

ÍNDICE

Prefacio	25
Organización de la Unidad Didáctica	25
Cómo utilizar el libro	26
Objetivos docentes	27
1. Fundamentos del diseño del hardware digital	29
1.1. Introducción	33
1.2. Lenguajes para la descripción de hardware	34
1.2.1. Usos de un programa HDL	35
1.2.2. HDL más ampliamente usados	35
1.3. Ciclo de diseño de los circuitos digitales	36
1.4. Tecnologías de circuitos integrados	38
1.4.1. Clasificación de las tecnologías	39
1.4.2. Comparación entre tecnologías	42
1.5. Propiedades de los circuitos digitales	47
1.5.1. Retardo de los dispositivos	47
1.5.2. Ejecución concurrente	49
1.5.3. Diseños marginales	50
1.5.4. Fortaleza de las señales	50
1.6. Test de los circuitos	51
1.6.1. Test en manufactura	52

1.6.2.	Test funcional	54
1.6.3.	Programas de test funcional	55
1.6.4.	Banco de pruebas	56
1.7.	Representaciones y niveles de abstracción	57
1.7.1.	Representación del sistema	57
1.7.2.	Niveles de abstracción	58
1.7.3.	VHDL en el flujo de desarrollo	63
1.8.	Conceptos básicos a través de un ejemplo	63
1.8.1.	Comportamiento al nivel de funciones lógicas	64
1.8.2.	Descripción de la estructura	70
1.8.3.	Descripción abstracta del comportamiento	72
1.8.4.	Banco de pruebas	76
1.8.5.	Configuración	78
1.9.	Dos simuladores de VHDL'93: VeriBest y ModelSim	79
1.9.1.	Diseño de un buffer triestado	79
1.9.2.	Diseño del banco de pruebas	80
1.10.	Lecturas recomendadas	82
1.11.	Ejercicios de autocomprobación	83
1.12.	Soluciones de los ejercicios	87
2.	Conceptos básicos de VHDL	91
2.1.	Introducción	95
2.2.	Unidades de diseño	95
2.3.	Entity	97
2.3.1.	Cláusula port	98
2.3.2.	Cláusula generic	100
2.3.3.	Declaraciones	101
2.3.4.	Sentencias	101

2.3.5. Resumen de la sintaxis de la entity	101
2.4. Architecture	102
2.5. Asignaciones concurrentes	104
2.5.1. Asignaciones concurrentes simples	104
2.5.2. Asignaciones concurrentes condicionales	107
2.5.3. Asignaciones concurrentes de selección	110
2.5.4. Sensibilidad de las sentencias concurrentes	114
2.6. Sentencia generate	115
2.6.1. Sentencia generate iterativa	115
2.6.2. Sentencia generate condicional	116
2.7. Bloque process	116
2.7.1. Sentencias wait	118
2.7.2. Lista de sensibilidad	120
2.8. Código secuencial	122
2.8.1. Asignación secuencial a una señal	122
2.8.2. Asignación secuencial a una variable	124
2.8.3. Sentencia if	124
2.8.4. Sentencia case	128
2.8.5. Bucle for	132
2.9. Descripción de la estructura	135
2.9.1. Diseños con estructura regular	139
2.10. Parametrización	142
2.10.1. Parametrización del comportamiento	142
2.10.2. Parametrización de la estructura	142
2.11. Señales, variables y constantes	143
2.12. Tipos de datos y operadores	145
2.12.1. Tipos predefinidos en VHDL	146

2.12.2. Tipos del paquete IEEE.std_logic_1164	149
2.12.3. Operadores sobre bit_vector y std_logic_vector	154
2.12.4. Tipos del paquete IEEE.numeric_std	156
2.12.5. Tipos time y string	163
2.12.6. Tipos definidos por el usuario	163
2.13. Atributos	166
2.14. Librerías	167
2.15. Assert	169
2.16. Subprogramas	170
2.16.1. Funciones	170
2.16.2. Procedimientos	172
2.16.3. Diferencias entre funciones y procedimientos	175
2.17. Paquetes	176
2.18. Lecturas recomendadas	179
2.19. Ejercicios de autocomprobación	180
2.20. Soluciones de los ejercicios	187
3. Simulación del código VHDL	201
3.1. Introducción	205
3.2. Procesamiento del código VHDL	206
3.3. Orden de compilación	207
3.4. Drivers	208
3.5. Inicialización	211
3.5.1. Ejemplo: señal con un driver	212
3.5.2. Ejemplo: señal con dos drivers	213
3.6. Atributos de las señales	216
3.7. El retardo delta	219
3.8. Gestión de la cola de transacciones del driver	221

3.8.1.	Ejemplo: simulación de formas de onda con retardo inercial . . .	222
3.8.2.	Ejemplo: simulación de formas de onda con retardo de transporte	225
3.9.	Ejemplo: simulación de un circuito sencillo	226
3.10.	Lecturas recomendadas	230
3.11.	Ejercicios de autocomprobación	231
3.12.	Soluciones de los ejercicios	240
4.	Diseño de lógica combinacional	273
4.1.	Introducción	277
4.2.	Diseño para síntesis de lógica combinacional	277
4.2.1.	Empleo de sentencias concurrentes	278
4.2.2.	Empleo de bloques process	280
4.3.	Funciones lógicas	280
4.3.1.	Diseño del circuito	280
4.3.2.	Programación del banco de pruebas	281
4.4.	Multiplexor de 4 entradas	285
4.4.1.	Diseño usando sentencias secuenciales	285
4.4.2.	Diseño usando sentencias concurrentes	286
4.5.	Restador completo de 1 bit	291
4.5.1.	Descripción del comportamiento	291
4.5.2.	Descripción de la estructura	293
4.5.3.	Programación del banco de pruebas	296
4.6.	Sumador completo de 1 bit	301
4.6.1.	Diseño del circuito	302
4.6.2.	Banco de pruebas	302
4.7.	Unidad aritmético lógica	306
4.7.1.	Diseño de la ALU	306
4.7.2.	Programación del banco de pruebas	308

4.8. Lecturas recomendadas	313
4.9. Ejercicios de autocomprobación	314
4.10. Soluciones de los ejercicios	321
5. Registros y memorias	353
5.1. Introducción	357
5.2. Registro de 4 bits	357
5.2.1. Descripción del comportamiento	358
5.2.2. Banco de pruebas	358
5.3. Registro multifunción	360
5.3.1. Descripción del comportamiento	361
5.3.2. Banco de pruebas	364
5.4. Registro de desplazamiento	368
5.4.1. Descripción del comportamiento	368
5.4.2. Banco de pruebas	369
5.4.3. Banco de pruebas con acceso a fichero	371
5.5. Register file	375
5.5.1. Registro triestado	376
5.5.2. Descripción estructural del register file	378
5.5.3. Drivers y función de resolución	378
5.5.4. Banco de pruebas del register file	380
5.5.5. Descripción del comportamiento del register file	384
5.6. Bus bidireccional y memorias	386
5.6.1. Memoria de sólo lectura	386
5.6.2. Memoria de lectura y escritura	388
5.6.3. Bus bidireccional	389
5.7. Lecturas recomendadas	391
5.8. Ejercicios de autocomprobación	392

5.9. Soluciones de los ejercicios	397
6. Diseño de lógica secuencial	415
6.1. Introducción	419
6.2. Diseño de máquinas de estado finito	419
6.2.1. Circuito detector de secuencias	420
6.3. Síntesis de lógica secuencial	422
6.3.1. Sentencias condicionales incompletas	423
6.3.2. Sentencias condicionales completas	423
6.3.3. Retardos	423
6.3.4. Inicialización	424
6.3.5. Bloques process	424
6.4. Flip-flop JK	425
6.4.1. Diseño del flip-flop	426
6.4.2. Banco de pruebas	426
6.5. Máquinas de estado finito de Moore	430
6.5.1. Diseño de la máquina	430
6.5.2. Banco de pruebas	433
6.5.3. Modelado estructural	437
6.6. Máquinas de estado finito de Mealy	439
6.6.1. Diseño de la máquina	439
6.6.2. Banco de pruebas	445
6.7. Máquinas de estado finito seguras	448
6.8. Lecturas recomendadas	451
6.9. Ejercicios de autocomprobación	452
6.10. Soluciones de los ejercicios	463
7. Metodología de transferencia entre registros	513

7.1. Introducción	517
7.2. Operaciones de transferencia entre registros	518
7.2.1. Operación RT básica	518
7.2.2. Programa RT	520
7.3. Máquinas de estado finito con camino de datos	522
7.3.1. Múltiples operaciones RT y camino de datos	522
7.3.2. Lógica de control mediante FSM	523
7.3.3. Diagrama de bloques básico de la FSMD	523
7.4. Descripción del programa RT usando VHDL	525
7.5. Circuito detector de secuencia	528
7.6. Control de una máquina expendedora	530
7.6.1. Protocolo de handshaking	531
7.6.2. Descripción del algoritmo	532
7.6.3. Diseño del circuito de control	532
7.6.4. Programación del banco de pruebas	537
7.7. Lecturas recomendadas	540
7.8. Ejercicios de autocomprobación	541
7.9. Soluciones de los ejercicios	545
APÉNDICES	561
A. VeriBest VB99.0	561
A.1. Instalación	563
A.2. Edición y compilación de un modelo	563
A.2.1. Arranque del simulador <i>VeriBest VHDL</i>	563
A.2.2. Creación de un espacio de trabajo	563
A.2.3. Edición de un fichero	564
A.2.4. Añadir un fichero al espacio de trabajo	565

A.2.5. Compilación de un fichero	566
A.2.6. Banco de pruebas	569
A.3. Simulación y visualización de los resultados	570
A.3.1. Establecer las condiciones de la simulación	570
A.3.2. Activación del simulador	571
A.3.3. Simulación y visualización de los resultados	571
A.4. Depurado usando el debugger	574
B. ModelSim PE Student Edition	577
B.1. Instalación	579
B.2. Edición y compilación de un modelo	579
B.2.1. Arranque del simulador	580
B.2.2. Creación de un proyecto	581
B.2.3. Añadir ficheros al proyecto	582
B.2.4. Compilación de los ficheros	586
B.2.5. Banco de pruebas	588
B.3. Simulación, visualización y depurado	591
B.3.1. Activación del modo simulación	591
B.3.2. Visualización de los resultados	593
B.3.3. Ejecución de la simulación	594
B.3.4. Inserción de puntos de ruptura	595

Finalmente, el **quinto paso** consiste en sintetizar el circuito secuencial síncrono que implemente las funciones lógicas anteriores. Si se usan flip-flops D, las entradas al flip-flop corresponden con el nuevo estado ($D_A = NA$, $D_B = NB$), mientras que las salidas del flip-flop corresponden con el estado actual ($Q_A = A$, $Q_B = B$). En la Figura 6.3 se muestra el circuito.

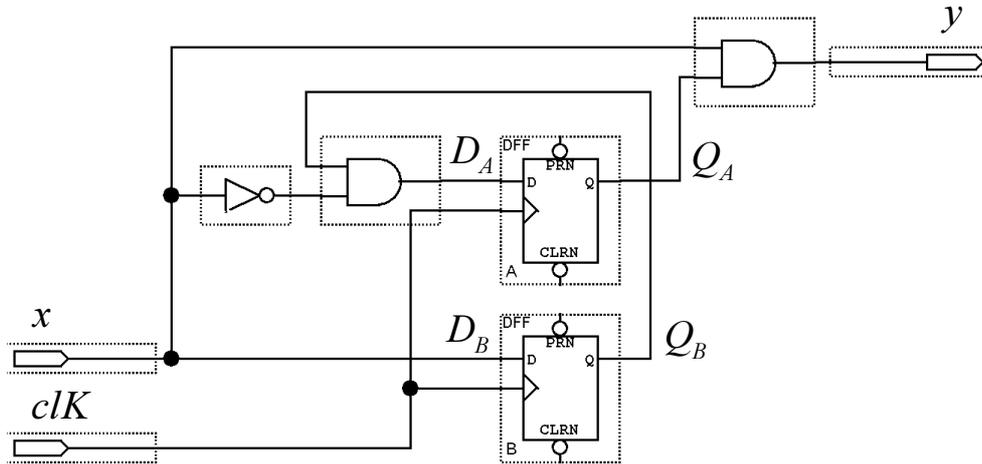


Figura 6.3: Circuito detector de la secuencia 101.

6.3. SÍNTESIS DE LÓGICA SECUENCIAL

El comportamiento y la salida de un bloque de lógica secuencial depende del valor actual de sus entradas y de sus variables de estado. El valor de las variables de estado se almacena típicamente en elementos de memoria, ya sea en *latches* o en *flip-flops*.

Se llama *latches* a aquellos circuitos cuyo estado cambia únicamente cuando su entrada *enable* está a un determinado nivel ('0' ó '1'). A continuación, se muestra parte de la **architecture** de un latch-D.

```

process (enable, D) is
begin
    if (enable = '1') then
        q <= D;
    end if;
end process;

```

Asimismo, se llama *flip-flops* a aquellos circuitos cuyo estado cambia o bien en el flanco de subida del reloj, o bien en el flanco de bajada del reloj. En la Sección 2.7 se describió el diseño de un flip-flop D.

6.3.1. Sentencias condicionales incompletas

Obsérvese que el latch se define empleando una cláusula **if-then** sin **else**. Se denomina *sentencias condicionales incompletas* a este tipo de sentencias. Este punto es importante, ya que muestra que las sentencias condicionales incompletas (**if**, **case**, etc. sin **else**) son sintetizadas mediante latches. Esto es debido a que la salida debe mantener su valor si ninguna de las condiciones de la cláusula se satisface.

Un motivo por el cual podría pensarse en omitir el caso **else**, es que sea indiferente el valor que se asigne a las señales en este caso (ya que en la práctica nunca se producirá). Si no deseamos que el circuito se sintetice mediante un latch, debemos incluir el caso **else** en la sentencia condicional y asignar a las señales el valor 'X'.

6.3.2. Sentencias condicionales completas

Una característica inherente a las sentencias **if-elsif-else** es que las condiciones no tienen forzosamente que ser excluyentes entre sí. La herramienta de síntesis asume, por tanto, que deben comprobarse las condiciones en un cierto orden de prioridad, generando la lógica para ello. Esto da lugar a circuitería innecesaria en el caso en que las condiciones sean excluyentes.

En aquellos casos en que las condiciones de la sentencia condicional completa sean excluyentes entre sí, es preferible emplear sentencias **case** o **with**. Esto es debido a que las sentencias **case** y **with** se sintetizan de manera natural en multiplexores o estructuras equivalente a multiplexores, que son rápidas y ocupan relativamente poco área.

6.3.3. Retardos

En la Sección 4.2 se explicó que en la descripción para síntesis de los circuitos combinacionales debe evitarse el uso de retardos. Lo mismo aplica a la descripción de circuitos secuenciales. Los retardos en el hardware son dependientes de la tecnología empleada en su fabricación y están sujetos a la variabilidad del proceso de fabricación, por ello es extremadamente difícil construir circuitos que presenten un determinado retardo.

Así pues, no es posible sintetizar sentencias tales como **wait for x ns**. Igualmente, no es posible sintetizar sentencias que empleen la cláusula **after**. Algunas

herramientas de síntesis ignoran estas sentencias, mientras que otras muestran mensajes de error.

Cuando es preciso emplear retardos para describir adecuadamente el circuito, puede definirse un retardo constante en la parte superior del código

```
constant DEL : time := 1 ns;
```

y usar este retardo en el código del circuito. Por ejemplo:

```
A <= B after DEL;
```

Puede asignarse un valor positivo a esta constante de retardo para realizar la simulación del circuito y posteriormente asignarle el valor cero cuando vaya a realizarse la síntesis.

Obsérvese que esta discusión acerca de los retardos aplica a los circuitos, no a los bancos de prueba, que no necesitan ser sintetizados.

6.3.4. Inicialización

Debe evitarse inicializar las variables y las señales al declararlas, ya que este tipo de inicialización no puede ser sintetizada.

La inicialización de una variable o señal implica una acción que se realiza únicamente una vez al comienzo de la simulación. Si es preciso realizar una acción al comienzo de la simulación, entonces debe situarse en la secuencia de acciones que se ejecutan cuando se activa la señal de reset. Estas acciones se definen típicamente dentro de un bloque **process** sensible a la señal de reset.

6.3.5. Bloques process

La descripción de la **architecture** de los circuitos secuenciales se basa fundamentalmente en el empleo de bloques **process**. Las asignaciones a señales dentro de los bloques **process** deben cumplir que:

- La señal en la parte izquierda de la asignación secuencial debe ser una señal definida dentro del bloque **process**, o una señal **out** o **inout** de la interfaz del circuito.

- Las señales en la parte derecha de la asignación secuencial deben ser señales definidas dentro del bloque **process**, o señales **in** o **inout** de la interfaz del circuito.
- No se debe asignar valor a una determinada señal dentro de más de un bloque **process**. Es decir, cada señal debe ser evaluada en un único bloque **process**. Aunque VHDL no impide que se asigne valor a una misma señal en varios bloques **process**, esta práctica da lugar a circuitos difíciles de depurar y típicamente no es soportada por las herramientas de síntesis comerciales.

Para asignar valor a las salidas de los flip-flops y latches dentro de un bloque **process**, pueden emplearse sentencias de asignación a señal (usan el operador \leq) y sentencias de asignación a variable (usan el operador $:=$). Las asignaciones a variable tienen efecto inmediatamente, mientras que las asignaciones a señal tienen efecto en un instante de tiempo δ (delta) unidades de tiempo posterior al tiempo simulado actual (suponiendo que no se ha especificado el retraso en la asignación empleando la cláusula **after**).

Cuando se emplean bloques **process** para describir un circuito secuencial, debe indicarse la lista de sensibilidad de cada bloque **process** con el fin de controlar cuándo se activa el bloque. Cuando se especifica una lista de sensibilidad, el simulador no ejecuta el bloque **process** hasta que no se produce un cambio en alguna de las señales que componen la lista.

Obsérvese que es posible controlar la ejecución del bloque **process** mediante la lista de sensibilidad o mediante el empleo de cláusulas **wait** dentro del cuerpo del bloque **process**. En cada bloque **process** debe optarse por uno de los dos métodos, ya que no es posible emplear simultáneamente una lista de sensibilidad y sentencias **wait**. Sin embargo, no es recomendable emplear este segundo método (uso de sentencias **wait** dentro del bloque **process**), ya que puede dar lugar a circuitos no sintetizables.

6.4. FLIP-FLOP JK

En esta sección se describe el diseño de un flip-flop JK con reset asíncrono activado al nivel LOW. Este circuito puede encontrarse en dos estados: $Q=0$ y $Q=1$. La tabla de transición de estados, considerando únicamente las entradas J y K , así como el estado actual (Q_t) y el siguiente estado (Q_{t+1}), es la mostrada a continuación. Las transiciones de estado se producen en el flanco de subida de la señal de reloj.

J	K	Nuevo estado
0	0	$Q_{t+1}=Q_t$
0	1	$Q_{t+1}=0$
1	0	$Q_{t+1}=1$
1	1	$Q_{t+1}=\text{not } Q_t$

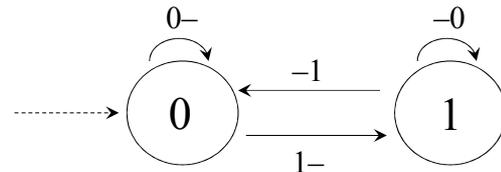


Figura 6.4: Transición de estados de un flip-flop JK. En los arcos del diagrama se muestra el valor de las señales JK. El bit ‘-’ es “don’t care” (por ejemplo, “0-” representa “00” ó “01”). La transición de reset se representa mediante una línea punteada.

Cuando la entrada de reset asíncrono pasa de valer ‘1’ a valer ‘0’, el circuito pasa al estado $Q=’0’$. Se denomina *reset asíncrono* porque la transición al estado ‘0’ se produce en el instante en que cambia la señal de reset, con independencia del valor de la señal de reloj. La transición de reset asíncrono se representa mediante una línea punteada en el diagrama situado en la parte derecha de la Figura 6.4.

El circuito tiene dos salidas: Q y \overline{Q} . La salida Q es igual al estado y la salida \overline{Q} es su inversa: $\overline{Q} = \text{not } Q$.

6.4.1. Diseño del flip-flop

El Código VHDL 6.1 es el diseño de un flip-flop JK con reset asíncrono activado al nivel LOW. La **architecture** contiene un bloque **process** que es sensible a la señal de reset y a la señal de reloj. En el cuerpo del bloque **process** se examina primero la señal de reset, ya que en caso de estar activa esa señal (con valor ‘0’) la máquina debe pasar al estado $Q=’0’$.

6.4.2. Banco de pruebas

En el caso de los circuitos combinacionales, el objetivo del programa de test es comprobar (cuando esto es posible) todas las posibles combinaciones de valores de las entradas al circuito. En los circuitos secuenciales, el programa de test debe recorrer (cuando esto es posible) todos los arcos del diagrama de estado, para cada uno de los valores de las entradas que producen la transición.

En el caso del biestable JK, esto implica testear 8 transiciones. En la Figura 6.5 se indica el orden en que el programa de test recorrerá los arcos (número inscrito

```

-----
-- Biestable JK con reset asíncrono en nivel LOW
-- Fichero: flipflop_JK.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity flipflop_JK is
    port ( q, q_n          : out std_logic;
           clk, J, K, reset_n : in std_logic );
end entity flipflop_JK;

architecture flipflop_JK of flipflop_JK is
    signal q_interna : std_logic;
begin
    q    <= q_interna;
    q_n  <= not q_interna;

    process (reset_n, clk) is
        variable JK : std_logic_vector(1 downto 0);
    begin
        if (reset_n = '0') then
            q_interna <= '0';
        elsif rising_edge(clk) then
            JK := J & K;
            case (JK) is
                when "01" => q_interna <= '0';
                when "10" => q_interna <= '1';
                when "11" => q_interna <= not q_interna;
                when others => null;
            end case;
        end if;
    end process;
end architecture flipflop_JK;
-----

```

Código VHDL 6.1: Diseño de un flip-flop JK con reset asíncrono.

en una circunferencia), y qué valores de las entradas se aplicará en cada caso para producir la transición. El Código VHDL 6.2 y 6.3 es el banco de pruebas.

En el banco de pruebas se ha definido un **procedure**, que comprueba que los valores actuales de la salida del flip-flop coinciden con los esperados, mostrando el correspondiente mensaje en caso de que no coincidan.

Obsérvese que la sentencia **wait** que hay al final del bloque **process** finaliza la ejecución de dicho bloque, pero no la simulación, ya que la simulación de las sentencias de asignación concurrente a las señales de reset y reloj se realiza indefinidamente. Por tanto, la simulación del banco de pruebas no finaliza por sí misma, siendo preciso fijar su duración: 900 ns. La simulación del banco de pruebas se realiza con cero errores. En la Figura 6.6 se muestra el resultado obtenido de la simulación.