Tecnología y Organización de Computadores

Cómo diseñar circuitos para el mercado y aprender la importancia del diseño en el rendimiento final del dispositivo

Alejandro Barrachina Argudo

Grado en Ingeniería informática
Facultad de Informática
Universidad Complutense de Madrid

2020-2021

Índice general

1 | Modelado hardware con VHDL. Introducción a las FPGAs

1.1. Flujo de diseño

TODO

1.2. Lenguaje de descripción hardware

Hardware Description Language (HDL) es un lenguaje específicamente creado para el diseño de circuitos, tanto a nivel de puerta como a nivel e comportamiento. La estructura del lenguaje sugiere el diseño Hardware (HW).

Los HDL se usan para poder descubrir problemas en el diseño del HW antes de su implementación física. Como la complejidad de los sistemas electrónicos crece exponencialmente, es necesaria una herramienta que trabaje con el ordenador. Por último, gracias a estas herramientas más de una persona puede trabajar en el mismo proyecto

En este curso usaremos VHDL, esto nos permite hacer descripciones de la estructura del circuito con:

- Descomposición en sub-circuitos
- Interconexión de sub-circuitos
- comportamiento
- Estructural

También permite la especificación de un circuito utilizando formas familiares de lenguajes de programación y la simulación del circuito antes de su fabricación.

Un HDL tiene que ser capaz de simular el comportamiento real del HW sin que el programador necesite imponer restricciones.

Ejemplo 1:

t = 5ns	t=10ns
A = 0	A = 1
B = 1	B = 1
C = 0	C = 0

Descripción 1:

D = A and B;

S = D or C;

Descripción 2:

S = D or C;

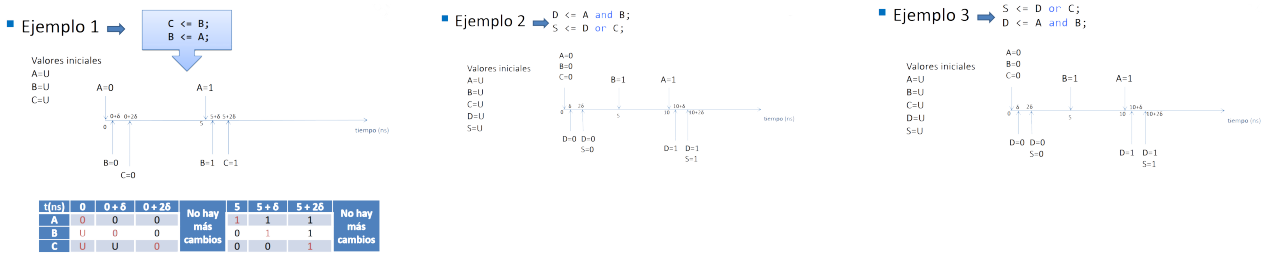
D = A and B;

¿Se obtiene el mismo resultado? No, en la descripción 1 en t = 10ns el valor de S será 1, mientras que en la descripción 2 seguirá siendo 0.

1.3. Simulación con VHDL

VHDL realiza la simulación siguiendo la técnica de **simulación por eventos discretos**, esto permite avanzar el tiempo a intervalos variables, en función de la planificación de ocurrencia de eventos. La simulación consta de tres fases:

- **Fase 0:** fase de inicialización donde las señales se les asignan unos valores iniciales y se pone el tiempo a cero. La asignación se hace rellenando una lista de eventos para el instante t = 0.
- **Fase 1:** todas las transiciones planificadas para ese tiempo se ejecutan
- **Fase 2:** las señales que se han modificado en consecuencia de las transiciones planificadas en el instante t se escriben en la lista de eventos planificándose para el instante t + δ, donde δ es un instante infinitesimal. Una vez acabado esta fase se vuelve a la fase 1 hasta que se termine la simulación.



t[ns]	0	0+δ	0+2δ		5	5+δ	5+2δ	
A	0	0	0	No hay más cambios	1	1	1	No hay más cambios
B	U	0	0		0	1	1	
C	U	U	0		0	0	1	

Pregunta ejemplo 3: ¿Qué pasaría si en 5 en vez de cambiar B cambia C? S=1 ya que D or C si c = 1 siempre es 1.

1.4. Estructura de un modelo VHDL

Un sistema digital está descrito por sus entradas y sus salidas, donde las salidas dependen de las entradas. Los modelos VHDL están formados por dos partes:

```
ENTITY nombre IS
    GENERIC (lista_de_parametros);
    PORT (
        A : IN BIT;
        B : OUT BIT);
END nombre;
```

Figura 1.4.1: Ejemplo Entity

```
ARCHITECTURE circuito OF nombre IS
    -- Signals
BEGIN
    -- Definición del circuito
END circuito; -- circuito
```

Figura 1.4.2: Ejemplo Architecture

La entity define externamente el circuito o subcircuito, especifica el nombre y el número de puertos, su tipo de datos y si son de entrada o salida, tiene toda la información necesaria para conectar tu circuito a otros circuitos. La architecture define internamente el circuito, sus señales internas, funciones, procedimientos, constantes, etc.

Cada architecture va asociada a una entity y se indica en la primera sentencia. Antes del begin se definen las señales, los tipos y los componentes

```
ENTITY F IS
    PORT (
        A, B : IN STD_LOGIC;
        Y : OUT STD_LOGIC);
END F;

ARCHITECTURE circuito OF F IS

    SIGNAL D, E : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL H : STD_LOGIC;

BEGIN

    -- Implementación del circuito

END circuito; -- circuito
```

Figura 1.4.3: Ejemplo Entity y Architecture

1.5. Elementos básicos de VHDL

Las sentencias concurrentes siempre se encuentran fuera de los PROCESS, aparecen en cualquier punto del programa después del begin de la arquitectura, es lógica combinacional pura y siempre tiene un ELSE final o un WHEN OTHERS.

```
signal_name <= valor_1 WHEN condition1 ELSE
    valor_2 WHEN condition2 ELSE
    valor_i WHEN condition1 ELSE
    otro_valor;
```

Figura 1.5.1: Ejemplo When-Else

```
WITH id SELECT
    signal_name <= valor1 WHEN valor_id1,
    valor2 WHEN valor_id2,
    valori WHEN valor_idi,
    otro_valor WHEN OTHERS;
```

Figura 1.5.2: Ejemplo With-Select-Else

Los procesos son sentencias concurrentes. Solo se ejecutan esas sentencias que se encuentran dentro del proceso si alguna de las señales de la lista de sensibilidad ha cambiado de valor. La lista de sensibilidad es opcional, si no existe el proceso se ejecuta infinitas veces. Es un código secuencial.

```
identificador : PROCESS (lista de sensibilidad)
BEGIN
    -- Sentencias secuenciales
    -- Sentencias condicionales
    -- Ecuaciones booleanas
END PROCESS; -- identificador
```

Figura 1.5.3: Ejemplo Process

Las secuencias condicionales van dentro de un process, si la lista de sensibilidad está incompleta, el comportamiento del process será incorrecto. Estas sentencias deben tener un caso por defecto.

```
ifthenelse : PROCESS (a, b, c)
BEGIN
    IF a = b THEN
        c <= a OR b;
    ELSIF a < b THEN
        c <= b;
    ELSE
        c <= "0000";
    END IF;
END PROCESS; -- ifthenelse
```

Figura 1.5.4: Ejemplo If-Then-Else

```
CASE expresion IS
    WHEN choice_1 => statements;
    WHEN choice_n => statemens;
    WHEN OTHERS => statemenst;
END CASE;
```

Figura 1.5.5: Ejemplo Case

Ejemplos de bucles en VHDL:

```
id : LOOP
    list_of_statements
    -- Use exit statement to
    get ount
END LOOP id; -- id
```

Figura 1.5.6: Loop simple

```
id : FOR var IN rango LOOP
    list_of_statements
END LOOP id; -- id
```

Figura 1.5.7: For Loop

```
id : WHILE condition LOOP
    list_of_statements
END LOOP id; --id
```

Figura 1.5.8: While Loop

Las sentencias wait se usan en procesos, procedimientos y funciones y, como indica su nombre, hace que haya un tiempo de espera. Puede ser de tres tipos:

- **wait for** <timeout clause, time delay>: wait for 10 ns;
- **wait until** <condition>: wait until clk = '1';
- **wait on** <sensitive clause, event>: wait on in1;

Atributos:

- S'**DELAYED**(t) es el valor de la señal S en el tiempo actual -t.
- S'**STABLE** es true si no hay ningún evento ocurriendo para S.
- S'**STABLE**(t) es true si no hay eventos ocurriendo en S durante t unidades de tiempo.
- S'**QUIET** es true si no hay eventos en ese ciclo de simulación para la señal S.
- S'**QUIET**(t) es true si no hay eventos en ese ciclo de simulación para la señal S durante t unidades de tiempo.
- S'**TRANSACTION** es un valor BIT, el inverso del valor previo de cada ciclo en que S está activo.
- S'**EVENT** es true si S tiene un evento en el ciclo de simulación.
- S'**ACTIVE** es true si S está activa durante el ciclo de simulación.
- S'**LAST_EVENT** es el tiempo desde el último evento en S.
- S'**LAST_ACTIVE** es el tiempo desde la última vez que S estuvo activa.
- S'**LAST_VALUE** es el valor anterior de S.

Tipos de datos:

- Tipos enteros y reales:
  - INTEGER
  - NATURAL
  - REAL
- Tipos físicos
  - TIME
  - RESISTANCE
- Tipos enumerados: conjuntos de posibles valores
  - BIT 0,1
  - BOOLEAN TRUE, FALSE
  - CHARACTER ASCII CHARACTERS
  - STD_LOGIC U, X, 0, 1, Z, W, L, H, -

Se pueden definir tipos personalizados

```
TYPE nuevo_tipo1 IS (v1, v2, v3, v4);
TYPE nuevo_tipo2 IS RANGE 0 TO 17;
SIGNAL A : nuevo_tipo1;
SIGNAL B, C : nuevo_tipo2;

-- Asignaciones a señales, dentro dde la arquitectura

A <= v1; -- Asignacion correcta
B <= 17; -- Asignacion correcta
C <= 18; -- Asignacion incorrecta

-- Los records son como Structs de C++

TYPE my_record IS
field_1 : STD_LOGIC_VECTOR(3 DOWNTO 0);
field_2 : STD_LOGIC;
END RECORD;

SIGNAL A, C : my_record;
SIGNAL B : STD_LOGIC_VECTOR(3 DOWNTO 0);

-- Asignaciones a "A", dentro de la arquitectura
A.field_1 <= B;
A.field_2 <= B(2);
C <= A; -- Se pueden hacer asignaciones directas o campo a campo
```

Figura 1.5.9: Ejemplo tipos personalizados

Arrays:

```
TYPE my_array IS ARRAY (3 DOWNTO 0) OF STD_LOGIC_VECTOR(3 DOWNTO 0);

SIGNAL A : my_array;
SIGNAL B : STD_LOGIC_VECTOR(7 DOWNTO 0);

-- Asignaciones a "A", dentro de la arquitectura

A(1) <= B(3 DOWNTO 0);
A(2) <= B(7 DOWNTO 4);

-- Tambien se pueden declarar constantes
CONSTANT M : INTEGER := 32;
SIGNAL A : STD_LOGIC_VECTOR(M - 1 DOWNTO 0);
```

Figura 1.5.10: Ejemplo Arrays

Un **componente** es una entidad que se describe dentro de una arquitectura

```
ARCHITECTURE arch_name OF entity_name IS
    COMPONENT component_name
        PORT ( < IO_ports_list >);
    END COMPONENT;
    --declaración de señales;
    begincomponent_i : component_name
    PORT MAP( < s1 >, < s2 >, ..., < sn >);
END arch_name;
```

Figura 1.5.11: Ejemplo Component

Generate: las secuencias de generación de componentes permiten crear una o más copias de un conjunto de interconexiones, lo cual facilita el diseño de circuitos mediante descripciones estructurales

```
COMPONENT comp
    PORT (
        x : IN BIT;
        y : OUT BIT);
END comp;

SIGNAL a, b : BIT_VECTOR(0 TO 7);

gen : FOR i IN 0 TO 7 GENERATE
    u : comp PORT MAP(a(i), b(i));
END GENERATE; -- gen
```

Figura 1.5.12: Ejemplo Generate

Un **paquete** son agrupaciones de tipos, funciones y procedimientos personalizados. Pueden ser incluidos en uno o varios diseños, normalmente se definen en ficheros .vhd dedicados.

```
PACKAGE nombre IS
    -- tipos personalizados
    -- constantes
    -- declaraciones de componentes
    -- funciones y procedimientos (SOLO declaraciones)
END nombre;

PACKAGE BODY nombre IS
    -- Funciones y procedimientos (cuerpo)
END nombre;
```

Figura 1.5.13: Ejemplo Package

1.6. Entidades parametrizables

Una entity puede tener una lista de parámetros. El valor por defecto de los parámetros genéricos es opcional, sin embargo, no se puede sintetizar un diseño con parámetros que no tengan asignado un valor.

Al instanciar este componente en otro fichero, hay que asignar un valor a n si y solo si no se le dio un valor por defecto en su declaración de entity.

Se puede utilizar la asignación (others=>0) o (others=>1) para asignar el valor 0...0 o 1...1 a una señal de longitud parametrizable.

1.7. Testbench de simulación

Para conocer si nuestro diseño funciona correctamente tendremos que introducir unos estímulos en las entradas y comprobar que las salidas obtenidas son las deseadas.

Estructura básica:

1. Se crea una entidad de simulación sin puertos de entrada ni de salida
2. Se añade como component la entity a verificar
3. Se definen tantas señales como puertos de la entity del diseño a verificar
4. Se instancia el component igualando las señales internas a las entradas
5. Se crea el process de simulación, no tiene lista de sensibilidad
6. Se definen los valores iniciales de las entradas
7. Se definen los valores de las entradas en los siguientes instantes de tiempo

Cómo implementar una entrada periódica: clk

```
clk_process : PROCESS ()
BEGIN
    clk <= '0';
    WAIT FOR clk_period/2;
    clk <= '1';
    WAIT FOR clk_period/2;
END PROCESS; -- clk_process
```

Figura 1.7.1: Ejemplo Clk

Como este process no acaba con un wait sin for, se repetirá indefinidamente.

1.8. Introducción a las FPGAs

Una Field-Programmable Gate Array (FPGA) es un HW de prototipado automático. Inicialmente servía para lo mismo que un entrenador, con la diferencia de que el circuito se implementaba automáticamente. Debido a su gran capacidad de procesamiento, versatilidad y a las herramientas de diseño 9. Introducción a las TC versatilidad y a las herramientas de diseño disponibles, actualmente existen circuitos comerciales que llevan FPGA incorporadas.

Los componentes de una FPGA son:

- Celdas de entrada salida
- Celdas lógicas programables
- Celdas de interconexión programables (las estructuras de interconexionado son muy costosas)
- Memorias(opcional)
- Multiplicadores(opcional)
- Micro-controladores(opcional)

Una CLB contiene:

- Look-Up Tables: son memorias Read Only Memory (ROM) que almacenan 1x16 bits, pueden implementar cualquier función lógica de 4 entradas.
- Biestables: se utilizan en el caso de que la celda deba implementar HW secuencial. Se pueden configurar si son disparados por flanco o por nivel.
- Multiplexores: para interconectar las entradas con los módulos o los módulos entre si se utilizan multiplexores

2 | Evaluación de parámetros físicos del diseño

2.1. Parámetros

Área (tamaño). Tamaño de silicio (cm^2 o mm^2) necesarios para implementar una determinada aplicación. Depende de la arquitectura del circuito y de la tecnología empleada para su fabricación. Si no tenemos el layout del circuito podemos estimarlo con un número de puertas NAND equivalente.
Velocidad (tiempo). La velocidad de un circuito se corresponde con el tiempo necesario para realizar una determinada función, normalmente este concepto se calcula sobre el peor caso posible. Siempre es deseable que el circuito sea lo más rápido posible, pero casi siempre para conseguirlo se necesita un diseño más sofisticado, lo que puede implicar un mayor área de silicio y/o un mayor consumo de energía. La velocidad de un circuito está directamente relacionada con la frecuencia de reloj, pero no es verdad que una mayor frecuencia de reloj garantice que el resultado se obtenga en menos tiempo. Tendremos que evaluar a qué frecuencia puede trabajar el circuito, cuántas operaciones puede realizar por unidad de tiempo y si funcionará correctamente a la frecuencia de reloj exigida.
Potencia (energía). La potencia hace referencia a la energía consumida por el circuito al realizar las operaciones para las cuales fue diseñado. En todos aquellos circuitos diseñados para dispositivos móviles la reducción del consumo de potencia es la prioridad. Antes de su fabricación el circuito puede ser rediseñado para cumplir los requisitos de consumo. Un mismo diseño consume menos en tecnologías. La temperatura que alcanza un circuito está directamente relacionada con el consumo de energía. Cuanto mas dure la batería de un circuito, menor será el impacto medioambiental.
Coste. El diseño de un circuito digital es rara vez un objetivo en sí mismo. Se trata de una actividad económica y el coste es un importante factor, si no el decisivo. En el coste final del circuito influyen:

- El coste de desarrollo (diseño)
- El coste de producción (diseño y fabricación)
- El tiempo de salida al mercado (comercialización)

La importancia de la evaluación es:

- Conseguir diseños más rápidos, con mayor capacidad de procesamiento y reducción en la cantidad de recursos HW para realizar las mismas operaciones.
- Diseños más pequeños, menor coste de fabricación, menor posibilidad de errores físicos durante la fabricación y aumentar la capacidad de integración.
- Diseños más eficientes energéticamente, con menor huella de CO_2, mayor duración de la batería y menor coste de operación.

2.2. Análisis estático de tiempos (STA)

El análisis de la temporización de un circuito recibe el nombre de Static Time Analisis (STA). Determina si el circuito satisface los requisitos de temporización impuestos por el reloj.
Para este análisis debemos calcular el retardo de todos los caminos del circuito y determinar si su valor es mayor o menor que el valor límite impuesto por la señal de reloj. Si es mayor el retardo, entonces se produce una violación del timing. Si es menor, entonces el camino más lento establece la frecuencia máxima del reloj.

2.2.1. Retardo de camino

Es la suma de los retardos de la lógica y conexiones desde:

- Las entradas primarias hasta las salidas primarias para todos los posibles caminos combinacionales.

- Las entradas primarias hasta los elementos de memoria.
- Los elementos de memoria hasta las salidas primarias.
- Desde los elementos de memoria hasta los elementos de memoria.

Retardo de lógica: retardo de las puertas o elementos lógicos del circuito.

- Combinacionales. OR, AND, sumadores, decodificadores...
- Secuenciales: FF, contadores, registros, memorias ...

Retardo de conexión: todas las conexiones del circuito tienen un retardo.

2.2.2. Modelado retardo: timing arcs

Combinacional: retardo desde cualquier entrada a cualquiera de sus salidas. Secuencial: retardo desde el pin del reloj hasta cualquiera de sus salidas, a este tiempo se le conoce como Clk-2-Q. Conexiones: retardo desde el pin de salida hasta cada uno de sus destinos.

2.3. Cálculo de tiempo de propagación

Se suman todos los retardos de la lógica y las conexiones para todos los caminos del circuito combinacional. Depende del camino, hay tantos tiempos como caminos. Se usa el valor del camino más lento, conocido como **camino crítico.**

2.3.1. Funcionamiento síncrono

Si en el sistema hay elementos de memoria debe existir un reloj. El reloj determina cuándo se actualiza el estado de los elementos de memoria del sistema.
La hipótesis de funcionamiento síncrono obliga a que la diferencia en tiempo entre dos flancos consecutivos (positivos o negativos los dos) sea superior al tiempo necesario para que los valores de las entradas de los registros sean estables.

2.3.2. Definiciones

Tiempo de set-up: tiempo mínimo que la entrada debe permanecer antes del suceso del reloj.
Tiempo de hold: tiempo mínimo que la entrada debe permanecer estable después del suceso del reloj.
Tiempo de propagación del registro: tiempo que tarda en propagarse el dato desde la entrada hasta la salida del registro cuando ocurre el flanco activo de reloj.
Tiempo de propagación combinacional: tiempo desde que cambian las entradas del circuito combinacional hasta que se produce un cambio en las salidas. Depende del camino. Se usa el valor del camino crítico.

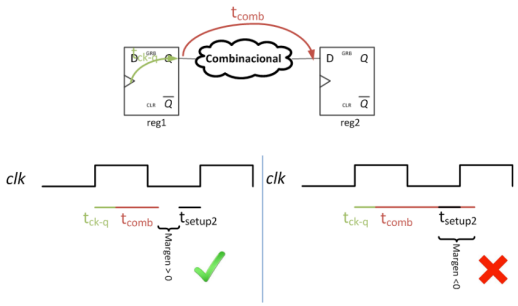


Figura 2.3.1: Margen de setup

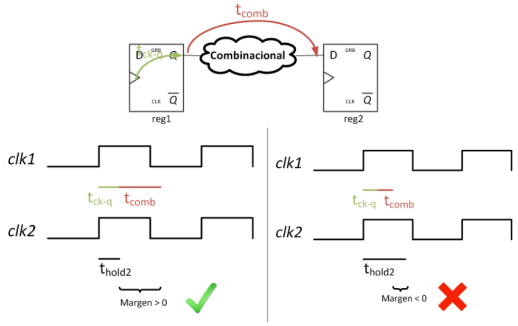


Figura 2.3.2: Margen de setup

2.3.3. Cálculo de tiempo de set-up y hold

Sin considerar el clock skew y el clock jitter:

$$Margen\ setup = t_{clk} - \left(t_{clk_q} + t_{comb} + t_{setup}\right)$$

$$Margen\ hold = t_{clk_q} + t_{comb} + t_{hold}$$

Margen negativo → error de temporización

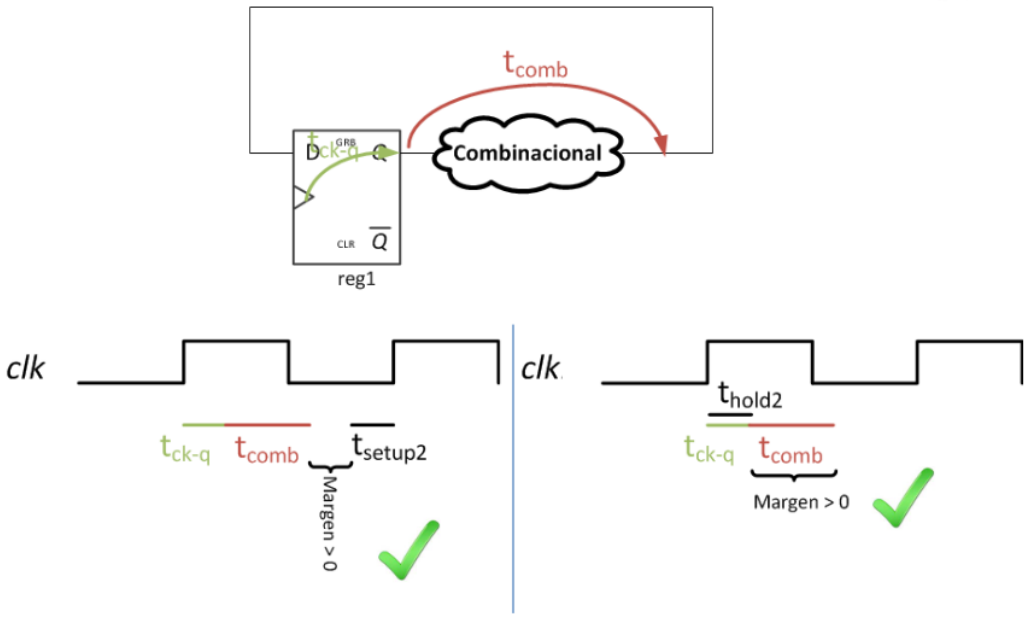


Figura 2.3.3: Circuito retroalimentado

2.3.4. Metaestabilidad

Consiste en una indecisión prolongada en el comportamiento lógico del biestable al intentar almacenar uno de sus dos estados estables. Ocurre si existe una violación de setup o de hold. Mientras ese biestable permanezca en metaestabilidad, sus señales de salida están indefinidas a nivel lógico.

2.3.5. Clock skew

El sesgo del reloj es un fenómeno que ocurre en los circuitos secuenciales cuando el reloj no llega al mismo tiempo a todos los componentes de memoria. Puede deberse a la longitud del cable, variaciones de temperatura, capacidades parásitas, imperfecciones del silicio, etc. Cuando el tiempo de ciclo es pequeño, este problema se convierte en uno de los más importantes a la hora de diseñar el circuito.
Hay dos tipos de clock skew:

- **Positive skew:** el registro que transmite recibe el reloj antes que el registro que recibe.
- **Negative skew:** el registro que transmite recibe el reloj después que el registro que recibe.

$$skew = t_{dest} - t_{orig}$$

2.3.6. Clock jitter

El jitter es una modificación no deseada en la periodicidad del reloj. En otras palabras, es la variación de los flancos de reloj respecto de su posición ideal en el tiempo. Tiene orígenes muy variados, variaciones en la fabricación de los osciladores, variaciones de temperatura etc.
Se especifica de tres maneras:

- **Jitter absoluto:** diferencia entre la posición real del flanco de reloj y su posición ideal.

- **Jitter periódico:** diferencia entre el periodo real del reloj y el periodo ideal. Es el más importante a efectos de STA
- **Jitter ciclo-a-ciclo:** diferencia en la duración de dos periodos de reloj adyacentes

Es necesario tenerlo en cuenta en el análisis temporal del circuito porque puede acortar la duración del ciclo de reloj. Especificado como RMS(Root Mean Square) o valor pico-a-pico, se mide la duración media del ciclo sobre una muestra y se obtiene su desviación estándar.

2.3.7. Cálculo de tiempo de setup y hold

Considerando el clock skew y el clock jitter

$$Margen\ setup = t_{clk} + skew - (t_{clk\_q} + t_{comb} + t_{setup} + jitter)$$

$$Margen\ hold = t_{clk\_q} + t_{comb} - (skew + jitter + t_{hold})$$

Margen negativo → error de temporización

$$t_{clk} + skew > (t_{clk\_q} + t_{comb} + t_{setup} + jitter)$$

$$t_{clk} > (t_{clk\_q} + t_{comb} + t_{setup} + jitter) - skew$$

2.3.8. Falso camino crítico

Parece el camino más lento del circuito pero en realidad no lo es, no propaga una transición. Los diseños que comparten lógica para distintos cálculos son susceptibles de tener falsos caminos.

2.4. Segmentación

Dividir un circuito en etapas usando registros. Las salidas de los registros de una etapa proporcionan las entradas de la siguiente etapa. Todas las etapas operan concurrentemente.

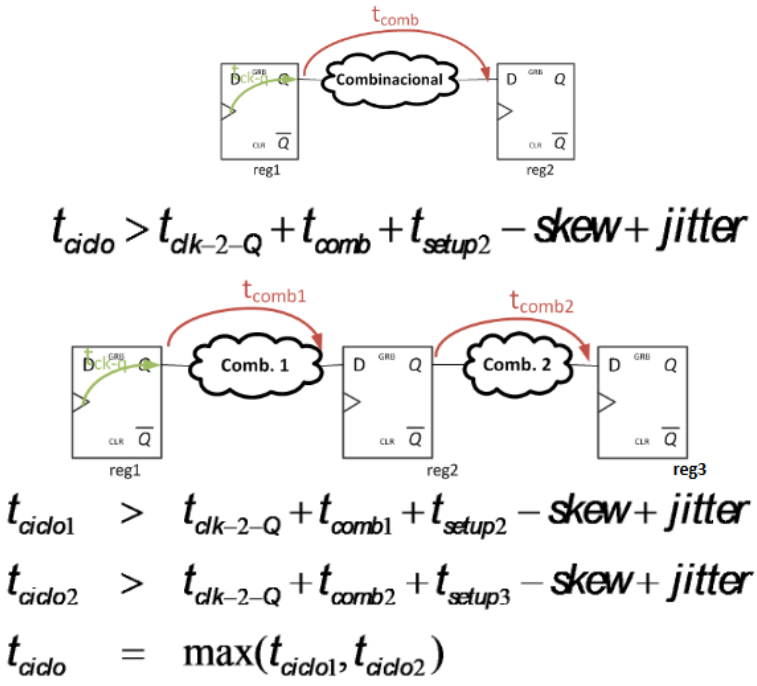


Figura 2.4.1: Tiempo de ciclo

2.4.1. Rendimiento

Latencia: tiempo transcurrido para obtener un nuevo resultado a la salida del circuito desde su entrada en este.

$$L = n * t_{ciclo}$$

Intervalo de inicialización (Throughput): tasa de entrada de nuevos datos en el sistema por unidad de tiempo.

$$T = \frac{1}{t_{ciclo}}$$

2.5. Comportamiento dinámico

La simulación lógica no mide retardos. Los retardos se miden en la Simulación Post-Place & Route. La herramienta calcula el retardo de todos los caminos, el más lento, llamado camino crítico, es el que define el retardo del circuito. El retardo de los distintos caminos del circuito puede originar errores funcionales.

2.5.1. Azares y glitches

El retardo de propagación de un circuito se define como el tiempo necesario para obtener un valor de salida válido. Los azares o riesgos son las posibles fluctuaciones de la señal de salida antes de alcanzar su valor final:

- Riesgo estático
- Riesgo dinámico

Estas fluctuaciones pueden ser uno o varios pulsos no deseados que reciben el nombre de glitches.

2.5.2. Riesgos estáticos

Un determinado cambio en la entrada produce un glitch en la salida cuando no debería producirse ningún cambio. Se deben a la existencia de distintos caminos que convergen hacia la salida con diferente retardo.

2.5.3. Riesgos dinámicos

La salida debe cambiar pero el cambio no es directo sino con fluctuaciones. Suelen producirse en circuitos con riesgo estático y un nivel más de puertas.

2.5.4. Análisis de consumo

Existen distintas métricas: potencia (temperatura) y energía (duración de la batería). Existe también el consumo estático y dinámico.

2.5.5. Consumo estático

Consumo de los bloques de circuito cuando no hay transiciones en las señales de entrada.

$$P_{static} = V_{dd} * I_{static}$$

Donde:

- I_{static} es la corriente estática
- V_{dd} es la tensión de alimentación

I_{static} depende de la corriente de fuga de los transistores, la tensión de alimentación y la temperatura.

2.5.6. Consumo dinámico

Consumo de los bloques del circuito cuando hay transiciones en las señales de entrada.

$$P = \frac{1}{2} * C_{load} * V_{dd}^2 * f * t$$

Donde:

- C_{load} es la capacidad de carga
- V_{dd} es la tensión de alimentación
- f es la frecuencia del reloj
- t es la tasa de actividad de la puerta

3 | Diseño combinacional avanzado

3.1. Módulos combinacionales

3.1.1. Decodificador

Circuito combinacional con n entradas y 2^n salidas. Cada salida es uno de los minterms que pueden generarse con n variables.

$$D_i = \left\{ \begin{array}{l} 1 \; si \; A = idonde \; A = \sum_{j=0}^{n-1} A_j 2^j \; para \; 0 \leq i \leq 2^n - 1 \\ 0 \; en \; caso \; contrario \end{array} \right\}$$

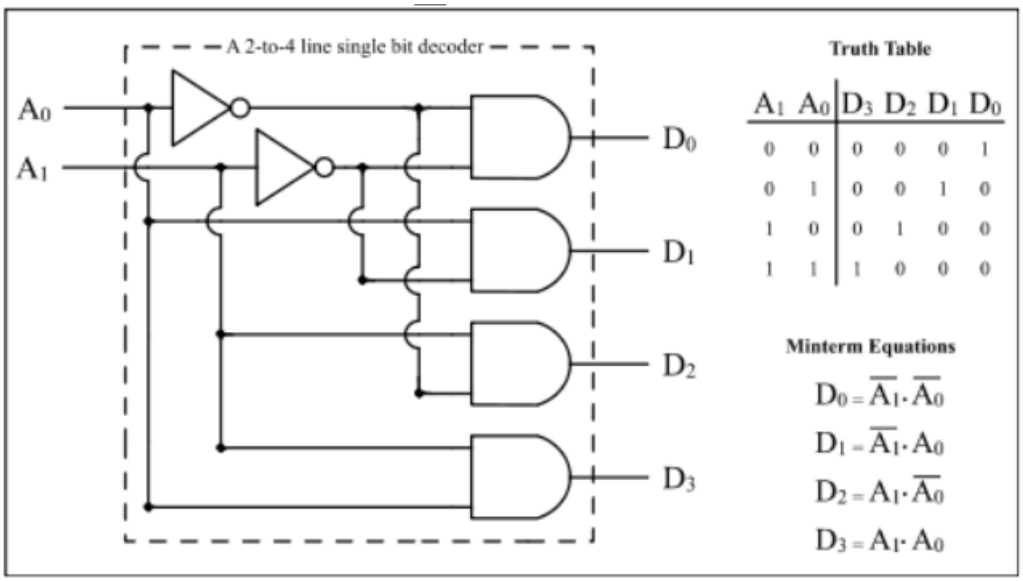


Figura 3.1.1: Decodificador

```
ENTITY decoder_1 IS
    PORT (
        sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        res : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END decoder_1;

ARCHITECTURE rtl OF decoder_1 IS
BEGIN
    res <= "00000001" WHEN sel = "000" ELSE
        "00000010" WHEN sel = "001" ELSE
        "00000100" WHEN sel = "010" ELSE
        "00001000" WHEN sel = "011" ELSE
        "00010000" WHEN sel = "100" ELSE
        "00100000" WHEN sel = "101" ELSE
        "01000000" WHEN sel = "110" ELSE
        "10000000";
    p_decode : PROCESS (sel)
    BEGIN
        res <= (OTHERS => '0');
        res(to_integer(unsigned(sel))) <= '1';
    END PROCESS; -- p_decode
END rtl;
```

Figura 3.1.2: Código Codificador

3.1.2. Decodificador

```
ENTITY priority_encoder IS
    PORT (
        sel : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        code : OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END priority_encoder;

ARCHITECTURE rtl OF priority_encoder IS
BEGIN
    code <= "111" WHEN sel(7) = '1' ELSE
        "110" WHEN sel(6) = '1' ELSE
        "101" WHEN sel(5) = '1' ELSE
        "100" WHEN sel(4) = '1' ELSE
        "011" WHEN sel(3) = '1' ELSE
        "010" WHEN sel(2) = '1' ELSE
        "001" WHEN sel(1) = '1' ELSE
        "000" WHEN sel(0) = '1' ELSE
        "---"; -- --- es el simbolo de don't care en
    VHDL
    END rtl; -- rtl
```

Figura 3.1.3: Código Decodificador

3.1.3. Multiplexor

Descripción de alto nivel:

$$y = x_s \; donde \; s = \sum_{j=0}^{n-1} x_j 2^j$$

Implementación

- 2 a 1 $y = x_0 * \overline{s} + x_1 * s$
- 4 a 1 $y = x_0 \, \overline{s_1} \, \overline{s_0} + x_1 \, \overline{s_1} \, s_0 + x_2 \, s_1 \, \overline{s_0} + x_3 \, s_1 \, s_0$

```
ENTITY mux8_1 IS
    PORT (
        sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        datos : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        z : OUT STD_LOGIC);
END mux8_1;

ARCHITECTURE rtl OF mux8_1 IS
BEGIN
    WITH sel SELECT
        z <= datos(0) WHEN "000",
        datos(1) WHEN "001",
        datos(2) WHEN "010",
        datos(3) WHEN "011",
        datos(4) WHEN "100",
        datos(5) WHEN "101",
        datos(6) WHEN "110",
        datos(7) WHEN OTHERS;
    -- z<= data(to_integer(unsigned(sel)))
END rtl; -- rtl
```

Figura 3.1.4: Multiplexor

3.1.4. Sumador

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY adder IS
    PORT (
        a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        ci : IN STD_LOGIC;
        sum : OUT STD_LOGIC_VECTOR(7 downto0);
        co : OUT STD_LOGIC);
END adder;

ARCHITECTURE rtl OF adder IS
    SIGNAL a_i, b_i, sum_i : STD_LOGIC_VECTOR
    (8 downto0);
BEGIN
    a_i <= '0' & a;
    b_i <= '0' & b;
    sum_i <= a_i + b_i + ci;
    sum <= sum_i(7 DOWNTO 0);
    co <= sum_i(8);
END rtl;
```

Figura 3.1.5: Sumador con std_logic

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY sum8 IS PORT (
    data1 : IN STD_LOGIC_VECTOR (7 downto0);
    data2 : IN STD_LOGIC_VECTOR (7 downto0);
    cin : IN STD_LOGIC;
    cout : OUT STD_LOGIC;
    salida : outSTD_LOGIC_VECTOR (7 downto0));
END sum8;

ARCHITECTURE Behavioral OF sum8 IS
    SIGNAL misennal1 : signed (8 downto0);
    SIGNAL tmp : signed(0 downto0);
BEGIN
    tmp(0) <= cin;
    misennal1 <= signed(data1(7) & data1) +
    signed(data2(7) & data2) + tmp;
    salida <= STD_LOGIC_VECTOR(misennal1(7
    downto0));
    cout <= misennal1(8);
END Behavioral;
```

Figura 3.1.6: Sumador con numeric y datos con signo

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY alu IS
    PORT (
        data1 : IN STD_LOGIC_VECTOR (7 downto0);
        data2 : IN STD_LOGIC_VECTOR (7 downto0);
        op : IN STD_LOGIC;
        salida : OUT STD_LOGIC_VECTOR (7 downto0));
END alu;

ARCHITECTURE Behavioral OF alu IS
    SIGNAL salida_aux : signed(7 downto0);
BEGIN
    Salida_aux <= signed(data1) + signed(data2) WHEN op = '0' ELSE
        signed(data1) - signed(data2) WHEN op = '1' ELSE
        (OTHERS => '0');
    Salida <= STD_LOGIC_VECTOR(salida_aux);
END Behavioral;
```

Figura 3.1.7: Restador

3.2. Aritmética en VHDL

3.2.1. Operadores incluidos en VHDL-93 sin incluir ningún paquete

Operador	Descripción	Tipo de operando a	Tipo de operando B	Tipo del resultado
a ** b	Exponenciación	Integer / Natural	Integer/ Natural	Integer / Natural
a * b	Multiplicación	Integer / Natural	Integer / Natural	Integer / Natural
a / b	División			
a mod b	Módulo			
a rem b	Resto			
+ a	Identidad	Integer / Natural		Integer / Natural
- a	Negación			
a + b	Sumador	Integer / Natural	Integer / Natural	Integer / Natural
a - b	Resta			
a + b	Concatenación	Array 1-D, elemento	Array 1-D, elemento	Array 1-D
a = b	Igual	Cualquiera	Mismo que a	boolean
a /= b	No igual			
a sll b	Desp. Lóg. Izq.	bit_vector	Integer / Natural	bit_vector
a srl b	Desp. Lóg. Der.			
a sla b	Desp. Arit. Der.			
a sra b	Desp. Arit. Der.			
a rol b	Rotación Izq.			
a ror b	Rotación der.			
a <b	Menor que	Escalar o Array de 1-D	Mismo que a	boolean
a <= b	Menor o igual			
a >b	Mayor que			
a >= b	Mayor o igual que			
a and b	and	boolean, bit o bit_vector	Mismo que a	Mismo que a
a or b	or			
a xor b	xor			
a nand b	nand			
a nor b	nor			
a xnor b	xnor			

3.2.2. Operadores y funciones del paquete IEEE std_logic_1164

Operador	Tipo de operando a	Tipo de operando b	Tipo del resultado
not a	std_logic_vector, std_logic		Mismo que a
a and b	std_logic_vector, std_logic	Mismo que a	Mismo que a
a or b			
a xor b			
a nand b			
a nor b			
a xnor b			

Función	Tipo de operando a	Tipo de resultado
to_bit(a)	std_logic	bit
to_stdulogic(a)	bit	std_logic
to_bitvector(a)	std_logic_vector	bit_vector
to_stdlogicvector(a)	bit_vector	std_logi_vector

3.2.3. Operadores del paquete IEEE numeric_std

Operador	Tipo de operando a	Tipo de operando b	Tipo de resultado
abs a- a	signed		signed
a * b	unsigned, natural, signed, integer	unsigned, natural, signed, integer	unsigned, signed
a / b			
a mod b			
a rem b			
a + b			
a - b			
a = b	unsigned, natural, signed, integer	unsigned, natural, signed, integer	boolean
a /= b			
a <b			
a <= b			
a >b			
a >= b			

3.2.4. Funciones de conversión de datos o casting

Para la conversión de señales de un tipo a otro se deben usar las siguientes funciones y operaciones de casting:

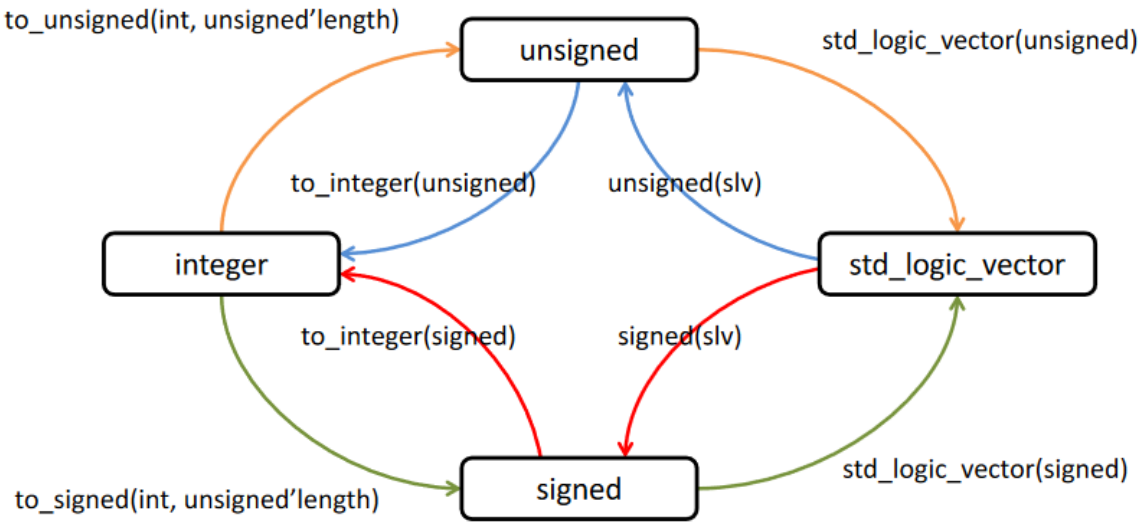


Figura 3.2.1: Funciones de conversión de tipos

3.3. Unidades funcionales multifunción

En este apartado se estudia la estructura del diseño y reducir la cantidad de recursos HW compartiendo lógica entre distintas operaciones.
Para ello agruparemos Unidad Funcional (UF) sencillas en UF más complejas, esto se conoce como UF multifunción. Esto lo utilizaremos cuando la UF multifunción y el coste de conexión es menor que el coste de las UF sencillas. Para lograr esto utilizaremos un algoritmo de particionamiento y un grafo de compatibilidad.

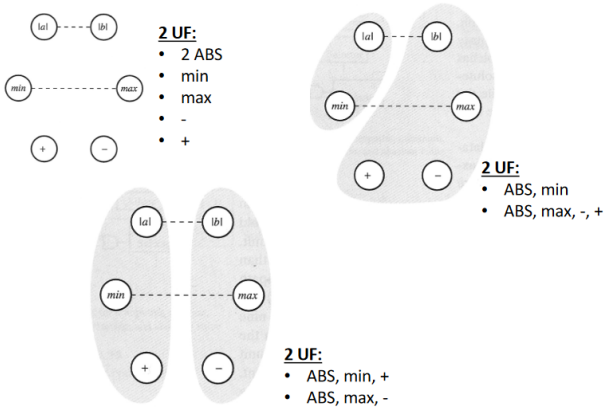


Figura 3.3.1: Optimización de múltiples UF

3.4. Redes iterativas

Las redes iterativas son conjuntos de módulos idénticos, cada uno conectado exclusivamente a los módulos vecinos.

Una red iterativa 1-D de orden k es una implementación de una función de n variables, donde:

- (n/k) celdas idénticas, G, con:
  - Entradas externas x, e internas c
  - Salidas externas z, e internas c

$$C_{j+1} = G\left(x_j, c_j\right)$$

$$Z_j = F\left(x_j, c_j\right)$$

- Celdas de los extremos pueden simplificarse aprovechando las condiciones de contorno

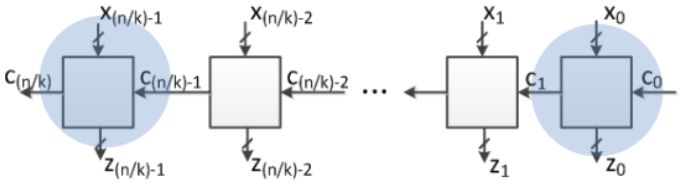


Figura 3.4.1: Ejemplo Red Iterativa 1-D

3.4.1. Diseño

Primero determinamos el valor de k, el equilibrio entre la complejidad de las celdas (nž de entradas) y el nž de celdas (retardo total).
Después determinamos los valores que deben trasmitirse entre los módulos intermedios (salidas internas) y el valor de la salida externa del último módulo.
Finalmente hacemos la descripción de alto nivel de los bloques intermedios, la implementación de las funciones de conmutación de los bloques, y la simplificación de las celdas inicial y final.

3.4.2. Temporización

Retardo (Δ)

- Función de retardo de las salidas externas de cada celda Δ_o
- Función de retardo de las salidas internas de cada celda Δ_c

$$\Delta = \left(\frac{n}{k} - 1\right)\Delta_c + max\left(\Delta_o, \Delta_c\right)$$

3.4.3. Redes 1-D en VHDL

En este ejemplo se diseñará una red de resolución de propiedades iterativa con n entradas y n salidas. LA salida $Z_i = 1$ *si* $X_i = 1$ *y* $X_j \forall j > i$

```
ENTITY celda IS
  PORT (
    x, c_in : IN STD_LOGIC;
    c_out, z : OUT STD_LOGIC);
END celda;

ARCHITECTURE arch OF celda IS
BEGIN
  c_out <= c_in OR x;
  z <= x AND (NOT(c_in));
END arch; -- arch
```

Figura 3.4.2: Ejemplo de celda de la red

```
gen1 : FOR i IN 0 TO M GENERATE
  u : celda PORT MAP(X(i), C(i), C(i + 1), Z(i
    ));
END GENERATE gen1;
```

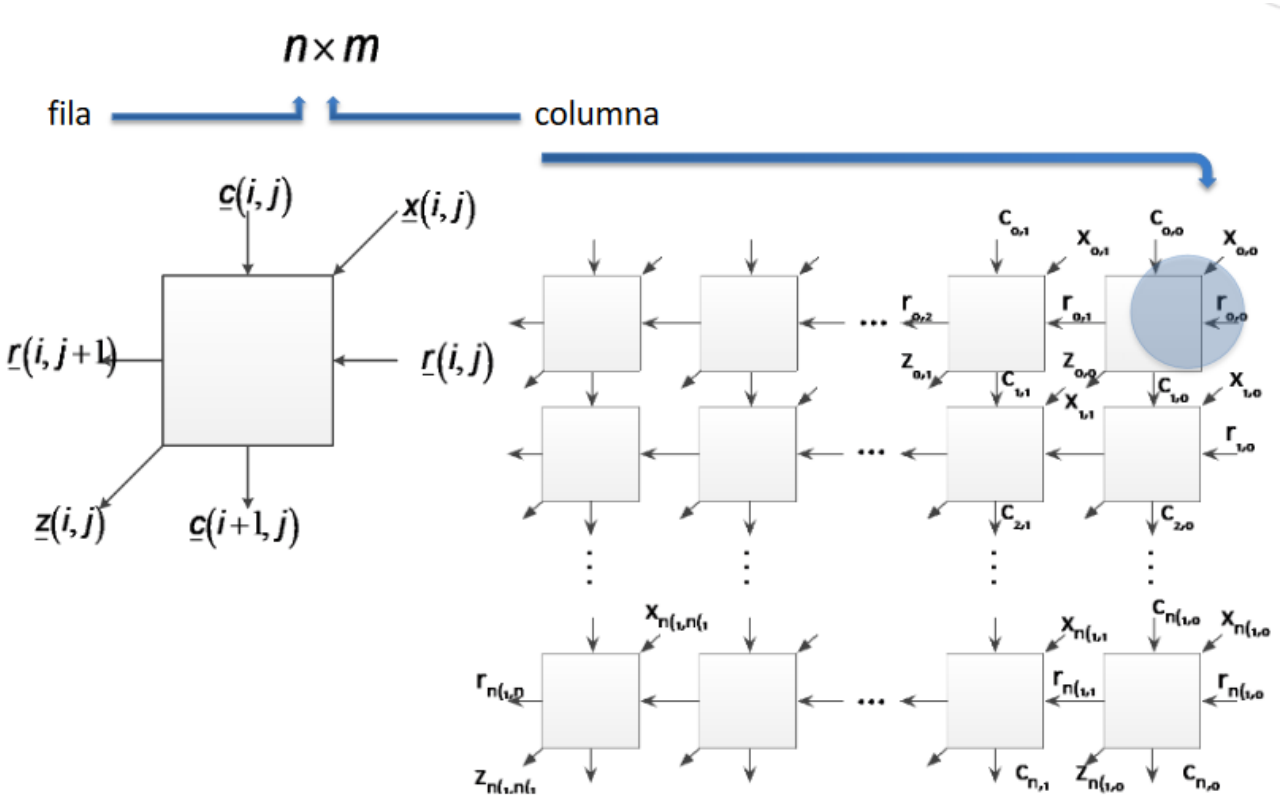
Figura 3.4.3: Generador de celdas conectadas

```
ENTITY red IS
  GENERIC (n : NATURAL := 4);
  PORT (
    x : IN STD_LOGIC_VECTOR(n - 1 DOWNTO 0);
    z : OUT STD_LOGIC_VECTOR(n - 1 DOWNTO 0));
END red;

ARCHITECTURE arch OF red IS

  SIGNAL c : STD_LOGIC_VECTOR(n DOWNTO 0);
  COMPONENT celda
    PORT (
      x, c_in : IN STD_LOGIC;
      c_out, z : OUT STD_LOGIC);
  END COMPONENT;

BEGIN
  gen1 : FOR i IN 0 TO M GENERATE
    u : celda PORT MAP(X(i), C(i), C(i + 1), Z(i));
  END GENERATE gen1;
  C(n) <= '0'; -- Condicion de entorno
END arch; -- arch
```

Figura 3.4.4: Ejemplo red iterativa con generate

3.4.4. Redes iterativas 2-D

Array de celdas idénticas conectadas con sus vecinas. Cada celda (i,j) posee:



Para generar una matriz 2D tendremos que usar dos bucles anidados.

3.5. Técnicas de mejora del rendimiento

La utilización de diseños combinacionales bastante grandes (datos de entrada a partir de 32 bits), hace que el tiempo asociado al cambio crítico sea grande y el retardo de la red elevado.
En las redes iterativas la última celda no podrá generar su salida hasta que no haya recibido su señal intermedia. Esta señal interna ha tenido que atravesar n - 1 celdas que componen la red, por lo tanto el retardo es muy elevado si n es grande.

3.5.1. Redes de anticipación

Para reducir el retardo de las redes iterativas, anticiparemos el valor de la señal que se propaga entre celdas.

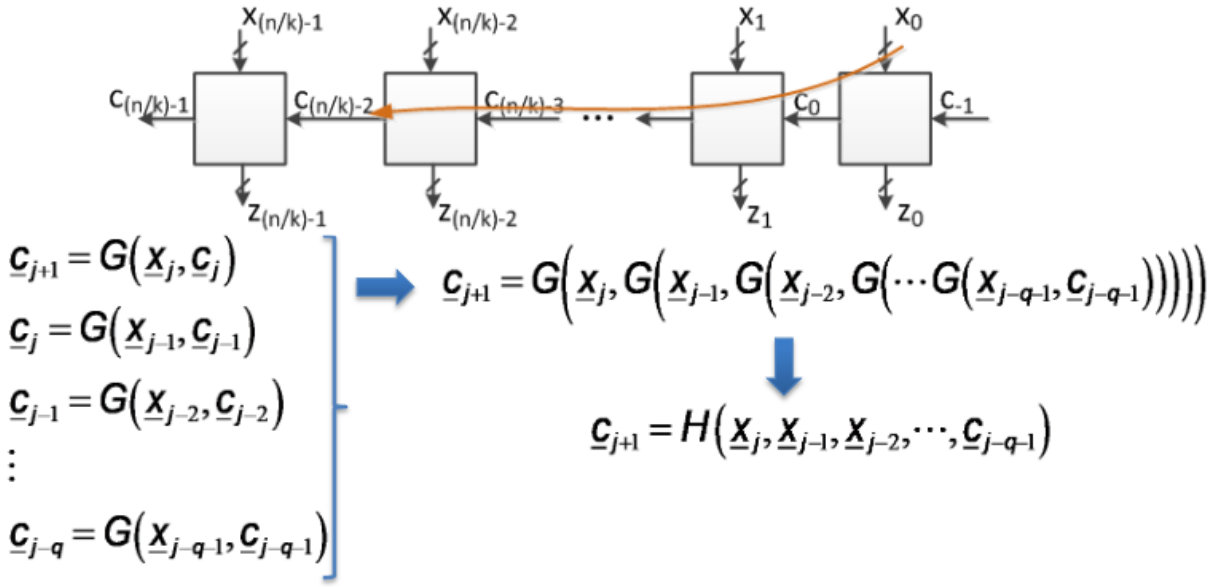


Figura 3.5.1: Red iterativa sin anticipación

$$c_{j+1} = H(x_j, x_{j-1}, x_{j-2}, \cdots, c_{j-q-1})$$

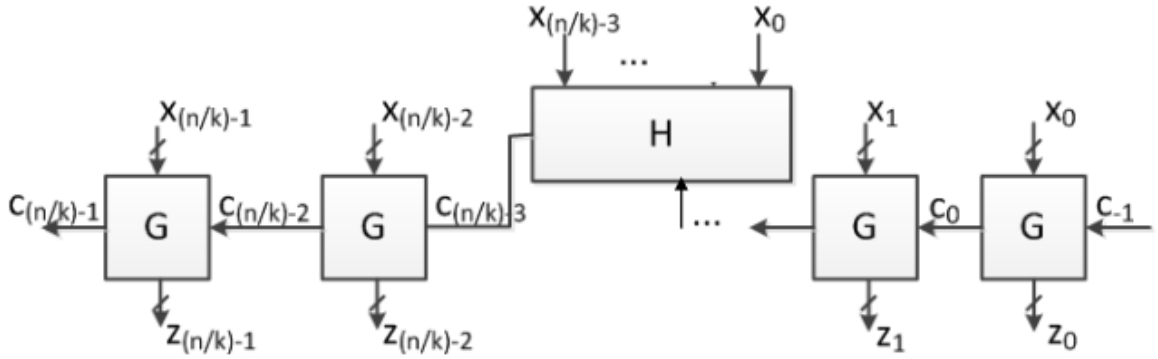


Figura 3.5.2: Red iterativa con anticipación

3.5.2. Redes en árbol

Implementar una función de n entradas usando bloques de k entradas:

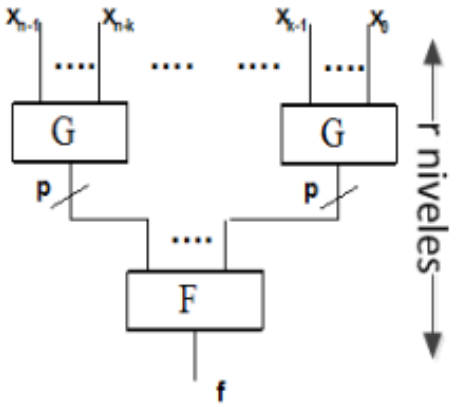


Figura 3.5.3: Red en árbol

- Módulo G: k entradas y p salidas $\rightarrow \frac{n}{k}$ módulos
- Módulo F: $p\frac{n}{k}$ entradas
- Retardo: $(r-1)\,\Delta_G + \Delta_F$

3.6. Errores de diseño

3.6.1. Lazos combinacionales

Estructuras lógicas que contienen realimentación sin ningún elemento síncrono en el camino. Por ejemplo $A <= A + 1;$ VHDL no permitirá ni la simulación ni la síntesis de este diseño.

3.6.2. Errores de diseño en circuitos combinacionales

Si implementamos la lógica combinacional mediante un process:

- **Error 1:** no incluir todas las entradas en la lista de sensibilidad
- **Error 2:** no terminar una sentencia if con un else
- **Error 3:** no especificar el valor de alguna salida en alguna de las ramas de la sentencia if

En los errores 2 y 3 se añadirá un latch, lo que implica lógica secuencial.

4 | Diseño algorítmico

4.1. Introducción

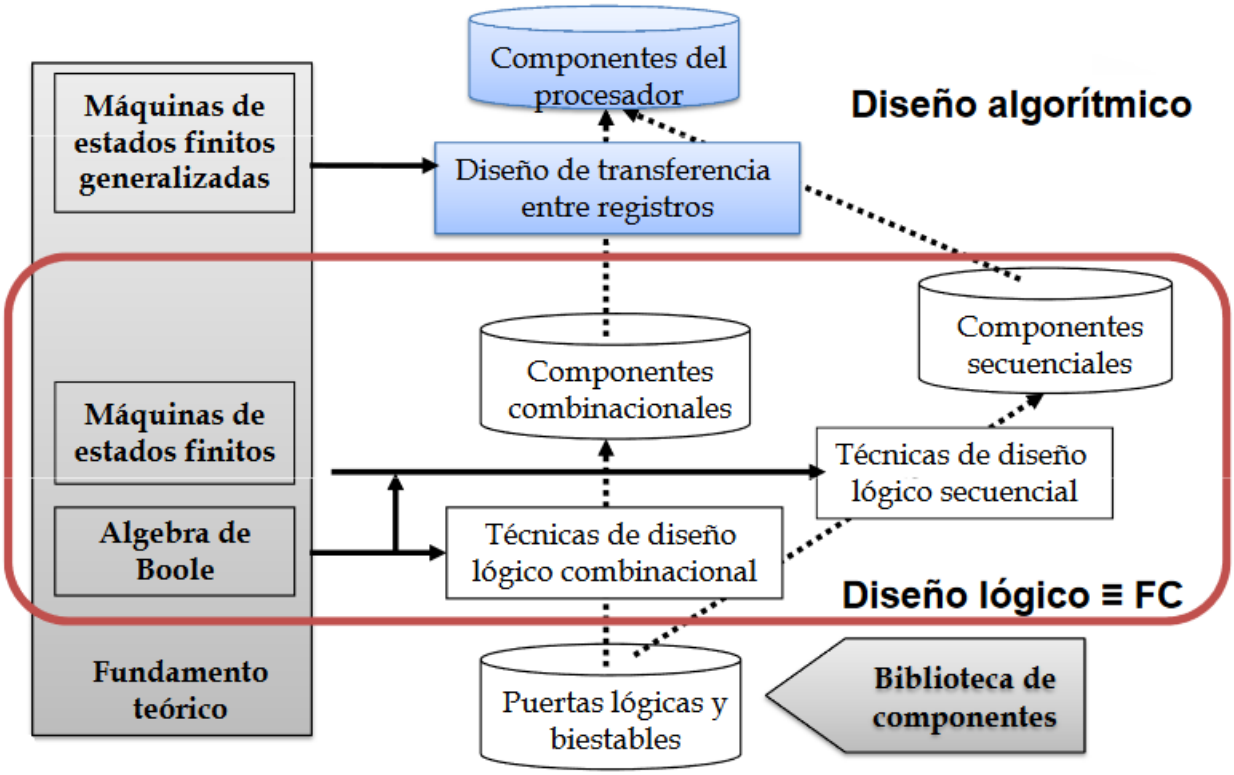


Figura 4.1.1: Flujo de diseño

4.2. Máquinas de estados: repaso

4.2.1. Máquinas de estados finitos (FSM)

Mealy: un cambio en la entrada en cualquier instante influye inmediatamente en la salida.

$$Z(t) = H(X(t), S(t))$$
$$S(t+1) = G(X(t), S(t))$$

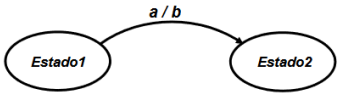


Figura 4.2.1: Ejemplo estados Mealy

Moore: sólo el cambio del estado influye en la salida

$$Z(t) = H(S(t))$$
$$S(t+1) = G(X(t), S(t))$$

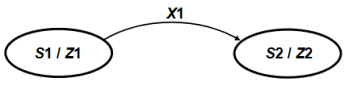


Figura 4.2.2: Ejemplo estados Moore

4.2.2. Funcionamiento síncrono

La hipótesis de funcionamiento síncrono de los sistemas secuenciales supone que:

- El estado sólo cambia cada vez que el ciclo de reloj y el cambio es simultáneo en todos los bits del cambio de estado.
- Tras un cambio de estado, las entradas de los bits del registro de estado tienen tiempo para alcanzar un valor estable antes del siguiente cambio de estado.

4.3. Elementos de memoria: repaso

Cuando estamos diseñando un sistema secuencial uno de los elementos HW más importantes son los elementos de almacenamiento:

- Almacenan el estado actual del sistema (máquina de control)
- Almacenan valores intermedios (registros de datos)

El elemento de memoria más sencillo es el **biestable**. Se denomina biestable porque presenta dos únicos estados estables: Salida 0 o Salida 1. Sirve para almacenar un bit de información.

4.3.1. Tipos de biestables

Según su comportamiento lógico:

- S-R
- D
- J-K
- T

S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	proh.

D	Q+
0	0
1	1

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

T	Q+
0	Q
1	$\overline{Q}$

Figura 4.3.1: Tabla de verdad de los distintos tipos de biestables

Según su comportamiento temporal:

- Latch
- Latch síncrono (sensible a nivel)
- Flip-Flop disparado por flanco
- Flip-flop maestro-esclavo

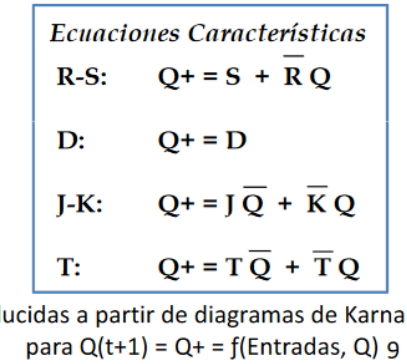


Figura 4.3.2: Ecuaciones características de los distintos tipos de biestables

4.3.2. Biestables asíncronos: Latch

La salida cambia cuando cambian las entradas:

S	R	Q(t+)
0	0	Q(t)
0	1	0
1	0	1
1	1	proh.

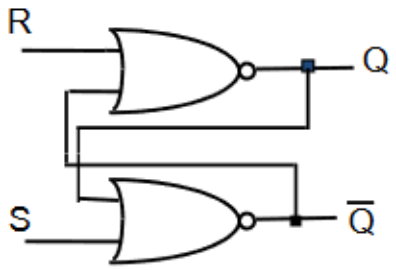
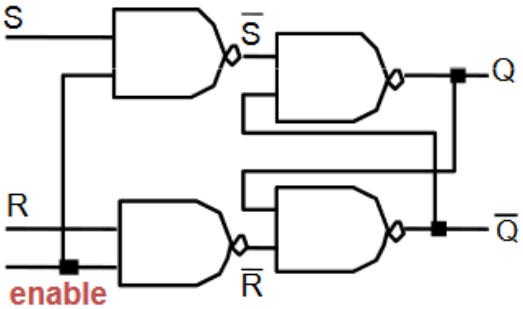


Figura 4.3.3: Ejemplo Latch S-R

4.3.3. Biestable síncrono sensible a nivel: Latch síncrono

La salida cambia cuando está activa la señal de capacitación (enable)

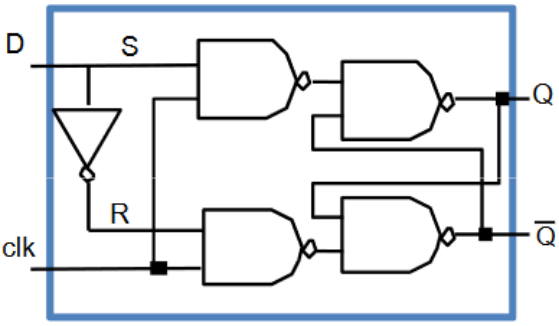


$\overline{S}$	$\overline{R}$	Q(t+)
0	0	prohibida
1	0	0
0	1	1
1	1	Q(t)

S	R	Q(t+)
0	0	Q(t)
0	1	0
1	0	1
1	1	prohibida

Figura 4.3.4: Ejemplo Latch síncrono S-R

Para que este biestable sea un tipo D, debemos cambiar el enable para que sea un clk



clk	D	Q (t+)
0	0	Q(t)
0	1	Q(t)
1	0	0
1	1	1

Figura 4.3.5: Ejemplo Latch tipo D

4.3.4. Problemas

Si utilizamos latches debemos garantizar que el pulso de reloj sea más corto que el retardo del latch y que las entradas se mantienen constantes durante el pulso del reloj.

Una alternativa es usar Flip-Flops, que son más fiables. Son disparados por flanco: la salida sólo varía durante la transición del reloj(que es una entrada dinámica).

4.3.5. Flip-Flop disparado por flanco

- clk = 0
  - R = 1 y S = 1 independientemente del valor de D
  - P1 = $\overline{D}$ y P2 = $\overline{P1}$ = D
- clk = 0 → 1
  - S = $\overline{P2}$ = $\overline{D}$ y R = $\overline{P1}$ = D
- clk = 1
  - Los cambios en D no afectan a R y S

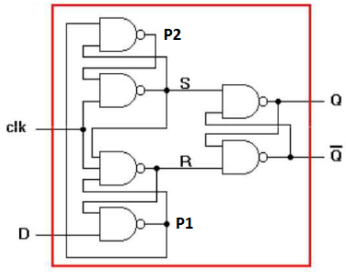


Figura 4.3.6: Ejemplo Flip-Flop disparado por flanco

4.3.6. Biestable maestro-esclavo: FF

Se lee la entrada en un nivel y se modifica la salida en el contrario:

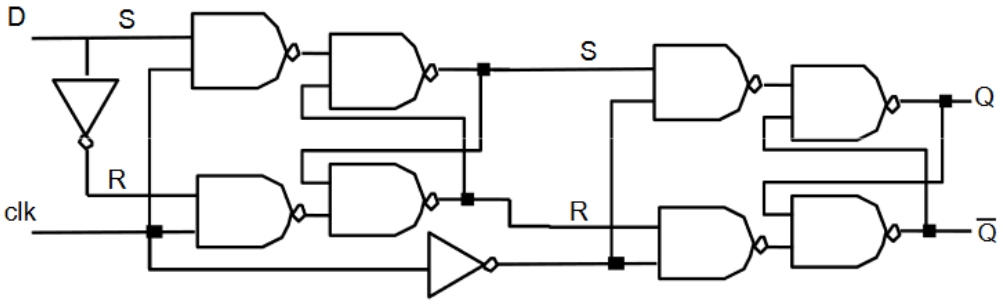


Figura 4.3.7: Biestable Maestro-Esclavo

4.3.7. Comparación del comportamiento temporal de los biestables

Tipo	¿Cuándo se muestrean las entradas?	¿Son válidas las entradas?
Latch sin reloj	siempre	retardo de propagación desde el cambio en la entrada
Latch sensible a nivel	reloj en alta (T_{setup} y T_{hold} a cada lado del eje de bajada)	retardo de propagación desde flanco de subida del reloj
Flip-Flop flanco de subida	transición de reloj de baja a alta (T_{setup} y T_{hold} a cada lado del eje de subida)	retardo de propagación desde flanco de subida de reloj
Flip-Flop flanco de bajada	transición de reloj de alta a baja (T_{setup} y T_{hold} a cada lado del eje de bajada)	retardo de propagación desde flanco de bajada del reloj

4.3.8. Registros

Hay de distintos tipos:

- Según temporización:
  - Disparado por nivel

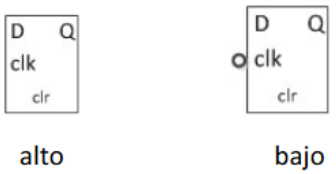


Figura 4.3.8: registros según nivel

  - Disparado por flanco

Figura 4.3.9: registros según nivel

- Según funcionalidad
  - Parallel-Input Parallel-Output (PIPO)
  - Serial-Input Parallel-Output (SIPO)
  - Parallel-Input Serial-Output (PISO)
  - Serial-Input Serial-Output (SISO)

4.4. Diseño Algorítmico

El diseño algorítmico es una modo de especificación e implementación de sistemas digitales, que permite sistematizar y automatizar en gran medida su construcción. Parte siempre de una especificación en la que el comportamiento del sistema se describe en forma de un algoritmo (cómo calcular la salida en función de la entrada). Esto se implementa con una unidad de control y una ruta de datos.

Los sistemas algorítmicos son sistemas secuenciales síncronos. El comportamiento está definido implícitamente: no se especifica el valor de z si no el modo de calcularlo (el algoritmo).

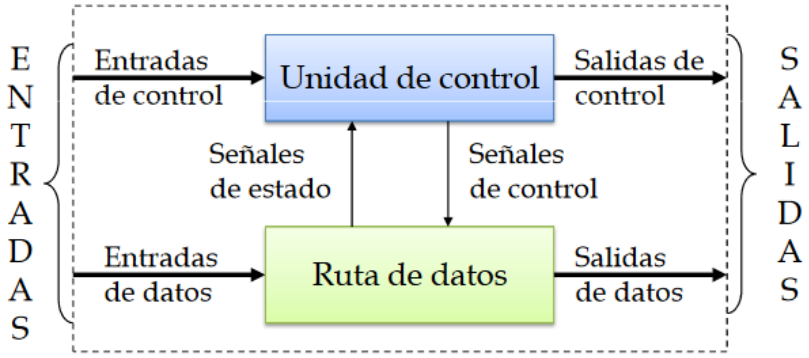


Figura 4.4.1: Modelo de sistema algorítmico

4.4.1. Proceso de diseño: resumen

Primero se define el sistema: se especifican las entradas, salidas, función y bloques disponibles.
Seguidamente se crea un diagrama Algorithmic State Machine (ASM): obtenemos un conjunto ordenado y finito de operaciones y un modelo de máquina de estados generalizada.

Por último, diseñamos la unidad de control, la ruta de datos y las ensamblamos

Glosario

This document is incomplete. The external file associated with the glossary 'acronym' (which should be called `TOC.acr`) hasn't been created.

Check the contents of the file `TOC.acn`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run xindy. Check the transcript file `TOC.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "TOC"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "TOC"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

Índice de figuras

Índice de cuadros