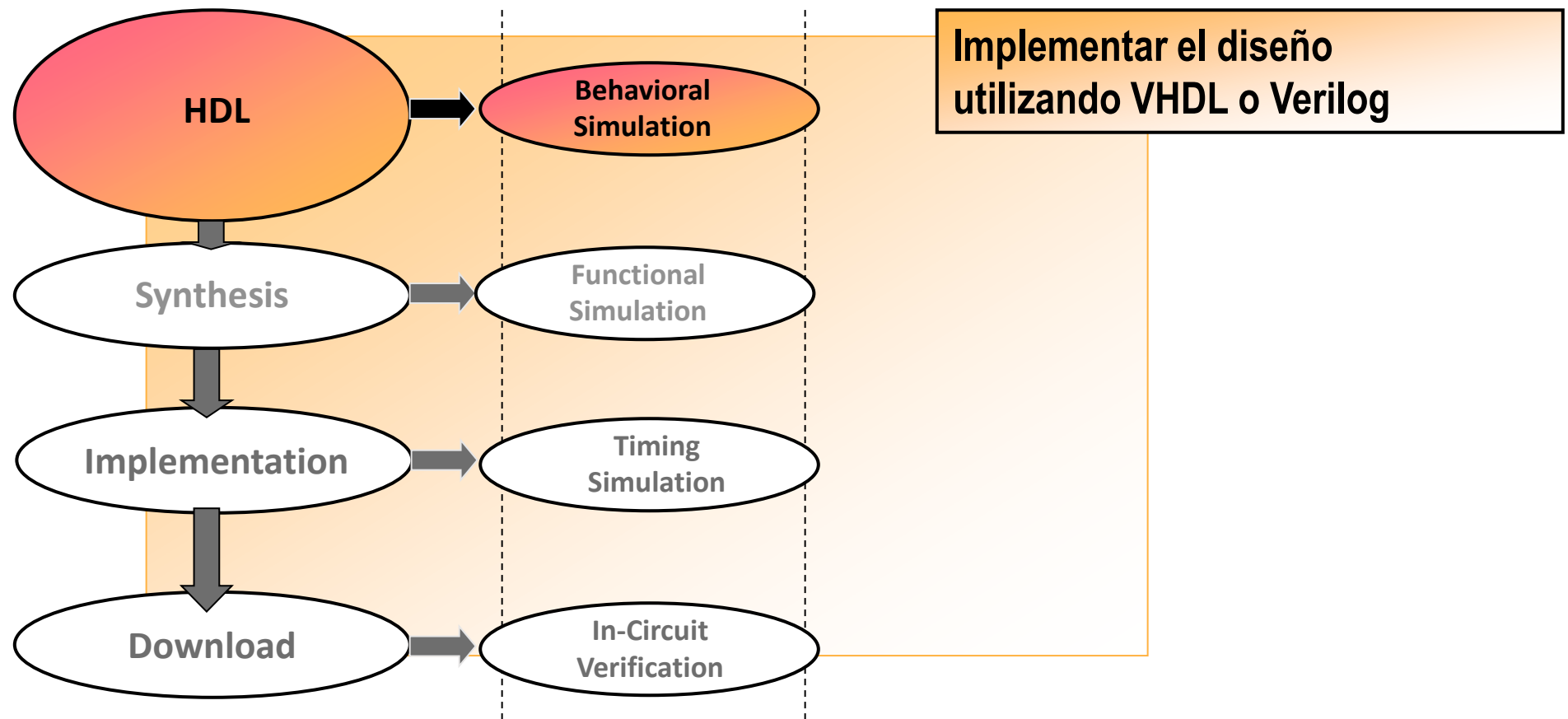
Implementar funciones

- Lo visto anteriormente sirve como muestra de cómo se implementan internamente las funciones.
- En la práctica se usan Lenguajes de Descripción de Hardware (HDL), como por ejemplo Verilog y VHDL.

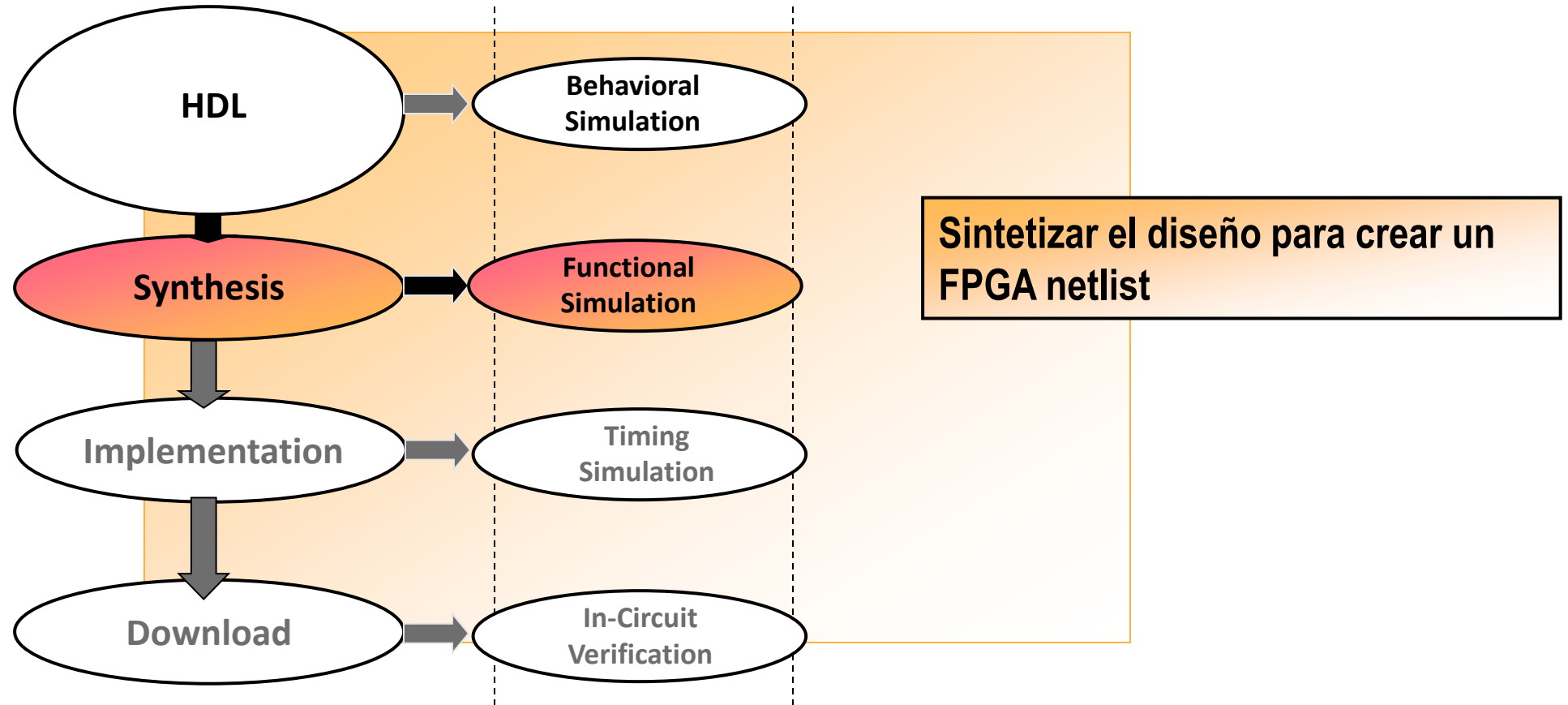
Índice

1. Introducción a la Práctica 4
2. ¿Qué es una FPGA?
3. Ciclo de vida en microelectrónica
4. Verilog HDL
5. Test

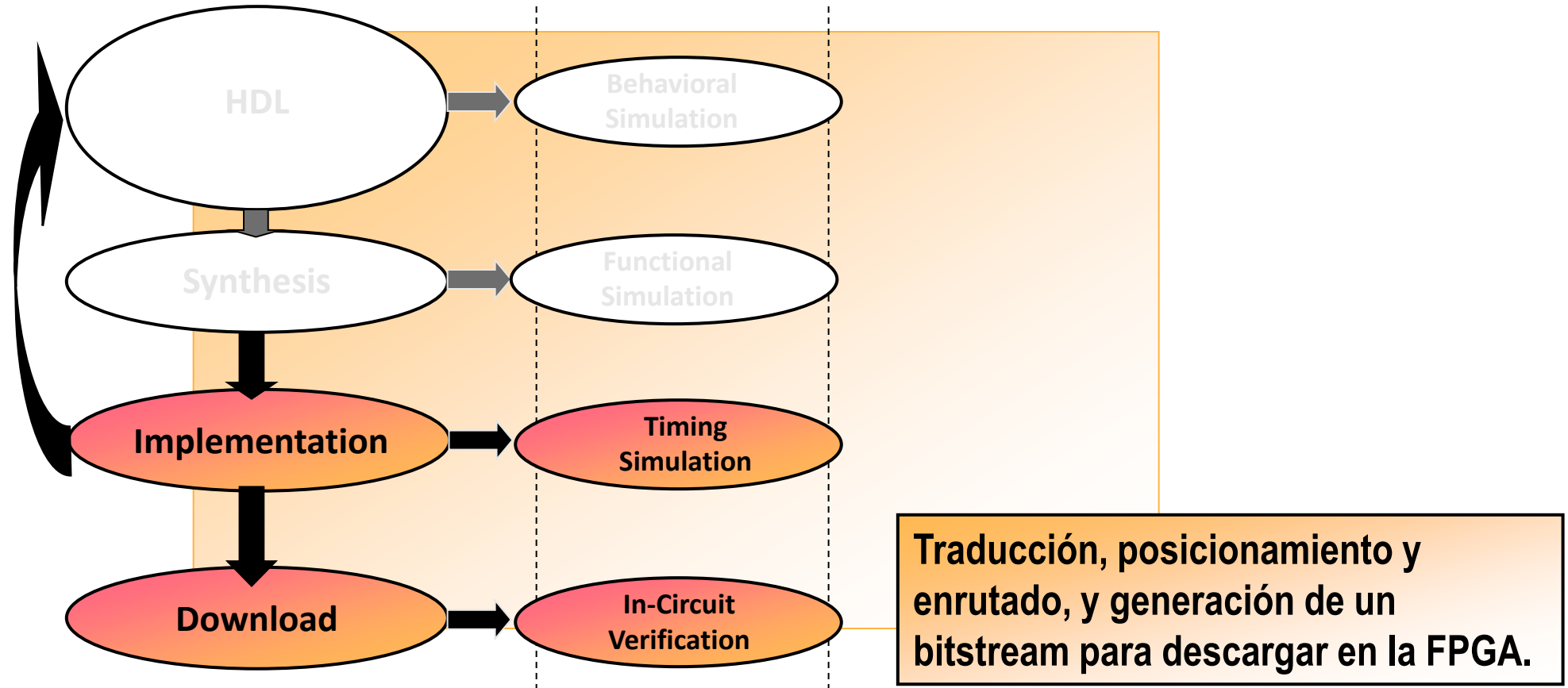
Flujo de diseño tradicional 1/3



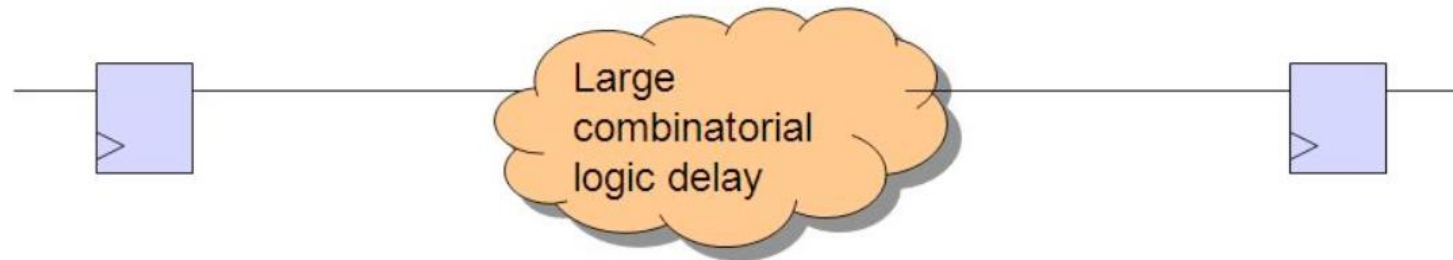
Flujo de diseño tradicional 2/3



Flujo de diseño tradicional 3/3



Ajustarse a requisitos de 'timing'



Before



After

Balance entre área y rendimiento

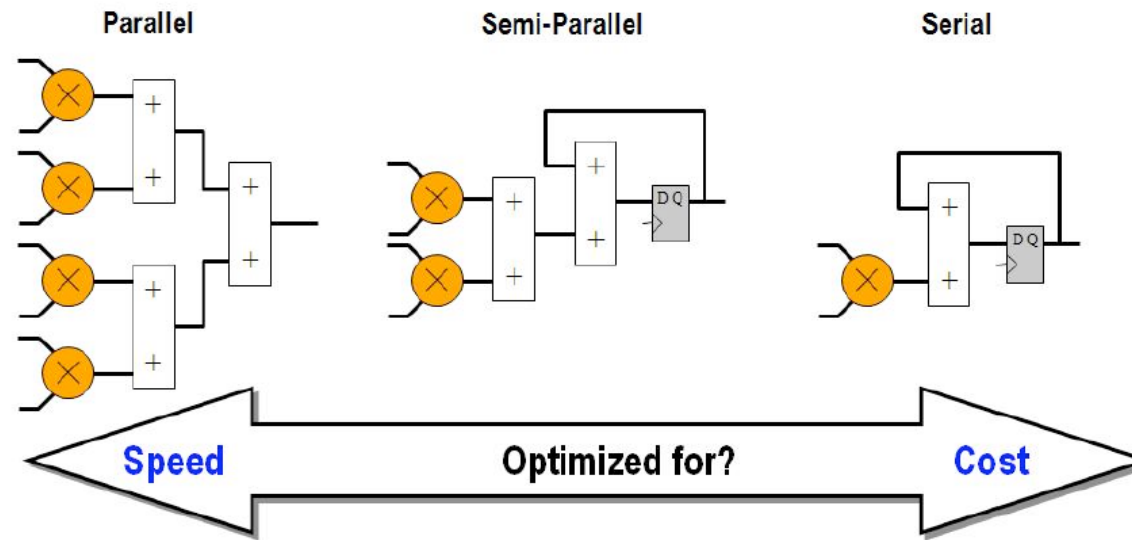


Fig. 4: Illustrating the speed/area trade-off in FPGAs

Índice

1. Introducción a la Práctica 4
2. ¿Qué es una FPGA?
3. Verilog HDL
4. Test

Índice contenidos Verilog

- Introducción a Verilog HDL
- Bloque I: Diseño de circuitos combinacionales
- Bloque II: Diseño de circuitos secuenciales

Introducción

- Verilog es un lenguaje formal para describir e implementar circuitos electrónicos.
- Es similar a un lenguaje de programación imperativo: formado por un conjunto de sentencias que indican como realizar una tarea.

Algunas diferencias:

- La mayoría de las sentencias se ejecutan concurrentemente
- Cada sentencia corresponde a un bloque de circuito

Bloque I: Índice

- Estructura general de una descripción Verilog
- Tipos de descripciones
- Señales, puertos E/S y arrays Sintaxis básica

Estructura de descripciones Verilog

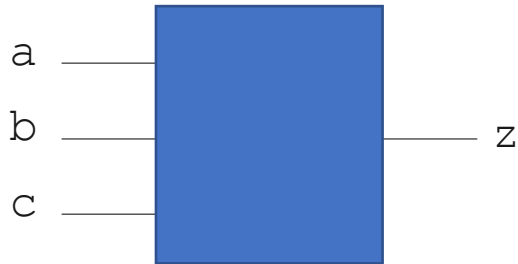
```
module mi_circuito (  
    input x, y,  
    input z,  
    output f1, f2  
);  
  
    wire cable_interno;  
    reg variable_a;  
  
    ...  
    ...  
    ...  
  
endmodule
```

Declaración del módulo con sus
entradas y salidas

Declaración de señales y variables
que se utilizarán internamente en
la descripción

Descripción del módulo. Hay
varias alternativas para realizarla

Ejemplo: circuito votador



► Expresión lógica:
 $z = ab + ac + bc$

```
module votador (input a,b,c,output z);  
  
    assign z= (a & b) | (a & c) | (b & c);  
  
endmodule
```

Tipos de descripciones

- **RTL:** Una funcionalidad de abstracción de hardware escrita con bloques 'always' y sentencias 'assign' que son sintetizables.
- **Behavioural:** Recrea la funcionalidad requerida pero no necesariamente de forma sintetizable. No existen reglas estrictas siempre y cuando el código genere el comportamiento deseado. Se suele buscar una descripción simple y legible.
- **Gate level:** Lógica descrita por puertas y módulos exclusivamente. Sin bloques 'always' ni sentencias 'assign'. Es una representación real de las puertas en el hardware.

Tipos de descripciones

Descripción funcional

Modela circuitos combinaciones.

Consiste en asignaciones de las salidas de manera continua utilizando **assign**.

Todas las sentencias **assign** se ejecutan de manera concurrente.

```
module votador(input a,b,c, output z);  
  
    assign z = a&b | a&c | b&c;  
  
endmodule
```

Tipos de descripciones

Descripción procedimental

Permite el uso de estructuras de control

La descripción es algorítmica, igual que el software

Facilita la creación de funciones complejas

Se basa en la sentencia **always**

```
module votador(  
    input a,b,c,  
    output reg z)  
  
    always @(a,b,c)  
        if(a==1)  
            if(b==1 || c==1)  
                z=1;  
            else  
                z=0;  
        else  
            if(b==1 && c==1)  
                z=1;  
            else  
                z=0;  
  
endmodule
```

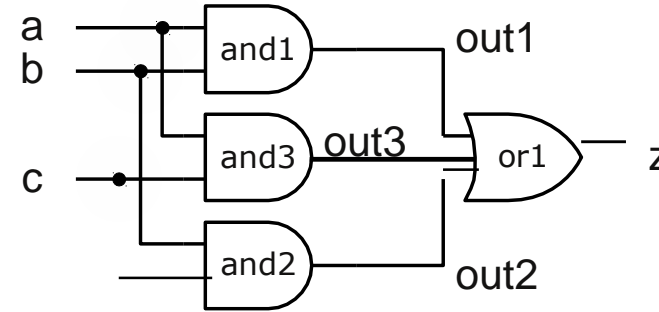
Tipos de descripciones

Descripción estructural

Se conectan módulos que ya están definidos previamente

Las puertas lógicas básicas ya están predefinidas en Verilog

Es muy útil para la interconexión de los módulos que se creen



```
module votador(
    input a,b,c,
    output z)

    wire out1,out2,out3;

    and and1(out1,a,b);
    and and2(out2,b,c);
    and and3(out3,a,c);
    or or1(z,out1,out2,out3);

endmodule
```


Tipos de descripciones

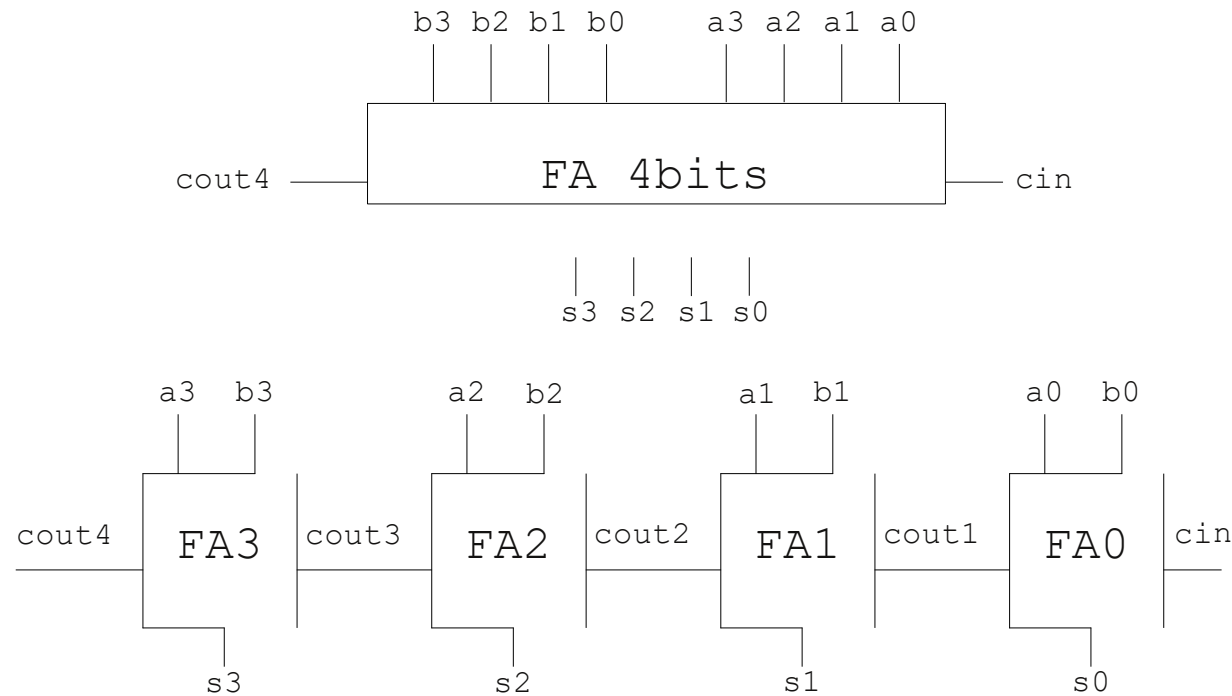
Todas las sentencias **assign** y **always** se ejecutan de manera concurrente.

La descripción estructural se utiliza para la interconexión de los diferentes módulos que se creen.

Las descripciones estructurales conforman la jerarquía del sistema que se está diseñando.

Tipos de descripciones

Ejemplo de descripción de un FULL-ADDER 4 bits a partir de varios FULL-ADDER de un bit



Tipos de descripciones

Pasos:

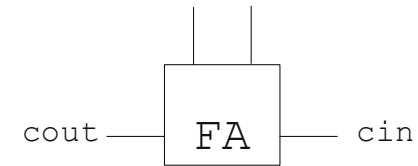
1. Descripción de un módulo para el FULL-ADDER de un bit.
2. Descripción de un módulo donde se utilizan 4 FULL-ADDER de un bit y se interconectan los cables de los módulos.

Tipos de descripciones

a b

Descripción del FA de un bit

```
module fulladder(  
    input a,  
    input b,  
    input cin,  
    output s,  
    output cout);  
  
    assign s = a ^ b ^ cin;  
    assign cout = a & b | a & cin | b & cin;  
endmodule
```



			s		
cin	a	b	cout	s	
0	0	0	0	0	
0	0	1	0	1	
0	1	0	0	1	
0	1	1	1	0	
1	0	0	0	1	
1	0	1	1	0	
1	1	0	1	0	
1	1	1	1	1	

$$s = a \oplus b \oplus c$$

$$cout = a \cdot b + a \cdot cin + b \cdot cin$$

Tipos de descripciones

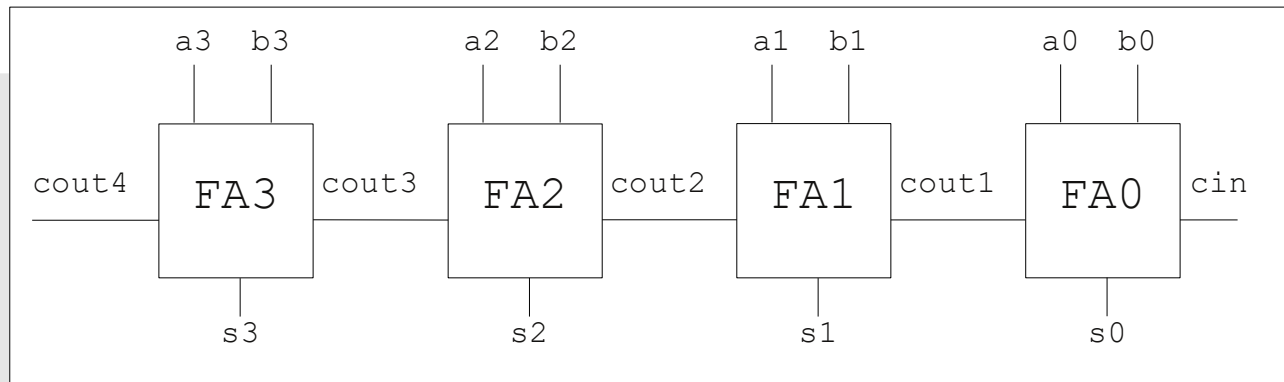
Unión de 4 FULL-ADDER: **conexión posicional.**

```
module fulladder4(  
  input [3:0] a,  
  input [3:0] b,  
  input cin,  
  output [3:0] s,  
  output cout4);
```

```
  wire cout1,cout2,cout3;
```

```
  fulladder fa0 (a[0], b[0], cin, s[0], cout1);  
  fulladder fa1 (a[1], b[1], cout1, s[1], cout2);  
  fulladder fa2 (a[2], b[2], cout2, s[2], cout3);  
  fulladder fa3 (a[3], b[3], cout3, s[3], cout4);
```

```
endmodule
```



Tipos de descripciones

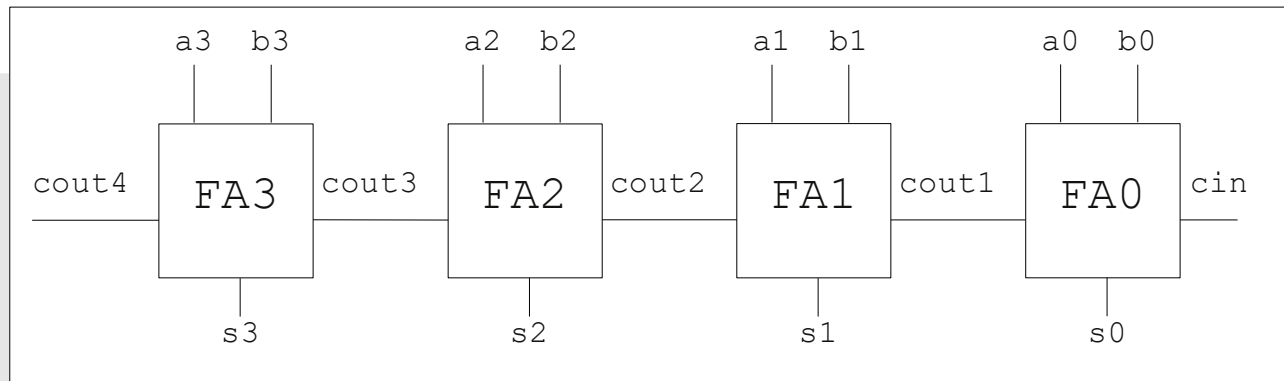
Unión de 4 FULL-ADDER: **conexión nombrada.**

```
module fulladder4(  
  input [3:0] a,  
  input [3:0] b,  
  input cin,  
  output [3:0] s,  
  output cout4);
```

```
  wire cout1,cout2,cout3;
```

```
  fulladder fa0 (.a(a[0]), .b(b[0]), .cin(cin), .s(s[0]), .cout(cout1));  
  fulladder fa1 (.a(a[1]), .b(b[1]), .cin(cout1), .s(s[1]), .cout(cout2));  
  fulladder fa2 (.a(a[2]), .b(b[2]), .cin(cout2), .s(s[2]), .cout(cout3));  
  fulladder fa3 (.a(a[3]), .b(b[3]), .cin(cout3), .s(s[3]), .cout(cout4));
```

```
endmodule
```



Tipos de señales

Existen dos tipos básicos de señales

wire: corresponden a cables físicos que interconectan componentes, por tanto, no tienen memoria.

reg: (también llamada variable). Son utilizados para almacenar valores, tienen memoria.

Los tipos (reg) se utilizan para modelar el almacenamiento de datos

Todas las asignaciones que se realicen dentro de un procedimiento (always) deben ser sobre una señal tipo reg

Puertos de entrada/salida

Cuando se declaran módulos se puede especificar si un puerto es tipo **wire** o **reg**

Si no se indica nada es por defecto wire

Los cables (wire) son utilizados con la sentencia **assign**

Los registros (reg) son asignados en los procedimientos

```
module mi_circuito (  
    input wire x,  
    input z,  
    output reg mem  
);  
...  
endmodule
```

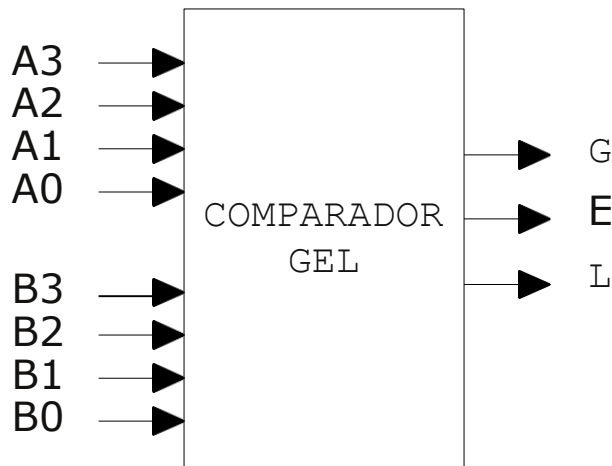

Arrays

Los arrays son agrupaciones de bits, motivos:

Los puertos de entrada/salida se agrupan (buses) para trabajar con mayor comodidad

Los registros pueden ser de varios bits

Sintaxis: [M:N]



```
module comparador_gel (
    input wire [3:0] a,
    input [3:0] b,
    output g,e,l
);
    ...
endmodule
```

Sintaxis básica

Literales

Sentencia assign

Sentencia always

Expresiones y operadores

Sentencias condicionales

Sintaxis básica

Verilog distingue entre mayúsculas y minúsculas

Se pueden escribir comentarios:

Comentario en línea: precedido de doble barra “//”

```
wire a; // Este cable se conecta con f2
```

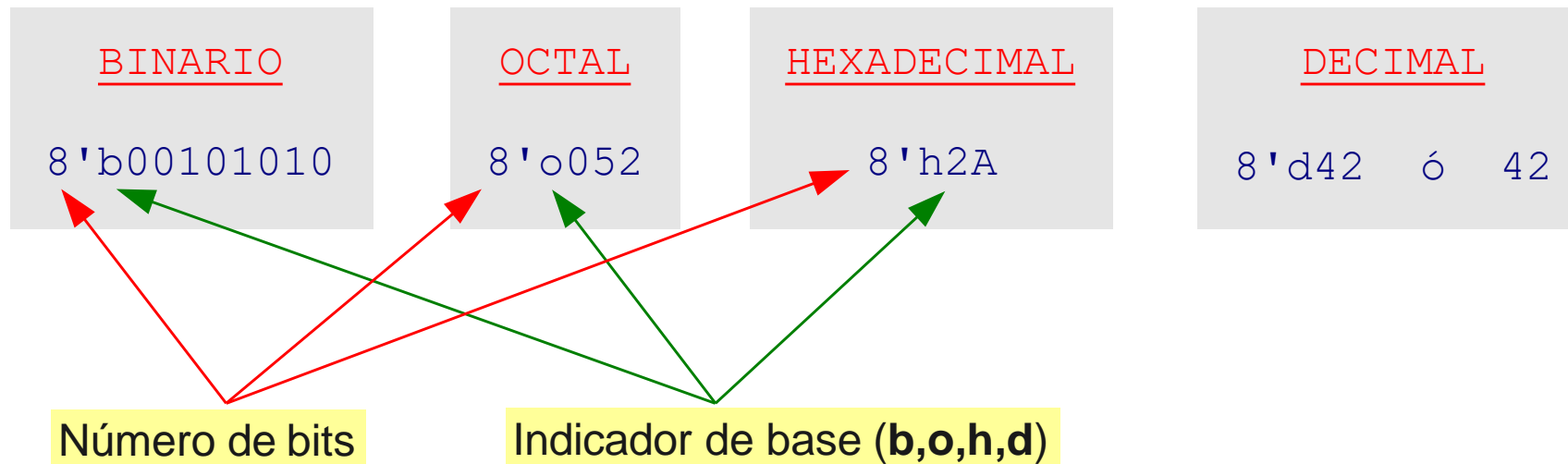
Comentario de varias líneas: comienza con /* y termina con */

```
/* Este cable conecta muchos componentes  
y necesito varias lineas para explicarlo  
correctamente */  
  
wire a;
```

Sintaxis básica

Literales: Se puede expresar en varios formatos

Ejemplo: “00101010” en binario



Sintaxis básica

Ejemplo de literales: Circuito que siempre tiene sus salidas a uno

```
module siempre_uno (  
    input  x,  
    output [7:0] salida1,  
    output [3:0] salida2  
);  
  
    assign salida2 = 4'b1111;  
    assign salida1 = 8'hFF;  
  
endmodule
```

Sentencia assign

Todas las sentencias **assign** se ejecutan de manera concurrente

En el ejemplo la salida f2 es equivalente a:

```
assign f2 = x & y & z;
```

```
module otro_ejemplo (  
    input  x, y, z,  
    output f1, f2  
);  
  
    assign f1 = x & y;  
    assign f2 = f1 & z;  
  
endmodule
```

Sentencia always

Un bloque always se ejecuta concurrentemente con los demás bloques always y assign que hay en la descripción HDL

Los bloques always tienen una **lista de sensibilidad**:

La lista de sensibilidad consiste en una lista de señales.

El código del bloque always se ejecuta sólo si cambia alguna de las señales de la lista de sensibilidad.

La sintaxis es:

```
always @(a,b)  
    c = a | b;
```

Sentencia always

Una sentencia **always** suele contener varias sentencias, en cuyo caso, debe utilizar un bloque “begin” ... “end”

Los bloques **begin/end** se utilizan para agrupar un conjunto de sentencias.

Son ampliamente utilizados

```
module (input a, b, c, d
        output reg f1,
        output reg f2);
  always @(a,b,c,d)
    begin
      f1 = a | b;
      f2 = c & d;
    end
endmodule
```


Sentencia always

- ▶ Importante regla general sobre la lista de sensibilidad:
 - ▶ Siempre que se esté describiendo un componente combinacional, se debe incluir en la lista de sensibilidad todas las entradas del componente
 - ▶ Se puede simplificar la sintaxis mediante: **always @(*)**

```
module (input a, b, c, d,  
        input e, f, g, h,  
        output f1, f2);  
  always @(a,b,c,d,e,f,g,h)  
  begin  
    ...  
  end  
endmodule
```

=

```
module (input a, b, c, d,  
        input e, f, g, h,  
        output f1, f2);  
  always @(*)  
  begin  
    ...  
  end  
endmodule
```

Operadores

Operadores a nivel de bits:

Operador	Ejemplo de código Verilog
&	<code>c = a&b; // Operación AND de todos los bits</code>
	<code>c = a b; // Operación OR de todos los bits</code>
^	<code>c = a^b; // Operación XOR de todos los bits</code>
~	<code>b = ~a; // Inversión de todo los bits</code>

Estos operadores trabajan con todos los bits.

Si la variable es de un único bit operan como los operadores del álgebra de conmutación.

Operadores

Más operadores a nivel de bits

Operador	Ejemplo de código Verilog
----------	---------------------------

<code>~&</code>	<code>d = a ~& b; // Operador NAND a nivel de bits</code>
---------------------	---

<code>~ </code>	<code>d = a ~ b; // Operador NOR a nivel de bits</code>
-----------------	--

<code>~^</code>	<code>d = a ~^ b; // Operador EXNOR a nivel de bits</code>
-----------------	--

Operadores

Ejemplo de uso de operadores a nivel de bits:
Módulo que realiza el complemento a uno de una palabra de 16 bits

```
module complemento_a1(  
    input [15:0] palabra,  
    output [15:0] complemento_1);  
  
    assign complemento_1 = ~palabra;  
  
endmodule
```

Operadores

Operadores relacionales: devuelven 1 si es verdadera la condición

Operador	Ejemplo de código en Verilog
<	<code>a < b; //¿Es a menor que b?</code>
>	<code>a > b; //¿Es a mayor que b?</code>
>=	<code>a >= b; //¿Es a mayor o igual que b?</code>
<=	<code>a <= b; //¿Es a menor o igual que b?</code>
==	<code>a == b; //Devuelve 1 si a es igual que b</code>
!=	<code>a != b; //Devuelve 1 si a es distinto de b</code>

Operadores

Operadores lógicos: No confundirlos con los operadores a nivel de bits.

Operador	Ejemplo de código Verilog
&&	<code>a && b; // Devuelve 1 si a y b son verdaderos</code>
	<code>a b; // Devuelve 1 si a ó b es verdadero</code>
!	<code>!a; // Devuelve 1 si a es falso ó 0 si a es verdadero</code>

Operadores

Operadores aritméticos

Operador	Ejemplo de código Verilog
*	<code>c = a * b; // Multiplicación</code>
/	<code>c = a / b; // División</code>
+	<code>sum = a + b; // Suma de a+b</code>
-	<code>resta = a - b; // Resta</code>

Operadores

Otros operadores

Operador	Ejemplos en código Verilog
<<	<pre>b = a << 1; //Desplazamiento a la //izq. de un bit</pre>
>>	<pre>b = a >> 1; //Desplazamiento a la //der. de un bit</pre>
?:	<pre>c = sel ? a : b; // si sel es cierto //entonces c = a,sino entonces c = b</pre>
{}	<pre>{a, b, c} = 3'b101; // Conatenación: // Asigna una palabra a bits // individuales: a=1, b=0 y c=1</pre>

Aparte de estos operadores existen más que se pueden encontrar en la bibliografía

Sentencias condicionales

- ▶ La sentencia condicional más común es la sentencia: **if ... else ...**

```
if ( a > 0 )  
    Sentencia  
else  
    Sentencia
```

```
if ( a == 0 )  
    Sentencia  
else if( b != 1 )  
    Sentencia
```

- ▶ Sólo se pueden usar en procedimientos “always”
- ▶ En las condiciones de esta sentencia se pueden utilizar todos los operadores lógicos y relacionales

Sentencias condicionales

Si hay más de una sentencia tras una condición, hay que utilizar bloques “begin” ... “end”

```
always @(a)
begin
    if ( a > 0 )
        f1 = 1;
        f2 = 1;
    else
        f1 = 0;
end
```

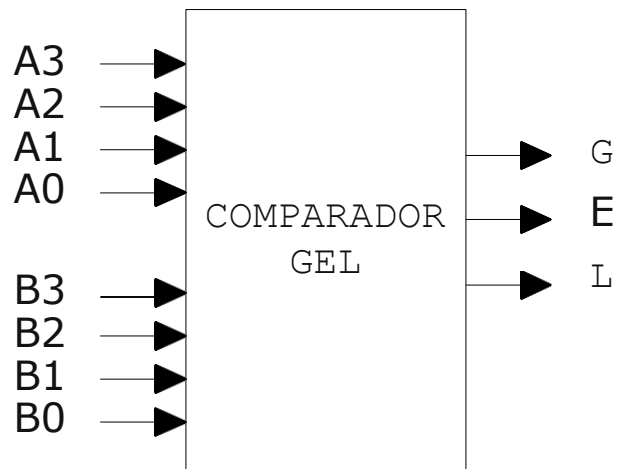
ERROR

```
always @(a)
begin
    if ( a > 0 )
        begin
            f1 = 1;
            f2 = 1;
        end
    else
        f1 = 0;
end
```

Correcto

Sentencias condicionales

Ejemplo de comparador GEL



```
module comparador_gel(  
    input [3:0] a,  
    input [3:0] b,  
    output g, // si a < b => (g,e,l) = (0,0,1)  
    output e, // si a = b => (g,e,l) = (0,1,0)  
    output l);  
  
    reg g, e, l;  
    always @(a, b)  
        begin  
            g = 0;  
            e = 0;  
            l = 0;  
            if (a > b)  
                g = 1;  
            else if (a < b)  
                l = 1;  
            else  
                e = 1;  
        end  
endmodule
```

Sentencias condicionales

Sentencia **case**

Se utiliza dentro de un proceso “always”

Si alguno de los casos tiene más de una sentencia hay que utilizar un bloque “begin” ... “end”

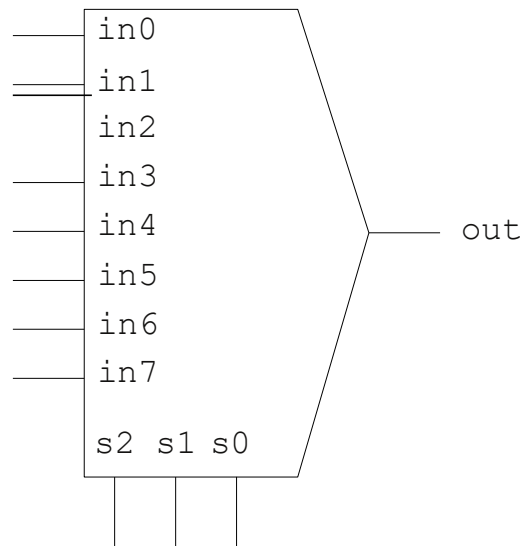
Se puede utilizar **default** para los casos no enumerados

```
reg [1:0] x;  
always @(x)  
begin  
    case(x)  
        0:  
            salida_1 = 1;  
        1:  
            begin  
                salida_1 = 1;  
                salida_2 = 0;  
            end  
        2:  
            salida_2 = 1;  
        3:  
            salida_1 = 0;  
    endcase  
end
```

Sentencias condicionales

Multiplexor 8:1

Ejemplo de acceso a
elementos individuales de
un array



```
module mux8_1(  
    input [2:0] s,  
    input [7:0] in,  
    output out);  
  
    reg out;  
    always @(s, in)  
        case (s)  
            3'h0: out = in[0];  
            3'h1: out = in[1];  
            3'h2: out = in[2];  
            3'h3: out = in[3];  
            3'h4: out = in[4];  
            3'h5: out = in[5];  
            3'h6: out = in[6];  
            default: out = in[7];  
        endcase  
endmodule
```

Bloque II: Índice

- Sintaxis II
- Biestables
- Máquinas de estados
- Registros
- Contadores

Sintaxis II

Definición de constantes

Operador de concatenación

Lista de sensibilidad con detección de flancos

Asignaciones bloqueantes / no bloqueantes

Sintaxis II

- Dentro de un módulo se pueden definir constantes utilizando **parameter**
- Es útil en la definición de máquinas de estados

Ejemplo:

```
parameter uno_con_tres_bits = 3'b001,  
           ultimo = 3'b111;  
  
reg [2:0] a;  
  
a = ultimo;
```


Sintaxis II

El operador concatenar se utiliza para agrupar señales para que formen un array

Sintaxis: {señal, señal,}

Ejemplo:

Detector del número 3

```
module concatena(  
    input a,b,c,  
    output reg igual_a_3  
);  
  
    always @(*)  
        case ({a,b,c})  
            3'b011:  
                igual_a_3 = 1;  
            default:  
                igual_a_3 = 0;  
        endcase  
endmodule
```

Sintaxis II

Detección de flanco

Sirve para que un proceso sólo se ejecute en determinados flancos de reloj de una o varias señales de entrada.

Se indica en la lista de sensibilidad de un proceso mediante un prefijo a la señal:

El prefijo **posedge** detecta el flanco de subida

El prefijo **negedge** detecta el flanco de bajada

Sintaxis II

Ejemplo de detección de flanco negativo de un reloj

```
module detector_flanco(  
    input clk,  
    output reg z);  
  
    always @(negedge clk)  
        ....  
endmodule
```

Sintaxis II

Asignamiento **bloqueante** signo =

Si en un proceso se desea que la salida cambie inmediatamente, se debe utilizar una asignación bloqueante.

Esto modela una salida combinacional.

Importa el orden en que se efectúan las asignaciones bloqueantes puesto que las acciones en un proceso se ejecutan secuencialmente

Sintaxis II

Asignamiento **no bloqueante** signo \leq

La asignación no bloqueante modela las escrituras en flip-flops.

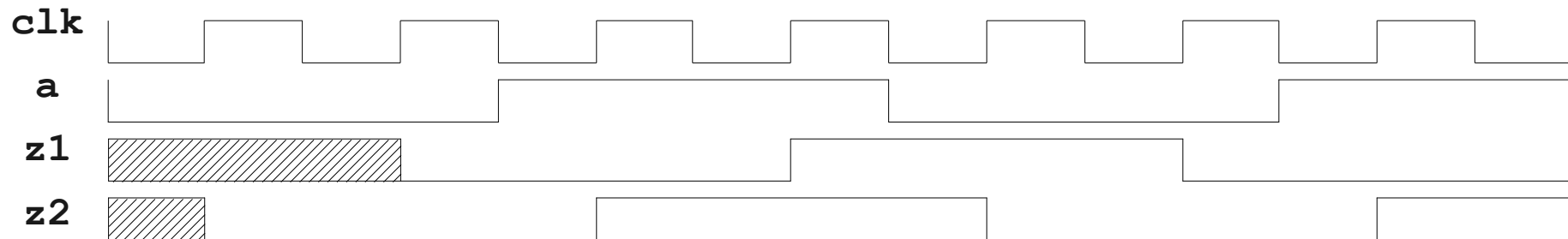
Se calculan primero los valores de la derecha de la asignación de todas las asignaciones \leq , tras esto, se asignan todas simultáneamente.

Cuando se tiene una serie de asignaciones no bloqueantes, no importa el orden en que son escritas.

Sintaxis II

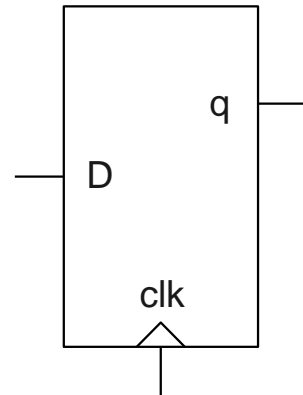
```
module no_bloqueante(input a,clk,  
output reg z1);  
reg q;  
always @(posedge clk)  
begin  
    q <= a;  
    z1 <= q;  
end  
endmodule
```

```
module bloqueante(input a,clk,  
output reg z2);  
reg q;  
always @(posedge clk)  
begin  
    q = a;  
    z2 = q;  
end  
endmodule
```

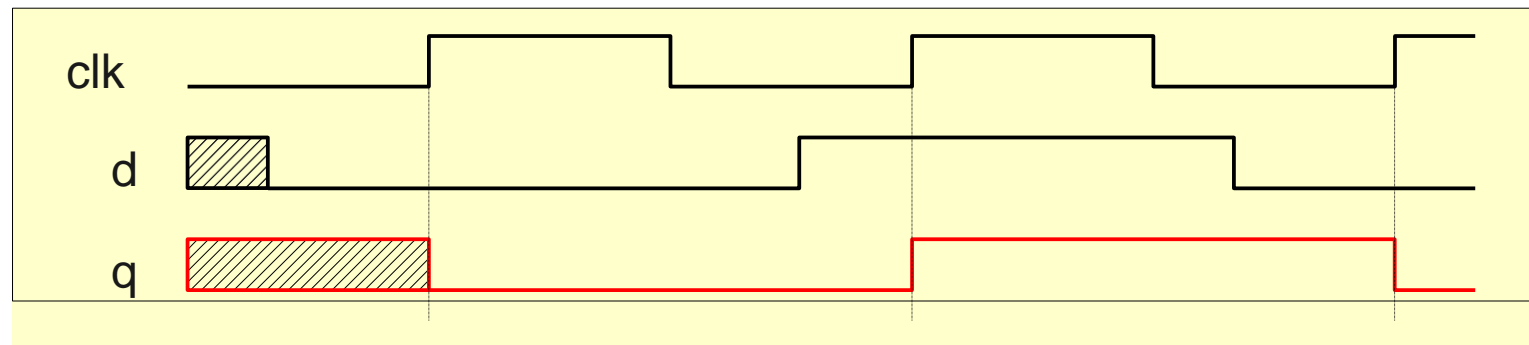


Biestables

Ejemplo de biestables:

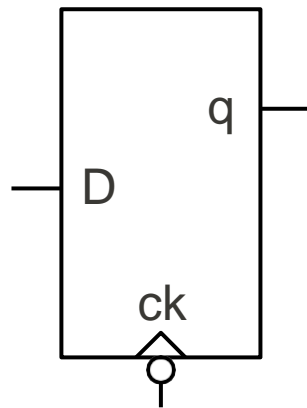


```
module biestable_d(  
    input clk,d,  
    output reg q);  
  
    always @ (posedge clk)  
        q <= d;  
  
endmodule
```

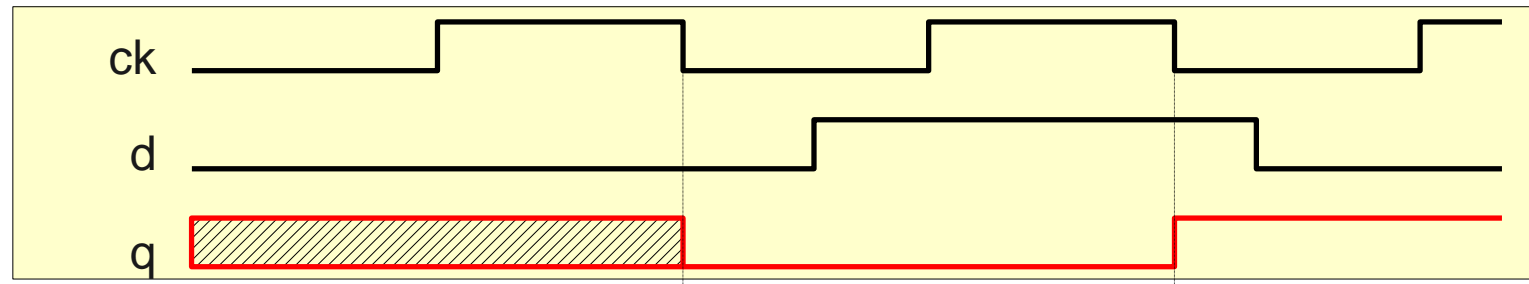


Biestables

Ejemplo de biestable D disparado en flanco negativo

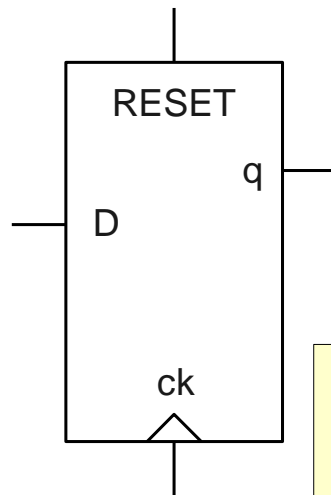


```
module biestable_d(  
    input ck,d,  
    output reg q);  
  
    always @ (negedge ck)  
        q <= d;  
  
endmodule
```

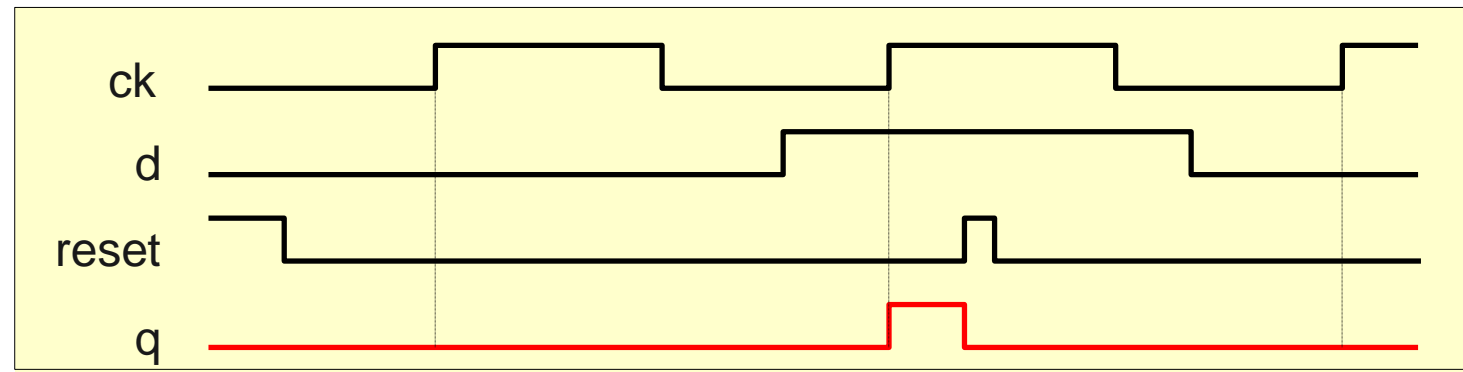


Biestables

Biestable D con reset asíncrono

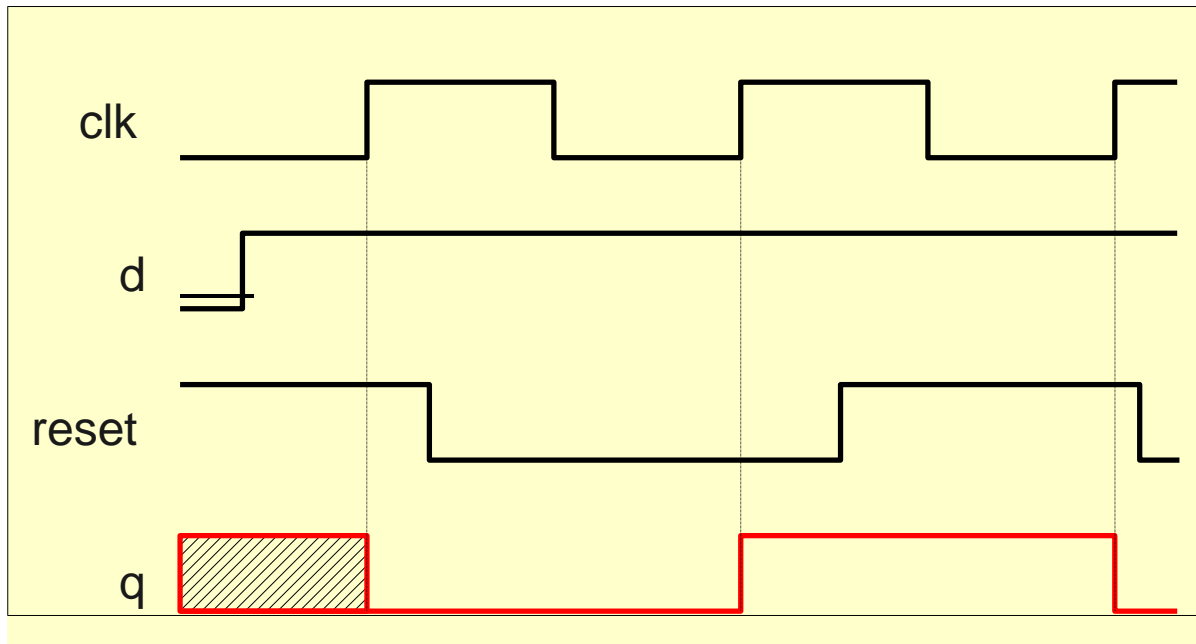


```
module biestable_d(  
    input ck,d,reset,  
    output reg q);  
  
    always @ (posedge ck or posedge reset)  
        if (reset)  
            q <= 1'b0;  
        else  
            q <= d;  
  
endmodule
```



Biestables

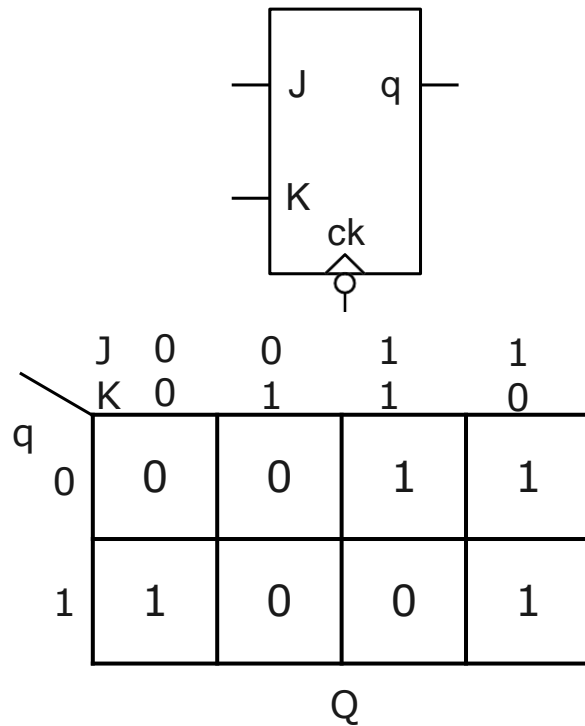
Biestable D con reset síncrono



```
module biestable_d(  
    input clk,d,reset,  
    output reg q);  
  
    always @ (posedge clk)  
        if (reset)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

Biestables

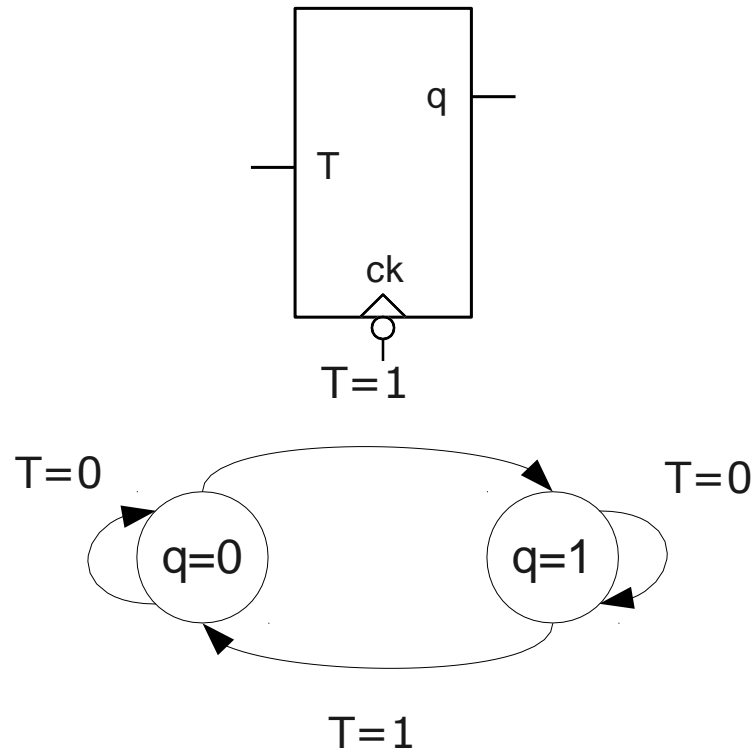
Biestable JK



```
module jk_flip_flop (  
    input      ck,  
    input      j,  
    input      k,  
    output reg q);  
  
    always @(negedge ck)  
        case ({j,k})  
            2'b11 : q <= ~q;      // Cambio  
            2'b01 : q <= 1'b0;    // reset.  
            2'b10 : q <= 1'b1;    // set.  
            2'b00 : q <= q;       //  
        endcase  
  
endmodule
```

Biestables

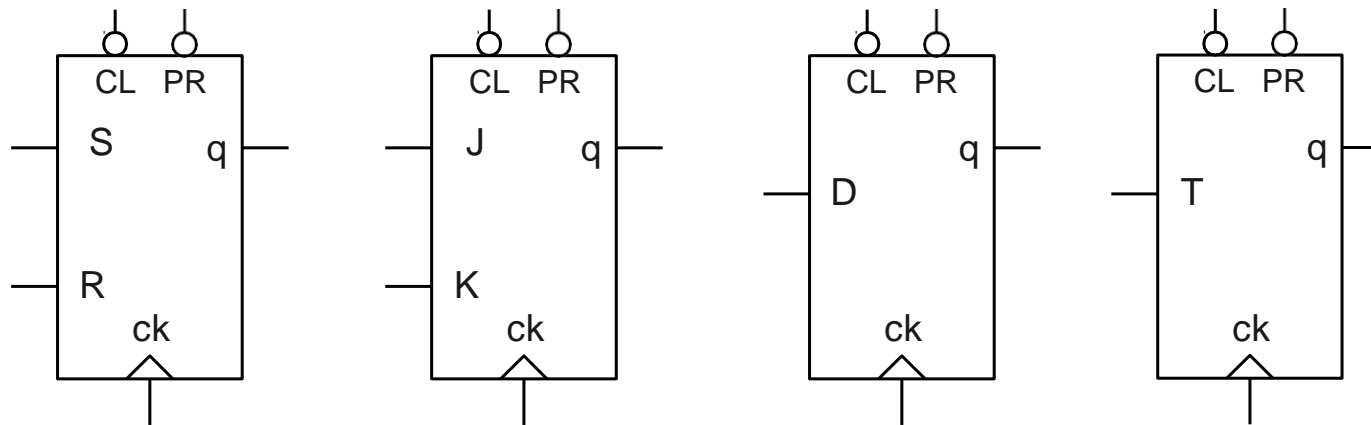
Biestable T



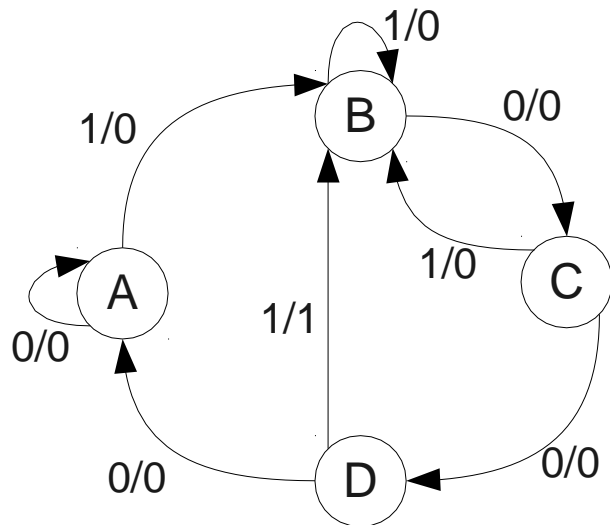
```
module biestable_t(  
    input ck,  
    input t,  
    output reg q);  
  
    always @(negedge ck)  
        if (t == 1)  
            q <= ~q;  
  
endmodule
```

Biestables

Ejercicios: Realice los siguientes biestables, sabiendo las señales CL y PR son síncronas



Máquinas de estado



Se utilizará una estructura general del código en la que hay 2 procesos

Uno de asignación de siguientes estados

Otro de calculo de siguiente estado y salidas

Máquinas de estado

```
module mi_diagrama_de_estados(  
    input  LISTA_DE_ENTRADAS,  
    output reg LISTA_DE_SALIDAS);  
  
    // DEFINICION Y ASIGNACIÓN DE ESTADOS  
    parameter LISTA_DE_ESTADOS  
  
    // VARIABLES PARA ALMACENAR EL ESTADO PRESENTE Y SIGUIENTE  
    reg [N:0] current_state, next_state;  
  
    // PROCESO DE CAMBIO DE ESTADO  
    always @(posedge clk or posedge reset)  
        .....  
    // PROCESO SIGUIENTE ESTADO Y SALIDA  
    always @(current_state, LISTA_DE_ENTRADAS)  
        .....  
endmodule
```

Máquinas de estado

En la estructura general hay que completar 4 partes de código:

1. Definición y asignación de estados, según el número de estados utilizaremos mas o menos bits.
2. Definición de registros para almacenar el estado actual y el siguiente. Deben ser del mismo tamaño en bits que el utilizado en el punto anterior
3. Proceso de cambio de estado: Siempre es el mismo código
4. Proceso de cálculo de siguiente estado y salida: Hay que rellenar el código correspondiente las transiciones del diagrama de estados

Máquinas de estado

```
module maquina_estados(  
  input x, clk, reset,  
  output reg z);
```

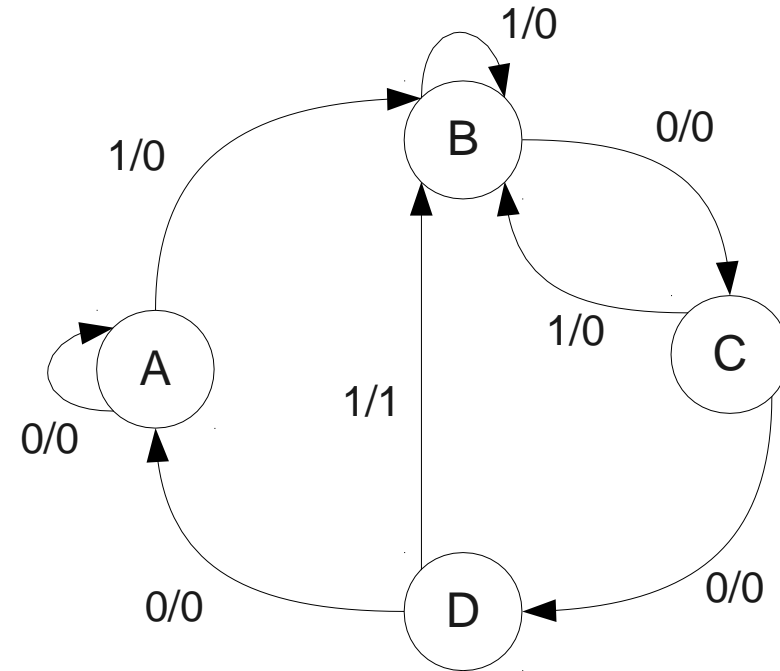
```
  parameter A = 2'b00,  
            B = 2'b01,  
            C = 2'b10,  
            D = 2'b11;
```

Asignación
de estados

```
  reg [1:0] current_state,next_state;
```

```
  always @(posedge clk, posedge reset)  
  begin  
    if(reset)  
      current_state <= A;  
    else  
      current_state <= next_state;  
  end
```

SIGUE ->



Proceso
de asignación
de siguiente estado

Máquinas de estado

- El proceso de calculo del siguiente estado y salida se realiza con una única sentencia **case**
 - La sentencia case debe contemplar todos los estados del diagrama de estados
 - Antes de la sentencia **case** se recomienda establecer por defecto a cero todas las salidas.

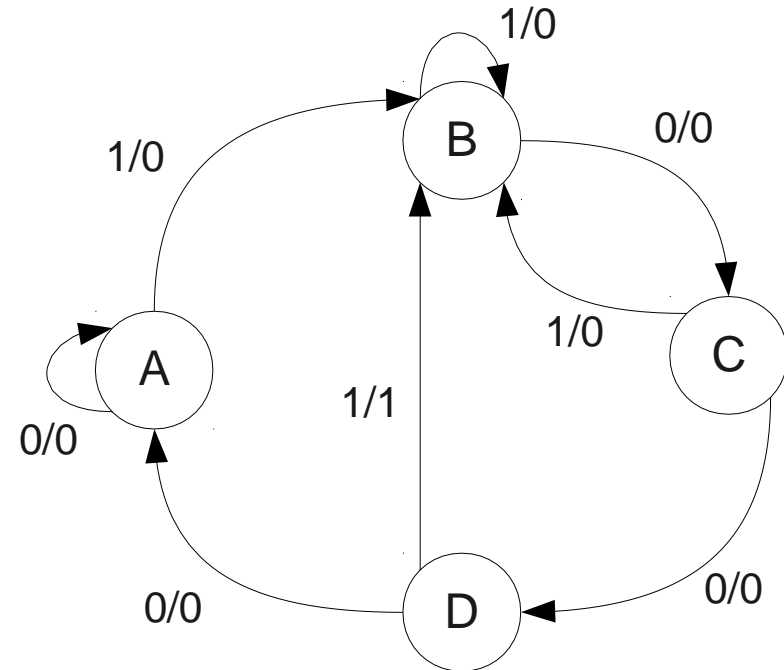
Máquinas de estado

```
always @(current_state,x)
begin
  z = 0;
  case(current_state)
    A:
      if(x == 1)
        next_state = B;
      else
        next_state = A;
    B:
      if(x == 1)
        next_state = B;
      else
        next_state = C;
```

Salida a cero

Estado A

Estado B

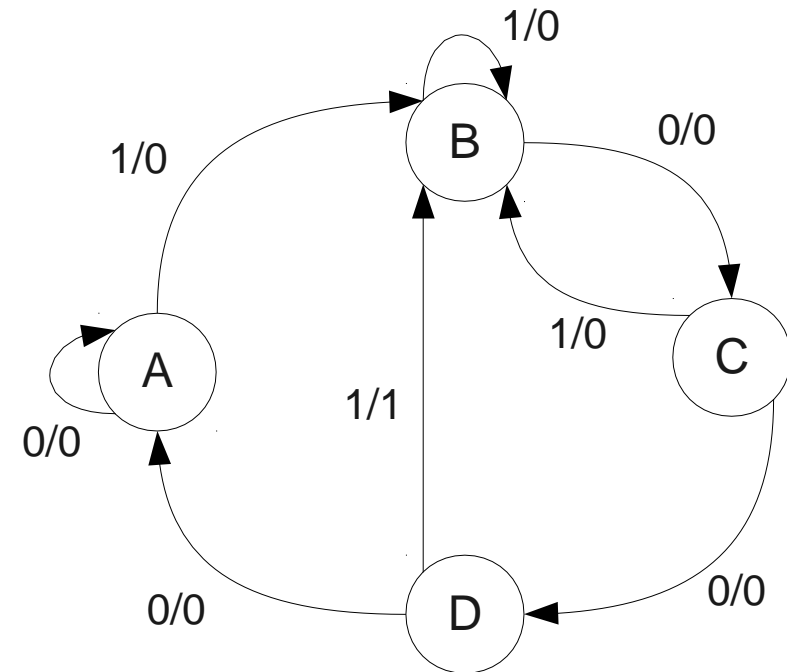


Máquinas de estado

```
C:
  if(x == 1)
    next_state = B;
  else
    next_state = D;
D:
  if(x == 1)
    begin
      z = 1;
      next_state = B;
    end
  else
    next_state = A;
  endcase
end
endmodule
```

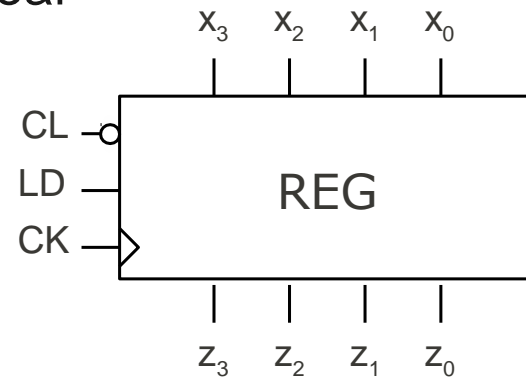
Estado C

Estado D



Registros

Registro con carga en paralelo y clear

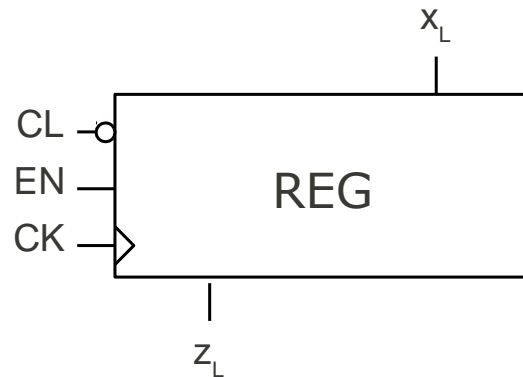


CL, LD	Operation	Type
0x	$q \leftarrow 0$	async.
11	$q \leftarrow x$	sync.
10	$q \leftarrow q$	sync.

```
module registro(  
    input ck,  
    input cl,  
    input ld,  
    input [3:0] x,  
    output [3:0] z  
);  
  
    reg [3:0] q;  
  
    always @(posedge ck, negedge cl)  
        if (cl == 0)  
            q <= 0;  
        else if (ld == 1)  
            q <= x;  
  
    assign z = q;  
  
endmodule
```

Registros

Registro de desplazamiento

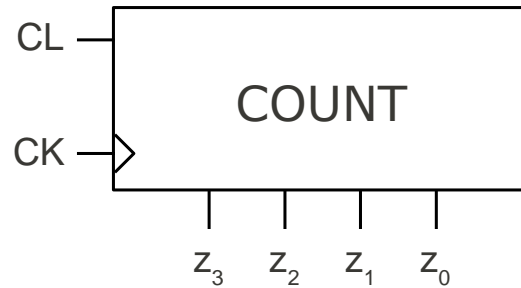


CL, EN	Operation	Type
0x	$q \leftarrow 0$	async.
11	$q \leftarrow \text{SHL}(q)$	sync.
10	$q \leftarrow q$	sync.

```
module reg_shl(  
    input ck,  
    input cl,  
    input en,  
    input x1,  
    output z1  
);  
  
    reg [3:0] q;  
  
    always @(posedge ck, negedge cl)  
        if (cl == 0)  
            q <= 0;  
        else if (en == 1)  
            q <= {q[2:0], x1};  
  
    assign z1 = q[3];  
  
endmodule
```

Contadores

Contador ascendente con clear

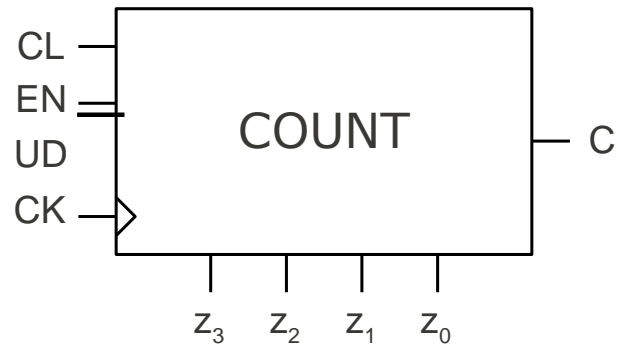


CL	Operation	Type
1	$q \leftarrow 0$	async.
0	$q \leftarrow q+1 \bmod 16$	sync.

```
module count_mod16(  
    input ck,  
    input cl,  
    output [3:0] z);  
  
    reg [3:0] q;  
  
    always @(posedge ck, posedge cl)  
        if (cl == 1)  
            q <= 0;  
        else  
            q <= q + 1;  
  
    assign z = q;  
  
endmodule
```

Contadores

Contador ascendente/
descendente con clear



CL, EN, UD	Operation	Type
1xx	$q \leftarrow 0$	async.
00x	$q \leftarrow q$	sync.
010	$q \leftarrow q+1 \bmod 16$	sync.
011	$q \leftarrow q-1 \bmod 16$	sync.

```
module rev_counter1(  
    input ck,  
    input cl,en, ud,  
    output [3:0] z, output c);  
  
    reg [3:0] q;  
  
    always @(posedge ck, posedge cl)  
        begin  
            if (cl == 1)  
                q <= 0;  
            else if (en == 1)  
                if (ud == 0)  
                    q <= q + 1;  
                else  
                    q <= q - 1;  
        end  
  
    assign z = q;  
    assign c = ud ? ~(|q) : &q;  
endmodule
```


Índice

1. Introducción a la Práctica 4
2. ¿Qué es una FPGA?
3. Verilog HDL
4. Test

Testbench architecture

Un testbench consiste en utilizar código Verilog no sintetizable para generar entradas al diseño y comprobar que las salidas son correctas.

La arquitectura típica de un testbench simple contiene los siguientes elementos:

- Estímulos: Genera las señales de entrada para el diseño de la FPGA.
- Design/Device Under Test (DUT): Es el propio diseño a probar.
- Comprobación de resultados: Compara los resultados con valores de referencia para comprobar si son correctos.

Testbench example: Half adder

Instancia DUT

```
// half_adder_procedural_tb.v

`timescale 1 ns/10 ps // time-unit = 1 ns, precision = 10 ps

module half_adder_procedural_tb;

    reg a, b;
    wire sum, carry;

    // duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
    localparam period = 20;

    half_adder UUT (.a(a), .b(b), .sum(sum), .carry(carry));

    reg clk;

    // note that sensitive list is omitted in always block
    // therefore always-block run forever
    // clock period = 2 ns
    always
    begin
        clk = 1'b1;
        #20; // high for 20 * timescale = 20 ns
        clk = 1'b0;
        #20; // low for 20 * timescale = 20 ns
    end
```

```
always @(posedge clk)
begin
    // values for a and b
    a = 0;
    b = 0;
    #period; // wait for period
    // display message if output not matched
    if(sum != 0 || carry != 0)
        $display("test failed for input combination 00");
    a = 0;
    b = 1;
    #period; // wait for period
    if(sum != 1 || carry != 0)
        $display("test failed for input combination 01");
    a = 1;
    b = 0;
    #period; // wait for period
    if(sum != 1 || carry != 0)
        $display("test failed for input combination 10");
    a = 1;
    b = 1;
    #period; // wait for period
    if(sum != 0 || carry != 1)
        $display("test failed for input combination 11");
    a = 0;
    b = 1;
    #period; // wait for period
    if(sum != 1 || carry != 1)
        $display("test failed for input combination 01");
    $stop; // end of simulation
end
endmodule
```

Generación de estímulos y comprobación de resultados en un mismo bloque 'always'